22nd International Conference on Rewriting Techniques and Applications

RTA'11, May 30–June 1, 2011, Novi Sad, Serbia

Edited by

Manfred Schmidt-Schauß

Editor

Manfred Schmidt-Schauß
Institut für Informatik
FB Informatik und Mathematik (12)
Johann Wolfgang Goethe-Universität
Postfach 11 19 32
60054 Frankfurt am Main, Germany
schauss@ki.informatik.uni-frankfurt.de

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 978-3-939897-30-9

Published online and open access by
Schloss Dagstuhl – Leibniz-Center for Informatics GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at http://www.dagstuhl.de/dagpub/978-3-939897-30-9.

Publication date
May, 2011

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

Digital Object Identifier: 10.4230/10.4230/LIPIcs.RTA.2011.i

ISBN 978-3-939897-30-9 **ISSN 1868-8969** **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.

ISSN 1868-8969

www.dagstuhl.de/lipics

Contents

22nd International Conference on Rewriting Techniques and Applications (RTA'11).
Editor: Manfred Schmidt-Schauß
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Regular Papers

RTA 2011 Conference Organization

RTA 2011 Conference Chair

Silvia Ghilezan	Novi Sad, Serbia

Program Chair

Manfred Schmidt-Schauß	Frankfurt am Main, Germany

Program Committee

Franz Baader	TU Dresden, Germany
Frédéric Blanqui	Inria, China
Véronique Cortier	CNRS, Loria, France
Dan Dougherty	Worcester Polytechnic Institute, United States
Maribel Fernández	King's College London, United Kingdom
Jürgen Giesl	RWTH Aachen, Germany
Florent Jacquemard	INRIA Saclay – LSV (CNRS/ENS Cachan), France
Fairouz Kamareddine	Heriot-Watt University, United Kingdom
Salvador Lucas	Universidad Politécnica de Valencia, Spain
Narciso Marti-Oliet	Universidad Complutense de Madrid, Spain
Aart Middeldorp	University of Innsbruck, Austria
Georg Moser	University of Innsbruck, Austria
Paliath Narendran	University at Albany – SUNY, United States
Joachim Niehren	INRIA Lille, France
Hitoshi Ohsaki	AIST Osaka, Japan
Vincent van Oostrom	Universiteit Utrecht, The Netherlands
Femke van Raamsdonk	Vrije Universiteit Amsterdam, The Netherlands
Grigore Rosu	University of Illinois at Urbana-Champaign, USA
Aleksy Schubert	The University of Warsaw, Poland
Jakob Grue Simonsen	University of Copenhagen, Denmark
René Thiemann	University of Innsbruck, Austria
Christian Urban	TU München, Germany
Johannes Waldmann	HTWK Leipzig, Germany
Hans Zantema	Technische Universiteit Eindhoven, The Netherlands
Florent Jacquemard	INRIA Saclay, France

RTA Steering Committee

Johannes Waldmann (Chair)	Leipzig, Germany
Ian Mackie	Palaiseau, France
Joachim Niehren	Lille, France
Frederic Blanqui	INRIA, China
Salvador Lucas	Valencia, Spain
Masahiko Sakai (Publicity Chair)	Nagoya, Japan

RTA Web Page

rewriting.loria.fr/rta/

22nd International Conference on Rewriting Techniques and Applications (RTA'11).
Editor: Manfred Schmidt-Schauß

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

External Reviewers

Abel, Andreas
Anantharaman, Siva
Anis, Altug
Aoto, Takahito
Avanzini, Martin
Bahr, Patrick
Bongaerts, Jochem
Burghardt, Jochen
Bursuc, Sergiu
Chlipala, Adam
Chrzaszcz, Jacek
De Moura, Flavio L C
De Vrijer, Roel
Drewes, Frank
Duran, Francisco
Emmes, Fabian
Endrullis, Joerg
Erbatur, Serdar
Felgenhauer, Bertram
Fuhs, Carsten
Gadducci, Fabio
Genaim, Samir
Goel, Amit
Gramlich, Bernhard
Guiraud, Yves
Gutiérrez, Raúl
Heam, Pierre-Cyrille
Heindel, Tobias
Hirokawa, Nao
Kahrs, Stefan
Kesner, Delia
Ketema, Jeroen
König, Barbara
Kuan, George
Kusakari, Keiichirou
Küsters, Ralf
Lafourcade, Pascal
Lippi, Sylvain
Lohrey, Markus
Lucanu, Dorel
Machkasova, Elena
Maniotis, Andreas
Manzonetto, Giulio
Marion, Jean-Yves
Marshall, Andrew
Mcbride, Conor
Mesnard, Fred
Mogensen, Torben
Mulligan, Dominic
Noschinski, Lars
Ould Biha, Sidi
Oyamaguchi, Michio
Pena, Ricardo
Pinaud, Bruno
Polonsky, Andrew
Popescu, Andrei
Raffelsieper, Matthias
Rety, Pierre
Rothe, Jörg
Sabel, David
Sakai, Masahiko
Schnabl, Andreas
Serbanuta, Traian
Sternagel, Christian
Stump, Aaron
Sznuk, Tadeusz
Urbain, Xavier
Verdejo, Alberto
Villaret, Mateu
Voigtländer, Janis
Winkler, Sarah
Zankl, Harald

22nd International Conference on Rewriting Techniques and Applications (RTA'11).
Editor: Manfred Schmidt-Schauß
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Author Index

22nd International Conference on Rewriting Techniques and Applications (RTA'11).
Editor: Manfred Schmidt-Schauß
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Preface

This volume contains the papers presented at the 22nd International Conference on Rewriting Techniques and Applications (RTA 2011) which was held from May 30 to June 1, 2011, in Novi Sad, Serbia as part of the RDP 2011 Federated Conference on Rewriting, Deduction, and Programming, together with the 10th Typed Lambda Calculi and Applications (TLCA 2011). Workshops associated with RTA 2011 were the Workshop on Compilers by Rewriting, Automated (COBRA 2011), the Annual Meeting of the IFIP Working Group 1.6 on Term Rewriting, the workshop "Two Faces of Complexity" (2FC 2011), the 10th International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2011), and the workshop on Theory and Practice of Delimited Continuations (TPDC 2011).

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), Brasilia (2009), and Edinburgh (2010). For RTA 2011, 20 regular research papers and eight system descriptions were accepted out of 46 submissions. Each paper was reviewed by at least three members of the Program Committee, with the help of 72 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 for gathering the papers for this proceedings.

The Program Committee gave the award for the Best Contribution to RTA 2011 to Aaron Stump, Garrin Kimmell and Roba El-Haj Omar for their paper "Type Preservation as a Confluence Problem". In addition to the contributed papers, the RTA program contained an invited talk by Sophie Tison with title "Tree Automata, (Dis-)Equality Constraints and Term Rewriting: What's New?", by Ashish Tiwari on "Rewriting in Practice", and also the common invited talk of RDP, given by Stephanie Weirich on "Combining Proofs and Programs".

Many people helped to make RTA 2011 as part of RDP 2011 a success. Thanks to all of them: the Members of the Organizing Committee of RDP are: Siniša Crvenković (University of Novi Sad), Ilija Ćosić (University of Novi Sad), Kosta Došen (Mathematical Institute, Belgrade), Silvia Ghilezan (University of Novi Sad), Predrag Janičić (University of Belgrade), Zoran Marković (Mathematical Institute, Belgrade), Zoran Ognjanović (Mathematical Institute, Belgrade), Jovanka Pantović (University of Novi Sad), Zoran Petrić (Mathematical Institute, Belgrade), Miroslav Popović (University of Novi Sad), Nataša Sladoje (University of Novi Sad), Miroslav Vesković (University of Novi Sad); and the Local Organizers: Sandra Buhmiler, Biljana Carić, Jelena Čolić, Ksenija Doroslovački, Tatjana Grbić, Gabrijela Grujić, Vladimir Ilić, Jelena Ivetić, Svetlana Jakšić, Tibor Lukić, Petar Maksimović, Bojan Marinković, Biljana Mihailović, Aleksandar Nikolić, Zoran Ovcin, Dragiša Žunić.

For their kind help in preparing the proceedings and supporting me in questions concerning LaTeX thanks go the David Sabel and Conrad Rau. I would like to thank Marc Herbstritt from Schloss Dagstuhl, Leibniz Center for Informatics, for his very helpful and always prompt support during production of the LIPIcs proceedings.

RDP was hosted by the Faculty of Technical Sciences and the Mathematical Institute SASA at the University of Novi Sad, Serbia. Support by the conference sponsors: – Provincial Secretariat of Science and Technological Development, Province of Vojvodina, the Ministry of

Education and Science, Republic of Serbia, the City of Novi Sad (http://www.novisad.rs/en), RT-RK Computer Based Systems, FIMEK, Faculty of Economy and Engineering Management, Telvent DMS LLC Novi Sad, Embassy of France in Serbia and Embassy of the USA in Serbia – is gratefully acknowledged.

April 2011 Manfred Schmidt-Schauß

Tree Automata, (Dis-)Equality Constraints and Term Rewriting: What's New?

Sophie Tison

University Lille 1, Mostrare project, INRIA Lille Nord-Europe & LIFL

Abstract

Connections between Tree Automata and Term Rewriting are now well known. Whereas tree automata can be viewed as a subclass of ground rewrite systems, tree automata are successfully used as decision tools in rewriting theory. Furthermore, applications, including rewriting theory, have influenced the definition of new classes of tree automata. In this talk, we will first present a short and not exhaustive reminder of some fruitful applications of tree automata in rewriting theory. Then, we will focus on extensions of tree automata, specially tree automata with local or/and global (dis-)equality constraints: we will emphasize new results, compare different extensions, and sketch some applications.

This talk will be strongly inspired by recent works of several researchers, specially but not exclusively: Emmanuel Filiot, Guillem Godoy, Florent Jacquemard, Jean-Marc Talbot, Camille Vacher.

1998 ACM Subject Classification F.4.2.

Keywords and phrases Tree Automata, Constraints, Term Rewriting

Digital Object Identifier 10.4230/LIPIcs.RTA.2011.1

Category Invited Talk

Tree automata can be used in numerous ways as decision tools in rewriting theory. An ideal way is to encode the reducibility relation by a recognizable relation. Of course, recognizability of the reducibility relation is a very strong requirement which limits this approach to very restricted subclasses of term rewriting systems. A weaker requirement is that the rewrite relation preserves recognizability. This approach is a key point in reachability analysis and tree regular model checking. More generally, encoding set of descendants of terms and possibly sets of normal forms by tree automata provide canonical techniques to obtain decidability results and a lot of work has been done to characterize subclasses of term rewriting systems having "good recognizability properties". Even powerful, these methods remain restricted. To enhance their power, numerous works have been developed, e.g. in reachability analysis, by using abstract interpretation or over-approximations. Other works have recently focused on using rewrite strategies. An other approach is the use of extended tree automata. E.g., equational tree automata have been proposed to handle equational theories and several extensions have been defined to take into account associativity. This talk will focus on new results about extensions of tree automata with equality and disequality constraints.

From Local Constraints ...

A typical example of language which is not recognized by a finite tree automaton is the set of ground instances of a non-linear term. This implies of course that the set of ground normal forms of a t.r.s. is not necessarily recognizable. An other point is that images by a

22nd International Conference on Rewriting Techniques and Applications.
Editor: M. Schmidt-Schauß; pp. 1–3

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

non-linear morphism of a recognizable tree language are not necessarily recognizable. E.g. if a morphism associates $h(x)$ with $f(x,x)$ and a with b, the image by this morphism of $h^*(a)$ is the non-recognizable set of well-balanced terms over $\{f,b\}$. So, extending tree automata to handle non-linearity is very natural and in the early 80's, M. Dauchet and J. Mongy have proposed a new class for this purpose, by enriching tree automata rules by equality constraints. E.g. if a rule is associated with the constraint $1.1 = 1.2$, it can be applied at position p in t only if the subterms at positions $p.1.1$ and $p.1.2$ are equal. A more general class can be defined by allowing also disequality constraints. Unfortunately, emptiness is undecidable even when only equality constraints are allowed. Several restrictions of this class have so been studied (see e.g. a survey in [2]). Let us remind two of them, which are of special interest for term rewriting. The first one, the class of automata with constraints between brothers, restricts equality and disequality tests to sibling positions. This class has good decidability and closure properties. E.g., this class allows to represent normal forms for left-shallow t.r.s. -but not for general ones - and it helped recently for providing new decidability results for normalization. The second one, the class of reduction automata, bounds, roughly speaking, the number of equality constraints. This class has provided the decidability of the reducibility theory and as a corollary a (new) way of deciding ground reducibility. Recently, a strong work by Godoy & alt. [4] has given some new emphasis on these classes. Indeed, it defines some new subclasses having good properties. This enables them to decide whether the homomorphic image of a tree recognizable language is recognizable. As a corollary, they get a new simple proof of decidability of recognizability of the set of normal forms of a t.r.s..

. . . to Global Ones

A new approach has been recently proposed : adding constraints to perform (dis-)equality tests, but globally. The idea is to enrich the automaton by two relations, $=, \neq$, over the states. Roughly speaking, the run will be correct if the subterms associated with two "equal" (resp. "not equal") states are equal (resp. different). E.g. this approach enables to check that all the subterms rooted by a f are equal or to encode that every identifier is different. This approach has led to (almost) simultaneous definitions of classes by different researchers to different purposes: rigid tree automata [5], tree automata with global equality and disequality tests (TAGED) [3], tree automata with global constraints [1]. The second part of this talk will give an overview of these classes and sketch their links with other classes.

References – A very incomplete list of references the talk will rely upon

1 Luis Barguñó, Carles Creus, Guillem Godoy, Florent Jacquemard, and Camille Vacher. The emptiness problem for tree automata with global constraints. In *Proceedings of the 25th Annual IEEE Symposium on Logic in Computer Science, LICS*, pages 263–272. IEEE Computer Society, 2010.

2 H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available at: `http://www.grappa.univ-lille3.fr/tata`, 2007. Release October 12th, 2007.

3 Emmanuel Filiot, Jean-Marc Talbot, and Sophie Tison. Tree automata with global constraints. *Int. J. Found. Comput. Sci.*, 21(4):571–596, 2010.

4 Guillem Godoy, Omer Giménez, Lander Ramos, and Carme Àlvarez. The hom problem is decidable. In Leonard J. Schulman (ed.), *Proceedings of the 42nd ACM Symposium on Theory of Computing, STOC 2010*, pages 485–494.

5 Florent Jacquemard, Francis Klay, and Camille Vacher. Rigid tree automata and applications. *Inf. Comput.*, 209(3):486–512, 2011.

Rewriting in Practice*

Ashish Tiwari[1]

1 **SRI International**
Menlo Park, CA 94025
tiwari@csl.sri.com

Abstract

We discuss applications of rewriting in three different areas: design and analysis of algorithms, theorem proving and term rewriting, and modeling and analysis of biological processes.

1998 ACM Subject Classification F.4.2 [Mathematical Logic and Formal Languages] Grammars and Other Rewriting Systems–Decision problems; I.6.5 [Simulation and Modeling] Model Development–Modeling Methodologies; I.1.2 [Symbolic and Algebraic Manipulation] Algorithms–Algebraic Algorithms

Keywords and phrases Rewriting, Polynomial constraints, Biochemical reaction networks

Digital Object Identifier 10.4230/LIPIcs.RTA.2011.3

Category Invited Talk

1 Introduction

The field of rewriting has contributed some fundamental results within the computer science discipline. Here, we explore a few impactful applications of rewriting. Any article that describes applications of a field has to be necessarily incomplete. We limit our focus on the use of rewriting technology in the following three areas.

1. Design of algorithms
2. Formal modeling and analysis
3. Term rewriting and theorem proving

We discuss the influence of the theory of rewriting in the above areas by identifying specific concrete instances of algorithms, tools, or techniques that have, or can, be impacted by rewriting.

The field of rewriting is broadly concerned with manipulating *representations* of *objects* so that we go from a larger representation to a *smaller* representation. Clearly, rewriting is concerned with three important entities: objects, representations, and orderings. We give a few examples below.

- In term rewriting, the objects are equivalence classes of terms, representations of these objects are the terms themselves, and orderings are certain binary relations on terms.
- In polynomial rings, objects are polynomials and representations are algebraic expressions constructed using the arithmetic (ring) operators.
- In theorem proving, objects are proofs and orderings are proof orderings on some representation of the proofs.

A significant part of the theory of rewriting abstracts away from the objects and/or their representations, and just studies properties of binary relations.

* Research supported in part by the National Science Foundation under grant CSR-EHCS-0834810 and CSR-0917398.

22nd International Conference on Rewriting Techniques and Applications (RTA'11).
Editor: M. Schmidt-Schauß; pp. 3–8

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

2 Rewriting in the design of algorithms

The theory of rewriting can be used as a basis to study the design and analysis of algorithms. There are at least two distinct ways in which rewriting helps in algorithmic development.

2.1 Rewrite-based Descriptions

An algorithm can often be viewed as a set of rewrite rules. This *rewrite-based description* cleanly separates the logical part of an algorithm from its implementation details. Rewrite-based views of algorithms can be very helpful especially from an educational perspective.

Consider, for example, the problem of sorting a sequence of numbers, which is one of the most commonly studied problems in a first course in Algorithms. A description of a sorting procedure can be given by a single rewrite rule

$$X, a, Y, b, Z \quad \to \quad X, b, Y, a, Z \quad \text{if } a > b$$

where , is an associative operator. This rewrite rule simply says that we can swap two elements in the current list if they are not in order. This rewrite rule describes a whole class of comparison-based sorting algorithms: different concrete sorting algorithms, such as bubble sort, are obtained by applying (specific instances of) the above rewrite rule using different *strategies*. The correctness – soundness, completeness, and termination – of comparison-based sorting algorithms just depends on properties of the rewrite system containing the above rule. The strategy (and the data structures used to represent the terms) is only an implementation detail (although an important detail since it determines the final time and space complexity). Graph algorithms can be similarly presented abstractly using rewrite rules.

2.2 General Paradigm

Rewriting provides a general paradigm for the design of algorithms. The (abstract) critical-pair completion algorithm is a generic procedure that can be instantiated in different domains to yield very important algorithms, such as,

- the algorithms implementing the union-find data structure [9],
- congruence closure [24, 8] and associate-commutative congruence closure [6],
- Gröbner basis algorithm [11, 3, 7], and
- Simplex algorithm for satisfiability of linear constraints [14, 15].

The completion-based rewriting view of these algorithms yield simpler proofs of correctness of these complex algorithms.

There is, however, an additional benefit of taking the completion-based view. It becomes possible to inherit certain optimizations. Consider the Gröbner basis algorithm. It involves adding equations arising from critical overlap of polynomial equations. If we have the following two equations in the current set of equations

$$x^2 \overset{(1)}{\to} p \qquad\qquad xy \overset{(2)}{\to} q$$

where x^2 is the maximal monomial in the polynomial $x^2 - p$ and xy is the maximal monomial in the polynomial $xy - q$ (assume some ordering, say a total degree lexicographic ordering with precedence $x \succ y$), then the procedure for computing Gröbner basis adds the new equation $py \overset{(3)}{=} qx$ (arising from the cricial pair between the two rules above) to the current set of polynomial equations. Efficiency of Gröbner basis algorithms is determined by the number of such critical equations generated. Hence, deletion is important: if we can delete an equation or rule, we compute fewer inferences subsequently. In the above example, we note

that we cannot delete any of the two original rules. However, we can delete the instances $x^2yX \to pyX$ of the first equation, where X is an arbitrary power product. The reason why we can delete these instances is that they all have a proof that does not use the first rule

$$x^2yX \overset{(2)}{\to} qxX \overset{(3)}{=} pyX$$

Moreover, in a suitably defined proof ordering, this new proof of $x^2yX = pyX$ can be shown to be smaller than the old proof that uses Rule (1). Thus, in the new optimized Gröbner basis procedure, rules are associated with a list of monomials called `forbidden`, with the interpretation that instances of a rule obtained by multiplying that rule with a monomial in the ideal generated by `forbidden` are assumed to have been deleted. This optimization has been mentioned before [1, 33], and it appears to be related to some of the new *signature-based* algorithms for Gröbner basis [16]. An exciting future work would be to study the signature-based algorithms using the rewriting framework.

3 Rewriting in Term Rewriting and Theorem Proving

The essence of the theory of rewriting, and in particular of the critical-pair completion procedure, can be described as

1. add facts that make proofs of provable facts smaller and
2. delete facts that already have smaller proofs

Here, by facts we mean equations, or clauses, or formulas (ground or non-ground), or any representation of the known object of interest. This more general view of completion inspires the design of several other algorithms; most notably the algorithms used in saturation-based theorem proving [4].

In equational reasoning, proofs have a certain nice structure that enables one to define interesting proof orderings, which leads to algorithms such as standard and ordered completion [2]. When considering non-equational theories, proofs do not necessarily have a very nice structure. So, complexity of a proof is often just the complexity of the facts used in the proof. Consider the resolution inference rule, where given the two (propositional) clauses

$$x \ \vee \ C_1 \qquad\qquad \neg x \ \vee \ C_2$$

the resolution inference rule adds the new clause $C_1 \vee C_2$ to the set of clauses. Inspired by the theory of rewriting, if we restrict addition of facts to only those facts that make proofs of provable facts smaller, then $C_1 \vee C_2$ must be smaller than the two facts that were used to derive it. Restricting resolution inference in this way leads to the *ordered resolution* calculus.

Note that a set of facts is unsatisfiable if the empty clause, $\bot$, is provable. The ordering on facts is defined such that $\bot$ is the minimal element in the ordering. Since proof calculi inspired by rewriting are designed to generate all "small" facts, then if $\bot$ is provable, then it is explicitly generated. If it is not explicitly generated, then the set of generated (small) facts can be used to construct a model for the facts. This argument can be used to prove refutational completeness of ordered resolution. Several first-order proof calculi are designed, and proved refutationally complete, based on this same principle [5].

The theory of rewriting can be seen as a paradigm for saturating a set of facts with new facts, guided by an ordering on the universe of facts, such that certain minimal facts (such as $\bot$) are generated. This view is very helpful when developing heuristics for searching the space of (provable) facts for a particular one. As remarked above, the field of theorem proving is concerned with deriving $\bot$ to obtain a refutation. Now, as a second example, consider the problem of deciding if a conjunction of polynomial equality and inequality constraints is

satisfiable in the theory of reals. First, consider the case when there are only linear equations and nonnegativity constraints on certain slack variables,

$$A \begin{bmatrix} \vec{x} \\ \vec{u} \end{bmatrix} = \vec{b}, \qquad \vec{u} > 0$$

where A is a $l \times (m+n)$ matrix, $\vec{x}$ is a $m \times 1$ vector, $\vec{u}$ is a $n \times 1$ vector, and $\vec{b}$ is a $l \times 1$ vector. We can prove that the above constraint is unsatisfiable over the reals if (and only if) we can find a linear expression $\vec{c}^T \vec{u}$ such that
(i) $\vec{c}^T \vec{u}$ can be written as a linear combination of the l linear expressions $A[\vec{x}; \vec{u}]$, and
(ii) $\vec{c} \geq \vec{0}$ and $\vec{c} \neq \vec{0}$.
Such a linear expression can be thought of as the *witness* for unsatisfiability of the original set of constraints. If we find an ordering in which the witness, $\vec{c}^T \vec{u}$, is a minimal element in the set of all expressions that can be written as a linear combination of the l linear expressions $A[\vec{x}; \vec{u}]$, then a rewriting-based saturation procedure will explicitly generate this linear expression. This is the idea behind the Simplex algorithm for linear constraints [15].

When the constraints are not necessarily linear, then there again exists a *witness* for unsatisfiability. This is stated by the Positivstellensatz [26, 31, 10]. Specifically, let P, Q, and R be sets of polynomials over $\mathbb{Q}[\vec{x}]$. The constraint

$$\{p = 0 : p \in P\} \cup \{q \geq 0 : q \in Q\} \cup \{r \neq 0 : r \in R\}$$

is unsatisfiable iff there exist polynomials p, q, and r such that $p + q + r^2 = 0$ where

$$\begin{aligned} p &\in \mathit{Ideal}(P) \\ r &\in [R] := \{\Pi_{i \in I}\ r_i : r_i \in R \text{ for all } i \in I\} \\ q &\in \mathit{Cone}[Q] := \{\Sigma_{i \in I}\ p_i^2 q_i : q_i \in [Q], p_i \in \mathbb{Q}[\vec{x}] \text{ for all } i \in I\} \end{aligned}$$

If we find an ordering in which the witness, p, which is equal to $-q - r^2$, is a minimal element in the set of all elements in the ideal generated by P, then a rewriting-based saturation procedure will explicitly generate this witness. This is the idea behind the procedure for unsatisfiability checking based on Gröbner basis computation [34].

A crucial aspect in the above applications is the flexibility provided by the choice of ordering. The choice of ordering decides which facts are generated, and hence it determines if we will ever find a particular desired fact. This observation can be used to generate (equational) invariants of a continuous dynamical system. Consider the differential equations for circular motion: $\frac{dx}{dt} = y$, $\frac{dy}{dt} = -x$. A constant-of-motion, or an equational invariant, for this system would be a polynomial p over variables x, y such that $dp/dt = 0$. Let us order the above two equations backwards and write them as

$$y \to d(x) \qquad\qquad x \to -d(y)$$

We also have (infinite) rewrite rules that come from the definition of the derivative operator d. Consider the following two rules

$$yd(y) \to d(y^2)/2 \qquad\qquad xd(x) \to d(x^2)/2$$

Doing a critical-pair completion and computing a Gröbner basis for this system will yield an equation $d(x^2) + d(y^2) = 0$, which is the equational invariant for circular motion.

The rewriting philosophy is also used extensively in proving results about rewrite systems. Several decidable criterion have been recently developed for checking confluence of restricted classes of term rewriting systems [19, 21, 22, 20]. The characterizations of confluence are proved correct by showing that (a) if there is a counter-example to the characterization, then there will be a smaller counter-example and (b) all minimal counter-examples are explicitly checked.

4 Rewriting in Formal Modeling and Analysis

Rewriting provides both a language for formal modeling of systems, as well as a tool for simulating and analyzing the formal models.

The system Maude [12] is an example of a modeling and analysis tool based on rewriting logic. Maude is being extensively used to formally represent knowledge about biological processes. This knowledge is captured in the form of a Petrinet (represented in Maude). A Petrinet is a set of ground rewrite rules over a signature containing an associative and commutative (AC) symbol. A biochemical reaction is naturally a Petrinet transition [13, 23, 32]. Elaborate models of cell signaling pathways have been formalized in the Pathway Logic tool [32] that is built over Maude.

The rewrite-rule based models of biological processes pose several interesting analysis questions. Apart from questions about reachability, one is also interested in characterizing certain kinds of *steady state* behaviors. A steady state behavior is a rewrite derivation with certain properties. Flux balance analysis (FBA) is a commonly used technique for finding such steady state behaviors of Petrinets [29, 28]; see also [35] for an adaptation of FBA.

Biology also motivates a study of probabilistic rewrite systems. Again, a special case of (timed) stochastic Petrinets has been extensively studied [17, 18] and the same ideas can be possibly applied to stochastic extensions of general rewriting systems too.

The biology domain is an extremely rich source of challenging problems and extensions for the theory of rewriting. One such challenge is *learning* rewrite rules or models from available data on rewrite derivations. One could use it to learn models of disease propagation in humans and develop therapeutics based on the learned model.

5 Conclusion

The theory of rewriting is an important part of the foundation of computer science. It provides an important paradigm for algorithmic design and correctness and a uniform high-level view of several intricate algorithms and techniques. It also provides a useful modeling language, especially for the emerging discipline of Systems Biology [25, 27, 30].

References

1 W. W. Adams and P. Loustaunau. *An Introduction to Gröbner Bases*, volume 3 of *Graduate Studies in Mathematics*. American Mathematical Society, 1994.

2 L. Bachmair. *Canonical Equational Proofs*. Birkhäuser, Boston, 1991.

3 L. Bachmair and H. Ganzinger. Buchberger's algorithm: A constraint-based completion procedure. In *CCL*, volume 845 of *LNCS*. Springer, 1994.

4 L. Bachmair and H. Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *J. of Logic and Computation*, 4:217–247, 1994.

5 L. Bachmair and H. Ganzinger. Resolution theorem proving. In *Handbook of Automated Reasoning*. Elsevier, 2001.

6 L. Bachmair, I.V. Ramakrishnan, A. Tiwari, and L. Vigneron. Congruence closure modulo AC. In *Proc. FroCoS*, volume 1794 of *LNAI*, pages 245–259. Springer, 2000.

7 L. Bachmair and A. Tiwari. D-bases for polynomial ideals over commutative noetherian rings. In *RTA*, volume 1103 of *LNCS*, pages 113–127. Springer, 1997.

8 L. Bachmair and A. Tiwari. Abstract congruence closure and specializations. In *Conf. on Automated Deduction, CADE 2000*, volume 1831 of *LNAI*, pages 64–78. Springer, 2000.

9 L. Bachmair, A. Tiwari, and L. Vigneron. Abstract congruence closure. *J. of Automated Reasoning*, 31(2):129–168, 2003.

10 J. Bochnak, M. Coste, and M.-F. Roy. *Real Algebraic Geometry*. Springer, 1998.
11 B. Buchberger. A critical-pair completion algorithm for finitely generated ideals in rings. In *Proc. Logic and Machines: Decision Problems and Complexity*, volume 171 of *LNCS*, pages 137–161, 1983.
12 M. Clavel et al. *Maude: Specification and Programming in Rewriting Logic*. http://maude.csl.sri.com/manual/, SRI International, Menlo Park, CA, 1999.
13 V. Danos, J. Feret, W. Fontana, R. Harmer, and J. Krivine. Rule-based modelling of cellular signalling. In *Proc. CONCUR*, volume 4703 of *LNCS*, pages 17–41, 2007.
14 D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. *J. of the ACM*, 52(3):365–473, 2005.
15 B. Dutertre and L. de Moura. A fast linear-arithmetic solver for DPLL(T). In *CAV 2006*, volume 4144 of *LNCS*, pages 81–94, 2006.
16 C. Eder and J. Perry. Signature-based algorithms to compute Groebner bases. In *ISSAC*, 2011. arXiv:1101.3589v2.
17 D. T. Gillespie. A general method for numerically simulating the stochastic time evolution of coupled chemical reactions. *J. Comp. Physics*, 22:403–434, 1976.
18 D. T. Gillespie. Approximate accelerated stochastic simulation of chemically reacting systems. *J. of Chemical Physics*, 115(4):1716–1733, 2001.
19 G. Godoy, R. Nieuwenhuis, and A. Tiwari. Classes of Term Rewrite Systems with Polynomial Confluence Problems. *ACM Trans. on Comp. Logic (TOCL)*, 5(2):321–331, 2004.
20 G. Godoy and A. Tiwari. Confluence of shallow right-linear rewrite systems. In *CSL 2005*, volume 3634 of *LNCS*, pages 541–556. Springer, 2005.
21 G. Godoy, A. Tiwari, and R. Verma. On the confluence of linear shallow term rewrite systems. In *STACS 2003*, volume 2607 of *LNCS*, pages 85–96. Springer, 2003.
22 G. Godoy, A. Tiwari, and R. Verma. Characterizing confluence by rewrite closure and right ground term rewrite systems. *AAECC*, 15(1):13–36, June 2004.
23 W. S. Hlavacek et al. Rules for modeling signal-transduction systems. *Sci STKE*, 344, 2006. PMID: 16849649.
24 D. Kapur. Shostak's congruence closure as completion. In *Rewriting Techniques and Applications, RTA 1997*, volume 1103 of *LNCS*, pages 23–37, 1997.
25 H. Kitano. Systems biology: A brief overview. *Science*, 295:1662–1664, 2002.
26 J. L. Krivine. Anneaux preordonnes. *J. Anal. Math.*, 12:307–326, 1964.
27 P. Lincoln and A. Tiwari. Symbolic systems biology: Hybrid modeling and analysis of biological networks. In *HSCC*, volume 2993 of *LNCS*, pages 660–672, 2004.
28 J.D. Orth, I. Thiele, and B.O. Palsson. What is flux balance analysis? *Nature Biotechnology*, 28:245–248, 2010.
29 B.O. Palsson. *Systems Biology: Properties of Reconstructed Networks*. Cambridge University Press, 2006.
30 C. Priami. Algorithmic systems biology. *CACM*, 52(5):80–88, 2009.
31 G. Stengle. A Nullstellensatz and a Positivstellensatz in semialgebraic geometry. *Math. Ann.*, 207, 1974.
32 C. L. Talcott. Pathway logic. In *Formal Meth. for Comp. Sys. Bio., SFM*, volume 5016 of *LNCS*, pages 21–53, 2008. `http://pl.csl.sri.com`.
33 A. Tiwari. *Decision procedures in automated deduction*. PhD thesis, State University of New York at Stony Brook, 2000.
34 A. Tiwari. An algebraic approach for the unsatisfiability of nonlinear constraints. In *CSL 2005*, volume 3634 of *LNCS*, pages 248–262. Springer, 2005.
35 A. Tiwari, C. Talcott, M. Knapp, P. Lincoln, and K. Laderoute. Analyzing pathways using SAT-based approaches. In *AB*, volume 4545 of *LNCS*. Springer, 2007.

Combining Proofs and Programs*

Stephanie Weirich[1]

1 **Department of Computer and Information Science, University of Pennsylvania Philadelphia, USA**
sweirich@cis.upenn.edu

Abstract

Programming languages based on dependent type theory promise two great advances: *flexibility* and *security*. With the type-level computation afforded by dependent types, algorithms can be more generic, as the type system can express flexible interfaces via programming. Likewise, type-level computation can also express data structure invariants, so that programs can be proved correct through type checking. Furthermore, despite these extensions, programmers already know everything. Via the Curry-Howard isomorphism, the language of type-level computation and the verification logic is the programming language itself.

There are two current approaches to the design of dependently-typed languages: Coq, Epigram, Agda, which grew out of the logics of proof assistants, require that all expressions terminate. These languages provide decidable type checking and strong correctness guarantees. In contrast, functional programming languages, like Haskell and Ωmega, have adapted the features dependent type theories, but retain a strict division between types and programs. These languages trade termination obligations for more limited correctness assurances.

In this talk, I present a work-in-progress overview of the Trellys project. Trellys is new core language, designed to provide a smooth path from functional programming to dependently-typed programming. Unlike traditional dependent type theories and functional languages, Trellys allows programmers to work with total and partial functions uniformly. The language itself is composed of two fragments that share a common syntax and overlapping semantics: a simple logical language that guarantees total correctness and an expressive call-by-value programming language that guarantees types safety but not termination.

Importantly, these two fragments interact. The logical fragment may soundly reason about effectful, partial functions. Program values may be used as evidence by the logic. We call this principle *freedom of speech*: whereas proofs themselves must terminate, they must be allowed to reason about any function a programmer might write. To retain consistency, the Trellys type system keeps track of where potentially non-terminating computations may appear, so that it can prevent them from being used as proofs.

1998 ACM Subject Classification F.3.3 Studies of Program Constructs (Type structure)

Keywords and phrases Dependent types, functional programming

Digital Object Identifier 10.4230/LIPIcs.RTA.2011.9

Category Invited Talk

Acknowledgements Trellys is a collaborative project between the University of Pennsylvania, the University of Iowa and Portland State University. This talk is based on joint work with Aaron Stump, Tim Sheard, Chris Casinghino, Vilhelm Sjöberg, Brent Yorgey, Harley D. Eades III, Garrin Kimmel, and Nathan Collins.

* This work was supported by the National Science Foundation (award 0910510)

22nd International Conference on Rewriting Techniques and Applications.
Editor: M. Schmidt-Schauß; pp. 9–9

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

FAST: An Efficient Decision Procedure for Deduction and Static Equivalence

Bruno Conchinha[1], David A. Basin[2], and Carlos Caleiro[3]

1 **Information Security Group, ETH Zürich, Zürich, Switzerland**
`bruno.conchinha@inf.ethz.ch`
2 **Information Security Group, ETH Zürich, Zürich, Switzerland**
`basin@inf.ethz.ch`
3 **SQIG - Instituto de Telecomunicações, Department of Mathematics, IST, TU Lisbon, Portugal**
`ccal@math.ist.utl.pt`

Abstract

Message deducibility and static equivalence are central problems in symbolic security protocol analysis. We present FAST, an efficient decision procedure for these problems under subterm-convergent equational theories. FAST is a C++ implementation of an improved version of the algorithm presented in our previous work [10]. This algorithm has a better asymptotic complexity than other algorithms implemented by existing tools for the same task, and FAST's optimizations further improve these complexity results.

We describe here the main ideas of our implementation and compare its performance with competing tools. The results show that our implementation is significantly faster: for many examples, FAST terminates in under a second, whereas other tools take several minutes.

Keywords and phrases Efficient algorithms, Equational theories, Deducibility, Static equivalence, Security protocols

Digital Object Identifier 10.4230/LIPIcs.RTA.2011.11

Category System Description

1 Introduction

Cryptographic protocols are widely used to provide secure network communication. It is therefore important that such protocols are proven correct. Automated tools for this task rely on symbolic protocol models, in which cryptographic primitives are modeled as function symbols and messages exchanged over the network are represented by terms in a term algebra. Properties of cryptographic primitives are represented as equational theories relating terms in the term algebra.

Message deducibility and static equivalence are two important problems in the analysis of security protocols. The deducibility problem consists of deciding whether an attacker can use a set of messages (for instance, the messages exchanged over a network) to compute a secret message. The knowledge of an attacker is often expressed in terms of deducibility, *i.e.*, the set of messages he can deduce, and most automated methods for protocol analysis rely on this notion [5, 4]. Static equivalence is used to model the notion that an attacker cannot distinguish between two sequences of messages [3]. It has been used to model several indistinguishability related security properties, including security against off-line guessing attacks [12, 6, 2] and anonymity in e-voting protocols [13].

22nd International Conference on Rewriting Techniques and Applications.
Editor: M. Schmidt-Schauß; pp. 11–20



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

We present an implementation of the algorithm proposed in [10] for efficiently deciding deduction and static equivalence. Although existing tools [7, 9] solve these problems for more general equational theories, our algorithm has significantly better asymptotic complexity for the class of subterm-convergent equational theories. This class is general enough to model many equational theories relevant in practice. FAST implements several optimizations to the algorithm presented in [10], further improving our asymptotic complexity results. We have analyzed its performance using as benchmarks different practically relevant examples of subterm-convergent theories. As expected, the results show that FAST is significantly faster than other algorithms for the equational theories it handles.

In Section 2 we formally introduce the notions of message deducibility and static equivalence. In Section 3 we describe our implementation and the improvements we made to the algorithm given in [10]. In Section 4 we present our benchmark tests and compare the performance of FAST with existing tools. We draw conclusions in Section 5.

2 Background and decision problems

We consider *signatures* $\Sigma = \biguplus_{n\in\mathbb{N}} \Sigma_n$ consisting of finitely many function symbols, where Σ_i contains the functions symbols of arity i. We also fix countably infinite, disjoint sets Var and Name of variables and names.

Given a set X, $T(\Sigma, X)$ is the set of Σ-terms over X, *i.e.*, the smallest set such that $X \subseteq T(\Sigma, X)$ and $f(t_1, \ldots, t_n) \in T(\Sigma, X)$ for all $t_1, \ldots, t_n \in T(\Sigma, X)$ and all $f \in \Sigma_n$. Given $t \in T(\Sigma, X)$, we define the set $sub(t)$ of *subterms* of t as usual: if $t \in X$, then $sub(t) = \{t\}$; if $t = f(t_1, \ldots, t_n)$ for some $f \in \Sigma_n$ and $t_1, \ldots, t_n \in T(\Sigma, X)$, then $sub(t) = \{t\} \cup \bigcup_{i=1}^{n} sub(t_i)$. We denote by $vars(t) = sub(t) \cap \mathsf{Var}$ the set of variables occurring in t. The *size* $|t|$ of a term $t \in T(\Sigma, X)$ is given by $|t| = 1$ if $t \in X$ and $|t| = 1 + \sum_{i=1}^{n} |t_i|$ if $t = f(t_1, \ldots, t_n)$.

We use the standard notion of *substitution* as a partial function $\sigma\colon \mathsf{Var} \nrightarrow T(\Sigma, X)$ with a finite domain. We abuse notation by using the same symbol σ for a substitution and its natural (homomorphic) extension to $T(\Sigma, X)$, where $dom(\sigma) \subseteq X$. We write $t\sigma$ instead of $\sigma(t)$.

A *frame* is a pair $(\tilde{n}, \sigma)$, written $\nu\tilde{n}.\sigma$, where $\tilde{n} \subseteq \mathsf{Name}$ is a finite set of names and $\sigma\colon \mathsf{Var} \nrightarrow T(\Sigma, \mathsf{Name})$ is a substitution with a finite domain. The *size* $|\phi|$ of a frame ϕ is given by $|\phi| = \sum_{x\in dom(\sigma)} |x\sigma|$. Given a frame $\phi = \nu\tilde{n}.\sigma$, we define $T_\phi = T(\Sigma, (\mathsf{Name} \setminus \tilde{n}) \cup dom(\sigma))$. We say that terms in T_ϕ are *ϕ-recipes*. A term t can be *constructed* from ϕ if there is a ϕ-recipe t' such that $t\sigma = t'$ (syntactically). Frames are used to represent an attacker's knowledge: the names in $\tilde{n}$ represent randomly generated nonces unknown to the attacker, and the terms in σ's range represent the messages learned by the attacker, for instance, the messages exchanged over a network.

A rewrite rule is a pair (l, r), written as $l \to r$, where $l, r \in T(\Sigma, \mathsf{Var})$. A rewriting system R over Σ is a set of rewrite rules. We always assume that rewrite systems are finite. Given a rewriting system R, we define the relation $\to_R \subseteq T(\Sigma, \mathsf{Name}) \times T(\Sigma, \mathsf{Name})$ as the smallest relation such that:

- if $(l \to r) \in R$ and $\sigma\colon vars(l) \to T(\Sigma, \mathsf{Name})$ is a substitution, then $l\sigma \to_R r\sigma$, and
- if $t_1, \ldots, t_n, t_i' \in T(\Sigma, \mathsf{Name})$, $t_i \to_R t_i'$, and $f \in \Sigma_n$, then $f(t_1, \ldots, t_i, \ldots, t_n) \to_R f(t_1, \ldots, t_i', \ldots, t_n)$.

If the rewriting system $\to_R$ is convergent, then each term t has a unique normal form $t{\downarrow_R} \in T(\Sigma, \mathsf{Name})$. In this case, we define $\approx_R \subseteq T(\Sigma, \mathsf{Name}) \times T(\Sigma, \mathsf{Name})$ as the equational theory such that $t \approx_R t'$ if and only if $t{\downarrow_R} = t'{\downarrow_R}$. As usual, we write $t \approx_R t'$ instead of $(t, t') \in \approx_R$.

A rewriting system R is subterm convergent if $\rightarrow_R$ is convergent and, for each $(l \rightarrow r) \in R$, $r \in sub(l)$. It is weakly subterm convergent if, for each $(l \rightarrow r) \in R$, either $r \in sub(l)$ or $r \in T(\Sigma, \emptyset)$ is in normal form.

Deducibility and static equivalence

▶ **Definition 2.1.** Given a frame $\phi = \nu\tilde{n}.\sigma$, a term $t \in T(\Sigma, \mathsf{Name})$, and a rewriting system R, we say that t is *deducible from* ϕ *under* R, and write $\phi \vdash_R t$, if there is a ϕ-recipe t' such that $t'\sigma \approx_R t$.

▶ **Definition 2.2.** Given two frames $\phi = \nu\tilde{n}.\sigma$ and $\phi' = \nu\tilde{n}'.\sigma'$ and a rewriting system R, we say that ϕ and ϕ' are *statically equivalent under* R, and write $\phi \approx^s_R \phi'$, if $T_\phi = T_{\phi'}$ (*i.e.*, $\tilde{n} = \tilde{n}'$ and $dom(\sigma) = dom(\sigma')$) and, for all $t, t' \in T_\phi$, $t\sigma \approx_R t'\sigma$ if and only if $t\sigma' \approx_R t'\sigma'$.

The message deducibility problem is concerned with whether an attacker who has learned the messages represented by the terms in the range of σ can use those messages to compute (*deduce*) a secret message t without using the (secret) names in $\tilde{n}$. Static equivalence formalizes that two sequences of messages are indistinguishable from an attacker's point of view. This is modeled as the condition that there are no two recipes that yield (equationally) equal terms under the substitution σ but not under σ' (or vice-versa). This is useful, for instance, in modeling off-line guessing attacks [12, 10].

3 Fast algorithm

Fast is a C++ implementation of the algorithm described in [10]. It solves the message deducibility and static equivalence problems for weakly subterm-convergent rewriting systems. Such rewriting systems are sufficiently expressive to represent a standard Dolev-Yao equational theory with one-way functions, pairing, projections and symmetric and asymmetric encryption and decryption. They can also model idempotent functions and signature schemes, among others.

Other existing tools for solving these two decision problem are Yapa [7], Kiss [9], and ProVerif [8]. However, ProVerif is designed for solving the harder problem of protocol security under active adversaries. Therefore, it is not surprising that it performs significantly slower for these two problems. Moreover, ProVerif is not guaranteed to terminate even under subterm-convergent equational theories, and Yapa claims a performance between one and two orders of magnitude faster than ProVerif [7]. We thus compare our algorithm with Yapa and Kiss.

Compared to Yapa and Kiss, Fast has a better asymptotic complexity, but a narrower application scope. For instance, Yapa and Kiss can handle rewriting systems representing blind signatures and homomorphic encryption. Kiss can also handle trapdoor committments. Fast cannot handle these rewriting systems. We believe, however, that many of the techniques and results that allow Fast to achieve a better asymptotic complexity do not depend on the subterm-convergent hypothesis (*cf.* Section 5).

3.1 Procedures and complexity

Data structures

Fast represents terms as DAGs (Directed Acyclic Graphs). Each term is represented by a C++ object which we call a *term vertex*. Term vertices contain as a class member a C++ object representing a recipe, which we will call a *recipe vertex*. Furthermore, each recipe

vertex contains as a class member an array of term vertices. These term vertices represent the normal forms of the terms corresponding to the recipe in each of the input frames. These normal forms are computed whenever a recipe vertex is created. Whenever a recipe is added to a vertex v, all the parent vertices of v are checked; if all children of a parent vertex pv have recipes, then a new recipe is computed and added to pv.

Saturation procedure

Similar to YAPA and KISS, FAST relies on a frame saturation procedure. However, FAST's saturation procedure generates less terms and is therefore more efficient than these other procedures. Namely, FAST only adds to the saturated frame those terms that are instances of right-hand sides of rules (rather than all deducible subterms). Furthermore, FAST only instantiates a rule if there is some subterm of the left-hand side that is mapped to a term already in the frame and each variable is mapped to either a subterm of a term in the original frame or a fresh nonce. A more detailed description of the saturation procedure, as well as a comparison with the procedures used in other tools, is given in [11].

Given a frame $\phi = \upsilon\tilde{n}.\sigma$, FAST's saturation procedure computes two sets Γ_s and Γ_l of term vertices. Γ_s represents a set of subterms in the range of σ that are deducible from ϕ. Γ_l represents the set of instantiations $l\sigma_l$ of left-hand sides l of rewrite rules such that $\sigma_l \colon vars(l) \to sub(ran(\sigma)) \cup \Upsilon$, where Υ is a finite set of fresh nonces computed from the rewrite system, and there is some $s \in sub(l)$ such that $s\sigma_l \in \Gamma_s$.

Γ_s is initialized as $ran(\sigma)$. Recipes are added to each vertex representing a term in $ran(\sigma)$ and to each leaf vertex. The program then tries to create instances $l\sigma_l$ of left-hand sides of rules which verify the conditions above. To do this, for each $t \in \Gamma_s$, each rewrite rule $l \to r$, and each $s \in sub(l)$, FAST checks whether t and s can be unified, and then attempts to extend the substitution unifying these terms to all the variables in l.

Whenever such an instance is found, the term vertices needed to represent that instance are created, and the term vertex representing $l\sigma_l$ is added to Γ_l. When a recipe ζ is added to a term vertex in Γ_l representing one such instance $l\sigma_l$, if $r\sigma_l$ does not have a recipe yet, then the recipe ζ is added to $r\sigma_l$, and $r\sigma_l$ is added to Γ_s. $r\sigma_l$ can then be used to find more instances of left-hand sides of rules.

Deducibility and static equivalence

After obtaining the saturated frame ϕ_s, deciding whether a term is deducible from ϕ amounts to deciding whether it can be constructed from the terms represented by term vertices in Γ_s. From Γ_s and Γ_l one also obtains a finite set Eq_ϕ of equations between ϕ-recipes satisfied by ϕ and such that, if another frame ϕ' with $T_\phi = T_{\phi'}$ satisfies all equations in Eq_ϕ, then ϕ' satisfies all equations satisfied by ϕ. These sets of equations are the same as those tested by the algorithm described in [10]. Therefore, given frames ϕ and ϕ' such that $T_\phi = T_{\phi'}$, one can decide whether $\phi \approx^s_R \phi'$ by building the corresponding saturated frames, obtaining the sets of equations Eq_ϕ and $\mathsf{Eq}_{\phi'}$ described above, and then testing whether ϕ satisfies all equations in $\mathsf{Eq}_{\phi'}$ and vice-versa.

Improvements over [10]

Our implementation improves the algorithm described in [10] in several ways. The most relevant improvement is that FAST only considers instances $l\sigma_l$ of left-hand sides of rules if there is some $s \in sub(l)$ such that $s\sigma_l$ is in the range of σ_s. This makes the saturation procedure faster and also reduces the number of equations that the algorithm must check

to decide static equivalence. Another important improvement is that recipes are computed at most once for each vertex. Furthermore, whenever a new recipe $t \in T_\phi$ is added, the implementation creates a recipe object containing pointers to the vertices $(t\sigma)\downarrow$ and $(t\sigma')\downarrow$. The proof of correctness of the algorithm presented in [10] shows that one must only consider instances $l\sigma_l$ of left-hand sides of rules if $r\sigma_l$ is in normal form; therefore, this only requires matching the term resulting from the recipe with each left-hand side of a rule, discarding it if the term obtained after one step of rewriting is not in normal form. This can be done in constant time.

3.2 Asymptotic complexity

The above improvements reduce the asymptotic complexity of the algorithm for some equational theories. For instance, consider the rewriting system

$$\mathcal{DY}_{asym} = \left\{ \pi_1(\langle x, y\rangle) \to x, \pi_2(\langle x, y\rangle) \to y, \left\{\{x\}_y\right\}_y^{-1} \to x, \left\{\{x\}_{y_{\mathsf{pub}}}\right\}_{y_{\mathsf{priv}}}^{-1} \to x \right\}$$

used in [10] for comparing the asymptotic complexity of the algorithm introduced there with YAPA and KISS. The asymptotic complexity of KISS is estimated in [10] to be $\mathcal{O}(|\phi|^7)$ under $\mathcal{DY}_{asym}$. For this rewrite system, YAPA has exponential complexity in the worst case scenario, since it does not implement DAG representation of terms. Even if DAGs were implemented, the asymptotic complexity of YAPA's saturation procedure is estimated (again in [10]) to be $\mathcal{O}(|\phi|^7)$. The asymptotic complexity of the algorithm presented in [10] for this rewrite system is estimated to be $\mathcal{O}((|\phi| + |\phi'|)^3 \log^2(|\phi| + |\phi'|)$ for static equivalence. Given that each recipe is associated with the normal forms of the terms represented by the recipe in each frame (thus eliminating the overhead of computing normal forms), FAST's complexity for static equivalence under $\mathcal{DY}_{asym}$ is only $\mathcal{O}((|t| + |\phi|)^2 \log^2(|t| + |\phi|))$. A detailed analysis of FAST's asymptotic complexity is given in [11].

4 Performance analysis

We have considered several families of interesting and practically relevant examples to compare the performance of our algorithm with YAPA and KISS. The results show great disparities in the performance of the three algorithms. Neither KISS nor YAPA show a clear advantage over the other: depending on the example, either algorithm may perform significantly faster than the other. As expected from the complexity results, FAST generally performs much better than either of these algorithms, particularly for static equivalence. Even for artificial equational theories designed to produce worst case performance for our algorithm, FAST is still more efficient for static equivalence, sometimes significantly so. For message deducibility under such equational theories, FAST performs better in most examples; however, in a few, it appears to be slower by a small constant.

All our tests were performed on a computer with an Intel Core 2 Duo processor running at 2.53GHz and with 4Gb memory. In all our static equivalence tests, we consider two equal frames. Similarly, in all our deduction tests, the input term is a secret that does not occur in the range of the substitution of the input frame. Therefore, the result is positive in all static equivalence tests and negative in all deducibility tests. This does not affect the algorithm's performance significantly, as both frames still have to be saturated in all implementations — that is, deducible subterms must still be added to the saturation, and the sets of equations which must be tested to check for static equivalence must still be generated (*cf.* Section 3). Static equivalence takes a slightly longer time in this case because all equations must be

checked rather than stopping as soon as a counter-example is found. However, the difference is minor. We present here an illustrative sample of the tests performed. For a more complete report on our results, see [1].

4.1 Chained keys

This family of tests uses the signature $\Sigma^{\mathcal{DY}}$, with $\Sigma_1^{\mathcal{DY}} = \{\pi_1, \pi_2\}$ and $\Sigma^{\mathcal{DY}} = \{\{\cdot\}_\cdot, \{\cdot\}_\cdot^{-1}, \langle\cdot,\cdot\rangle\}$, and a rewriting system representing a standard Dolev-Yao intruder without asymmetric encryption, specified by the set of rewrite rules

$$\mathcal{DY} = \left\{\pi_1(\langle x, y\rangle) \to x, \pi_2(\langle x, y\rangle) \to y, \left\{\{x\}_y\right\}_y^{-1} \to x\right\}.$$

For $n \in \mathbb{N}$, we define the frame $\phi_n^{ck} = \upsilon\tilde{n}_n^{ck}.\sigma_n^{ck}$, where $\tilde{n}_n^{ck} = \{k, k_0, \ldots, k_n\}$ and $\sigma = \left\{x_1 \mapsto \{k_0\}_{k_1}, \ldots, x_n \mapsto \{k_{n-1}\}_{k_n}, x_{n+1} \mapsto k_n\right\}$. For each parameter n, the deduction problem is to decide whether $\phi_n^{ck} \vdash_{\mathcal{DY}} k$, and the static equivalence problem is to decide whether $\phi_n^{ck} \approx_{\mathcal{DY}}^s \phi_n^{ck}$.

Fast has a much better performance than both Yapa and Kiss for these examples. Yapa also performs much better than Kiss. Tables 1 and 2 illustrate these relationships.

Table 1 Performance on *chained keys* for deduction (time in ms)

Parameter	50	100	200	500	1000	2000	5000
Fast	11	20	40	143	224	474	1526
Kiss	259	1730	12655	288606	> 300000	> 300000	> 300000
Yapa	31	108	415	4624	11297	62457	> 300000

Table 2 Performance on *chained keys* for static equivalence (time in ms)

Parameter	50	100	200	500	750	1500	2500
Fast	20	41	88	247	424	1020	1546
Kiss	1341	12185	127828	> 300000	> 300000	> 300000	> 300000
Yapa	143	744	5516	18467	44451	197648	> 300000

4.2 Composed keys

This family of examples uses the same signature $\Sigma^{\mathcal{DY}}$ and the same rewriting system $\mathcal{DY}$ used in Section 4.1.

For $n, s, i \in \mathbb{N}$, define $t_{n,s}^i$ recursively by

$$t_{n,s}^0 = \{\langle k_{2s-1}, k_{2s-2}\rangle\}_{\langle k_{2s}, k_{s2+1}\rangle},$$
$$t_{n,s}^i = \left\{\langle t_{n,s}^{i-1}\langle k_{2s+1+2i(n-1)}, k_{2s+2i(n-1)}\rangle, \right\}_{\langle k_{2s+2+2i(n-1)}, k_{2s+3+2i(n-1)}\rangle}.$$

For $k > 0$, the frame $\phi_k^c = \upsilon\tilde{n}_n^c.\sigma_n^c$ is such that $\tilde{n}_n^c = \{k, k_0, \ldots, k_{2n^2+1}\}$ and $\sigma_n^c = \left\{x_1 \mapsto t_{n,1}^{n-1}, \ldots, x_n \mapsto t_{n,n}^{n-1}, x_{n+1} \mapsto k_{2n^2}, x_{n+2} \mapsto k_{2n^2+1}\right\}$. The deduction problem corresponding to parameter n considered in our tests is to decide whether $\phi_n^c \vdash_{\mathcal{DY}} k$. The static equivalence problem corresponding to parameter n is to decide whether $\phi_n^c \approx_{\mathcal{DY}}^s \phi_n^c$.

This family of examples is particularly challenging because the decryption keys are pairs of secrets. At each point of the algorithm's execution, decrypting the right message yields

a pair of previously unknown secrets. This pair may then be used to compose the next decryption key by exchanging the order of the terms in the pair. As illustrated in Tables 3 and 4, the difference in FAST's performance is particularly marked in this example. KISS also performs much better than YAPA.

Table 3 Performance on *composed* for deduction (time in ms)

Parameter	3	4	5	7	9	10	20
FAST	7	11	17	34	61	126	945
KISS	138	867	3760	46369	245207	> 300000	> 300000
YAPA	158	34118	> 300000	> 300000	> 300000	> 300000	> 300000

Table 4 Performance on *composed* for static equivalence (time in ms)

Parameter	3	4	5	6	8	10	20
FAST	12	21	28	48	92	148	1635
KISS	469	2625	10428	252000	> 300000	> 300000	> 300000
YAPA	936	157358	> 300000	> 300000	> 300000	> 300000	> 300000

4.3 Denning-Sacco shared key protocol

The Denning-Sacco symmetric key protocol [14] is used to establish session keys in a network with a single server and multiple agents. Each agent shares a (secret) symmetric key with the server, but there are no shared keys between agents. In Alice&Bob notation, the protocol is as follows.

1. $\mathsf{A} \to \mathsf{S}$: A, B
2. $\mathsf{S} \to \mathsf{A}$: $\{\mathsf{A}, K_{\mathsf{A},\mathsf{B}}, T, \{K_{\mathsf{A},\mathsf{B}}, \mathsf{A}, T\}_{K_{\mathsf{S},\mathsf{B}}}\}_{K_{\mathsf{S},\mathsf{A}}}$
3. $\mathsf{A} \to \mathsf{B}$: $\{K_{\mathsf{A},\mathsf{B}}, \mathsf{A}, T\}_{K_{\mathsf{S},\mathsf{B}}}$

Here, A and B are two participants, and S is the server. A requests from the server a session key to communicate with B. The server generates a new session key, $K_{\mathsf{A},\mathsf{B}}$, and sends it to A, encrypted with the (symmetric) key shared between A and S. This message also contains a timestamp T, used to determine the validity of the new session key, and the ticket $\{K_{\mathsf{A},\mathsf{B}}, \mathsf{A}, T\}_{K_{\mathsf{S},\mathsf{B}}}$. A then forwards this ticket to B, who can decrypt it using the key $K_{\mathsf{S},\mathsf{B}}$ shared between B and S, to obtain the new session key $K_{\mathsf{A},\mathsf{B}}$, the name A of the intended communication partner, and the time T of the request.

This example uses the result of executing multiple sessions of the Denning-Sacco protocol. For the parameter n we assume a network with $3n$ participants, each of which initiates one session with each other participant. We assume that one third of the shared keys between the server and the agents are compromised, *i.e.*, available to the attacker.

We will once again use the signature $\Sigma^{\mathcal{DY}}$ and the rewriting system $\mathcal{DY}$ from Section 4.1. For parameter n, we thus use the frame $\phi_n^{ds} = \tilde{n}_n^{ds}.\sigma_n^{ds}$, with $\sigma_n^{ds} = \sigma_n^1 \cup \sigma_n^2 \cup \sigma_n^3 \cup \sigma_n^4$, where

- $\sigma_n^1 = \left\{x_i^1 \mapsto K_{\mathsf{S},i} \mid i \in \{1,\ldots,n\}\right\}$;
- $\sigma_n^2 = \left\{x_{i,j}^2 \mapsto \langle \mathsf{A}_i, \mathsf{A}_j\rangle \mid i,j \in \{1,\ldots,3n\}, i \neq j\right\}$;
- $\sigma_n^3 = \left\{x_{i,j}^3 \mapsto \left\{\langle \mathsf{A}_j, \langle K_{i,j}, \langle T_{i,j}, \{\langle K_{i,j}, \langle \mathsf{A}_i, T_{i,j}\rangle\rangle\}_{K_{\mathsf{S},j}}\rangle\rangle\right\}_{K_{\mathsf{S},i}} \mid i,j \in \{1,\ldots,3n\}, i \neq j\right\}$;
- $\sigma_n^4 = \left\{x_{i,j}^4 \mapsto \{\langle K_{i,j}, \langle \mathsf{A}_i, T_{i,j}\rangle\rangle\}_{K_{\mathsf{S},j}} \mid i,j \in \{1,\ldots,3n\}, i \neq j\right\}$,

and $\tilde{n}_n^{ds} = \{K_{i,j}, T_{i,j}, K_{\mathsf{S},i} \mid i,j \in \{1,\ldots,3n\}\}$.
Here, σ_n^1 represents the keys compromised by the attacker and σ_n^2, σ_n^3, and σ_n^4 represent the messages exchanged as part of the execution of the first, second, and third steps of the protocol, respectively. The deduction problem is to decide whether $\phi_n^{ds} \vdash_{\mathcal{DY}} K_{\mathsf{S},3n}$ and the static equivalence problem is to decide whether $\phi_n^{ds} \approx^s_{\mathcal{DY}} \phi_n^{ds}$.

YAPA performs noticeably better than KISS in this example. FAST, as before, is significantly faster than both. The results are shown in Tables 5 and 6.

Table 5 Performance on *Denning-Sacco* for deduction (time in ms)

Parameter	5	7	9	11	12	14	24
FAST	337	768	1336	2588	2239	5083	20073
KISS	6637	30195	91743	232093	> 300000	> 300000	> 300000
YAPA	2409	10320	30732	68845	74298	172249	> 300000

Table 6 Performance on *Denning-Sacco* for static equivalence (time in ms)

Parameter	3	5	7	9	11	13	20
FAST	181	585	1281	2300	6598	7507	24614
KISS	1219	8543	34726	158717	> 300000	> 300000	> 300000
YAPA	446	2836	12300	52506	134391	269781	> 300000

4.4 FAST worst case

In this family of examples, for the parameter n, we use the signature Σ^{wc}, where $\Sigma_1^{wc} = \{f\}$ and $\Sigma_n^{wc} = \{h\}$. The rewriting system is given by the set $wc_n = \{h(f(x_1),\ldots,f(x_n)) \to x_1\}$. We define the frame $\phi_n^{wc} = \tilde{n}_n^{wc}.\sigma_n^{wc}$, where $\tilde{n}_n^{wc} = \{k, k_1, \ldots, k_n\}$ and $\sigma_n^{wc} = \{x_1 \mapsto f(k_1), \ldots, x_n \mapsto f(k_n)\}$. The deduction problem is to decide whether $\phi_n^{wc} \vdash_{wc_n} k$ and the static equivalence problem is to decide whether $\phi_n^{wc} \approx^s_{wc_n} \phi_n^{wc}$.

This example is challenging because, to saturate this frame, FAST must instantiate each element of the tuple with each of the secret names. Therefore, the asymptotic complexity of FAST for this family is $\mathcal{O}(n^n)$. Note that this does not contradict the fact that, for a given rewriting system, FAST has polynomial-time complexity; the exponential complexity results from the fact that the size of the rewriting system itself increases with the parameter n.

Table 7 Performance on *worst case* for deduction (time in ms)

Parameter	3	4	5	6
FAST	9	72	1192	32487
KISS	10	47	866	21446
YAPA	11	161	6607	> 300000

None of the existing algorithms perform well on this example: FAST's performance is comparable to that of KISS and YAPA performs significantly worse. This is illustrated in Tables 7 and 8.

Table 8 Performance on *worst case* for static equivalence (time in ms)

Parameter	3	4	5	6
Fast	15	142	2199	56312
Kiss	16	146	2125	69533
Yapa	16	297	8862	> 300000

Nonlinear terms

It is interesting to note that Fast's complexity depends chiefly on the number of different variables in the rewriting system. Therefore, it's performance is not significantly affected if the left-hand sides of rewrite rules are non-linear. This is not the case for the other algorithms, whose performance degrades when the complexity of the terms in the rewriting system increases, even when the number of variables remains the same. Tables 9 and 10 illustrate this point. Here, the rewriting system considered is $wc2_n = \{h(f(x_1), f(x_1), \ldots, f(x_n), f(x_n)) \to x_1\}$. The frames and problems considered here are the same as above.

Table 9 Performance on *worst case 2* for deduction (time in ms)

Parameter	2	3	4	5	6
Fast	7	9	148	1567	43282
Kiss	9	99	4381	183236	> 300000
Yapa	45	> 300000	> 300000	> 300000	> 300000

Table 10 Performance on *worst case 2* for static equivalence (time in ms)

Parameter	2	3	4	5	6
Fast	4	17	396	6197	47937
Kiss	10	292	16135	> 300000	> 300000
Yapa	56	> 300000	> 300000	> 300000	> 300000

5 Discussion and future work

Fast is an efficient algorithm for deciding deduction and static equivalence under weakly subterm-convergent rewriting systems. The implementation and our benchmarks are available for download at [1]. Fast's scope is narrower than that of other existing tools for these problems, but is broad enough to represent many practically relevant theories. As expected from the results in [10], Fast is significantly faster than both Yapa and Kiss in almost all our tests. Even for the artificial examples designed to degrade its performance, Fast still compares favorably to other algorithms: it is either faster, or slower by only a small constant. This constitutes a significant advantage since the problematic cases for Yapa and Kiss degrade these algorithms' performances dramatically.

We believe that many of the ideas and results that allow Fast to achieve better asymptotic results may still be valid with weaker hypotheses on the rewriting system. Therefore, extending the algorithm to handle more general equational theories without significantly degrading its performance is an important research goal.

Acknowledgements This work was partly supported by FCT and EU FEDER, namely via the project PTDC/EIA-CCO/113033/2009 ComFormCrypt of SQIG-IT and the project UTAustin/MAT/0057/2008 AMDSC of IST. The first author acknowledges the support of FCT via the PhD grant SFRH/BD/44204/2008 and the National Competence Center in Research on Mobile Information and Communication Systems (NCCR-MICS), a center supported by the Swiss National Science Foundation. We would also like to thank Mohammad Torabi Dashti and Benedikt Schmidt for their helpful comments on this paper.

References

1 `http://www.infsec.ethz.ch/people/brunoco`, 2011.
2 Martin Abadi, Mathieu Baudet, and Bogdan Warinschi. Guessing attacks and the computational soundness of static equivalence. *Journal of Computer Security*, pages 909–968, December 2010.
3 Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. *SIGPLAN Not.*, 36:104–115, January 2001.
4 Alessandro Armando, David Basin, Yohan Boichut, Yannick Chevalier, Luca Compagna, Jorge Cuellar, Paul Hankes Drielsma, Pierre-Cyrille Heám, Jacopo Mantovani, Sebastian Mödersheim, David von Oheimb, Michaël Rusinowitch, Judson Santiago, Mathieu Turuani, Luca Viganò, and Laurent Vigneron. The AVISPA Tool for the Automated Validation of Internet Security Protocols and Applications. In Kousha Etessami and Sriram K. Rajamani, editors, *Proceedings of the 17th International Conference on Computer Aided Verification (CAV'05)*, volume 3576 of *LNCS*. Springer, 2005.
5 Charu Arora and Mathieu Turuani. Validating Integrity for the Ephemerizer's Protocol with CL-Atse. In *Formal to Practical Security: Papers Issued from the 2005-2008 French-Japanese Collaboration*, volume 5458 of *LNCS*, pages 21–32. Springer, 2009.
6 Mathieu Baudet. Deciding security of protocols against off-line guessing attacks. In *Proceedings of the 12th ACM conference on Computer and communications security*, CCS '05, pages 16–25, New York, NY, USA, 2005. ACM.
7 Mathieu Baudet, Véronique Cortier, and Stéphanie Delaune. YAPA: A generic tool for computing intruder knowledge. In Ralf Treinen, editor, *RTA*, volume 5595 of *Lecture Notes in Computer Science*, pages 148–163. Springer, 2009.
8 Bruno Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In *Proceedings of the 14th IEEE workshop on Computer Security Foundations*, CSFW '01, pages 82–96, Washington, DC, USA, 2001. IEEE Computer Society.
9 Stefan Ciobâca, Stéphanie Delaune, and Steve Kremer. Computing knowledge in security protocols under convergent equational theories. In *CADE*, pages 355–370, 2009.
10 Bruno Conchinha, David Basin, and Carlos Caleiro. Efficient algorithms for deciding deduction and static equivalence. In *Proc. 7th Int. Workshop on Formal Aspects of Security and Trust (FAST'10)*, 2010.
11 Bruno Conchinha, David Basin, and Carlos Caleiro. Efficient algorithms for deciding deduction and static equivalence. volume 680 of *ETH Technical Reports*. ETH Zürich, Information Security Group D-INFK, September 2010.
12 Ricardo Corin, Jeroen Doumen, and Sandro Etalle. Analysing password protocol security against off-line dictionary attacks. *Electron. Notes Theor. Comput. Sci.*, 121:47–63, February 2005.
13 Stéphanie Delaune, Steve Kremer, and Mark Ryan. Verifying privacy-type properties of electronic voting protocols. *J. Comput. Secur.*, 17:435–487, December 2009.
14 Dorothy E. Denning and Giovanni Maria Sacco. Timestamps in key distribution protocols. *Commun. ACM*, 24:533–536, August 1981.

Automated Certified Proofs with CiME3*

Évelyne Contejean[1], Pierre Courtieu[2], Julien Forest[3], Olivier Pons[2], and Xavier Urbain[4]

1 **CNRS, LRI, UMR 8623, Orsay, F-91405**
Univ Paris-Sud 11, Orsay, F-91405, France
INRIA Saclay - Île-de-France, ProVal, Orsay, F-91893, France
evelyne.contejean@lri.fr
2 **CNAM, Lab. Cédric, Paris, F-75141, France**
{pierre.courtieu,olivier.pons}@cnam.fr
3 **ENSIIE, Évry, F-91025**
Lab. Cédric, Paris, F-75141, France
julien.forest@ensiie.fr
4 **ENSIIE, Évry, F-91025**
INRIA Saclay - Île-de-France, ProVal, Orsay, F-91893, France
LRI, Univ Paris-Sud 11, CNRS, Orsay, F-91405, France
xavier.urbain@ensiie.fr

Abstract

We present the rewriting toolkit C*i*ME3. Amongst other original features, this version enjoys two kinds of engines: to handle and discover proofs of various properties of rewriting systems, and to generate Coq scripts from proof traces given in *certification problem format* in order to certify them with a sceptical proof assistant like Coq. Thus, these features open the way for using C*i*ME3 to add automation to proofs of termination or confluence in a formal development in the Coq proof assistant.

Keywords and phrases Rewriting, formal proof, proof automation, proof assistant

Digital Object Identifier 10.4230/LIPIcs.RTA.2011.21

Category System Description

1 Introduction

Automated tools based on complex mathematical arguments have become widely used in various domains of computer science. Cryptographic systems, proof assistants or systems analysing programs involve highly intricate deduction procedures that are indeed beyond human capabilities. However, this situation raises a real issue as the downside of such deductive power is the difficulty to trust a result that no human can check.

Regarding rewriting, and in particular automated termination proof, there have been many new tools since the introduction of the dependency pair approach [1] at the end of the 90's: C*i*ME 2, AProVE [13], T$_{\text{T}}$T$_2$ [17], Jambox,[1] etc., to cite a few of them.

However, all these tools exhibited incorrect behaviour at some point, in particular during the Termination Competition [20] or its preparatory rounds. Several approaches to certify the results of automated provers with skeptical proof assistants have been developed: A3PAT [5, 4] and CoLoR/Rainbow [2] for Coq, CeTA [22] for Isabelle/HOL.

* This work was partially supported by the french ANR project A3PAT (ANR-05-BLAN-0146).
[1] http://joerg.endrullis.de/jambox.html

As a new member of the C*i*ME family (`http://cime.lri.fr`), C*i*ME3 is a toolbox dedicated to the handling and analysis of rewriting programs. It allows one to define term algebras, rewriting systems, etc., and to perform a range of treatments over them: computation, normalisation, matching and unification (modulo equational theories), completion (Knuth-Bendix) and proofs of equality (both modulo equational theories), etc. An important part of C*i*ME3 is dedicated to proofs of termination (including solving of ordering constraints), local confluence and convergence. C*i*ME3 enjoys a top level mode for interactive development of rewrite programs; it can also be used in batch mode with full automation.

C*i*ME3 has been developed within the A3PAT project for certification of automated proofs, and automation for proof assistants (`http://a3pat.ensiie.fr`). In comparison to its ancestors, C*i*ME3 features, in addition to its various proof engines, a certification mechanism which issues proof traces and *certificates*, and which allows one to check that the results are correct. To be certified, a proof trace can be translated into a script checked by a trusted tool: a skeptical proof assistant like Coq, or it can be given directly to a certified dedicated tool like CeTA. A unique feature of C*i*ME3 is in particular the (discovery and) certification of proofs of convergence.

The present article is organised as follows. In Section 2 we present some of the proof engines at work in C*i*ME3, for termination in Sections 2.1.2 and 2.1.3, and local confluence, Knuth-Bendix completion and convergence in Section 2.2. Some criteria allow for parallel solving, this is presented in Section 2.1.4. The certification engine is then described in Section 3, along with its input, output, and the general structure of emitted Coq scripts. Section 3.3 sketches the use of C*i*ME3 in batch mode. We conclude with a list of resources and companion tools, and with future work regarding the current implementation.

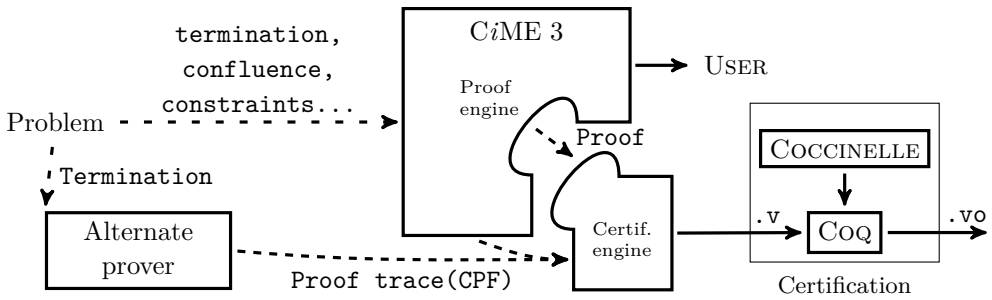


Figure 1 Global C*i*ME 3 architecture.

2 Proof Engines

2.1 Core Language, Termination

C*i*ME3 can be used in interactive or batch mode. We hereafter describe some of the basic commands for the interactive mode. Note that the batch mode can accept additional files in various formats (like `TRS` or `SRS` formats from the Termination Competition, or in CPF[2] (an XML format widely accepted by the certified rewriting community), developed by the CeTA group and our team), however the commands described here can be used in both modes. See Section 3.3 for batch specific command line options.

[2] The Certification Problem Format, `http://cl-informatik.uibk.ac.at/software/cpf/`

The core top level language of C*i*ME3 is a simple language which allows the user to type in expressions terminated by a semicolon. Simple expressions consist of booleans, integers, strings, and basic operations on them. As in previous versions of C*i*ME the nominal language is powerful enough to define functions (polymorphic higher-order, allowing partial application, functions as arguments) with a `let fun` construct. Application is denoted by juxtaposition of the function and its arguments, as in LISP or other functional languages based on the λ-calculus. A short manual is available online on the A3PAT web page. We focus here on the new syntax and features of C*i*ME3 for rewrite systems and constraints.

2.1.1 Declarations

The user can define and name objects, algebras, and systems in the C*i*ME input language akin to the definitions found in the literature: sets of variables, signatures (symbols with arities), term algebras, rewrite systems, term ordering constraints...

A `let` construct introduces *global* declarations. Listing 1 declares successively the global set `X` of variables, a signature `F` for Peano numbers and addition, and the corresponding term algebra `T`. One can then easily declare terms, rewrite systems, and even ordering constraints (built from conjunctions `/\` and disjunctions `\/` of atomic constraints).

Listing 1 A few simple declarations.

```
let X = variables "x,y";
let F = signature "plus : binary; 0:constant; S:unary;";
let T = algebra F;
let t1 = term T "S(0)";
let R = trs T " plus(0,x) -> x; plus(S x, y) -> S(plus(x,y)); ";
let c = order_constraints T "0 < S(0) /\ S(plus(x,y)) < plus(S(x),y)";
```

2.1.2 Ordering Parameters

C*i*ME3 features a proof engine dedicated to the discovery of (well-founded) term orderings that fulfil a set of (ordering) constraints. These constraints may come from a direct declaration by the user (see Listing 1), or may appear as the result of operations like the application of termination criteria on a termination problem, i.e. from termination constraints. As soon as the ordering constraints are provided, one can try to discover a relevant well-suited ordering pair $(\preceq, <)$ using the command `ordering_solve` or `ordering_solve_strict` depending on the expected monotonicity of the ordering, that is depending on whether only $\preceq$, or both $\preceq$ and $<$, respectively, are supposed to be monotonic.

Listing 2 Ordering constraints solving commands.

```
ordering_solve c;   (* Search for weakly monotonic well-founded ordering fulfilling c *)
ordering_solve_strict c;      (* Search for strictly monotonic well-founded ordering *)
```

C*i*ME3 can search for different kinds of orderings, all of which being parameterised by the user. The presently implemented orderings range from full RPO [10], with or without argument filterings (AFS) [1], to various polynomial [19, 18, 7] and matrix [11, 9] interpretations. Note that the *certification* engine can accept traces from other provers, and, thus, can handle more orderings than those that are currently searched for by the termination proof engine.

The parameters for the search are provided using the grammar in Figure 2. Ordering specifications (*Ordering_param*) are of the generic form n_1 *ord_kind* n_2 n_3 $n_4 \ldots$ where

Ordering_param ::= `linear` n | `simple` n | `quadratic` n | `rpo` | `matrix` n n $n+$
Solver_params ::= n *Ordering_param* (`;` *Solver_params*)?
Criterion ::= `manna_ness` | `lex_manna_ness` | `DP` | `DPM` | `DPG` | `ST` | `RMVx` | `RMC`
Heuristic ::= *Criterion* `{`*Solver_params*`}`? | *id* `{`*Solver_params*`}`?
| `Then [`*Heuristic_l*`]` | `Do` n *Heuristic* | `Repeat` *Heuristic* | `Solve [`*Heuristic_l*`]`
Heuristic_l ::= *Heuristic* (`;` *Heuristic_l*)?

Figure 2 Grammar for heuristics and ordering parameters.

ord_kind specifies the kind of ordering for which to search, n_1 is the (optional) timeout of the solver in seconds, n_2 (only for polynomial and matrix interpretations) specifies the upper bound of the (non-negative integral) coefficients in polynomials or matrices, n_3 (matrices only) specifies the size of the matrix coefficients, and n_4 and following parameters are the list of allowed sizes of strict sub-matrices as defined in [9]. For example, `matrix 1 3 1 2` specifies a search for matrix interpretations, with matrix coefficients less or equal to 1, 3×3 matrices, and strict sub-matrices of size either 1 *or* 2. As the optional timeout is not provided, the default will be used (no timeout). Similarly, `30 linear 3` specifies a search for linear polynomials with coefficients less or equal to 3, and a search timeout after 30s.

The user can declare several ordering parameters for different situations. This is illustrated in Listing 3 where two different ordering parameter sequences `op1` and `op2` are declared, in order to be used later in heuristics (see Listing 4).

Listing 3 Ordering parameters.

```
let op1 = params "30 linear 3; 30 simple 2; 30 rpo; 30 matrix 2 2 1; 30 matrix 1 3 1 2";
let op2 = params "linear 3; simple 2; rpo; matrix 2 2 1; matrix 1 3 1";
```

Ordering parameters are interpreted sequentially, unless a parallel search is enabled. In the latter case, several orderings are searched for at the same time, depending on the computer architecture and the specific parameters given by the user, see Section 2.1.4.

2.1.3 Criteria and Heuristics

Termination criteria are commonly seen as transformations of termination problems into sets of other termination problems (possibly empty); they can be represented as inference rules [5, 8] or processors [14]. The strategy of application of termination criteria is specified in C*i*ME3 with the help of *heuristics*. Intuitively, a successful heuristic must describe a cover of a closed termination proof tree.

C*i*ME comes with a simple heuristic description language which contains built-in criteria and criteria combinators. Heuristics are specified following the grammar described in Figure 2 where: *id* is a previously declared heuristic, `manna_ness` stands for the well-known Manna and Ness criterion [19], and `lex_manna_ness` for rule removal using lexicographic combination of orderings; `DP` and `DPM` stand respectively for unmarked and marked dependency pair criteria [1], `DPG` for the graph refinement (estimation EDG) of dependency pairs [1] (DPG returns the set of strongly connected components); `ST` denotes the extension [4] of the original subterm criterion [16]; `RMVx` denotes vertex removal in dependency graphs components [14, 15], and `RMC` stands for its variant removing all vertices of the component.

A heuristic x applies to a problem and returns the set of generated sub-problems. The following constructs allow one to combine heuristics. `Then[` x`;`y`...` `]` applies heuristic x on a problem and then, if it succeeds, heuristic y on each generated sub-problem. It fails if

either x or y fails. `Repeat` x calls x on a problem and, if x succeeds, applies `Repeat` x again recursively on sub-problems. It never fails. `Do` n x is similar to `Repeat` but uses a limit n for recursion depth. Finally `Solve [` x`;`y`... ]` tries to apply x, and in the case x does not succeed tries to apply y, etc., it fails if all heuristics failed. When all recursive calls have ended, the proof is successful if all sub-problems are empty.

Listing 4 provides two examples. Heuristic `h2` first tries to apply repeatedly rule removal by lexicographic combination (using strictly monotonic orderings) until it fails to do so. It transforms then the remaining TRS termination problem into a marked dependency pairs termination problem and repeats recursively the previously declared `h1`. Heuristic `h1` first splits the given (DP) problem into sub-problems corresponding to the strongly connected components of the dependency graph, and then tries on each component: firstly the extended subterm criterion, and then, if it failed, vertex removal with a weakly monotonic ordering.

Some criteria may be parameterised specifically, for instance the number of rewrite steps allowed in the extended subterm criterion may be set to n using command `subtermparams` n.

Listing 4 Heuristics declarations and commands.

```
let h1 = heuristic " Repeat Then [ DPG ; Solve [ ST ; RMVx {op2} ] ] ";
let h2 = heuristic " Then [ Repeat lex_manna_ness {op1} ; DPM ; h1] ";
set_heuristic "h2";                                  (* Sets the termination heuristic. *)
```

When relevant parameters are set, proof search may be launched using commands `termination`, `confluence`..., which print answers and possibly store traces.

Listing 5 Proof search.

```
termination R;
local_confluence R;
convergence R;
complete o R;                                  (* Completion of R using term ordering o *)
prove_goal o R t1 t2; (* Ordered completion of R, stops when t1 and t2 are found equal*)
unify (term T "plus(0,x)") (term T "plus(y,0)");
```

2.1.4 External Solvers, Parallel Search for Orderings

External Solvers Ordering constraints are nowadays usually translated into SAT problems. Thus, the main solving parts are delegated to a SAT solver [12, 21] or an SMT solver [17]. This scheme is implemented in C*i*ME3, and the discovery of interpretations, RPO and AFS, and application of the extended subterm criterion amount to solving SAT problems.

The directive `#Set_sat_solver` *invoc_name* allows one to specify the invocation name of the external SAT solver, which must fulfil the requirements of the SAT format for input and output, and the invocation of which must be of the form: *invoc_name in_file out_file*.

Note that polynomial interpretations and LPO+AFS may still be found without the help of any external solver, using the internal Diophantine constraint solver of C*i*ME [7].

Parallel Search Ordering solvers may be used in parallel. When ordering constraints are generated during termination analysis, C*i*ME3 forks one solver process by ordering constraints set and ordering parameter (Listing 3), thus searching for different orderings in parallel. Note that solving of different ordering constraints is thus also parallelised. The maximum number of processes that can be launched in parallel may be set using the `#Set_nb_proc` directive. If this limit is reached, new forks are put in a queue and wait for previous ones to stop before they can be activated. Default behaviour is sequential computation.

2.2 Local Confluence, Completion and Convergence

A noticeable feature of C*i*ME3 is its ability to check and moreover certify local confluence. Combining this with its possibility to prove and certify termination, one can easily obtain *proof and certification* of convergence. To date and to our knowledge C*i*ME3 is the only tool that can prove and certify convergence of rewrite systems. The proof search of local confluence (and hence of convergence) is obtained by checking joinability of critical pairs. When proving local confluence, C*i*ME3 stores a trace in order to be able to certify it later.

Since critical pairs computation is the core of the standard Knuth-Bendix completion, our implementation of Knuth-Bendix completion (commands `complete` or `prove_goal`) also benefits from the trace production: given two terms which are found equal thanks to ordered completion, C*i*ME3 yields a trace of equality between these terms, and can also produce a Coq certificate [3]. Completion modulo AC, also a part of C*i*ME3, is not instrumented yet.

Certification of convergence proofs is obtained by an application of Newman's lemma, either using a known proof of termination, or assuming termination of the considered system. Convergence is then proved by showing that in particular each critical pair is joinable: we try to normalise each member of the pair and to show that both reduce to the same term.

3 Certification Engine

As proof engines perform proof search, various verbosity levels allow one to control the process. Once a proof is discovered, its description is printed on the standard output. Generation of a CPF trace,[3] or directly of the corresponding Coq script (for termination, local confluence and convergence), is enabled by command or through batch mode (Section 3.3).

The purpose of the certification engine is to take a proof trace as an input, and to output a Coq script for this trace's certification. Traces may come from proofs discovered with C*i*ME3, or with other provers (AProVE, $\mathsf{T}_{\mathsf{T}}\mathsf{T}_2\ldots$) provided they come as CPF files[3].

3.1 Input, Output

The normal input *trace* format for termination proofs is CPF. C*i*ME3 is not yet able to certify all criteria supported by the CPF format, but may generate scripts for all criteria it uses in proof discovery. For properties other than termination (including termination for convenience), one can take as an input a *problem* instead of a proof trace. In this case all the problem formats described in Section 2.1 are supported, C*i*ME will search the proof by itself and will generate the script, without any intermediate trace. See Listing 7.

The techniques for the generation of Coq scripts for certification have been described in previous works [4, 6, 8]. The compilation of those scripts relies on the Coccinelle library, which allows for deep and shallow embeddings of the theory of rewriting. Relying on a deep embedding to reuse generic theorems instead of reproving them for each instances is usually faster. However it is interesting to notice that C*i*ME does use both embeddings depending on the proofs performed. We claim in particular that avoiding an actual deep representation of the dependency graph is very efficient from a computational point of view, as presented in [8]. Listing 6 illustrates the fact that the formalisation is partly shallow: on the one hand, symbols and rewriting rules are defined by new inductive types (`symb` and `R_rules`), on the other hand, term structure and rewriting relations rely on generic deep definitions

[3] As CPF is not yet extended to handle confluence proofs, hence confluence and convergence proofs output Coq scripts instead of CPF.

Listing 6 Coq script structure (notations are simplified).

```
1  Require Import equational_theory...          (* Preamble: require coccinelle files. *)
2  Module algebra.
3    Module F <:term_spec.Signature.                          (* Signature definition *)
4      Inductive symb:Set := plus: symb | S: symb | O: symb.          (* Symbols *)
5      ...
6    End F.
7    Module Alg := term.Make(F)(IntVars)          (* Algebra by functor instantiation *)
8  End Algebra.
9
10 Inductive R_rules: term → term → Prop :=              (* Definition of the TRS *)
11 |R_rule0: R_rules (Var 1) (Term plus [Term O [];Var 1])          (* plus(O,x)→x *)
12 |R_rule1: R_rules (Term S [Term plus [Var 1;Var 2]])   (* plus(S(x),y)→S(plus(x,y)) *)
13                   (Term plus [Term S [Var 1];Var 2]).
14
15 Module WF_R.
16   ...                                          (* Criteria application proofs *)
17   Lemma wf:  well_founded (one_step R_rules).          (* Main termination lemma *)
18     ...
19   Qed.
20 End WF_R.
21 Module Confluence := Newman.Confluence(...).
22 ...
23 Lemma confluence: ∀ x, Newman.confluence _ (one_step R_rules) x.
24   ...                                          (* Joinability of critical pairs.  *)
25 Qed.
```

(`Term`, `Var` and `one_step`). However, depending on the content of the proof, a deep version of `R_rules`, automatically proved equivalent, is defined when needed.

3.2 Structure of a Proof

Listing 6 presents an excerpt of the proof of convergence for the example in Listing 1. It displays in particular the general structure of a proof script for certification using the A3PAT approach. Symbols (line 4) and algebra (line 7) are defined at the beginning of the file, followed by the definition of the system itself (line 10).[4] The main lemma for termination is at the end of the last termination related module: `WF_R`. The proofs related to the instances of the different criteria mentioned in the input trace are formalised and proved in a (nested) sequence of modules (from line 16), thus mirroring roughly the structure of proof tree described in [4].

Eventually, the confluence proof may be generated with or without an actual proof of termination (in the latter case, the confluence is proved *on the assumption* that the system terminates). As termination is shown in our example, there is no assumption in Listing 6.

To compile the script, and thus to ensure its validity, one must have the Coccinelle library installed, and a shell variable `COCCINELLE` set to its root directory. The last lines of the generated Coq file contains the `coqc` command line to run.

[4] Notice that `R_rules t u` means that `u` rewrites to `t`.

3.3 Batch Mode Command Line Options

In batch mode C*i*ME takes an input file in a specified format, and returns a result file in a specified output format. The whole process may include proof searches and/or generation of Coq scripts. Options `-term`/`-noterm` and `-confl`/`-noconfl` specify which properties must be considered or not. Input options include: `-itrs` (`-isrs`) for `TRS` (`SRS`) (non XML) formats, `-itptp` for TPTP format,[5] `-icime` for C*i*ME language, and `-icpf` for CPF format. Similarly, output options include: `-ocpf`, `-ocoq`, and `-ocime` (outputs the global environment in C*i*ME format).

Listing 7 Sample command lines.

```
1 cime -itrs foo.trs -term -ocpf foo.cpf
2 cime -icpf foo.cpf -ocoq foo.v
3 cime -itrs foo.trs -term -ocoq foo.v
4 cime -itrs foo.trs -term -confl -ocoq foo.v
5 cime -icime preamble.cim3 -itrs foo.trs -term -confl -ocoq foo.v
6 cime -icime preamble.cim3 -noterm -confl  -itrs foo.trs -ocoq foo.v
```

Line 1 asks for a termination proof for the system in file `foo.trs` and then generates its CPF trace into `foo.cpf`. Line 2 generates the Coq script `foo.v` to certify CPF trace `foo.cpf`. Line 3 is equivalent to line 1 followed by line 2 (no CPF file is generated). Line 4 additionally generates the Coq proof of confluence. Line 5 is similar to 4 but loads a preamble in C*i*ME format before proof search, this is a usual way to set parameters in batch mode. Line 6 produces a Coq confluence proof parameterised by a proof of termination (not discovered). All these options may finally be chained in one command line:

```
cime  -noconfl -itrs foo1.trs -ocpf foo1Term.cpf -confl -ocoq foo1TermConf.v\
      -icpf foo2.cpf -ocoq foo2Term.v -noterm -ocoq foo2Conf.v
```

This asks for a termination proof for the system in `foo1.trs`, generates the corresponding CPF file `foo1Term.cpf`, then produces (without any new search) a termination and confluence Coq proof `foo1TermConf.v`. It then generates the Coq script `foo2Term.v` to certify the CPF trace `foo2.cpf`, and build script `foo2Conf.v` to certify the confluence of the TRS in `foo2.cpf`. The last proof is parameterised by a proof of termination (not computed).

3.4 Experiments

The following tables present some experiments run on a 24GB machine equipped with 12 cores and running Linux. The base for termination is the category TRS (taken *raw*) of the TPDB 5.0 consisting of 2506 problems;[6] 11 simultaneous processes are allowed. Local confluence and convergence tests are run on the 1155 problems that are proved terminating within the global time limit, set to 180s. The version of Coq is the 8.3 release.

The vast majority of termination proofs are discovered within 10s, and 30% within $\frac{1}{10}$s. Regarding certification, the compilation of Coq script is slow compared to certification by extracted tools like CeTA. However, 730 termination proofs are certified within 180s, of which 78% within a minute. The only failures observed (out of this run) are due to time or memory limitations. Regarding local confluence and convergence, 310 proofs of convergence are discovered, 306 of which are certified within the time limit. The average compile time for

[5] `www.tptp.org` up to version 4.
[6] In particular strategies are *not* taken into account.

convergence is less than 17s. Over 53% of the convergence proofs are certified within 10s.

Discovery	< 0.1s	< 1s	< 10s
Term. 1155	354	730	1012
Confl. 310	287	310	—

Certification		< 5s	< 10s	< 30s	< 60s
Term.	730	200	293	481	573
Confl.	306	157	234	287	299
Conv.	306	53	164	271	293

It is difficult to compare certifiers from the result of the certified termination competition. Firstly, C*i*ME was the only certifier targeted to Coq participating in 2010. Secondly, the number of proofs that certifiers have validated essentially reflects the certifiers accordance to the provers' proving strategy, as they formalise criteria/orderings that are distinct.

4 Resources and Perspectives

C*i*ME3 is an open source piece of software; it is available under the CeCiLL-C licence on the resources page of the A3PAT project: `http://a3pat.ensiie.fr/pub`. The web site provides ELF executables for 64 bit architectures, a tarball of the sources and installation instructions, as well as a short user-manual (`http://a3pat.ensiie.fr/pub/manual-cime3.html`), and a script tool to ease benches and tests over databases of problems. Note that it is also possible to give C*i*ME3 a try online through a dedicated web interface with a limited choice of options at `http://a3pat.ensiie.fr/online`. The companion library Coccinelle is also available from this page, and requires version `8.3` of the Coq proof assistant.

Perspectives regards proof and certification engines. Some proof discovery techniques implemented in C*i*ME 2 have not been transferred yet, notably termination modulo equational theories and modular techniques, including usable rules refinements.

Regarding certification techniques for termination, a short term perspective is the handling of arctic matrices, min/max polynomials, usable rules, and proofs under strategies, as all the formal material is ready in Coccinelle.

Acknowledgements The authors would like to thank the anonymous referees for their fruitful comments and their help in improving the presentation of this article.

References

1 T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236:133–178, 2000.

2 F. Blanqui, S. Coupet-Grimal, W. Delobel, S. Hinderer, and A. Koprowski. Color, a coq library on rewriting and termination. In A. Geser and H. Sondergaard, ed., *Extended Abstracts of the 8th Int. Workshop on Termination, WST'06*, Aug. 2006.

3 É. Contejean and P. Corbineau. Reflecting proofs in first-order logic with equality. In *20th Int. Conf. on Automated Deduction*, vol. 3632 of *LNAI*, pp. 7–22, Tallinn, Estonia, July 2005. Springer.

4 É. Contejean, P. Courtieu, J. Forest, A. Paskevich, O. Pons, and X. Urbain. A3PAT, an Approach for Certified Automated Termination Proofs. In *ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pp. 63–72. ACM, 2010.

5 É. Contejean, P. Courtieu, J. Forest, O. Pons, and X. Urbain. Certification of Automated Termination Proofs. In B. Konev and F. Wolter, ed., *6th Int. Symp. on Frontiers of Combining Systems*, vol. 4720 of *LNAI*, pp. 148–162, Liverpool, UK, Sept. 2007. Springer.

6 E. Contejean, J. Forest, and X. Urbain. Deep-Embedded Unification. Technical Report 1547, Cédric lab., CNAM Paris, France, 2008.

7 É. Contejean, C. Marché, A. P. Tomás, and X. Urbain. Mechanically proving termination using polynomial interpretations. *Journal of Automated Reasoning*, 34(4):325–363, 2005.

8 P. Courtieu, J. Forest, and X. Urbain. Certifying a Termination Criterion Based on Graphs, Without Graphs. In C. Muñoz and O. Ait Mohamed, ed., *21st Int. Conf. on Theorem Proving in Higher Order Logics*, vol. 5170 of *LNCS*, pp. 183–198, Montréal, Canada, Aug. 2008. Springer.

9 P. Courtieu, G. Gbedo, and O. Pons. Improved Matrix Interpretation. In J. van Leeuwen, A. Muscholl, D. Peleg, J. Pokorný, and B. Rumpe, ed., *36th Conf. on Current Trends in Theory and Practice of Computer Science*, vol. 5901 of *LNCS*, pp. 283–295, Špindlerův Mlýn, Czech Republic, Jan. 2010. Springer.

10 N. Dershowitz. Orderings for term rewriting systems. *Theoretical Computer Science*, 17(3):279–301, Mar. 1982.

11 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.

12 C. Fuhs, A. Middeldorp, P. Schneider-Kamp, and H. Zankl. SAT Solving for Termination Analysis with Polynomial Interpretations. In *SAT 07*, vol. 4501 of *LNCS*, pp. 340–354, May 2007. Springer.

13 J. Giesl, P. Schneider-Kamp, and R. Thiemann. AProVE 1.2: Automatic Termination Proofs in the Dependency Pair Framework. In U. Furbach and N. Shankar, ed., *3rd Int. Joint Conf. on Automated Reasoning*, vol. 4130 of *LNCS*, pp. 281–286, Seattle, USA, Aug. 2006. Springer.

14 J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Mechanizing and Improving Dependency Pairs. *Journal of Automated Reasoning*, 37(3):155–203, 2006.

15 N. Hirokawa and A. Middeldorp. Automating the dependency pair method. *Information and Computation*, 199(1-2):172–199, 2005.

16 N. Hirokawa and A. Middeldorp. Tyrolean Termination Tool: Techniques and Features. *Information and Computation*, 205(4):474–511, 2007.

17 M. Korp, C. Sternagel, H. Zankl, and A. Middeldorp. Tyrolean Termination Tool 2. In R. Treinen, ed., *20th Int. Conf. on Rewriting Techniques and Applications*, vol. 5595 of *LNCS*, pp. 295–304, Brasília, Brazil, July 2009. Springer.

18 D. S. Lankford. On proving term rewriting systems are Noetherian. Technical Report MTP-3, Mathematics Department, Louisiana Tech. Univ., 1979.

19 Z. Manna and S. Ness. On the termination of Markov algorithms. In *3rd Hawaii Int. Conf. on Systems Sciences*, pp. 789–792, Honolulu, USA, 1970.

20 C. Marché and H. Zantema. The Termination Competition. In F. Baader, ed., *18th Int. Conf. on Rewriting Techniques and Applications*, vol. 4533 of *LNCS*, pp. 303–313, Paris, France, June 2007. Springer.

21 P. Schneider-Kamp, R. Thiemann, E. Annov, M. Codish, and J. Giesl. Proving Termination Using Recursive Path Orders and SAT Solving. In B. Konev and F. Wolter, ed., *6th Int. Symp. on Frontiers of Combining Systems*, vol. 4720 of *LNAI*, pp. 267–282, Liverpool, UK, Sept. 2007. Springer.

22 R. Thiemann and C. Sternagel. Certification of Termination Proofs using CeTA. In T. Nipkow and C. Urban, ed., *22nd Int. Conf. on Theorem Proving in Higher Order Logics*, vol. 5674 of *LNCS*, pp. 452–468, Munich, Germany, Aug. 2009. Springer.

Variants, Unification, Narrowing, and Symbolic Reachability in Maude 2.6*

Francisco Durán[1], Steven Eker[2], Santiago Escobar[3], José Meseguer[4], and Carolyn Talcott[2]

1 **Universidad de Málaga, Spain.** duran@lcc.uma.es
2 **SRI International, CA, USA.** eker@csl.sri.com,clt@cs.stanford.edu
3 **Universidad Politécnica de Valencia, Spain.** sescobar@dsic.upv.es
4 **University of Illinois at Urbana-Champaign, IL, USA.** meseguer@illinois.edu

Abstract

This paper introduces some novel features of Maude 2.6 focusing on the *variants* of a term. Given an equational theory $(\Sigma, Ax \cup E)$, the E, Ax-variants of a term t are understood as the set of all pairs consisting of a substitution σ and the E, Ax-canonical form of $t\sigma$. The equational theory $(Ax \cup E)$ has the *finite variant property* iff there is a finite set of most general variants. We have added support in Maude 2.6 for: (i) order-sorted unification modulo associativity, commutativity and identity, (ii) variant generation, (iii) order-sorted unification modulo finite variant theories, and (iv) narrowing-based symbolic reachability modulo finite variant theories. We also explain how these features have a number of interesting applications in areas such as unification theory, cryptographic protocol verification, business processes, and proofs of termination, confluence and coherence.

1998 ACM Subject Classification D.2.4 [Software Engineering]: Software/Program Verification, D.3.2 [Programming Languages]: Language Classifications, F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs

Keywords and phrases Rewriting logic, narrowing, unification, variants

Digital Object Identifier 10.4230/LIPIcs.RTA.2011.31

Category System Description

1 Introduction

In [4] the Maude 2.4 features for order-sorted unification modulo axioms Ax, including commutativity (C) and associativity commutativity (AC), and for narrowing-based reachability analysis of rewrite theories modulo such axioms Ax were described. In this paper we present the new features of variant-generation, variant-based unification, and symbolic reachability analysis modulo a theory with the finite variant property supported by Maude 2.6. The key distinction, now supported for the first time in Maude, is one between *dedicated* unification algorithms for a limited set of axioms Ax, and *generic* unification algorithms such as variant-based unification which can be applied to a much wider range of user-definable theories. As explained in Section 6, this opens up many applications, including: (i) unification-related applications; (ii) cryptographic protocol analysis; (iii) symbolic reachability analysis of concurrent systems; and (iv) formal reasoning capabilities such as

* S. Escobar has been supported by MICINN grant TIN 2010-21062-C02-02. J. Meseguer has been supported by NSF grant CCF 09-05584. C. Talcott and S. Eker have been supported by NSF grant 09-05607. F. Durán was supported by MICINN grant TIN2008-03107 and JA grant P07-TIC-03184.

22nd International Conference on Rewriting Techniques and Applications (RTA'11).
Editor: M. Schmidt-Schauß; pp. 31–40

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

termination proofs, and proofs of local confluence and coherence that can now be performed *modulo* a much wider set of equational theories thanks to the use of variants.

Comon-Lundh and Delaune's notion of *variant* [7] characterizes the instances of a term w.r.t. an equational theory $E \cup Ax$ such that the equations E are covergent and coherent modulo axioms Ax. The E,Ax-*variants* of a term t are *pairs* (t', θ), with θ a substitution and t' the E,Ax-canonical form of $t\theta$. A preorder relation of generalization that holds between such pairs provides a notion of most general variants and also of completeness of a set of variants. An equational theory $E \cup Ax$ has the *finite variant property* iff there is a finite complete set of most general variants for each term. This property also ensures the existence of a generic *finitary* $E \cup Ax$-unification algorithm based on computing variants. Such generic unification algorithm involves performing Ax-unification using a dedicated algorithm and computing the E,Ax-variants.

As we explain in Section 2, the base of axioms now supported by Maude with a dedicated unification algorithm has been extended to include associative-commutative with identity (ACU) function symbols, in combination with the previously supported C, AC, and free ($\emptyset$) function symbols. On this extended axiom base, *Full Maude 2.6*, an extension of Maude written in Maude itself by taking advantage of its reflective capabilities, now offers the following new features for user-definable order-sorted theories $E \cup Ax$ with the finite variant property and satisfying some simple requirements: (i) *variant generation*, that is, computing the most general E,Ax-variants of a term (Section 3); (ii) *variant-based order-sorted unification modulo* $E \cup Ax$ (Section 4); and (iii) *narrowing-based symbolic reachability analysis* of a concurrent system whose equational subtheory satisfies the finite variant property (Section 5). There are several programming languages based on narrowing but none supporting narrowing modulo finite variant theories.

2 Implementation of Order-Sorted ACU Unification

The addition of ACU to the theories handled by the dedicated unification algorithm in Maude required substantial changes to the unification infrastructure implemented in previous versions of Maude for C and AC theories because of the problems associated with collapse theories. In this section we give an overview of the techniques used and highlight a novel algorithm for selecting sets of Diophantine basis elements during the computation of ACU unifiers.

Combining Unification Algorithms. The basic approach to solving unification problems where function symbols are drawn from more than one theory is variable abstraction where *alien subterms*, i.e., subterms headed by a symbol from a theory different from that of the top symbol of the parent term, are replaced by fresh variables to form *pure* unification subproblems which only involve variables and functions symbols from a single theory and which can be passed to a unification algorithm for such a theory. Proving termination of combinations of algorithms is nontrivial, as variables are necessarily shared between theories and the unification of variables in one theory can create new unification subproblems in another theory, potentially ad infinitum. Stickel's algorithm [22], which combined the AC and free theories, required an elaborate termination proof by Fages [15]. Boudet et al. [2] proposed a much simpler approach where all unification subproblems and variable bindings in a given theory are solved (and re-solved if another subproblem in that theory is created) simultaneously. This method requires a simultaneous E-unification algorithm for each theory E and was the method implemented in Maude for C, AC, and $\emptyset$ prior to the addition of ACU.

Collapse theories add two major complications to the combination of unification algorithms. Firstly, theory clashes where two terms with top symbols from different theories are required to unify, can no longer be treated as a failure, since if one or other top symbol belongs to a collapse theory, a collapse may occur, yielding solutions. Secondly, *compound cycles*, that is, problems of the form $x_1 =^? t_1(\ldots,x_2,\ldots), x_2 =^? t_2(\ldots,x_3,\ldots), \ldots, x_n =^? t_n(\ldots,x_1,\ldots)$ where the terms t_i are pure in

different theories, can no longer be treated as failure, since solutions may be possible via collapse.

Several authors have proposed combination schemes that can handle collapse theories. We use a simplified version of an algorithm due to Boudet [1]. The original algorithm also handles nonregular theories but we omit that capability to simplify the implementation. The key idea is that each theory E needs a restricted simultaneous E-unification algorithm which solves the simultaneous unification problem for pure equations that are pure in E but where certain variables may be marked as only being allowed to unify with other variables. A theory clash subproblem $f(\ldots) =^? g(\ldots)$, is split into a disjunction of two subproblems each of which is a conjunction $x =^? f(\ldots) \wedge x =^? g(\ldots)$ where x is a fresh variable. In one subproblem x is marked in the f equation and in the other subproblem x is marked in the g equation; either or both branches of the search may return solutions. Restricted unification is also used to break compound cycles. Because we do not handle nonregular theories, Boudet-style variable-elimination algorithms are unnecessary.

Boudet's algorithm assumes that theories are disjoint; i.e., that they do not share function symbols. Because in Maude this is not quite true – identities can contain symbols from other theories – we need to handle a special kind of variable elimination. We illustrate the issue with the following example:

```
fmod CYCLE is sort S . vars X Y : S . ops a b c d : -> S .
  op f : S S -> S [assoc comm id: g(c, d)] .
  op g : S S -> S [assoc comm id: f(a, b)] .
endfm
Maude> unify X =? f(Y, a, b) /\ Y =? g(X, c, d) .
```

Here the unification problem would already be in solved form but for the compound cycle formed by the X and Y variables. Restricted unification cannot break this cycle, since neither of the right-hand sides can collapse out of their theory. However, putting $Y = g(c,d)$ eliminates Y from the first equation yielding $X = f(a,b)$ which eliminates X from the second equation, yielding a solution. This situation is somewhat pathological in Maude programs, and we do not really care about performance in its handling. Maude handles it by looking for this kind of cyclic dependency between theories when the signature is preprocessed and setting a flag so that a brute force variable elimination algorithm will be used to try and break compound cycles at unification time.

Diophantine Basis Element Selection. We solve restricted simultaneous ACU unification using an extension of the simultaneous AC unification algorithm in [2]. For an ACU function symbol f we are presented with a set of flattened pure equations that take the form $f(x_1^{p_1},\ldots,x_n^{p_n}) =^? f(y_1^{q_1},\ldots,y_m^{q_m})$ or $x_1 =^? f(y_1^{q_1},\ldots,y_m^{q_m})$. Each f-equation yields a Diophantine equation $p_1X_1 + \cdots + p_nX_n = q_1Y_1 + \cdots + q_mY_m$ or respectively, $X_1 = q_1Y_1 + \cdots + q_mY_m$ where the X_i's and Y_i's are non-negative Diophantine variables. If an original variable is marked in some equation, the corresponding Diophantine variable receives an upper-bound of 1. Also, we may be able to obtain an upper-bound from order-sorting information, using the signature analysis technique in [12].

The general solution to a set of non-negative Diophantine equations is a set of basis elements from which all solutions can be obtained by linear combination. Upper-bound information may trivially eliminate some basis elements from consideration and can be used by the Diophantine solver to terminate the search for basis elements early.

A fresh variable z_k is allocated for each basis element α_k and unifiers are formed by finding sets of basis elements that satisfy certain properties and constructing assignments $x_i \leftarrow f(\ldots,z_k^{\alpha_{k,i}},\ldots)$ where k ranges over the indices of the selected basis elements and $\alpha_{k,i}$ is the value of X_i in the basis element α_k.

The criteria for choosing the sets of basis elements is the key difference between AC unification, ACU unification, and restricted ACU unification. With AC unification, every selection of basis elements whose sum yields a nonzero value for each X_i and Y_i must be considered. With ACU unification that requirement is lifted because of the availability of an identity element. The identity

element also means that any assignment including basis element α_k generalizes the same assignment with α_k removed by assigning the identity element to z_k and thus there is a single most general solution, formed by selecting all the basis elements.

In the case of restricted ACU unification, we may have upper-bounds on variables because they are marked. In Maude, order-sorted considerations may place upper-bounds on variables, and may also place a lower-bound of 1 on variables where the corresponding original variable has a sort that cannot take the identity element. In order to find a complete set of unifiers we need to find all maximal sets of basis elements whose sum satisfies the upper and lower-bounds on the variables.

Several explicit schemes for searching the subsets of basis elements were tried but the search was typically the dominant cost for ACU unification, often rendering the solution of quite modest unification problems impractical. In the current implementation this search is performed symbolically using a Binary Decision Diagram (BDD) [3] based algorithm. A BDD variable is allocated for each basis element, whose value, true or false, denotes whether the basis element is included in the subset. A BDD, called *legal*, is constructed, which evaluates to true on exactly those valuations that correspond to selections of basis elements that satisfy the upper- and lower-bound constraints on each Diophantine variable. Enforcement of the upper-bounds on the sum is done using dynamic programming and the BDD *ite* operation. Using the BDD *legal*, a second BDD, called *maximal*, is constructed which is true on exactly those valuations where *legal* is true, and changing a false into a true makes *legal* false. These valuations of the BDD variables and thus the subsets of basis elements they encode are then recovered by tracing the paths from the root to the true terminal in *maximal*. This method yielded a dramatic speed up (from hours to milliseconds) on problems of useful size.

Admissible Equational Theories. Maude 2.6 currently provides a built-in order-sorted Ax-unification algorithm for all order-sorted theories (Σ, Ax) such that:

- the order-sorted signature Σ is preregular modulo Ax (see [5, Section 3.8]);
- the axioms Ax associated to function symbols are as follows:
  - there can be arbitrary function symbols and constants with no equational attributes;
  - the `iter` equational attribute[1] can be declared for some unary symbols;
  - the `comm` or `assoc comm` or `assoc comm id:` attributes[2] can be declared for some binary function symbols, but then no other equational attributes must be given for such symbols.

Explicitly excluded are theories with binary function symbols having any combination of: (i) the `idem` attribute[3]; (ii) the `id:`, `left id:`, or `right id:` attributes without `assoc comm`; or (iii) the `assoc` attribute without `comm`.

3 Variants and Variant Generation

Variant generation for an equational theory $(\Sigma, E \cup Ax)$ is defined modulo Ax using the order-sorted Ax-unification procedure described in Section 2.

The equational theories that are admissible for variant generation are as follows. Let `fmod` $(\Sigma, Ax \cup E)$ `endfm` be an order-sorted functional module where E is a set of equations specified with the `eq` keyword, and Ax is a set of axioms such that (Σ, Ax) satisfies the restrictions of Section 2. Furthermore, the equations E must satisfy the following extra conditions:

- The equations E are unconditional and convergent, sort-decreasing and coherent modulo Ax.
- An equation's left-hand side cannot be a variable, and the `owise` feature is not allowed.

[1] Maude provides a built-in mechanism called the `iter` (short for iterated operator) theory whose goal is to permit the efficient input, output, and manipulation of very large stacks of a unary operator. See [6] for additional details.
[2] The operator attribute `assoc` stands for associativity, `comm` for commutativity and `id:` for identity.
[3] The operator attribute `idem` stands for idempotency.

- All equations must be *variant-preserving* [14], i.e., if two left-hand sides of E (possibly renamed) overlap — i.e., there is a substitution θ s.t. $(l_1\theta)|_p =_{Ax} l_2\theta$, where p can be a variable or non-variable position of l_1 — then either:
  1. $l_1\theta$ does not have a pattern modulo Ax, i.e., for every term u s.t. $u =_{Ax} l_1\theta$, u is reducible in E modulo Ax below the root position, or
  2. $l_1\theta$ has a pattern modulo Ax, i.e., there is a term u s.t. $u =_{Ax} l_1\theta$ and u is reducible in E modulo Ax only at the root position, but then the matching substitution is E,Ax-irreducible.

  Variant-preservingness is necessary for an eager generation of variants; see [5] for details.
- An equation's right-hand side must be a *strongly irreducible term*, i.e., for any E,Ax-normalized substitution σ, the term $t\sigma$ is E,Ax-irreducible. A term containing only variables and non-defined (constructor) symbols is strongly irreducible.

The above conditions ensure that $(\Sigma, E \cup Ax)$ has the finite variant property. We refer the reader to [14] for a detail explanation of variants and variant generation as well as for automated methods for ensuring the finite variant property. Any rewrite theory `mod` $(\Sigma, Ax \cup E \cup G, R)$ `endm` where G is an additional set of equations is also considered admissible for variant generation if the equational part $(\Sigma, Ax \cup E)$ satisfies the conditions described above. Note that when an equational theory $(\Sigma, Ax \cup E \cup G)$ is provided to Full Maude, each equation in E (used for variant computation) must include the `variant` attribute.

Given a module *ModId*, Full Maude provides a variant generation command of the form:

`(get variants [ in` *ModId* `: ]` t `.)`

ACU-Coherence Completion. The convergence and sort-decreasingness of equational Maude specifications can be checked using Maude's Church-Rosser Checker (CRC) [10] and Termination Checker (MTT) [8]. For theories Ax that are combinations of associativity, commutativity, and identity axioms, we can make any specification Ax-coherent by using a procedure which adds Ax-extensions and always terminates (see [20], and [6, Section 4.8] for a more informal explanation).

The user modules are automatically completed for Ax-coherence when used for variant generation and variant-based unification (Section 4) and narrowing (Section 5). The user can access these automatically completed user modules by invoking the command

`(acu coherence completion [ <`*module-expr.*`> ] .)`

where <*module-expr.*> is any module expression. If no module expression is given the default current module is completed.

A corresponding `acuCohComplete` function is available at the metalevel of Maude.

```
op acuCohComplete : Module -> Module .
```

If no module expression is given, the default current module is used.

A Motivating Example. Consider, for example, the following Petri-net-like specification of a vending machine to buy apples (`a`) or cakes (`c`) with dollars (`$`) and/or quaters (`q`):

```
(mod VENDING-MACHINE is
   sorts Coin Item Marking Money State . subsort Money Item < Marking .
   op empty : -> Money . op <_> : Marking -> State . subsort Coin < Money .
   op __ : Money Money -> Money [assoc comm id: empty] .
   op __ : Marking Marking -> Marking [assoc comm id: empty] .
   ops $ q : -> Coin . ops a c : -> Item . var M : Marking .
   rl [buy-c] : < M $ > => < M c > .
   rl [buy-a] : < M $ > => < M a q > .
   eq [change]: q q q q = $ [variant] .
endm)
```

The equational theory underlying this rewrite theory contains two subsort-overloaded ACU symbols and an equation `q q q q = $` (for variant computation). Note that the module is not ACU-coherent. It is automatically completed for coherence modulo ACU by replacing the `change` equation by the

equation "`eq [change-Ext]: M q q q q = M $ [variant] .`". Note also that this equation satisfies all the conditions above for admissible theories, especially strongly right irreducibility and variant preservingness. We can get variants of a term as follows.

```
Maude> (get variants in VENDING-MACHINE : < $ q q X:Marking > .)
Variant 1
< $ q q X:Marking >, empty substitution
Variant 2
< $ $ #5:Marking >, X:Marking --> q q #5:Marking
```

These two variants represent a finite, complete, and maximal set of variants for the given term. For instance, the variant

```
{< $ $ q q Y:Marking >, X:Marking --> q q q q Y:Marking}
```

is an instance of the first variant above, i.e., the canonical form `< $ $ q q Y:Marking >` is an instance of the normal form `< $ q q X:Marking >` of the first variant, and the (normalized version) of the instantiating substitution (`X:Marking --> $ Y:Marking`) is an instance of the empty substitution of the first variant. Note that this variant is not an instance of the second variant above because the substitution `X:Marking --> q q q q Y:Marking` is normalized before comparing it with the substitution `X:Marking --> q q #5:Marking` of the second variant above.

The procedure for variant generation is also available at the metalevel of Maude thanks to the `getVariants` function.

```
op getVariants : Module Term -> VariantFourSet .
```

Handling of Other Axioms. Variant generation and variant-based unification (Section 4) and narrowing (Section 5) have also been extended to deal with *any* combination of associativity and/or commutativity and/or identity axioms except associativity without commutativity. The general idea, borrowed from [9], is to replace a specification $(\Sigma, (Ax \cup Id) \cup E)$ where Ax contains C, AC, or ACU axioms and $Id : maude - rta11.tex, v1.52011/04/0512 : 19 : 20 schaussExp$ contains all other identity axioms, by a *semantically equivalent* specification $(\Sigma, Ax \cup (\vec{Id} \cup \widehat{E}))$, where the $Id : maude - rta11.tex, v1.52011/04/0512 : 19 : 20 schaussExp$ axioms have been oriented as rules, and the equations $\widehat{E}$ are the $\vec{Id}, Ax$-*variants* of the original equations E.

A command is available in Full Maude of the form:

```
(remove id attributes [ <module-expr.> ] .)
```

It shows the specified module with the identity attributes (`id`, `right id`, and `left id`) transformed into rules and the equations $\widehat{E}$ obtained using $\vec{Id}, Ax$-variants.

A corresponding function `removeIds` is available at the metalevel of Maude.

```
op removeIds : Module -> Module .
```

If no module expression is given, the default current module is used.

4 Variant-based Equational Order-Sorted Unification

The intimate connection between E, Ax-variants and $E \cup Ax$-unification is as follows. Suppose that we extend the equational theory $(\Sigma, E \cup Ax)$ to $(\widehat{\Sigma}, \widehat{E} \cup Ax)$ by adding to Σ a new sort Truth, not related to any sort in Σ, with a constant `tt`, and for each top sort [s] of each connected component s, an operator `eq` : [s] $\times$ [s] $\rightarrow$ Truth; and where $\widehat{E}$ extendes E by adding for each top sort [s] and x of sort [s] an extra rule $eq(x,x) \rightarrow tt$. Then, given any two terms t, t', if θ is a (E, Ax)-unifier of t and t', then the E, Ax-canonical forms of $t\theta$ and $t'\theta$ must be Ax-equal and therefore the pair (tt, θ) must be a variant of the term $eq(t,t')$, i.e., $eq(t,t')\theta \rightarrow^{!} tt$. Furthermore, if the term $eq(t,t')$ has a finite set of most general variants, then we are *guaranteed* that the set of most general (E, Ax)-unifiers of t and t' is *finite* and subsumes (tt, θ).

Given a module *ModId* of the general form `mod` $(\Sigma, Ax \cup E \cup G, R)$ `endm` where $(\Sigma, Ax \cup E)$ satisfies the requirements of Section 3, Full Maude provides a command for $E \cup Ax$-equational unification

based on variant generation of the form:

(`variant unify` [`in` *ModId* `:`] t `=?` t' `.`)

Consider again the vending machine. We can ask whether there is an $E \cup Ax$-equational unifier of two configurations, one containing a dollar and two quarters and another containing two quarters:

```
Maude> (variant unify in VENDING-MACHINE : < q q X:Marking > =? < $ Y:Marking > .)
Solution 1
X:Marking --> q q Y:Marking
Solution 2
X:Marking --> $ #12:Marking ; Y:Marking --> q q #12:Marking
```

There are no more general unifiers. For instance, `X:Marking --> q q`, `Y:Marking --> empty` is an instance of the first solution by using the identity property of the operator for markings.

The procedure for variant-based equational unification is also available at the metalevel thanks to the `metaVariantUnify` function.

```
op metaVariantUnify : Module Term Term -> SubstitutionSet .
```

A useful special case of the variant-based equational unification feature is that of Ax'-unification for theories (Σ, Ax') where: (i) $Ax' = Ax \cup Ids$, (ii) (Σ, Ax) satisfies the requirements in Section 3, and (iii) Ids is a collection of `id:`, `left id:`, `right id:` axioms. This case is handled by invoking the `variant unify` command directly on (Σ, Ax'), since Full Maude first invokes the `remove id attributes` transformation command described in Section 3.

5 Narrowing-based Symbolic Reachability Analysis

Narrowing [16] generalizes term rewriting by allowing free variables in terms and by performing unification instead of matching. Likewise, narrowing *modulo* $Ax \cup E$ [18] generalizes rewriting with rules R modulo $Ax \cup E$. Given an order-sorted rewrite theory $(\Sigma, Ax \cup E, R)$, where R is a set of unconditional rewrite rules such that the lefthand sides are non-variable terms and the rules are explicitly $Ax \cup E$-coherent [19], and $(\Sigma, Ax \cup E)$ is an equational theory such that a finitary $Ax \cup E$-unification procedure is available, the $(R, Ax \cup E)$-*narrowing* relation is defined as $t \leadsto_{\sigma,p,R,Ax\cup E} t'$ iff there is a non-variable position p of t, a (possibly renamed) rule $l \to r$ in R, and a unifier $\sigma \in Unif_{Ax\cup E}(t|_p, l)$ such that $t' = \sigma(t[r]_p)$.

The classical application of $(R, Ax \cup E)$-narrowing is to perform $R \cup Ax \cup E$-*unification* when the rules R are understood as *equations*. Indeed the variant-based equational order-sorted unification algorithm of Section 4 is based on an E, Ax-narrowing strategy, called *folding variant narrowing* [14], that terminates when $E \cup Ax$ has the finite variant property [7], even though full E, Ax-narrowing typically does not terminate when Ax contains AC axioms (see [7, 14]).

Instead, when the rules R are understood as *transition rules*, a completely different application of $R, Ax \cup E$-narrowing is that of *symbolic reachability analysis* [19]. Specifically, we consider transition systems specified by order-sorted rewrite theories of the form `mod` $(\Sigma, Ax \cup E, R)$ `endm` where: (i) $E \cup Ax$ satisfies the requirements of Section 3, and (ii) the transition rules R are $E \cup Ax$-coherent and *topmost* (so that rewriting is always done at the top of the term). Then, narrowing modulo $E \cup Ax$ is a *complete* deductive method [19] for symbolic reachability analysis, that is, for solving existential queries of the form $\exists \overline{x} : t(\overline{x}) \to^* t'(\overline{x})$ in the sense that the formula holds for $(\Sigma, Ax \cup E, R)$ iff there is a narrowing sequence $t \leadsto^*_{R,E\cup Ax} u$ such that u and t' have a $E \cup Ax$-unifier.

This symbolic reachability analysis is supported by Full Maude's `search` command, which has the form:

(`search` [`[`n`,`m`]`] [`in` *ModId* `:`] t_1 *SearchArrow* t_2 `.`)

where: n and m are optional arguments providing, respectively, a bound on the number of desired solutions and the maximum depth of the search; *ModId* is the module where the search takes place; t_1 is the starting *non-variable term*, which may contain variables; t_2 is the term specifying the pattern

that has to be reached, with variables, some of which possibly shared with t_1; and *SearchArrow* is an arrow indicating the form of the narrowing proof from t_1 until t_2, where `~>1` indicates a narrowing proof consisting of exactly one step; `~>+` indicates a proof of one or more steps; `~>*` indicates a proof of none, one, or more steps; and `~>!` indicates that the reached term cannot be further narrowed. This narrowing-based search command was already introduced in [4] but now can be performed modulo theories with the finite variant property.

Consider again the vending machine of Section 3. We can use the narrowing search command to answer the question: *Is there any combination of one or more coins that returns exactly an apple and a cake?* This can be done by searching for states that are reachable from a term `< M:Money >` and match the desired pattern at the end.

```
Maude> (search [1] in VENDING-MACHINE : < M:Money > ~>* < a c > .)
Solution 1
M:Money --> $ q q q
```

Note that we must restrict the search to just one solution, because narrowing does not terminate for this reachability problem even though the above solution is indeed the *only* solution.

Narrowing-based reachability analysis is also available at the metalevel by using the following `metaNarrowSearch` function.

```
op metaNarrowSearch :
  Module Term Term Substitution Qid Bound Bound Bound -> ResultTripleSet .
```

If a non-identity substitution is provided in the fourth argument, then any computed substitution must be an instance of the provided one, i.e., we can restrict the computed narrowing sequences to some concrete shape. The `Qid` metarepresents the appropriate search arrow, similar to the `metaSearch` command (see [5, Section 11.4.6]). For the bounds, the first one is the number of computed solutions, the second one is the maximum length of the narrowing sequences, i.e., the depth of the narrowing tree, and the third one is the chosen solution (in order to provide all solutions in a sequential way, as many meta-level commands in Maude do).

Full Maude's `search` command also supports a more general form of symbolic reachability analysis that uses narrowing *with simplification*. We can allow more general rewrite theories of the form `mod` $(\Sigma, Ax \cup E \cup G, R)$ `endm` where: (i) $E \cup Ax$ satisfies the requirements of Section 3, (ii) G is an additional set of equations, and (iii) the rules R are $E \cup Ax \cup G$-coherent and topmost. The remaining equations G are now used in the combined relation $\leadsto_{R,E\cup Ax}; \rightarrow^{!}_{E\cup G,Ax}$. Note that this combined relation may be incomplete, i.e., given a reachability problem of the form $\exists \overline{x} : t(\overline{x}) \rightarrow^{*} t'(\overline{x})$ and a solution σ (i.e., $\sigma(t) \rightarrow^{*}_{R,E\cup Ax\cup G} \sigma(t')$), the relation $\leadsto_{R,E\cup Ax}; \rightarrow^{!}_{E\cup G,Ax}$ may not be able to find a solution more general than σ.

6 Applications

The key usefulness of the new `variant unify` feature is to *greatly extend the range of theories* for which a unification algorithm can be provided by Maude. The key distinction is one between *dedicated* algorithms for a given theory, and *generic* algorithms such as folding variant narrowing which can be applied to a wide range of user-defined theories. As explained in this paper, Maude 2.6 has a dedicated algorithm supporting order-sorted unification modulo axioms Ax which may contain C, AC, and ACU axioms. The `variant unify` feature then allows us to *automatically* derive a finitary unification algorithm for any theory $E \cup Ax$ such that satisfies the requirements in Section 3 and therefore enjoys the finite variant property. In particular, as shown in [7], a good number of cryptographic theories of practical interest satisfy the finite variant property modulo axioms such as AC or $\emptyset$.

Support for variant-based unification can therefore be exploited by cryptographic protocol analysis tools performing symbolic reachability analysis. Such protocols can be modeled as rewrite

theories $(\Sigma, E \cup Ax, R)$, where the algebraic properties of the cryptographic functions are specified by equations $E \cup Ax$, and the protocol's transition rules are specified by the rewrite rules R. Thus the narrowing search feature modulo a theory $E \cup Ax$ satisfying the finite variant property is a feature which, by being available also at the metalevel, can be the basis of a protocol analysis tool performing reachability analysis for protocol verification. This is exactly the approach that has been followed for analyzing cryptographic protocols modulo algebraic properties in the Maude-NPA tool [13, 21], which has been able to analyze a substantial collection of cryptographic protocols modulo their algebraic properties. With the `metaNarrowSearch` operator, this same functionality becomes now available to other protocol analysis tools. As an example, business processes can be similarly analyzed to check for violations. The Document Logic Analysis tool [17] represents document processing protocols as theories in rewriting logic and uses symbolic reachability analysis in Maude to look for forgeries and invalid signatures.

The usefulness of variants and variant generation goes beyond the availability of finitary unification algorithms and symbolic reachability analysis for cryptographic protocols and for other concurrent systems. As demonstrated by its recent applications to termination algorithms modulo axioms in [9], and to algorithms for checking confluence and coherence of rewrite theories modulo axioms, such as those used in the most recent Maude CRC and ChC tools [10, 11], computing the E,Ax-*variants* of a term may be just as important as computing $E \cup Ax$-unifiers. The key idea is the following. Suppose that R is a collection of rewrite rules modulo axioms Ax for which we want to prove, say, termination, or confluence. We may not have any tools for checking such properties that can work modulo the given set of axioms Ax. However, we can *decompose* Ax as a disjoint union $E \cup Ax'$, where E is convergent, sort-decreasing and coherent modulo Ax', and where we have methods to prove, e.g., termination or confluence modulo Ax'. As shown in [9], we can transform R,Ax into a semantically equivalent theory $\widehat{R} \cup E, Ax'$, where $\widehat{R}$ specializes each rule in R to the family of E,Ax'-variants of their lefthand sides. If $E \cup Ax'$ has the finite variant property, we are sure that $\widehat{R}$ will be a finite set; but in practice $\widehat{R}$ can often be finite without such a property. For example, Ax can be the theory A of associativity, for which unification is not even finitary, yet in an order-sorted setting A can often be added as a rule so that $\widehat{R}$ is finite in practice. We refer to [9, 10, 11] for details.

Acknowledgements. We are very thankful to the other members of the Maude team, namely, Manuel Clavel, Patrick Lincoln, and Narciso Martí-Oliet, with whom we have designed and built the Maude language and system.

References

1 A. Boudet. Unification in a combination of equational theories: an efficient algorithm. In *Proceedings of the tenth international conference on Automated deduction*, CADE-10, pages 292–307. Springer-Verlag, 1990.

2 A. Boudet, E. Contejean, and H. Devie. A new AC-unification algorithm with a new algorithm for solving Diophantine equations. In *Proceedings of the 5th IEEE Symposium on Logic in Computer Science*, pages 289–299. IEEE Computer Society Press, 1990.

3 R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 35:677–691, 1986.

4 M. Clavel, F. Durán, S. Eker, S. Escobar, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott. Unification and narrowing in Maude 2.4. In R. Treinen, editor, *Rewriting Techniques and Applications, 20th International Conference, RTA 2009, Brasília, Brazil, June 29 - July 1, 2009, Proceedings*, volume 5595 of *Lecture Notes in Computer Science*, pages 380–390. Springer, 2009.

5 M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. Maude Manual (Version 2.6). March 2011, `http://maude.cs.uiuc.edu`.

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: How to Specify, Program, and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

7 H. Comon-Lundh and S. Delaune. The finite variant property: How to get rid of some algebraic properties. In J. Giesl, editor, *Term Rewriting and Applications, 16th International Conference, RTA 2005, Nara, Japan, April 19-21, 2005, Proceedings*, volume 3467 of *Lecture Notes in Computer Science*, pages 294–307. Springer, 2005.

8 F. Durán, S. Lucas, and J. Meseguer. MTT: The Maude termination tool (system description). In A. Armando, P. Baumgartner, and G. Dowek, editors, *Automated Reasoning 4th International Joint Conference, IJCAR 2008 Sydney, Australia, August 12-15, 2008 Proceedings*, volume 5195 of *Lecture Notes in Computer Science*, pages 313–319. Springer, 2008.

9 F. Durán, S. Lucas, and J. Meseguer. Termination modulo combinations of equational theories. In S. Ghilardi and R. Sebastiani, editors, *Frontiers of Combining Systems, 7th International Symposium, FroCoS 2009, Trento, Italy, September 16-18, 2009. Proceedings*, volume 5749 of *Lecture Notes in Computer Science*, pages 246–262. Springer, 2009.

10 F. Durán and J. Meseguer. A Church-Rosser checker tool for conditional order-sorted equational Maude specifications. In P. C. Ölveczky, editor, *8th International Workshop on Rewriting Logic and its Applications*, 2010.

11 F. Durán and J. Meseguer. A Maude coherence checker tool for conditional order-sorted rewrite theories. In P. C. Ölveczky, editor, *8th International Workshop on Rewriting Logic and its Applications*, 2010.

12 S. Eker. Fast matching in combinations of regular equational theories. In J. Meseguer, editor, *Proceedings of the First International Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996.

13 S. Escobar, C. Meadows, and J. Meseguer. Maude-NPA: Cryptographic protocol analysis modulo equational properties. In *Foundations of Security Analysis and Design V, FOSAD 2007/2008/2009 Tutorial Lectures*, volume 5705 of *Lecture Notes in Computer Science*, pages 1–50. Springer, 2009.

14 S. Escobar, R. Sasse, and J. Meseguer. Folding variant narrowing and optimal variant termination. *The Journal of Logic and Algebraic Programming*, 2011. In Press.

15 F. Fages. Associative-commutative unification. *J. of Symbolic Computation*, 3:257–275, 1987.

16 J.-M. Hullot. Canonical forms and unification. In W. Bibel and R. A. Kowalski, editors, *CADE*, volume 87 of *Lecture Notes in Computer Science*, pages 318–334. Springer, 1980.

17 S. Iida, G. Denker, and C. Talcott. Document logic: Risk analysis of business processes through document authenticity. In *Second International Workshop on Dynamic and Declarative Business Processes (DDBP)*. IEEE Digital Library, 2009. Extended version to appear in Journal of Research and Practice in Information Technology, 2011.

18 J.-P. Jouannaud, C. Kirchner, and H. Kirchner. Incremental construction of unification algorithms in equational theories. In J. Díaz, editor, *ICALP*, volume 154 of *Lecture Notes in Computer Science*, pages 361–373. Springer, 1983.

19 J. Meseguer and P. Thati. Symbolic reachability analysis using narrowing and its application to verification of cryptographic protocols. *Higher-Order and Symbolic Computation*, 20(1-2):123–160, 2007.

20 G. Peterson and M. Stickel. Complete sets of reductions for some equational theories. *Journal of ACM*, 28(2):233–264, 1981.

21 R. Sasse, S. Escobar, C. Meadows, and J. Meseguer. Protocol analysis modulo a combination of theories: A case study in Maude-NPA. In *6th International Workshop on Security and Trust Management (STM'10)*, Lecture Notes in Computer Science. Springer, 2010. To appear.

22 M. E. Stickel. A complete unification algorithm for associative-commutative functions. In *Proceedings of the 4th international joint conference on Artificial intelligence - Volume 1*, pages 71–76. Morgan Kaufmann Publishers Inc., 1975.

Termination Analysis of C Programs Using Compiler Intermediate Languages

Stephan Falke[1], Deepak Kapur[2], and Carsten Sinz[1]

1 **Institute for Theoretical Computer Science**
Karlsruhe Institute of Technology (KIT), Germany
{stephan.falke, carsten.sinz}@kit.edu
2 **Department of Computer Science**
University of New Mexico, Albuquerque, NM, USA
kapur@cs.unm.edu

Abstract

Modeling the semantics of programming languages like C for the automated termination analysis of programs is a challenge if complete coverage of all language features should be achieved. On the other hand, low-level intermediate languages that occur during the compilation of C programs to machine code have a much simpler semantics since most of the intricacies of C are taken care of by the compiler frontend. It is thus a promising approach to use these intermediate languages for the automated termination analysis of C programs. In this paper we present the tool KITTeL based on this approach. For this, programs in the compiler intermediate language are translated into term rewrite systems (TRSs), and the termination proof itself is then performed on the automatically generated TRS. An evaluation on a large collection of C programs shows the effectiveness and practicality of KITTeL on "typical" examples.

Digital Object Identifier 10.4230/LIPIcs.RTA.2011.41

Category System Description

1 Introduction

Methods for automatically proving termination of imperative programs operating on integers have received increased attention recently. The most commonly used automatic method for this is based on linear ranking functions which linearly combine the values of the program variables in a given state [5, 6, 19, 20]. More recently, the combination of abstraction refinement and linear ranking functions has been considered [8, 9]. Based on this idea, the tool Terminator [10] has reportedly been used for showing termination of device drivers.

Developing a tool that can handle all intricacies of C is a challenge since C employs a complex syntax and semantics. It is not clear to what extent the implementations of the aforementioned methods can handle real-life C programs since the papers are typically based on idealized transition systems and the implementations are not publicly available.

We advocate to perform the termination analysis of C programs not on the source code level but rather on the level of a compiler intermediate representation (IR). This approach has the following advantages:

1. The IR is considerably simpler than C. This makes it relatively easy to accept *any* C program as an input. Features of the IR that are not (yet) supported by the termination analysis techniques can easily be abstracted automatically.

This work was supported in part by the "Concept for the Future" of Karlsruhe Institute of Technology within the framework of the German Excellence Initiative.

22nd International Conference on Rewriting Techniques and Applications (RTA'11).
Editor: M. Schmidt-Schauß; pp. 41–50

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

2. The program that is analyzed is much closer to the program that is actually executed on the computer since ambiguities of C's semantics have already been resolved.
3. In producing the IR, compilers already use program optimizations that might simplify the termination analysis significantly.

For similar reasons, termination analysis of Java programs is often performed on the bytecode level and not on the source code [1, 22, 18, 4].

In this paper, we focus on the LLVM compiler framework and its intermediate language LLVM-IR [17]. The method itself is independent of the concrete IR, however. Since there are compilers for various programming languages built atop of LLVM, the methods presented in this paper can be used for the termination analysis of programs written in C, C++, Objective-C, and further programming languages.

Termination analysis of LLVM-IR programs is then performed by generating a term rewrite system (TRS) from the LLVM-IR program. Termination analysis of TRSs has been investigated extensively in the past (see [24] for a survey). In this paper, TRSs with constraints over the integers (`int`-*based TRSs*) are used, where the constraints are relations on the variables expressed as quantifier-free formulas from non-linear arithmetic. Similarly to what was proposed in [12, 15], well-known methods from the term rewriting literature can be adapted for the termination analysis of `int`-based TRSs.

▶ **Example 1.** Consider the C program on the left-hand side:

```
int power(int x, int y) {
    int r = 1;
    while (y > 0) {
        r = r * x;
        y = y - 1;
    }
    return r;
}
```

$$\begin{aligned}
\mathsf{state}_{\mathsf{start}}(v_x, v_y, v_{y.0}, v_{r.0}) &\to \mathsf{state}_{\mathsf{entry_{in}}}(v_x, v_y, v_{y.0}, v_{r.0}) \\
\mathsf{state}_{\mathsf{entry_{in}}}(v_x, v_y, v_{y.0}, v_{r.0}) &\to \mathsf{state}_{\mathsf{bb1_{in}}}(v_x, v_y, v_y, 1) \\
\mathsf{state}_{\mathsf{bb1_{in}}}(v_x, v_y, v_{y.0}, v_{r.0}) &\to \mathsf{state}_{\mathsf{bb_{in}}}(v_x, v_y, v_{y.0}, v_{r.0}) \ [\![v_{y.0} > 0]\!] \\
\mathsf{state}_{\mathsf{bb1_{in}}}(v_x, v_y, v_{y.0}, v_{r.0}) &\to \mathsf{state}_{\mathsf{return_{in}}}(v_x, v_y, v_{y.0}, v_{r.0}) \ [\![v_{y.0} \leq 0]\!] \\
\mathsf{state}_{\mathsf{bb_{in}}}(v_x, v_y, v_{y.0}, v_{r.0}) &\to \mathsf{state}_{\mathsf{bb1_{in}}}(v_x, v_y, v_{y.0} - 1, v_{r.0} * v_x) \\
\mathsf{state}_{\mathsf{return_{in}}}(v_x, v_y, v_{y.0}, v_{r.0}) &\to \mathsf{state}_{\mathsf{stop}}(v_x, v_y, v_{y.0}, v_{r.0})
\end{aligned}$$

Using the methods presented in this paper, the `int`-based TRS shown on the right-hand side is automatically obtained from the LLVM-IR of the C program. Intuitively, the variables v_x and v_y represent the inputs to the function, whereas the variables $v_{y.0}$ and $v_{r.0}$ correspond to the (changing) program variables y and r used inside the loop of the function (why the program variable y gives rise to v_y and $v_{y.0}$ is explained in Section 3). The function symbols used in the `int`-based TRS intuitively correspond to a program counter. ◀

The approach has been implemented in the publicly available termination tool `KITTeL`. An empirical evaluation on a large collection of examples taken from various sources clearly shows the effectiveness and practicality of our method.

2 `int`-Based TRSs

In order to model integers, the function symbols from $\mathcal{F}_{\texttt{int}} = \mathcal{F}_{\mathbb{Z}} \cup \{+, *, -\}$ with $\mathcal{F}_{\mathbb{Z}} = \{\mathsf{n} \mid n \in \mathbb{Z}\}$ and types $+, * : \texttt{int} \times \texttt{int} \to \texttt{int}$, and $- : \texttt{int} \to \texttt{int}$ are used. Terms built from these function symbols and a disjoint set $\mathcal{V}$ of variables are called `int`-*terms*. We use a simplified notation for `int`-terms, e.g., the `int`-term $(x + (-(y * y))) + 3$ is written as $x - y^2 + 3$. A *linear* `int`-term is an `int`-term that does not contain any occurrence of "$*$".

$\mathcal{F}_{\texttt{int}}$ is extended by finitely many function symbols f with types $\texttt{int} \times \ldots \times \texttt{int} \to \texttt{univ}$, where `univ` is a type distinct from `int`. The set containing these additional function symbols

is denoted by $\mathcal{F}$ and $\mathcal{T}(\mathcal{F}, \mathcal{F}_{\texttt{int}}, \mathcal{V})$ denotes the set of terms of the form $f(s_1, \ldots, s_n)$ where $f \in \mathcal{F}$ and $s_1, \ldots, s_n$ are `int`-terms. A *substitution* is a mapping from variables to `int`-terms.

`int`-constraints are quantifier-free formulas from (non-linear) integer arithmetic. An *atomic* `int`*-constraint* has the form $s \simeq t$, $s \geq t$, or $s > t$ for `int`-terms s, t and the set of `int`*-constraints* is the closure of atomic `int`-constraints under $\top$ (truth), $\neg$ (negation), and $\wedge$ (conjunction). The Boolean connectives $\bot$, $\vee$, $\Rightarrow$, and $\Leftrightarrow$ are defined as usual. `int`-constraints have the expected semantics regarding `int`*-validity* and `int`*-satisfiability*. These properties are in general only decidable for linear or variable-free `int`-constraints.

The rewrite rules of `int`-based TRSs are equipped with `int`-constraints. These constraints are used in order to restrict the applicability of the rewrite rules, see Definition 3. The rules generalize the $\mathcal{PA}$-based rewrite rules from [12]. Alternatively, they can be interpreted as a restricted form of the rewrite rules considered in [15] which allow nested function symbols.

▶ **Definition 2.** An `int`*-based rewrite rule* has the form $l \to r[\![\varphi]\!]$ such that $l = f(x_1, \ldots, x_n)$ where $x_1, \ldots, x_n$ are pairwise distinct variables, $r \in \mathcal{T}(\mathcal{F}, \mathcal{F}_{\texttt{int}}, \mathcal{V})$, and φ is an `int`-constraint.

Notice that r and φ may contain variables that are not occurring in l. The restriction that the arguments on the left-hand side are pairwise distinct variables simplifies the definition of the rewrite relation of an `int`-based TRS since matching becomes trivial. Notice that equality between the arguments x_i and x_j can be enforced by adding the `int`-constraint $x_i \simeq x_j$. The constraint $\top$ is omitted in an `int`-based rewrite rule $l \to r[\![\top]\!]$. An `int`*-based term rewrite system (*`int`*-based TRS)* $\mathcal{R}$ is a finite set of `int`-based rewrite rules.

▶ **Definition 3.** For an `int`-based TRS $\mathcal{R}$, the relation $s \to_{\texttt{int}\backslash\mathcal{R}} t$ for terms s, t of the form $f(\mathsf{n}_1, \ldots, \mathsf{n}_k)$ with $\mathsf{n}_1, \ldots, \mathsf{n}_k \in \mathcal{F}_{\mathbb{Z}}$ holds iff there exist $l \to r[\![\varphi]\!] \in \mathcal{R}$ and an $\mathcal{F}_{\mathbb{Z}}$-based substitution σ such that **1.** $s = l\sigma$, **2.** $\varphi\sigma$ is `int`-valid, and **3.** $t = \mathsf{norm}(r\sigma)$. Here, a substitution σ is $\mathcal{F}_{\mathbb{Z}}$*-based* iff $\sigma(x) \in \mathcal{F}_{\mathbb{Z}}$ for all variables x and $\mathsf{norm}(r\sigma)$ evaluates according to the usual semantics of "+", "∗", and "−" on variable-free terms.

Termination of `int`-based TRSs can be shown by using an extension of the methods presented in [12] which are motivated by the dependency pair method [2, 16, 11] and are based on the notion of *chains*. For an `int`-based TRS $\mathcal{R}$, a (possibly infinite) sequence of `int`-based rewrite rules $l_1 \to r_1[\![\varphi_1]\!], l_2 \to r_2[\![\varphi_2]\!], \ldots$ from $\mathcal{R}$ is an $\mathcal{R}$*-chain* iff there exists an $\mathcal{F}_{\mathbb{Z}}$-based substitution σ such that $\mathsf{norm}(r_i\sigma) = l_{i+1}\sigma$ and $\varphi_i\sigma$ is `int`-valid for all $i \geq 1$.

Chains provide a precise characterization of termination in the sense that an `int`-based TRS $\mathcal{R}$ is terminating if and only if there are no infinite $\mathcal{R}$-chains. This characterization of termination is utilized by introducing sound *processors* which are used to transform an `int`-based TRS into a set of `int`-based TRSs such the input TRS is terminating if all output TRSs are terminating. The following are the two most important processors for `int`-based TRSs (details on these processors and their implementation can be found in [13]):

- SCC decomposition: Here, it is approximated which rules may follow each other in chains. Then, $\mathcal{R}$ is decomposed into the non-trivial SCCs of the thus obtained graph.
- Polynomial interpretations: A polynomial interpretation maps each n-ary $f \in \mathcal{F}$ to a polynomial $\mathcal{P}ol(f) \in \mathbb{Z}[x_1, \ldots, x_n]$. This mapping extends to terms from $\mathcal{T}(\mathcal{F}, \mathcal{F}_{\texttt{int}}, \mathcal{V})$ by letting $[f(t_1, \ldots, t_n)]_{\mathcal{P}ol} = \mathcal{P}ol(f)(t_1, \ldots, t_n)$. Then, terms are compared (in the context of a constraint) by comparing polynomials, and all strictly decreasing rules may be deleted.

3 Translating LLVM-IR Programs into `int`-Based TRSs

Converting programs from a real-life programming language such as C into `int`-based TRSs is non-trivial. C has a complex syntax and semantics, resulting in many cases that need to

be considered. An alternative to operating on the source code level is the use of compiler intermediate languages. These intermediate languages typically have a simple syntax and semantics, thus simplifying the translation into `int`-based TRSs significantly.

In this paper, we consider LLVM and its intermediate language LLVM-IR [17]. An LLVM-IR program is an assembly program for a register machine with an unbounded number of registers. A program consists of type definitions, global variable declarations, and the program itself, given in the form of one or more functions. Each function is represented as a graph of basic blocks (see Example 4 for an LLVM-IR program), where each basic block is a list of instructions, and execution of a function starts at the basic block named `entry`. For our purpose, LLVM-IR instructions can be categorized into six classes:

- *Three-address code (TAC)* instructions such as `%2 = mul i32 %r.0, %x`.
- *Control flow* instructions: *Branch* (`br`), *return* (`ret`), *phi* (`phi`).
- *Function calls* using `call` instructions.
- *Memory access* instructions, namely `load` and `store`.
- *Address calculations* using `getelementptr` instructions.
- *Auxiliary instructions* like type casts or bit-level instructions.

Branches and return instructions are only allowed as the last instruction of a basic block and each basic block is terminated by one of these instructions.

LLVM-IR programs are in *static single assignment (SSA)* form, i.e., each register (variable) is assigned exactly once in the static LLVM-IR program. Due to this, it becomes necessary to introduce the phi-instruction `phi`, which is used to select one of several values whenever the control flow in a program converges again (e.g., after an `if-then-else` statement). For example, the meaning of `%r.0 = phi i32 [ 1, %entry ], [ %1, %bb ]` contained in the basic block `bb1` in Example 4 is that the register `%r.0` is assigned the value `1` if the control flow passed from `entry` to `bb1`. If control passed from `bb` to `bb1`, then `%r.0` is assigned the value contained in `%1`. Phi-instructions only occur at the beginning of basic blocks.

All variables in LLVM-IR are typed. Available types include a void type, integer types like `i32` (where the bit-width is given explicitly), floating-point types, and derived types (such as pointer, array and structure types). The integer type `i1` is used as a dedicated Boolean type. Aggregate types (structures and arrays) are accessed using memory load/store operations and offset calculations using the `getelementptr` instruction.

▶ Assumption 1. All LLVM-IR integer types `i`k with $k > 1$ are identified with $\mathbb{Z}$.

3.1 Single Non-Recursive Function Operating on Integers

First, it is assumed that the LLVM-IR program operates only on integer types. Furthermore, it is assumed that there is exactly one function, and that this function does not contain any `call` instruction. It thus only contains arithmetical instructions (`add`, `sub`, `mul`, signed and unsigned `div` and `rem`), comparison instructions (equality `eq`, disequality `neq`, (un)signed greater-than (`u`|`s`)`gt`, greater-or-equal (`u`|`s`)`ge`, less-than (`u`|`s`)`lt`, and less-or-equal (`u`|`s`)`le`), control flow instructions, and type cast instructions.

▶ **Example 4.** For the C program from Example 1, the LLVM-IR program shown in Figure 1 is obtained using the LLVM compiler frontend `llvm-gcc`. Here, the basic blocks `bb1` and `bb` correspond to the while-loop in the C program. ◀

An LLVM-IR program is now translated into an `int`-based TRS as follows. Each integer-typed function argument, each register defined by an integer-typed TAC instruction, and each register defined by an integer-typed phi-instruction is mapped to a variable in the TRS.

```
define i32 @power(i32 %x, i32 %y) {
entry:
  br label %bb1

bb1:
  %y.0 = phi i32 [ %y, %entry ], [ %2, %bb ]
  %r.0 = phi i32 [ 1, %entry ], [ %1, %bb ]
  %0 = icmp sgt i32 %y.0, 0
  br i1 %0, label %bb, label %return
```

```
bb:
  %1 = mul i32 %r.0, %x
  %2 = sub i32 %y.0, 1
  br label %bb1

return:
  ret i32 %r.0
}
```

Figure 1 LLVM-IR program for the C program from Example 1.

Then, each integer-typed TAC instruction I is assigned two function symbols $\mathsf{state}_{I_{\mathsf{in}}}$ and $\mathsf{state}_{I_{\mathsf{out}}}$ and gives rise to a rewrite rule $\mathsf{state}_{I_{\mathsf{in}}}(\ldots) \to \mathsf{state}_{I_{\mathsf{out}}}(\ldots)[\![\varphi]\!]$ that mimics the effect of I. Here, division and remainder instructions are handled by introducing fresh variables on the right-hand side and adding appropriate constraints on that variable.

The control flow of the LLVM-IR program is mimicked as follows. The function symbols $\mathsf{state}_{\mathsf{start}}$ and $\mathsf{state}_{\mathsf{stop}}$ are introduced, denoting starting and stopping states, respectively. Next, each basic block bb is assigned two function symbols $\mathsf{state}_{bb_{\mathsf{in}}}$ and $\mathsf{state}_{bb_{\mathsf{out}}}$. These function symbols correspond to the points after the final phi-instruction in bb and before the branch or return instruction of bb, respectively. If bb contains the (possibly empty) sequence $\Omega = \langle I_1, \ldots, I_n \rangle$ of integer-typed TAC instructions, then, for two consecutive instructions I_k and I_{k+1}, the function symbols $\mathsf{state}_{I_{k_{\mathsf{out}}}}$ and $\mathsf{state}_{I_{k+1_{\mathsf{in}}}}$ are identified. Furthermore, rules $\mathsf{state}_{bb_{\mathsf{in}}}(\ldots) \to \mathsf{state}_{I_{1_{\mathsf{in}}}}(\ldots)$ and $\mathsf{state}_{I_{n_{\mathsf{out}}}}(\ldots) \to \mathsf{state}_{bb_{\mathsf{out}}}(\ldots)$ (if Ω is non-empty) or $\mathsf{state}_{bb_{\mathsf{in}}}(\ldots) \to \mathsf{state}_{bb_{\mathsf{out}}}(\ldots)$ (if Ω is empty) is added. If bb is terminated by a return instruction, then the rule $\mathsf{state}_{bb_{\mathsf{out}}}(\ldots) \to \mathsf{state}_{\mathsf{stop}}(\ldots)$ is added. Otherwise, bb is terminated by a branch instruction. For an unconditional branch to bb', a rule $\mathsf{state}_{bb_{\mathsf{out}}}(\ldots) \to \mathsf{state}_{bb'_{\mathsf{in}}}(\ldots)$ is added, where the variables on the right-hand side that correspond to phi-instructions are instantiated according to their value in the case where control flow passes from bb to bb'. A conditional branch is treated similarly, but now the rules are equipped with the (possibly negated) branch condition as a constraint.

▶ **Example 5.** Consider the C program from Example 1 and its LLVM-IR from Example 4. Using the translation outlined above, the `int`-based TRS

$$\begin{aligned}
\mathsf{state}_{\mathsf{start}}(v_x, v_y, v_{y.0}, v_{r.0}, v_1, v_2) &\to \mathsf{state}_{\mathsf{entry}_{\mathsf{in}}}(v_x, v_y, v_{y.0}, v_{r.0}, v_1, v_2)\\
\mathsf{state}_{\mathsf{entry}_{\mathsf{in}}}(v_x, v_y, v_{y.0}, v_{r.0}, v_1, v_2) &\to \mathsf{state}_{\mathsf{entry}_{\mathsf{out}}}(v_x, v_y, v_{y.0}, v_{r.0}, v_1, v_2)\\
\mathsf{state}_{\mathsf{entry}_{\mathsf{out}}}(v_x, v_y, v_{y.0}, v_{r.0}, v_1, v_2) &\to \mathsf{state}_{\mathsf{bb1}_{\mathsf{in}}}(v_x, v_y, v_y, 1, v_1, v_2)\\
\mathsf{state}_{\mathsf{bb1}_{\mathsf{in}}}(v_x, v_y, v_{y.0}, v_{r.0}, v_1, v_2) &\to \mathsf{state}_{\mathsf{bb1}_{\mathsf{out}}}(v_x, v_y, v_{y.0}, v_{r.0}, v_1, v_2)\\
\mathsf{state}_{\mathsf{bb1}_{\mathsf{out}}}(v_x, v_y, v_{y.0}, v_{r.0}, v_1, v_2) &\to \mathsf{state}_{\mathsf{bb}_{\mathsf{in}}}(v_x, v_y, v_{y.0}, v_{r.0}, v_1, v_2)\ [\![v_{y.0} > 0]\!]\\
\mathsf{state}_{\mathsf{bb1}_{\mathsf{out}}}(v_x, v_y, v_{y.0}, v_{r.0}, v_1, v_2) &\to \mathsf{state}_{\mathsf{return}_{\mathsf{in}}}(v_x, v_y, v_{y.0}, v_{r.0}, v_1, v_2)\ [\![v_{y.0} \leq 0]\!]\\
\mathsf{state}_{\mathsf{bb}_{\mathsf{in}}}(v_x, v_y, v_{y.0}, v_{r.0}, v_1, v_2) &\to \mathsf{state}_1(v_x, v_y, v_{y.0}, v_{r.0}, v_1, v_2)\\
\mathsf{state}_1(v_x, v_y, v_{y.0}, v_{r.0}, v_1, v_2) &\to \mathsf{state}_2(v_x, v_y, v_{y.0}, v_{r.0}, v_{r.0} * v_x, v_2)\\
\mathsf{state}_2(v_x, v_y, v_{y.0}, v_{r.0}, v_1, v_2) &\to \mathsf{state}_3(v_x, v_y, v_{y.0}, v_{r.0}, v_1, v_{y.0} - 1)\\
\mathsf{state}_3(v_x, v_y, v_{y.0}, v_{r.0}, v_1, v_2) &\to \mathsf{state}_{\mathsf{bb}_{\mathsf{out}}}(v_x, v_y, v_{y.0}, v_{r.0}, v_1, v_2)\\
\mathsf{state}_{\mathsf{bb}_{\mathsf{out}}}(v_x, v_y, v_{y.0}, v_{r.0}, v_1, v_2) &\to \mathsf{state}_{\mathsf{bb1}_{\mathsf{in}}}(v_x, v_y, v_2, v_1, v_1, v_2)\\
\mathsf{state}_{\mathsf{return}_{\mathsf{in}}}(v_x, v_y, v_{y.0}, v_{r.0}, v_1, v_2) &\to \mathsf{state}_{\mathsf{return}_{\mathsf{out}}}(v_x, v_y, v_{y.0}, v_{r.0}, v_1, v_2)\\
\mathsf{state}_{\mathsf{return}_{\mathsf{out}}}(v_x, v_y, v_{y.0}, v_{r.0}, v_1, v_2) &\to \mathsf{state}_{\mathsf{stop}}(v_x, v_y, v_{y.0}, v_{r.0}, v_1, v_2)
\end{aligned}$$

is obtained. Here, simplified names have been used for the function symbols. ◀

Now an LLVM-IR program is terminating if the `int`-based TRS $\mathcal{R}_P$ is terminating, but $\mathcal{R}_P$ might be non-terminating even if P is terminating (see Section 3.5 for a partial remedy).

3.2 Simplification of `int`-Based Rewrite Rules

The translation given above produces a large number of `int`-based rewrite rules since each integer-typed TAC instruction and each transition between basic blocks gives rise to one or more rules. In order to decrease the number of `int`-based rewrite rules, it is possible to combine several rules into a single one. Intuitively, this corresponds to the composition of the effect of several integer-typed TAC instructions into a single state change.

For `int`-based TRSs obtained from LLVM-IR, the set of *control points* C consists of the function symbols $\mathsf{state}_{\mathsf{start}}$, $\mathsf{state}_{\mathsf{stop}}$, and $\mathsf{state}_{bb_{\mathsf{in}}}$ for each basic block bb of the program. It is then possible to eliminate `int`-based rewrite rules that contain a function symbol not occurring in C by combining an `int`-based rewrite rule $\mathsf{state}_i(x_1,\ldots,x_n) \to \mathsf{state}_j(e_1,\ldots,e_n)[\![\varphi]\!]$, where $\mathsf{state}_i \in C$ and $\mathsf{state}_j \notin C$, with a rule $\mathsf{state}_j(x_1,\ldots,x_n) \to \mathsf{state}_k(e'_1,\ldots,e'_n)[\![\psi]\!]$, resulting in $\mathsf{state}_i(x_1,\ldots,x_n) \to \mathsf{state}_k(e'_1\omega,\ldots,e'_n\omega)[\![\varphi \wedge \psi\omega]\!]$ where $\omega = \{x_1 \mapsto e_1,\ldots,x_n \mapsto e_n\}$. This *chaining* needs to be done for all possible rules that have state_j on their left-hand side. The old rules are replaced by the new rules and the process is iterated until all rules with a function symbol from C on the left-hand side also have a function symbol from C on their right-hand side.

▶ **Example 6.** Continuing Example 5, the control points are $\mathsf{state}_{\mathsf{start}}$, $\mathsf{state}_{\mathsf{stop}}$, $\mathsf{state}_{\mathsf{entry}_{\mathsf{in}}}$, $\mathsf{state}_{\mathsf{bb1}_{\mathsf{in}}}$, $\mathsf{state}_{\mathsf{bb}_{\mathsf{in}}}$, and $\mathsf{state}_{\mathsf{return}_{\mathsf{in}}}$. Combining rules w.r.t. these control points produces

$$\begin{aligned}
\mathsf{state}_{\mathsf{start}}(v_x,v_y,v_{y.0},v_{r.0},v_1,v_2) &\to \mathsf{state}_{\mathsf{entry}_{\mathsf{in}}}(v_x,v_y,v_{y.0},v_{r.0},v_1,v_2)\\
\mathsf{state}_{\mathsf{entry}_{\mathsf{in}}}(v_x,v_y,v_{y.0},v_{r.0},v_1,v_2) &\to \mathsf{state}_{\mathsf{bb1}_{\mathsf{in}}}(v_x,v_y,v_y,1,v_1,v_2)\\
\mathsf{state}_{\mathsf{bb1}_{\mathsf{in}}}(v_x,v_y,v_{y.0},v_{r.0},v_1,v_2) &\to \mathsf{state}_{\mathsf{bb}_{\mathsf{in}}}(v_x,v_y,v_{y.0},v_{r.0},v_1,v_2)\ [\![v_{y.0} > 0]\!]\\
\mathsf{state}_{\mathsf{bb1}_{\mathsf{in}}}(v_x,v_y,v_{y.0},v_{r.0},v_1,v_2) &\to \mathsf{state}_{\mathsf{return}_{\mathsf{in}}}(v_x,v_y,v_{y.0},v_{r.0},v_1,v_2)\ [\![v_{y.0} \leq 0]\!]\\
\mathsf{state}_{\mathsf{bb}_{\mathsf{in}}}(v_x,v_y,v_{y.0},v_{r.0},v_1,v_2) &\to \mathsf{state}_{\mathsf{bb1}_{\mathsf{in}}}(v_x,v_y,v_{y.0}-1,v_{r.0}*v_x,v_1,v_{y.0}-1)\\
\mathsf{state}_{\mathsf{return}_{\mathsf{in}}}(v_x,v_y,v_{y.0},v_{r.0},v_1,v_2) &\to \mathsf{state}_{\mathsf{stop}}(v_x,v_y,v_{y.0},v_{r.0},v_1,v_2)
\end{aligned}$$

as a new `int`-based TRS. ◀

After the combination of `int`-based rewrite rules, it is possible to remove some arguments from the function symbols. Notice that the effect of instructions that are only used in the same basic block where they are defined or in phi-instructions has been propagated by the combination of rules. Thus, the corresponding variables can be removed as arguments from the function symbols. On the syntactic level of rewrite rules, an argument position i is *unneeded* if, for all rewrite rules $l \to r[\![\varphi]\!]$, the variable occurring in position i of l does not occur in φ and only in argument position i of r.

▶ **Example 7.** After removing the unneeded arguments in Example 6,

$$\begin{aligned}
\mathsf{state}_{\mathsf{start}}(v_x,v_y,v_{y.0},v_{r.0}) &\to \mathsf{state}_{\mathsf{entry}_{\mathsf{in}}}(v_x,v_y,v_{y.0},v_{r.0})\\
\mathsf{state}_{\mathsf{entry}_{\mathsf{in}}}(v_x,v_y,v_{y.0},v_{r.0}) &\to \mathsf{state}_{\mathsf{bb1}_{\mathsf{in}}}(v_x,v_y,v_y,1)\\
\mathsf{state}_{\mathsf{bb1}_{\mathsf{in}}}(v_x,v_y,v_{y.0},v_{r.0}) &\to \mathsf{state}_{\mathsf{bb}_{\mathsf{in}}}(v_x,v_y,v_{y.0},v_{r.0})\ [\![v_{y.0} > 0]\!]\\
\mathsf{state}_{\mathsf{bb1}_{\mathsf{in}}}(v_x,v_y,v_{y.0},v_{r.0}) &\to \mathsf{state}_{\mathsf{return}_{\mathsf{in}}}(v_x,v_y,v_{y.0},v_{r.0})\ [\![v_{y.0} \leq 0]\!]\\
\mathsf{state}_{\mathsf{bb}_{\mathsf{in}}}(v_x,v_y,v_{y.0},v_{r.0}) &\to \mathsf{state}_{\mathsf{bb1}_{\mathsf{in}}}(v_x,v_y,v_{y.0}-1,v_{r.0}*v_x)\\
\mathsf{state}_{\mathsf{return}_{\mathsf{in}}}(v_x,v_y,v_{y.0},v_{r.0}) &\to \mathsf{state}_{\mathsf{stop}}(v_x,v_y,v_{y.0},v_{r.0})
\end{aligned}$$

is obtained since arguments 5 and 6 are not needed. The methods outlined in Section 2 can easily prove termination of this TRS. ◀

3.3 Several Functions Operating on Integers

In this section it is discussed how the translation from LLVM-IR programs into `int`-based TRSs can be extended to the case of several functions. For this, the user first specifies which function should be the starting function for the termination analysis (often, this is the `main` function). It is then necessary to include all functions that are (transitively) called by this starting function in the termination analysis.

A given LLVM-IR program might not contain implementations of all functions being called. Instead, some functions may only be given as prototype declarations (e.g., library functions).

▶ Assumption 2. All functions that are only declared as prototypes are terminating. Furthermore, these functions do not call functions defined in the program.

If the user-defined functions have a function call hierarchy with arbitrary recursion, then it needs to be ensured that the sequence of recursive calls is terminating. For this, each call instruction to a function with non-void type gives rise to *two* rewrite rules. One rewrite rule introduces a fresh variable on the right-hand side which abstracts the return value of the called function.[1] This rule has the form $\mathsf{state}_i(\ldots) \to \mathsf{state}_j(\ldots, z, \ldots)$, where z is a fresh variable. The second rewrite rule has the form $\mathsf{state}_i(\ldots) \to \mathsf{state}^f_{\mathsf{start}}(\ldots)$, where $\mathsf{state}^f_{\mathsf{start}}$ is the called function's start symbol. A call to a function with void type is handled similarly, but no fresh variable is introduced on the right-hand side.

▶ **Example 8.** The C and LLVM-IR programs in Figure 2 compute Ackermann's function. Termination of the generated TRS can easily be shown using the methods from Section 2. ◀

3.4 Programs Containing Pointers and Floating Point Numbers

`int`-based TRSs (currently) do not support pointers or floating point numbers. Thus, all instructions of these types are ignored in the translation. In order to have a non-termination preserving translation, instructions that take a pointer or a floating point number and return an integer (such as `load` or `fptosi`) are abstracted to an unspecified value which corresponds to a fresh variable on the right-hand side of the generated rule. Similarly, comparisons of floating point numbers are abstracted to return arbitrary results.

3.5 Utilizing Static Analysis Methods

Notice that the translation from LLVM-IR programs into `int`-based TRSs does not propagate information about the initial state of the program. Thus, the `int`-based TRS $\mathcal{R}_P$ might be non-terminating even if the program P is terminating since reductions w.r.t. $\mathcal{R}_P$ are not restricted to reductions that are reachable from the initial state. It is thus desirable to make information about the initial state explicit throughout the program. Furthermore, a successful automatic termination proof might require simple invariants on the program variables (such as "a variable is always non-negative"). This kind of information can be obtained automatically using static analysis tools such as `Aspic`/`C2fsm` [14]. The obtained information can be added to the C program in the form of calls to an `assume` function with a built-in semantics. In the translation, these calls are converted into constraints that correspond to the invariants. We are planning to integrate a static analysis tool into the translation process so that these annotations do not need to be added manually.

[1] This simple abstraction is already sufficient in many cases. It is of course also possible to add constraints on the return value. Non-recursive functions can also be inlined on the LLVM-IR level, thus precisely tracking their return value.

```
int ack(int m, int n) {
    if (m <= 0)
        return n + 1;
    else if (n <= 0)
        return ack(m − 1, 1);
    else
        return ack(m − 1, ack(m, n − 1));
}
```

$$\begin{aligned}
\mathsf{state}_{\mathsf{start}}(v_m, v_n) &\to \mathsf{state}_{\mathsf{entry_{in}}}(v_m, v_n)\\
\mathsf{state}_{\mathsf{entry_{in}}}(v_m, v_n) &\to \mathsf{state}_{\mathsf{bb_{in}}}(v_m, v_n)\ [\![v_m \leq 0]\!]\\
\mathsf{state}_{\mathsf{entry_{in}}}(v_m, v_n) &\to \mathsf{state}_{\mathsf{bb1_{in}}}(v_m, v_n)\ [\![v_m > 0]\!]\\
\mathsf{state}_{\mathsf{bb_{in}}}(v_m, v_n) &\to \mathsf{state}_{\mathsf{stop}}(v_m, v_n)\\
\mathsf{state}_{\mathsf{bb1_{in}}}(v_m, v_n) &\to \mathsf{state}_{\mathsf{bb2_{in}}}(v_m, v_n)\ [\![v_n \leq 0]\!]\\
\mathsf{state}_{\mathsf{bb1_{in}}}(v_m, v_n) &\to \mathsf{state}_{\mathsf{bb3_{in}}}(v_m, v_n)\ [\![v_n > 0]\!]\\
\mathsf{state}_{\mathsf{bb2_{in}}}(v_m, v_n) &\to \mathsf{state}_{\mathsf{start}}(v_m - 1, 1)\\
\mathsf{state}_{\mathsf{bb2_{in}}}(v_m, v_n) &\to \mathsf{state}_{\mathsf{stop}}(v_m, v_n)\\
\mathsf{state}_{\mathsf{bb3_{in}}}(v_m, v_n) &\to \mathsf{state}_{\mathsf{start}}(v_m, v_n - 1)\\
\mathsf{state}_{\mathsf{bb3_{in}}}(v_m, v_n) &\to \mathsf{state}_{\mathsf{start}}(v_m - 1, z)\\
\mathsf{state}_{\mathsf{bb3_{in}}}(v_m, v_n) &\to \mathsf{state}_{\mathsf{stop}}(v_m, v_n)
\end{aligned}$$

```
define i32 @ack(i32 %m, i32 %n) {
entry:
  %0 = icmp sle i32 %m, 0
  br i1 %0, label %bb, label %bb1

bb:
  %1 = add nsw i32 %n, 1
  ret i32 %1

bb1:
  %2 = icmp sle i32 %n, 0
  br i1 %2, label %bb2, label %bb3

bb2:
  %3 = sub nsw i32 %m, 1
  %4 = call i32 @ack(i32 %3, i32 1)
  ret i32 %4

bb3:
  %5 = sub nsw i32 %n, 1
  %6 = call i32 @ack(i32 %m, i32 %5)
  %7 = sub nsw i32 %m, 1
  %8 = call i32 @ack(i32 %7, i32 %6)
  ret i32 %8
}
```

Figure 2 Ackermann's function in C, LLVM-IR, and as an `int`-based TRS.

4 Evaluation

In order to show the effectiveness and practicality of the proposed approach, it has been implemented in the tool `KITTeL` (KIT `int`-based TRS Termination Laboratory). Like its predecessor `pasta` [12], `KITTeL` consists of about 2400 lines of OCaml code. The input to `KITTeL` is an `int`-based TRS, the translation from LLVM-IR into `int`-based TRSs has been implemented in the separate tool `llvm2kittel` using about 3800 lines of C++ code.

The implementation in `KITTeL/llvm2kittel` has been evaluated on a collection of 174 examples that were taken from various places, including several recent papers on the termination of imperative programs [3, 5, 6, 8, 9, 19, 20], the textbook [21], and the `zlib` compression library. Furthermore, 31 examples were taken from TPDB's Java category [23] and converted to C. The collection of examples includes "classical" algorithms such as searching and sorting algorithms, cyclic redundancy check and hash code algorithms, encryption/decryption algorithms, image processing algorithms, and numerical algorithms. 14 out of these 174 examples require simple invariants on the program variables (such as "a variable is always non-negative") for a successful termination proof. This kind of information can be obtained automatically using static program analysis tools such as `Aspic/C2fsm` [14].

`KITTeL/llvm2kittel` has been able to show termination of all[2] examples fully automatically, on average taking less than 0.3 seconds (on a 2.4 GHz Intel® Core™2 Duo processor with 4 GB main memory) for each example, with the longest time being slightly more than 3 seconds. These times include the compilation from C into LLVM-IR, the translation from

[2] If the invariants are omitted from the aforementioned 14 examples, then termination cannot be shown.

LLVM-IR into a TRS, and the termination analysis of the obtained TRS. The following table contains details for some of the examples. Here, "LOC" gives the number of code lines in the C program and "RR" gives the number of rewrite rules that are generated.

C program	LOC	RR	Time / s	C program	LOC	RR	Time / s
allroots	200	77	0.861	fft	99	30	0.342
almabench	390	42	0.370	hash	241	80	0.566
barr-crc16	265	45	0.398	jfdctint	366	15	0.374
barr-crc32	265	45	0.402	lpbench	419	134	1.155
barr-crc-ccitt	265	35	0.318	mergesort-recursive	42	50	0.634
bellman-ford	75	39	0.369	n-body	141	35	0,287
blit	98	28	0.311	prim	83	44	0.487
blowfish	476	43	0.389	sort	138	90	0.757
bmpfile	749	254	3.050	spectral-norm	52	39	0.312
bresenham	36	9	0.106	sphere	157	68	0.617
c-aes	236	64	0.385	spiral	176	80	0.722
c-des	399	64	0.477	zlib-adler32	124	34	0.891
cube	146	68	0.616	zlib-crc32-BYFOUR	335	41	1.182
dijkstra	78	58	0.693	zlib-crc32	333	13	0.170

Notice that an empirical comparison with the methods from [5, 6, 8, 9, 19, 20] is not possible since implementations of these methods are not publicly available. The 31 Java programs from TPDB were also analyzed using the web interfaces of the Java termination tools COSTA [1] and AProVE [18, 4] using the default settings.

	Successful proofs	Unsuccessful attempts	Timeouts (60s)	Average time / s
KITTeL	31	–	–	0.133
COSTA	22	9	–	0.265
AProVE	28	–	3	13.265

Thus, KITTeL clearly shows the practicality and effectiveness of the proposed approach on a collection of "typical" examples. The examples, detailed results, an a link to a web interface of KITTeL are available at http://baldur.iti.kit.edu/~falke/kittel/.

5 Conclusions

We have presented a method for showing termination of C programs that is based on compiler intermediate languages and term rewriting techniques. For this, a C program is translated into an intermediate language by the compiler frontend and the obtained intermediate representation is then translated into a term rewrite system. Finally, termination of the obtained TRS is shown using term rewriting techniques.

In this paper, all integer types of the intermediate language are identified with $\mathbb{Z}$. Notice, however, that this abstraction might alter the termination behavior of the program under investigation. The methods from [5, 6, 8, 9, 19, 20] also exhibit this problem, and only [7] investigates the generation of ranking functions for bitvectors. In future work, we are planning to investigate how to model the bitvector behavior more precisely. While the translation into TRSs does not need to be modified substantially, proving termination of a TRS operating on bitvectors has not been investigated thus far. A further topic for future work is to suitably model the memory content (stack, heap, and global variables).

References

1 Elvira Albert, Puri Arenas, Michael Codish, Samir Genaim, Germán Puebla, and Damiano Zanardini. Termination analysis of Java bytecode. In *FMOODS '08*, pages 2–18, 2008.

2 Thomas Arts and Jürgen Giesl. Termination of term rewriting using dependency pairs. *TCS*, 236(1–2):133–178, 2000.

3 Aaron Bradley, Zohar Manna, and Henny Sipma. Linear ranking with reachability. In *CAV '05*, pages 491–504, 2005.

4 Marc Brockschmidt, Carsten Otto, and Jürgen Giesl. Modular termination proofs of recursive Java bytecode programs by term rewriting. In *RTA '11*, 2011.

5 Michael Colón and Henny Sipma. Synthesis of linear ranking functions. In *TACAS '01*, pages 67–81, 2001.

6 Michael Colón and Henny Sipma. Practical methods for proving program termination. In *CAV '02*, pages 442–454, 2002.

7 Byron Cook, Daniel Kroening, Philipp Rümmer, and Christoph M. Wintersteiger. Ranking function synthesis for bit-vector relations. In *TACAS '10*, pages 236–250, 2010.

8 Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Abstraction refinement for termination. In *SAS '05*, pages 87–101, 2005.

9 Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. In *PLDI '06*, pages 415–426, 2006.

10 Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Terminator: Beyond safety. In *CAV '06*, pages 415–418, 2006.

11 Stephan Falke and Deepak Kapur. Dependency pairs for rewriting with built-in numbers and semantic data structures. In *RTA '08*, pages 94–109, 2008.

12 Stephan Falke and Deepak Kapur. A term rewriting approach to the automated termination analysis of imperative programs. In *CADE '09*, pages 277–293, 2009.

13 Stephan Falke, Deepak Kapur, and Carsten Sinz. Termination analysis of C programs using compiler intermediate languages. Technical Report 2011-6, Department of Informatics, Karlsruhe Institute of Technology, Germany, 2011. Available at `http://digbib.ubka.uni-karlsruhe.de/volltexte/1000021789`.

14 Paul Feautrier and Laure Gonnord. Accelerated invariant generation for C programs with Aspic and C2fsm. *ENTCS*, 267(2):3–13, 2010.

15 Carsten Fuhs, Jürgen Giesl, Martin Plücker, Peter Schneider-Kamp, and Stephan Falke. Proving termination of integer term rewriting. In *RTA '09*, pages 32–47, 2009.

16 Jürgen Giesl, René Thiemann, and Peter Schneider-Kamp. The dependency pair framework. In *LPAR '04*, pages 301–331, 2005.

17 Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO '04*, pages 75–88, 2004.

18 Carsten Otto, Marc Brockschmidt, Christian von Essen, and Jürgen Giesl. Automated termination analysis of Java bytecode by term rewriting. In *RTA '10*, pages 259–276, 2010.

19 Andreas Podelski and Andrey Rybalchenko. A complete method for the synthesis of linear ranking functions. In *VMCAI '04*, pages 239–251, 2004.

20 Andreas Podelski and Andrey Rybalchenko. Transition invariants. In *LICS '04*, pages 32–41, 2004.

21 Robert Sedgewick. *Algorithms in C*. Addison-Wesley, 1990.

22 Fausto Spoto, Fred Mesnard, and Étienne Payet. A termination analyzer for Java bytecode based on path-length. *ACM TOPLAS*, 32(3):8:1–8:70, 2010.

23 TPDB. Termination problem data base 7.0.2, 2010. Available from `http://termcomp.uibk.ac.at/2010/downloads/`.

24 Hans Zantema. Termination. In TeReSe, editor, *Term Rewriting Systems*, chapter 6. Cambridge University Press, 2003.

First-Order Unification on Compressed Terms

Adrià Gascón[1], Sebastian Maneth[2], and Lander Ramos[1]

1 **Universitat Politècnica de Catalunya**
Jordi Girona 1-3 08034 Barcelona, Spain
adriagascon@gmail.com, landertxu@gmail.com
2 **NICTA and University of New South Wales**
Sydney, Australia
sebastian.maneth@nicta.com.au

Abstract

Singleton Tree Grammars (STGs) have recently drawn considerable attention. They generalize the sharing of subtrees known from DAGs to sharing of connected subgraphs. This allows to obtain smaller in-memory representations of trees than with DAGs. In the past years some important tree algorithms were proved to perform efficiently (without decompression) over STGs; e.g., type checking, equivalence checking, and unification. We present a tool that implements an extension of the unification algorithm for STGs. This algorithm makes extensive use of equivalence checking. For the latter we implemented two variants, the classical exact one and a recent randomized one. Our experiments show that the randomized algorithm performs better. The running times are also compared to those of unification over uncompressed trees.

Digital Object Identifier 10.4230/LIPIcs.RTA.2011.51

Category System Description

1 Introduction

Trees are a basic and very common data structure in computer science. In many applications, trees are stored in memory for fast processing. Some recent applications deal with *very large trees*. For instance, the nowadays ubiquitous data exchange format XML stores data in the form of unranked trees; typically, each data item is accompanied by several XML tree nodes describing its structure. This results in huge trees, often consisting of many millions of nodes. The problem arises that such trees do not fit into main memory, especially if stored as conventional (machine) pointer data structure. Therefore, *compressed* in-memory representations have been developed; for instance, succinct trees (see, e.g., [20]), or grammar-compressed trees [4, 17].

Here we deal with grammar-compressed trees. Grammar compression was invented for strings in the 1990s, see [19] for a survey. The idea is to find a small grammar that generates only the given string. It is a form of dictionary compression where grammar nonterminals represent repeated substrings. For instance, a smallest *context-free (cf) grammar* that generates a given string can be (at most) exponentially smaller than the given string. Finding a minimal cf grammar is NP-complete, but several well-behaved approximation algorithms exist [5]. While in general algorithms run slower when executed over a compressed representation, there are certain special algorithms which can execute in one pass (without decompression) through the grammar. This induces a *speed-up* that is proportional to the compression. For instance, testing whether two cf grammars generate the same string can be performed in cubic time with respect to the sizes of the grammars [13].

The idea of grammar-compression was generalized from strings to trees in [4], where they present an approximation algorithm that finds a small cf tree grammar. We call a cf tree

22nd International Conference on Rewriting Techniques and Applications (RTA'11).
Editor: M. Schmidt-Schauß; pp. 51–60



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

grammar that generates only one tree, a *Singleton Tree Grammar* (STG). Note that the classical idea of representing a tree by its minimal unique DAG is an instance of grammar compression: the DAG is equivalent to the minimal regular tree grammar of the tree. For typical XML documents, DAGs allow to shrink their tree structures to about 12% of the original number of edges [3]. The algorithm of [4] finds STGs that contain only 4% of the original edges. The new grammar compressor "TreeRePair" [15] compresses even further ($< 3\%$) and runs almost as fast as building a minimal DAG.

Examples of algorithms that run efficiently (without decompression) over STGs are tree automata evaluation [14], XPath query processing [17], and equivalence testing [4, 21]. STGs have also been used for complexity analysis of unification algorithms [12]. Recently, *first-order unification* was shown to be solvable in polynomial time over STGs [9, 10]. Note that an application domain for which unification over compressed terms can be useful are logic-programming languages for XML. Examples of such languages are Xcerpt [2] (it uses a form of asymmetric unification called "simulation) and Xcentric [6] (it uses the unification studied in [11]). Here, we present an implementation of the unification and matching algorithms of [10]. The algorithms run a variant of Robinson's standard unification algorithm [18] over two given STGs; it builds string grammars for the preorder traversals of the grammars, and then applies equivalence checking for singleton cf string grammars, while instantiating the encountered variables. For the equivalence check we implemented two competing algorithms: (1) the exact algorithm due to Lifshits [13], and (2) the recent randomized algorithms by Schmidt-Schauß and Schnitger [21]. Our tool is integrated with TreeRePair: it takes as input two terms represented in XML syntax and runs TreeRePair to build STGs. It then runs the unification algorithm. Through experiments we evaluate the performance of the resulting three unification algorithms and compare them to an implementation of a classical unification algorithm over uncompressed terms. Roughly speaking, unification over STGs is more efficient than over uncompressed terms, whenever the terms are well-compressible and larger than 100,000 nodes. At `www.lsi.upc.edu/~agascon/unif-stg` our system can be tested online. All our code is open source and will be available over the same web page.

2 Preliminaries

A *ranked alphabet* is a set $\mathcal{F}$ together with a function $\mathrm{ar} : \mathcal{F} \to \mathbb{N}$. Members of $\mathcal{F}$ are called function symbols, and $\mathrm{ar}(f)$ is called the *arity* of the function symbol f. Function symbols of arity 0 are called constants. Let $\mathcal{X}$ be a set disjoint from $\mathcal{F}$ whose elements have arity 0. The elements of $\mathcal{X}$ are called first-order variables. The set $\mathcal{T}(\mathcal{F} \cup \mathcal{X})$ of terms over $\mathcal{F}$ and $\mathcal{X}$, also denoted $\mathcal{T}(\mathcal{F}, \mathcal{X})$, is defined to be the smallest set having the property that $\alpha(t_1, \ldots, t_m) \in \mathcal{T}(\mathcal{F} \cup \mathcal{X})$ whenever $\alpha \in (\mathcal{F} \cup \mathcal{X})$, $m = \mathrm{ar}(\alpha)$ and $t_1, \ldots, t_m \in \mathcal{T}(\mathcal{F} \cup \mathcal{X})$. A *substitution* is a mapping $\sigma : X \to \mathcal{T}(\mathcal{F}, \mathcal{X})$ relating first-order variables to terms. Substitutions can also be applied to arbitrary terms by homomorphically extending them by $\sigma(f(t_1, \ldots, t_m)) = f(\sigma(t_1), \ldots, \sigma(t_m))$.

The *size* $|t|$ of a term t is the number of occurrences of variables and function symbols in t. The *height* of a term t, denoted $\mathrm{height}(t)$, is 0 if t is a constant or a first-order variable, and $1 + \max\{\mathrm{height}(t_1), \ldots, \mathrm{height}(t_m)\}$ if $t = \alpha(t_1, \ldots, t_m)$, with $m \geq 1$. The set $Pos(t)$ of *positions* of t is defined by $Pos(t) = \{\lambda\}$ if t is a constant or a variable, and $Pos(t) = \{\lambda\} \cup \{1 \cdot p \mid p \in Pos(t_1)\} \cup \ldots \cup \{m \cdot p \mid p \in Pos(t_m)\}$ if $t = \alpha(t_1, \ldots, t_m)$, where $m \geq 1$, λ denotes the empty sequence and $p \cdot q$, or simply pq, denotes the concatenation of p and q. If t is a term and p a position, then $t|_p$ is the subterm of t at position p. More formally defined, $t|_\lambda = t$ and $\alpha(t_1, \ldots, t_m)|_{i \cdot p} = t_i|_p$. We denote by $\mathrm{Pre}(t)$ the preorder

traversal (as a word) of a term t. It is recursively defined as $\mathrm{Pre}(t) = t$, if t has arity 0, and $\mathrm{Pre}(t) = \alpha \cdot \mathrm{Pre}(t_1) \cdot \ldots \cdot \mathrm{Pre}(t_m)$, if $t = \alpha(t_1, \ldots, t_m)$.

▶ **Definition 2.1.** A *Singleton (String) Grammar (SG)* G is a tuple $\langle \mathcal{N}, \Sigma, R \rangle$, where $\mathcal{N}$ is a finite set of non-terminals, Σ is a finite set of symbols (a signature), and R is a finite set of rules of the form $N \to \alpha$ where $N \in \mathcal{N}$ and $\alpha \in (\mathcal{N} \cup \Sigma)^*$. The sets $\mathcal{N}$ and Σ must be disjoint, and each non-terminal X appears as a left-hand side of just one rule of R. Let $N_1 >_G N_2$ for two non-terminals N_1, N_2, iff $(N_1 \to \alpha) \in R$, and N_2 occurs in α. The SG must be non-recursive, i.e. the transitive closure $>_G^+$ must be terminating. The word generated by a non-terminal N of G, denoted by $w_{G,N}$ or w_N when G is clear from the context, is the word in Σ^* reached from N by successive applications of the rules of G. SGs are also called *Straight Line Programs*.

With SG words of exponential length can be represented in linear space.

▶ **Example 2.2.** Let G be an SG with set of rules $\{A_0 \to a, A_1 \to A_0A_0, \ldots, A_n \to A_{n-1}A_{n-1}\}$. Then, $w_{A_n} = a^{2^n}$.

Let us fix a countable set $\mathcal{Y} = \{y_1, y_2, \ldots\}$ whose elements are function symbols of arity 0 called *parameters*. Given a ranked alphabet $\mathcal{F}$, we assume that $\mathcal{Y}$ and $\mathcal{F}$ are disjoint and define $\mathcal{T}(\mathcal{F} \cup \mathcal{Y})$ analogously to how $\mathcal{T}(\mathcal{F} \cup \mathcal{X})$ was defined in the preliminaries. We call the elements of $\mathcal{T}(\mathcal{F} \cup \mathcal{Y})$ *term patterns*.

▶ **Definition 2.3.** A *Singleton Tree Grammar (STG)* G is a 4-tuple $\langle \mathcal{N}, \Sigma, R, S \rangle$, where

- $\mathcal{N}$ is a ranked alphabet whose elements are called non-terminals.
- Σ is a ranked alphabet called signature.
- R is a finite set of rules of the form $N \to t$ where $N \in \mathcal{N}$, $t \in \mathcal{T}(\mathcal{F} \cup \mathcal{N} \cup \{y_1, \ldots, y_{\mathrm{ar}(N)}\})$, $t \notin \mathcal{Y}$, and each of the parameters $\{y_1, \ldots, y_{\mathrm{ar}(N)}\}$ appears in t.
- S is the initial non-terminal of rank 0.

The sets $\mathcal{N}$ and Σ must be disjoint, each non-terminal N appears as a left-hand side of just one rule of R. Let $N_1 >_G N_2$ for two non-terminals N_1, N_2, iff $(N_1 \to \alpha) \in R$, and N_2 occurs in α. The STG must be non-recursive, i.e., the transitive closure $>_G^+$ must be terminating. The *depth* of G is the maximal length of a chain in $>_G^+$. We define the derivation relation $\Rightarrow_G$ on $\mathcal{T}(\mathcal{F} \cup \mathcal{N} \cup \mathcal{Y})$ as follows: $t \Rightarrow_G t'$ iff there exists $(A \to s) \in R$ with $\mathrm{ar}(A) = n, t = C(A(t_1, \ldots, t_n))$ and $t' = C(\sigma(s))$, where $\sigma = \{y_1 \to t_1, \ldots, y_n \to t_n\}$ and C is a context, i.e. a term in $\mathcal{T}(\mathcal{F} \cup \mathcal{N} \cup \mathcal{Y} \cup \{\bullet\})$ such that $\bullet$ appears only once in the positions of C, and $C(A)$ is simply the replacement of $\bullet$ by A in C. The term pattern generated by a non-terminal N of G, denoted by $t_{G,N}$ or t_N when G is clear from the context, is the term pattern in $\mathcal{T}(\mathcal{F} \cup \mathcal{Y})$ such that $N \Rightarrow_G^* t_N$.

Note that the rules of our grammars will always have ≤ 2 occurrences of non-parameter symbols in their right-hand sides. This is not a loss of generality since every STG can be efficiently normalized to satisfy this constraint (see [16]). Hence, we define the *size* of an STG/SG G, denoted $|G|$, as its number of nonterminals. An STG is *linear* if, for every rule $(N \to t)$, the term t is linear in $\mathcal{Y}$. An STG is called *k-bounded* if every non-terminal has arity $\leq k$. Finally, an STG is called *monadic* if it is 1-bounded. As shown by the next examples, STGs can represent terms of exponential height.

▶ **Example 2.4.** Let G be a monadic (and linear) STG with the following set of rules. $\{A_a \to a, A_0(y_1) \to f(y_1), A_1(y_1) \to A_0(A_0(y_1)), A_2(y_1) \to A_1(A_1(y_1)) \ldots, A_n(y_1) \to A_{n-1}(A_{n-1}(y_1)), S \to A_n(A_a)\}$. Then t_S generates a monadic tree $f(f(\ldots(a)\ldots))$ of size $2^n + 1$.

It is not difficult to prove that, for any linear STG $G = (\mathcal{N}, \Sigma, R, S)$, it holds that $|t_S| \leq 2^{O(|G|)}$. STGs can be considered as a generalization of directed acyclic graphs in which not only repeated subterms are shared but also repeated term patterns. In fact, DAGs can be seen as 0-bounded STGs.

In this work, STGs are used in the context of first-order unification. Hence, we want to represent terms containing first-order variables. From the point of view of the grammar, every first-order variable X initially is just a terminal symbol. As will be explained in the next section, they will be transformed in non-terminals as soon as they get instantiated due to the unification process by adding a rule of the form $X \to A$, where A is a non-terminal of rank 0. We call this kind of rules λ rules.

3 First-order unification and matching

Consider a ranked alphabet $\mathcal{F}$ and a set of first-order variables $\mathcal{X}$. The first-order unification problem consists of, given two terms $s, t \in \mathcal{T}(\mathcal{F} \cup \mathcal{X})$, finding a substitution σ such that $\sigma(s)$ and $\sigma(t)$ are syntactically equal. The first-order matching problem is a particular case of first-order unification in which only one of the terms given as input may contain variables. Both first-order unification and matching are common problems in areas like functional and logic programming, automated deduction, deductive databases, and compiler design.

Our tool Unif-STG offers three algorithms for solving first-order unification where the input terms s, t are represented compressed using STGs. Moreover, as a yardstick for comparison we implemented a variant of Corbin-Bidoit [7] that uses directed acyclic graphs for term representation. All four algorithms correspond, essentially, to the schema presented in Figure 1. Note that in this schema we consider indexes in the preorder traversal words of terms instead of just positions. This is not relevant for the algorithm that works on uncompressed terms but will make a difference in the compressed case. Note that two arbitrary different trees may have the same preorder traversal, but when they represent terms over a fixed signature where the arity of every function symbol is fixed, the preorder traversal is unique for every term. Thus, we can recursively define the mapping $\text{iPos}(t, i) \to Pos(t)$ relating positions in a term with indexes in its preorder traversal word as follows: $\text{iPos}(\alpha(t_1, \ldots, t_m), 1 + |t_1| + \ldots + |t_{i-1}| + k) = i.\text{iPos}(t_i, k)$ for $1 \leq k \leq |t_i|$ and $\text{iPos}(t_i, 1) = \lambda$. This observation was crucial in [10] to improve the algorithm of first-order unification with STGs since positions of a term represented with an STG G may have exponential size w.r.t. $|G|$ and need to be compressed which makes the computation of subterms inefficient. On the other hand, computing a subterm $t|_{\text{iPos}(t,i)}$ given the index i can be done in a much more efficient way. It is important to remark that XML trees are unranked in general and hence two different trees may have the same preorder traversal. Hence, it is important to transform XML trees to ranked trees (terms) to apply the approach mentioned above. Note that the basic operations in that schema are to decide equality between two terms, apply a substitution, compute the preorder traversal word of a term, compute a subterm of a term given an index of its preorder traversal word, find the first different positions of two words, and check whether a certain symbol occurs within a term. In the setting of Unif-STG, the input terms s, t are represented using STGs. Since STGs can represent terms of exponential size, the difficulty of applying that schema to compressed terms relies on being able to solve all these subproblems in polynomial time with respect to the size of the input STG. In [9], this problem was solved in time $\mathcal{O}(|V|(m + |V|n)^4)$, where m represents the size of the input STG, n represents its depth, and V represents the set of different first-order variables occurring in the input terms. Then this result was improved in [10] to $\mathcal{O}(|V|(m + |V|n)^3)$.

```
Unify(s : term, t : term):
  σ: substitution
  σ := ∅
  While (σ(s) ≠ σ(t)):
    Look for the first position k such that Pre(σ(s))[k] ≠ Pre(σ(t))[k].
    If both Pre(σ(s))[k] and Pre(σ(t))[k] are function symbols, Then
      Return false (clash)
    // Assume w.l.g that Pre(σ(s))[k], is a variable x.
    If x occurs in σ(t)|_p, where p = iPos(σ(t), k), Then
      Return false (occur-check)
    σ := σ ∪ {x ↦ σ(t)|_p}
  EndWhile
  Return true
```

Figure 1 General schema for first-order unification

From [10], we know that the following problems can be solved in linear time:

- Given a SG/STG G, compute the number $|t_N|/|w_N|$ for every non-terminal N of G.
- Given a STG G and a non-terminal N, construct an SG of linear size for $\mathrm{Pre}(t_{G,N})$.
- Given an STG G, a non-terminal N of G, and an integer k, compute an extension G' of G such that G' generates $t_{G,N}|_{\mathrm{iPos}(t_{G,N},k)}$.

Also from [10], we know that, given two words compressed with SGs, we can find the first position in which they differ in linear time using a data structure computed by Lifshit's algorithms for checking equality. This problem can also be solved probabilistically in linear time as sketched in the following subsection. Moreover, an application of a substitution $\{X \to t\}$ is simulated by transforming X from a terminal to a non-terminal of the grammar and adding the rule $X \to N$, where the non-terminal N generates t. In this way, the grammar may be extended with at most n new non-terminals after each variable assignment as proved in [10]. Hence, the final size of the grammar is bounded by $m + |V|n$. Thus, the resulting running time corresponds to decide equality for STGs $|V|$ times on a grammar of size $\mathcal{O}(m + |V|n)$. In [10], the problem of deciding equality between two terms represented by STGs is reduced to equality between words represented by SGs. Lifshits' algorithm [13] is, with respect to big-O complexity, the most efficient known exact algorithm for checking equality between two words represented by SGs. It is cubic with respect to the sizes of the input SGs. Our three implementations of first-order unification with STGs correspond to different algorithms for solving this subproblem. Unif-STG allows the user to choose among Lifshits' algorithm and two of the recent randomized algorithms by Schmidt-Schauß and Schnitger [21]. Since our implementation of the randomized algorithms runs in linear time, the cost of first-order unification is $\mathcal{O}(|V|(m + |V|n))$ when they are used. See below where we describe these equality testing algorithms.

With respect to first-order matching with STGs, an algorithm with cost $\mathcal{O}((m + |V|n)^3)$ was presented also in [10]. The improvement with respect to the unification case relies in the fact that, in contrast to unification schema presented in Figure 1, in the matching case we just need to look for the index of first occurrence of a variable in $\mathrm{Pre}(s)$ instead of looking for the index of the first difference between $\mathrm{Pre}(s)$ and $\mathrm{Pre}(t)$ and do the corresponding assignment until every variable is replaced. At the end we just perform equivalence testing *once*. For the details of that algorithm we refer the reader to [10]. Using the randomized algorithms of [21] first-order matching can be solved in $\mathcal{O}(m + |V|n)$.

Note that in [9] and [10] only monadic grammars were considered. This is not a loss of generality since every linear STG can be transformed in polynomial time into a monadic (and linear) one [16]. However, their algorithm is rather involved and difficult to implement. We

therefore extended the unification algorithm of [10] to unbounded grammars. This mainly consists of generalizing the construction for the computation of a subterm to unbounded grammars. The solutions implemented in Unif-STG for the rest of the subproblems are straightforward adaptations of those in [10] and are not further discussed here.

Equality testing. Given an SG $G = (\mathcal{N}, \Sigma, R)$ and two non-terminals A, B, equality testing consists of deciding whether $w_A = w_B$. Let us assume that $|w_A| = |w_B|$ since otherwise inequality is easily stated in linear time. As commented above, the fastest known exact algorithm for equality testing for SGs is Lifshits' algorithm [13]. In Unif-STG we implemented, in addition to Lifshits' algorithm, two new algorithms of [21]. These algorithms run faster than Lifshits' by using a randomized approach. They work by considering an SG to generate a natural number, in addition to a word. The number coded by $w_A = w'a$, where $w' \in \Sigma^*$ and $a \in \Sigma$, is defined in terms of a fixed mapping $f : \Sigma \to \{0, \ldots, |\Sigma| - 1\}$, as $code(w_A) = code(w') * |\Sigma| + f(a)$. The main idea of the algorithm is very simple. If we want to check whether A and B represent the same word, we choose a natural number m satisfying certain properties, and compute $\alpha = code(w_A) \mod m$ and $\beta = code(w_B) \mod m$. If $\alpha \neq \beta$ then the words are obviously different. Otherwise, it is possible that $w_A \neq w_B$, but $\alpha = \beta$. In this case we do not detect inequality. In [21], two upper bounds for the choice of the m that guarantee that we detect inequality with a probability $\geq \frac{1}{2}$ for any pair of words are given: either $m \leq |w_A|^2 * c$ or $m \leq |w_A| * c$ if m is prime, for a certain constant c. We implemented both options in Unif-STG. By repeating the test k times the probability of not detecting inequality is $< \frac{1}{2^k}$. In Unif-STG the value of k is set to 10 by default.

In order to assure that the chosen m is prime we implement a simple algorithm: generate a random number, and test primality. If the number is not prime, then generate another number, and so on. We test primality with the Fermat primality test, checking if $a^{p-1} \equiv 1 \mod p$ for $a \in \{2, 3, 5, 7\}$. Due to the Prime number theorem, the average number of times we generate a number until getting a prime is the logarithm of m, and hence linear in $|G|$, and the Fermat primality test is also performed in logarithmic time.

We also need an algorithm to compute if w_A is a prefix of w_B, in order to find the first difference between two words represented with SGs (see Figure 1). This problem can be reduced to computing $code(w_B[1 \ldots |w_A|])$ and applying the probabilistic algorithm. To perform this task in linear time it is enough to precompute, for each non-terminal A of the grammar, the numbers $code(w_A)$ and $|w_A|$, and to compute $code(w_B[1 \ldots |w_A|])$ recursively.

Finally, it is important to remark a certain peculiarity of the version of the probabilistic algorithms implemented in Unif-STG. They run in linear time thanks to the fact that $|w_A|$ is limited by default to $\sqrt{L}$ where $L = 2^{64}$, the maximum value for a long long int, in the case of the algorithm using primes; and to $\sqrt[4]{L}$, in the algorithm using natural numbers. Otherwise, computing $code(w_A)$ modulo m is not guaranteed to run in linear time. The current implementation allows bigger values, but then does not guarantee an error probability of less than $\frac{1}{2}$ for every possible instance of the problem. In our experiments we never encountered a false reply by the probabilistic algorithm.

Note that Unif-STG has been built to work with arbitrary arithmetic; the size limitation has been added for efficiency reasons only and can be removed at any time.

4 Unif-STG

Unif-STG is written in C++ using the standard template library. The system implements three algorithms for solving the equality testing with SGs: Lifshits' exact algorithm, plus two versions of the randomized algorithm by Schmidt-Schauß and Schnitger (one with integers and

one with primes). We refer to the corresponding three versions of the unification algorithm for STG grammars by STG-exact, STG-rand, and STG-rand-prime. Our implementation of unification over uncompressed terms is denoted "tUnif". As commented before, this is a variant of the Corbin-Bidoit algorithm. We refer the reader to Chapter 8, Section 2.3 of [1] for the details of this algorithm. Note that Unif-STG outputs a compressed representation of the solution (again compressed with STGs).

5 Experiments

Experimental Setup. All tests are executed on a machine with Intel Xeon Core 2 Duo, 3 Ghz processor, with 4GB of RAM. We use the Ubuntu Linux 9.10 distribution, with kernel 2.6.32 and 64 bits userland. Our implementation was compiled using g++ 4.4.1. We used TreeRePair (build data 01-19-2011) which was kindly made available to us by Roy Mennicke. It is essentially the version available at `http://code.google.com/p/treerepair`, with the only difference that it allows to compress without prior applying a binary tree encoding. We run TreeRePair with the switches "-multiary -bplex -c -nodag -optimize edges" and default value (4) for maxbound. The latter means that only 4-bounded STGs are generated.

Protocol. Each test is executed three times, and the fastest time of the three runs is reported. We only measure the pure unification time, i.e, we ignore loading time and setup of basic data structures, etc.

Design of the Experiments. Instead of trying to find instances of unification problems with large terms that are realistic and likely to appear in practice, we present results over artificial examples. The idea behind these examples is to test the behaviour of our algorithms in the different extreme corner cases. The main aspects that make up these cases are (a) are the terms well-compressible by TreeRePair? (b) do they unify or not? (c) how many variables? (d) is it only matching; how much copying of variables? For question (a) we need to distinguish further: (a1) is the "top-matching part", i.e., parts where both terms do not have variables (and therefore must match exactly) well-compressible? And (a2) is the "binding part", that is, parts that will be bound to variables during unification well compressible? We constructed a family of instances that allows to test many of these aspects. First we show two simple examples which compress well.

Bin and Mon. For a natural number n, let $f^n(a,b)$ denote a full binary tree with leaf sequence $ababab\ldots$. Given a natural number n, $\mathrm{Bin}(n)$ consists of the pair of trees

$$\mathrm{Bin}(n) = (g(g(f^n(a,b), f^n(a,b)), g(f^n(a,b), f^n(a,b))), \quad g(g(X,X), g(X, f^n(a,Y)))).$$

Similarly, $f^n(a)$ denotes a monadic tree of height n with internal nodes labeled f and leaf a. The second example, called $\mathrm{Mon}(n)$ consists of this pair of trees

$$\mathrm{Mon}(n) = (h(f^n(X), f^n(Y), Y), \quad h(T,Z,T)).$$

Clearly, both Bin and Mon are unifiable for every n. Moreover, they are well compressible with TreeRePair. To see this, consider the right part of Table 1 which shows the compression time, the number of edges in the original tree and in the STG grammar, plus the file sizes of the original tree (in XML format) and of the grammar (in text format). It also shows the file size of the grammar in CNF in the special format that our unification program uses.

For both Bin and Mon, the TreeRePair algorithms achieves exponential compression rates. As can be seen, for $n > 20000$, the STG-rand algorithm is the fastest. Interestingly, for such small grammars we are punished for using prime numbers and STG-rand-prime is slower than STG-rand. This is different for larger grammars as the later examples show.

	Runtime (in ms)				Input			
n/1000	tUnif	STG-randp	STG-rand	STG-exact	edges	compr. time	STG edges	CNF file
5	**2**	8	8	24	10008 (69K)	55ms	38 (388B)	1K
10	**5**	10	8	28	20008 (137K)	62ms	40 (398B)	1.1K
20	11	11	**9**	30	40008 (157K)	140ms	42 (420B)	1.1K
50	44	11	**9**	30	100T (684K)	341ms	45 (434B)	1.2K
100	107	12	**10**	31	200T (1.4M)	681ms	47 (457B)	1.3K
200	232	13	**10**	32	400T (2.7M)	1387ms	49 (467B)	1.3K

Table 1 The example Mon(n)

	Runtime (in ms)				Input		
randSize	tUnif	STG-rp	STG-r	STG-e	edges	STG edges	CNF file
10	**3**	18	19	78	20484 (111K)	62 (578B)	1.9K
11	**7**	20	20	96	40964 (221K)	66 (596B)	2.1K
12	**16**	22	22	108	81924 (441K)	70 (638B)	2.2K
13	35	**23**	**23**	131	163844 (881K)	74 (656B)	2.3K
14	72	26	**25**	146	327684 (1.8M)	78 (698B)	2.4K
16	290	30	**28**	(*)	1310724 (6.9M)	86 (758B)	2.7K

(*) STG-exact ran out of (int) bounds.

Table 2 The example Bin(n)

Note that here the exact algorithm still shows reasonable performance. This will not be the case for larger grammars. Note that, in terms of XML, Mon is actually quite relevant: a long list of items usually becomes a long list of siblings in XML. Using the common "first-child/next-sibling"-encoding of unranked into binary trees, such a list becomes a long path, similar to Mon.

Bad Instances for STG-Unif. Here consider instances where the STG-based unification algorithm does *not* perform well. In general, this is the case when the terms are *not well compressible* (see below). But, there are even simpler reasons for this to happen. Consider unifying the trees $f(t)$ and $g(t')$ for large (arbitrary) terms t and t'. The run time of tree-based unification is only 0.005ms for this instance. While, even for highly compressible $t = t'$, STG-Unif will take >15ms. This is due to the fact that STG-Unif always needs to traverse the whole grammar to find the position of the first difference between $f(t)$ and $g(t')$ and tUnif traverses the input tree *only* until the position of the first difference is reached.

Meta. We now define a highly configurable example instance. Consider the pair of trees (t_1, t_2), where both t_1 and t_2 are full binary trees (with internal nodes labeled f) of height n. At the leaves of t_1 and t_2 appear monadic trees of random height h, with $\mathit{minHeight} \leq h \leq \mathit{maxHeight}$. These monadic trees are identical in t_1 and t_2. Now, t_1 contains variables as leaves of the monadic trees, randomly chosen from a given set *Vars* of variables. While t_2 contains random trees at those leaf positions, chosen over a given signature Σ, and maximal size of up to *randSize*. Moreover, a Boolean determines whether at variable copies we force the random trees in t_2 to be equal (which will guarantee that the instance is unifiable). Thus, the specification of an instance is as follows: Meta(n, *minHeight*, *maxHeight*, *Vars*, *randSize*, Σ, Bool U).

Number of Variables. Using Meta, we experimented with the number of different unification variables. The results were convoluted and no clear trends were observable; both algorithms seemed similarly impacted by the number of variables. For instance, for $n = 4$, $\mathit{maxHeight} = \mathit{minHeight} = 1000$, $\mathit{randSize} = 1$ and $|\Sigma| = 3$ we obtain, for 3 variables: 3ms/18ms (tUnif/STG-rand-prime), for 5 variables: 7ms/32ms, and for 10 variables: 10ms/44ms.

Incompressible Terms. An interesting case is if large incompressible terms appear

at positions that will instantiate variables. In terms of Meta, it suffices to take $n = 1$, and to use large random trees. For the other parameters we use $minHeight = maxHeight = 0$, $Vars = \{X, Y\}$, $\Sigma = \{g^{(2)}, f^{(1)}, a^{(0)}\}$, and Boolean U set to true. As Table 3 shows, STG-Unif is indeed about 100-times slower than tree-based unification. The difference in speed seems to get slightly smaller for very large inputs. As comparison, if we add a larger binary tree on top of t_1, i.e., use a larger n, then the tree becomes more compressible and therefore STG-based unification becomes efficient. This is shown in the right of Figure 2, where we pick $randSize = 20000$, but now use monadic trees of size 0–1000.

	Runtime (in ms)				Input		
randSize	tUnif	STG-rp	STG-r	STG-e	edges	STG edges	CNF file
1000	**0.1**	21	34	222	981 (5.9K)	214 (1.5K)	6.6K
5000	**0.6**	78	116	2430	4778 (29K)	774 (15K)	25K
20000	**2.4**	405	598	43632	26114 (157K)	3308 (21K)	107K
50000	**12**	1396	2074	(*)	94280 (564K)	9975 (63K)	327K
200000	**43**	5464	8036	(*)	334586 (2M)	30740 (196K)	1.1M

Table 3 Incompressible Terms in Substitution Positions

With respect to *unifiability*, we observed that changing a few nodes to make the input non-unifiable, causes STG-rand to take ca. twice the time given in Table 3, while tUnif gets slightly faster.

There are also examples where the solution consists of deeper trees than the input. This works well for the uncompressed algorithm too. But, we can see the effect of compression: consider $t_1 = h(X, Y, Z)$ and $t_2 = h(s_1, s_2, s_3)$, where s_1 is a full binary tree (over f's) of height n with all leaves labeled Y, s_2 is a full binary tree (over f's) of height n with all leaves labeled Z, and s_3 the same but with leaves labeled X. Note that X will be assigned to s_1. Hence, all the X's in s_3 will be replaced by s_1. Then, Y will be replaced by s_2 everywhere (also in s_3). So finally $t' = Y \to s_2(X \to s_1(s_3))$ will be compared to Z. Note that t' is the complete tree of depth $3 * 20$ whose leaves are all labeled Z. Since Z occurs in t' unification fails. This example is called "3-Stack" and timings are shown in Figure 2.

n	tUnif	STG-rp	STG-r	STG-e
18	99	**7**	9	35
19	200	**8**	9	38
20	401	**8**	10	(*)

n	tUnif	STG-rp	STG-r	STG-e
7	**369**	1694	2130	(*)
8	**688**	1726	2149	(*)
9	**1730**	2391	2982	(*)

Figure 2 The example 3-Stack (left) and randSize=20000 of Table 3 (right)

6 Conclusion and Further Work

Besides the rather immediate application of our work to logic programming with XML (mentioned in the Introduction), it might also be possible to apply compressed terms within theorem provers. The latter do not usually store very large trees, but store many trees. Of course, many small trees could be combined into one large tree, prior to grammar compression. However, this might induce extra costs for referencing those trees. In future work it should be studied how unification and matching (and other operations needed in theorem provers) can be applied to a set of trees represented by a cf tree grammar. Moreover, efficient updates need to be supported over such grammars. Updates on STGs have been considered in [8], but have not been implemented in any large system yet.

Acknowledgments The authors would like to thank Miguel Florido for his help during this work.

References

1 F. Baader and J.H. Siekmann. Unification theory. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming, Volume 2*, pages 41–125. Oxford Univ. Press, 1994.

2 F. Bry and S. Schaffert. Towards a declarative query and transformation language for XML and semistructured data: Simulation unification. In *ICLP*, pages 255–270, 2002.

3 P. Buneman, M. Grohe, and C. Koch. Path queries on compressed XML. In *VLDB*, pages 141–152, 2003.

4 G. Busatto, M. Lohrey, and S. Maneth. Efficient memory representation of XML document trees. *Inf. Syst.*, 33:456–474, 2008.

5 M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 51(7):2554–2576, 2005.

6 J. Coelho and M. Florido. XCentric: logic programming for XML processing. In *WIDM*, pages 1–8, 2007.

7 J. Corbin and M. Bidoit. A rehabilitation of Robinson's unification algorithm. In *IFIP Congress*, pages 909–914, 1983.

8 D. K. Fisher and S. Maneth. Structural selectivity estimation for XML documents. In *ICDE*, pages 626–635, 2007.

9 A. Gascón, G. Godoy, and M. Schmidt-Schauß. Unification with singleton tree grammars. In *RTA*, pages 365–379, 2009.

10 A. Gascón, G. Godoy, and M. Schmidt-Schauß. Unification and matching on compressed terms. *CoRR*, abs/1003.1632, 2010.

11 T. Kutsia. Unification with sequence variables and flexible arity symbols and its extension with pattern-terms. In *AISC*, pages 290–304, 2002.

12 J. Levy, M. Schmidt-Schauß, and M. Villaret. The complexity of monadic second-order unification. *SIAM J. Comput.*, 38:1113–1140, 2008.

13 Y. Lifshits. Processing compressed texts: A tractability border. In *CPM*, pages 228–240, 2007.

14 M. Lohrey and S. Maneth. The complexity of tree automata and XPath on grammar-compressed trees. *Theor. Comput. Sci.*, 363(2):196–210, 2006.

15 M. Lohrey, S. Maneth, and R. Mennicke. Tree structure compression with RePair. *CoRR*, abs/1007.5406, 2010. Short version to appear as paper in *Proc. DCC'2011*.

16 M. Lohrey, S. Maneth, and M. Schmidt-Schauß. Parameter reduction in grammar-compressed trees. In *FOSSACS*, pages 212–226, 2009.

17 S. Maneth and T. Sebastian. Fast and tiny structural self-indexes for XML. *CoRR*, abs/1012.5696, 2010.

18 J.A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12:23–41, 1965.

19 W. Rytter. Grammar compression, LZ-encodings, and string algorithms with implicit input. In *Proc. ICALP*, volume 3142 of *LNCS*, pages 15–27, 2004.

20 K. Sadakane and G. Navarro. Fully-functional succinct trees. In *SODA*, pages 134–149, 2010.

21 M. Schmidt-Schauß and G. Schnitger. Fast equality test for straight-line compressed strings. unpublished manuscript, October 2010.

Anagopos: A Reduction Graph Visualizer for Term Rewriting and Lambda Calculus

Niels Bjørn Bugge Grathwohl[1], Jeroen Ketema[2], Jens Duelund Pallesen[1], and Jakob Grue Simonsen[1]

1 **Department of Computer Science, University of Copenhagen (DIKU)**
Njalsgade 126–128, 2300 Copenhagen S, Denmark
{bugge,pallesen,simonsen}@diku.dk
2 **Faculty EEMCS, University of Twente**
PO Box 217, 7500 AE Enschede, The Netherlands
j.ketema@ewi.utwente.nl

Abstract

We present Anagopos, an open source tool for visualizing reduction graphs of terms in lambda calculus and term rewriting. Anagopos allows step-by-step generation of reduction graphs under six different graph drawing algorithms. We provide ample examples of graphs drawn with the tool.

1998 ACM Subject Classification F.3.1, F.4.2, H.5.0, I.3.0

Keywords and phrases term rewriting, lambda calculus, reduction graphs, visualization

Digital Object Identifier 10.4230/LIPIcs.RTA.2011.61

Category System Description

1 Introduction

Anagopos[1] is a tool for visualizing reduction graphs (see, e.g., Figure 1). We created the software with the following two goals in mind:

Automation Allow for the drawing of large numbers of reduction graphs, which is infeasible by hand; this will hopefully allow researchers to formulate new hypotheses regarding the topological properties of reduction graphs.

Visualization Allow for the dynamics of rewriting to be shown more clearly to students. For example, in the case of the Church-Rosser Theorem, it is often hard for students to comprehend the dynamics when moving beyond either single step reductions, or beyond the tiling diagram usually drawn in the proof for orthogonal systems.

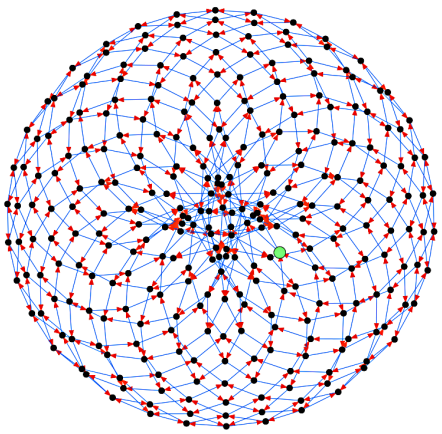


Figure 1 The sphere-like reduction graph of $(ttI)(ttI)$ with $I = \lambda x.x$ and $t = \lambda xy.\vec{y}xxy$, where $\vec{y} = \underbrace{yy\cdots y}_{15}$.

Anagopos can draw reduction graphs of both (untyped) lambda terms and terms from (first-order) term rewriting. To provide immediate

[1] Roughly a contraction of the Greek words anagogí (reduction) and tópos (place).

22nd International Conference on Rewriting Techniques and Applications (RTA'11).
Editor: M. Schmidt-Schauß; pp. 61–70
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

access to a large number of predefined term rewriting systems, the tool supports the XML-format used by the Termination Problems Data Base (TPDB) from version 7 onwards[2]. As it is a priori not clear what constitutes a good layout for a reduction graph, Anagopos supports multiple algorithms for drawing graphs.

Anagopos is implemented in Python 2.6 [23] and available for download from:

`http://code.google.com/p/anagopos/`

Packages for Ubuntu and other Debian-based systems are provided, as well as a binary for Mac OS X. The source code is available under the GNU General Public License (GPL).

Related software. The only other tool capable of drawing reduction graphs seems to be the *traces* function from PLT Redex [21]; it appears to support only a hierarchical graph drawing algorithm (displaying the initial term on the left and drawing successive reducts further and further to the right). More common are tools that compute reductions: The user specifies the starting term of a reduction and the subsequent redexes contracted are then either selected by the user, or by the program following some pre-defined reduction strategy. Roughly, these tools can be classified according to their presentation of terms, this either in some textual form [27, 37], or in the form of a parse tree [20, 24, 33, 31, 5].

Although not a software tool, also worth mentioning in this context for its graphical qualities is the Alligator Eggs puzzle game that teaches lambda calculus to children by representing lambda abstractions as alligators and variables as their eggs [36].

Outline. The paper is structured as follows: In Section 2 we provide background on reduction graphs. In Sections 3, 4, and 5 we describe, respectively, the interface, graph drawing algorithms, and architecture of Anagopos. In Section 6 we conclude, mentioning some directions in which Anagopos can potentially be extended, and suggesting a number of open problems whose solutions could potentially be found with the help of Anagopos.

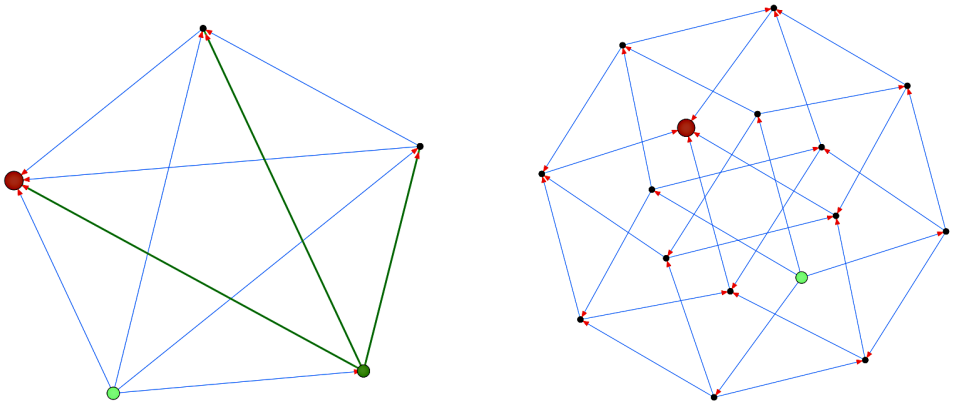
2 Reduction Graphs

Recall that an *abstract rewrite system* (ARS) consists of a set A and a set of binary relations over A; throughout this paper, we consider only a single relation at a time, writing an ARS as a pair $(A, \to)$. Also recall that the abstract reduction system induced by a term rewriting system (TRS) is obtained by letting A be the set of terms and $\to$ be the rewrite relation over the set of terms. The following is a slight adaptation of Definition 1.1.7 in [30]:

▶ **Definition 2.1.** Let $(A, \to)$ be an ARS and let $a \in A$. The *reduction graph* $\mathcal{G}(a)$ of a is the directed graph (V, E) such that V is the set of reducts of a and $(b, c) \in E$ iff $b \to c$.

The literature on reduction graphs for term rewriting systems is sparse, consisting of few general results beyond the standard confluence and Church-Rosser diagrams. For lambda calculus, many basic notions were defined, and conjectures posed, by Barendregt [2] and Klop [16]. The first comprehensive study of reduction graphs appears to be by Venturini Zilli [35, 34]. Subsequent studies by Hirokawa and Sekimoto, and Intrigila and Laurenzi have disproved several early conjectures concerning reduction graphs [26, 11]. Intrigila and Venturini Zilli have investigated representability of ordinals as reduction graphs of lambda terms [12].

[2] See `http://www.termination-portal.org/` for the database and details on the XML-format.



(a) $\mathcal{G}(k_5)$ whose underlying undirected graph is K_5.

(b) $\mathcal{G}(c_4)$ whose underlying undirected graph is the 4-dimensional hypercube.

Figure 2 The reduction graphs of two of the lambda terms defined in Proposition 2.3. Both graphs are drawn with the Neato drawing algorithm (see Section 4).

Very few general, positive results are known for reduction graphs; the following does hold:

▶ **Proposition 2.2.** *For every connected digraph G with precisely one source node, there exists a TRS R and a term t of that TRS such that $\mathcal{G}(t) = G$. If G is finite, then R may be chosen to have finitely many rules.*

Proof. Let the signature of R have a distinct, nullary function symbol for each node of G, and for any two function symbols a and b let there be a rule $a \to b$ iff the node corresponding to a has a directed edge to the node corresponding to b. Let t be the function symbol corresponding to the source node of G. Then, $\mathcal{G}(t) = G$. ◀

Clearly, by the above proposition, Anagopos requires general graph drawing algorithms to draw reduction graphs.

For lambda calculus, we have:

▶ **Proposition 2.3.** *The following (families of) graphs are realizable as the underlying undirected versions of reduction graphs of lambda terms:*

1. *For every positive natural number n, the complete undirected graph K_n on n nodes.*
2. *For every positive natural number n, the underlying undirected graph of the n-dimensional hypercube.*

Proof. We give the families of terms explicitly (see also Figure 2):

1. Fix a variable y and define $k_1 = y$, and $k_{n+1} = (\lambda x.y)\, k_n$ for all $n \geq 1$. Then, for any $n \geq 1$, a straightforward induction shows that for all $1 \leq i \leq n$: $k_{n+1} \to_\beta k_i$, whence the result follows.
2. Fix a variable y and define $s = (\lambda x.x)\, y$. Set $s^1 = s$ and $s^n = s\, s^{n-1}$ for $n > 1$. Then the set of nodes of the reduction graph of s^n are all of the form $u_1 \cdots u_n$ with $u_i \in \{s, y\}$ for all $1 \leq i \leq n$, and directed edges between all pairs $u_1 \cdots u_{i-1}\, s\, u_{i+1} \cdots u_n$ and $u_1 \cdots u_{i-1}\, y\, u_{i+1} \cdots u_n$. Clearly, the underlying undirected graph is (isomorphic) to the n-dimensional hypercube. ◀

By the above, K_5 may occur as the underlying undirected graphs of lambda terms. Thus, by Kuratowski's theorem, reduction graphs of lambda terms are not, in general, planar.

Not every digraph can occur as reduction graph in lambda calculus; an example is the digraph $\bullet \leftarrow\!\!-\!\!\bullet \longrightarrow \bullet$, which cannot occur by the Church-Rosser theorem.

In addition to the above, observe that a reduction graph need not have a lattice structure, even in case of orthogonal systems [4, 18]: Consider, for example, the reduction graph of the lambda term $I((\lambda y.Ix)z)$ in Figure 3, taken from [19] and with $I = \lambda x.x$, where the starting term occurs at the bottom; note that the set consisting of the two nodes with two successors each does not have a least upper bound.

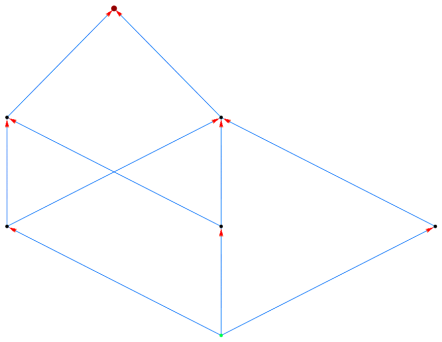


Figure 3 The graph of $I((\lambda y.Ix)z)$, as drawn with Dot (see Section 4). The starting term occurs at the bottom.

For lambda calculus and orthogonal term rewriting systems, one can instead consider another type of graph: The Hasse diagram of *reductions* with extension as ordering. As complete developments in such systems satisfy the Cube Lemma, the resulting graph is a complete lattice [4, 18]. Anagopos currently does not support visualization of these reductions.

3 Anagopos—Interface and User's Guide

The main interface of Anagopos is shown in Figure 4. On the right, the reduction graph of the current term is shown (to improve aesthetic quality self-loops are omitted). The area on the left is divided into three distinct parts:

- At the top, see Figure 5(a), we can choose between visualization of either a lambda term or a first-order term and input the term we are interested in. The syntax of the terms is as expected, except that in lambda terms λ is replaced by \. A random term can also be generated.
- In the middle, see Figure 5(b), a number of buttons occur that influence the step-by-step drawing of reduction graphs, as explained below.
- At the bottom the term is displayed that corresponds to the last node from the reduction graph selected by the user.

Graph drawing—the middle-left area. The middle-left area supports the step-by-step drawing of reduction graphs. The *Draw Graph* button will display the nodes representing the initial term and its immediate reducts. Successors of reducts are, respectively, added and removed by pressing the *Forward* and *Backward* buttons, where the numbers indicate the number of reducts that will be treated.

Of the remaining interface elements, the *Redraw Graph* button resets the positions of the vertices and re-computes the layout of the reduction graph. The *Optimize Graph* button instructs the force-directed graph drawing algorithms (see Section 4) to attempt to improve the layout. Finally, the particular graph drawing algorithm can be selected under *Select Layout Algorithm* and *Show start* and *Show newest* mark, respectively, the initial vertex of the reduction graph and the last one whose immediate reducts were computed.

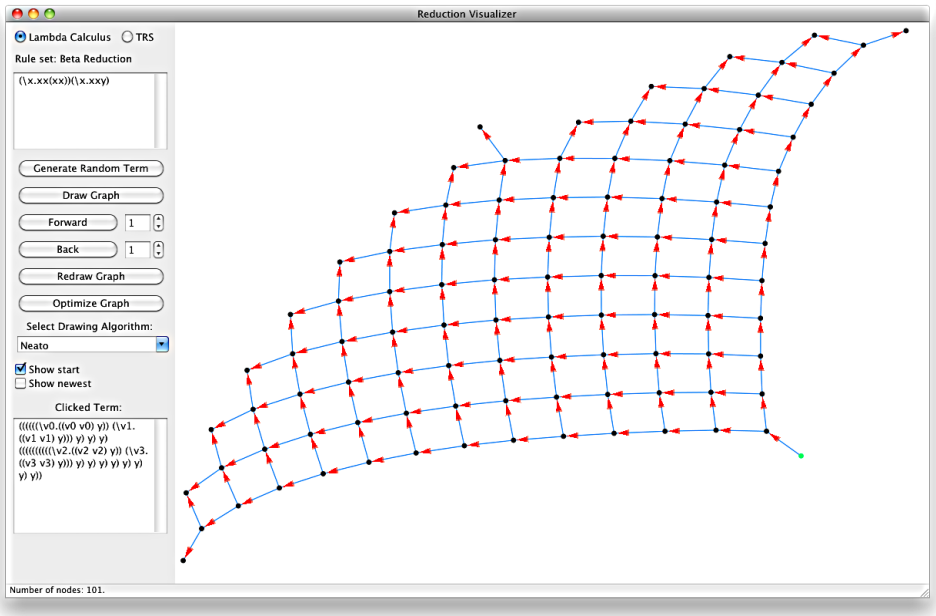
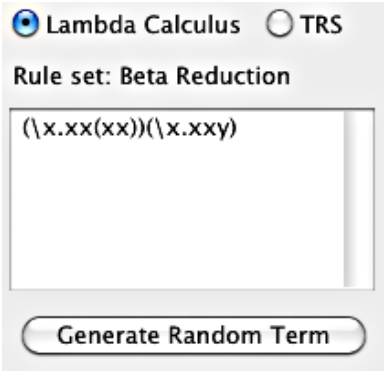
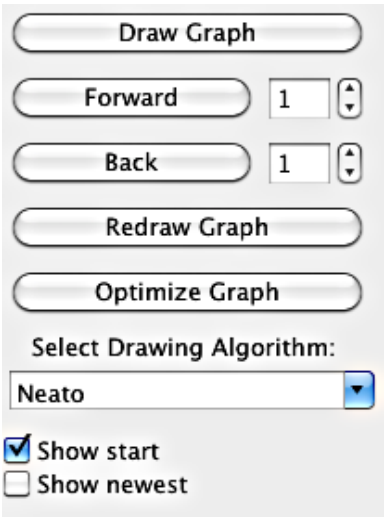


Figure 4 Anagopos displaying part of $\mathcal{G}((\lambda x.xx(xx))(\lambda x.xxy))$ using Neato (see Section 4).



(a) The top-left area.



(b) The middle-left area.

Figure 5 Close-ups of two parts of the user interface.

4 Anagopos—Graph Drawing Algorithms

Anagopos supports six general graph drawing algorithms, including three variations from the class of so-called *force-directed* algorithms, the current *de facto* standard in general graph drawing; all algorithms in this class draw graphs by employing minimization methods involving mechanical attraction and repulsion of nodes. For background on these drawing algorithms and others, including the ones mentioned below, we refer the reader to the relevant survey literature [3, 10, 14].

The six supported graph drawing algorithms are as follows, where we refer the reader to Figure 6 to get a taste of the drawings produced by each of the algorithms:

Neato and Neato Animated These are, respectively, the force-directed algorithm from [13, 8] and a slight variation with an animation-like appearance, which also draws the graph at intermediate stages of the minimization taking place.

Fdp This is an implementation of the force-directed graph drawing algorithm from [7].

Dot Constructs a hierarchical layout using Bézier curves for edges.

Circo and Twopi These are, respectively, an implementation of the algorithm from [28, 15], placing nodes on a circle, and the algorithm from [38], placing nodes on several concentric circles.

Implementation Details. Except for the Neato Animated algorithm, which we implemented ourselves to allow for its animation-like appearance, we draw heavily on the *GraphViz* graph visualization library [1], providing off-the-shelf implementations of all mentioned algorithms. This too explains the choice of the drawing algorithms.

To facilitate the implementation of the Neato Animated algorithm, we also implemented the highly efficient algorithm from [25] for solving the distance version of the all-pairs-shortest-path problem. As many graph drawing algorithms (and also other graph related tasks) depend on finding the graph-theoretical distance between all pairs of nodes, we expect this algorithm will find further uses in future extensions of Anagopos.

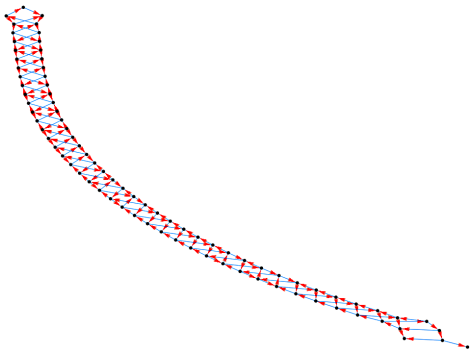
Anagopos lends itself to easy addition of new graph drawing algorithms; one is only required to implement a very simple interface `DrawingAlgorithm` through which (a) a graph can be passed to the drawing algorithm, and through which (b) the algorithm can be told to compute a layout for the given graph.

5 Anagopos—Architecture

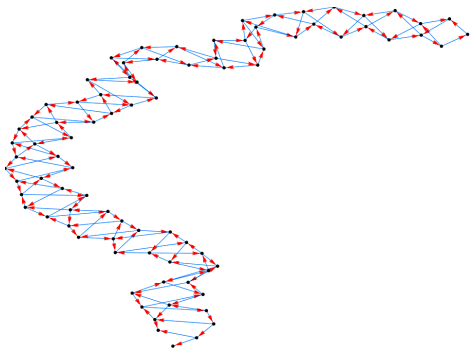
Anagopos is implemented in Python 2.6 [23], making heavy use of the object-oriented features of the language. The tool has a Model-View-Controller architecture [17], separating the model (the reduction graph of a term) from the manipulation of the model (user input) and the presentation of the model (display of the reduction graph).

The interface is implemented using wxWidgets [29], a cross-platform GUI library. The various parsers of Anagopos (for parsing TRSs, first-order terms, and lambda terms), are simple and implemented either by hand or using pyparsing [22]. The only exception is the TPDB parser which is constructed around the Expat XML parser [6].

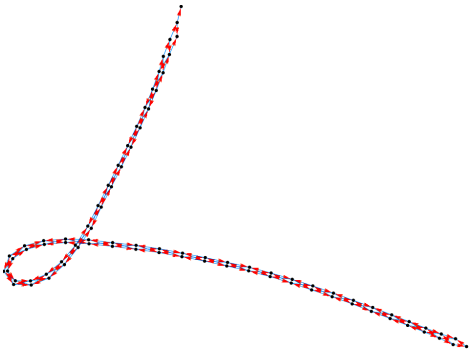
Internally, terms are represented as DAGs; bound variables of lambda terms are canonized to ensure alpha equivalent terms have a unique representation (canonization retains information on unbound variables within the term structure). Reduction graphs are represented as instances of a custom-made `Graph`-class.



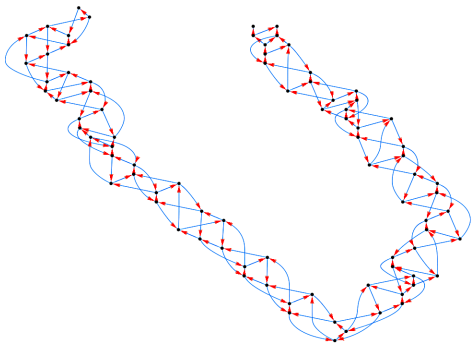
(a) Neato: Attempts to ensure that neighboring nodes are at equal distance from each other.



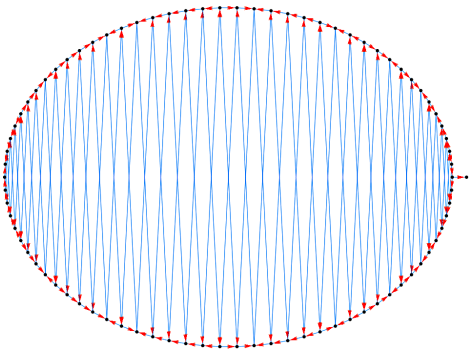
(b) Neato Animated: As Neato, but with optimizations animated. The picture shows the graph at a stage where it is not fully optimized.



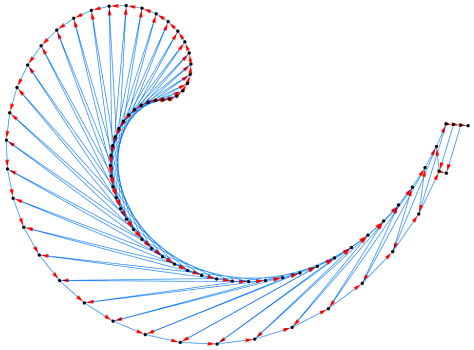
(c) Fdp: Produces a graph comparable to Neato, but in this case reaches a local minimum with a loop-like quality.



(d) Dot: Attempts to create a hierarchy, which proves difficult in this case due to the large number of cycles in the reduction graph.

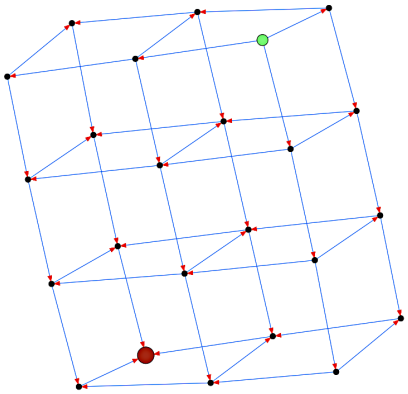


(e) Circo: Places all nodes of the reduction graph on a circle.

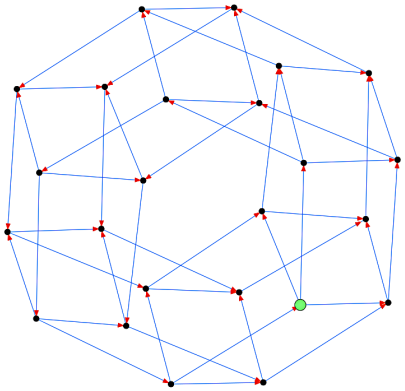


(f) Twopi: Places the nodes on several concentric circles.

Figure 6 A partial reduction graph of the lambda term HIH from [2, Exercise 3.5.5(i)], with $H = \lambda xy.x(\lambda z.yzy)x$ and $I = \lambda x.x$, as drawn with each of the supported drawing algorithms.



(a) The reduction graph of the term $((\lambda x.x)y)((\lambda x.xxx)\lambda y.y)((\lambda x.xx)\lambda y.y)$.



(b) $\mathcal{G}((ttI)c_2)$ with $t = \lambda xy.yyyyxxy$, $I = \lambda x.x$, and c_2 as in Proposition 2.3.

Figure 7 The "poor man's 3D-effect" visible in some of the reduction graphs drawn with Neato.

The reduction graphs are generated in a breadth-first fashion. Currently, Anagopos contracts at most 10^6 redexes per graph; this value is easily changed, but GraphViz performance becomes problematic for graphs of larger size.

6 Conclusion and Future Work

We have presented Anagopos, a tool for drawing reduction graphs of both lambda terms and terms from term rewriting. While we regard Anagopos in its current incarnation mainly as a tool for education and leisure, we believe that the tool may prove useful for hypothesis formation and, possibly, the formulation of proofs of new results concerning reduction graphs. As an appetizer, we mention the following problems:

- The set of reduction graphs of the set of terms of a TRS with finite signature is recursively enumerable. *Which recursively enumerable classes of finite graphs are realizable as the set of reduction graphs of a terminating TRS*? And, if we drop the requirement that graphs need to be finite, *which classes of countable graphs are realizable as the set of reduction graphs of a TRS*?
- *Precisely which* un*directed graphs can be realized as the underlying undirected graphs of lambda terms*?

Anagopos is open source and easily extended. Based on our initial experimentation with the tool, it would be obvious to include support for some general higher-order rewriting format and to implement zooming, panning, and manual rearrangements of the nodes of graphs. Furthermore, considering the tantalizing "poor man's 3D-effect" of force-directed algorithms such as Neato (see Figure 7), it seems natural to try to generate three-dimensional representations of the graphs combined with rotation. Moreover, use could be made of a general-purpose visualization framework like Tulip [32], which not only allows for easy visualization of graphs, but also of meta-data concerning the graphs. Finally, while we have considered only the most basic notion of a reduction graph, as induced by the "raw" rewrite relation of lambda calculus and TRSs, the standard literature on rewriting also considers equivalence relations over reductions (e.g., in orthogonal TRSs) [30, Chapter 8]; it would be natural if Anagopos could also take an equivalence relation as input and generate

the resulting graphs. This could include equivalence relations designed specifically to give insight into the global structure of reduction graphs; something which has already shown its usefulness in the area of state space visualization [9].

Acknowledgments The authors thank Jan Willem Klop, Peter Sestoft, and the anonymous referees for useful feedback.

References

1 AT&T Research. Graphviz – graph visualization software. Available from: `http://www.graphviz.org/`. Accessed 31 December 2010.

2 H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. Elsevier Science, revised edition, 1984. First edition 1981.

3 G. D. Battista, P. Eades, R. Tamassia, and I. G. Tollis. Algorithms for drawing graphs: an annotated bibliography. *Computational Geometry: Theory and Applications*, 4(5):235–282, 1994.

4 G. Berry and J.-J. Lévy. Minimal and optimal computations of recursive programs. *Journal of the ACM*, 26:148–175, 1979.

5 J. Endrullis. Graphical lambda calculator. Available from: `http://joerg.endrullis.de/lambdaCalculator.html`. Accessed 31 December 2010.

6 Expat maintainers. The Expat XML parser library. Available from: `http://expat.sourceforge.net/`. Accessed 5 January 2011.

7 T. M. J. Fruchterman and E. M. Reingold. Graph drawing by force-directed placement. *Software – Practice and Experience*, 21(11):1129–1164, 1991.

8 E. R. Gansner, Y. Koren, and S. North. Graph Drawing by Stress Majorization. In J. Pach, editor, *Proceedings of the 12th International Symposium on Graph Drawing (GD 2004)*, volume 3383 of *Lecture Notes in Computer Science*, pages 239–250, 2004.

9 J. F. Groote and F. van Ham. Interactive visualization of large state spaces. *International Journal on Software Tools for Technology Transfer*, 8(1):77–91, 2006.

10 I. Herman, G. Melançon, and M. S. Marshall. Graph visualization and navigation in information visualization: a survey. *IEEE Transactions on Visualization and Computer Graphics*, 6(10):24–43, 2000.

11 B. Intrigila and A. R. Laurenzi. Two problems on reduction graphs in lambda calculus. *Fundamenta Informaticae*, 44(1-2):133–144, 2000.

12 B. Intrigila and M. Venturini Zilli. Orders, reduction graphs and spectra. *Theoretical Computer Science*, 212(1-2):211–231, 1999.

13 T. Kamada and S. Kawai. An algorithm for drawing general undirected graphs. *Information Processing Letters*, 31(1):7–15, 1989.

14 M. Kaufmann and D. Wagner, editors. *Drawing Graphs: Methods and Models*, volume 2025 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.

15 M. Kaufmann and R. Wiese. Maintaining the mental map for circular drawings. In S. G. Kobourov and M. T. Goodrich, editors, *Proceedings of the 10th International Symposium on Graph Drawing (GD 2002)*, volume 2528 of *Lecture Notes in Computer Science*, pages 12–22, 2002.

16 J. W. Klop. Reduction cycles in combinatory logic. In J. Seldin and J. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 193–214. Academic Press, 1980.

17 G. E. Krasner and S. T. Pope. A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. *Journal of Object Oriented Programming*, 1:26–49, August/September 1988.

18 J.-J. Lévy. *Réductions correctes et optimales dans le lambda-calcul*. PhD thesis, Université de Paris VII, 1978.

19 J.-J. Lévy. Generalized finite developments. In Y. Bertot, G. Huet, J.-J. Lévy, and G. Plotkin, editors, *From Semantics to Computer Science: Essys in Honour of Gilles Kahn*, pages 185–204. Cambridge University Press, 2009.

20 S. Lippi. in^2 : A graphical interpreter for interaction nets. In S. Tison, editor, *Proceedings of the 13th International Conference on Rewriting Techniques and Applications (RTA 2002)*, volume 2378 of *Lecture Notes in Computer Science*, pages 380–386, 2002.

21 J. Matthews, R. B. Findler, M. Flatt, and M. Felleisen. A visual environment for developing context-sensitive term rewriting systems. In V. van Oostrom, editor, *Proceedings of the 15th International Conference on Rewriting Techniques and Applications (RTA 2004)*, volume 3091 of *Lecture Notes in Computer Science*, pages 301–311, 2004.

22 P. McGuire. pyparsing. Available from: `http://pyparsing.wikispaces.com/`. Accessed 5 January 2011.

23 Python Software Foundation. Python programming language. Available from: `http://www.python.org/`. Accessed 4 January 2011.

24 D. Ruiz and M. Villaret. TILC: The Interactive Lambda-Calculus Tracer. *Electronic Notes in Theoretical Computer Science*, 248:173–183, 2009.

25 R. Seidel. On the all-pairs-shortest-path problem. In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing (STOC'92)*, pages 745–749, 1992.

26 S. Sekimoto and S. Hirokawa. One-step recurrent terms in lambda-beta-calculus. *Theoretical Computer Science*, 56:223–231, 1988.

27 P. Sestoft. Standard ML on the web server: Visualizing lambda calculus reduction. Technical report, Royal Veterinary and Agricultural University, Denmark, 1996.

28 J. M. Six and I. G. Tollis. A framework for circular drawings of networks. In J. Kratochvíl, editor, *Proceedings of the 7th International Symposium on Graph Drawing (GD'99)*, volume 1731 of *Lecture Notes in Computer Science*, pages 107–116, 1999.

29 J. Smart, R. Roebling, et al. wxWidgets – cross-platform GUI library. Available from: `http://www.wxwidgets.org/`. Accessed 5 January 2011.

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

31 M. Thyer. Lambda Animator: animated reduction of the lambda calculus. Available from: `http://thyer.name/lambda-animator/`. Accessed 31 December 2010.

32 University of Bordeaux I. Tulip: Data Visualization Software. Available from: `http://www.tulip-software.org/`. Accessed 28 March 2011.

33 D. van den Eijkel. Animated Lambda Calculus Evaluator (ALCE). Available from: `http://substitut-fuer-feinmotorik.net/projects/animated-lambda-calculus-evaluator`. Accessed 31 December 2010.

34 M. Venturini Zilli. Cofinality in reduction graphs. In G. Ausiello and M. Protasi, editors, *Proceedings of the 8th Colloquium on Trees in Algebra and Programming (CAAP'83)*, volume 159 of *Lecture Notes in Computer Science*, pages 405–416, 1983.

35 M. Venturini Zilli. Reduction graphs in the lambda calculus. *Theoretical Computer Science*, 29:251–275, 1984.

36 B. Victor. Alligator Eggs – a puzzle game. Available from: `http://worrydream.com/AlligatorEggs/`. Accessed 31 December 2010.

37 F. Wiedijk. Untyped lambda calculus. Available from: `http://www.cs.ru.nl/~freek/notes/lambda.ml`. Accessed 31 December 2010.

38 G. J. Wills. NicheWorks – interactive visualization of very large graphs. In G. D. Battista, editor, *Proceedings of the 5th International Symposium on Graph Drawing (GD'97)*, volume 1353 of *Lecture Notes in Computer Science*, pages 403–414, 1997.

Maximal Completion*

Dominik Klein[1] **and Nao Hirokawa**[1]

1 **School of Information Science**
Japan Advanced Institute of Science and Technology
Nomi, Japan
{dominik.klein,hirokawa}@jaist.ac.jp

Abstract

Given an equational system, completion procedures compute an equivalent and complete (terminating and confluent) term rewrite system. We present a very simple and efficient completion procedure, which is based on MaxSAT solving. Experiments show that the procedure is comparable to recent powerful completion tools.

1998 ACM Subject Classification F.4.2 Grammars and Other Rewriting Systems

Keywords and phrases Term Rewriting, Knuth-Bendix Completion, Multi-completion

Digital Object Identifier 10.4230/LIPIcs.RTA.2011.71

Category System Description

1 Introduction

Completion tries to construct from an equational system (ES) an equivalent complete (confluent and terminating) term rewrite system (TRS). The standard completion procedure by Knuth and Bendix [14] and Huet [12] takes not only a target ES but also a reduction order as a parameter. This reduction order is used to ensure termination of a resulting complete TRS. Because the choice of a reduction order is critical for getting a successful run of the procedure, several attempts have been made to automatically find such an order. Here we mention the pioneering work of Kurihara and Kondo [16] on running completion using multiple orders in parallel and the approach by Wehrman et al. [23] to automatically construct a reduction order using a termination tool on the fly. Very recently Sato et al. [21] showed how both approaches can be combined.

We present a new completion procedure, dubbed maximal completion. This procedure induces a set of (exponentially) many TRSs to find a desired complete TRS from the set. Via a natural encoding into maximal satisfiability problems, the procedure can be easily implemented by a MaxSAT (or MaxSMT) solver. Experiments by our completion tool Maxcomp show that this approach performs comparable with the above approaches. The tool Maxcomp is available at:

http://www.jaist.ac.jp/project/maxcomp/

This paper is concerned with constructing *complete* term rewriting systems only. But we anticipate that our approach can be adapted for unfailing completion [6], which gives up the aim of trying to construct a complete system. Instead it only aims to construct a *ground*

* The research described in this paper is supported by the Grant-in-Aid for Young Scientists (B) 22700009 of the Japan Society for the Promotion of Science.

22nd International Conference on Rewriting Techniques and Applications (RTA'11).
Editor: M. Schmidt-Schauß; pp. 71–80
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

complete (ground terminating and ground confluent) system, which is effectively used in first-order theorem proving.

The paper is structured as follows: In Section 2 we introduce maximal completion, and its automation techniques are described in Section 3. Section 4 relates the procedure to existing ones. Empirical results are reported in Section 5, where our tool Maxcomp is compared with the state-of-art completion tools Slothrop [23] and mkbTT [25]. Finally, we conclude the presentation in Section 6 by mentioning potential future work. Throughout the paper, we assume familiarity with term rewriting in general, and most notions and notations are borrowed from [3, 22].

2 Maximal Completion

A TRS is *complete* if it is terminating and confluent. We say that $\mathcal{R}$ is a complete TRS *for* an ES $\mathcal{E}$ if $\mathcal{R}$ is a complete TRS with $\leftrightarrow_{\mathcal{R}}^* = \leftrightarrow_{\mathcal{E}}^*$. The completion problem is to find a complete TRS for a given $\mathcal{E}$.

To derive a procedure for completion, we recall the definition of *critical pairs*. An *overlap* $(\ell_1 \to r_1, p, \ell_2 \to r_2)_\mu$ of a TRS $\mathcal{R}$ consists of variants $\ell_1 \to r_1$ and $\ell_2 \to r_2$ of rewrite rules of $\mathcal{R}$ without common variables, a non-variable position $p \in \mathcal{P}\mathsf{os}(\ell_2)$, and a most general unifier μ of ℓ_1 and $\ell_2|_p$. If $p = \epsilon$ then we require that $\ell_1 \to r_1$ and $\ell_2 \to r_2$ are not variants of the same rewrite rule. The induced *critical pair* is $\ell_2\mu[r_1\mu]_p \approx r_2\mu$, and the set of all such pairs of $\mathcal{R}$ is written as $CP(\mathcal{R})$. Note that pairs (s, t) of terms are denoted by $s \approx t$ or $s \to t$ depending on the contexts. Below, we write $\downarrow_{\mathcal{R}}$ for the joinability relation $\to_{\mathcal{R}}^* \cdot {}_{\mathcal{R}}^*\!\leftarrow$.

▶ **Lemma 1.** *$\mathcal{R}$ is a complete TRS for an ES $\mathcal{E}$ if and only if $\mathcal{R}$ is terminating, $\mathcal{R} \subseteq \leftrightarrow_{\mathcal{E}}^*$, and $\mathcal{E} \cup CP(\mathcal{R}) \subseteq \downarrow_{\mathcal{R}}$.*

Proof. For the "if"-direction, by Knuth and Bendix' confluence criterion [14, 11], confluence of $\mathcal{R}$ follows from $CP(\mathcal{R}) \subseteq \downarrow_{\mathcal{R}}$ and termination of $\mathcal{R}$. Moreover, $\mathcal{E} \subseteq \downarrow_{\mathcal{R}}$ and $\mathcal{R} \subseteq \leftrightarrow_{\mathcal{E}}^*$ yield $\leftrightarrow_{\mathcal{R}}^* = \leftrightarrow_{\mathcal{E}}^*$. The "only if"-direction is immediate from $\leftrightarrow_{\mathcal{E}}^* \subseteq \leftrightarrow_{\mathcal{R}}^* \subseteq \downarrow_{\mathcal{R}}$. ◀

Lemma 1 yields a simple completion procedure. Let $\mathcal{E}$ be an ES. We assume that two parameter functions $\mathfrak{R}$ and S are given and the next two conditions hold for every ES $\mathcal{C}$: $S(\mathcal{C})$ is a set of equalities in $\leftrightarrow_{\mathcal{E}}^*$, and $\mathfrak{R}(\mathcal{C})$ is a set of terminating TRSs $\mathcal{R}$ with $\mathcal{R} \subseteq \leftrightarrow_{\mathcal{E}}^*$.

▶ **Definition 2.** Given ESs $\mathcal{E}$ and $\mathcal{C}$, the procedure φ is defined as

$$\varphi(\mathcal{C}) = \begin{cases} \mathcal{R} & \text{if } \mathcal{E} \cup CP(\mathcal{R}) \subseteq \downarrow_{\mathcal{R}} \text{ for some } \mathcal{R} \in \mathfrak{R}(\mathcal{C}) \\ \varphi(\mathcal{C} \cup S(\mathcal{C})) & \text{otherwise} \end{cases}$$

Note that $\varphi(\mathcal{C})$ is neither unique nor defined in general.

▶ **Theorem 3.** *$\varphi(\mathcal{E})$ is a complete TRS for an ES $\mathcal{E}$, if it is defined.*

The procedure φ repeatedly expands $\mathcal{C}$ (initially $\mathcal{E}$) by $S(\mathcal{C})$ until $\mathfrak{R}(\mathcal{C})$ contains a complete TRS for $\mathcal{E}$. For its success the choice of $\mathfrak{R}(\mathcal{C})$ and $S(\mathcal{C})$ is crucial. Let $t{\downarrow_{\mathcal{R}}}$ denote a fixed normal form of t with respect to $\mathcal{R}$. We say that a TRS $\mathcal{R}$ is *over* an ES $\mathcal{C}$ if $\mathcal{R} \subseteq \mathcal{C} \cup \mathcal{C}^{-1}$. The set of all terminating TRSs over $\mathcal{C}$ is denoted by $\mathfrak{T}(\mathcal{C})$. We propose to use

$$\mathfrak{R}(\mathcal{C}) = \mathit{Max}\,\mathfrak{T}(\mathcal{C})$$
$$S(\mathcal{C}) = \bigcup_{\mathcal{R} \in \mathfrak{R}(\mathcal{C})} \{s{\downarrow_{\mathcal{R}}} \approx t{\downarrow_{\mathcal{R}}} \mid s \approx t \in \mathcal{E} \cup CP(\mathcal{R}) \text{ and } s{\downarrow_{\mathcal{R}}} \neq t{\downarrow_{\mathcal{R}}}\}$$

Here *Max* computes all maximal sets of rewrite rules (called *maximal* TRSs) in its given family of TRSs, and this is the reason that we call our method *maximal* completion. Part (b) in the next lemma explains why *non*-maximal TRSs in $\mathfrak{T}(\mathcal{R})$ can be ignored safely.

▶ **Lemma 4.** *The following claims hold:*

(a) Let $\mathcal{R} \subseteq \mathcal{R}' \subseteq \leftrightarrow_{\mathcal{E}}^*$ and $\mathcal{R}'$ terminating. $\mathcal{R}'$ is complete for $\mathcal{E}$ if $\mathcal{R}$ is complete for $\mathcal{E}$.
(b) $\mathcal{E} \cup CP(\mathcal{R}) \subseteq \downarrow_{\mathcal{R}}$ for some $\mathcal{R} \in \mathfrak{T}(\mathcal{C})$ iff $\mathcal{E} \cup CP(\mathcal{R}) \subseteq \downarrow_{\mathcal{R}}$ for some $\mathcal{R} \in \mathfrak{R}(\mathcal{C})$.

Proof. (a) Due to completeness of $\mathcal{R}$, we have $\mathcal{E} \cup CP(\mathcal{R}') \subseteq \downarrow_{\mathcal{R}}$. The claim follows together with $\to_{\mathcal{R}} \subseteq \to_{\mathcal{R}'}$. (b) The 'only if'-direction is straightforward from the first claim, and the converse is trivial. ◀

We illustrate maximal completion with an example. In examples the inverse $t \approx s$ of an indexed rule $i\colon s \approx t$ is denoted as i'.

▶ **Example 5.** Consider the ES $\mathcal{E}$ consisting of the equalities:

$$1\colon\ \mathsf{s}(\mathsf{p}(x)) \approx x \qquad 2\colon\ \mathsf{p}(\mathsf{s}(x)) \approx x \qquad 3\colon\ \mathsf{s}(x) + y \approx \mathsf{s}(x + y)$$

We compute $\varphi(\mathcal{E})$ with the above $S(\mathcal{C})$ and $\mathfrak{R}(\mathcal{C})$.

(i) $\mathfrak{R}(\mathcal{E})$ consists of two TRSs $\{1,2,3\}$ and $\{1,2,3'\}$. Since the join-condition of φ does not hold, we have $\varphi(\mathcal{E}) = \varphi(\mathcal{E} \cup S(\mathcal{E}))$. Here $S(\mathcal{E})$ consists of two equalities:

$$4\colon\ x + y \approx \mathsf{s}(\mathsf{p}(x) + y) \qquad 5\colon\ \mathsf{p}(\mathsf{s}(x) + y) \approx x + y$$

(ii) $\mathfrak{R}(\{1,\dots,5\})$ consists of the two TRSs $\{1,2,3,4',5\}$ and $\{1,2,3',4',5\}$. Again the join-condition does not hold. Thus, $\varphi(\{1,\dots,5\}) = \varphi(\{1,\dots,9\})$, where

$$6\colon\ (x + y) + z \approx \mathsf{s}((\mathsf{p}(x) + y) + z) \qquad 7\colon\ \mathsf{p}((\mathsf{s}(x) + y) + z) \approx (x + y) + z$$
$$8\colon\ \mathsf{p}(x) + y \approx \mathsf{p}(x + y) \qquad 9\colon\ \mathsf{p}((x + y) + z) \approx (\mathsf{p}(x) + y) + z$$

(iii) $\mathfrak{R}(\{1,\dots,9\})$ consists of four TRSs including the TRS $\mathcal{R}$ of

$$\{1,2,3,4',5,6',7,8,9'\}$$

which satisfies the join-condition. Thus, $\varphi(\{1,\dots,9\}) = \mathcal{R}$.

Hence, $\varphi(\mathcal{E}) = \mathcal{R}$ and it is a complete TRS for $\mathcal{E}$.

Very often a complete TRS resulting from maximal completion contains many superfluous rules. It is known that this problem is resolved by computing reduced TRSs (cf. [12]). A TRS $\mathcal{R}$ is *reduced* if $\ell \in NF(\mathcal{R} \setminus \{\ell \to r\})$ and $r \in NF(\mathcal{R})$ for all rules $\ell \to r \in \mathcal{R}$. We write $\widehat{\mathcal{R}}$ for the reduced TRS

$$\{\ell \to r \in \widetilde{\mathcal{R}} \mid \ell \in NF(\widetilde{\mathcal{R}} \setminus \{\ell \to r\})\}$$

where $\widetilde{\mathcal{R}} = \{\ell \to r{\downarrow_{\mathcal{R}}} \mid \ell \to r \in \mathcal{R}\}$. The TRS $\widehat{\mathcal{R}}$ fulfills the desired property:

▶ **Lemma 6.** *If a TRS $\mathcal{R}$ is complete for $\mathcal{E}$, then $\widehat{\mathcal{R}}$ is complete for $\mathcal{E}$.*

Proof. Using the fact that $\widehat{\mathcal{R}}$ is complete and $\leftrightarrow_{\mathcal{R}}^* = \leftrightarrow_{\widehat{\mathcal{R}}}^*$ (see [19]). ◀

▶ **Example 7** (continued from Example 5)**.** The reduced version $\widehat{\mathcal{R}}$ is $\{1,2,3,8\}$.

As Example 5 illustrates, maximality dismisses undesirable complete TRSs like empty or singletons in $\mathfrak{T}(\mathcal{C})$. This is one major source of efficiency in maximal completion. We refer to the subsequent two sections for further discussion on $\mathfrak{R}(\mathcal{C})$ and $S(\mathcal{C})$.

3 Automation

We describe how to automate the approach of Section 2.

3.1 Computing $\mathfrak{R}(\mathcal{C})$

Since termination is undecidable, for automation we compute maximal elements in the set of TRSs over a given $\mathcal{C}$, for which we can show termination with reduction orders automatically. However, since there are exponentially many TRSs over $\mathcal{C}$ in general, it is impractical to check termination of each of them to compute maximal elements. We present a solution using MaxSAT solving.

In last years, SAT/SMT-encodings of termination conditions based on existing subclasses of reduction orders have been extensively investigated, and today they are well-established. Here we mention recursive path orders [17, 7], Knuth-Bendix orders [26] and orders based on matrix interpretations [9]. Importantly, all of them can test the existence of a reduction order $>$ that satisfies arbitrary Boolean combinations of order constraints:

$$C ::= s > t \mid \top \mid \bot \mid \neg\, C \mid C \vee C \mid C \wedge C$$

We exploit this fact to encode a maximal termination problem into a maximal satisfiability problem. Even though NP-hard in general, nowadays solving can be efficiently done by SMT solvers.

Computing maximal terminating TRSs is done iteratively: Given a set of equalities $\mathcal{C}$, assume we already found k maximal terminating TRSs $\mathcal{R}_1, \dots, \mathcal{R}_k$ over $\mathcal{C}$. We construct the following optimization problem ψ:

$$\text{Maximize} \quad \bigvee_{s \approx t \in \mathcal{C}} (s > t) \oplus (t > s) \quad \text{subject to} \quad \bigwedge_{i=1}^{k} \bigvee_{\ell \to r \in (\mathcal{C} \cup \mathcal{C}^{-1}) \setminus \mathcal{R}_i} \ell > r$$

where $C_1 \oplus C_2$ stands for the exclusive-or $(C_1 \wedge (\neg C_2)) \vee ((\neg C_1) \wedge C_2)$. Since each $\ell > r$ can be encoded w.r.t. a particular class of reduction orders to Boolean constraints, ψ can be treated as an instance of MaxSAT/MaxSMT. A solution yields a maximal subset of oriented equalities from $\mathcal{C}$, that forms a terminating TRS $\mathcal{R}_{k+1}$ and is different from all $\mathcal{R}_1, \dots, \mathcal{R}_k$. If ψ is unsatisfiable, we found all maximal terminating TRSs over $\mathcal{C}$ (w.r.t. the considered reduction order) and return $\{\mathcal{R}_1, \dots, \mathcal{R}_k\}$. Otherwise, we re-encode ψ w.r.t. $\mathcal{R}_{k+1}$ for another MaxSAT/MaxSMT-instance.

Finally, in our implementation we do not compute all maximal terminating TRSs. This is because there still may be exponentially many maximal terminating TRSs. Instead, we fix a number K to stop the enumeration of maximal terminating TRSs whenever the number reaches K. This is motivated by the following observation: Assume that there exists a complete TRS $\mathcal{R} \in \mathfrak{R}(\mathcal{C})$, but we fail to select it. Since $\mathcal{R}$ is a terminating TRS over $\mathcal{C} \cup S(\mathcal{C})$, by Lemma 4 (a) there exists a maximal terminating, complete TRS $\mathcal{R}' \in \mathfrak{R}(\mathcal{C} \cup S(\mathcal{C}))$ with $\mathcal{R} \subseteq \mathcal{R}'$. Thus when missing the complete TRS $\mathcal{R}$ in one iteration, there is still a chance to select $\mathcal{R}'$ in the next one.

3.2 Filtering $S(\mathcal{C})$

Our implementation of the parameter function $S(\mathcal{C})$ follows closely the proposed one of Section 2 but adds a few small operations as described below.

When orienting equalities to rules, some equalities tend to generate a lot of critical pairs. This is why Knuth-Bendix completion employs selection heuristics (cf. [3, 25, 23]) that select

only certain kinds of equalities. We also heuristically select equalities, since otherwise the number of critical pairs grows too fast and our implementation fails to handle it. In order to address it, we first normalize the equalities to filter out all those whose size exceeds a bound d. Then, we select n smallest equalities. We formulate this filtering. For a set of equalities $\mathcal{C}$, we write $\mathcal{C}^{<d}$ to denote all equalities $s \approx t$ of $\mathcal{C}$ with $|s| + |t| < d$. Moreover we write $\mathcal{C}\restriction_n$ for the set of the n smallest equalities in $\mathcal{C}$. With these notations, $S(\mathcal{C})$ can be described as follows:

$$S(\mathcal{C}) = \bigcup_{\mathcal{R} \in \mathfrak{R}(\mathcal{C})} \left(\{ s\downarrow_{\mathcal{R}} \approx t\downarrow_{\mathcal{R}} \mid s \approx t \in \mathcal{E} \cup CP(\mathcal{R}) \text{ and } s\downarrow_{\mathcal{R}} \neq t\downarrow_{\mathcal{R}} \}^{<d}\restriction_n \right)$$

4 Related Work and Comparison

We relate our procedure φ to existing completion methods. Due to their algorithmic nature precise simulations are difficult, but we capture their main features. We say that $\mathcal{S}$ is an *inter-reduced* version of a terminating TRS $\mathcal{R}$, if $\mathcal{S}$ is a terminating reduced TRS whose rules are obtained by rewriting rules in $\mathcal{R}$ by $\mathcal{R}$ itself.

- **Knuth-Bendix Completion** [14, 12]. Let $>$ be a reduction order for the Knuth-Bendix completion procedure and for the orientable part $\{\ell \to r \in \mathcal{C} \cup \mathcal{C}^{-1} \mid \ell > r\}$ we write $\mathcal{C}^{>}$. This procedure can be simulated by φ if one uses

  $$\mathfrak{R}(\mathcal{C}) = \{\mathcal{C}^{>}\} \qquad \text{and} \qquad S(\mathcal{C}) = \{s\downarrow_{\mathcal{R}'} \approx t\downarrow_{\mathcal{R}'}\}$$

  where, $\mathcal{R}'$ is an inter-reduced version of $\mathcal{C}^{>}$ and $s \approx t \in \mathcal{C} \cup CP(\mathcal{R}')$.

- **Multi-completion** [16]. Multi-completion uses a class of reduction orders $>_1, \ldots, >_n$ to run Knuth-Bendix completion in parallel. Typically, the class is the set of all possible recursive path orders. Its run can be simulated in our method as follows:

  $$\mathfrak{R}(\mathcal{C}) = \{\mathcal{C}^{>_1}, \ldots, \mathcal{C}^{>_n}\} \qquad \text{and} \qquad S(\mathcal{C}) = \{s\downarrow_{\mathcal{R}'} \approx t\downarrow_{\mathcal{R}'}\}$$

  where, $\mathcal{R}'$ is an inter-reduced version of $\mathcal{C}^{>_i}$ and $s \approx t \in \mathcal{C} \cup CP(\mathcal{R}')$. A naive implementation of this approach fails due to the large number of compatibility checks as well as computations of normal forms. In order to gain efficiency Kondo and Kurihara [16] provided a specialised data structure, so-called *node* for sharing these computations among the orders.

- **Completion with termination tools** [23]. This procedure does not require a reduction order as an input parameter, because during its process a necessary reduction order is constructed on the fly:

  $$\mathfrak{R}(\mathcal{C}) = \{\mathcal{R}\} \qquad \text{and} \qquad S(\mathcal{C}) = \{s\downarrow_{\mathcal{R}'} \approx t\downarrow_{\mathcal{R}'}\}$$

  where, $\mathcal{R}$ is a TRS over $\mathcal{C}$ whose termination is shown by a *termination tool*, $\mathcal{R}'$ is an inter-reduced version of $\mathcal{R}$, and $s \approx t \in \mathcal{C} \cup CP(\mathcal{R}')$. Unlike a fixed single reduction order, a termination tool can find a number of terminating TRSs over $\mathcal{C}$, which avoids failure of Knuth-Bendix completion. But its drawback is a similar problem as with multi-completion. In the paper [23] a heuristic for the *best search strategy* is suggested to select one of the terminating TRSs. This approach significantly extends the power of Knuth-Bendix completion, and has been adopted in their completion tool Slothrop.

- **Multi-completion with termination tools** [21, 25]. The method replaces reduction orders in multi-completion by a termination tool:

  $$\mathfrak{R}(\mathcal{C}) = \{\mathcal{R}_1, \ldots, \mathcal{R}_n\} \qquad \text{and} \qquad S(\mathcal{C}) = \{s{\downarrow_{\mathcal{R}'}} \approx t{\downarrow_{\mathcal{R}'}}\}$$

  where, $\mathcal{R}_1, \ldots, \mathcal{R}_n$ are all TRS over $\mathcal{C}$ whose termination is shown by a termination tool, $\mathcal{R}'$ is an inter-reduced version of some $\mathcal{R} \in \mathfrak{R}(\mathcal{C})$, and $s \approx t \in \mathcal{C} \cup CP(\mathcal{R}')$. A variant of the node data structure in multi-completion provides a compact representation of $\mathfrak{R}(\mathcal{C})$ as well as an efficient algorithm to compute it. This approach has been implemented in the very effective completion tool mkbTT.

As described in the earlier sections, maximal completion only computes maximal terminating TRSs, which are often much fewer than all terminating TRSs, but it does not miss a complete TRS. This is the main idea of our approach. One drawback is the current limited power of maximal termination provers. Theoretically, Brute force search allows using a termination tool to compute maximal terminating TRSs. However, it is practically infeasible due to exponentially many calls of the termination tool.

Another difference is the definition of $S(\mathcal{C})$. Except for maximal completion, all procedures use a singleton of an equality for $S(\mathcal{C})$ and its *selection* is critical for successful runs. The next example illustrates this.

▶ **Example 8.** Recall the ES $\mathcal{E}$ in Example 5:

$$1\colon\ \mathsf{s}(\mathsf{p}(x)) \approx x \qquad 2\colon\ \mathsf{p}(\mathsf{s}(x)) \approx x \qquad 3\colon\ \mathsf{s}(x) + y \approx \mathsf{s}(x + y)$$

We perform $\varphi(\mathcal{E})$ as the simulated run of multi-completion, where the class of reduction orders is all LPOs with total precedence. Assume that our selection strategy for $S(\mathcal{C})$ prefers an equality derived from the critical pair of rule 3 and the rule of the biggest possible index for some TRS in $\mathfrak{R}(\mathcal{C})$.

(i) $\mathfrak{R}(\{1,2,3\}) = \{\{1,2,3\}, \{1,2,3'\}\}$, which both do not satisfy the join-condition of φ. Thus, $\varphi(\{1,2,3\}) = \varphi(\{1,\ldots,4\})$, where 4 is the single equality in $S(\{1,2,3\})$:

$$4\colon\quad x + y \approx \mathsf{s}(\mathsf{p}(x) + y)$$

(ii) $\mathfrak{R}(\{1,\ldots,4\}) = \{\{1,2,3,4'\}, \{1,2,3',4'\}\}$ and the join-condition does not hold again. We continue the run with $\varphi(\{1,\ldots,4\}) = \varphi(\{1,\ldots,5\})$, where 5 in $S(\{1,\ldots,4\})$ is

$$5\colon\quad (x + y) + z \approx \mathsf{s}((\mathsf{p}(x) + y) + z)$$

(iii) Generally, $\mathfrak{R}(\{1,\ldots,n\}) = \{\{1,2,3,4',\ldots,n'\}, \{1,2,3',4',\ldots,n'\}\}$ and $S(\{1,\ldots,n\})$ is the singleton of

$$n{+}1\colon\quad ((x_1 + x_2) + \cdots) + x_{n-1} \approx \mathsf{s}(((\mathsf{p}(x_1) + x_2) + \cdots) + x_{n-1})$$

for $n \geq 3$. Thus, the join-condition never holds and the procedure does not terminate.

Admittedly, for the above example it is easy to choose an appropriate selection strategy that succeeds. In general however, it is difficult to know a suitable selection strategy a priori. This is why mkbTT provides several selection strategies as a user parameter. Maximal completion does not use a singleton but a *set* of equalities for $S(\mathcal{C})$, which reduces the risk to get stuck.

To conclude, we like to stress the simplicity of maximal completion, due to avoiding a dedicated search algorithm like one in Slothrop, and a sophisticated but complex data structure like that of multi-completion.

Table 1 Summary for all 115 equational systems

	LPO		KBO		termination tool	
	mkbTT	Maxcomp	mkbTT	Maxcomp	mkbTT	Slothrop
completed	70	**86**	67	**69**	81	71
failure	6	**6**	3	**3**	3	4
timeout	39	**23**	45	**43**	31	40

5 Experiments

We implemented maximal completion in the tool Maxcomp. The tool employs the SMT solver Yices [8] to support LPO and KBO. Concerning parameter K for $\mathfrak{R}(\mathcal{C})$ in Section 3, at the beginning we use $K = 2$ to compute two maximal terminating TRSs. During the recursion of φ, we increase K whenever $S(\mathcal{C}) \subseteq \mathcal{C}$. Moreover, we fix $n = 7$ and $d = 20$ for $S(\mathcal{C})$, simply motivated by the fact that our tool cannot process larger n and d due to the number of equalities. If, at some point, no new equalities are generated and all maximal terminating TRSs are computed (i.e. parameter K cannot be increased anymore), the tool stops with failure.

We compare Maxcomp with the two existing completion tools Slothrop and mkbTT. Since the latter two tools require termination provers, we used AProVE [10] for Slothrop, and for mkbTT its internally supplied prover T$_{\text{T}}$T$_2$ [15]. The test-bed consists of 115 equational systems from the distribution of mkbTT.[1] The tests were single-threaded run on a system equipped with an Intel Core Duo L7500 with 1.6 GHz and 2 GB of RAM using a timeout of 300 seconds.

Table 1 gives a summary of the overall results. Here we also included results, where mkbTT's termination proving power was limited to LPO and KBO[2] (this is not possible for Slothrop). Aside from this, all parameters of all tools were left as default. More detailed results for a selected number of systems[3] are depicted in Table 2, where we omitted systems that could be solved by all tools with every termination method, or that could not be solved by any tool. Moreover for the two scalable systems `BGK.Dxx` and `BGK.Mxx`, that are constructed by using a natural number as an input parameter, only the largest ones considered, `BGK.D16` and `BGK.M14`, are shown here. Numbers denote execution time in seconds, $\times$ denotes the tool stopped with failure, and ∞ denotes timeout.

It should be noted that the complete systems found by using a termination tool are mostly different from those found with LPO or KBO, since the reduction order constructed using a termination tool is usually different from them. Also, for fairness it should be noted that by choosing specific, suitable selection strategies for each equational system individually, mkbTT can complete more systems than with its default selection strategy. To name one

[1] `http://cl-informatik.uibk.ac.at/software/mkbtt/`
[2] `mkbtt -s lpo` for LPO, and `mkbtt -s kbo` for KBO.
[3] The complete list is available at `http://www.jaist.ac.jp/project/maxcomp/`.

Table 2 Experimental results

	LPO		KBO		termination tool	
problem	mkbTT	Maxcomp	mkbTT	Maxcomp	mkbTT	Slothrop
BGK94.D16	∞	37.48	∞	∞	74.91	∞
BGK94.M14	2.73	∞	5.73	∞	2.06	25.88
Chr89.A3	40.06	3.17	20.85	∞	∞	∞
fib	0.87	100.32	∞	∞	0.85	7.18
Les83.fib	0.10	0.01	∞	∞	0.23	2.82
Les83.subset	0.22	0.01	∞	∞	0.09	3.27
OKW95.dt1.theory	0.78	46.15	∞	∞	0.91	6.22
rl.theory	2.36	0.17	∞	1.21	1.76	8.07
SK90.3.04	6.72	1.75	∞	∞	99.98	∞
SK90.3.05	1.58	0.35	1.07	0.33	3.46	∞
SK90.3.06	4.22	0.90	∞	∞	∞	∞
SK90.3.07	∞	3.03	∞	∞	∞	∞
SK90.3.15	∞	∞	0.12	0.02	0.08	1.39
SK90.3.18	0.15	0.01	∞	∞	0.30	3.50
SK90.3.22	∞	4.03	∞	8.03	∞	∞
SK90.3.27	13.31	21.09	36.47	0.53	72.31	∞
SK90.3.28	∞	20.23	53.58	∞	∞	110.76
slothrop.ackermann	0.02	0.01	∞	∞	0.03	0.68
slothrop.cge	∞	∞	∞	∞	173.03	∞
slothrop.endo	∞	0.62	∞	0.38	3.88	7.60
slothrop.equiv.proofs	×	×	∞	∞	2.52	262.32
slothrop.equiv.proofs.or	×	×	∞	∞	2.81	∞
slothrop.groups	0.45	0.08	∞	0.08	0.54	2.22
TPDB.zantema.z115	∞	0.73	5.76	42.63	15.27	203.69
TPTP.COL060.1.theory	×	×	0.01	0.01	0.02	1.14
TPTP.GRP454.1.theory	17.13	2.65	49.63	0.14	109.87	∞
TPTP.GRP457.1.theory	17.18	2.66	49.98	1.62	112.73	∞
TPTP.GRP460.1.theory	∞	0.89	17.74	2.91	16.99	∞
TPTP.GRP463.1.theory	∞	1.76	17.80	3.26	16.91	∞
TPTP.GRP481.1.theory	∞	2.40	∞	57.20	∞	69.56
TPTP.GRP484.1.theory	∞	0.52	∞	13.52	∞	∞
TPTP.GRP487.1.theory	∞	1.00	∞	7.05	∞	×
TPTP.GRP490.1.theory	∞	0.84	168.93	13.21	∞	∞
TPTP.GRP493.1.theory	∞	5.13	40.54	2.83	∞	∞
TPTP.GRP496.1.theory	∞	0.90	∞	11.13	∞	241.10
WS06.proofreduction	∞	∞	∞	∞	162.66	∞

example, mkbTT completes `SK90.3.22` with LPO and selection strategy `old` [25] within roughly 40 seconds, however times out after 300 seconds with all other predefined strategies (`max`, `slothrop`, `sum`). While the chosen selection strategy vastly affects the outcome of mkbTT, it is in general non-trivial to decide which selection strategy to choose in advance. Lastly, concerning the timeout, with very few exceptions, a higher timeout seems not to affect the results. For overall performance, whenever all tools succeeded, they usually (with four exceptions) did so in less than 35 seconds. For the rest of systems, timing values do not show a clear trend. The parameter K mostly remains unchanged at 2, and for the vast majority of successful runs does not exceed 5, the maximum being 14.

6 Conclusion and future work

We have illustrated a very compact framework for (Knuth-Bendix) completion and demonstrated how to effectively automate it by employing modern MaxSAT solvers. Despite relying on (not very powerful) simplification orders to show termination in our implementation, experimental results indicate practical viability and, within the test-scenario as indicated here, competitiveness compared to all other known approaches. To conclude, we mention potential future work:

- Recapitulating termination techniques into maximal termination is crucial to extend the capability of maximal completion. We plan to implement matrix interpretations [9], which are a recently emerged very powerful class of reduction orders: For instance, a lexicographic combination of matrix interpretations can prove termination of a complete TRS for the system CGE2 [23]. Moreover, adaptation of the dependency pair method [2] and semantic labeling [27] for maximal termination is an important challenge.
- In order to achieve better scalability the growth of $S(\mathcal{C})$ in the iteration of φ needs to be limited. In Knuth-Bendix completion and its variants, inter-reduction is employed as well as the application of various critical pair criteria ([4, 13]). We expect that these techniques can be adapted in our framework.
- Last but not least, application to theorem proving is an important direction. There are variations of Knuth-Bendix completion including unfailing completion [5] and rewriting induction [20, 1], which are very successfully employed in powerful theorem provers like Waldmeister [18]. However, these variations require a fixed reduction order. Very recently Winkler and Middeldorp [24] adapted multi-completion with termination tools for unfailing completion to overcome this restriction. We anticipate that our approach can be integrated in these settings.

Acknowledgements We thank Sarah Winkler and Aart Middeldorp for their valuable comments.

References

1 T. Aoto. Dealing with non-orientable equations in rewriting induction. In *RTA 2006*, volume 4098 of *LNCS*, pages 242–256, 2006.
2 T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236(1-2):133–178, 2000.
3 F. Baader and T. Nipkow. *Term rewriting and all that*. Cambridge University Press, 1998.
4 L. Bachmair and N. Dershowitz. Critical pair criteria for completion. *Journal of Symbolic Computation*, 6(1):1–18, 1988.

5 L. Bachmair, N. Dershowitz, and J. Hsiang. Orderings for equational proofs. In *LICS*, pages 346–357. IEEE Computer Society, 1986.

6 L. Bachmair, N. Dershowitz, and D. A. Plaisted. *Resolution of Equations in Algebraic Structures: Completion without Failure*, volume 2, pages 1–30. Academic Press, 1989.

7 M. Codish, V. Lagoon, and P. Stuckey. Solving partial order constraints for LPO termination. In *RTA 2006*, volume 4098 of *LNCS*, pages 4–18, 2006.

8 B. Dutertre and L. D. Moura. A fast linear-arithmetic solver for dpll(t). In *CAV*, pages 81–94, 2006.

9 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.

10 J. Giesl, P. Schneider-Kamp, and R. Thiemann. AProVE 1.2: Automatic termination proofs in the dependency pair framework. In *IJCAR 2006*, volume 4130 of *LNAI*, pages 281–286, 2006.

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

12 G. Huet. A complete proof of correctness of the Knuth-Bendix completion algorithm. *Journal of Computer and System Sciences*, 21(1):11–21, 1981.

13 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.

14 D. Knuth and P. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. 1970.

15 M. Korp, C. Sternagel, H. Zankl, and A. Middeldorp. Tyrolean termination tool 2. In *RTA 2009*, volume 5595 of *LNCS*, pages 295–304, 2009.

16 M. Kurihara and H. Kondo. Completion for multiple reduction orderings. *Journal of Automated Reasoning*, 23(1):25–42, 1999.

17 M. Kurihara and H. Kondo. Efficient BDD encodings for partial order constraints with application to expert systems in software verification. In *IEA/AEI*, volume 3029 of *LNAI*, pages 827–837, 2004.

18 B. Löchner and T. Hillenbrand. A phytography of Waldmeister. *AI Communications*, 15(2–3):127–133, 2002.

19 Y. Métivier. About the rewriting systems produced by the Knuth-Bendix completion algorithm. *Information Processing Letters*, 16(1):31–34, 1983.

20 U. S. Reddy. Term rewriting induction. In *CADE 1990*, volume 449 of *LNCS*, pages 162–177, 1990.

21 H. Sato, S. Winkler, M. Kurihara, and A. Middeldorp. Multi-completion with termination tools (system description). In *IJCAR 2008*, LNCS, pages 306–312, 2008.

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

23 I. Wehrman, A. Stump, and E. M. Westbrook. Slothrop: Knuth-Bendix completion with a modern termination checker. In *RTA 2006*, LNCS, pages 287–296, 2006.

24 S. Winkler and A. Middeldorp. Termination tools in ordered completion. In *IJCAR 2010*, volume 6173 of *LNAI*, pages 518–532, 2010.

25 S. Winkler, H. Sato, A. Middeldorp, and M. Kurihara. Optimizing mkbtt (system description). In *RTA 2010*, LIPIcs, 2010.

26 H. Zankl, N. Hirokawa, and A. Middeldorp. KBO orientability. *Journal of Automated Reasoning*, 43(2):173–201, 2009.

27 H. Zantema. Termination of term rewriting by semantic labelling. *Fundamenta Informaticae*, 24:89–105, 1995.

CRSX—Combinatory Reduction Systems with Extensions

Kristoffer H. Rose

IBM Thomas J. Watson Research Center
P. O. Box 704, Yorktown Heights, NY 10598, USA
krisrose@us.ibm.com

Abstract

Combinatory Reduction Systems with Extensions (CRSX) is a system available from `http://crsx.sourceforge.net` and characterized by the following properties:

- Higher-order rewriting engine based on pure Combinatory Reduction Systems with full strong reduction (but no specified reduction strategy).
- Rule and term syntax based on λ-calculus and term rewriting conventions including Unicode support.
- Strict checking and declaration requirements to avoid idiosyncratic errors in rewrite rules.
- Interpreter is implemented in Java 5 and usable stand-alone as well as from an Eclipse plugin (under development).
- Includes a custom parser generator (front-end to JavaCC parser generator) designed to ease parsing directly into higher-order abstract syntax (as well as permitting the use of custom syntax in rules files).
- Experimental (and evolving) sort system to help rule management.
- Compiler from (well-sorted deterministic subset of) CRSX to stand-alone C code.

1998 ACM Subject Classification F.4.2, F.4.3, F.3.3

Keywords and phrases Higher-Order Rewriting, Compilers

Digital Object Identifier 10.4230/LIPIcs.RTA.2011.81

Category System Description

1 Introduction

CRSX is a project on SourceForge, specifically `http://crsx.sourceforge.net`, currently at version 20 but under intense development. CRSX aims at providing a high quality rewrite engine and development environment for experimenting with higher order rewriting in general and higher order rewrite systems for compilers in particular, and has recently been rather successful at the latter including being the basis for a large internal compiler project in IBM.

CRSX stands for *Combinatory Reduction Systems* (CRS) with "eXtensions," which we detail below. CRS were invented and studied in depth by Jan Willem Klop [2, 3], who in turn credits the idea to Peter Aczel [1]. They are "combinatory" because every meta-variable carries all the (locally bound) variables it depends on, which enables the use of higher order terms and substitution.

In this paper we summarize what the CRSX system includes and is capable of in general terms. We assume some familiarity with rewriting (even higher order rewriting) as well as a general computer science background (such as an understanding of what a compiler and an interpreter are, *etc.*).

22nd International Conference on Rewriting Techniques and Applications.
Editor: M. Schmidt-Schauß; pp. 81–90



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

Components. CRSX consists of a number of pieces that work together.

Interpreter (CRSX). Executable that is used to load and "execute" rewrite systems. Loads a "script," which is itself a rewrite system using special internal "directive" terms to bootstrap the system. The *default script* is setup to load a file with CRSX rewrite rules and another with an input term, rewrite the term, and output the result in one of several customizable forms. The interpreter is written in Java 5.

Parser Generator (PG). Tool for building parsers from source text directly to CRSX higher-order terms such that input terms and rules can use custom syntax. Includes special support for directly generating higher order abstract syntax (HOAS) [4] and configuring scopes during parsing. Currently written as a JavaCC [8] script.

Eclipse Plugin. A helper to allow editing CRSX rules files with syntax highlighting and rule outlines in Eclipse, and with ambitions for simple debugging and such. Available directly from an Eclipse update site at `http://crsx.sourceforge.net/eclipse-plugin/`.

RulesCompiler. Tool for converting a rule system to a stand-alone C program. Written as a CRSX script including a large rules file for generating C, however, does invoke various experimental low-level directives programmed in Java.

In the following sections we briefly describe some of the special characteristics of these components.

Contributors. Most of the code so far has been written by the author, except the Eclipse plugin, which was written by Takahide Nogayama (IBM Tokyo Research Lab).

Availability. CRSX is available from the SourceForge archive `http://sourceforge.net/projects/crsx/` (click on "Code"). The code is Open Source and all CRSX documentation is being managed using the SourceForge source control system.

2 CRSX Extensions

The notation of CRSX modifies the original base CRS formalism in several ways, described briefly here (for a full description see the evolving system documentation [7]).

Symbols. All symbols can be used as constructor symbols. Unless the symbol has another meaning, it can be used directly, otherwise it can be used in quotes, so although lower case letters and words are normally variables, `'x'` is a constructor, and all symbols containing a `#` are normally meta-variables but `"Expr#"` is a constructor. In addition, CRSX uses Unicode, so many exotic characters are available.

Structure brackets. Basic term formation uses square brackets for subterms, and every symbol not otherwise specified is a constructor symbol. So simple arithmetic terms can be written, *e.g.*, as `+[*[1,2],3]`, where all of the symbols `+`, `*`, `1`, `2`, and `3`, are constructors; notice how we have omitted `[]` after the nullary constructors.

Subterm binder dot. Bound variables are specified with a "dot" prefix and are restricted to construction subterms. A standard "**let** $a = 1 \times 2$ **in** $a + 3$" binder construct can for example correspond to the term `Let[*[1,2], a.+[a,3]]` where the term makes the binder's scope explicit.[1]

λ-calculus conventions. Two special conventions facilitate λ-calculus notation: prefixing a constructor to some binders corresponds to nesting, *e.g.*, λ `x y z . x` really means

[1] The main effect of the subterm restriction for binders is that a fragment like `x.x` is not in itself a term.

`λ[x.λ[y.λ[z.x]]]`, and juxtaposition and normal parenthesis correspond to left-associative use of a special application operator, *e.g.*, `x z (y z)` really means `@[@[x,z],@[y,z]]`.[2]

Meta-application #tags. Words with `#` in them are meta-variables used to form meta-applications with brackets, *e.g.*, `Expr#1[x,Sub#Expr]`, where we can also omit `[]` for ground meta-applications.

Free variables. Unlike standard CRS, CRSX permits *free variables*, *i.e.*, `x` is a proper term. (This turns out to be crucial when defining recursive compilation and analysis schemes as these invariably have a case for free variables.)

Property lists. CRSX includes special syntax for constant properties and for properties of variables, written as *prefixes* of the form `{` *`Constant`*`:` *`Term`*`;...; v:` *`Term`*`;...; }`, where a *`Constant`* is a constructor symbol, `v` a variable, and *`Term`* arbitrary terms. We explain below how rules can match against the existence and value of properties.

Rules. Directives of the form *`Name`*`[`*`Options`*`]:` *`Pattern`* $\to$ *`Contraction`* are rules describing how subterms that match the *`Pattern`* can be replaced by subterms built according to the *`Contraction`*.[3] As usual in CRS, the *`Pattern`* is restricted to constructions where each contained meta-application must be applied to distinct bound variables, and the arity of each meta-variable must be consistent throughout the rule. We shall discuss rules in more details below, including the role and form of the *`Options`*.

Sorts. Directives of the form *`Sort`* `::=(` *`Form`*`;...)` and *`Form`* `::` *`Sort`* declare algebraic sorts of the included *`Form`*s. Each *`Sort`* is just a constant name,[4] and each *`Form`*, in turn, defines the shape of one construction by *`Constructor`*`[`B_1*`Sort`*$_1$`,...,`B_n*`Sort`*$_n$`]`, which specifies several constraints. First the *`Constructor`* can only occur where instances of the defined *`Sort`* are permitted. Second, the i'th subterm of the constructor must have the *`Sort`*$_i$ with precisely the binders described by B_i, which may be empty or have the form $\vec{\mathtt{x}}_i$`.{`$\mathtt{x}_{i1}$`:`*`Sort`*$_{ik}$`;...;`$\mathtt{x}_{i1}$`:`*`Sort`*$_{i1}$`}`, which combines the constraints that

- there must be precisely k binders on the i'th subterm, and
- each bound variable corresponding to binder $\mathtt{x}_{ij}$ must occur inside the subterm scope with $Sort_{ij}$.

Sort declarations with `::=` are called *data sort declarations* in that they enumerate forms of a sort that can only be included as non-outermost constructions in patterns, whereas `::`-declarations are called *function sort declarations* that (separately) enumerate forms of a sort that can only be included in patterns as the outermost construction. (Constructors that have not been included as part of a sort do not have any such restrictions.)

Evaluators. Rules can contain special evaluation terms of the form `$[`*`Constant`*`,...]`, where the *`Constant`* indicates what evaluation to perform in the pattern or contraction in question. The evaluator `$[Plus,#arg,1]`, for example has these properties:

- if used in a contraction it first contracts `#arg`, and if the result is a constant that can be interpreted as a number, then the contraction of the entire evaluator is the constant representing the number obtained by adding one;
- if used when matching a pattern then it first matches `#arg` against the redex subterm and, if that succeeds for some *`Term`*, then it records that the valuation of `#arg` should effectively be `$[Minus,#arg,1]` (with the obvious meaning).

The available evaluators are summarized in the documentation, including the functional restrictions to prevent the need for backtracking.

---

[2] We further follow the common convention that application binds tighter than other constructs.
[3] The "$\to$" in the directive is a the Unicode character U2192.
[4] We *are* working on parametrically polymorphic sorts...

Meta-rules. Special directives of the form `$Meta[( Rule;... )]` permit rules that rewrite the rewrite rules themselves.

Notice that CRSX does not specify a reduction strategy and has no special evaluation strategy mechanism. Evaluation is only specified to always make progress and thus be weakly normalizing, and indeed the interpreter and C implementations use quite different strategies that we furthermore change over time. (Like all weakly normalizing strategies over higher order terms both reduction strategies reduce terms under binders, of course.) To enforce a particular reduction order it is necessary to insert control symbols that block unwanted reductions.[5]

We present a simple example of how to enter and run a rewrite system here; a larger example with properties and custom syntax will follow below.

▶ **Example 1** (λ-calculus)**.** Let us illustrate the use of the CRSX interpreter from the command line on a simple example. Create a file *beta-eta.crs* with the following content:[6]

```
BetaEta[(
  β[Copy[#X]] : ((λx.#[x]) #X) → #[#X] ;  η[Weak[#]] : (λx. # x) → # ;
  S → (λ x y z.x z (y z)) ;  K → (λ x y.x) ;  I → (λ x.x) ;
)]
```

Now copy the *crsx.jar* file from the CRSX release [7] and execute the following command (using your proper Java 5 `java` command):

```
$ java -jar crsx.jar rules=beta-eta.crs term="S K I"
(λ z . z)
```

The rules use the λ-calculus conventions discussed above, and further illustrate several points:

- A named group of rules (here BetaEta) is specified as a construction with a parenthesized sequence of `;`-terminated rules.
- Each rule is named (here β and η).
- After the rule name some options are needed whenever a rule does something beyond straight linear term rewriting: here we have to declare that, in β, the argument to the application may be copied (by the substitution), and, in η, the second argument under the abstraction omits a free variable from the substitution parameter list (which corresponds to the η-rule not permitting occurrences of that variable).
- The name part of a rule can be omitted (then the rule will be named for the pattern constructor).

To see the purpose of the options, removing them and running the same command would give these errors:

```
Error: BetaEta-β rule contractum uses non-shared/duplicatable meta-variable in place that may be copied (#X).
Error: BetaEta-η rule pattern meta-application # omits bound variables yet is not declared Weak.
Errors prevent normalization.
```

▶ **Example 2** (simply typed λ-calculus)**.** To illustrate the use of environments, we include a rule system for rewriting a simply typed λ-calculus to its type in Figure 1. The rule system defines a `T` scheme and some helper schemes that together rewrite a simply typed λ-term to its type, which can be used as follows:

---

[5] We are experimenting with generating force/delay operators automatically from the sorts but this is not available yet.

[6] Use the Unicode coding for the Greek letters and encode with UTF-8.

```
BetaTypes[(
 TYPE ::=( BASE; TYPE→TYPE; ERROR; );
 TERM ::=( x; λ[TYPE, x.{x : TERM}TERM]; TERM TERM; );

 T[TERM] :: TYPE ;
 -[Free[x]] : {x : #α} T[x] → #α ;
 -[Free[x]] : {¬x} T[x] → BASE ;
 -[Fresh[v],Copy[#α]] : {#Γ} T[λ[#α, x.#[x]]] → (#α → {#Γ; v : #α} T[#[v]]) ;
 {#Γ} T[#1 #2] → {#Γ} TA[{#Γ} T[#1], #2] ;

 TA[TYPE, TERM] :: TYPE ;
 -[Discard[#]] : TA[BASE, #] → ERROR ;
 {#Γ} TA[#α → #β, #] → M[#α, {#Γ} T[#], #β] ;

 M[TYPE, TYPE, TYPE] :: TYPE ;
 M[BASE, BASE, #γ] → #γ ;
 M[#α→#β, #α2→#β2, #γ] → M[#α2, #α, M[#β, #β2, #γ]] ;
 -[Discard[#α2,#β2]] : M[BASE, #α2→#β2, #γ] → #γ ;
 -[Discard[#γ]] : M[BASE, ERROR, #γ] → ERROR ;
 -[Discard[#α,#β,#γ]] : M[#α→#β, BASE, #γ] → ERROR ;
 -[Discard[#α,#β,#γ]] : M[#α→#β, ERROR, #γ] → ERROR ;
)]
```

Figure 1 *samples/lambda/beta-type.crs*: type analysis of simply typed λ-calculus.

```
$ java -jar crsx.jar check-sorts rules=beta-type.crs term="T[λ[BASE,x.x]]"
(BASE → BASE)
```

(the precise options used can be found in the system manual).

The system includes two sorts `TYPE` and `TERM` with terms describing simple types and simply typed terms, respectively. Notice that `TERM` permits free variables as well as a typed binder under λ; in addition to the λ-calculus conventions it makes use of the built-in parsing of infix use of $\to$.

The second block defines the sort and rules for the `T` scheme. The first two rules give cases for `T` on free variables: one rule for variables that are defined in the type environment and another for globally free variables. Both rules include a `Free[x]` option, which is required to allow the free variable `x`, and both have a property constraint on the properties of the root `T` (the first requiring it to be `#`α and the second requiring it to be absent).

The third rule constructs the function type of an abstraction, which involves replacing the bound variable `x` with a fresh variable `v`, therefore declared `Fresh[v]` (in addition to `Copy[#`α`]` for the copied substructure). It illustrates how the property list in the contraction extends the matched properties with an additional binding.

The fourth and final `T` rule constructs the type of an application with a helper `TA` scheme that in turn uses the `M` scheme for type matching of the formal and actual argument types; the last two blocks define these. Notice the use of the `Discard` option, which is required for rules to not contract specific matched subterms: this is the single most effective error catching mechanism in the system.

```
// Simple XQuery-like language.
grammar net.sf.crsx.samples.x.X : <P>, <E>, <S>, <Q>

meta[<E>] ::= "#<PRODUCTION_NAME>" i?, "[", "]" .    // Meta-applications over AST.

skip ::= " " | "\r" | "\n" | "\t" .                  // White space.

<P> ::= {program} <E> .                              // Program.

<E> ::= <S>:#S ("," <E>:#E ⟦"comma"[#S,#E]⟧ | ⟦#S⟧) . // Expression.

<S> ::= "(" (<E> | {empty}) ")"                      // Simple expression.
     |  "element"_{} <N> "{" <E> "}"
     |  {query} <Q>
     |  "if"_{} <S> "then" <S> "else" <S>
     |  {call} <N> "(" (<E> | {empty}) ")"
     |  v_?
     |  {literal} <L>
     .

<Q> ::= "for"_{} v_x "in" <S> <Q>[x]                 // Query.
     |  "let"_{} v_x ":=" <S> <Q>[x]
     |  "where"_{} <S> <Q>
     |  "return"_{} <S>
     .

<N> ::= n_{} .                                       // Names.
<L> ::= l_{} .                                       // Literals.

token l ::= i | "'" (¬[\'] | "''")* "'" .
token i ::= [0-9]+ .
token n ::= [A-Za-z_] [A-Za-z0-9_-]* .
token v ::= "$" n .
```

Figure 2 *samples/x/x.pg*: sample nested loop language parser.

3 Parser Generator

The system includes the "PG" parser generator for generating custom parsers for higher order terms. Figure 2 contains the input file to PG for a small "nested loop" query language like XQuery [6]. The PG notation is designed to express common higher order term constructions.

- The top "grammar" declaration declares the name of the grammar as well as the externally available syntactic categories, with the first, `<P>`, being the default.
- The `meta` declaration sets up the format for meta-variables in the language, in this case using the same `#-[-]` notation as in CRSX itself.
- The `skip` declaration just specifies the format for white space.
- The `<P>` production states that a program will be represented as a *program* construction (indicated by the `{}`s) with the contained `<E>` as a subterm.
- The `<E>` production says that an `<E>` is an `<S>` followed either by a comma and a sub-`<E>` or nothing. The `:#S` allows us to refer to the `<S>` as `#S`, and in the first case `#E` can be

used to reference the sub-`<E>`; in both cases the result term is specified in ⟦...⟧s.[7]

- The `<S>` production has several choices; for each the generated term is specified either by giving the constructor prefix in `{}`s, or by indicating with a `_{}` marker that an existing token should itself be used as a constructor prefix, or (for the single variable token case) with the marker `_?` that the token should be the name of an in-scope variable.
- The `<Q>` production introduces binding constructs with the `_x` markers that indicate that the `v` token should be interpreted as a variable and, in each case, the correponding `[x]` marker indicates the (single) subterm that is the scope of the variable.
- The `<N>` and `<L>` productions just use the tokens as constructors.
- Finally, the four token kinds are defined using regular expressions.

In Java, the parser is then generated by first invoking PG to generate *X.jj* and then JavaCC [8] to generate the appropriate Java classes which then permits parsing such as the following, where we explicitly request the grammar name and syntax category to use:

```
$ java -jar crsx.jar "grammar=('net.sf.crsx.samples.x.X';)" category=P \
>   term="for $x in child(doc()) for $y in child(doc()) where eq($x,$y) return plus($x,$y)"
"program"[
 "query"[
  "for"["call"["child", "call"["doc", "empty"]],
   v"$x" . "for"[
     "call"["child", "call"["doc", "empty"]],
     v"$y" . "where"["call"["eq", "comma"[v"$x", v"$y"]],
                     "return"["call"["plus", "comma"[v"$x", v"$y"]]]]]]]]
```

Finally, the `meta`-declaration makes it possible to use rules files with embedded syntax, *e.g.*, the *samples/x/N.crs* rules file contains a rule

```
-[Free[id],Fresh[f]] :
 NQ[%Q⟦ for $v in #S #Q[$v] ⟧, id, t.#op[t]]
 → NQ[#Q[f], id, id3.MapConcat[
        Dep[id2.Map[Dep[id1.Tuple[ACons[f id1, ANil]]], N[#S, id2]]],
        #op[id3]]] ;
```

where the special `%Q⟦...⟧` notation invokes the parser described above to parse the `for`-construct allowing for binders and embedded meta-variables of the appropriate sorts.

4 Eclipse Plugin

We include an Eclipse plugin for editing CRSX files (and, to some extent, PG grammar files). Figure 3 shows a screen shot of the plugin running in Eclipse. The plugin follows the usual Eclipse style of having a main editing pane for the rules and support panes with additional navigation and status information. The views include:

- Syntax highlighting in the editor view, including flagging of syntax errors.
- A structured outline view for quick navigation to rules and rule components.
- Simple stepping through an execution of a rewrite system with highlighting of the current rule and a way to access the current state of the term.
- A simple "observe rule" mechanism to allow very primitive break points; we hope to extend this capability to a proper debugging facility.

At the time of writing, the plugin is still under heavy development thus should not be expected to always work fully.

[7] The double brackets are Unicode characters U27E6 and 7.

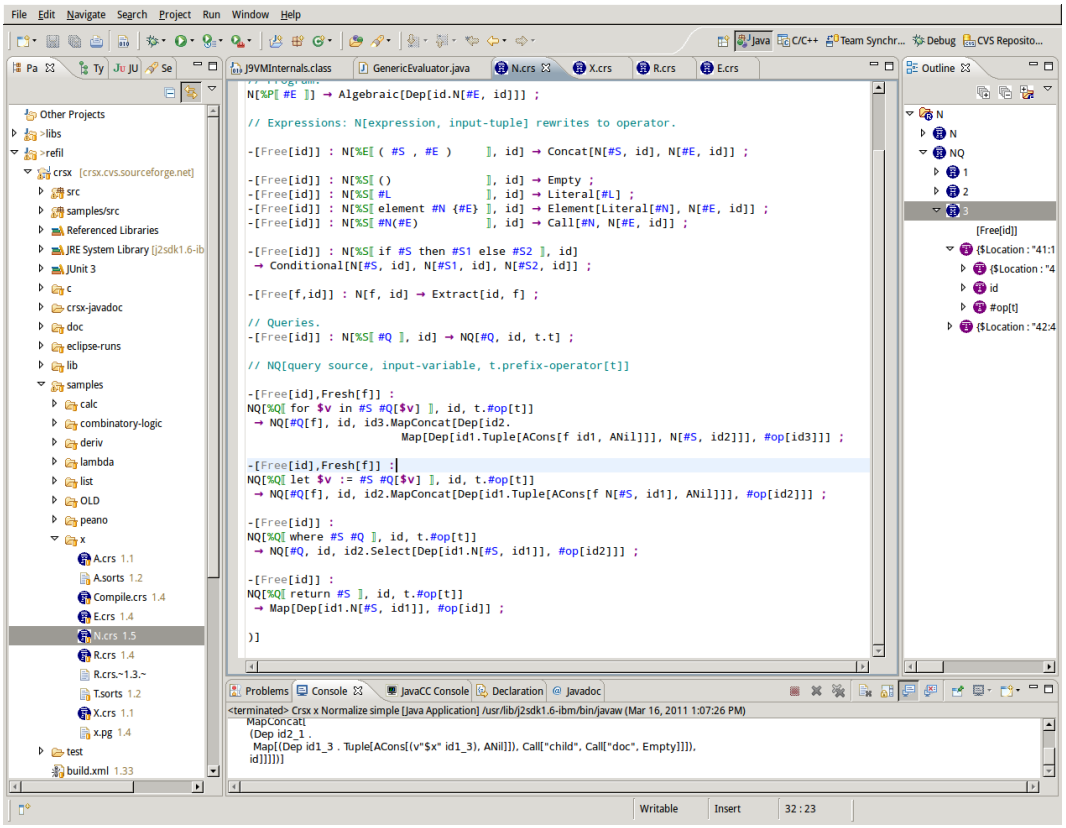


Figure 3 Screen shot of Eclipse plugin.

5 RulesCompiler

In order for CRSX to be used as a compiler generator, it needs to generate code. We have chosen to do this by translating the rules of the rewrite systems itself directly to C in the following stages:

1. First make sure that the rewrite system is an orthogonal sorted constructor system (this should be automated with a completion procedure but we do not have that yet): any actual term should only match a single rule and all symbols should be sorted (which implies that they are categorized cleanly as function or data symbols).
2. Assign sorts to all constructors of the rewrite system to make sure that it is precisely understood what the possible constructors at every subterm are. Sort assignment is currently a standard monomorphic algebraic type analysis (we hope to extend this to permit polymorphic data constructors as these add convenience without complication).
3. The rewrite system is "dispatchified" by splitting all rules with nested patterns into a separate rule per choice point (similarly to the way normal functional language pattern matching is compiled).
4. For each constructor a C structure is created with the specifics of that construction, notably the number of subterms and rank (binder count) for each subterm.
5. Code is generated for every rule where
   a. pattern matching is translated to a switch on the root constructor of the investigated subterm that generates the specific rule function symbol for that pattern case; the switch has a case for "free variable" precisely when the sort of the subterm permits it;

```
// RULE: β : ((λ x . #[x]) #X) → #[#X]
int stepFunction_M__40(Sink sink_1, Term term_1)
{
  DEBUGT(sink_1->context, "\n==========\nMATCH β \n==========\n", term_1);
  Term sub_1 = FORCE(sink_1->context, SUB(term_1,0));
  if (!IS_DATA(sub_1)) return 0;
  Term m_M__23 = SUB(sub_1,0);
  Variable mbind0_M__23 = BINDER(sub_1,0,0);
  Term m_M__23X = SUB(term_1,1);
  int mcount_M__23X = 0;
  DEBUGF(sink_1->context, "%s","\n==========\nCONTRACT β\n==========\n");
  PROPERTIES_RESET(sink_1);
  {
    Variable vars_1[1] = {mbind0_M__23};
    Term subs_1[1];
    {
      Sink buf_1 = MAKE_BUFFER(sink_1->context);
      COPY(buf_1, m_M__23X, mcount_M__23X++);
      subs_1[0] = BUFFER_TERM(buf_1);
      FREE_BUFFER(sink_1->context, buf_1);
    }
    struct _SubstitutionFrame substitution_1 = {NULL, 1, vars_1, subs_1};
    SUBSTITUTE(sink_1, m_M__23, 0, &substitution_1);
  }
  DEBUGF(sink_1->context, "%s","\n==========\nEND β\n==========\n");
  return 1;
}
```

Figure 4 Fragment of generated C "step" code.

   b. rules drive evaluation of arguments that need to be pattern matched;
   c. every component of each pattern is extracted, including binders;
   d. the result term is constructed from left to right by combining new constructors and binders with copies of existing terms, possibly subject to substitutions.

6. A top level normalization algorithm is applied that repeatedly attempts to reduce the outermost functional subterm until there are none or reduction is stuck.

Figure 4 contains step function code generated for the β rule of the λ-calculus.

The `sub_1` term is set to the value of the first argument after it has been "forced" to be a value. If the value is then not a construction (so a free variable) then the step fails because some rule in the context must perform the substitution. In pure λ calculus we then now know that it must be a λ construction and thus we can merely extract the binder and subterm of the λ construction. The contraction can then start: we create a substitution valuation for replacing instances of the bound variable with copies of the application argument and process the substitution sending the result to the buffer sink that the rewrite step is configured to produce.

For general β reduction this is very naive, however, in practice compilers do few operations as general as this: most rewrite steps are subject to optimizations. The most important one is to eliminate copies and substitutions where possible. Specifically, many substitutions replace a bound variable with another variable either bound or free. In most cases this can be replaced with permuting and reusing the existing variables such that the substitution itself is a copy. Similarly, most rules are linear in that they contract to use just a single copy

of the matched subterms. The rules will reuse a pointer to the existing copy for the first use.

6 Conclusions

CRSX is the result of more than three years of development [5], and is beginning to mature. It is definitely there and can be played with, even if the documentation is not scheduled to be properly available until "later this year." The author encourages anyone interested in compilers and (especially) higher order rewriting to download and try CRSX; if you furthermore wish to participate I will be thrilled!

Current work is rather focused on getting a proper understanding of the formal underpinnings of the extension that have seemed necessary to get CRSX to do what it does, and also the cleanest way to allow rewrite systems themselves direct access to the "formal toolbox" that is effectively embedded in the system.

In addition, CRSX is in full production use as a compiler generator framework, which the author plans to keep pushing as far as it can because it is only getting more interesting with time.

Acknowledgements. I am grateful to our "Compilers by Higher Order Rewriting" team at IBM Research: Scott Boag, Takahide Nogoyama, Naoto Sato, Lionel Villard, and Bob Schloss, for participating in the great adventure it is to build a compiler mostly from scratch in this way (as well as serve as first line of defense against interesting if sometimes unintended features), to Morris Matsa for believing that such compilers can eventually be effective, to the anonymous referees for numerous helpful suggestions, and finally, thanks to SourceForge for enabling the disseminiation of projects such as this.

References

1 Peter Aczel. A general Church-Rosser theorem. `http://www.ens-lyon.fr/LIP/REWRITING/MISC/AGeneralChurch-RosserTheorem.pdf`, July 1978. Corrections at `http://www.ens-lyon.fr/LIP/REWRITING/MISC/AGRT_corrections.pdf`.

2 Jan Willem Klop. *Combinatory Reduction Systems*. PhD thesis, University of Utrecht, 1980. Also available as Mathematical Centre Tracts 127.

3 Jan Willem Klop, Vincent van Oostrom, and Femke van Raamsdonk. Combinatory reduction systems: Introduction and survey. *Theoretical Computer Science*, 121:279–308, 1993.

4 Frank Pfenning and Conal Elliot. Higher-order abstract syntax. *SIGPLAN Notices*, 23(7):199–208, 1988.

5 Kristoffer Rose. CRSX – an open source platform for experimenting with higher order rewriting. Presented in absentia at HOR 2007—`http://kristoffer.rose.name/papers`, June 2007.

6 Kristoffer Rose. Higher-order rewriting for executable compiler specifications. In Eduardo Bonelli, editor, *Proceedings 5th International Workshop on Higher-Order Rewriting*, volume 49 of *EPTCS*, pages 31–45, Edinburgh, Scotland, July 2010.

7 Kristoffer Rose. Combinatory reduction systems with extensions. `http://crsx.sourceforge.net`, March 2011.

8 Sreeni Viswanadha, Sriram Sankar, et al. *Java Compiler Compiler (JavaCC) - The Java Parser Generator*. Sun, 4.0 edition, January 2006.

A Reduction-Preserving Completion for Proving Confluence of Non-Terminating Term Rewriting Systems

Takahito Aoto[1] **and Yoshihito Toyama**[1]

1 **RIEC, Tohoku University**
2-1-1 Katahira, Aoba-ku, Sendai, Miyagi, 980-8577, Japan
{aoto,toyama}@nue.riec.tohoku.ac.jp

Abstract

We give a method to prove confluence of term rewriting systems that contain non-terminating rewrite rules such as commutativity and associativity. Usually, confluence of term rewriting systems containing such rules is proved by treating them as equational term rewriting systems and considering E-critical pairs and/or termination modulo E. In contrast, our method is based solely on usual critical pairs and usual termination. We first present confluence criteria for term rewriting systems whose rewrite rules can be partitioned into terminating part and possibly non-terminating part. We then give a reduction-preserving completion procedure so that the applicability of the criteria is enhanced. In contrast to the well-known Knuth-Bendix completion procedure which preserves the equivalence relation of the system, our completion procedure preserves the reduction relation of the system, by which confluence of the original system is inferred from that of the completed system.

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

Keywords and phrases Confluence, Completion, Equational Term Rewriting Systems, Confluence Modulo Equations

Digital Object Identifier 10.4230/LIPIcs.RTA.2011.91

Category Regular Research Paper

1 Introduction

Confluence is one of the most important properties of term rewriting systems (TRSs for short) and hence many efforts have been spent on developing techniques to prove this property [3, 15]. One of the classes of TRSs for which many known confluence proving methods are not effective is the class of TRSs containing associativity and commutativity rules (AC-rules). Such TRSs are non-terminating by the existence of AC-rules (more precisely, commutativity rules are self-looping and associativity rules are looping under the presence of commutativity rules) and hence the Knuth-Bendix criterion does not apply. Furthermore, confluence criteria regardless of termination based on critical pairs often do not apply either.

A well-known approach to deal with TRSs containing AC-rules is to deal them as equational term rewriting systems [6, 7, 13]. In this approach, non-terminating rules such as AC-rules are treated exceptionally as an equational subsystem $\mathcal{E}$. Then the confluence of equational term rewriting system $\langle \mathcal{R}, \mathcal{E} \rangle$ is obtained if $\mathcal{R}$ is terminating modulo $\mathcal{E}$ [6, 7, 13]

22nd International Conference on Rewriting Techniques and Applications.
Editor: M. Schmidt-Schauß; pp. 91–106

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


and either $\mathcal{E}$-critical pairs of $\mathcal{R}$ satisfy certain conditions [7, 13] or $\mathcal{R}$ is left-linear and $\mathcal{E}/\mathcal{R}$-critical pairs satisfy a certain condition [6]. This approach, however, only works if $\mathcal{R}$ is terminating modulo $\mathcal{E}$. Furthermore, the computation of $\mathcal{E}$-critical pairs requires a finite and complete $\mathcal{E}$-unification algorithm which depends on $\mathcal{E}$.

In this paper, we give a method to prove confluence of TRSs that contain non-terminating rewrite rules such as AC-rules. In contrast to the traditional approach described above, our method is based solely on usual critical pairs and usual termination. Thus the implementation of the method requires little special ingredients and the method is easily integrated into confluence provers to combine with other confluence proving methods. We first present confluence criteria for TRSs whose rewrite rules can be partitioned into terminating part and possibly non-terminating part (Section 3). We then give a reduction-preserving completion procedure so that the applicability of the criteria is enhanced (Section 4). In contrast to the well-known Knuth-Bendix completion procedure which preserves the equivalence relation of the system, our completion procedure preserves the reduction relation of the system, by which confluence of the original system is inferred from that of the completed system. Finally we report on our implementation and results of experiments (Section 5).

2 Preliminaries

This section fixes some notions and notations used in this paper. We refer to [3] for omitted definitions.

Let $\to$ be a relation on a set A. The *reflexive closure* (the *symmetric closure*, the *transitive closure*, the *reflexive and transitive closure*, the *equivalence closure*) of $\to$ is denoted by $\overline{\to}$ ($\leftrightarrow$, $\overset{+}{\to}$, $\overset{*}{\to}$ $\overset{*}{\leftrightarrow}$, respectively). The union $\to_i \cup \to_j$ of indexed relations $\to_i$ and $\to_j$ is written as $\to_{i\cup j}$. A symmetric relation is written as $\vdash\!\dashv$. A relation $\to$ is *well-founded* if there exists no infinite descending chain $a_0 \to a_1 \to \cdots$. The composition of relations R, S is written as $R \circ S$. A relation $\to$ on a set A is *confluent* if $\overset{*}{\leftarrow} \circ \overset{*}{\to} \subseteq \overset{*}{\to} \circ \overset{*}{\leftarrow}$ holds.

Let $\mathcal{F}$ be a set of arity-fixed function symbols and $\mathcal{V}$ be the set of variables. The set of terms over $\mathcal{F}$ and $\mathcal{V}$ is denoted by $\mathrm{T}(\mathcal{F}, \mathcal{V})$. The sets of function symbols and variables occurring in a term t are denoted by $\mathcal{F}(t)$ and $\mathcal{V}(t)$, respectively. A *linear* term is a term in which any variable occur at most once. *Positions* are finite sequences of positive integers. The *empty sequence* is denoted by ϵ. The set of positions in a term t is denoted by $\mathrm{Pos}(t)$. The *concatenation* of positions p, q is denoted by $p.q$. We use $\leq$ for prefix ordering on positions, i.e. $p \leq q$ iff $\exists o.\ p.o = q$. For p, q such that $p \leq q$, the position o satisfying $p.o = q$ is denoted by $p\backslash q$. Positions $p_1, \ldots, p_n$ are *parallel* if $p_i \not\leq p_j$ for any $i \neq j$. We write $p \parallel q$ if two positions p, q are parallel. If p is a position in a term t, then the symbol in t at the position p is written as $t(p)$, the subterm of t at the position p is written as t/p, and the term obtained by replacing the subterm t/p by a term s is written as $t[s]_p$. For $X \subseteq \mathcal{F} \cup \mathcal{V}$, we put $\mathrm{Pos}_X(t) = \{p \in \mathrm{Pos}(t) \mid t(p) \in X\}$. For parallel positions $p_1, \ldots, p_n$ in a term t, the term obtained by replacing each subterm t/p_i by a term s_i is written as $t[s_1, \ldots, s_n]_{p_1,\ldots,p_n}$. A map σ from $\mathcal{V}$ to $\mathrm{T}(\mathcal{F}, \mathcal{V})$ is a *substitution* if the domain $\mathrm{dom}(\sigma)$ of σ is finite where $\mathrm{dom}(\sigma) = \{x \in \mathcal{V} \mid \sigma(x) \neq x\}$. As usual, we identify each substitution with its homomorphic extension. For a substitution σ and a term t, $\sigma(t)$ is also written as $t\sigma$. For a set $\mathcal{E}$ of equations, we write $\mathcal{E}^{-1} = \{r \approx l \mid l \approx r \in \mathcal{E}\}$. A set $\mathcal{E} = \{s_1 \approx t_1, \ldots, s_n \approx t_n\}$ of equations is *unifiable* if there exists a substitution σ such that $s_i\sigma = t_i\sigma$ for all i; the substitution σ is a *unifier* of $\mathcal{E}$. A relation R on $\mathrm{T}(\mathcal{F}, \mathcal{V})$ is *stable* if for any terms $s, t \in \mathrm{T}(\mathcal{F}, \mathcal{V})$, $s\,R\,t$ implies $s\theta\,R\,t\theta$ for any substitution θ; it is *monotone* if $s\,R\,t$ implies $f(\ldots, s, \ldots)\,R\,f(\ldots, t, \ldots)$ for any $f \in \mathcal{F}$. A relation R on $\mathrm{T}(\mathcal{F}, \mathcal{V})$ is a *rewrite relation* if it is stable and monotone.

An equation $l \approx r$ is a *rewrite rule* if it satisfies the conditions (1) $l \notin \mathcal{V}$ and (2) $\mathcal{V}(l) \subseteq \mathcal{V}(r)$. A rewrite rule $l \approx r$ is written as $l \to r$. Rewrite rules are identified modulo renaming of variables. A rewrite rule $l \to r$ is *linear* (*left-linear*) if l, r is linear (l is linear, respectively); it is *bidirectional* if $r \approx l$ is a rewrite rule. A *term rewriting system* (*TRS* for short) is a finite set of rewrite rules. A TRS is left-linear (linear, bidirectional) if so are all its rewrite rules. If a TRS $\mathcal{R}$ is bidirectional then $\mathcal{R}^{-1} = \{r \to l \mid l \to r \in \mathcal{R}\}$ is a TRS. Let $\mathcal{R}$ be a TRS. If there exists a rewrite rule $l \to r \in \mathcal{R}$ and a position p in a term s and substitution θ such that $s/p = l\theta$ and $t = s[r\theta]_p$, we write $s \to_{p,\mathcal{R}} t$. If not necessary, $s \to_{p,\mathcal{R}} t$ is written as $s \to_{\mathcal{R}} t$ or $s \to t$. We call $s \to_{\mathcal{R}} t$ a *rewrite step*; $\to_{\mathcal{R}}$ is a rewrite relation and called *the* rewrite relation of $\mathcal{R}$. A term s is *normal* if $s \to_{\mathcal{R}} t$ for no term t. The set of normal terms is denoted by $\mathrm{NF}(\mathcal{R})$. A *normal form* (or $\mathcal{R}$-*normal form*) of a term s is a term $t \in \mathrm{NF}(\mathcal{R})$ such that $s \xrightarrow{*}_{\mathcal{R}} t$. A TRS $\mathcal{R}$ is *terminating* if $\to_{\mathcal{R}}$ is well-founded; $\mathcal{R}$ is confluent if $\to_{\mathcal{R}}$ is confluent. The *parallel extension* $\Rightarrow_{\mathcal{R}}$ of the rewrite relation $\to_{\mathcal{R}}$ and the parallel extension $\Leftrightarrow\!\!\!\!\Rightarrow_{\mathcal{R}}$ of the symmetric closure $\leftrightarrow_{\mathcal{R}}$ of the rewrite relation $\to_{\mathcal{R}}$ are defined like this: $s \Rightarrow_{\{p_1,\ldots,p_n\},\mathcal{R}} t$ ($s \Leftrightarrow\!\!\!\!\Rightarrow_{\{p_1,\ldots,p_n\},\mathcal{R}} t$) iff $p_1, \ldots, p_n$ are parallel positions in the term s and there exist rewrite rules $l_1 \to r_1, \ldots, l_n \to r_n \in \mathcal{R}$ (equations $l_1 \approx r_1, \ldots, l_n \approx r_n \in \mathcal{R} \cup \mathcal{R}^{-1}$, respectively) and substitution $\theta_1, \ldots, \theta_n$ such that $s/p_i = l_i\theta_i$ for each i and $t = s[r_1\theta_1, \ldots, r_n\theta_n]_{p_1,\ldots,p_n}$. If not necessary, $s \Rightarrow_{\{p_1,\ldots,p_n\},\mathcal{R}} t$ ($s \Leftrightarrow\!\!\!\!\Rightarrow_{\{p_1,\ldots,p_n\},\mathcal{R}} t$) is written as $s \Rightarrow_{\mathcal{R}} t$ or $s \Rightarrow t$ ($s \Leftrightarrow\!\!\!\!\Rightarrow_{\mathcal{R}} t$ or $s \Leftrightarrow\!\!\!\!\Rightarrow t$, respectively). We call $s \Rightarrow_{\mathcal{R}} t$ a *parallel rewrite step*. We note that $\Rightarrow_{\mathcal{R}}$ is a reflexive rewrite relation and $\Leftrightarrow\!\!\!\!\Rightarrow_{\mathcal{R}}$ is a reflexive symmetric rewrite relation. Note that $\Leftrightarrow\!\!\!\!\Rightarrow_{\mathcal{R}}$ differs from the symmetric closure of $\Rightarrow_{\mathcal{R}}$ in general and coincides with $\Rightarrow_{\mathcal{R}\cup\mathcal{R}^{-1}}$ if $\mathcal{R}$ is bidirectional.

Let s, t be terms whose variables are disjoint. The term s *overlaps* on t (at a 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 a position p and σ is the most general unifier of l_1 and l_2/p. Then the term $l_2[l_1]_p\sigma$ yields a *critical pair* $\langle l_2[r_1]_p\sigma, r_2\sigma\rangle$ obtained by the overlap of $l_1 \to r_1$ on $l_2 \to r_2$ at the 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 $p = \epsilon$. We call the critical pair *outer* if $p = \epsilon$ and *inner* if $p > \epsilon$. The set of outer (inner) critical pairs obtained by the overlaps of a rewrite rule from $\mathcal{R}_1$ on a rewrite rule from $\mathcal{R}_2$ is denoted by $\mathrm{CP}_{out}(\mathcal{R}_1, \mathcal{R}_2)$ ($\mathrm{CP}_{in}(\mathcal{R}_1, \mathcal{R}_2)$, respectively). We put $\mathrm{CP}(\mathcal{R}_1, \mathcal{R}_2) = \mathrm{CP}_{out}(\mathcal{R}_1, \mathcal{R}_2) \cup \mathrm{CP}_{in}(\mathcal{R}_1, \mathcal{R}_2)$. Critical pairs are often regarded as equations.

3 Confluence criteria

In this section, we give new confluence criteria for term rewriting systems. We first present an abstract confluence criterion that will be used as the basis of our confluence criteria.

▶ **Lemma 3.1.** *Let $\vdash\!\!\!\to_0, \to_1$ be relations on a set A such that $\vdash\!\!\!\to_0$ is symmetric and $\to_1$ is well-founded. Let $\to_{0\cup1} = \vdash\!\!\!\to_0 \cup \to_1$. Suppose (i) $\leftarrow_1 \circ \to_1 \subseteq \xrightarrow{*}_1 \circ \overset{=}{\vdash\!\!\!\to}_0 \circ \xleftarrow{*}_1$ and (ii) $\vdash\!\!\!\to_0 \circ \to_1 \subseteq \xrightarrow{*}_1 \circ \overset{=}{\vdash\!\!\!\to}_0 \circ \xleftarrow{*}_1$. Then $\overset{*}{\leftrightarrow}_{0\cup1} \subseteq \xrightarrow{*}_1 \circ \vdash\!\!\!\to^*_0 \circ \xleftarrow{*}_1$.*

Proof. Let the weight of a rewrite step $a \leftrightarrow_{0\cup1} b$ be given by the multiset $w(a \leftrightarrow_{0\cup1} b)$ defined like this: $w(a \vdash\!\!\!\to_0 b) = \{a, b\}$, $w(a \to_1 b) = \{a\}$ and $w(a \leftarrow_1 b) = \{b\}$. For each rewrite sequence $a_0 \leftrightarrow_{0\cup1} a_1 \leftrightarrow_{0\cup1} \cdots \leftrightarrow_{0\cup1} a_n$ let its weight be the multiset consisting of the weights of the rewrite steps $a_i \leftrightarrow_{0\cup1} a_{i+1}$, i.e. $\{w(a_0 \leftrightarrow_{0\cup1} a_1), w(a_1 \leftrightarrow_{0\cup1} a_2), \ldots, w(a_{n-1} \leftrightarrow_{0\cup1} a_n)\}$. Let $\gg$ be the multiset extension of the well-founded order $\xrightarrow{+}_1$ and $\gg_{\mathrm{mul}}$ the multiset extension of $\gg$. We show by noetherian induction on the weight of the rewrite sequence

w.r.t. $\gg_{\mathrm{mul}}$ that for any rewrite sequence $a_0 \overset{*}{\leftrightarrow}_{0\cup 1} a_n$ there exists a rewrite sequence $a_0 \overset{*}{\to}_1 \circ \overset{*}{\vdash\!\dashv}_0 \circ \overset{*}{\leftarrow}_1 a_n$.

1. Suppose there exists k such that $a_{k-1} \leftarrow_1 a_k \to_1 a_{k+1}$. Then by assumption (i), there exist $b_0, \ldots, b_m$ such that $a_{k-1} = b_0 \overset{*}{\to}_1 b_l \overset{=}{\vdash\!\dashv}_0 b_{l+1} \overset{*}{\leftarrow}_1 b_m = a_{k+1}$. Thus we have a rewrite sequence $a_0 \overset{*}{\leftrightarrow}_{0\cup 1} a_{k-1} = b_0 \overset{*}{\to}_1 b_l \overset{=}{\vdash\!\dashv}_0 b_{l+1} \overset{*}{\leftarrow}_1 b_m = a_{k+1} \overset{*}{\leftrightarrow}_{0\cup 1} a_n$. We now show this new rewrite sequence has less weight than the original rewrite sequence $a_0 \overset{*}{\leftrightarrow}_{0\cup 1} a_n$. We here only show the case of $l \neq 0$, $l+1 \neq m$ and $b_l \vdash\!\dashv_0 b_{l+1}$. Then the weight decreases as $\{\ldots, \{a_k\}, \{a_k\}, \ldots\} \gg_{\mathrm{mul}} \{\ldots, \{b_0\}, \ldots, \{b_{l-1}\}, \{b_l, b_{l+1}\}, \{b_{l+2}\}, \ldots, \{b_m\}, \ldots\}$. For other cases, one can easily check that the weight of the rewrite sequence decreases in a similar way. Thus, it follows that there exists a rewrite sequence $a_0 \overset{*}{\to}_1 \circ \overset{*}{\vdash\!\dashv}_0 \circ \overset{*}{\leftarrow}_1 a_n$ by the induction hypothesis.
2. Suppose that there exists k such that $a_{k-1} \vdash\!\dashv_0 a_k \to_1 a_{k+1}$. Then by assumption (ii), there exist $b_0, \ldots, b_m$ such that $a_{k-1} = b_0 \overset{*}{\to}_1 b_l \overset{=}{\vdash\!\dashv}_0 b_{l+1} \overset{*}{\leftarrow}_1 b_m = a_{k+1}$. Thus we have a rewrite sequence $a_0 \overset{*}{\leftrightarrow}_{0\cup 1} a_{k-1} = b_0 \overset{*}{\to}_1 b_l \overset{=}{\vdash\!\dashv}_0 b_{l+1} \overset{*}{\leftarrow}_1 b_m = a_{k+1} \overset{*}{\leftrightarrow}_{0\cup 1} a_n$. In a way similar to the first case, one can easily check that this new rewrite sequence has less weight than the original rewrite sequence $a_0 \overset{*}{\leftrightarrow}_{0\cup 1} a_n$. We here only show the case of $l \neq 0$, $l+1 \neq m$ and $b_l \vdash\!\dashv_0 b_{l+1}$. Then the weight decreases as $\{\ldots, \{b_0, a_k\}, \{a_k\}, \ldots\} \gg_{\mathrm{mul}} \{\ldots, \{b_0\}, \ldots, \{b_{l-1}\}, \{b_l, b_{l+1}\}, \{b_{l+2}\}, \ldots, \{b_m\}, \ldots\}$. Thus, it follows that there exists a rewrite sequence $a_0 \overset{*}{\to}_1 \circ \overset{*}{\vdash\!\dashv}_0 \circ \overset{*}{\leftarrow}_1 a_n$ by the induction hypothesis.
3. Suppose that there exists k such that $a_{k-1} \leftarrow_1 a_k \vdash\!\dashv_0 a_{k+1}$. Then one can show that there exists a rewrite sequence $a_0 \overset{*}{\to}_1 \circ \overset{*}{\vdash\!\dashv}_0 \circ \overset{*}{\leftarrow}_1 a_n$ in the same way as the case (2).
4. It remains to show the case that (α) there exists no k such that $a_{k-1} \leftarrow_1 a_k \to_1 a_{k+1}$, ($\beta$) there exists no k such that $a_{k-1} \vdash\!\dashv_0 a_k \to_1 a_{k+1}$ and (γ) there exists no k such that $a_{k-1} \leftarrow_1 a_k \vdash\!\dashv_0 a_{k+1}$. We show by induction on the length of $a_0 \overset{*}{\leftrightarrow}_{0\cup 1} a_n$ that this rewrite sequence has the form $a_0 \overset{*}{\to}_1 \circ \overset{*}{\vdash\!\dashv}_0 \circ \overset{*}{\leftarrow}_1 a_n$. The case $n = 0$ is trivial. Suppose $a_0 \leftrightarrow_{0\cup 1} a_1 \overset{*}{\leftrightarrow}_{0\cup 1} a_n$. By induction hypothesis we have $a_1 \overset{*}{\to}_1 a_l \overset{*}{\vdash\!\dashv}_0 a_m \overset{*}{\leftarrow}_1 a_n$. We distinguish three cases:
   a. Case of $a_0 \vdash\!\dashv_0 a_1$. By (β), it follows that we have $a_0 \vdash\!\dashv_0 a_1 = a_l \overset{*}{\vdash\!\dashv}_0 a_m \overset{*}{\leftarrow}_1 a_n$. Hence the conclusion follows.
   b. Case of $a_0 \to_1 a_1$. Since we have $a_0 \to_1 a_1 \overset{*}{\to}_1 a_l \overset{*}{\vdash\!\dashv}_0 a_m \overset{*}{\leftarrow}_1 a_n$, the conclusion follows.
   c. Case of $a_0 \leftarrow_1 a_1$. Then by (α), it follows that we have $a_0 \leftarrow_1 a_1 = a_l \overset{*}{\vdash\!\dashv}_0 a_m \overset{*}{\leftarrow}_1 a_n$. Furthermore, by ($\gamma$), it follows that $a_0 \leftarrow_1 a_1 = a_l = a_m \overset{*}{\leftarrow}_1 a_n$. Hence the conclusion follows.

◀

▶ Remark. Let $\sim$ be an equivalence relation on a set A. Then a relation $\to_1$ on A is said to be *confluent modulo* $\sim$ (CR$\sim$) if $\overset{*}{\leftarrow}_1 \circ \sim \circ \overset{*}{\to}_1 \subseteq \overset{*}{\to}_1 \circ \sim \circ \overset{*}{\leftarrow}_1$ holds; *locally confluent modulo* $\sim$ (LCR$\sim$) if (i′) $\leftarrow_1 \circ \to_1 \subseteq \overset{*}{\to}_1 \circ \sim \circ \overset{*}{\leftarrow}_1$ and (ii′) $\sim \circ \to_1 \subseteq \overset{*}{\to}_1 \circ \sim \circ \overset{*}{\leftarrow}_1$ hold [6]. It is shown in [6] that CR$\sim$ and LCR$\sim$ coincide provided that $\to_1$ is well-founded. Suppose $\to_1$ is well-founded and $\sim = \overset{*}{\vdash\!\dashv}_0$. Then the property CR$\sim$ is equivalent to the conclusion of the lemma, i.e. $\overset{*}{\leftrightarrow}_{0\cup 1} \subseteq \overset{*}{\to}_1 \circ \overset{*}{\vdash\!\dashv}_0 \circ \overset{*}{\leftarrow}_1$; hence so are (i′) and (ii′). In our lemma, in contrast to (i′) and (ii′), the condition part of (ii) are localized (i.e. we only assume $\vdash\!\dashv_0 \circ \to_1$ rather than $\overset{*}{\vdash\!\dashv}_0 \circ \to_1$) in price of requesting joinability sequences to have zero or one $\vdash\!\dashv_0$-step in the conclusion part of (i) and (ii) (i.e. we need to guarantee $\overset{*}{\to}_1 \circ \overset{=}{\vdash\!\dashv}_0 \circ \overset{*}{\leftarrow}_1$ rather than $\overset{*}{\to}_1 \circ \overset{*}{\vdash\!\dashv}_0 \circ \overset{*}{\leftarrow}_1$). A different localization given in [6] is that if $\to_1 \circ \sim$ is well-founded then (i′) $\leftarrow_1 \circ \to_1 \subseteq \overset{*}{\to}_1 \circ \sim \circ \overset{*}{\leftarrow}_1$ and (iii′) $\vdash\!\dashv_0 \circ \to_1 \subseteq \overset{*}{\to}_1 \circ \sim \circ \overset{*}{\leftarrow}_1$ imply CR$\sim$. Contrast to our lemma, this localization allows an arbitrary number of $\vdash\!\dashv_0$-steps in the conclusion part of (i′) and (iii′) in price of requesting (not only $\to_1$ but) $\to_1 \circ \sim$ is well-founded. In [8] (see also

[9]), another localization is obtained: if $\to_1$ is well-founded then (i′) $\leftarrow_1 \circ \to_1 \subseteq \overset{*}{\to}_1 \circ \sim \circ \overset{*}{\leftarrow}_1$ and (iv′) $\vdash\!\!\dashv_0 \circ \to_1 \subseteq \overset{+}{\to}_1 \circ \sim$ imply CR$\sim$ (and that $\to_1 \circ \sim$ is well-founded). Contrast to our lemma, this localization allows an arbitrary number of $\vdash\!\!\dashv_0$-steps in the conclusion part of (i′) and (iv′) in price of restricting the form of joinability sequences in the conclusion part of (iv′).

▶ **Theorem 3.2** (abstract confluence criterion). *Let $\vdash\!\!\dashv_0, \to_1$ be relations on a set A such that $\vdash\!\!\dashv_0$ is symmetric and $\to_1$ is well-founded. Let $\to_{0\cup1} = \vdash\!\!\dashv_0 \cup \to_1$. Suppose (i) $\leftarrow_1 \circ \to_1 \subseteq \overset{*}{\to}_1 \circ \overset{=}{\vdash\!\!\dashv}_0 \circ \overset{*}{\leftarrow}_1$ and (ii) $\vdash\!\!\dashv_0 \circ \to_1 \subseteq \overset{*}{\to}_1 \circ \overset{=}{\vdash\!\!\dashv}_0 \circ \overset{*}{\leftarrow}_1$. Then $\to_{0\cup1}$ is confluent.*

Proof. We prove $\overset{*}{\leftrightarrow}_{0\cup1} \subseteq \overset{*}{\to}_{0\cup1} \circ \overset{*}{\leftarrow}_{0\cup1}$. Suppose $a \overset{*}{\leftrightarrow}_{0\cup1} b$. Then $a \overset{*}{\to}_1 \circ \overset{*}{\vdash\!\!\dashv}_0 \circ \overset{*}{\leftarrow}_1 b$ by Lemma 3.1. Hence $a \overset{*}{\to}_{0\cup1} \circ \overset{*}{\leftarrow}_{0\cup1} b$. ◀

For the rest of this section, we develop some confluence criteria for TRSs based on this abstract confluence criterion.

▶ **Lemma 3.3.** *Let $\mathcal{S}$ be a TRS and $\vdash\!\!\dashv$ be a symmetric rewrite relation. Suppose that $\mathrm{CP}(\mathcal{S},\mathcal{S}) \subseteq \overset{*}{\to}_{\mathcal{S}} \circ \overset{=}{\vdash\!\!\dashv} \circ \overset{*}{\leftarrow}_{\mathcal{S}}$. Then $\leftarrow_{\mathcal{S}} \circ \to_{\mathcal{S}} \subseteq \overset{*}{\to}_{\mathcal{S}} \circ \overset{=}{\vdash\!\!\dashv} \circ \overset{*}{\leftarrow}_{\mathcal{S}}$.*

Proof. Suppose $t_0 \leftarrow_{p,\mathcal{S}} s \to_{q,\mathcal{S}} t_1$. We distinguish the cases by relative positions of p and q. The case of $p \parallel q$ is straightforward. Suppose $q \le p$. Let $s/q = l\sigma$ and $l \to r \in \mathcal{S}$. Then either (1) $q\backslash p \in \mathrm{Pos}_{\mathcal{F}}(l)$ or (2) there exists $q_x \in \mathrm{Pos}_{\mathcal{V}}(l)$ such that $l/q_x = x \in \mathcal{V}$ and $q.q_x \le p$.

1. Then $t_0 = s[u\rho]_q$ and $t_1 = s[v\rho]_q$ for some $\langle u, v\rangle \in \mathrm{CP}(\mathcal{S},\mathcal{S})$ and substitution ρ. Thus by assumption $u \overset{*}{\to}_{\mathcal{S}} u' \overset{=}{\vdash\!\!\dashv} v' \overset{*}{\leftarrow}_{\mathcal{S}} v$ for some u', v'. Then, since $\overset{=}{\vdash\!\!\dashv}$ and $\to_{\mathcal{S}}$ are rewrite relations, we have $t_0 = s[u\rho]_q \overset{*}{\to}_{\mathcal{S}} s[u'\rho]_q \overset{=}{\vdash\!\!\dashv} s[v'\rho]_q \overset{*}{\leftarrow}_{\mathcal{S}} s[v\rho]_q = t_1$.
2. Then $t_1 = s[r\sigma]_q$ and $s = s[l\sigma]_q \to_{p,\mathcal{S}} t_0 \overset{*}{\to}_{\mathcal{S}} s[l\sigma']_q$ for some substitution σ' such that $\sigma(x) \to_{(q.q_x)\backslash p,\mathcal{S}} \sigma'(x)$ and $\sigma'(y) = \sigma(y)$ for any $y \ne x$. Thus $t_0 \overset{*}{\to}_{\mathcal{S}} s[l\sigma']_q \to_{\mathcal{S}} s[r\sigma']_q \overset{*}{\leftarrow}_{\mathcal{S}} s[r\sigma]_q = t_1$. The claim follows since $\overset{=}{\vdash\!\!\dashv}$ is reflexive.

The case of $p \le q$ follows similarly to the case of $q \le p$, using the symmetry of $\overset{=}{\vdash\!\!\dashv}$. ◀

▶ **Lemma 3.4.** *Let $\mathcal{P},\mathcal{S}$ be TRSs. Suppose that $\mathrm{CP}(\mathcal{S},\mathcal{S}) \subseteq \overset{*}{\to}_{\mathcal{S}} \circ \twoheadleftarrow_{\mathcal{P}} \circ \overset{*}{\leftarrow}_{\mathcal{S}}$. Then $\leftarrow_{\mathcal{S}} \circ \to_{\mathcal{S}} \subseteq \overset{*}{\to}_{\mathcal{S}} \circ \twoheadleftarrow_{\mathcal{P}} \circ \overset{*}{\leftarrow}_{\mathcal{S}}$.*

Proof. Take $\vdash\!\!\dashv := \twoheadleftarrow_{\mathcal{P}}$ (hence $\overset{=}{\vdash\!\!\dashv} = \twoheadleftarrow_{\mathcal{P}}$) in Lemma 3.3. ◀

▶ **Lemma 3.5.** *Let $\mathcal{P},\mathcal{S}$ be TRSs such that $\mathcal{S}$ is left-linear and $\mathcal{P}$ is bidirectional. Suppose (i) $\mathrm{CP}_{in}(\mathcal{P}\cup\mathcal{P}^{-1},\mathcal{S}) = \emptyset$ and (ii) $\mathrm{CP}(\mathcal{S},\mathcal{P}\cup\mathcal{P}^{-1}) \subseteq \overset{*}{\to}_{\mathcal{S}} \circ \twoheadleftarrow_{\mathcal{P}} \circ \overset{*}{\leftarrow}_{\mathcal{S}}$. Then $\twoheadleftarrow_{\mathcal{P}} \circ \to_{\mathcal{S}} \subseteq \overset{*}{\to}_{\mathcal{S}} \circ \twoheadleftarrow_{\mathcal{P}} \circ \overset{*}{\leftarrow}_{\mathcal{S}}$.*

Proof. Suppose $t_0 \mathrel{\rotatebox{180}{$\twoheadrightarrow$}}_{U,\mathcal{P}\cup\mathcal{P}^{-1}} s \to_{q,\mathcal{S}} t_1$. Let $U = \{p_1,\dots,p_n\}$ where $p_1,\dots,p_n$ are positions from left to right, $s/p_i = l_i\sigma_i$ for $l_i \to r_i \in \mathcal{P}\cup\mathcal{P}^{-1}$ and substitutions σ_i $(1 \le i \le n)$ and $s/q = l'\rho$ for $l' \to r' \in \mathcal{S}$ and a substitution ρ. We distinguish two cases: (1) the case that $\exists p \in U.\ p \le q$ and (2) the case that $\forall p \in U.\ p \not\le q$.

1. Suppose $p_i \in U$ and $p_i \le q$. Then either (a) $p_i\backslash q \in \mathrm{Pos}_{\mathcal{F}}(l_i)$ or (b) there exists $p_x \in \mathrm{Pos}_{\mathcal{V}}(l_i)$ such that $l_i/p_x = x \in \mathcal{V}$ and $p_i.p_x \le q$.
   a. Then $t_0/p_i = v\rho$ and $t_1/p_i = u\rho$ for some $\langle u, v\rangle \in \mathrm{CP}(\mathcal{S},\mathcal{P}\cup\mathcal{P}^{-1})$ and substitution ρ. Then, from our assumption (ii), we have $u \overset{*}{\to}_{\mathcal{S}} u' \twoheadleftarrow_{\mathcal{P}} v' \overset{*}{\leftarrow}_{\mathcal{S}} v$ for some u', v'. Thus $t_0/p_i = v\rho \overset{*}{\to}_{\mathcal{S}} v'\rho \twoheadleftarrow_{\mathcal{P}} u'\rho \overset{*}{\leftarrow}_{\mathcal{S}} u\rho = t_1/p_i$. Hence we have

$$\begin{array}{rcl} t_0 & = & s[r_1\sigma_1,\dots,t_0/p_i,\dots,r_n\sigma_n]_{p_1,\dots,p_i,\dots,p_n} \\ & \overset{*}{\to}_{\mathcal{S}} & s[r_1\sigma_1,\dots,v'\rho,\dots,r_n\sigma_n]_{p_1,\dots,p_i,\dots,p_n} \\ & \twoheadleftarrow_{\mathcal{P}} & s[l_1\sigma_1,\dots,u'\rho,\dots,l_n\sigma_n]_{p_1,\dots,p_i,\dots,p_n} \\ & \overset{*}{\leftarrow}_{\mathcal{S}} & s[l_1\sigma_1,\dots,t_1/p_i,\dots,l_n\sigma_n]_{p_1,\dots,p_i,\dots,p_n} = t_1. \end{array}$$

b. Then $t_0/p_i = r_i\sigma_i$ and $t_1/p_i \xrightarrow{*}_{\mathcal{S}} l_i\sigma'_i$ for some substitution σ'_i such that $\sigma_i(x) \to_{(p_i.p_x)\backslash q,\mathcal{S}} \sigma'_i(x)$ and $\sigma'_i(y) = \sigma_i(y)$ for any $y \neq x$. Thus we have

$$\begin{array}{lll} t_0 & = & C[r_1\sigma_1,\dots,r_i\sigma_i,\dots,r_n\sigma_n]_{p_1,\dots,p_i,\dots,p_n} \\ & \xrightarrow{*}_{\mathcal{S}} & C[r_1\sigma_1,\dots,r_i\sigma'_i,\dots,r_n\sigma_n]_{p_1,\dots,p_i,\dots,p_n} \\ & \Mapsto_{\mathcal{P}} & C[l_1\sigma_1,\dots,l_i\sigma'_i,\dots,l_n\sigma_n]_{p_1,\dots,p_i,\dots,p_n} \\ & \xleftarrow{*}_{\mathcal{S}} & C[l_1\sigma_1,\dots,t_1/p_i,\dots,l_n\sigma_n]_{p_1,\dots,p_i,\dots,p_n} = t_1. \end{array}$$

2. Suppose $\forall p \in U.\ p \not\leq q$. Let $U' = \{p_i \in U \mid q < p_i\} = \{p_l,\dots,p_k\}$, $q_i = q\backslash p_i$ for $l \leq i \leq k$, and thus $l'\rho = l'\rho[l_l\sigma_l,\dots,l_k\sigma_k]_{q_l,\dots,q_k}$. By our assumption (i), for each $p_i \in U'$ there exists $q_x \in \mathrm{Pos}_{\mathcal{V}}(l')$ such that $l'/q_x = x \in \mathcal{V}$ and $q.q_x \leq p_i$. Thus, $s/q = l'\rho = l'\rho[l_l\sigma_l,\dots,l_k\sigma_k]_{q_l,\dots,q_k} \to_{\mathcal{S}} r'\rho = r'\rho[l_{j_1}\sigma_{j_1},\dots,l_{j_m}\sigma_{j_m}]_{o_1,\dots,o_m} = t_1/q$ for some positions $o_1,\cdots,o_m$ and $j_1,\dots,j_m \in \{l,\dots,k\}$. Furthermore, by the left-linearity of $\mathcal{S}$, we have $l'\rho[r_l\sigma_l,\dots,r_k\sigma_k]_{q_l,\dots,q_k} \to_{\mathcal{S}} r'\rho[r_{j_1}\sigma_{j_1},\dots,r_{j_m}\sigma_{j_m}]_{o_1,\dots,o_m}$. Thus,

$$\begin{array}{lll} t_0 & = & s[r_1\sigma_1,\dots,l'\rho[r_l\sigma_l,\dots,r_k\sigma_k]_{q_l,\dots,q_k},\dots,r_n\sigma_n]_{p_1,\dots,q,\dots,p_n} \\ & \to_{\mathcal{S}} & s[r_1\sigma_1,\dots,r'\rho[r_{j_1}\sigma_{j_1},\dots,r_{j_m}\sigma_{j_m}]_{o_1,\dots,o_m},\dots,r_n\sigma_n]_{p_1,\dots,q,\dots,p_n} \\ & \Mapsto_{\mathcal{P}} & s[l_1\sigma_1,\dots,r'\rho[l_{j_1}\sigma_{j_1},\dots,l_{j_m}\sigma_{j_m}]_{o_1,\dots,o_m},\dots,l_n\sigma_n]_{p_1,\dots,q,\dots,p_n} = t_1. \end{array}$$

◀

▶ **Definition 3.6** (reversible relation)**.** A relation $\to$ is said to be *reversible* if $\to \subseteq \xleftarrow{*}$. A TRS $\mathcal{R}$ is reversible if $\to_{\mathcal{R}}$ is reversible.

Note that, by the definition of rewrite rules, reversible TRSs are bidirectional.

▶ **Theorem 3.7** (confluence criterion)**.** *Let $\mathcal{P},\mathcal{S}$ be TRSs such that $\mathcal{S}$ is left-linear and terminating and $\mathcal{P}$ is reversible. Suppose (i)* $\mathrm{CP}(\mathcal{S},\mathcal{S}) \subseteq \xrightarrow{*}_{\mathcal{S}} \circ \Mapsto_{\mathcal{P}} \circ \xleftarrow{*}_{\mathcal{S}}$*, (ii)* $\mathrm{CP}_{in}(\mathcal{P} \cup \mathcal{P}^{-1},\mathcal{S}) = \emptyset$ *(iii)* $\mathrm{CP}(\mathcal{S},\mathcal{P} \cup \mathcal{P}^{-1}) \subseteq \xrightarrow{*}_{\mathcal{S}} \circ \Mapsto_{\mathcal{P}} \circ \xleftarrow{*}_{\mathcal{S}}$*. Then $\mathcal{S} \cup \mathcal{P}$ is confluent.*

Proof. By our assumption (i) and Lemma 3.4, we have (a) $\leftarrow_{\mathcal{S}} \circ \to_{\mathcal{S}} \subseteq \xrightarrow{*}_{\mathcal{S}} \circ \Mapsto_{\mathcal{P}} \circ \xleftarrow{*}_{\mathcal{S}}$. From our assumptions (ii) and (iii), it follows that (b) $\Mapsto_{\mathcal{P}} \circ \to_{\mathcal{S}} \subseteq \xrightarrow{*}_{\mathcal{S}} \circ \Mapsto_{\mathcal{P}} \circ \xleftarrow{*}_{\mathcal{S}}$ by Lemma 3.5. Take $\vdash\!\!\!\to_0 := \Mapsto_{\mathcal{P}}$ and $\to_1 := \to_{\mathcal{S}}$. Then, by the termination of $\mathcal{S}$, $\to_1$ is well-founded. Hence by Theorem 3.2, $\Mapsto_{\mathcal{P}} \cup \to_{\mathcal{S}}$ is confluent. Furthermore, since $\to_{\mathcal{P}}$ is reversible, $\to_{\mathcal{P}} \subseteq \Mapsto_{\mathcal{P}} \subseteq \xrightarrow{*}_{\mathcal{P}}$. Hence $\to_{\mathcal{P}\cup\mathcal{S}}$ is confluent. ◀

We are now going to slightly weaken the condition (ii) $\mathrm{CP}_{in}(\mathcal{P}\cup\mathcal{P}^{-1},\mathcal{S}) = \emptyset$ of the theorem using the notion of parallel critical pairs [5, 14]. Let $s_1,\dots,s_n,t$ be terms whose variables are disjoint. The terms $s_1,\dots,s_n$ *parallel-overlap* on t (at parallel positions $p_1,\dots,p_n$) if $t/p_i \notin \mathcal{V}$ for any $1 \leq i \leq n$ and $\{s_1 \approx t/p_1,\dots,s_n \approx t/p_n\}$ is unifiable. Let $l_1 \to r_1,\dots,l_n \to r_n$ and $l' \to r'$ be rewrite rules w.l.o.g. whose variables are mutually disjoint. Suppose that $l_1,\dots,l_n$ parallel-overlap on l' at parallel positions $p_1,\dots,p_n$ and σ is the most general unifier of $\{l_1 \approx l'/p_1,\dots,l_n \approx l'/p_n\}$. Then the term $l'[l_1,\dots,l_n]_{p_1,\dots,p_n}\sigma$ yields a *parallel critical pair* $\langle l'[r_1,\dots,r_n]_{p_1,\dots,p_n}\sigma, r'\sigma\rangle$ obtained by the parallel-overlap of $l_1 \to r_1,\dots,l_n \to r_n$ on $l' \to r'$ at positions $p_1,\dots,p_n$. In the case of self-overlap (i.e. when $n = 1$ and $l_1 \to r_1$ and $l' \to r'$ are identical modulo renaming), we do not consider the case $p_1 = \epsilon$. We write $\langle l'[r_1,\dots,r_n]_{p_1,\dots,p_n}\sigma, r'\sigma\rangle_X$ if $X = \bigcup_{1\leq i\leq n} \mathcal{V}(l'\sigma/p_i)$. We call the parallel critical pair *outer* if $n = 1$ and $p_1 = \epsilon$, and *inner* if $p_i > \epsilon$ for all i. The set of outer (inner) parallel critical pairs obtained by the parallel-overlaps of rewrite rules from $\mathcal{R}_1$ on a rewrite rule from $\mathcal{R}_2$ is denoted by $\mathrm{PCP}_{out}(\mathcal{R}_1,\mathcal{R}_2)$ ($\mathrm{PCP}_{in}(\mathcal{R}_1,\mathcal{R}_2)$, respectively). (Note, however, that $\mathrm{PCP}_{out}(\mathcal{R}_1,\mathcal{R}_2) = \mathrm{CP}_{out}(\mathcal{R}_1,\mathcal{R}_2)$.) We put $\mathrm{PCP}(\mathcal{R}_1,\mathcal{R}_2) = \mathrm{PCP}_{out}(\mathcal{R}_1,\mathcal{R}_2) \cup \mathrm{PCP}_{in}(\mathcal{R}_1,\mathcal{R}_2)$.

▶ **Lemma 3.8.** *Let $\mathcal{P},\mathcal{S}$ be TRSs such that $\mathcal{S}$ is left-linear and $\mathcal{P}$ is bidirectional. Suppose that (i) for all $\langle u,v\rangle_X \in \mathrm{PCP}_{in}(\mathcal{P}\cup\mathcal{P}^{-1},\mathcal{S})$, $u \xrightarrow{*}_{\mathcal{S}} u' \mathrel{\Vdash}_{V,\mathcal{P}} v' \xleftarrow{*}_{\mathcal{S}} v$ for some u',v' and V satisfying $\bigcup_{o\in V}\mathcal{V}(v'/o)\subseteq X$, and (ii) $\mathrm{CP}(\mathcal{S},\mathcal{P}\cup\mathcal{P}^{-1})\subseteq \xrightarrow{*}_{\mathcal{S}}\circ\mathrel{\Vdash}_{\mathcal{P}}\circ\xleftarrow{*}_{\mathcal{S}}$. Then $\mathrel{\Vdash}_{\mathcal{P}}\circ\to_{\mathcal{S}}\subseteq\xrightarrow{*}_{\mathcal{S}}\circ\mathrel{\Vdash}_{\mathcal{P}}\circ\xleftarrow{*}_{\mathcal{S}}$.*

Proof. Suppose $t_0 \mathrel{\Vdash}_{U,\mathcal{P}\cup\mathcal{P}^{-1}} s \to_{q,\mathcal{S}} t_1$. Let $U=\{p_1,\ldots,p_n\}$ where $p_1,\ldots,p_n$ are positions from left to right, $s/p_i = l_i\sigma_i$ for $l_i\to r_i\in\mathcal{P}\cup\mathcal{P}^{-1}$ and substitutions σ_i $(1\le i\le n)$ and $s/q = l'\rho$ for $l'\to r'\in\mathcal{S}$ and substitution ρ. The same proof as in Lemma 3.5 applies other than the case of $\forall p\in U.\ p\not\le q$. Let $\{p_k,\ldots,p_m\}=\{p_i\in U\mid q\le p_i\}$. For each p_i $(k\le i\le m)$ either $p_i\backslash q\in\mathrm{Pos}_{\mathcal{F}}(l')$ or there exists $q_x\in\mathrm{Pos}_{\mathcal{V}}(l')$ such that $q.q_x\le p_i$. W.l.o.g. let $\{p_k,\ldots,p_l\}=\{p_i\mid p_i\backslash q\in\mathrm{Pos}_{\mathcal{F}}(l')\}$ and $\{p_{l+1},\ldots,p_m\}=\{p_i\mid\exists q_x\in\mathrm{Pos}_{\mathcal{V}}(l').\ q.q_x\le p_i\}$. Then there exists a parallel critical pair $\langle u,v\rangle_X$ obtained from overlaps of $l_k\to r_k,\ldots,l_l\to r_l$ on $l'\to r'$ at $p_k\backslash q,\ldots,p_l\backslash q$. Then, by our assumption $u\xrightarrow{*}_{\mathcal{S}}u'\mathrel{\Vdash}_{V,\mathcal{P}}v'\xleftarrow{*}_{\mathcal{S}}v$ for some u',v' satisfying $\bigcup_{o\in V}\mathcal{V}(v'/o)\subseteq X$. Let $Y=\mathcal{V}(l'\sigma)\setminus X$. Then, since l' is linear (and $\mathcal{V}(l'),\mathcal{V}(l_l),\ldots,\mathcal{V}(l_m)$ are mutually disjoint), we have $\{l'(q_x)\mid q_x\in\mathrm{Pos}_{\mathcal{V}}(l'),\exists i\,(q.q_x\le p_i)\}\subseteq Y$. Furthermore, $t_0/q=u\theta'$ and $t_1/q=v\theta$ for some substitution θ,θ' such that $\theta(y)\xrightarrow{*}_{\mathcal{S}}\theta'(y)$ for $y\in Y$ and $\theta(z)=\theta'(z)$ for $z\notin Y$. Hence, by the left-linearity of $\mathcal{S}$, we have $u\theta'\xrightarrow{*}_{\mathcal{S}}u'\theta'$. Now we claim that any position $o_1\in\mathrm{Pos}_Y(v')$ and $o_2\in V$ are parallel. Since $Y\subseteq\mathcal{V}$, it suffices to show $o_2\not\le o_1$. If $o_2\le o_1$ then $v'/o_1\in\mathcal{V}(v'/o_2)$ holds, and hence $\mathcal{V}(v'/o_2)\cap Y\neq\emptyset$. Then, by $\mathcal{V}(v'/o_2)\subseteq X$, $X\cap Y\neq\emptyset$ holds. This is a contradiction. Hence any position $o_1\in\mathrm{Pos}_Y(u')\cup\mathrm{Pos}_Y(v')$ and $o_2\in V$ are parallel. Now, we have

$$\begin{array}{lll} t_0 & = & s[r_1\sigma_1,\ldots,u\theta',\ldots,r_n\sigma_n]_{p_1,\ldots,q,\ldots,p_n}\\ & \xrightarrow{*}_{\mathcal{S}} & s[r_1\sigma_1,\ldots,u'\theta',\ldots,r_n\sigma_n]_{p_1,\ldots,q,\ldots,p_n}\\ & \mathrel{\Vdash}_{U',\mathcal{P}} & s[l_1\sigma_1,\ldots,v'\theta,\ldots,l_n\sigma_n]_{p_1,\ldots,q,\ldots,p_n}\\ & \xleftarrow{*}_{\mathcal{S}} & s[l_1\sigma_1,\ldots,v\theta,\ldots,l_n\sigma_n]_{p_1,\ldots,q,\ldots,p_n}=t_1 \end{array}$$

where $U'=\{p_1,\ldots,p_{k-1}\}\cup\{p_{m+1},\ldots,p_n\}\cup\{q.o\mid o\in V\}\cup W$ where W is the set of descendants of $\{q_{l+1},\ldots,q_m\}$ in s along the rewrite steps $s=s[l'\theta]_q\to_{q,\mathcal{S}}t_1=s[v\theta]_q\xrightarrow{*}_{\mathcal{S}}s[v'\theta]_q=s[l_1\sigma_1,\ldots,v'\theta,\ldots,l_n\sigma_n]_{p_1,\ldots,q,\ldots,p_n}$. Clearly, $U'\setminus W$ and $U'\setminus\{q.o\mid o\in V\}$ are sets of parallel positions. Thus it remains to show that positions from W are parallel to the positions from $\{q.o\mid o\in V\}$. By the fact $\{l'(q_x)\mid q_x\in\mathrm{Pos}_{\mathcal{V}}(l'),\exists i\,(q.q_x\le p_i)\}\subseteq Y$, for any $o_1\in W$ there exists $o_y\in\mathrm{Pos}_Y(v')$ such that $q.o_y\le o_1$. Since any position $o_y\in\mathrm{Pos}_Y(v')$ and $o_2\in V$ are parallel, any $o_1\in W$ and any $q.o_2$ $(o_2\in V)$ are parallel. ◀

▶ **Theorem 3.9** (confluence criterion using parallel critical pairs)**.** *Let $\mathcal{P},\mathcal{S}$ be TRSs such that $\mathcal{S}$ is left-linear and terminating and $\mathcal{P}$ is reversible. Suppose (i) $\mathrm{CP}(\mathcal{S},\mathcal{S})\subseteq\xrightarrow{*}_{\mathcal{S}}\circ\mathrel{\Vdash}_{\mathcal{P}}\circ\xleftarrow{*}_{\mathcal{S}}$, (ii) for all $\langle u,v\rangle_X\in\mathrm{PCP}_{in}(\mathcal{P}\cup\mathcal{P}^{-1},\mathcal{S})$, $u\xrightarrow{*}_{\mathcal{S}}u'\mathrel{\Vdash}_{V,\mathcal{P}}v'\xleftarrow{*}_{\mathcal{S}}v$ for some u',v' and V satisfying $\bigcup_{q\in V}\mathcal{V}(v'/q)\subseteq X$ and (iii) $\mathrm{CP}(\mathcal{S},\mathcal{P}\cup\mathcal{P}^{-1})\subseteq\xrightarrow{*}_{\mathcal{S}}\circ\mathrel{\Vdash}_{\mathcal{P}}\circ\xleftarrow{*}_{\mathcal{S}}$. Then $\mathcal{S}\cup\mathcal{P}$ is confluent.*

Proof. Similar to proof of the Theorem 3.7, using Lemmas 3.4, 3.8. ◀

Since, by the definition of parallel critical pairs, $\mathrm{CP}_{in}(\mathcal{P}\cup\mathcal{P}^{-1},\mathcal{S})\subseteq\mathrm{PCP}_{in}(\mathcal{P}\cup\mathcal{P}^{-1},\mathcal{S})$ holds. Thus the condition (ii) of Theorem 3.7 is a particular case of condition (ii) of Theorem 3.9. Hence Theorem 3.7 is subsumed by Theorem 3.9.

▶ **Example 3.10.** Let

$$\mathcal{R}=\left\{\begin{array}{llllllll} (a) & +(0,y) & \to & y & (b) & +(\mathsf{s}(x),y) & \to & \mathsf{s}(+(x,y))\\ (c) & +(x,0) & \to & x & (d) & +(x,\mathsf{s}(y)) & \to & \mathsf{s}(+(x,y))\\ (e) & +(x,y) & \to & +(y,x) & (f) & +(+(x,y),z) & \to & +(x,+(y,z)) \end{array}\right\}.$$

Put $\mathcal{S} = \{(a), (b), (c), (d)\}$ and $\mathcal{P} = \{(e), (f)\}$. Then $\mathcal{S}$ is linear and terminating. We have $+(x, +(y, z)) \to_{\mathcal{P}} +(+(y, z), x) \to_{\mathcal{P}} +(+(z, y), x) \to_{\mathcal{P}} +(z, +(y, x)) \to_{\mathcal{P}} +(z, +(x, y)) \to_{\mathcal{P}} +(+(x, y), z)$. Thus $\mathcal{P}$ is reversible. We have $\mathrm{CP}(\mathcal{S}, \mathcal{S}) =$

$$\left\{ \begin{array}{llll} \langle \mathsf{0}, \mathsf{0} \rangle & \in \overset{*}{\leftarrow}_{\mathcal{S}} & \langle \mathsf{s}(y), \mathsf{s}(+(\mathsf{0}, y)) \rangle & \in \leftarrow_{\mathcal{S}} \\ \langle \mathsf{s}(+(x, \mathsf{0})), \mathsf{s}(x) \rangle & \in \to_{\mathcal{S}} & \langle \mathsf{s}(x), \mathsf{s}(+(x, \mathsf{0})) \rangle & \in \leftarrow_{\mathcal{S}} \\ \langle \mathsf{s}(+(\mathsf{0}, y)), \mathsf{s}(y) \rangle & \in \to_{\mathcal{S}} & \langle \mathsf{s}(+(x, \mathsf{s}(y))), \mathsf{s}(+(\mathsf{s}(x), y)) \rangle & \in \to_{\mathcal{S}} \circ \leftarrow_{\mathcal{S}} \\ \langle \mathsf{s}(+(\mathsf{s}(x), y)), \mathsf{s}(+(x, \mathsf{s}(y))) \rangle & \in \to_{\mathcal{S}} \circ \leftarrow_{\mathcal{S}} & & \end{array} \right\},$$

$\mathrm{CP}_{in}(\mathcal{P} \cup \mathcal{P}^{-1}, \mathcal{S}) = \emptyset$ and $\mathrm{CP}(\mathcal{S}, \mathcal{P} \cup \mathcal{P}^{-1}) =$

$$\left\{ \begin{array}{lll} \langle y, +(y, \mathsf{0}) \rangle & \in \leftarrow_{\mathcal{S}} & \langle +(y, z), +(\mathsf{0}, +(y, z)) \rangle \in \leftarrow_{\mathcal{S}} \\ \langle +(y, z), +(+(\mathsf{0}, y), z) \rangle & \in \leftarrow_{\mathcal{S}} & \langle +(x, z), +(+(x, \mathsf{0}), z) \rangle \in \leftarrow_{\mathcal{S}} \\ \langle \mathsf{s}(+(x, y)), +(y, \mathsf{s}(x)) \rangle & \in \leftrightarrow_{\mathcal{P}} \circ \leftarrow_{\mathcal{S}} & \\ \langle +(\mathsf{s}(+(x, y)), z), +(\mathsf{s}(x), +(y, z)) \rangle & \in \to_{\mathcal{S}} \circ \leftrightarrow_{\mathcal{P}} \circ \leftarrow_{\mathcal{S}} & \\ \langle \mathsf{s}(+(x, +(y, z))), +(+(\mathsf{s}(x), y), z) \rangle & \in \leftrightarrow_{\mathcal{P}} \circ \overset{*}{\leftarrow}_{\mathcal{S}} & \\ \langle +(x, \mathsf{s}(+(y, z))), +(+(x, \mathsf{s}(y)), z) \rangle & \in \to_{\mathcal{S}} \circ \leftrightarrow_{\mathcal{P}} \circ \overset{*}{\leftarrow}_{\mathcal{S}} & \\ \langle x, +(\mathsf{0}, x) \rangle & \in \leftarrow_{\mathcal{S}} & \langle +(x, y), +(x, +(y, \mathsf{0})) \rangle \in \leftarrow_{\mathcal{S}} \\ \langle +(y, z), +(y, +(\mathsf{0}, z)) \rangle & \in \leftarrow_{\mathcal{S}} & \langle +(x, y), +(+(x, y), \mathsf{0}) \rangle \in \leftarrow_{\mathcal{S}} \\ \langle \mathsf{s}(+(x, y)), +(\mathsf{s}(y), x) \rangle & \in \leftrightarrow_{\mathcal{P}} \circ \leftarrow_{\mathcal{S}} & \\ \langle \mathsf{s}(+(+(x, y), z)), +(x, +(y, \mathsf{s}(z))) \rangle & \in \leftrightarrow_{\mathcal{P}} \circ \overset{*}{\leftarrow}_{\mathcal{S}} & \\ \langle +(\mathsf{s}(+(x, y)), z), +(x, +(\mathsf{s}(y), z)) \rangle & \in \to_{\mathcal{S}} \circ \leftrightarrow_{\mathcal{P}} \circ \overset{*}{\leftarrow}_{\mathcal{S}} & \\ \langle +(x, \mathsf{s}(+(y, z))), +(+(x, y), \mathsf{s}(z)) \rangle & \in \to_{\mathcal{S}} \circ \leftrightarrow_{\mathcal{P}} \circ \leftarrow_{\mathcal{S}} & \end{array} \right\}.$$

Thus one can apply Theorem 3.9 (or Theorem 3.7) to obtain the confluence of $\mathcal{R} = \mathcal{S} \cup \mathcal{P}$.

For the case the terminating TRS $\mathcal{S}$ is linear, one can obtain another confluence criterion from the abstract confluence criterion using $\vdash\!\!\dashv_0 := \leftrightarrow_{\mathcal{P}}$ instead of $\vdash\!\!\dashv_0 := \Mapsto_{\mathcal{P}}$.

▶ **Lemma 3.11.** *Let $\mathcal{P}, \mathcal{S}$ be TRSs. Suppose* $\mathrm{CP}(\mathcal{S}, \mathcal{S}) \subseteq \overset{*}{\to}_{\mathcal{S}} \circ \overset{=}{\leftrightarrow}_{\mathcal{P}} \circ \overset{*}{\leftarrow}_{\mathcal{S}}$. *Then* $\leftarrow_{\mathcal{S}} \circ \to_{\mathcal{S}} \subseteq \overset{*}{\to}_{\mathcal{S}} \circ \overset{=}{\leftrightarrow}_{\mathcal{P}} \circ \overset{*}{\leftarrow}_{\mathcal{S}}$.

Proof. Take $\vdash\!\!\dashv := \leftrightarrow_{\mathcal{P}}$ in Lemma 3.3. ◀

▶ **Lemma 3.12.** *Let $\mathcal{P}, \mathcal{S}$ be TRSs such that $\mathcal{S}$ is linear and $\mathcal{P}$ is bidirectional. Suppose* $\mathrm{CP}(\mathcal{S}, \mathcal{P} \cup \mathcal{P}^{-1}) \cup \mathrm{CP}(\mathcal{P} \cup \mathcal{P}^{-1}, \mathcal{S}) \subseteq \overset{*}{\to}_{\mathcal{S}} \circ \overset{=}{\leftrightarrow}_{\mathcal{P}} \circ \overset{*}{\leftarrow}_{\mathcal{S}}$. *Then* $\overset{=}{\leftrightarrow}_{\mathcal{P}} \circ \to_{\mathcal{S}} \subseteq \overset{*}{\to}_{\mathcal{S}} \circ \overset{=}{\leftrightarrow}_{\mathcal{P}} \circ \overset{*}{\leftarrow}_{\mathcal{S}}$.

Proof. In a similar way to the proof of Lemma 3.5. ◀

▶ **Theorem 3.13** (confluence criterion for linear $\mathcal{S}$)**.** *Let $\mathcal{P}, \mathcal{S}$ be TRSs such that $\mathcal{S}$ is linear and terminating and $\mathcal{P}$ is reversible. Suppose (i)* $\mathrm{CP}(\mathcal{S}, \mathcal{S}) \subseteq \overset{*}{\to}_{\mathcal{S}} \circ \overset{=}{\leftrightarrow}_{\mathcal{P}} \circ \overset{*}{\leftarrow}_{\mathcal{S}}$, *(ii)* $\mathrm{CP}(\mathcal{P} \cup \mathcal{P}^{-1}, \mathcal{S}) \subseteq \overset{*}{\to}_{\mathcal{S}} \circ \overset{=}{\leftrightarrow}_{\mathcal{P}} \circ \overset{*}{\leftarrow}_{\mathcal{S}}$ *and (iii)* $\mathrm{CP}(\mathcal{S}, \mathcal{P} \cup \mathcal{P}^{-1}) \subseteq \overset{*}{\to}_{\mathcal{S}} \circ \overset{=}{\leftrightarrow}_{\mathcal{P}} \circ \overset{*}{\leftarrow}_{\mathcal{S}}$. *Then $\mathcal{S} \cup \mathcal{P}$ is confluent.*

Proof. Similar to proof of the Theorem 3.7, using Lemmas 3.11, 3.12. ◀

The next examples show that Theorem 3.13 and Theorems 3.7/3.9 are incomparable (Figure 1).

▶ **Example 3.14.** Let $\mathcal{R}$ be the one given in Example 3.10. Consider a TRS $\mathcal{R}_1 = \mathcal{R} \cup \{\mathsf{dbl}(x) \to +(x, x)\}$. One can easily confirm that the confluence of $\mathcal{R}_1$ is shown in the same way as $\mathcal{R}$ using Theorem 3.7. Since $\mathcal{R}_1$ is not linear, however, Theorem 3.13 does not apply. Consider a TRS $\mathcal{R}_2 = \mathcal{R} \cup \{\mathsf{s}(x) \to \mathsf{s}(\mathsf{s}(x)), \mathsf{s}(\mathsf{s}(x)) \to \mathsf{s}(x)\}$. By putting $\mathcal{S}_2 = \mathcal{S}$ and $\mathcal{P}_2 = \mathcal{P} \cup \{\mathsf{s}(x) \to \mathsf{s}(\mathsf{s}(x)), \mathsf{s}(\mathsf{s}(x)) \to \mathsf{s}(x)\}$, one can show the confluence of $\mathcal{R}_2$ using Theorem 3.13. On the other hand, $\langle +(\mathsf{s}(\mathsf{s}(x)), y), \mathsf{s}(+(x, y)) \rangle_{\{x\}} \in \mathrm{PCP}_{in}(\mathcal{P} \cup \mathcal{P}^{-1}, \mathcal{S})$ and thus the condition of Theorem 3.9 is not satisfied. Theorem 3.9 does not apply to other partitions of $\mathcal{R}_2$ either. Thus one can not show the confluence of $\mathcal{R}_2$ using Theorem 3.9.

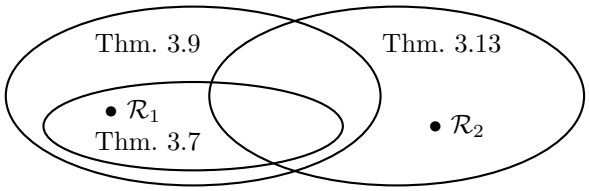


Figure 1 Relation of three confluence criterion

4 Reduction-preserving completion

There are cases where our confluence criteria can be applicable indirectly. The idea is to construct a TRS suitable for applying our theorems by exchanging or adding rewrite rules without changing the equivalence of the reduction so that the confluence of the transformed TRS implies that of the original TRS. Using the reversibility of $\mathcal{P}$, there are several flexibilities on such transformations. The notion of reduction equivalence and following properties of reduction equivalence are well-known in literature and the latter are easily proved.

▶ **Definition 4.1** (reduction equivalence)**.** Two relation $\to_0$ and $\to_1$ are said to be *reduction equivalent* if $\overset{*}{\to}_0 = \overset{*}{\to}_1$. Two TRSs $\mathcal{R}$ and $\mathcal{R}'$ are reduction equivalent if so are $\to_{\mathcal{R}}$ and $\to_{\mathcal{R}'}$.

▶ **Proposition 4.2** (properties of reduction equivalence)**.** *(i) If* $\to_{\mathcal{R}} \subseteq \overset{*}{\to}_{\mathcal{R}'}$ *and* $\to_{\mathcal{R}'} \subseteq \overset{*}{\to}_{\mathcal{R}}$ *then* $\mathcal{R}$ *and* $\mathcal{R}'$ *are reduction equivalent. (ii) If* $\mathcal{R}$ *and* $\mathcal{R}'$ *are reduction equivalent then the confluence of* $\mathcal{R}$ *and* $\mathcal{R}'$ *coincide.*

We now demonstrate how the confluence criteria in the previous section can be applied indirectly using the notion of reduction equivalence.

▶ **Example 4.3** (confluence by reduction equivalence)**.** Let (a)–(f) be rewrite rules given in Example 3.10. We show the confluence of $\mathcal{R} = \{(a), (b), (e), (f)\}$. Theorems 3.9 and 3.13 can not be applied directly to prove this—for example, if we put $\mathcal{S} = \{(a), (b)\}$ and $\mathcal{P} = \{(e), (f)\}$, then we have $\langle y, +(y, \mathsf{0})\rangle \in \mathrm{CP}(\mathcal{S}, \mathcal{P} \cup \mathcal{P}^{-1})$ which is not joinable by $\overset{*}{\to}_{\mathcal{S}} \circ \Mapsto_{\mathcal{P}} \circ \overset{*}{\leftarrow}_{\mathcal{S}}$. Let $\mathcal{R}' = \mathcal{R} \cup \{(c), (d)\}$. Then since we have $+(x, \mathsf{0}) \to_{\mathcal{P}} +(\mathsf{0}, x) \to_{\mathcal{S}} x$ and $+(x, \mathsf{s}(y)) \to_{\mathcal{P}} +(\mathsf{s}(y), x) \to_{\mathcal{S}} \mathsf{s}(+(y, x) \to_{\mathcal{P}} \mathsf{s}(+(x, y))$, the inclusions $\to_{\mathcal{R}} \subseteq \to_{\mathcal{R}'} \subseteq \overset{*}{\to}_{\mathcal{R}}$ hold. Hence $\mathcal{R}$ and $\mathcal{R}'$ are reduction equivalent by Proposition 4.2 (i). As we have shown in Example 3.10, $\mathcal{R}'$ is confluent. Thus by Proposition 4.2 (ii), $\mathcal{R}$ is confluent either.

In this example, two additional rewrite rules (c) and (d) are given by hand. But in automated confluence proving procedures, one needs to find such new rewrite rules automatically. We next present a completion-like procedure to automate such additions (or more generally transformations) of rewrite rules. We first present an abstract version of the procedure in the form of inference rules and prove its soundness w.r.t. the confluence proof.

▶ **Definition 4.4** (abstract reduction-preserving completion procedure)**.** Inference rules of an *abstract reduction-preserving completion procedure* are listed in Figure 2. The inference rules act on a pair of TRSs. One step derivation using any of inference rules (from upper to lower) is denoted by $\rightsquigarrow$. We also write $\rightsquigarrow^p$ ($\rightsquigarrow^r, \rightsquigarrow^a$) for an inference step by the rule *Partition* (*Replacement*, *Addition*, respectively).

$$\textit{Partition}\quad \frac{\langle \mathcal{S}, \mathcal{P}\rangle}{\langle \mathcal{S}', \mathcal{P}'\rangle}\ \ \mathcal{S}\cup\mathcal{P} = \mathcal{S}'\cup\mathcal{P}',\ \mathcal{P}'\text{: reversible}$$

$$\textit{Replacement}\quad \frac{\langle \mathcal{S}\cup\{l\to r\}, \mathcal{P}\rangle}{\langle \mathcal{S}\cup\{l\to r'\}, \mathcal{P}\rangle}\ \ r \overset{*}{\leftrightarrow}_{\mathcal{P}} r'$$

$$\textit{Addition}\quad \frac{\langle \mathcal{S}, \mathcal{P}\rangle}{\langle \mathcal{S}\cup\{l\to r\}, \mathcal{P}\rangle}\ \ l \overset{*}{\leftrightarrow}_{\mathcal{P}} \circ \overset{*}{\to}_{\mathcal{S}} r$$

Figure 2 Inference rules of reduction-preserving completion

▶ **Theorem 4.5** (soundness of the abstract reduction-preserving completion procedure)**.** *Let* $\langle \mathcal{R}, \emptyset\rangle = \langle \mathcal{S}_0, \mathcal{P}_0\rangle \overset{*}{\rightsquigarrow} \langle \mathcal{S}_n, \mathcal{P}_n\rangle$ *be a derivation of abstract reduction-preserving completion procedure. Suppose that* $\mathcal{S}_n, \mathcal{P}_n$ *satisfy the conditions of Theorem 3.9 or Theorem 3.13. Then* $\mathcal{R}$ *is confluent.*

Proof. We show, for any inference step $\langle \mathcal{S}_i, \mathcal{P}_i\rangle \rightsquigarrow \langle \mathcal{S}_{i+1}, \mathcal{P}_{i+1}\rangle$, that $\mathcal{S}_i \cup \mathcal{P}_i$ and $\mathcal{S}_{i+1} \cup \mathcal{P}_{i+1}$ are reduction equivalent and that $\mathcal{P}_{i+1}$ is reversible whenever so is $\mathcal{P}_i$.

- Case $\langle \mathcal{S}_i, \mathcal{P}_i\rangle \rightsquigarrow \langle \mathcal{S}_{i+1}, \mathcal{P}_{i+1}\rangle$ by *Partition*. Then since $\mathcal{S}_i \cup \mathcal{P}_i = \mathcal{S}_{i+1} \cup \mathcal{P}_{i+1}$ and $\mathcal{P}_{i+1}$ is reversible by the side condition, the claim follows immediately.
- Case $\langle \mathcal{S}_i, \mathcal{P}_i\rangle \rightsquigarrow \langle \mathcal{S}_{i+1}, \mathcal{P}_{i+1}\rangle$ by *Replacement*. Then $\mathcal{S}_i = \mathcal{S}'_i \cup \{l \to r\}$, $r \overset{*}{\leftrightarrow}_{\mathcal{P}_i} r'$ and $\mathcal{S}_{i+1} = \mathcal{S}'_i \cup \{l \to r'\}$ for some $\mathcal{S}'_i$, l, r, r' and $\mathcal{P}_{i+1} = \mathcal{P}_i$. By the reversiblity of $\mathcal{P}_i$, we have $l \to_{\mathcal{S}_i} r \overset{*}{\to}_{\mathcal{P}_i} r'$ hence $\to_{\mathcal{S}_{i+1}\cup\mathcal{P}_{i+1}} \subseteq \overset{*}{\to}_{\mathcal{S}_i\cup\mathcal{P}_i}$. By the reversiblity of $\mathcal{P}_i$, we also have $l \to_{\mathcal{S}_{i+1}} r' \overset{*}{\to}_{\mathcal{P}_i} r$, hence $\to_{\mathcal{S}_i\cup\mathcal{P}_i} \subseteq \overset{*}{\to}_{\mathcal{S}_{i+1}\cup\mathcal{P}_{i+1}}$. Thus by Proposition 4.2 (i), $\mathcal{S}_i \cup \mathcal{P}_i$ and $\mathcal{S}_{i+1} \cup \mathcal{P}_{i+1}$ are reduction equivalent. Hence, by $\mathcal{P}_{i+1} = \mathcal{P}_i$, the claim follows.
- Case $\langle \mathcal{S}_i, \mathcal{P}_i\rangle \rightsquigarrow \langle \mathcal{S}_{i+1}, \mathcal{P}_{i+1}\rangle$ by *Addition*. Then $l \overset{*}{\leftrightarrow}_{\mathcal{P}_i} \circ \overset{*}{\to}_{\mathcal{S}_i} r$ and $\mathcal{S}_{i+1} = \mathcal{S}_i \cup \{l \to r\}$ for some l, r and and $\mathcal{P}_{i+1} = \mathcal{P}_i$. Since $\mathcal{S}_i \cup \mathcal{P}_i \subseteq \mathcal{S}_{i+1} \cup \mathcal{P}_{i+1}$, we have $\to_{\mathcal{S}_i\cup\mathcal{P}_i} \subseteq \overset{*}{\to}_{\mathcal{S}_{i+1}\cup\mathcal{P}_{i+1}}$. By the reversiblity of $\mathcal{P}_i$, $l \overset{*}{\to}_{\mathcal{P}_i} \circ \overset{*}{\to}_{\mathcal{S}_i} r'$. Hence $\to_{\mathcal{S}_{i+1}\cup\mathcal{P}_{i+1}} \subseteq \overset{*}{\to}_{\mathcal{S}_i\cup\mathcal{P}_i}$. Thus by Proposition 4.2 (i), $\mathcal{S}_i \cup \mathcal{P}_i$ and $\mathcal{S}_{i+1} \cup \mathcal{P}_{i+1}$ are reduction equivalent. Hence, by $\mathcal{P}_{i+1} = \mathcal{P}_i$, the claim follows.

Thus by induction on n, it follows that $\mathcal{R}$ and $\mathcal{S}_n \cup \mathcal{P}_n$ are reduction equivalent. By Theorem 3.9 or 3.13, $\mathcal{S}_n \cup \mathcal{P}_n$ is confluent, and hence $\mathcal{R}$ is confluent by Proposition 4.2 (ii). ◀

▶ **Example 4.6** (derivations in abstract reduction-preserving completion procedure)**.** The confluence proof of Example 4.3 is derived by the abstract reduction-preserving completion procedure. Let rewrite rules (a)–(f) be those given in Example 3.10. Give $\mathcal{R} = \{(a), (b), (e), (f)\}$ as the input to the procedure. Let us consider the following derivation.

$$\begin{array}{lll}
\langle \mathcal{S}_0, \mathcal{P}_0\rangle = \langle \{(a),(b),(e),(f)\}, \emptyset\rangle & \rightsquigarrow^p & \langle \{(a),(b)\}, \{(e),(f)\}\rangle = \langle \mathcal{S}_1, \mathcal{P}_1\rangle \\
& \rightsquigarrow^a & \langle \{(a),(b),(c)\}, \{(e),(f)\}\rangle = \langle \mathcal{S}_2, \mathcal{P}_2\rangle \\
& \rightsquigarrow^a & \langle \{(a),(b),(c),(d')\}, \{(e),(f)\}\rangle = \langle \mathcal{S}_3, \mathcal{P}_3\rangle \\
& \rightsquigarrow^r & \langle \{(a),(b),(c),(d)\}, \{(e),(f)\}\rangle = \langle \mathcal{S}_4, \mathcal{P}_4\rangle
\end{array}$$

where $(d') : +(x, \mathsf{s}(y)) \to \mathsf{s}(+(y, x))$. Then $\mathcal{S}_4 = \{(a), (b), (c), (d)\}$ and $\mathcal{P}_4 = \{(e), (f)\}$ satisfy the conditions of Theorem 3.7. Thus, by Theorem 4.5, $\mathcal{R}$ is confluent.

Next we present a concrete reduction-preserving completion procedure that can be used as the basis of an automated completion procedure. The procedure presented below is designed so as to apply Theorem 3.7, but it is straightforward to modify the procedure suitable for Theorem 3.9 and/or Theorem 3.13.

▶ **Definition 4.7** (concrete reduction-preserving completion procedure)**.**

Input: a TRS $\mathcal{R}$
Output: `Success` or `Failure` (or may diverge)

Step 1. Put $\mathcal{R}_0 := \mathcal{R}$ and $i := 0$. Proceed to Step 2.
Step 2. Take a partition $\mathcal{S}_i \cup \mathcal{P}_i = \mathcal{R}_i$ such that $\mathcal{S}_i$ is left-linear and terminating, $\mathcal{P}_i$ is reversible and $\mathrm{CP}_{in}(\mathcal{P}_i \cup \mathcal{P}_i^{-1}, \mathcal{S}_i) = \emptyset$. Proceed to Step 3. If there is no such a partition then return `Failure`.
Step 3. Set $\mathcal{U} := \emptyset$. For each $\langle u, v\rangle \in \mathrm{CP}(\mathcal{S}_i, \mathcal{P}_i \cup \mathcal{P}_i^{-1})$, take $\mathcal{S}_i$-normal forms $\hat{u}, \hat{v}$ of u, v, respectively and check whether $\hat{u} \twoheadleftarrow_{\mathcal{P}_i} \hat{v}$. If not $\hat{u} \twoheadleftarrow_{\mathcal{P}_i} \hat{v}$ then put $\mathcal{U} := \mathcal{U} \cup \{v \to \hat{u}\}$. Finally if $\mathcal{U} = \emptyset$ then proceed to Step 4. Otherwise take some non-empty $\mathcal{U}' \subseteq \mathcal{U}$ and put $\mathcal{R}_{i+1} := \mathcal{R}_i \cup \mathcal{U}'$, $i := i + 1$ and go to Step 2.
Step 4. Set $\mathcal{U} := \emptyset$. For each $\langle u, v\rangle \in \mathrm{CP}(\mathcal{S}_i, \mathcal{S}_i)$, take $\mathcal{S}_i$-normal forms $\hat{u}, \hat{v}$ of u, v, respectively and check whether $\hat{u} \twoheadleftarrow_{\mathcal{P}_i} \hat{v}$. If not $\hat{u} \twoheadleftarrow_{\mathcal{P}_i} \hat{v}$ then put $\mathcal{U} := \mathcal{U} \cup \{\hat{u} \approx \hat{v}\}$. Finally if $\mathcal{U} = \emptyset$ then return `Success`. Otherwise take some set $\mathcal{U}' \subseteq (\mathcal{U} \cup \mathcal{U}^{-1}) \cap \overset{*}{\leftrightarrow}_{\mathcal{P}_i}$ of rewrite rules and put $\mathcal{R}_{i+1} := \mathcal{R}_i \cup \mathcal{U}'$, $i := i + 1$ and go to Step 2.

During the step 2, one may perform the following additional steps.

Step 2a. If there exist $l \to r \in \mathcal{S}_i$ and r' such that $r \leftrightarrow_{\mathcal{P}_i} r'$ and $\mathrm{CP}_{in}(\mathcal{P}_i \cup \mathcal{P}_i^{-1}, \{l \to r\}) \neq \emptyset$, then put $\mathcal{R}_{i+1} := (\mathcal{R}_i \setminus \{l \to r\}) \cup \{l \to r'\}$, $i := i + 1$.
Step 2b. Let $\langle u, v\rangle \in \mathrm{CP}_{in}(\mathcal{P}_i \cup \mathcal{P}_i^{-1}, \mathcal{S}_i)$ and let $\hat{v}$ be $\mathcal{S}_i$-normal form of v. Then put $\mathcal{R}_{i+1} := \mathcal{R}_i \cup \{u \to \hat{v}\}$ and $i := i + 1$.

Before moving from the step 3 to the step 2, one may perform the following additional step.

Step 3a. Set $\mathcal{S}_i := \mathcal{S}_{i-1}, \mathcal{P}_i := \mathcal{P}_{i-1}$. If there exist $l \to r \in \mathcal{S}_i$ and r' such that $r \leftrightarrow_{\mathcal{P}_i} r'$ and there exists $\langle u, v\rangle \in \mathrm{CP}(\{l \to r\}, \mathcal{P}_i \cup \mathcal{P}_i^{-1})$ such that $\hat{u} \twoheadleftarrow_{\mathcal{P}_i} \hat{v}$ does not hold where $\hat{u}, \hat{v}$ are $\mathcal{S}_i$-normal forms of u, v, respectively, then put $\mathcal{R}_{i+1} := (\mathcal{R}_i \setminus \{l \to r\}) \cup \{l \to r'\}$, $i := i + 1$.

Before moving from the step 4 to the step 2, one may perform the following additional step.

Step 4a. Set $\mathcal{S}_i := \mathcal{S}_{i-1}, \mathcal{P}_i := \mathcal{P}_{i-1}$. If there exist $l \to r \in \mathcal{S}_i$ and r' such that $r \leftrightarrow_{\mathcal{P}_i} r'$ and there exists $\langle u, v\rangle \in \mathrm{CP}(\{l \to r\}, \mathcal{S}_i) \cup \mathrm{CP}(\mathcal{S}_i, \{l \to r\})$ such that $\hat{u} \twoheadleftarrow_{\mathcal{P}_i} \hat{v}$ does not hold where $\hat{u}, \hat{v}$ are $\mathcal{S}_i$-normal forms of u, v, respectively, then put $\mathcal{R}_{i+1} := (\mathcal{R}_i \setminus \{l \to r\}) \cup \{l \to r'\}$, $i := i + 1$.

▶ **Corollary 4.8** (soundness of the concrete reduction-preserving completion procedure)**.** *If the procedure of Definition 4.7 succeeds for the input $\mathcal{R}$, then $\mathcal{R}$ is confluent.*

Proof. It suffices to show if the procedure succeeds then there exists a successful derivation of the abstract reduction-preserving completion procedure. Step 1 corresponds to the empty derivation. Step 2 corresponds to an inference step by *Partition*. For any $\langle u, v\rangle \in \mathrm{CP}(\mathcal{S}_i, \mathcal{P}_i \cup \mathcal{P}_i^{-1})$, we have $u \leftarrow_{\mathcal{S}_i} \circ \leftrightarrow_{\mathcal{P}_i} v$, and hence $v \leftrightarrow_{\mathcal{P}_i} \circ \overset{*}{\to}_{\mathcal{S}_i} \hat{u}$. Thus, Step 3 is simulated by multiple inference steps by *Addition*. Similarly, Steps 4 and 2b are simulated by multiple inference steps by *Addition*. Steps 2a, 3a, 4a are simulated by inference steps by *Replace*. ◀

▶ **Example 4.9.** Let

$$\mathcal{R} = \left\{ \begin{array}{llcl|llcl} (a) & +(\mathsf{0}, y) & \to & y & (b) & +(x, \mathsf{s}(y)) & \to & \mathsf{s}(+(x, y)) \\ (c) & +(x, y) & \to & +(y, x) & (d) & +(+(x, y), z) & \to & +(x, +(y, z)) \end{array} \right\}$$

1. (Step 1) We put $\mathcal{R}_0 := \{(a), (b), (c), (d)\}$.
2. (Step 2) We take $\mathcal{S}_0 = \{(a), (b)\}$ and $\mathcal{P}_0 = \{(c), (d)\}$. Then $\mathcal{S}_0$ is left-linear and terminating, $\mathcal{P}_0$ is reversible and $\mathrm{CP}_{in}(\mathcal{P}_0 \cup {\mathcal{P}_0}^{-1}, \mathcal{S}_0) = \emptyset$.
3. (Step 3) We have $\mathrm{CP}(\mathcal{S}_0, \mathcal{P}_0 \cup {\mathcal{P}_0}^{-1}) =$

$$\left\{ \begin{array}{llll} (1) & \langle +(y,z), +(0, +(y,z)) \rangle & (5) & \langle \mathsf{s}(+(+(x,z),y)), +(x, +(z, \mathsf{s}(y))) \rangle \\ (2) & \langle +(y,z), +(+(0,y),z) \rangle & (6) & \langle +(\mathsf{s}(+(x,y)),z), +(x, +(\mathsf{s}(y),z)) \rangle \\ (3) & \langle +(x,y), +(+(x,0),y) \rangle & (7) & \langle +(z, \mathsf{s}(+(x,y))), +(+(z,x), \mathsf{s}(y)) \rangle \\ (4) & \langle y, +(y,0) \rangle & (8) & \langle \mathsf{s}(+(x,y)), +(\mathsf{s}(y), x) \rangle \end{array} \right\}.$$

   Then for $\langle u, v \rangle \in \{(3), (4), (6), (8)\}$, $\mathcal{S}_0$-normal forms of u, v are not joinable by a $\Lsh\!\!\Rsh_{\mathcal{P}_0}$-step. Put $\mathcal{R}_1 := \mathcal{R}_0 \cup \mathcal{U}' =$

$$\mathcal{R}_0 \cup \left\{ \begin{array}{llll} (e) & +(y,\mathsf{0}) \to y & (f) & +(\mathsf{s}(y), x) \to \mathsf{s}(+(x,y)) \end{array} \right\}.$$

   and go to the step 2.
4. (Step 2) We take $\mathcal{S}_1 = \{(a), (b), (e), (f)\}$ and $\mathcal{P}_1 = \{(c), (d)\}$. Then $\mathcal{S}_1$ is left-linear and terminating, $\mathcal{P}_1$ is reversible, and $\mathrm{CP}_{in}(\mathcal{P}_1 \cup {\mathcal{P}_1}^{-1}, \mathcal{S}_1) = \emptyset$.
5. (Step 3) There are four elements including (9) $\langle +(\mathsf{s}(+(x,y)),z), +(x, +(\mathsf{s}(y),z)) \rangle$ in $\mathrm{CP}(\mathcal{S}_1, \mathcal{P}_1 \cup {\mathcal{P}_1}^{-1})$ whose $\mathcal{S}_1$-normal forms are not joinable by a $\Lsh\!\!\Rsh_{\mathcal{P}_1}$-step. Here we put $\mathcal{U}' := \emptyset$, $\mathcal{R}_2 := \mathcal{R}_1$, $i := 2$ and proceed to Step 3a.
6. (Step 3a) Since (9) $\in \mathrm{CP}(\{(f)\}, \mathcal{P}_1 \cup {\mathcal{P}_1}^{-1})$ and $\mathsf{s}(+(x,y)) \to_{\mathcal{P}_2} \mathsf{s}(+(y,x))$. Hence put $\mathcal{R}_3 := (\mathcal{R}_2 \setminus \{(f)\}) \cup \left\{ \begin{array}{llll} (g) & +(\mathsf{s}(y), x) \to \mathsf{s}(+(y,x)) \end{array} \right\}$ and $i := 3$ and go to Step 2.
7. (Step 2) We take $\mathcal{S}_3 = \{(a), (b), (e), (g)\}$ and $\mathcal{P}_3 = \{(c), (d)\}$. Then $\mathcal{S}_3$ is left-linear and terminating, $\mathcal{P}_3$ is reversible and $\mathrm{CP}_{in}(\mathcal{P}_3 \cup {\mathcal{P}_3}^{-1}, \mathcal{S}_3) = \emptyset$. Thus proceed to Step 3.
8. (Step 3) For any $\langle u, v \rangle \in \mathrm{CP}(\mathcal{S}_3, \mathcal{P}_3 \cup {\mathcal{P}_3}^{-1})$, $\mathcal{S}_3$-normal forms of u, v are joinable by a $\Lsh\!\!\Rsh_{\mathcal{P}_3}$-step (Example 3.10). Hence proceed to Step 4.
9. (Step 4) For any $\langle u, v \rangle \in \mathrm{CP}(\mathcal{S}_3, \mathcal{S}_3)$, $\mathcal{S}_3$-normal forms of u, v are joinable by a $\Lsh\!\!\Rsh_{\mathcal{P}_3}$-step (Example 3.10). Thus `Success` is returned.

5 Implementation and experiments

All results of this paper have been implemented. The program is written in SML/NJ[1] and is built upon confluence prover ACP[2] [1, 2, 21].

In Figure 3, we present a pseudo-code of main function of our implementation of reduction-preserving completion procedure enough for describing some heuristics employed in the implementation. A short description of functions involved in our pseudo-code and heuristics employed follows.

- (`checkConfluence` $\mathcal{R}$) is the main function of the procedure. It simulates multiple runs in the breadth-first strategy.

  Let $\mathcal{D} = \{l(\epsilon) \mid l \to r \in \mathcal{R}\}$ and $\mathcal{C} = \mathcal{F} \setminus \mathcal{C}$.

[1] `http://www.smlnj.org/`
[2] `http://www.nue.riec.tohoku.ac.jp/tools/acp/`

```
fun check (S,P,i) = if i = 0 then (apply Theorem 3.9)
                    else (apply Theorem 3.13)
fun checkConfluence R =
    let fun step [] = Failure
        |   step ((S,P,i)::rest) = case check (S,P,i) of
                NONE ⇒ step rest
              | SOME ([],[]) ⇒ Succeess
              | SOME nj ⇒ step (rest @
                                (mapAppend decompose (trans (S,P) nj)))
    in step (decompose R) end
```

Figure 3 Pseudo-code of the main function

- (`decompose` $\mathcal{R}$) decomposes $\mathcal{R}$ into $\mathcal{S} \cup \mathcal{P}$ and duplicates $\mathcal{S} \cup \mathcal{P}$. Hence a list of triples $(\mathcal{S}, \mathcal{P}, i)$ where $\mathcal{S} \cup \mathcal{P} = \mathcal{R}$ and $i \in \{0, 1\}$ are returned. Here, however, not all partitions but only one partition of $\mathcal{R}$ are returned based on a heuristic, namely that $\mathcal{P}$ is the set of the rules $l \to r$ satisfying either (1) $r \to l \in \mathcal{R}$ or (1′) $\mathcal{F}(l) = \mathcal{F}(r) \subseteq \mathcal{D}$ and (2′) $l(\epsilon), r(\epsilon) \in \mathcal{D}$ implies $l(\epsilon) = r(\epsilon)$.
- (`check` $(\mathcal{S}, \mathcal{P}, i)$) checks whether conditions of Theorem 3.9 (or Theorem 3.13) are satisfied. If $\mathcal{S}$ is not left-linear or it fails to prove termination of $\mathcal{S}$ or reversibility of $\mathcal{P}$, then `NONE` is returned. Reversibility is tested by checking $r \overset{\leq k}{\to} l$ for some constant k (in our implementation, we set $k = 10$). If all conditions other than the critical pairs conditions are satisfied then non-joinable critical pairs and rewrite rules generating such critical pairs are returned in the form `SOME` (U_1, U_2). For example, in the case of $i = 0$, from $\mathrm{CP}(\mathcal{S}, \mathcal{S})$ the list $U_1 = \bigcup_{l \to r, l' \to r' \in \mathcal{S}} \{\langle l \to r, l' \to r', u, v\rangle \mid \langle u, v\rangle \in \mathrm{CP}(\{l \to r\}, \{l' \to r'\}) \setminus \overset{*}{\to}_{\mathcal{S}} \circ \leftrightarrow\!\!\!\!\!\!\to_{\mathcal{P}} \circ \overset{*}{\leftarrow}_{\mathcal{S}}\}$ is returned. Similarly U_2 is obtained from $\mathrm{PCP}_{in}(\mathcal{P} \cup \mathcal{P}^{-1}, \mathcal{S}) \cup \mathrm{CP}(\mathcal{S}, \mathcal{P} \cup \mathcal{P}^{-1})$. If both of these lists are empty then the conditions of Theorem 3.9 (or Theorem 3.13) are satisfied and thus the procedure succeeds (`Success` is returned).
- (`trans` $(\mathcal{S}, \mathcal{P})$ (U_1, U_2)) returns a collection of transformed TRSs obtained by addition and replacement of rewrite rules constructed from non-joinable critical pairs and rewrite rules generating such critical pairs as described in the Definition 4.7. Here, the addition of rewrite rules are restricted based on the following heuristic: $l \to r$ is added if (1) $l \in \mathrm{NF}(\mathcal{S})$, (2) $l(\epsilon) = r(\epsilon) \in \mathcal{D}$ implies $(\mathcal{F}(l) \cup \mathcal{F}(r)) \cap \mathcal{C} = \emptyset$ and (3) $l(\epsilon) \neq r(\epsilon)$ and $l(\epsilon), r(\epsilon) \in \mathcal{D}$ imply $\mathcal{F}(r) \cap \mathcal{C} = \emptyset$.

Table 1 shows the summary of our experiments. We have tested various combinations of our results: (1)–(4) are proofs by confluence criterion of Theorem 3.7, of Theorem 3.9, of Theorem 3.13 and by the combination of those of Theorem 3.9 and Theorem 3.13. (5)–(7) are proofs by the reduction-preserving completion without the *Replacement* rule, i.e. without the Steps $2a, 3a, 4a$ of the concrete reduction-preserving completion (Definition 4.7). (8)–(10) are proofs by the reduction-preserving completion with the *Replacement* rule. For the experiments, we used a collection of 81 TRSs involving non-terminating rules such as commutativity and associativity rules which have been developed in the course of experiments. All experiments have been performed on a FreeBSD platform of a PC equipped with 1.2GHz CPU and 1GB memory. We set the timout 60 sec. Total time is indicated in millisecond.

Table 1 Summary of experiments

	success	failure	diverge	timeout	time(msec.)
(1) main (Theorem 3.7)	19	62	0	0	1308
(2) PCP (Theorem 3.9)	28	53	0	0	1318
(3) linear (Theorem 3.13)	27	54	0	0	901
(4) PCP&linear	29	52	0	0	1725
(5) completion (PCP)	50	31	0	0	2258
(6) completion (linear)	46	35	0	0	1451
(7) completion (PCP&linear)	51	30	0	0	2995
(8) completion (repl., PCP)	64	17	(3)	0	3773
(9) completion (repl., linear)	59	22	0	0	2146
(10) completion (repl., PCP&linear)	66	15	(2)	0	4885
ACP [1, 2, 21]	12	67	—	2	164943

The maximal steps of the completion procedure is limited to 20 steps; the columns below the title "diverge" show the numbers of examples which exceed this limit, where these numbers are included in those of "failure."

The applicability of our incomparable confluent criteria (Theorem 3.9 and Theorem 3.13) does not have much differences. The applicability of Theorem 3.7, which is subsumed by Theorem 3.9, is limited compared to these two criteria. There is a clear advantage of using the completion procedure. The introducton of the *Replacement* inference rule also makes clear difference. The increase of total time by the introduction of completion procedure based on a confluence criterion are within 3 times of total time required in proving confluence only by checking that confluence criterion. This is partly due to our heuristics and the limitation on the number of limit of completion steps. The number of successful examples, however, does not change in the case we increase that limit to 100 steps. The collection of examples and all details of the experiments are available on the webpage `http://www.nue.riec.tohoku.ac.jp/tools/acp/experiments/rta11/all.html`.

We have also tested the confluence prover ACP on our collection. ACP is an automated confluence prover in which divide–and–conquer approach based on the persistent, layer-preserving, commutative decompositions is employed and involving many confluence criteria [4, 6, 10, 11, 14, 16, 17, 12, 19] as well as the decreasing diagram techniques [18, 20]. As shown in the table, most of our examples are not coped with the confluence prover ACP.

We have also tested on the 71 examples containing associativity and commutativity rules selected from the termination problem database 8.0[3] which have been developed to test termination modulo AC or C. ACP succeeded at 30 examples among which 27 examples are proved as non-confluent and 3 examples are proved as confluent. By our methods, 7 examples have been proved as confluent. We have also tested on a collection of 106 examples from [2, 1]. By enhancing ACP by our methods, confluence proving succeeded at 3 more examples.

6 Conclusion

We have presented a method for proving confluence of TRSs which can be applied even if the TRSs contain non-terminating rules such as commutativity and associativity. We have given

[3] `http://www.termination-portal.org`

confluence criteria for TRSs that can be partitioned into terminating part and reversible part which may be non-terminating. Then we have given a reduction-preserving completion procedure so that the criteria can be applied indirectly. In contrast to the well-known method for proving confluence of equational TRSs [7], our method is based solely on usual critical pairs and usual termination and hence easily integrated into confluence provers based on other confluence proving methods for TRSs. We have implemented the proposed techniques and reported experimental results. It turns out that our method is effective for TRSs for which most of standards methods for proving confluence of TRSs are not effective.

The following examples show that our method and the methods of [6, 7] are incomparable.

▶ **Example 6.1.** Let

$$\mathcal{R} = \left\{ \begin{array}{ll} +(x,\mathsf{0}) & \to x \\ +(x,\mathsf{s}(y)) & \to \mathsf{s}(+(x,y)) \\ *(x,\mathsf{0}) & \to \mathsf{0} \\ *(x,\mathsf{s}(y)) & \to +(*(x,y),x) \\ *(x,+(y,z)) & \to +(*(x,y),*(x,z)) \end{array} \right\} \text{ and } \mathcal{E} = \left\{ \begin{array}{ll} +(x,y) & \leftrightarrow +(y,x) \\ +(+(x,y),z) & \leftrightarrow +(x,+(y,z)) \\ *(x,y) & \leftrightarrow *(y,x) \\ *(*(x,y),z) & \leftrightarrow *(x,*(y,z)) \end{array} \right\}.$$

It can be shown by the method of [7] that $\mathcal{R}$ is confluent modulo $\mathcal{E}$ and hence $\mathcal{R} \cup \mathcal{E}$ is confluent. Our method, however, failed to prove this example. Let

$$\mathcal{R}' = \left\{ *(+(x,y),z) \to +(*(x,z),*(y,z)) \right\} \text{ and } \mathcal{E}' = \left\{ \begin{array}{ll} +(x,y) & \leftrightarrow +(y,x) \\ +(+(x,y),z) & \leftrightarrow +(x,+(y,z)) \end{array} \right\}.$$

It can be shown by the method of [6] that $\mathcal{R}'$ is confluent modulo $\mathcal{E}'$ and hence $\mathcal{R}' \cup \mathcal{E}'$ is confluent. Our method, however, failed to prove this example. Let

$$\mathcal{R}'' = \left\{ \begin{array}{l} \mathsf{f}(\mathsf{0},\mathsf{0}) \to \mathsf{f}(\mathsf{0},\mathsf{1}) \\ \mathsf{f}(\mathsf{1},\mathsf{0}) \to \mathsf{f}(\mathsf{0},\mathsf{0}) \end{array} \right\} \text{ and } \mathcal{E}'' = \left\{ \mathsf{f}(x,y) \leftrightarrow \mathsf{f}(y,x) \right\}.$$

It can be shown by our method that $\mathcal{R}'' \cup \mathcal{E}''$ is confluent. Because $\mathcal{R}''$ is not terminating modulo $\mathcal{E}''$, the methods of [6, 7] fail to prove this example. We also note that the method of [8] also fails to prove this example by the same reason.

Acknowledgment

Thanks are due to Junichi Mitimata for discussions and experiments on preliminary results of this paper. The authors are grateful for Harald Zankl, Aart Middeldorp and anonymous referees for pointers to related works and helpful comments. This work was partially supported by grants from JSPS Nos. 20500002 and 22500002.

References

1 T. Aoto. Automated confluence proof by decreasing diagrams based on rule-labelling. In *Proc. of RTA 2010*, volume 6 of *LIPIcs*, pages 7–16. Schloss Dagstuhl, 2010.

2 T. Aoto, Y. Yoshida, and Y. Toyama. Proving confluence of term rewriting systems automatically. In *Proc. of RTA 2009*, volume 5595 of *LNCS*, pages 93–102. Springer-Verlag, 2009.

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

4 H. Gomi, M. Oyamaguchi, and Y. Ohta. On the Church-Rosser property of root-E-overlapping and strongly depth-preserving term rewriting systems. *Transactions of IPSJ*, 39(4):992–1005, 1998.

5 B. Gramlich. Confluence without termination via parallel critical pairs. In *Proc. of CAAP'96*, volume 1996 of *LNCS*, pages 211–225. Springer-Verlag, 2006.

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

7 J.-P. Jouannaud and H. Kirchner. Completion of a set of rules modulo a set of equations. *SIAM Journal of Computing*, 15(4):1155–1194, 1986.

8 J.-P. Jouannaud and M. Munoz. Termination of a set of rules modulo a set of equations. In *Proc. of CADE-7*, volume 170 of *LNCS*, pages 175–193. Springer-Verlag, 1984.

9 E. Ohlebusch. Church-Rosser theorems for abstract reduction modulo an equivalence relation. In *Proc. of RTA-98*, volume 1379 of *LNCS*, pages 17–31. Springer-Verlag, 1998.

10 S. Okui. Simultaneous critical pairs and Church-Rosser property. In *Proc. of RTA-98*, volume 1379 of *LNCS*, pages 2–16. Springer-Verlag, 1998.

11 M. Oyamaguchi and Y. Ohta. A new parallel closed condition for Church-Rosser of left-linear TRS's. In *Proc. of RTA-97*, volume 1232 of *LNCS*, pages 187–201. Springer-Verlag, 1997.

12 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.

13 G. E. Peterson and M. E. Stickel. Complete sets of reductions for some equational theories. *Journal of the ACM*, 28(2):233–264, 1981.

14 Y. Toyama. On the Church-Rosser property of term rewriting systems. Technical Report 17672, NTT ECL, 1981. In Japanese.

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

16 Y. Toyama and M. Oyamaguchi. Church-Rosser property and unique normal form property of non-duplicting term rewriting systems. In *Proc. of CTRS-94*, volume 968 of *LNCS*, pages 316–331. Springer-Verlag, 1994.

17 Y. Toyama and M. Oyamaguchi. Conditional linearization of non-duplicating term rewriting systems. *IEICE Trans. Information and Systems*, E84-D(5):439–447, 2001.

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

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

20 V. van Oostrom. Confluence by decreasing diagrams: converted. In *Proc. of RTA 2008*, volume 5117 of *LNCS*, pages 306–320. Springer-Verlag, 2008.

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

Natural Inductive Theorems for Higher-Order Rewriting

Takahito Aoto[1], Toshiyuki Yamada[2], and Yuki Chiba[3]

1 **RIEC, Tohoku University**
2-1-1 Katahira, Aoba-ku, Sendai, Miyagi, 980-8577, Japan
aoto@nue.riec.tohoku.ac.jp

2 **Graduate School of Engineering, Mie University**
1577 Kurimamachiya, Tsu, Mie, 514-8507, Japan
toshi@cs.info.mie-u.ac.jp

3 **School of Information Science, Japan Advanced Institute of Science and Technology**
1-1 Asahidai, Nomi, Ishikawa, 923-1292, Japan
chiba@jaist.ac.jp

Abstract

The notion of inductive theorems is well-established in first-order term rewriting. In higher-order term rewriting, in contrast, it is not straightforward to extend this notion because of extensionality (Meinke, 1992). When extending the term rewriting based program transformation of Chiba et al. (2005) to higher-order term rewriting, we need extensibility, a property stating that inductive theorems are preserved by adding new functions via macros. In this paper, we propose and study a new notion of inductive theorems for higher-order rewriting, *natural inductive theorems*. This allows to incorporate properties such as extensionality and extensibility, based on simply typed S-expression rewriting (Yamada, 2001).

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

Keywords and phrases Inductive Theorems, Higher-Order Equational Logic, Simply-Typed S-Expression Rewriting Systems, Term Rewriting Systems

Digital Object Identifier 10.4230/LIPIcs.RTA.2011.107

Category Regular Research Paper

1 Introduction

Properties of programs are often proved by induction on data structures such as natural numbers or lists. In the case of first-order term rewriting, such properties are captured by the notion of *inductive theorems* (e.g. [5]): an equation $s \approx t$ is said to be an inductive theorem of a term rewriting system (TRS for short) $\mathcal{R}$ if all ground instances are equational consequences, i.e. $s\theta \leftrightarrow^*_{\mathcal{R}} t\theta$ holds for any ground substitution θ. Inductive theorems form the initial semantics of first-order equational theories. In the higher-order case, one often expects *extensionality*, meaning that expressions denoting the same function are equivalent. The proof system and semantics of higher-order equational theories as well as the initial semantics of such theories based on *extensional inductive theorems* have been studied in [16, 17, 18]. In the simply typed S-expression rewriting framework [1, 2, 3, 21], the notion

22nd International Conference on Rewriting Techniques and Applications.
Editor: M. Schmidt-Schauß; pp. 107–121

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


of higher-order inductive theorems and *inductionless induction* [12, 14, 15, 19] for proving higher-order inductive theorems automatically have been studied in [4].

Several transformations for optimizing functional programs have been developed [6, 10, 11, 13, 20]. One such framework is *program transformation by templates*, proposed by Huet and Lang [13]. Chiba et al. [7, 8, 9] developed a framework of program transformation by templates based on first-order term rewriting. In this framework, the correctness of the transformation—the equivalence of input and output TRSs—is formalized based on inductive equality. One of the ingredients for ensuring the correctness of this program transformation is *extensibility* of inductive theorems, meaning that inductive theorems are preserved when a new function by a macro (i.e. non-recursive function in terms of existing functions) is added.

In the case of higher-order term rewriting, in contrast, extensibility of extensional inductive theorems is not guaranteed. Consider the following simply typed S-expression rewriting system (STSRS for short):

$$\mathcal{R} = \left\{ \begin{array}{lcl} + \ \mathsf{0}\ y & \to & y \\ + \ (\mathsf{s}\ x)\ y & \to & \mathsf{s}\ (+\ x\ y) \\ \mathsf{zero}\ \mathsf{s} & \to & \mathsf{0} \end{array} \right\}.$$

Then $+\ x\ y \approx +\ y\ x$ is an extensional inductive theorem of $\mathcal{R}$, that is, for any ground substitution θ, $(+\ x\ y)\theta \overset{\mathrm{ext}*}{\leftrightarrow}_{\mathcal{R}}^{*} (+\ y\ x)\theta$ holds. Here $\overset{\mathrm{ext}*}{\leftrightarrow}_{\mathcal{R}}^{*}$ is an equivalence relation induced by $\mathcal{R}$ where extensionality is taken into account. However, if we add a new constant f and a rewrite rule $\mathsf{f}\ x \to \mathsf{0}$ to $\mathcal{R}$, then this does not hold anymore. For, we do not have $+\ (\mathsf{zero}\ \mathsf{f})\ \mathsf{0} \overset{\mathrm{ext}*}{\leftrightarrow}_{\mathcal{R}}^{*} +\ \mathsf{0}\ (\mathsf{zero}\ \mathsf{f})$. Hence, the equation $+\ x\ y \approx +\ y\ x$ is not an inductive theorem of $\mathcal{R} \cup \{\mathsf{f}\ x \to \mathsf{0}\}$.

To see why extensibility is needed, consider the following program transformation. The *recursive* definition of rev, given by $\mathcal{R}_{in}$, is transformed into the *iterative* definition, given by $\mathcal{R}_{out}$. Both TRSs are first-order and given by:

$$\mathcal{R}_{in} = \left\{ \begin{array}{ll} \mathsf{rev}([\,]) & \to [\,] \\ \mathsf{rev}(x:xs) & \to \mathsf{app}(\mathsf{rev}(xs), x:[\,]) \\ \mathsf{app}([\,], ys) & \to ys \\ \mathsf{app}(x:xs, ys) & \to x:\mathsf{app}(xs, ys) \end{array} \right\} \quad \Rightarrow \quad \mathcal{R}_{out} = \left\{ \begin{array}{ll} \mathsf{rev}(xs) & \to \mathsf{rev1}(xs, [\,]) \\ \mathsf{rev1}([\,], ys) & \to ys \\ \mathsf{rev1}(x:xs, ys) & \to \mathsf{rev1}(xs, x:ys) \\ \mathsf{app}([\,], ys) & \to ys \\ \mathsf{app}(x:xs, ys) & \to x:\mathsf{app}(xs, ys) \end{array} \right\}.$$

The correctness of the transformation is guaranteed by the fact that the equations $\mathsf{app}(xs, [\,]) \approx xs$ and $\mathsf{app}(\mathsf{app}(xs, ys), zs) \approx \mathsf{app}(xs, \mathsf{app}(ys, zs))$ are inductively valid w.r.t. the input TRS $\mathcal{R}_{in}$. The transformation is carried out in three steps: $\mathcal{R}_{in} \overset{*}{\Rightarrow}_I \mathcal{R}_I \overset{*}{\Rightarrow}_A \mathcal{R}_A \overset{*}{\Rightarrow}_E \mathcal{R}_{out}$. In the first step, the definition of a new function $\mathsf{rev1}$ is introduced as $\mathsf{rev1}(xs, ys) \to \mathsf{app}(\mathsf{rev}(xs), ys)$. Note that the definition of $\mathsf{rev1}$ given here is defined in terms of the original rev and app functions and is different from the final form occurring in $\mathcal{R}_{out}$ which is defined recursively. In the second step, new rewrite rules which are inductively valid are added. For example, the rewrite rule $\mathsf{rev1}(x:xs, ys) \to \mathsf{app}(\mathsf{rev}(xs), x:ys)$ is added based on the inductive equivalence $\mathsf{rev1}(x:xs, ys) \leftrightarrow_{\mathcal{R}_I} \mathsf{app}(\mathsf{rev}(x:xs), ys) \leftrightarrow_{\mathcal{R}_I} \mathsf{app}(\mathsf{app}(\mathsf{rev}(xs), x:[\,]), ys) \approx \mathsf{app}(\mathsf{rev}(xs), \mathsf{app}(x:[\,], ys)) \leftrightarrow_{\mathcal{R}_I} \mathsf{app}(\mathsf{rev}(xs), x:\mathsf{app}([\,], ys)) \leftrightarrow_{\mathcal{R}_I} \mathsf{app}(\mathsf{rev}(xs), x:ys)$. Likewise, $\mathsf{rev}(xs) \to \mathsf{rev1}(xs, [\,])$ and $\mathsf{rev1}([\,], ys) \to ys$ are added. In the last step, auxiliary rewrite rules (typically original rules) are eliminated. By extensibility of first-order inductive theorems, the inductive theorems of $\mathcal{R}_{in}$ are still inductively valid in $\mathcal{R}_I$, and thus one can use inductive theorems safely in the second step $\mathcal{R}_I \overset{*}{\Rightarrow}_A \mathcal{R}_A$, after the introduction of $\mathsf{rev1}$ in the first step.

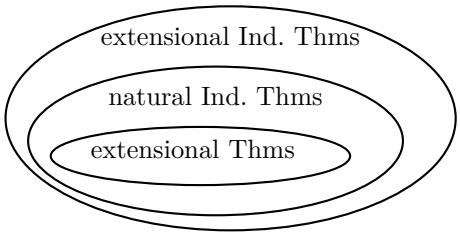


Figure 1 Inclusion relation on the three notions of theorems

The lack of extensibility for higher-order inductive theorems prevents us from extending the template based framework for program transformations of [7, 8, 9] to the higher-order setting. To overcome this difficulty, we introduce in this paper a new notion of inductive theorems—*natural inductive theorems*—for higher-order rewriting satisfying the following properties: (1) these inductive theorems are extensional and extensible, (2) extensional theorems are natural inductive theorems, and (3) natural inductive theorems are extensional inductive theorems (see Figure 1). Once the notion of natural inductive theorems is obtained, the higher-order extension of the framework is achieved in the following way. As in the first-order case, we first establish some natural inductive theorems of the input STSRS $\mathcal{R}_{in}$. Then a transformation $\mathcal{R}_{in} \overset{*}{\Rightarrow}_I \mathcal{R}_I \overset{*}{\Rightarrow}_A \mathcal{R}_A \overset{*}{\Rightarrow}_E \mathcal{R}_{out}$ is performed as before. By extensibility, natural inductive theorems are preserved in the transformation $\overset{*}{\Rightarrow}_I$. This, together with the property (2), allows to add new rules which are sound w.r.t. natural inductive validity. Hence the equivalence of $\mathcal{R}_{in}$ and $\mathcal{R}_{out}$ is obtained w.r.t. natural inductive validity. By property (3) this ensures the equivalence of input and output STSRSs w.r.t. extensional inductive validity.

The remainder of this paper is structured as follows. Having fixed the terminology and notations used in this paper (Section 2), we review a semantics of simply typed equational theories that captures extensionality (Section 3). In Section 4, we arrive at the restriction of simply typed algebras to give a notion of natural inductive theorems. Then we show that the set of natural inductive theorems covers that of extensional theorems and is covered by that of extensional inductive theorems. We then show extensibility of natural inductive theorems under certain conditions. In Section 5, we give a sufficient condition that partially allows to check whether an equation is a natural inductive theorem. Section 6 concludes.

2 Preliminaries

In this section, we briefly recall the terminology and notations of simply typed S-expression rewriting (simply typed term rewriting in [21]).

Let B be a set of *base types*. The set ST of simple types is defined inductively as: $B \subseteq \mathrm{ST}$; if $\tau_0, \ldots, \tau_n \in \mathrm{ST}$ then $\tau_1 \times \cdots \times \tau_n \to \tau_0 \in \mathrm{ST}$ $(n \geq 1)$. Non-base types are called *function types*. A set $T \subseteq \mathrm{ST}$ of simple types is a *simple type structure* if (1) $B \subseteq T$ and (2) T is closed under subtypes, i.e. $\tau_1 \times \cdots \times \tau_n \to \tau_0 \in T$ implies $\tau_0, \ldots, \tau_n \in T$. For any simple type structure T, we put $T^{\mathrm{f}} = T \setminus B$. *Second-order* simple types are defined inductively as follows: (1) base types are second-order simple types, (2) if τ_0 is a second-order simple type and $\tau_1, \ldots, \tau_n$ are base types then $\tau_1 \times \cdots \times \tau_n \to \tau_0$ is a second-order simple type. Let Σ be a set of *constants* and V the set of *variables*. Each constant or variable a is equiped with

a simple type (denoted by $\mathrm{type}(a)$). We assume that there are countably infinite variables of type τ for each $\tau \in \mathrm{ST}$. For any $\tau \in \mathrm{ST}$ and $U \subseteq \Sigma \cup V$, we put $U^\tau = \{a \in U \mid \mathrm{type}(a) = \tau\}$. Let T be a simple type structure. We say $a \in \Sigma \cup V$ ($U \subseteq \Sigma \cup V$) is *over* T if $\mathrm{type}(a) \in T$ ($U \subseteq \bigcup_{\tau \in T} U^\tau$, respectively). A simply typed constant is said to be second-order if its type is second-order. We assume that the set Σ of constants is partitioned into two categories[1]: the set Σ_d of *defined* constants and the set Σ_c of *constructor* constants. The set Σ is said to be *elementary* if any constructor constant $c \in \Sigma_\mathrm{c}$ is second-order.

Let Σ be a set of constants over a simple type structure T and X be a set of variables over T. The set $\mathrm{S}(\Sigma, X)^\tau$ of *simply typed S-expressions* of type $\tau \in T$ over Σ and X is defined as follows: (1) $\Sigma^\tau \cup X^\tau \subseteq \mathrm{S}(\Sigma, X)^\tau$, (2) if $t_0 \in \mathrm{S}(\Sigma, X)^{\tau_1 \times \cdots \times \tau_n \to \tau}$ and $t_i \in \mathrm{S}(\Sigma, X)^{\tau_i}$ for all $i \in \{1, \ldots, n\}$ then $(t_0\ t_1\ \cdots\ t_n) \in \mathrm{S}(\Sigma, X)^\tau$. The outermost parentheses of an S-expression can be omitted. The set of all simply typed S-expressions over Σ and X is denoted by $\mathrm{S}(\Sigma, X)$. We often refer to simply typed S-expressions as S-expressions, for brevity. The type of an S-expression t is denoted by $\mathrm{type}(t)$. For any set $U \subseteq \mathrm{S}(\Sigma, V)$, we put $U^\mathrm{b} = \{s \in U \mid \mathrm{type}(s) \in B\}$ and $U^\mathrm{f} = \{s \in U \mid \mathrm{type}(s) \notin B\}$. The set of variables in an S-expression t (of base type, of function type) is denoted by $\mathrm{V}(t)$ ($\mathrm{V}^\mathrm{b}(t)$, $\mathrm{V}^\mathrm{f}(t)$, respectively). An S-expression t is said to be *ground* if $\mathrm{V}(t) = \emptyset$. The set of ground S-expressions is denoted by $\mathrm{S}(\Sigma)$. An S-expression is *linear* if every variable occurs at most once in it. The *head symbol* of an S-expression is defined recursively as follows: $\mathrm{head}(a) = a$ for $a \in \Sigma \cup V$; $\mathrm{head}((t_0\ t_1 \cdots t_n)) = \mathrm{head}(t_0)$. The set $\mathrm{Args}(s)$ of arguments of an S-expression s is defined recursively as follows: $\mathrm{Args}(a) = \emptyset$ for $a \in \Sigma \cup V$; $\mathrm{Args}((t_0\ t_1 \cdots t_n)) = \mathrm{Args}(t_0) \cup \{t_1, \ldots, t_n\}$. A *full expansion* $t{\uparrow}$ of an S-expression t is defined recursively as follows: (1) if $\mathrm{type}(t) \in B$ then $t{\uparrow} = t$, (2) if $\mathrm{type}(t) = \tau_1 \times \cdots \times \tau_n \to \tau_0$ then $t{\uparrow} = (t\ x_1 \cdots x_n){\uparrow}$ where $x_1, \ldots, x_n$ are fresh variables of type $\tau_1, \ldots, \tau_n$, respectively.

A simply typed *context* over Σ and X is a simply typed S-expression over Σ and X that contains special symbols $\Box^\tau$, called the *holes*, prepared for each type $\tau \in T$. Let C be a context having a hole of type τ. The S-expression obtained by replacing the hole in C with an S-expression t of the same type is denoted by $C[t]$. A context of the form $\Box^\tau$ is said to be *empty*. We omit the type of a hole when it is not important. An S-expression s is a *subexpression* of an S-expression t (denoted by $s \trianglelefteq t$) if $C[s] = t$ for some context $C[\,]$.

A simply typed *substitution* over Σ is a mapping $\sigma : V \to \mathrm{S}(\Sigma, V)$ such that $\mathrm{type}(x) = \mathrm{type}(\sigma(x))$ for all $x \in V$ and $\mathrm{dom}(\sigma) = \{x \mid \sigma(x) \neq x\}$ is finite. The set $\mathrm{dom}(\sigma)$ is called the *domain* of σ. The *range* of σ is given by $\mathrm{ran}(\sigma) = \{\sigma(x) \mid x \in \mathrm{dom}(\sigma)\}$. For a substitution σ, we write $\sigma : U \to W$ if $\mathrm{dom}(\sigma) \subseteq U$ and $\mathrm{ran}(\sigma) \subseteq W$. As usual, we identify a substitution with its homomorphic extension.

An *instance* of an S-expression t is written as $t\sigma$. When we write $t\sigma$ for a substitution $\sigma : U \to W$, we assume that $\mathrm{V}(t) \cap U \subseteq \mathrm{dom}(\sigma)$. A substitution σ is ground if $\mathrm{ran}(\sigma) \subseteq \mathrm{S}(\Sigma)$. For a set $Y \subseteq X$ and a substitution σ over Σ and X, $\sigma{\restriction_Y}$ denotes a substitution given by $\sigma{\restriction_Y}(x) = \sigma(x)$ for $x \in Y$, $\sigma{\restriction_Y}(x) = x$ otherwise.

Let Σ be a set of constants over T. A *simply typed rewrite rule* $l \to r$ over Σ is a pair of simply typed S-expressions over Σ and $X = \bigcup_{\tau \in T} V^\tau$ which satisfies the following conditions: (1) $\mathrm{type}(l) = \mathrm{type}(r)$, (2) $\mathrm{head}(l) \in \Sigma$ and (3) $\mathrm{V}(r) \subseteq \mathrm{V}(l)$. A set $\mathcal{R}$ of rewrite rules over Σ is called a *simply typed S-expression rewriting system* (STSRS for short) over Σ. Let Y be a set of variables over T. For any $s, t \in \mathrm{S}(\Sigma, Y)$, we have $s \to_\mathcal{R} t$ if $s = C[l\sigma]$ and $t = C[r\sigma]$ for some rewrite rule $l \to r \in \mathcal{R}$, context $C[\,]$ over Σ and Y, and substitution $\sigma : X \to \mathrm{S}(\Sigma, Y)$.

[1] We do not assume in this paper that Σ_d coincides with the set of head symbols of left-hand sides of the rewrite rules, i.e. $\{\mathrm{head}(l) \mid l \to r \in \mathcal{R}\} = \Sigma_\mathrm{d}$.

The relation $\to_{\mathcal{R}}$ (over $\mathrm{S}(\Sigma, Y)$) is called the *rewrite relation* induced by an STSRS $\mathcal{R}$. An STSRS $\mathcal{R}$ is *left-linear* if l is linear for any $l \to r \in \mathcal{R}$. The symmetric closure and the reflexive transitive closure of a relation $\to$ is denoted by $\leftrightarrow$ and $\to^*$, respectively.

A *simply typed equation* over Σ and X is a pair $\langle l, r\rangle$ of simply typed S-expressions over Σ and X such that $\mathrm{type}(l) = \mathrm{type}(r)$. We write $l \approx r$ to denote that $\langle l, r\rangle$ is a simply typed equation. The set of simply typed equations over Σ and X is denoted by $\mathrm{Eqn}(\Sigma, X)$. For any $s \approx t \in \mathrm{Eqn}(\Sigma, X)$ a full expansion $s{\uparrow} \approx t{\uparrow}$ of $s \approx t$ is defined similarly to the full expansion of an S-expression by choosing the same variables in corresponding arguments in left-hand sides (lhss) and right-hand sides (rhss) of the equation. Any $E \subseteq \mathrm{Eqn}(\Sigma, X)$ where Σ is a set of constants over T and X is the set of variables over T is called a $\langle T, \Sigma\rangle$*-theory*. We sometime refer to an STSRS $\mathcal{R}$ over Σ as a $\langle T, \Sigma\rangle$-theory given by $\{l \approx r \mid l \to r \in \mathcal{R}\}$. A $\langle T, \Sigma\rangle$-theory is said to be *elementary* if Σ is elementary.

3 Extensional Semantics

In this section, we present a semantics for simply typed equational theories that captures extensionality and recall some basic results that will be used in the next section. Most of the material is incorporated from [17] into our framework.

▶ **Definition 3.1** (typed algebras)**.** Let Σ be a set of simply typed constants over a simple type structure T. A T-typed Σ-algebra ($\langle T, \Sigma\rangle$-algebra for short) is a triple

$$\mathcal{A} = \langle (A^\tau)_{\tau\in T}, (ap^\tau)_{\tau\in T^{\mathrm{f}}}, (c^{\mathcal{A}})_{c\in\Sigma}\rangle$$

where $(A^\tau)_{\tau\in T}$ are mutually disjoint non-empty sets, $ap^\tau \in [A^\tau \times A^{\tau_1} \times \cdots \times A^{\tau_n} \to A^{\tau_0}]$ for each $\tau = \tau_1 \times \cdots \times \tau_n \to \tau_0 \in T^{\mathrm{f}}$, and $c^{\mathcal{A}} \in A^{\mathrm{type}(c)}$ for each $c \in \Sigma$. Here, for sets $A_0, \ldots, A_n$, $[A_1 \times \cdots \times A_n \to A_0]$ is the set of functions from $A_1 \times \cdots \times A_n$ to A_0. The set $\bigcup_{\tau\in T} A^\tau$ is called the *carrier set* of the algebra $\mathcal{A}$ and denoted by $|\mathcal{A}|$.

We now incorporate standard notions on the validity and equational consequences for our semantics. Let $\mathcal{A} = \langle (A^\tau)_{\tau\in T}, (ap^\tau)_{\tau\in T^{\mathrm{f}}}, (c^{\mathcal{A}})_{c\in\Sigma}\rangle$ be a $\langle T, \Sigma\rangle$-algebra and X the set of variables over T. A family of mappings $\rho = (\rho^\tau)_{\tau\in T}$ where $\rho^\tau \in [X^\tau \to A^\tau]$ is called an *environment* for $\mathcal{A}$. We abbreviate $\rho^\tau(x)$ as $\rho(x)$. For each S-expression $s \in \mathrm{S}(\Sigma, X)$ its *interpretation* $[\![s]\!]_\rho$ in $\mathcal{A}$ over the environment ρ is defined inductively like this: $[\![c]\!]_\rho = c^{\mathcal{A}}$ for each $c \in \Sigma$, $[\![x]\!]_\rho = \rho(x)$ for each $x \in X$, $[\![(s_0\ s_1 \cdots s_n)]\!]_\rho = ap^\tau([\![s_0]\!]_\rho, [\![s_1]\!]_\rho, \ldots, [\![s_n]\!]_\rho)$ where $\tau = \mathrm{type}(s_0)$. An equation $l \approx r \in \mathrm{Eqn}(\Sigma, X)$ is *valid* on $\mathcal{A}$ (denoted by $\mathcal{A} \models l \approx r$) if $[\![l]\!]_\rho = [\![r]\!]_\rho$ for all environments ρ for $\mathcal{A}$. A $\langle T, \Sigma\rangle$-theory E is said to be valid on $\mathcal{A}$ or $\mathcal{A}$ is a *model* of E (denoted by $\mathcal{A} \models E$) if all equations in E are valid on $\mathcal{A}$. An equation $l \approx r \in \mathrm{Eqn}(\Sigma, X)$ is a *theorem of* E or *equational consequence of* E (denoted by $E \models l \approx r$) if $l \approx r$ is valid on every model of E. An equivalence relation $\sim$ on $|\mathcal{A}|$ is said to be a *congruence* on $\mathcal{A}$ if (1) $a \sim b$ implies $a, b \in A^\tau$ for some $\tau \in T$ and (2) $a_0 \sim b_0, a_1 \sim b_1, \ldots, a_n \sim b_n$ implies $ap^\tau(a_0, a_1, \ldots, a_n) \sim ap^\tau(b_0, b_1, \ldots, b_n)$ for any $a_0, b_0 \in A^\tau$, $a_i, b_i \in A^{\tau_i}$ $(1 \le i \le n)$ where $\tau = \tau_1 \times \cdots \times \tau_n \to \tau_0$. We denote the $\sim$-equivalence class containing $a \in |\mathcal{A}|$ by $[a]$ i.e. $[a] = \{b \in |\mathcal{A}| \mid a \sim b\}$. The *quotient algebra* $\mathcal{A}/\!\sim$ has the carrier set $\bigcup_\tau (A/\!\sim)^\tau$ where $(A/\!\sim)^\tau = \{[a] \mid a \in A^\tau\}$, operations $ap^\tau_{\mathcal{A}/\sim}([a_0], [a_1], \ldots, [a_n]) = [ap^\tau_{\mathcal{A}}(a_0, a_1, \ldots, a_n)]$ and $c^{\mathcal{A}/\sim} = [c^{\mathcal{A}}]$ for each $c \in \Sigma$. It is readily checked that for a given $\langle T, \Sigma\rangle$-algebra $\mathcal{A}$ and a congruence $\sim$ on $\mathcal{A}$, the quotient algebra $\mathcal{A}/\!\sim$ is again a $\langle T, \Sigma\rangle$-algebra.

The following lemma will be used later.

$$\frac{l \approx r \in E}{l \approx r}\ \text{ax.} \qquad \frac{}{s \approx s}\ \text{refl.} \qquad \frac{t \approx s}{s \approx t}\ \text{sym.}$$

$$\frac{s \approx t \quad t \approx u}{s \approx u}\ \text{trans.} \qquad \frac{s_0 \approx t_0 \quad \cdots \quad s_n \approx t_n}{(s_0 \cdots s_n) \approx (t_0 \cdots t_n)}\ \text{cong.}$$

$$\frac{s \approx t}{s\theta \approx t\theta}\ \text{subst.} \qquad \frac{(s\ x_1 \cdots x_n) \approx (t\ x_1 \cdots x_n)}{s \approx t}\ \text{ext.}\quad x_1, \ldots, x_n \notin \mathrm{V}(s) \cup \mathrm{V}(t)$$

Figure 2 Inference rules for $E \vdash_{\mathrm{ext}}$

▶ **Lemma 3.2.** Let E be a $\langle T, \Sigma\rangle$-theory, $\mathcal{A}$ a $\langle T, \Sigma\rangle$-algebra and X the set of variables over T. Then $[\![s\theta]\!]_\rho = [\![s]\!]_{\rho/\theta}$ holds for any S-expression $s \in \mathrm{S}(\Sigma, X)$, environment ρ for $\mathcal{A}$ and substitution $\theta : X \to \mathrm{S}(\Sigma, X)$. The environment ρ/θ is defined as: $(\rho/\theta)(x) = [\![\theta(x)]\!]_\rho$.

We next introduce a characterization of $\langle T, \Sigma\rangle$-algebras that incorporates extensionality to the semantics.

▶ **Definition 3.3** (extensional algebras and theorems)**.** Let $\mathcal{A} = \langle (A^\tau)_{\tau \in T}, (ap^\tau)_{\tau \in T^{\mathrm{f}}}, (c^{\mathcal{A}})_{c \in \Sigma}\rangle$ be a $\langle T, \Sigma\rangle$-algebra. Then $\mathcal{A}$ is said to be *extensional* if for all $\tau = \tau_1 \times \cdots \times \tau_n \to \tau_0 \in T^{\mathrm{f}}$ and $a_0, b_0 \in A^\tau$, $a_0 = b_0$ holds whenever $ap^\tau(a_0, a_1, \ldots, a_n) = ap^\tau(b_0, a_1, \ldots, a_n)$ for all $a_1 \in A^{\tau_1}, \ldots, a_n \in A^{\tau_n}$. An equation $l \approx r \in \mathrm{Eqn}(\Sigma, X)$ where X is the set of variables over T is said to be an *extensional theorem* (written as $E \models_{\mathrm{ext}} l \approx r$) if $\mathcal{A} \models E$ implies $\mathcal{A} \models l \approx r$ for every extensional $\langle T, \Sigma\rangle$-algebra $\mathcal{A}$.

Let $\mathcal{A}$ be an extensional $\langle T, \Sigma\rangle$-algebra where $\mathcal{A} = \langle (A^\tau)_{\tau \in T}, (ap^\tau)_{\tau \in T^{\mathrm{f}}}, (c^{\mathcal{A}})_{c \in \Sigma}\rangle$. A congruence $\sim$ on $\mathcal{A}$ is said to be *extensional* if $ap^\tau(a_0, a_1, \ldots, a_n) \sim ap^\tau(b_0, a_1, \ldots, a_n)$ for all $a_1 \in A^{\tau_1}, \ldots, a_n \in A^{\tau_n}$ implies $a_0 \sim b_0$, for all $a_0, b_0 \in A^\tau$ where $\tau = \tau_1 \times \cdots \times \tau_n \to \tau_0$. It is straightforward to show that the quotient algebra $\mathcal{A}/\!\sim$ is an extensional $\langle T, \Sigma\rangle$-algebra if $\sim$ is an extensional congruence on $\mathcal{A}$.

The syntactic counterpart of extensional theorems is given as follows.

▶ **Definition 3.4** (extensional equational deduction)**.** Let E be a $\langle T, \Sigma\rangle$-theory and X the set of variables over T. The inference rules of *extensional equational deduction* are given in Figure 2. We write $E \vdash_{\mathrm{ext}} s \approx t$ if $s \approx t \in \mathrm{Eqn}(\Sigma, X)$ is derivable by extensional equational deduction.

It is easy to see that $E \vdash_{\mathrm{ext}} s \approx t$ if and only if $E \vdash_{\mathrm{ext}} s{\uparrow} \approx t{\uparrow}$.

Our next aim is to develop the completeness theorem for extensional equational deduction (w.r.t. extensional theorems). For this, we need a couple of preparations.

Let E be a $\langle T, \Sigma\rangle$-theory and X the set of variables over T. The *extensional equivalence relation* $\overset{\mathrm{ext}*}{\leftrightarrow}_E$ of E on $\mathrm{S}(\Sigma, X)$ is the smallest equivalence relation satisfying (1) $l \approx r \in E$ implies $l\theta \overset{\mathrm{ext}*}{\leftrightarrow}_E r\theta$ for all substitutions θ, (2) $(s\ x_1 \cdots x_n) \overset{\mathrm{ext}*}{\leftrightarrow}_E (t\ x_1 \cdots x_n)$ implies $s \overset{\mathrm{ext}*}{\leftrightarrow}_E t$ where $x_1, \ldots, x_n \notin \mathrm{V}(s) \cup \mathrm{V}(t)$ and (3) $s_i \overset{\mathrm{ext}*}{\leftrightarrow}_E t_i$ for all $0 \le i \le n$ implies $(s_0 \cdots s_n) \overset{\mathrm{ext}*}{\leftrightarrow}_E (t_0 \cdots t_n)$. It is easy to see that $E \vdash_{\mathrm{ext}} s \approx t$ if and only if $s \overset{\mathrm{ext}*}{\leftrightarrow}_E t$. From here on, we assume[2] that $\mathrm{S}(\Sigma)^\tau \neq \emptyset$ for any $\tau \in T$.

[2] This assumption is required to guarantee the carrier sets of the term algebras satisfy the non-emptiness condition.

Let E be a $\langle T, \Sigma\rangle$-theory, X the set of variables over T, and Y a set of variables such that $Y \subseteq X$. A $\langle T, \Sigma\rangle$-algebra given by

$$\mathcal{T}_\Sigma(Y) = \langle (\mathrm{S}(\Sigma, Y)^\tau)_{\tau\in T}, (ap^\tau)_{\tau\in T^{\mathrm{f}}}, (c^{\mathcal{T}_\Sigma(Y)})_{c\in\Sigma}\rangle$$

where ap^τ and $c^{\mathcal{T}_\Sigma(Y)}$ are defined by $ap^\tau(s_0, s_1, \ldots, s_n) = (s_0\ s_1 \cdots s_n)$ and $c^{\mathcal{T}_\Sigma(Y)} = c$ is called a $\langle T, \Sigma\rangle$-*term algebra (with the set Y of generators)*. Note that by our assumption that $\mathrm{S}(\Sigma)^\tau \neq \emptyset$ for any $\tau \in T$, $\mathrm{S}(\Sigma, Y)^\tau \neq \emptyset$ for any set $Y \subseteq X$ and hence any $\langle T, \Sigma\rangle$-term algebra is a $\langle T, \Sigma\rangle$-algebra. It is not difficult to show that $\overset{\mathrm{ext}_*}{\leftrightarrow}_E$ is an extensional congruence on the $\langle T, \Sigma\rangle$-term algebra $\mathcal{T}_\Sigma(X)$. If the set of generators is an arbitrary $Y \subseteq X$, however, then $\overset{\mathrm{ext}_*}{\leftrightarrow}_E$ may not be an extensional congruence on $\mathcal{T}_\Sigma(Y)$ and cannot be used to define the initial extensional $\langle T, \Sigma\rangle$-algebra. To overcome this, Meinke [17] introduced an ω-evaluation rule. Here we use the following equivalence relation $\overset{\mathrm{ext}_*}{\leftrightarrow}_{E,\omega}$. Let E be a $\langle T, \Sigma\rangle$-theory and Y a set of variables over T. The *ω-extensional equivalence relation* $\overset{\mathrm{ext}_*}{\leftrightarrow}_{E,\omega}$ of E on $\mathrm{S}(\Sigma, Y)$ is obtained by replacing condition (2) in the definition of $\overset{\mathrm{ext}_*}{\leftrightarrow}_E$ by (2′) $(s\ u_1 \cdots u_n) \overset{\mathrm{ext}_*}{\leftrightarrow}_{E,\omega} (t\ u_1 \cdots u_n)$ for any $u_1 \in \mathrm{S}(\Sigma, Y)^{\tau_1}, \ldots, u_n \in \mathrm{S}(\Sigma, Y)^{\tau_n}$ implies $s \overset{\mathrm{ext}_*}{\leftrightarrow}_{E,\omega} t$, where $\mathrm{type}(s) = \mathrm{type}(t) = \tau_1 \times \cdots \times \tau_n \to \tau_0$. Then $\overset{\mathrm{ext}_*}{\leftrightarrow}_{E,\omega}$ is an extensional congruence on any $\langle T, \Sigma\rangle$-term algebra $\mathcal{T}_\Sigma(Y)$. Hence we get an extensional $\langle T, \Sigma\rangle$-algebra $\mathcal{T}_E(Y) = \mathcal{T}_\Sigma(Y)/\overset{\mathrm{ext}_*}{\leftrightarrow}_{E,\omega}$. It is easy to see that for any $s\uparrow \approx t\uparrow \in \mathrm{Eqn}(\Sigma, Y)$, $\mathcal{T}_E(Y) \models s \approx t$ if and only if $\mathcal{T}_E(Y) \models s\uparrow \approx t\uparrow$.

Using standard arguments [5], the soundness and completeness of extensional equational deduction can be shown [17].

▶ **Theorem 3.5** (soundness and completeness [17])**.** Let E be a $\langle T, \Sigma\rangle$-theory and X the set of variables over T. For any $l \approx r \in \mathrm{Eqn}(\Sigma, X)$, $E \vdash_{\mathrm{ext}} l \approx r$ if and only if $E \models_{\mathrm{ext}} l \approx r$.

Our extensional semantics naturally leads to the notion of extensional inductive theorems.

▶ **Definition 3.6** (extensional inductive theorem [17])**.** Let E be a $\langle T, \Sigma\rangle$-theory and X the set of variables over T. An equation $s \approx t \in \mathrm{Eqn}(\Sigma, X)$ is said to be an *extensional inductive theorem* of E (denoted by $E \models_{\mathrm{eind}} s \approx t$) if $\mathcal{T}_E(\emptyset) \models s \approx t$.

The following characterization of extensional inductive theorems will be used later.

▶ **Lemma 3.7.** Let E be a $\langle T, \Sigma\rangle$-theory and X the set of variables over T. For any $s \approx t \in \mathrm{Eqn}(\Sigma, X)$, $E \models_{\mathrm{eind}} s \approx t$ if $s\theta \overset{\mathrm{ext}_*}{\leftrightarrow}_E t\theta$ (on $\mathrm{S}(\Sigma, X)$) for any ground substitution $\theta : X \to \mathrm{S}(\Sigma)$.

4 Natural Semantics and Natural Inductive Theorems

Extensional semantics developed in the previous section and the notion of extensional inductive theorems introduced there seems to form a firm basis for simply typed equational theories. However, the notion of extensional inductive theorems lacks a property of inductive theorems in first-order term rewriting: *extensibility*, meaning that inductive theorems are preserved when a new function by a macro is added.

▶ **Example 4.1.** Let $T = \{\mathrm{Nat},\ \mathrm{Nat} \to \mathrm{Nat},\ \mathrm{Nat} \times \mathrm{Nat} \to \mathrm{Nat},\ (\mathrm{Nat} \to \mathrm{Nat}) \to \mathrm{Nat}\}$, $\Sigma = \{+^{\mathrm{Nat}\times\mathrm{Nat}\to\mathrm{Nat}}, \mathsf{0}^{\mathrm{Nat}}, \mathsf{s}^{\mathrm{Nat}\to\mathrm{Nat}}, \mathsf{zero}^{(\mathrm{Nat}\to\mathrm{Nat})\to\mathrm{Nat}}\}$ and

$$E = \left\{ \begin{array}{lcl} +\ \mathsf{0}\ y & \approx & y \\ +\ (\mathsf{s}\ x)\ y & \approx & \mathsf{s}\ (+\ x\ y) \\ \mathsf{zero}\ \mathsf{s} & \approx & \mathsf{0} \end{array} \right\}.$$

Then $E \models_{\mathrm{eind}} \mathsf{zero}\ F \approx \mathsf{0}$. (For, the only possible instantiation of F is s.) By introducing a new constant $\mathsf{id}^{\mathrm{Nat}\to\mathrm{Nat}}$, define $E' = \{\mathsf{id}\ x \approx x\} \cup E$. Then we do not have $E' \models_{\mathrm{eind}} \mathsf{zero}\ F \approx \mathsf{0}$ any more, since $\mathsf{zero}\ \mathsf{id} \overset{\mathrm{ext}*}{\leftrightarrow}_{E',\omega} \mathsf{0}$ does not hold.

From this example, it is observed that we may not conclude $\mathsf{zero}\ F \approx \mathsf{0}$ is an "inductive theorem" since this fact depends on the limited possibility of instantiating the variable F of function type. Hence this example suggests that the notion of extensional inductive theorems may be too general if validity needs to be preserved under addition of new function definitions. This motivates us to restrict extensional $\langle T, \Sigma\rangle$-algebras to *natural* $\langle T, \Sigma\rangle$-algebras.

▶ **Definition 4.2** (natural algebras)**.** A $\langle T, \Sigma\rangle$-algebra $\mathcal{A} = \langle (A^\tau)_{\tau\in T}, (ap^\tau)_{\tau\in T^{\mathrm{f}}}, (c^{\mathcal{A}})_{c\in\Sigma}\rangle$ is said to be *natural* if for any $\tau \in T^{\mathrm{f}}$ with $\tau = \tau_1 \times \cdots \times \tau_n \to \tau_0$, (1) $A^\tau = [A^{\tau_1} \times \cdots \times A^{\tau_n} \to A^{\tau_0}]$ and (2) $ap^\tau(f, a_1, \ldots, a_n) = f(a_1, \ldots, a_n)$. Henceforth, a natural $\langle T, \Sigma\rangle$-algebra $\mathcal{A} = \langle (A^\tau)_{\tau\in T}, (ap^\tau)_{\tau\in T^{\mathrm{f}}}, (c^{\mathcal{A}})_{c\in\Sigma}\rangle$ is specified as $\mathcal{A} = \langle (A^\tau)_{\tau\in B}, (c^{\mathcal{A}})_{c\in\Sigma}\rangle$. A natural $\langle T, \Sigma\rangle$-algebra $\langle (A^\tau)_{\tau\in B}, (c^{\mathcal{A}})_{c\in\Sigma}\rangle$ is a *natural $\langle T, \Sigma\rangle$-term algebra (with the set X of generators)* if there exists $\Sigma' \subseteq \Sigma$ such that $A^\tau = \mathrm{S}(\Sigma', X)^\tau$ for each $\tau \in B$.

▶ **Example 4.3.** Consider T, Σ, and E from Example 4.1 and let $\Sigma' = \{\mathsf{0}, \mathsf{s}\}$. Let X be a set of variables over T and put $A^{\mathrm{Nat}} = \mathrm{S}(\Sigma', X)^{\mathrm{Nat}}$. Take $A^\tau = [A^{\tau_1} \times \cdots \times A^{\tau_n} \to A^{\tau_0}]$ for $\tau = \tau_1 \times \cdots \times \tau_n \to \tau_0 \in T^{\mathrm{f}}$ and any $c^{\mathcal{A}}$ for each $c^\tau \in \{\mathsf{0}, \mathsf{s}, +, \mathsf{zero}\}$ such that $c^{\mathcal{A}} \in A^\tau$. Then $\langle (A^\tau)_{\tau\in B}, (c^{\mathcal{A}})_{c\in\Sigma}\rangle$ is a natural $\langle T, \Sigma\rangle$-term algebra (with the set X of generators).

▶ **Lemma 4.4.** Any natural $\langle T, \Sigma\rangle$-algebra is extensional.

Proof. For any $f, g \in [A^{\tau_1} \times \cdots \times A^{\tau_n} \to A^{\tau_0}]$, $f = g$ iff $f(a_1, \ldots, a_n) = g(a_1, \ldots, a_n)$ holds for any $a_1 \in A^{\tau_1}, \ldots, a_n \in A^{\tau_n}$. ◀

Extensional inductive theorems were defined (in Definition 3.6) based on the $\langle T, \Sigma\rangle$-term algebra with the empty set of generators. Similarly, we will define a notion of *natural inductive theorems* from natural $\langle T, \Sigma\rangle$-term algebras with the empty set of generators. It is, however, not possible to directly relate the notion of natural inductive theorems to those of extensional theorems and extensional inductive theorems; we further require *consistency* of the natural $\langle T, \Sigma\rangle$-term algebras to connect these notions.

▶ **Definition 4.5** (natural inductive theorems)**.** Let E be a $\langle T, \Sigma\rangle$-theory and X be the set of variables over T. Furthermore, assume that the set Σ_{c} of constructors is *free*, i.e. for any $s, t \in \mathrm{S}(\Sigma_{\mathrm{c}}, X)$ $s \overset{\mathrm{ext}*}{\leftrightarrow}_E t$ implies $s = t$.

1. A natural $\langle T, \Sigma\rangle$-term algebra $\mathcal{A} = \langle (A^\tau)_{\tau\in B}, (c^{\mathcal{A}})_{c\in\Sigma}\rangle$ is said to be *consistent* with E if (1) $s \overset{\mathrm{ext}*}{\leftrightarrow}_E [\![s]\!]$ holds for any $s \in \mathrm{S}(\Sigma)^{\mathrm{b}}$ and (2) $\mathcal{A} \models E$. Here, note that ρ of $[\![s]\!]_\rho$ can be safely omitted because $\mathrm{V}(s) = \emptyset$.
2. A natural $\langle T, \Sigma\rangle$-term algebra $\mathcal{A} = \langle (A)_{\tau\in B}, (c^{\mathcal{A}})_{c\in\Sigma}\rangle$ is said to be a *natural $\langle T, \Sigma\rangle$-term algebra for E* if $A^\tau = \mathrm{S}(\Sigma_{\mathrm{c}})^\tau$ for each $\tau \in B$ and $\mathcal{A}$ is consistent with E, where Σ_{c} is the set of constructors designated in E. E is said to be a *natural $\langle T, \Sigma\rangle$-theory* if there exists a natural $\langle T, \Sigma\rangle$-term algebra $\mathcal{A}$ for E.
3. Suppose that E is a natural $\langle T, \Sigma\rangle$-theory. Then an equation $l \approx r \in \mathrm{Eqn}(\Sigma, X)$ is said to be a *natural inductive theorem* of E (denoted by $E \models_{\mathrm{nind}} l \approx r$) if $\mathcal{A} \models l \approx r$ for any natural $\langle T, \Sigma\rangle$-term algebra $\mathcal{A}$ for E.

▶ **Example 4.6.** Consider T, Σ, and E from Example 4.1 and let $\Sigma_{\mathrm{c}} = \{\mathsf{0}, \mathsf{s}\}$. Put $A^{\mathrm{Nat}} = \mathrm{S}(\Sigma_{\mathrm{c}})^{\mathrm{Nat}}$ and $A^\tau = [A^{\tau_1} \times \cdots \times A^{\tau_n} \to A^{\tau_0}]$ for $\tau = \tau_1 \times \cdots \times \tau_n \to \tau_0 \in T^{\mathrm{f}}$. Let $\mathsf{0}^{\mathcal{A}} = \mathsf{0}$, $\mathsf{s}^{\mathcal{A}}(x) = (\mathsf{s}\ x)$, $+^{\mathcal{A}}(x, y)$ be the unique normal form of $(+\ x\ y)$ w.r.t. $\{l \to r \mid l \approx r \in E\}$,

and $\mathsf{zero}^{\mathcal{A}}(f) = \mathsf{0}$. Then $\mathcal{A} = \langle (A^{\tau})_{\tau \in B}, (c^{\mathcal{A}})_{c \in \Sigma} \rangle$ is a natural $\langle T, \Sigma \rangle$-term algebra for E. For example, $(+\ (\mathsf{s}\ \mathsf{0})\ (\mathsf{s}\ \mathsf{0})) \overset{\mathrm{ext}*}{\leftrightarrow}_E (\mathsf{s}\ (\mathsf{s}\ \mathsf{0})) = [\![(+\ (\mathsf{s}\ \mathsf{0})\ (\mathsf{s}\ \mathsf{0}))]\!]$ holds for (1) and $[\![(+\ 0\ y)]\!]_\rho = \rho(y) = [\![y]\!]_\rho$ holds for any ρ for (2). One can also set $\mathsf{zero}^{\mathcal{A}}(succ) = \mathsf{0}$ for $succ \in [\mathrm{S}(\Sigma_{\mathrm{c}})^{\mathrm{Nat}} \to \mathrm{S}(\Sigma_{\mathrm{c}})^{\mathrm{Nat}}]$ such that $succ(x) = (\mathsf{s}\ x)$; $\mathsf{zero}^{\mathcal{A}}(f) = (\mathsf{s}\ \mathsf{0})$ otherwise, to obtain a natural $\langle T, \Sigma \rangle$-term algebra for E. This implies that an equation $(\mathsf{zero}\ F) \approx \mathsf{0}$ is not a natural inductive theorem of E. In contrast, interpretations $\mathsf{s}^{\mathcal{A}}$ and $+^{\mathcal{A}}$ are common to all natural $\langle T, \Sigma \rangle$-term algebras for E and hence it follows that an equation $(+\ x\ y) \approx (+\ y\ x)$ is a natural inductive theorem of E.

Our first aim is to show the relation of natural inductive theorems to extensional theorems and extensional inductive theorems.

▶ **Lemma 4.7.** Let E be a natural $\langle T, \Sigma \rangle$-theory. The set of natural inductive theorems of E is closed under the inference rules of Figure 2.

Proof. This follows from the fact that for any extensional $\langle T, \Sigma \rangle$-algebra $\mathcal{A}$ for E, the set $\mathrm{Th}(\mathcal{A}) = \{s \approx t \mid \mathcal{A} \models s \approx t\}$ is closed under the inference rules of Figure 2. ◀

It easily follows from this lemma that for any equation $l \approx r \in \mathrm{Eqn}(\Sigma, X)$, $E \models_{\mathrm{nind}} l \approx r$ iff $E \models_{\mathrm{nind}} l{\uparrow} \approx r{\uparrow}$.

▶ **Lemma 4.8.** Let $\mathcal{A} = \langle (A^{\tau})_{\tau \in B}, (c^{\mathcal{A}})_{c \in \Sigma} \rangle$ be a natural $\langle T, \Sigma \rangle$-term algebra for E. For any $s, t \in \mathrm{S}(\Sigma)^{\mathrm{b}}$, $s \overset{\mathrm{ext}*}{\leftrightarrow}_E t$ iff $[\![s]\!] = [\![t]\!]$.

Proof. Let $s, t \in \mathrm{S}(\Sigma)^{\mathrm{b}}$. By condition (1) of consistency, $[\![s]\!] \overset{\mathrm{ext}*}{\leftrightarrow}_E s$ and $[\![t]\!] \overset{\mathrm{ext}*}{\leftrightarrow}_E t$. Hence, $s \overset{\mathrm{ext}*}{\leftrightarrow}_E t$ iff $[\![s]\!] \overset{\mathrm{ext}*}{\leftrightarrow}_E [\![t]\!]$. Furthermore, since $[\![s]\!], [\![t]\!] \in \mathrm{S}(\Sigma_{\mathrm{c}})$, $[\![s]\!] \overset{\mathrm{ext}*}{\leftrightarrow}_E [\![t]\!]$ iff $[\![s]\!] = [\![t]\!]$ by our assumption that Σ_{c} is free (Definition 4.5). ◀

We arrive at one of the main theorems of this section.

▶ **Theorem 4.9.** Let E be a natural $\langle T, \Sigma \rangle$-theory and X be the set of variables over T. For any $l \approx r \in \mathrm{Eqn}(\Sigma, X)$, (1) $E \models_{\mathrm{ext}} l \approx r$ implies $E \models_{\mathrm{nind}} l \approx r$; (2) $E \models_{\mathrm{nind}} l \approx r$ implies $E \models_{\mathrm{eind}} l \approx r$.

Proof. (1) By Lemma 4.7. (2) Since $E \models_{\mathrm{nind}} l \approx r$ iff $E \models_{\mathrm{nind}} l{\uparrow} \approx r{\uparrow}$ holds and $E \models_{\mathrm{eind}} l \approx r$ iff $E \models_{\mathrm{eind}} l{\uparrow} \approx r{\uparrow}$ holds, w.l.o.g. we assume that l, r have a base type. If $l \approx r$ is a natural inductive theorem of E then so is $l\theta \approx r\theta$ for any ground substitution θ by Lemma 4.7. Thus $[\![l\theta]\!] = [\![r\theta]\!]$ for any natural $\langle T, \Sigma \rangle$-term algebra $\mathcal{A}$ for E. Then by Lemma 4.8, $l\theta \overset{\mathrm{ext}*}{\leftrightarrow}_E r\theta$. Thus, by Lemma 3.7, $E \models_{\mathrm{eind}} l \approx r$. ◀

Our next aim is to show extensibility of natural inductive theorems. We introduce two new conditions for this.

▶ **Definition 4.10** (constructor-based theories)**.** A $\langle T, \Sigma \rangle$-theory E is said to be *constructor-based* if for any $f \in \Sigma_{\mathrm{d}}$ and any substitution $\sigma : \mathrm{V}^{\mathrm{b}}(f{\uparrow}) \to \mathrm{S}(\Sigma_{\mathrm{c}})$, there exists $t \in \mathrm{S}(\Sigma_{\mathrm{c}}, \mathrm{V}^{\mathrm{f}}(f{\uparrow}))$ such that $(f{\uparrow})\sigma \overset{\mathrm{ext}*}{\leftrightarrow}_E t$.

▶ **Definition 4.11** (conservative extensions)**.** Let E be a $\langle T, \Sigma \rangle$-theory and E' a Σ'-theory such that $E \subseteq E'$ and $\Sigma \subseteq \Sigma'$. Then E' is said to be a *conservative extension of* E if (1) $\Sigma_{\mathrm{c}} = \Sigma'_{\mathrm{c}}$ and (2) for all S-expressions $s, t \in \mathrm{S}(\Sigma_{\mathrm{c}})$, $s \overset{\mathrm{ext}*}{\leftrightarrow}_E t$ iff $s \overset{\mathrm{ext}*}{\leftrightarrow}_{E'} t$.

We introduce a saturated set for E to prove a property of elementary constructor-based theories (Lemma 4.16).

▶ **Definition 4.12** (saturated sets for E)**.** Let E be a $\langle T, \Sigma\rangle$-theory. We define a set W^τ for each $\tau \in T$ like this: $W^\tau = \{s \in \mathrm{S}(\Sigma)^\tau \mid \exists t \in \mathrm{S}(\Sigma_\mathrm{c})\ s \overset{\mathrm{ext}*}{\leftrightarrow}_E t\}$ for $\tau \in B$; $W^\tau = \{s_0 \in \mathrm{S}(\Sigma)^\tau \mid \forall s_1 \in W^{\tau_1} \cdots \forall s_n \in W^{\tau_n}\ (s_0\ s_1 \cdots s_n) \in W^{\tau_0}\}$ if $\tau = \tau_1 \times \cdots \times \tau_n \to \tau_0 \in T^\mathrm{f}$. The *saturated set* W for E is given by $W = \bigcup_{\tau \in T} W^\tau$.

▶ **Lemma 4.13.** Let W be the saturated set for a $\langle T, \Sigma\rangle$-theory E. (1) For any S-expression $s \in \mathrm{S}(\Sigma)$, $s \in W$ iff $(s\uparrow)\sigma \in W$ for any substitution $\sigma : \mathrm{V}(s\uparrow) \to W$. (2) If $s \in W$ and $s_g \overset{\mathrm{ext}*}{\leftrightarrow}_E t$ then $t \in W$.

Proof. (1) By definition. (2) By (1). ◀

▶ **Lemma 4.14.** Let E be an elementary $\langle T, \Sigma\rangle$-theory, X the set of variables over T and W the saturated set for E. Then $s\theta \in W$ holds for any S-expression $s \in \mathrm{S}(\Sigma_\mathrm{c}, X)$ and substitution $\theta : X \to W$.

Proof. By induction on s. ◀

▶ **Lemma 4.15.** Let E be an elementary constructor-based $\langle T, \Sigma\rangle$-theory and W the saturated set for E. Then $\mathrm{S}(\Sigma) = W$.

Proof. Use Lemmas 4.13 and 4.14 to show that $(s\uparrow)\sigma \in W$ for every substitution $\sigma : \mathrm{V}(s\uparrow) \to W$ by induction on $s \in \mathrm{S}(\Sigma)$. ◀

The next lemma follows immediately from Lemma 4.15.

▶ **Lemma 4.16.** If E is an elementary constructor-based $\langle T, \Sigma\rangle$-theory then for any S-expression $s \in \mathrm{S}(\Sigma)^\mathrm{b}$ there exists an S-expression $t \in \mathrm{S}(\Sigma_\mathrm{c})^\mathrm{b}$ such that $s \overset{\mathrm{ext}*}{\leftrightarrow}_E t$.

We arrive at the other main theorem of this section.

▶ **Theorem 4.17** (extensibility of natural inductive theorems)**.** Let E be an elementary constructor-based natural $\langle T, \Sigma\rangle$-theory, X be the set of variables over T, f a new defined symbol of second-order type τ such that $f \notin \Sigma$ and $r \in \mathrm{S}(\Sigma, \{x_1, \ldots, x_n\})$ where $x_1, \ldots, x_n \in X^\mathrm{b}$. Let $T' = T \cup \{\tau\}$, $\Sigma' = \Sigma \cup \{f\}$ and suppose $E' = E \cup \{f\ x_1 \cdots x_n \approx r\}$ is a conservative extension of E. Then the following hold. (1) E' is an elementary constructor-based natural $\langle T', \Sigma'\rangle$-theory. (2) For any $s \approx t \in \mathrm{Eqn}(\Sigma, X)$, $E \models_\mathrm{nind} s \approx t$ iff $E' \models_\mathrm{nind} s \approx t$.

Proof. We first show ($\Rightarrow$) of (2). Suppose that there exists a natural $\langle T', \Sigma'\rangle$-term algebra $\mathcal{A}' = \langle (A'^\tau)_{\tau \in B}, (c^{\mathcal{A}'})_{c \in \Sigma'}\rangle$ for E' such that $s \approx t$ does not hold. By just omitting $f^{\mathcal{A}'}$ (and A'^τ for $\tau \in T' \backslash T$), we obtain a natural $\langle T, \Sigma\rangle$-term algebra $\mathcal{A}$ for E such that $s \approx t$ does not hold. Next we show (1) and ($\Leftarrow$) of (2). Since E is elementary and $\Sigma_\mathrm{c} = \Sigma'_\mathrm{c}$, E' is elementary. Let θ be a substitution such that $\theta : X^\mathrm{b} \to \mathrm{S}(\Sigma_\mathrm{c})$. Then, $(f\uparrow)\theta \overset{\mathrm{ext}*}{\leftrightarrow}_{E'} (r\uparrow)\theta$. Furthermore, by Lemma 4.16, there exists $t \in \mathrm{S}(\Sigma_\mathrm{c})$ such that $(r\uparrow)\theta \overset{\mathrm{ext}*}{\leftrightarrow}_E t$. Thus $(f\uparrow)\theta \overset{\mathrm{ext}*}{\leftrightarrow}_{E'} t$. Hence E' is constructor-based. It remains to show that E' is natural. Since E is a natural $\langle T, \Sigma\rangle$-theory, there exists a natural $\langle T, \Sigma\rangle$-term algebra $\mathcal{A}$ for E. Let $\mathcal{A} = \langle (A^\tau)_{\tau \in B}, (c^{\mathcal{A}})_{c \in \Sigma}\rangle$. Then, by the definition, (1) for any $s \in \mathrm{S}(\Sigma)^\mathrm{b}$, $[\![s]\!] \overset{\mathrm{ext}*}{\leftrightarrow}_E s$ holds, and (2) for any $l \approx r \in E$ and for any environment ρ for $\mathcal{A}$, $[\![l]\!]_\rho = [\![r]\!]_\rho$ holds. We define a natural $\langle T', \Sigma'\rangle$-term algebra $\mathcal{A}'$ by $\mathcal{A}' = \langle (A'^\tau)_{\tau \in B}, (c^{\mathcal{A}'})_{c \in \Sigma'}\rangle$ where $A'^\tau = A^\tau$ for any $\tau \in B$, $c^{\mathcal{A}'} = c^{\mathcal{A}}$ for all $c \in \Sigma$ and $f^{\mathcal{A}'}(a_1, \ldots, a_n) = [\![r]\!]_\rho$ where $\rho = \{x_i \mapsto a_i \mid 1 \le i \le n\}$. We now show that $\mathcal{A}'$ is a natural $\langle T', \Sigma'\rangle$-term algebra for E'.

- $[\![s]\!] \overset{\text{ext}*}{\leftrightarrow}_{E'} s$ holds for any $s \in \mathrm{S}(\Sigma')^{\mathrm{b}}$: Let $s \in \mathrm{S}(\Sigma')^{\mathrm{b}}$. Then, since E' is elementary constructor-based, by Lemma 4.16 there exists an S-expression $u \in \mathrm{S}(\Sigma_{\mathrm{c}})$ such that $s \overset{\text{ext}*}{\leftrightarrow}_{E'} u$. Then $[\![s]\!] = [\![u]\!]$ by Lemma 4.8. Furthermore, $u = [\![u]\!]$ by $u \in \mathrm{S}(\Sigma_{\mathrm{c}})$. Therefore, $[\![s]\!] = [\![u]\!] = u_g \overset{\text{ext}*}{\leftrightarrow}_{E'} s$.
- For any $l \approx r \in E'$ and for any environment ρ for $\mathcal{A}'$, $[\![l]\!]_\rho = [\![r]\!]_\rho$ holds: this follows from the assumption and the definition of $f^{\mathcal{A}'}$.

Thus the proof of (1) has been completed. To show ($\Leftarrow$) of (2), suppose that $s \approx t$ does not hold in $\mathcal{A}$. Then, by construction, $s \approx t$ does not hold in $\mathcal{A}'$ either. ◀

▶ **Example 4.18.** If we drop the condition that f has a second-order type, then E' is not constructor-based in general. Let $T = \{\mathrm{Nat}, \mathrm{Nat} \to \mathrm{Nat}, \mathrm{Nat} \times \mathrm{Nat} \to \mathrm{Nat}\}$, $\Sigma_{\mathrm{d}} = \{+^{\mathrm{Nat}\times\mathrm{Nat}\to\mathrm{Nat}}\}$, $\Sigma_{\mathrm{c}} = \{\mathsf{s}^{\mathrm{Nat}\to\mathrm{Nat}}, \mathsf{0}^{\mathrm{Nat}}\}$ and

$$E = \left\{ \begin{array}{lcl} + \ \mathsf{0} \ y & \approx & y \\ + \ (\mathsf{s} \ x) \ y & \approx & \mathsf{s} \ (+ \ x \ y) \end{array} \right\}.$$

Then E is an elementary constructor-based $\langle T, \Sigma\rangle$-theory. Take $E' = E \cup \{\mathsf{g} \ F \ x \approx + \ (F \ x) \ x\}$. Then there exists no $t \in \mathrm{S}(\Sigma_{\mathrm{c}}, \{F\})$ such that $\mathsf{g} \ F \ \mathsf{0} \overset{\text{ext}*}{\leftrightarrow}_{E'} t$.

5 Checking Natural Inductive Theorems

In this section, we present partial answers to the following questions:

1. When can one prove or check an equational theory is constructor-based?
2. When can one prove or check an equation is a natural inductive theorem?

We first answer the second question by giving a sufficient condition for natural inductive theorems.

▶ **Lemma 5.1.** Let E be a natural $\langle T, \Sigma\rangle$-theory, X the set of variables over T and $\mathcal{A}$ a natural $\langle T, \Sigma\rangle$-term algebra for E. Then for any environment ρ for $\mathcal{A}$, there exists a substitution $\sigma_g : X^{\mathrm{b}} \to \mathrm{S}(\Sigma_{\mathrm{c}})$ such that $\rho = \rho{\restriction_{X^{\mathrm{f}}}}/\sigma$.

Proof. Take a substitution $\sigma = \{x \mapsto \rho(x) \mid x \in X^{\mathrm{b}}\}$. We now show $\rho = (\rho{\restriction_{X^{\mathrm{f}}}})/\sigma$. For $x \in X^{\mathrm{b}}$, we have $((\rho{\restriction_{X^{\mathrm{f}}}})/\sigma)(x) = [\![\sigma(x)]\!]_{\rho\restriction_{X^{\mathrm{f}}}} = [\![\rho(x)]\!]_{\rho\restriction_{X^{\mathrm{f}}}} = \rho(x)$. For $F \in X^{\mathrm{f}}$, we have $((\rho{\restriction_{X^{\mathrm{f}}}})/\sigma)(F) = [\![\sigma(F)]\!]_{\rho\restriction_{X^{\mathrm{f}}}} = [\![F]\!]_{\rho\restriction_{X^{\mathrm{f}}}} = \rho(F)$. ◀

▶ **Theorem 5.2** (sufficient condition for natural inductive theorem)**.** Let E be a natural $\langle T, \Sigma\rangle$-theory, X the set of variables over T and $s \approx t \in \mathrm{Eqn}(\Sigma, X)$. If $s\sigma \overset{\text{ext}*}{\leftrightarrow}_E t\sigma$ for any substitution $\sigma : X^{\mathrm{b}} \to \mathrm{S}(\Sigma_{\mathrm{c}})$, then $s \approx t$ is a natural inductive theorem of E.

Proof. Let $\mathcal{A}$ be a natural $\langle T, \Sigma\rangle$-term algebra for E and ρ an environment for $\mathcal{A}$. By Lemma 5.1, there exists $\sigma : X^{\mathrm{b}} \to \mathrm{S}(\Sigma_{\mathrm{c}})$ such that $\rho = \rho{\restriction_{X^{\mathrm{f}}}}/\sigma$. By Lemma 3.2, Then $[\![u\sigma]\!]_{\rho\restriction_{X^{\mathrm{f}}}} = [\![u]\!]_{\rho\restriction_{X^{\mathrm{f}}}/\sigma} = [\![u]\!]_\rho$ for any $u \in \mathrm{S}(\Sigma, X)$. Thus $[\![s]\!]_\rho = [\![t]\!]_\rho$ iff $[\![s\sigma]\!]_{\rho\restriction_{X^{\mathrm{f}}}} = [\![t\sigma]\!]_{\rho\restriction_{X^{\mathrm{f}}}}$. By our assumption, $s\sigma \overset{\text{ext}*}{\leftrightarrow}_E t\sigma$. By Proposition 3.5, $E \models_{\text{ext}} s\sigma \approx t\sigma$ holds. By our assumption, $\mathcal{A} \models E$. Thus, since $\mathcal{A}$ is an extensional $\langle T, \Sigma\rangle$-algebra by Lemma 4.4, $\mathcal{A} \models s\sigma \approx t\sigma$ holds. Hence for any environment ρ' for $\mathcal{A}$, $[\![s\sigma]\!]_{\rho'} = [\![t\sigma]\!]_{\rho'}$ and thus, in particular, $[\![s\sigma]\!]_{\rho\restriction_{X^{\mathrm{f}}}} = [\![t\sigma]\!]_{\rho\restriction_{X^{\mathrm{f}}}}$. This concludes $[\![s]\!]_\rho = [\![t]\!]_\rho$. ◀

We next answer the first question by giving a sufficient condition of equational theories (specified by STSRSs) to be constructor-based.

▶ **Definition 5.3** (simple S-expressions)**.** An S-expression s is said to be *simple* if for all $(u\ t_1 \cdots t_n) \trianglelefteq s$, (1) if $\mathrm{head}(u) \in V$ then $\mathrm{type}(t_i) \in B$ for $i = 1, \ldots, n$ and (2) if $\mathrm{head}(u) \in \Sigma_\mathrm{d}$ and $\mathrm{type}(t_i) \in B$ then $t_i \in \mathrm{S}(\Sigma)$ for $i = 1, \ldots, n$.

▶ **Lemma 5.4.** Let $f \in \Sigma_\mathrm{d}$. For any substitution $\theta : V^\mathrm{b} \to \mathrm{S}(\Sigma)$, $(f\uparrow)\theta$ is simple.

Proof. Let $(f\uparrow)\theta = ((\cdots(f\ t_{11}\cdots t_{1n_1})\cdots)\ t_{m1}\cdots t_{mn_m})$. By our assumption, if $\mathrm{type}(t_{ij}) \in B$ then $t_{ij} \in \mathrm{S}(\Sigma)$ and if $\mathrm{type}(t_{ij}) \notin B$ then $t_{ij} \in V^\mathrm{f}$. Thus for any $(u\ s_1 \cdots s_n) \trianglelefteq t_{ij} \in \mathrm{S}(\Sigma)$, the condition (1) holds since $\mathrm{head}(u) \notin V$ and the condition (2) holds since $s_1, \ldots, s_n \in \mathrm{S}(\Sigma)$. Furthermore, if $t_{ij} \in V^\mathrm{f}$ then $(u\ s_1 \cdots s_n) \trianglelefteq t_{ij}$ does not happen. Thus it remains to show the conditions (1) and (2) hold for $(u\ t_{k1} \cdots t_{kn_k})$ where $\mathrm{head}(u) = f$ and $1 \leq k \leq m$. The condition (1) holds since $f \notin V$. The condition (2) holds since $\mathrm{type}(t_{ki}) \in B$ implies $t_{ki} \in \mathrm{S}(\Sigma)$ for $i = 1, \ldots, n_k$. ◀

An S-expression s such that $s \to_\mathcal{R} t$ for no t is said to be *normal*; the set of normal S-expressions of $\mathcal{R}$ is denoted by $\mathrm{NF}(\mathcal{R})$. An STSRS $\mathcal{R}$ is said to be *higher-order quasi-reducible* (denoted by $\mathrm{HQR}(\mathcal{R})$) if $s \notin \mathrm{NF}(\mathcal{R})$ for any S-expression $s \in \mathrm{S}(\Sigma, V^\mathrm{f})^\mathrm{b}$ such that (i) $\mathrm{head}(s) \in \Sigma_\mathrm{d}$ and (ii) for any $u \in \mathrm{Args}(s)$, if $\mathrm{type}(u) \in B$ then $u \in \mathrm{S}(\Sigma_\mathrm{c})$ and otherwise $u \in V^\mathrm{f}$ [4].

▶ **Lemma 5.5.** Let $\mathcal{R}$ be a left-linear elementary STSRS such that $\mathrm{HQR}(\mathcal{R})$ hold. Let $s \in \mathrm{S}(\Sigma, V^\mathrm{f})^\mathrm{b} \cap \mathrm{NF}(\mathcal{R})$. If s is simple then $s \in \mathrm{S}(\Sigma_\mathrm{c}, V^\mathrm{f})$.

Proof. Take a minimal (w.r.t. subexpression relation $\trianglelefteq$) $s \in \mathrm{S}(\Sigma, V^\mathrm{f})^\mathrm{b} \cap \mathrm{NF}(\mathcal{R})$ such that s is simple and $s \notin \mathrm{S}(\Sigma_\mathrm{c}, V^\mathrm{f})$. Then there exists a subexpression u of s such that $\mathrm{head}(u) = f \in \Sigma_\mathrm{d}$. Take a maximal (w.r.t. subexpression relation $\trianglelefteq$) such subexpression u. We first claim that $\mathrm{type}(u) \in B$. Suppose $\mathrm{type}(u) \notin B$. Then since $\mathrm{type}(s) \in B$, there exists a subexpression $(u_0 \cdots u \cdots)$ of s. If $\mathrm{head}(u_0) \in V$ then, since s is simple, $\mathrm{type}(u) \in B$ and hence this contradicts our assumption. Otherwise by the maximality of u, $\mathrm{head}(u_0) \in \Sigma_\mathrm{c}$. Then, since $\mathcal{R}$ is elementary, it follows $\mathrm{type}(u) \in B$. Hence this also contradicts our assumption. Therefore $\mathrm{type}(u) \in B$. Let $u = ((\cdots(f\ t_{11}\cdots t_{1n_1})\cdots)\ t_{m1}\cdots t_{mn_m})$. Since s is simple, if $\mathrm{type}(t_{ij}) \in B$ then $t_{ij} \in \mathrm{S}(\Sigma)$; furthermore, by $t_{ij} \trianglelefteq s$, $t_{ij} \in \mathrm{NF}(\mathcal{R})$ and simple. Thus by the minimality of s, $\mathrm{type}(t_{ij}) \in B$ implies $t_{ij} \in \mathrm{S}(\Sigma_\mathrm{c})$. Then, since $\mathcal{R}$ is left-linear and higher-order quasi-reducible and $\mathrm{type}(u) \in B$, $u \notin \mathrm{NF}(\mathcal{R})$. This is a contradiction. ◀

▶ **Definition 5.6** (GAV/quasi-simple/simplicity-preserving)**.** **1.** The set $\mathrm{GAV}(s)$ of *ground-augmenting variables* of an S-expression s is defined like this: $\mathrm{GAV}(a) = \emptyset$ for $a \in \Sigma \cup V$; $\mathrm{GAV}((t_0\ t_1 \cdots t_n)) = (\bigcup\{\mathrm{GAV}(t_i) \mid 0 \leq i \leq n\}) \cup (\bigcup\{\mathrm{V}(t_i) \mid \mathrm{type}(t_i) \in B, 1 \leq i \leq n, \mathrm{head}(t_0) \in \Sigma_\mathrm{d}\})$.

2. An S-expression s is said to be *quasi-simple w.r.t. a set* X of variables if for any subexpression $(u\ t_1 \cdots t_n)$ of s, (1) if $\mathrm{head}(u) \in V$ then $t_i \in \mathrm{S}(\Sigma, X)^\mathrm{b}$, and (2) if $\mathrm{head}(u) \in \Sigma_\mathrm{d}$ and $\mathrm{type}(t_i) \in B$ then $t_i \in \mathrm{S}(\Sigma, X)$.

3. A rewrite rule $l \to r$ of type τ is said to be *simplicity-preserving* if (1) $\mathrm{head}(r) \in V$ implies τ is second-order, (2) $\mathrm{head}(r) \notin \Sigma \cup \mathrm{GAV}(l)$ implies $\tau \in B$ and (3) r is quasi-simple w.r.t. $\mathrm{GAV}(l)$. An STSRS $\mathcal{R}$ is simplicity-preserving if it consists of simplicity-preserving rewrite rules.

Let $l \to r$ be a rewrite rule. Suppose $\mathrm{head}(l)$ has the second-order type. Then $\mathrm{GAV}(l) = \mathrm{V}(l)$. Hence if moreover $\mathrm{V}(r) \subseteq V^\mathrm{b}$ then r is quasi-simple w.r.t. $\mathrm{GAV}(l)$. Therefore, if moreover $l \to r$ has a base type then $l \to r$ is simplicity-preserving.

▶ **Example 5.7.** Let $T = \{\mathrm{Nat}, \mathrm{Nat} \to \mathrm{Nat}, \mathrm{Nat} \times \mathrm{Nat} \to \mathrm{Nat}, \mathrm{List}, \mathrm{Nat} \times \mathrm{List} \to \mathrm{List}, \mathrm{List} \times \mathrm{List} \to \mathrm{List}, (\mathrm{Nat} \to \mathrm{Nat}) \times \mathrm{List} \to \mathrm{List}\}$, $\Sigma_c = \{0^{\mathrm{Nat}}, \mathsf{s}^{\mathrm{Nat}\to\mathrm{Nat}}, []^{\mathrm{List}}, :^{\mathrm{Nat}\times\mathrm{List}\to\mathrm{List}}\}$, $\Sigma_d = \{+^{\mathrm{Nat}\times\mathrm{Nat}\to\mathrm{Nat}}, \mathsf{app}^{\mathrm{List}\times\mathrm{List}\to\mathrm{List}}, \mathsf{map}^{(\mathrm{Nat}\to\mathrm{Nat})\times\mathrm{List}\to\mathrm{List}}\}$ and

$$\mathcal{R} = \left\{ \begin{array}{lllll} (1) & + \ 0 \ y & \to & y \\ (2) & + \ (\mathsf{s} \ x) \ y & \to & \mathsf{s} \ (+ \ x \ y) \\ (3) & \mathsf{app} \ [] \ ys & \to & ys \\ (4) & \mathsf{app} \ (: \ x \ xs) \ ys & \to & : \ x \ (\mathsf{app} \ xs \ ys) \\ (5) & \mathsf{map} \ F \ [] & \to & [] \\ (6) & \mathsf{map} \ F \ (: \ x \ xs) & \to & : \ (F \ x) \ (\mathsf{map} \ F \ xs) \end{array} \right\}.$$

By the remark above, rules (1)–(4) are simplicity-preserving. Let $l = \mathsf{map}\ F\ []$ and $r = []$. Then $\mathrm{GAV}(l) = \emptyset$. r is quasi-simple because there is no subexpression of the form $(u\ t_1 \cdots t_n)$. Hence, since $l \to r$ has a base type, $l \to r$ is simplicity-preserving. Let $l = \mathsf{map}\ F\ (:\ x\ xs)$ and $r = :\ (F\ x)\ (\mathsf{map}\ F\ xs)$. The set $\mathrm{GAV}(l) = \{x, xs\}$. Let $X = \{x, xs\}$. For $(u\ t_1) = (F\ x)$, we have $t_1 = x \in \mathrm{S}(\Sigma, X)^{\mathrm{b}}$ and for $(u\ t_1\ t_2) = (\mathsf{map}\ F\ xs)$, we have $t_2 = xs \in \mathrm{S}(\Sigma, X)^{\mathrm{b}}$. Since $l \to r$ has a base type and r is quasi-simple w.r.t. $\mathrm{GAV}(l)$, $l \to r$ is simplicity-preserving.

▶ **Lemma 5.8.** Let s be an S-expression and σ a substitution. If $s\sigma$ is simple then $\theta(x) \in \mathrm{S}(\Sigma)$ for any $x \in \mathrm{GAV}(s)$.

Proof. By induction on s. ◀

▶ **Lemma 5.9.** Let $\mathcal{R}$ be a simplicity-preserving STSRS. If s is simple and $s \to_{\mathcal{R}} t$ then t is simple.

Proof. Suppose $s = C[l\sigma]$, $t = C[r\sigma]$ and $l \to r$ is simplicity-preserving. We first show $r'\sigma$ is simple for any $r' \trianglelefteq r$, by induction on r'. The case of $r' \in \Sigma \cup V$ follows easily. Let $r' = (r_0\ r_1 \cdots r_n)$.

(1) Suppose $\mathrm{head}(r_0\sigma) \in V$. Then $\mathrm{head}(r_0) \in V$. Then $\mathrm{type}(r_i) \in B$ for $1 \le i \le n$ by quasi-simplicity of r. Thus $\mathrm{type}(r_i\sigma) \in B$ for $1 \le i \le n$.

(2) Suppose $\mathrm{head}(r_0\sigma) \in \Sigma_{\mathrm{d}}$. We distinguish two cases. Case $\mathrm{head}(r_0) = \mathrm{head}(r_0\sigma)$. By the quasi-simplicity of r w.r.t. $\mathrm{GAV}(l)$, $\mathrm{type}(r_i) \in B$ implies $r_i \in \mathrm{S}(\Sigma, \mathrm{GAV}(l))$ $(1 \le i \le n)$. Since $l\sigma$ is simple, $\sigma(x) \in \mathrm{S}(\Sigma)$ for any $x \in \mathrm{GAV}(l)$ by Lemma 5.8. Hence $\mathrm{type}(r_i\sigma) \in B$ implies $r_i\sigma \in \mathrm{S}(\Sigma)$. Case $\mathrm{head}(r_0) \in V$. Then, by the quasi-simplicity of r w.r.t. $\mathrm{GAV}(l)$, $r_i \in \mathrm{S}(\Sigma, \mathrm{GAV}(l))^{\mathrm{b}}$ for $1 \le i \le n$. Again, by Lemma 5.8, it follows that $r_i\sigma \in \mathrm{S}(\Sigma)$.

Hence we conclude that $r\sigma$ is simple. Next, we show $C'[r\sigma]$ is simple by induction on $C' \trianglelefteq C$ (B.S.) follows from the fact that $r\sigma$ is simple. To show (I.S.), we distinguish two cases.

- Case $C' = (C''\ w_1 \cdots w_n)$.
  (1) Suppose $\mathrm{head}(C''[r\sigma]) \in V$. Case of $\mathrm{head}(C''[l\sigma]) \in V$ is trivial. Suppose $\mathrm{head}(C''[l\sigma]) \notin V$. Then $\mathrm{head}(C''[r\sigma]) = \mathrm{head}(r\sigma)$ and hence $\mathrm{head}(r) \in V$. Thus, $\mathrm{type}(l\sigma)$ is second-order and hence $\mathrm{type}(w_i) \in B$ for all $1 \le i \le n$.
  (2) Suppose $\mathrm{head}(C''[r\sigma]) \in \Sigma_{\mathrm{d}}$. If $\mathrm{head}(C''[r\sigma]) = \mathrm{head}(C'')$, then $\mathrm{head}(C''[l\sigma]) \in \Sigma_{\mathrm{d}}$ and hence $\mathrm{type}(w_i) \in B$ implies $w_i \in \mathrm{S}(\Sigma)$. If $\mathrm{head}(C''[r\sigma]) = \mathrm{head}(r\sigma)$, then $\mathrm{head}(C''[l\sigma]) \in \Sigma_{\mathrm{d}}$. Thus $\mathrm{type}(w_i) \in B$ implies $w_i \in \mathrm{S}(\Sigma)$.
- Case $C' = (w_0 \cdots C'' \cdots)$. If $\mathrm{head}(w_0) \in V$ then $\mathrm{type}(C''[l\sigma]) \in B$ and hence $\mathrm{type}(C''[r\sigma]) \in B$. If $\mathrm{head}(w_0) \in \Sigma_{\mathrm{d}}$ and $\mathrm{type}(C''[r\sigma]) \in B$ then $C''[l\sigma] \in \mathrm{S}(\Sigma)$ and hence $C''[r\sigma] \in \mathrm{S}(\Sigma)$.

◀

An STSRS $\mathcal{R}$ is *weakly normalizing* (denoted by $\mathrm{WN}(\mathcal{R})$) if for any S-expression s there exists an S-expression $t \in \mathrm{NF}(\mathcal{R})$ such that $s \to_{\mathcal{R}}^* t$.

▶ **Lemma 5.10.** Let E be an elementary $\langle T, \Sigma\rangle$-theory and $\mathcal{R}$ be a left-linear simplicity-preserving STSRS on the same signature satisfying $\to_{\mathcal{R}}^* \subseteq \overset{\text{ext}*}{\leftrightarrow}_E$, $\mathrm{HQR}(\mathcal{R})$ and $\mathrm{WN}(\mathcal{R})$. Then E is constructor-based.

Proof. Let $f \in \Sigma_{\mathrm{d}}$ and σ_g be a substitution such that $\sigma_g : V^{\mathrm{b}} \to \mathrm{S}(\Sigma_{\mathrm{c}})$. By $\mathrm{WN}(\mathcal{R})$, there exists $w \in \mathrm{NF}(\mathcal{R})$ such that $(f\uparrow)\sigma_g \to_{\mathcal{R}}^* w$. Furthermore, since $(f\uparrow)\sigma_g \in \mathrm{S}(\Sigma, V^{\mathrm{f}})$, $w \in \mathrm{S}(\Sigma, V^{\mathrm{f}})$. By Lemma 5.4, $(f\uparrow)\sigma_g$ is simple. Since $\mathcal{R}$ is simplicity-preserving w is simple by Lemma 5.9. Thus by Lemma 5.5 and our assumption that $\mathcal{R}$ is left-linear and elementary and that $\mathrm{HQR}(\mathcal{R})$ holds, $w \in \mathrm{S}(\Sigma_{\mathrm{c}}, V^{\mathrm{f}})$. By $\to_{\mathcal{R}}^* \subseteq \overset{\text{ext}*}{\leftrightarrow}_E$, it follows that for any $f \in \Sigma_{\mathrm{d}}$ and substitution $\sigma_g : V^{\mathrm{b}} \to \mathrm{S}(\Sigma_{\mathrm{c}})$, there exists $w \in \mathrm{S}(\Sigma_{\mathrm{c}}, V^{\mathrm{f}})$ such that $(f\uparrow)\sigma_g \overset{\text{ext}*}{\leftrightarrow}_E w$. Thus E is constructor-based. ◀

▶ **Theorem 5.11** (checking natural inductive theorems)**.** Let $\mathcal{R}$ be a left-linear elementary natural simplicity-preserving STSRS such that $\mathrm{WN}(\mathcal{R})$ and $\mathrm{HQR}(\mathcal{R})$ hold. If $s\theta \overset{\text{ext}*}{\leftrightarrow}_{\mathcal{R}} t\theta$ for any substitution $\theta : V^{\mathrm{b}} \to \mathrm{S}(\Sigma_{\mathrm{c}})$, then $s \approx t$ is a natural inductive theorem of $\mathcal{R}$.

Proof. By Lemma 5.10, $\mathcal{R}$ is constructor-based. Then by Theorem 5.2 $s \approx t$ is a natural inductive theorem of $\mathcal{R}$. ◀

▶ **Example 5.12.** Let $\mathcal{R}$ be the STSRS given in Example 5.7. Then $\mathcal{R}$ is left-linear, elementary, simplicity-preserving and $\mathrm{WN}(\mathcal{R})$ and $\mathrm{HQR}(\mathcal{R})$ hold. To show that $\mathcal{R}$ is natural, we now show that there exists a natural $\langle \mathrm{T}, \Sigma\rangle$-term algebra $\mathcal{A}$ for $\mathcal{R}$. Let $\mathsf{0}^{\mathcal{A}} = \mathsf{0}$, $\mathsf{s}^{\mathcal{A}}(x) = (\mathsf{s}\ x)$, $[\,]^{\mathcal{A}} = [\,]$, $:^{\mathcal{A}}(x, xs) = (:\ x\ xs)$, $+^{\mathcal{A}}(x, y)$ be the unique normal form of $(+\ x\ y)$, $\mathsf{app}^{\mathcal{A}}(xs, ys)$ be the unique normal form of $(\mathsf{app}\ xs\ ys)$ and $\mathsf{map}^{\mathcal{A}}(f, xs)$ be defined inductively as: $\mathsf{map}^{\mathcal{A}}(f, [\,]) = [\,]$; $\mathsf{map}^{\mathcal{A}}(f, :\ x\ xs) = (:\ f(x)\ \mathsf{map}^{\mathcal{A}}(f, xs))$. Then we have $[\![l]\!]_\rho = [\![r]\!]_\rho$ for any $l \to r \in \mathcal{R}$. Furthermore, one easily shows $s \overset{\text{ext}*}{\leftrightarrow}_{\mathcal{R}} [\![s]\!]$ for any $s \in \mathrm{S}(\Sigma)^{\mathrm{Nat}}$ by induction on s. Using this, it also follows that $s \overset{\text{ext}*}{\leftrightarrow}_{\mathcal{R}} [\![s]\!]$ for any $s \in \mathrm{S}(\Sigma)^{\mathrm{List}}$ by induction on s. Thus $\mathcal{A}$ is a natural $\langle \mathrm{T}, \Sigma\rangle$-term algebra for $\mathcal{R}$. Let

$$l \approx r = \mathsf{map}\ F\ (\mathsf{app}\ xs\ ys) \approx \mathsf{app}\ (\mathsf{map}\ F\ xs)\ (\mathsf{map}\ F\ ys).$$

Then for any substitution $\theta : V^{\mathrm{b}} \to \mathrm{S}(\Sigma_{\mathrm{c}})$, $l\theta \overset{\text{ext}*}{\leftrightarrow}_{\mathcal{R}} r\theta$ holds. Thus, by Theorem 5.11, $l \approx r$ is a natural inductive theorem of $\mathcal{R}$.

6 Conclusion

Extensibility of inductive theorems is indispensable to extend the framework of program transformation by templates based on first-order term rewriting [7, 8, 9] to the higher-order setting. We have studied a new notion of inductive theorems for higher-order rewriting, *natural inductive theorems*, to incorporate properties such as extensionality and extensibility. The class of this theorems is placed between extensional theorems and extensional inductive theorems. We also have given sufficient conditions for natural inductive theorems which enables us to prove simply typed equations to be natural inductive theorems.

Acknowledgments

Thanks are due to anonymous referees for detailed comments. This work was partially supported by a grant from JSPS No. 20500002.

References

1 T. Aoto and T. Yamada. Termination of simply typed term rewriting systems by translation and labelling. In *Proc. of RTA 2003*, volume 2706 of *LNCS*, pages 380–394. Springer-Verlag, 2003.

2 T. Aoto and T. Yamada. Dependency pairs for simply typed term rewriting. In *Proc. of RTA 2005*, volume 3467 of *LNCS*, pages 120–134. Springer-Verlag, 2005.

3 T. Aoto and T. Yamada. Argument filterings and usable rules for simply typed dependency pairs. In *Proc. of FroCoS 2009*, volume 5749 of *LNAI*, pages 117–132. Springer-Verlag, 2009.

4 T. Aoto, T. Yamada, and Y. Toyama. Inductive theorems for higher-order rewriting. In *Proc. of RTA 2004*, volume 3091 of *LNCS*, pages 269–284. Springer-Verlag, 2004.

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

6 R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, 1977.

7 Y. Chiba, T. Aoto, and Y. Toyama. Program transformation by templates based on term rewriting. In *Proc. of PPDP 2005*, pages 59–69. ACM Press, 2005.

8 Y. Chiba, T. Aoto, and Y. Toyama. Program transformation by templates: A rewriting framework. *IPSJ Trans. on Programming*, 47(SIG 16 (PRO 31)):52–65, 2006.

9 Y. Chiba, T. Aoto, and Y. Toyama. Program transformation templates for tupling based on term rewriting. *IEICE Trans. on Inf. & Sys.*, E93-D(5):963–973, 2010.

10 W. N. Chin. Towards an automated tupling strategy. In *Proc. of PEPM'93*, pages 119–132. ACM Press, 1993.

11 A. Gill, J. Launchbury, and S. Peyton-Jones. A short cut to deforestation. In *Proc. of FPCA'93*, pages 223–232. ACM Press, 1993.

12 G. Huet and J.-M. Hullot. Proof by induction in equational theories with constructors. *Journal of Computer and System Sciences*, 25(2):239–266, 1982.

13 G. Huet and B. Lang. Proving and applying program transformations expressed with second order patterns. *Acta Informatica*, 11:31–55, 1978.

14 J.-P. Jouannaud and E. Kounalis. Automatic proofs by induction in theories without constructors. *Information and Computation*, 82:1–33, 1989.

15 D. Kapur, P. Narendran, and H. Zhang. Automating inductionless induction using test sets. *Journal of Symbolic Computation*, 11(1–2):81–111, 1991.

16 K. Kusakari, M. Sakai, and T. Sakabe. Primitive inductive theorems bridge implicit induction methods and inductive theorems in higher-order rewriting. *IEICE Trans. on Inf. & Sys.*, E88–D(12):2715–2726, 2005.

17 K. Meinke. Universal algebra in higher types. *Theoretical Computer Science*, 100:385–417, 1992.

18 K. Meinke. Proof theory of higher-order equations: conservativity, normal forms and term rewriting. *Journal of Computer and System Sciences*, 67:127–173, 2003.

19 Y. Toyama. How to prove equivalence of term rewriting systems without induction. *Theoretical Computer Science*, 90(2):369–390, 1991.

20 P. Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.

21 T. Yamada. Confluence and termination of simply typed term rewriting systems. In *Proc. of RTA 2001*, volume 2051 of *LNCS*, pages 338–352. Springer-Verlag, 2001.

A Path Order for Rewrite Systems that Compute Exponential Time Functions*

Martin Avanzini[1], Naohi Eguchi[2], and Georg Moser[1]

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

2 **School of Information Science,**
Japan Advanced Institute of Science and Technology, Japan
n-eguchi@jaist.ac.jp

Abstract

In this paper we present a new path order for rewrite systems, the *exponential path order* EPO*. Suppose a term rewrite system is compatible with EPO*, then the runtime complexity of this rewrite system is bounded from above by an exponential function. Furthermore, the class of function computed by a rewrite system compatible with EPO* equals the class of functions computable in exponential time on a Turing machine.

1998 ACM Subject Classification F.2.2, F.4.1, F.4.2, D.2.4, D.2.8

Keywords and phrases Runtime Complexity, Exponential Time Functions, Implicit Computational Complexity

Digital Object Identifier 10.4230/LIPIcs.RTA.2011.123

Category Regular Research Paper

1 Introduction

In this paper we are concerned with the complexity analysis of term rewrite systems (TRSs) and the ramifications of such an analysis in implicit computational complexity (ICC for short).

Several notions to assess the complexity of a terminating term rewrite system (TRS) have been proposed in the literature, compare [12, 19, 13, 18]. The conceptually simplest one was suggested by Hofbauer and Lautemann in [19]: the complexity of a given TRS is measured as the maximal length of derivation sequences. More precisely, the *derivational complexity function* with respect to a terminating TRS $\mathcal{R}$ relates the maximal derivation height to the size of the initial term. A more fine-grained approach is introduced in [12] (compare also [18]), where the derivational complexity function is refined so that in principle only argument normalised (aka basic) terms are considered. This notion, in the following referred to as the *runtime complexity* of TRSs, aims at capturing the complexity of the *functions computed* by the analysed TRS (see [13]).

* The first and third author are partly supported by FWF (Austrian Science Fund) projects P20133-N15. The second author is supported in part by the JSPS Institutional Program for Young Researcher Overseas Visits.

22nd International Conference on Rewriting Techniques and Applications (RTA'11).
Editor: M. Schmidt-Schauß; pp. 123–138

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

In recent years the field of complexity analysis of rewrite systems matured and some advances towards an automated complexity analysis of TRSs evolved (see [23] for an overview). The current focus of modern complexity analysis of rewrite systems is on techniques that yield *polynomial* derivational or runtime complexity. In this paper we study a complementary view. We establish following results:

- We present a new path order for rewrite systems, the *exponential path order* EPO*. Suppose a TRS $\mathcal{R}$ is compatible with EPO*. Then the runtime complexity of $\mathcal{R}$ is at most exponential.
- EPO* is sound, that is, any function computed by a TRS compatible with EPO* is computable on a Turing machine in exponential time.
- EPO* is complete, that is, any function computable in exponential time can be computed by a TRS that is compatible with EPO*.

Note that the first and second result relate two different notions of the complexity of a TRS $\mathcal{R}$: the *runtime complexity* with respect to $\mathcal{R}$ and the complexity of the function *computed* with $\mathcal{R}$. Furthermore, we have implemented the order EPO* so that our research yields a fully automatic complexity tool for exponential time functions. Our research is motivated by earlier successful order-theoretic characterisations of complexity classes. We mention the *light multiset path order* introduced by Marion [22]. Roughly speaking the light multiset path order is a tamed version of the multiset path order, characterising the functions computable in polytime (compare also [4]). In a similar spirt the here presented path order EPO* characterises the functions computable in exptime.

The definition of EPO* makes use of *tiering* [8] and is strongly influenced by a recursion theoretic characterisation $\mathcal{N}$ of the class of functions computable in exponential time by Arai and the second author (see [1]) and a very recent term-rewriting characterisation of $\mathcal{N}$ by the second author (see [15]). We motivate our study through the following example.

▶ **Example 1.1.** Consider the following TRS $\mathcal{R}_{\mathsf{fib}}$ which is easily seen to represent the computation of the n^{th} Fibonacci number.

$$\mathsf{fib}(x) \to \mathsf{dfib}(x, \mathsf{0}) \qquad \mathsf{dfib}(\mathsf{0}, y) \to \mathsf{s}(y)$$
$$\mathsf{dfib}(\mathsf{s}(\mathsf{0}), y) \to \mathsf{s}(y) \qquad \mathsf{dfib}(\mathsf{s}(\mathsf{s}(x)), y) \to \mathsf{dfib}(\mathsf{s}(x), \mathsf{dfib}(x, y))$$

Then all rules in the TRS $\mathcal{R}_{\mathsf{fib}}$ can be oriented with EPO*, which allows us to (automatically) deduce that the runtime complexity of this system is exponential. Using the machinery of [5], exploiting graph rewriting, we can even show that any function computed by a TRS compatible with EPO* is computable in exponential time on a Turing machine. Conversely we show that any function f that can be computed in exponential time on a Turing machine can be computed by a TRS $\mathcal{R}(f)$ such that $\mathcal{R}(f)$ is compatible with EPO*. In total, we obtain an alternative, syntactic characterisation of the exponential time functions.

Related Work. With respect to rewriting we mention [16], where it is shown that *matrix interpretations* yield exponential derivational complexity, hence at most exponential runtime complexity. Our work is also directly related to work in ICC (see [7] for an overview). We want to mention [10, 21], were alternative characterisations of the class of functions computable in exponential time are given. For less directly related work we cite [9], where a complete characterisation of (imperative) programs that admit linear and polynomial runtime complexity is established. As these characterisations are decidable, we obtain a decision procedure for programs that admit a runtime complexity that is at most exponential.

The remainder of the paper is organised as follows. In Section 2 we recall definitions. The order EPO* is presented in Section 3. In Section 4 we introduce an intermediate

order EPO critical for establishing our soundness result, and in Section 5 we prove that EPO* induces exponentially bounded runtime complexity. In Section 6 we present the aforementioned soundness and completeness result. Finally, we conclude in Section 7. Due to space limitations we omit some proofs of auxiliary lemmas. Missing proofs are available in a separate technical report [3].

2 Preliminaries

We assume familiarity with the basics of term rewriting, see [6, 25] and briefly review definitions and notations used. Let $\mathcal{V}$ denote a countably infinite set of variables and let $\mathcal{F}$ be a finite signature. The set of terms over $\mathcal{F}$ and $\mathcal{V}$ is written as $\mathcal{T}(\mathcal{F},\mathcal{V})$. We denote by $\vec{s},\vec{t},\dots$ sequences of terms, and for a set of terms T we write $\vec{t}\subseteq T$ to indicate that for each t_i appearing in $\vec{t}$, $t_i\in T$. We suppose that the signature $\mathcal{F}$ is partitioned into *defined symbols* $\mathcal{D}$ and *constructors* $\mathcal{C}$. The set of *basic* terms $\mathcal{B}\subseteq\mathcal{T}(\mathcal{F},\mathcal{V})$ is defined as $\mathcal{B}:=\{f(t_1,\dots,t_n)\mid f\in\mathcal{D}\text{ and }t_i\in\mathcal{T}(\mathcal{C},\mathcal{V})\text{ for }i\in\{1,\dots,n\}\}$.

We write $\trianglelefteq$ and $\trianglerighteq$ to denote the *subterm* and *superterm* relation, the strict part of $\trianglelefteq$ (respectively $\trianglerighteq$) is denoted by $\triangleleft$ (respectively $\triangleright$). We denote by $|t|$ and $\mathsf{dp}(t)$ the *size* and *depth* of the term t. The *root symbol* (denoted as $\mathsf{rt}(t)$) of a term t is either t itself, if $t\in\mathcal{V}$, or the symbol f, if $t=f(t_1,\dots,t_n)$.

A *preorder* is a reflexive and transitive binary relation. If $\gtrsim$ is a preorder, we write $\sim\;:=\;\gtrsim\cap\precsim$ and $>\;:=\;\gtrsim\setminus\sim$ to denote the *equivalence* and *strict part* of $\gtrsim$ respectively. A *quasi-precedence* (or simply *precedence*) is a preorder $\gtrsim\;=\;>\uplus\sim$ on the signature $\mathcal{F}$ so that the strict part $>$ is well-founded. We lift the equivalence $\sim$ induced by the precedence $\gtrsim$ to terms in the obvious way: $s\sim t$ if and only if (i) $s=t$, or (ii) $s=f(s_1,\dots,s_n)$, $t=g(t_1,\dots,t_n)$, $f\sim g$ and $s_i\sim t_i$ for all $i\in\{1,\dots,n\}$. The precedence $\gtrsim$ induces a *rank* $\mathsf{rk}(f)$ for any $f\in\mathcal{F}$ as follows: $\mathsf{rk}(f):=\max\{1+\mathsf{rk}(g)\mid g\in\mathcal{F}\text{ and }f>g\}$.

Let $\mathcal{R}$ be a TRS over $\mathcal{F}$. We write $\to_\mathcal{R}$ for the induced rewrite relation. A term $s\in\mathcal{T}(\mathcal{F},\mathcal{V})$ is called a *normal form* if there is no $t\in\mathcal{T}(\mathcal{F},\mathcal{V})$ such that $s\to_\mathcal{R}t$. We use $\mathrm{NF}(\mathcal{R})$ to denote the set of normal-forms of $\mathcal{R}$. With $\xrightarrow{\mathsf{i}}_\mathcal{R}$ we denote the *innermost rewrite relation*. We write $s\to^!_\mathcal{R}t$ (respectively $s\xrightarrow{\mathsf{i}}^!_\mathcal{R}t$) if $s\to^*_\mathcal{R}t$ (respectively $s\xrightarrow{\mathsf{i}}^*_\mathcal{R}t$) and $t\in\mathrm{NF}(\mathcal{R})$. A TRS is a *constructor* TRS if left-hand sides are basic terms and it is *completely defined* if each defined symbol is completely defined. Here a symbol is completely defined if it does not occur in any normal form. A TRS $\mathcal{R}$ is called *terminating* if $\to_\mathcal{R}$ is well-founded, $\mathcal{R}$ is *confluent* if for all terms s,t_1,t_2 with $s\to^*_\mathcal{R}t_1$ and $s\to^*_\mathcal{R}t_2$, there exists u such that $t_1\to^*_\mathcal{R}u$ and $t_2\to^*_\mathcal{R}u$.

Let $\to$ be a finitely branching, well-founded binary relation on terms. The *derivation height* of a term t with respect to $\to$ is given by $\mathrm{dh}(t,\to):=\max\{n\mid\exists u.\,t\to^n u\}$. Here $\to^n$ denotes the n-fold application of $\to$. The (innermost) *runtime complexity* of a terminating TRS $\mathcal{R}$ is defined as $\mathrm{rc}^{(\mathsf{i})}_\mathcal{R}(n):=\max\{\mathrm{dh}(t,\to)\mid t\in\mathcal{B}\text{ and }|t|\leqslant n\}$, where $\to$ denotes $\to_\mathcal{R}$ or $\xrightarrow{\mathsf{i}}_\mathcal{R}$ respectively. We say the (innermost) runtime complexity is exponential to assert the existence of an exponential function that binds $\mathrm{rc}^{(\mathsf{i})}_\mathcal{R}$ from above.

Furthermore, we assume (at least nodding) acquaintance with complexity theory, compare [20]. We write $\mathbb{N}$ for the set of *natural numbers*. Let M be a *Turing machine* (TM for short) with alphabet Σ, and let $w\in\Sigma^*$. We say that M computes $v\in\Sigma^*$ on input w, if M accepts w and v is written on a dedicated output tape. Note that when M is nondeterministic, then v computed on input w may not be unique. We say that M computes a binary relation $R\subseteq\Sigma^*\times\Sigma^*$ if for all $w,v\in\Sigma^*$ with $w\,R\,v$, M computes v on input w. Note that if M is deterministic then R induces a partial function $f_R:\Sigma^*\to\Sigma^*$. In this case we say

that M computes the function f_R. Let $S : \mathbb{N} \to \mathbb{N}$ denote a bounding function. We denote by $\mathsf{FTIME}(S(n))$ the class of functions computable by some TM M in time $S(n)$. Then $\mathsf{FEXP} := \bigcup_{k\in\mathbb{N}} \mathsf{FTIME}(2^{\mathsf{O}(n^k)})$ denotes the class of *exponential-time computable functions*.

3 Exponential Path Order EPO*

In this section we present the *exponential path order* (EPO* for short). Throughout the following, we fix $\gtrsim$ to denote an *admissible* quasi-precedence on $\mathcal{F}$. Here a precedence is called admissible if constructors are minimal, i.e., for all defined symbols f we have $f > c$ for all constructors c.

In addition to the precedence $\gtrsim$, an instance of EPO* is induced by a *safe mapping* $\mathsf{safe} : \mathcal{F} \to 2^{\mathbb{N}}$. This mapping associates with every n-ary function symbol f the set of *safe argument positions* $\{i_1, \dots, i_m\} \subseteq \{1, \dots, n\}$. Argument positions included in $\mathsf{safe}(f)$ are called *safe*, those not included are called *normal* and collected in $\mathsf{nrm}(f)$. For n-ary constructors c we require that all argument positions are safe, i.e., $\mathsf{safe}(c) = \{1, \dots, n\}$. To simplify the presentation, we write $f(t_{i_1}, \dots, t_{i_k}; t_{j_1}, \dots, t_{j_l})$ for the term $f(t_1, \dots, t_n)$ with $\mathsf{nrm}(f) = \{i_1, \dots, i_k\}$ and $\mathsf{safe}(f) = \{j_1, \dots, j_l\}$.

We restrict term equivalence $\sim$ in the definition of $\stackrel{s}{\sim}$ below so that the separation of arguments through safe is taken into account: We define $s \stackrel{s}{\sim} t$ if either (i) $s = t$, or (ii) $s = f(s_1, \dots, s_l\,;\, s_{l+1}, \dots, s_{l+m})$, $t = g(t_1, \dots, t_l\,;\, t_{l+1}, \dots, t_{l+m})$ where $f \sim g$ and $s_i \stackrel{s}{\sim} t_i$ for all $i \in \{1, \dots, l\}$. The definition of an instance $>_{\mathsf{epo}\star}$ of EPO* is split into the following two definitions.

▶ **Definition 3.1.** Let $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ such that $s = f(s_1, \dots, s_l\,;\, s_{l+1}, \dots, s_{l+m})$. Then $s \sqsupset_{\mathsf{epo}\star} t$ if $s_i \sqsupseteq_{\mathsf{epo}\star} t$ for some $i \in \{1, \dots, l+m\}$. Further, if $f \in \mathcal{D}$, then $i \in \mathsf{nrm}(f)$. Here we set $\sqsupseteq_{\mathsf{epo}\star} := \sqsupset_{\mathsf{epo}\star} \cup \stackrel{s}{\sim}$.

▶ **Definition 3.2.** Let $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ such that $s = f(s_1, \dots, s_l\,;\, s_{l+1}, \dots, s_{l+m})$. Then $s >_{\mathsf{epo}\star} t$ with respect to the admissible precedence $\gtrsim$ and safe mapping safe if either

1) $s_i \geqslant_{\mathsf{epo}\star} t$ for some $i \in \{1, \dots, l+m\}$, or
2) $t = g(t_1, \dots, t_k\,;\, t_{k+1}, \dots, t_{k+n})$, $f > g$ and
   i) $s \sqsupset_{\mathsf{epo}\star} t_1, \dots, s \sqsupset_{\mathsf{epo}\star} t_k$, and
   ii) $s >_{\mathsf{epo}\star} t_{k+1}, \dots, s >_{\mathsf{epo}\star} t_{k+n}$, or
3) $t = g(t_1, \dots, t_k\,;\, t_{k+1}, \dots, t_{k+n})$, $f \sim g$ and for some $i \in \{1, \dots, \min(l, k)\}$
   i) $s_1 \stackrel{s}{\sim} t_1, \dots, s_{i-1} \stackrel{s}{\sim} t_{i-1}$, $s_i \sqsupset_{\mathsf{epo}\star} t_i$, $s \sqsupset_{\mathsf{epo}\star} t_{i+1}, \dots, s \sqsupset_{\mathsf{epo}\star} t_k$, and
   ii) $s >_{\mathsf{epo}\star} t_{k+1}, \dots, s >_{\mathsf{epo}\star} t_{k+n}$.

Here we set $\geqslant_{\mathsf{epo}\star} := >_{\mathsf{epo}\star} \cup \stackrel{s}{\sim}$.

We write $\rhd\!\!\!/_{\sim}$ for the *superterm relation modulo term equivalence* $\sim$, defined as follows: $f(s_1, \dots, s_n) \rhd\!\!\!/_{\sim} t$ if $s_i \rhd\!\!\!/_{\sim} t$ or $s_i \sim t$ for some $i \in \{1, \dots, m\}$. Further, we set $\trianglerighteq\!\!\!/_{\sim} := \rhd\!\!\!/_{\sim} \cup \sim$. As immediate consequence of the definitions we obtain the following lemma.

▶ **Lemma 3.3.** *The inclusions* $\sqsupset_{\mathsf{epo}\star} \subseteq \rhd\!\!\!/_{\sim} \subseteq >_{\mathsf{epo}\star}$ *hold and further, if* $s \in \mathcal{T}(\mathcal{C}, \mathcal{V})$ *and* $s >_{\mathsf{epo}\star} t$ *then* $t \in \mathcal{T}(\mathcal{C}, \mathcal{V})$.

Note that the last property holds due to the restrictions imposed on precedence and safe mapping. The central theorem of this section states that EPO* induces exponential innermost runtime complexity.

▶ **Theorem 3.4.** *Suppose* $\mathcal{R}$ *is a constructor TRS compatible with* $>_{\mathsf{epo}\star}$*, i.e.,* $\mathcal{R} \subseteq >_{\mathsf{epo}\star}$*. Then the innermost runtime complexity* $\mathrm{rc}^{\mathsf{i}}_{\mathcal{R}}(n)$ *is bounded by an exponential* $2^{\mathsf{O}(n^k)}$ *for some fixed* $k \in \mathbb{N}$.

The proof of this theorem needs further preparation: We introduce in Section 4 an auxiliary order EPO, akin to the order presented in [1]. Although this auxiliary order is admittedly technical, it is easier to reason about its induced complexity. In Section 5 we then use this order to measure the derivation height of terms with respect to $\mathcal{R} \subseteq >_{\mathsf{epo}\star}$, proving Theorem 3.4.

▶ **Example 3.5.** [Example 1.1 continued]. Let safe be the safe mapping such that $\mathsf{safe}(\mathsf{fib}) = \varnothing$ and $\mathsf{safe}(\mathsf{dfib}) = \{2\}$. Further, let $\gtrsim$ be the admissible precedence with $\mathsf{fib} > \mathsf{dfib} > \mathsf{s} \sim \mathsf{0}$. It is easy to verify that $\mathcal{R}_{\mathsf{fib}} \subseteq >_{\mathsf{epo}\star}$ for the induced order $>_{\mathsf{epo}\star}$. By Theorem 3.4 we conclude that the innermost runtime complexity of $\mathcal{R}_{\mathsf{fib}}$ is exponentially bounded.

We emphasise that Theorem 3.4 *does not* hold for full rewriting.

▶ **Example 3.6.** Consider the TRS $\mathcal{R}_{\mathsf{d}}$ consisting of the rules

$$\mathsf{d}(;x) \to \mathsf{c}(;x,x) \qquad \mathsf{f}(0;y) \to y \qquad \mathsf{f}(\mathsf{s}(;x);y) \to \mathsf{f}(x;\mathsf{d}(;\mathsf{f}(x;y)))\ .$$

Then $\mathcal{R}_{\mathsf{d}} \subseteq >_{\mathsf{epo}\star}$ for the precedence $\mathsf{f} > \mathsf{d} > \mathsf{c}$ and safe mapping as indicated in the definition of $\mathcal{R}_{\mathsf{d}}$. Theorem 3.4 proves that the innermost runtime complexity of $\mathcal{R}_{\mathsf{d}}$ is exponentially bounded.

On the other hand, the runtime complexity of $\mathcal{R}_{\mathsf{d}}$ (with respect to full rewriting) grows strictly faster than any exponential: Consider for arbitrary $t \in \mathcal{T}(\mathcal{F},\mathcal{V})$ the term $\mathsf{f}(\mathsf{s}^n(0),t)$. We verify, for $n > 0$, $\mathrm{dh}(\mathsf{f}(\mathsf{s}^n(0),t),\to_{\mathcal{R}}) \geqslant 2^{2^{n-1}} \cdot (1 + \mathrm{dh}(t,\to_{\mathcal{R}}))$ by induction on n. For $m \in \mathbb{N}$, set $\underline{m} := \mathsf{s}^m(0)$. Consider the base case $n = 1$. Observe that, unlike for innermost rewriting, $\mathsf{f}(\underline{1},t) \to^5_{\mathcal{R}} \mathsf{c}(t,t)$. Since $\mathrm{dh}(\mathsf{c}(t,t),\to_{\mathcal{R}}) = 2 \cdot \mathrm{dh}(t,\to_{\mathcal{R}})$, the claim is easy to establish for this case. For the inductive step, consider a maximal derivation $\mathsf{f}(\underline{n+1},t) \to_{\mathcal{R}} \mathsf{f}(\underline{n},\mathsf{d}(\mathsf{f}(\underline{n},t))) \to_{\mathcal{R}} \cdots$. Applying induction hypothesis twice we obtain

$$\begin{aligned}\mathrm{dh}(\mathsf{f}(\underline{n+1},t),\to_{\mathcal{R}}) > \mathrm{dh}(\mathsf{f}(\underline{n},\mathsf{d}(\mathsf{f}(\underline{n},t))),\to_{\mathcal{R}}) &> \mathrm{dh}(\mathsf{f}(\underline{n},\mathsf{f}(\underline{n},t)),\to_{\mathcal{R}})\\ &> 2^{2^{n-1}} \cdot (2^{2^{n-1}} \cdot (1 + \mathrm{dh}(t,\to_{\mathcal{R}})))\\ &= 2^{2^n} \cdot (1 + \mathrm{dh}(t,\to_{\mathcal{R}}))\ .\end{aligned}$$

4 Exponential Path Order EPO

In this section we introduce the aforementioned order EPO that is used in the proof of Theorem 3.4. We slightly extend the definitions and results originally presented by the second author in [15].

The path order EPO is defined over *sequences* of terms from $\mathcal{T}(\mathcal{F},\mathcal{V})$. To denote sequences, we use an auxiliary function symbol $\mathsf{list} \notin \mathcal{F}$. The function symbol list is variadic, i.e., the arity of list is finite, but arbitrary. We write $[t_1\ \cdots\ t_n]$ instead of $\mathsf{list}(t_1,\dots,t_n)$. For sequences $[s_1\ \cdots\ s_n]$ and $[t_1\ \cdots\ t_m]$, we write $[s_1\ \cdots\ s_n]^\frown[t_1\ \cdots\ t_m]$ to denote the concatenation $[s_1\ \cdots\ s_n\ t_1\ \cdots\ t_m]$. We write $\mathcal{T}^\star(\mathcal{F},\mathcal{V})$ for the set of finite sequences of terms from $\mathcal{T}(\mathcal{F},\mathcal{V})$, i.e. $\mathcal{T}^\star(\mathcal{F},\mathcal{V}) := \{[t_1\ \cdots\ t_n] \mid n \in \mathbb{N} \text{ and } t_1,\dots,t_n \in \mathcal{T}(\mathcal{F},\mathcal{V})\}$. Each term $t \in \mathcal{T}(\mathcal{F},\mathcal{V})$ is identified with the single list $[t] = \mathsf{list}(t) \in \mathcal{T}^\star(\mathcal{F},\mathcal{V})$. This identification ensures $\mathcal{T}(\mathcal{F},\mathcal{V}) \subseteq \mathcal{T}^\star(\mathcal{F},\mathcal{V})$. We use $a,b,c,\dots$ to denote elements of $\mathcal{T}^\star(\mathcal{F},\mathcal{V})$, possibly extending them by subscripts.

▶ **Definition 4.1.** Let $a,b \in \mathcal{T}^\star(\mathcal{F},\mathcal{V})$, and let $\ell \geqslant 1$. Below we assume $f,g \in \mathcal{F}$. We define $a >^\ell_{\mathsf{epo}} b$ with respect to the precedence $\gtrsim$ if either

1) $a = f(s_1,\dots,s_m)$ and $s_i \geqslant^\ell_{\mathsf{epo}} b$ for some $i \in \{1,\dots,m\}$, or

2) $a = f(s_1,\dots,s_m)$, $b = [t_1\ \cdots\ t_n]$ with $n = 0$ or $2 \leqslant n \leqslant \ell$, f is a defined function symbol, and $a >^{\ell}_{\mathsf{epo}} t_j$ for all $j \in \{1,\dots,n\}$, or
3) $a = f(s_1,\dots,s_m)$, $b = g(t_1,\dots,t_n)$ with $n \leqslant \ell$, f is a defined function symbol with $f > g$, and a is a strict superterm (modulo $\sim$) of all t_j ($j \in \{1,\dots,n\}$), or
4) $a = [s_1\ \cdots\ s_m]$, $b = b_1 \frown \cdots \frown b_m$, and for some $j \in \{1,\dots,m\}$,
   - $s_1 \sim b_1$, …, $s_{j-1} \sim b_{j-1}$,
   - $s_j >^{\ell}_{\mathsf{epo}} b_j$, and
   - $s_{j+1} \geqslant^{\ell}_{\mathsf{epo}} b_{j+1}$, …, $s_m \geqslant^{\ell}_{\mathsf{epo}} b_m$, or
5) $a = f(s_1,\dots,s_m)$, $b = g(t_1,\dots,t_n)$ with $n \leqslant \ell$, f and g are defined function symbols with $f \sim g$, and for some $j \in \{1,\dots,\min(m,n)\}$,
   - $s_1 \sim t_1$, …, $s_{j-1} \sim t_{j-1}$,
   - $s_j \rhd\!\!\!/_{\sim} t_j$, and
   - $a \rhd\!\!\!/_{\sim} t_{j+1}$, …, $a \rhd\!\!\!/_{\sim} t_n$.

Here we set $\geqslant^{\ell}_{\mathsf{epo}} := >^{\ell}_{\mathsf{epo}} \cup \sim$. Finally, we set $>_{\mathsf{epo}} := \bigcup_{k\geqslant 1} >^{\ell}_{\mathsf{epo}}$ and $\geqslant_{\mathsf{epo}} := \bigcup_{k\geqslant 1} \geqslant^{\ell}_{\mathsf{epo}}$.

We note that, by Definition 4.1.2 with $n = 0$, we have $f(s_1,\dots,s_m) >^{\ell}_{\mathsf{epo}} [\,]$ for all $\ell \geqslant 1$ if f is a defined function symbol. It is not difficult to see that $l \leqslant k$ implies $>^{l}_{\mathsf{epo}} \subseteq >^{k}_{\mathsf{epo}}$. Unfortunately EPO is not a restriction of lexicographic path orders, as the length of lists is not bounded globally. However, the critical Clause 4 amounts to a *lifting* from terms to sequences of terms in the sense of [17, Section 3]. Conclusively an application of the main result of [17, Section 3] gives well-foundedness of $>^{\ell}_{\mathsf{epo}}$.

▶ **Lemma 4.2.** *Let $a = a_1 \frown \cdots \frown a_{j-1} \frown a_j \frown a_{j+1} \cdots \frown a_m$. Suppose that $a_j >^{\ell}_{\mathsf{epo}} b$. Then $a >^{\ell}_{\mathsf{epo}} a_1 \frown \cdots \frown a_{j-1} \frown b \frown a_{j+1} \cdots \frown a_m$.*

Following Arai and the second author [2] we define G_ℓ that measures the $>^{\ell}_{\mathsf{epo}}$-descending lengths:

▶ **Definition 4.3.** We define $G_\ell : \mathcal{T}^\star(\mathcal{F},\mathcal{V}) \to \mathbb{N}$ as

$$G_\ell(a) := \max\{G_\ell(b) + 1 \mid b \in \mathcal{T}^\star(\mathcal{F},\mathcal{V}) \text{ and } a >^{\ell}_{\mathsf{epo}} b\}\ .$$

▶ **Lemma 4.4.** *For all $\ell \geqslant 1$ we have*
1) $\rhd\!\!\!/_{\sim} \subseteq >^{\ell}_{\mathsf{epo}}$,
2) if $t \in \mathcal{T}(\mathcal{C},\mathcal{V})$ then $G_\ell(t) = \mathsf{dp}(t)$, and
3) $G_\ell([t_1\ \cdots\ t_m]) = \sum_{i=1}^{m} G_\ell(t_i)$.

Proof. The Properties 1) and 2) can be shown by straight forward inductive arguments. We prove Property 3) for the non-trivial case $m \geqslant 2$. It is not difficult to check that $G_\ell([t_1\ \cdots\ t_m]) \geqslant \sum_{i=1}^{m} G_k(t_i)$. We show that $G_\ell([t_1\ \cdots\ t_m]) \leqslant \sum_{i=1}^{m} G_\ell(t_i)$ by induction on $G_\ell([t_1\ \cdots\ t_m])$.

Let $a = [t_1\ \cdots\ t_m]$. Then, it suffices to show that for any $b \in \mathcal{T}^\star(\mathcal{F},\mathcal{V})$, $a >^{\ell}_{\mathsf{epo}} b$ implies $G_\ell(b) < \sum_{i=1}^{m} G_\ell(t_i)$. Fix $b \in \mathcal{T}^\star(\mathcal{F},\mathcal{V})$ and suppose that $a >^{\ell}_{\mathsf{epo}} b$. Then, by Definition 4.1.4, there exist some $b_1,\dots,b_m \in \mathcal{T}^\star(\mathcal{F},\mathcal{V})$ and $j \in \{1,\dots,m\}$ such that $b = b_1 \frown \cdots \frown b_m$, $t_i \geqslant^{\ell}_{\mathsf{epo}} b_i$ for each $i \in \{1,\dots m\}$, and $t_j >^{\ell}_{\mathsf{epo}} b_j$. By the definition of G_ℓ, we have that $G_\ell(t_i) \geqslant G_\ell(b_i)$ for each $i \in \{1,\dots m\}$, and $G_\ell(t_j) > G_\ell(b_j)$. Thus $\sum_{i=1}^{m} G_\ell(b_i) < \sum_{i=1}^{m} G_\ell(t_i)$ follows. Let $b_i = [u_{i,1} \cdots u_{i,n_i}]$ for each $i \in \{1,\dots,m\}$. Then, since $G_\ell(b) < G_\ell(a)$, induction hypothesis gives $G_\ell(b) \leqslant \sum_{i=1}^{m}\sum_{j=1}^{n_i} G_\ell(u_{i,j})$. Recalling that $\sum_{j=1}^{n_i} G_\ell(u_{i,j}) \leqslant G_\ell(b_i)$ also holds for each $i \in \{1,\dots,m\}$. Summing up, we obtain that $G_\ell(b) \leqslant \sum_{i=1}^{m}\sum_{j=1}^{n_i} G_\ell(u_{i,j}) \leqslant \sum_{i=1}^{m} G_\ell(b_i) < \sum_{i=1}^{m} G_\ell(t_i)$. ◀

We finally arrive at the main theorem of this section.

▶ **Theorem 4.5.** *Suppose that $f \in \mathcal{F}$ with arity $n \leqslant \ell$ and $t_1, \dots, t_n \in \mathcal{T}(\mathcal{F}, \mathcal{V})$. Let $N := \max\{G_\ell(t_i) \mid 1 \leqslant i \leqslant n\} + 1$. Then*

$$G_\ell(f(t_1, \dots, t_n)) \leqslant (\ell+1)^{N^\ell \cdot \mathsf{rk}(f) + \sum_{i=1}^n N^{\ell-i} G_\ell(t_i)} \,. \tag{1}$$

Proof. Let $t = f(t_1, \dots, t_n)$. We prove the inequality (1) by induction on $G_\ell(t)$. In the base case, $G_\ell(t) = 0$, and hence the inequality (1) trivially holds. In the case $G_\ell(t) > 0$, it suffices to show that for any $b \in \mathcal{T}^\star(\mathcal{F}, \mathcal{V})$, $t >^\ell_{\mathsf{epo}} b$ implies $G_\ell(b) < (\ell+1)^{N^\ell \cdot \mathsf{rk}(f) + \sum_{i=1}^n N^{\ell-i} G_\ell(t_i)}$. The induction case splits into four cases depending on which rule of Definition 4.1 concludes $t >^\ell_{\mathsf{epo}} b$. For the sake of convenience, we start with the case corresponding to Definition 4.1.2. Namely, we consider the case $b = [s_1 \ \cdots \ s_k]$ where $2 \leqslant k \leqslant \ell$ and $t >^\ell_{\mathsf{epo}} s_i$ for all $i \in \{1, \dots, k\}$. We show that for all $i \in \{1, \dots, k\}$,

$$G_\ell(s_i) \leqslant (\ell+1)^{(N^\ell \cdot \mathsf{rk}(f) + \sum_{i=1}^n N^{\ell-i} G_\ell(t_i)) - 1} \,. \tag{2}$$

We prove the inequality (2) by case analysis according to the last rule that concludes $t >^\ell_{\mathsf{epo}} s_i$. Fix some element $u \in \{s_i \mid i \in \{1, \dots, k\}\}$.

1) Case $t_j \geqslant^\ell_{\mathsf{epo}} u$ for some $j \in \{1, \dots, n\}$: In this case we trivially see

$$G_\ell(u) \leqslant G_\ell(t_j) \leqslant (\ell+1)^{(N^\ell \cdot \mathsf{rk}(f) + \sum_{i=1}^n N^{\ell-i} G_\ell(t_i)) - 1} \,. \tag{3}$$

2) Case $u = g(u_1, \dots, u_m)$ where $m \leqslant \ell$, g is a defined symbol with $f > g$ and for all $i \in \{1, \dots, m\}$, t is a strict superterm (modulo $\sim$) of u_i: Let $M := \max\{G_\ell(u_i) \mid 1 \leqslant i \leqslant m\} + 1$. Then, we have $M \leqslant N$ since t is a strict superterm (modulo $\sim$) of every u_i. We claim

$$M^\ell \cdot \mathsf{rk}(g) + \sum_{i=1}^m M^{\ell-i} G_\ell(u_i) < N^\ell \cdot \mathsf{rk}(f) + \sum_{i=1}^n N^{\ell-i} G_\ell(t_i) \,.$$

To see this, conceive left- and right-hand side as numbers represented in base M and respectively N of length ℓ (observe $G_\ell(u_i) < M$ and $G_\ell(t_i) < N$). From $\mathsf{rk}(g) < \mathsf{rk}(f)$ and $M \leqslant N$ the above inequality is obvious. This allows us to conclude

$$\begin{aligned} G_\ell(u) &\leqslant (\ell+1)^{M^\ell \cdot \mathsf{rk}(g) + \sum_{i=1}^m M^{\ell-i} G_\ell(u_i)} \\ &\leqslant (\ell+1)^{N^\ell \cdot \mathsf{rk}(f) + \sum_{i=1}^n N^{\ell-i} G_\ell(t_i) - 1} \,. \end{aligned} \tag{4}$$

Here the first inequality follows by induction hypothesis.

3) Case $u = g(u_1, \dots, u_m)$ where $m \leqslant \ell$, g is a defined symbol with $f \sim g$ and there exists $j \in \{1, \dots, \min(n, m)\}$ such that $t_i \sim u_i$ for all $i < j$, t_j is a strict superterm (modulo $\sim$) of u_j, and t is a strict superterm (modulo $\sim$) of u_i for all $i > j$: Let $M := \max\{G_\ell(u_i) \mid 1 \leqslant i \leqslant m\} + 1$ and consider the following claim:

▶ **Claim 4.6.** $\sum_{i=1}^m M^{\ell-i} G_\ell(u_i) < \sum_{i=1}^n N^{\ell-i} G_\ell(t_i)$.

To prove this claim, observe that the assumptions give $G_\ell(u_i) = G_\ell(t_i)$ for all $i < j$, $G_\ell(u_j) < G_\ell(t_j)$, and $G_\ell(u_i) < N$ for all $i > j$: This implies that $M \leqslant N$ and

$$\begin{aligned} \sum_{i=1}^m M^{\ell-i} G_\ell(u_i) &\leqslant \sum_{i=1}^{j-1} N^{\ell-i} G_\ell(t_i) + N^{\ell-j}(G_\ell(t_j) - 1) + \sum_{i=j+1}^n N^{\ell-i}(N-1) \\ &< \sum_{i=1}^n N^{\ell-i} G_\ell(t_i) \,. \end{aligned}$$

As above, the claim together with induction hypothesis yields

$$G_\ell(u) \leqslant (\ell+1)^{N^\ell \cdot \mathsf{rk}(f) + \sum_{i=1}^n N^{\ell-i} G_\ell(t_i) - 1} \,. \tag{5}$$

Summing up the inequality (3), (4) and (5) concludes inequality (2). Thus, having $G_\ell(b) = \sum_{i=1}^{k} G_\ell(s_i)$ by Lemma 4.4, and employing $k \leqslant \ell$, we see

$$\begin{aligned} G_\ell(b) &\leqslant \ell \cdot (\ell+1)^{(N^\ell \cdot \mathsf{rk}(f) + \sum_{i=1}^{n} N^{\ell-i} G_\ell(t_i)) - 1} && \text{(by the inequality (2))} \\ &< (\ell+1)^{N^\ell \cdot \mathsf{rk}(f) + \sum_{i=1}^{n} N^{\ell-i} G_\ell(t_i)} \,. \end{aligned}$$

This completes the case for Definition 4.1.2. The cases for Definition 4.1.1, 4.1.3 and 4.1.5 follow respectively from the inequality (3), (4) and (5). ◀

5 Embedding EPO* in EPO

In this section we define *predicative interpretations* $\mathcal{I}$ that embed innermost rewrite steps into $>_{\mathsf{epo}}^{\ell}$, i.e., if $s \xrightarrow{\mathsf{i}}_{\mathcal{R}} t$, then $\mathcal{I}(s) >_{\mathsf{epo}}^{\ell} \mathcal{I}(t)$. The definition of $\mathcal{I}$ makes use of mapping safe underlying the definition of $>_{\mathsf{epo}\star}$, Based on this embedding we then use Theorem 4.5 to prove that EPO* induces exponential (innermost) runtime complexity (Theorem 3.4).

Before we define predicative interpretations, we start with a simple observation. Let $\mathcal{R}$ be a TRS compatible with some instance $>_{\mathsf{epo}\star}$, i.e., $\mathcal{R} \subseteq >_{\mathsf{epo}\star}$. For the moment, suppose $\mathcal{R}$ is completely defined. We replace this restriction by constructor TRS later on. Since $\mathcal{R}$ is completely defined, normal forms and constructor terms coincide, and thus $s \xrightarrow{\mathsf{i}}_{\mathcal{R}} t$ if $s = C[l\sigma], t = C[r\sigma]$ for some rule $l \to r \in \mathcal{R}$ where additionally $l\sigma \in \mathcal{B}$. Let t be obtained by rewriting a basic term s. By the inclusion $\mathcal{R} \subseteq >_{\mathsf{epo}\star}$, every normal argument t_i of t is irreducible, i.e., $t_i \in \mathcal{T}(\mathcal{C}, \mathcal{V})$. We capture this observation in the definition of $\mathcal{B}^{\to}$:

▶ **Definition 5.1.** The set $\mathcal{B}^{\to}$ is the least set of terms such that
1) $\mathcal{T}(\mathcal{C}, \mathcal{V}) \subseteq \mathcal{B}^{\to}$, and
2) if $f \in \mathcal{F}$, $\vec{s} \subseteq \mathcal{T}(\mathcal{C}, \mathcal{V})$ and $\vec{t} \subseteq \mathcal{B}^{\to}$ then $f(\vec{s}\,;\vec{t}) \in \mathcal{B}^{\to}$.

Note that $\mathcal{B} \subseteq \mathcal{B}^{\to}$. The verification of the next Lemma is straight forward:

▶ **Lemma 5.2.** *Let $\mathcal{R}$ be a completely defined TRS compatible with $>_{\mathsf{epo}\star}$. If $s \in \mathcal{B}^{\to}$ and $s \xrightarrow{\mathsf{i}}_{\mathcal{R}} t$ then $t \in \mathcal{B}^{\to}$.*

We define predicative interpretation $\mathcal{I}$ as follows. Since we are only interested in the length of derivations starting from basic terms, Lemma 5.2 justifies that only terms from $\mathcal{B}^{\to}$ are considered. For each defined symbol f, let f_{n} be a fresh function symbol, and let $\mathcal{F}_{\mathsf{n}} = \{f_{\mathsf{n}} \mid f \in \mathcal{D}\} \cup \mathcal{C}$. Here the arity of f_{n} is k where $\mathsf{nrm}(f) = \{i_1, \ldots, i_k\}$, moreover f_{n} is still considered a defined function symbol when applying Definition 4.1. We extend the (admissible) precedence $\gtrsim$ to $\mathcal{F}_{\mathsf{n}}$ in the obvious way: $f_{\mathsf{n}} \sim g_{\mathsf{n}}$ if $f \sim g$ and $f_{\mathsf{n}} > g_{\mathsf{n}}$ if $f > g$.

▶ **Definition 5.3.** A *predicative interpretation* $\mathcal{I}$ is a mapping $\mathcal{I} : \mathcal{B}^{\to} \to \mathcal{T}^\star(\mathcal{F}, \mathcal{V})$ defined as follows:
1) $\mathcal{I}(t) = [\,]$ if $t \in \mathcal{T}(\mathcal{C}, \mathcal{V})$, and otherwise
2) $\mathcal{I}(t) = [f_{\mathsf{n}}(t_1, \ldots, t_k)] \frown \mathcal{I}(t_{k+1}) \frown \cdots \frown \mathcal{I}(t_{k+n})$ where $t = f(t_1, \ldots, t_k\,; t_{k+1}, \ldots, t_{k+n})$.

The next lemma provides the embedding of root steps for completely defined, compatible, TRSs $\mathcal{R}$. Here we could simply define $\mathcal{I}(t) = f_{\mathsf{n}}(t_1, \ldots, t_k)$ in Case 2). The complete definition becomes only essential when we look at closure under context in Lemma 5.5.

▶ **Lemma 5.4.** *Let $s \in \mathcal{B}$ and let $\sigma : \mathcal{V} \to \mathcal{T}(\mathcal{C}, \mathcal{V})$ be a substitution. If $s >_{\mathsf{epo}\star} t$ then $\mathcal{I}(s\sigma) >_{\mathsf{epo}}^{|t|} \mathcal{I}(t\sigma)$.*

Proof. Let f denote the (defined) root symbol of s, and let $s_1, \dots, s_l$ denote the normal arguments of s. Thus $\mathcal{I}(s\sigma) = [f_{\mathsf{n}}(s_1\sigma, \dots, s_l\sigma)] = f_{\mathsf{n}}(s_1\sigma, \dots, s_l\sigma)$. If $t \in \mathcal{T}(\mathcal{C}, \mathcal{V})$ then the lemma trivially follows as $\mathcal{I}(t\sigma) = [\,]$. Hence suppose $t \notin \mathcal{T}(\mathcal{C}, \mathcal{V})$.

We continue by induction on the definition of $>_{\mathsf{epo}\star}$. Let $t = g(t_1, \dots, t_k\,;\, t_{k+1}, \dots, t_{k+n})$ and so

$$\mathcal{I}(t\sigma) = [g_{\mathsf{n}}(t_1\sigma, \dots, t_k\sigma)] \frown \mathcal{I}(t_{k+1}\sigma) \frown \cdots \frown \mathcal{I}(t_{k+n}\sigma)\ .$$

Observe that $\mathcal{I}(x\sigma) = [\,]$ for all variables x in t. Using this we see that the length of the list $\mathcal{I}(t\sigma)$ is bound by $|t|$. Hence by Definition 4.1.2, it suffices to verify $\mathcal{I}(s\sigma) >_{\mathsf{epo}}^{|t|} \mathcal{I}(t_i\sigma)$ for all safe arguments t_i ($i \in \{k+1, \dots, k+n\}$), and further to show

$$f_{\mathsf{n}}(s_1\sigma, \dots, s_l\sigma) >_{\mathsf{epo}}^{|t|} g_{\mathsf{n}}(t_1\sigma, \dots, t_k\sigma)\ . \tag{6}$$

Since $s \in \mathcal{B}$ but $t \notin \mathcal{T}(\mathcal{C}, \mathcal{V})$, a consequence of Lemma 3.3 is that $s >_{\mathsf{epo}\star} t$ follows either by Definition 3.2.2 or Definition 3.2.3. Let t_i be a safe argument. Then by definition $s >_{\mathsf{epo}\star} t_i$ and induction hypothesis yields $\mathcal{I}(s\sigma) >_{\mathsf{epo}}^{|t|} \mathcal{I}(t_i\sigma)$ (employing $|t_i| \leqslant |t|$). It thus remains to verify (6). We continue by case analysis.

1) Suppose $f > g$, i.e., Definition 3.2.2 applies. Then $f_{\mathsf{n}} > g_{\mathsf{n}}$ by definition. By Definition 4.1.3 it suffices to prove $f_{\mathsf{n}}(s_1\sigma, \dots, s_l\sigma) \rhd\!\!\!/_{\sim} t_i\sigma$ for all $i \in \{1, \dots, k\}$. Fix $i \in \{1, \dots, k\}$. According to Definition 3.2.2 $s \sqsupset_{\mathsf{epo}\star} t_i$ holds, and thus there exists $j \in \{1, \dots, l\}$ such that $s_j \sqsupseteq_{\mathsf{epo}\star} t_i$. Hence $s_j \trianglerighteq\!\!\!/_{\sim} t_i$ by Lemma 3.3, from which we conclude $f_{\mathsf{n}}(s_1\sigma, \dots, s_l\sigma) \rhd\!\!\!/_{\sim} t_i\sigma$ since we suppose $\sigma : \mathcal{V} \to \mathcal{T}(\mathcal{C}, \mathcal{V})$.
2) Suppose $f \sim g$, i.e., Definition 3.2.3 applies. Then $f_{\mathsf{n}} \sim g_{\mathsf{n}}$. By Definition 4.1.5 it suffices to prove (i) $s_1\sigma \sim t_1\sigma$, …, $s_{\ell-1}\sigma \sim t_{\ell-1}\sigma$, (ii) $s_\ell\sigma \rhd\!\!\!/_{\sim} t_\ell\sigma$, and further (iii) $f_{\mathsf{n}}(s_1\sigma, \dots, s_l\sigma) \rhd\!\!\!/_{\sim} t_{\ell+1}\sigma$, …, $f_{\mathsf{n}}(s_1\sigma, \dots, s_l\sigma) \rhd\!\!\!/_{\sim} t_k\sigma$ for some $\ell \in \{1, \dots, k\}$. The assumptions in Definition 3.2.3 yield $s_1 \stackrel{s}{\sim} t_1$, …, $s_{\ell-1} \stackrel{s}{\sim} t_{\ell-1}$ from which we conclude (i), further $s_\ell \sqsupset_{\mathsf{epo}\star} t_\ell$ from which we conclude (ii) with the help of Lemma 3.3 (using $s_\ell \in \mathcal{T}(\mathcal{C}, \mathcal{V})$), and finally $s \sqsupset_{\mathsf{epo}\star} t_{\ell+1}$, …, $s \sqsupset_{\mathsf{epo}\star} t_k$ from which we obtain (iii) as in the case above.

◀

▶ **Lemma 5.5.** *Let $s, t \in \mathcal{B}^{\to}$ and let C be a context such that $C[s] \in \mathcal{B}^{\to}$. If $\mathcal{I}(s) >_{\mathsf{epo}}^{\ell} \mathcal{I}(t)$ then $\mathcal{I}(C[s]) >_{\mathsf{epo}}^{\ell} \mathcal{I}(C[t])$.*

Proof. We show the lemma by induction on C. It suffices to consider the step case. Observe that by the assumption $\mathcal{I}(s) >_{\mathsf{epo}}^{\ell} \mathcal{I}(t)$, $s \notin \mathcal{T}(\mathcal{C}, \mathcal{V})$ since otherwise $\mathcal{I}(s) = [\,]$ is $>_{\mathsf{epo}}^{\ell}$-minimal. By definition of $\mathcal{B}^{\to}$ we can thus assume $C = f(s_1, \dots, s_k\,;\, s_{k+1}, \dots, C'[\Box], \dots s_{k+l})$ for some context C'. Thus, for each $u \in \{s, t\}$,

$$\mathcal{I}(C[u]) = [f_{\mathsf{n}}(s_1, \dots, s_k)] \frown \mathcal{I}(s_{k+1}) \frown \cdots \frown \mathcal{I}(C'[u]) \frown \cdots \frown \mathcal{I}(s_{k+l})\ .$$

By induction hypothesis $\mathcal{I}(C'[s]) >_{\mathsf{epo}}^{\ell} \mathcal{I}(C'[t])$. We conclude using Lemma 4.2. ◀

Combining Lemma 5.4 and Lemma 5.5 completes the embedding.

▶ **Lemma 5.6.** *Let $\mathcal{R}$ be a completely defined TRS compatible with $>_{\mathsf{epo}\star}$. Set $\ell := \max\{|r| \mid l \to r \in \mathcal{R}\}$. If $s \in \mathcal{B}^{\to}$ and $s \xrightarrow{\mathsf{i}}_{\mathcal{R}} t$ then $\mathcal{I}(s) >_{\mathsf{epo}}^{\ell} \mathcal{I}(t)$.*

Proof. Suppose $s \xrightarrow{\mathsf{i}}_{\mathcal{R}} t$. Hence there exists a context C, substitution σ and rule $l \to r \in \mathcal{R}$ such that $s = C[l\sigma]$ and $t = C[r\sigma]$. By the assumption that $\mathcal{R}$ is completely defined, $l \in \mathcal{B}$ and $\sigma : \mathcal{V} \to \mathcal{T}(\mathcal{C}, \mathcal{V})$. Since $\mathcal{R} \subseteq >_{\mathsf{epo}\star}$, we obtain $\mathcal{I}(l\sigma) >_{\mathsf{epo}}^{\ell} \mathcal{I}(r\sigma)$ by Lemma 5.4 (additionally employing $>_{\mathsf{epo}}^{|r|} \subseteq >_{\mathsf{epo}}^{\ell}$). Lemma 5.5 then establishes $\mathcal{I}(s) >_{\mathsf{epo}}^{\ell} \mathcal{N}(t)$. ◀ ◀

We obtain Theorem 3.4 formulated for completely defined TRSs.

▶ **Theorem 5.7.** *Let $\mathcal{R}$ be a completely defined, possibly infinite, TRS compatible with $>_{\mathsf{epo}\star}$. Suppose $\max\{|r| \mid l \to r \in \mathcal{R}\}$ is well-defined. There exists $k \in \mathbb{N}$ such that $\mathrm{rc}^{\mathsf{i}}_{\mathcal{R}}(n) \leqslant 2^{\mathsf{O}(n^k)}$.*

Proof. Set $\ell := \max\{\max\{|r| \mid l \to r \in \mathcal{R}\}, \max\{\mathsf{ar}(f) \mid f \in \mathcal{F}\}\}$. Not that ℓ is well-defined as $\mathcal{F}$ is finite and non-variadic. We prove the existence of $c_1, c_2 \in \mathbb{N}$ so that for any $s \in \mathcal{B}$, $\mathrm{dh}(s, \xrightarrow{\mathsf{i}}_{\mathcal{R}}) \leqslant 2^{c_1 \cdot |s|^{c_2}}$. Consider a maximal derivation $s = t_0 \xrightarrow{\mathsf{i}}_{\mathcal{R}} t_1 \xrightarrow{\mathsf{i}}_{\mathcal{R}} \cdots \xrightarrow{\mathsf{i}}_{\mathcal{R}} t_n$ with $s \in \mathcal{B}$. Let $i \in \{0, \ldots, n-1\}$. We observed $t_i \in \mathcal{B}^{\to}$ in Lemma 5.2, and thus $\mathcal{I}(t_i) >^{\ell}_{\mathsf{epo}} \mathcal{I}(t_{i+i})$ due to Lemma 5.6. So in particular $\mathrm{dh}(s, \xrightarrow{\mathsf{i}}_{\mathcal{R}}) \leqslant G_\ell(\mathcal{I}(s))$. It remains to estimate $G_\ell(\mathcal{I}(s))$ in terms of $|s|$: for this, suppose $s = f(s_1, \ldots, s_k\,; s_{k+1}, \ldots, s_{k+l})$ for some $f \in \mathcal{D}$ and $s_i \in \mathcal{T}(\mathcal{C}, \mathcal{V})$ $(i \in \{1, \ldots, k+l\})$. By definition $\mathcal{I}(s) = f_{\mathsf{n}}(s_1, \ldots, s_k)$. Set $N := \max\{G_\ell(s_i) \mid 1 \leqslant i \leqslant k\} + 1$, and verify

$$N \leqslant 1 + \sum_{i=1}^{k} G_\ell(s_i) \leqslant 1 + \sum_{i=1}^{k} \mathsf{dp}(s_i) \leqslant |s| \,. \tag{7}$$

For the second inequality we employ Lemma 4.4, which gives $G_\ell(s_i) = \mathsf{dp}(s_i)$ as $s_i \in \mathcal{T}(\mathcal{C}, \mathcal{V})$ for all $i \in \{1, \ldots, k\}$. Applying Theorem 4.5 we see

$$\begin{aligned}
G_\ell(\mathcal{I}(s)) &= G_\ell(f_{\mathsf{n}}(s_1, \ldots, s_k)) \\
&\leqslant (\ell+1)^{N^\ell \cdot \mathsf{rk}(f_{\mathsf{n}}) + \sum_{i=1}^{k} N^{\ell-i} \cdot G_\ell(s_i)} && \text{(by Theorem 4.5, using } k \leqslant \ell) \\
&\leqslant (\ell+1)^{|s|^\ell \cdot \mathsf{rk}(f_{\mathsf{n}}) + |s|^\ell \cdot \sum_{i=1}^{k} G_\ell(s_i)} && \text{(by Equation 7)} \\
&\leqslant (\ell+1)^{|s|^\ell \cdot \mathsf{rk}(f_{\mathsf{n}}) + |s|^\ell \cdot |s|} && \text{(by Equation 7)} \\
&\leqslant (\ell+1)^{(\mathsf{rk}(f_{\mathsf{n}})+1) \cdot |s|^{\ell+1}} \,.
\end{aligned}$$

Since ℓ depends only on $\mathcal{R}$ and $\mathcal{F}$, and $\mathsf{rk}(f_{\mathsf{n}})$ is bounded by some constant depending only on $\mathcal{F}$, simple arithmetical reasoning gives the constants c_1, c_2 such that $\mathrm{dh}(s, \xrightarrow{\mathsf{i}}_{\mathcal{R}}) \leqslant G_\ell(\mathcal{I}(s)) \leqslant 2^{c_1 \cdot |s|^{c_2}}$. This concludes the Theorem. ◀

We now lift the restriction that $\mathcal{R}$ is completely defined for constructor TRSs $\mathcal{R}$. The idea is to extend $\mathcal{R}$ with sufficiently many rules so that the resulting system is completely defined and Theorem 5.7 applicable.

▶ **Definition 5.8.** Let $\bot$ be a fresh constructor symbol. We define

$$\mathcal{S} := \{t \to \bot \mid t \in \mathcal{T}(\mathcal{F} \cup \{\bot\}, \mathcal{V}) \cap \mathrm{NF}(\mathcal{R}) \text{ and the root symbol of } t \text{ is defined}\} \,.$$

We extend the precedence $\gtrsim$ to $\mathcal{F} \cup \{\bot\}$ so that $\bot$ is minimal. Thus $\mathcal{S} \subseteq >_{\mathsf{epo}\star}$ follows by one application of Definition 3.2.2. Further, the *completely defined* TRS $\mathcal{R} \cup \mathcal{S}$ is able to simulate $\xrightarrow{\mathsf{i}}_{\mathcal{R}}$ derivations for constructor TRS $\mathcal{R}$:

▶ **Lemma 5.9.** *Suppose $\mathcal{R}$ is a constructor TRS. Then $\mathcal{R} \cup \mathcal{S}$ is completely defined. Further, if $s \xrightarrow{\mathsf{i}}{}^{\ell}_{\mathcal{R}} t$ then $s \xrightarrow{\mathsf{i}}{}^{\ell'}_{\mathcal{R}\cup\mathcal{S}} t'$ for some t' and $\ell' \geqslant \ell$.*

For the latter property we show that $s \xrightarrow{\mathsf{i}}_{\mathcal{R}} t$ implies $s' \xrightarrow{\mathsf{i}}{}^{+}_{\mathcal{R}\cup\mathcal{S}} t'$ for s' and t' $\mathcal{S}$-normal forms of s and t. Here the key observation is that rewriting according to $\mathcal{S}$ does not interfere with pattern matching with respect to $\mathcal{R}$.

An immediate consequence of Lemma 5.9 is $\mathrm{rc}^{\mathsf{i}}_{\mathcal{R}}(n) \leqslant \mathrm{rc}^{\mathsf{i}}_{\mathcal{R}\cup\mathcal{S}}(n)$, i.e., the innermost runtime complexity of $\mathcal{R}$ can be analysed through $\mathcal{R} \cup \mathcal{S}$. We arrive at the proof of our main theorem:

Proof of Theorem 3.4. Suppose $\mathcal{R}$ is a constructor TRS compatible with $>_{\mathsf{epo}\star}$. We verify that $\mathrm{rc}^{\mathsf{i}}_{\mathcal{R}}(n) \leqslant 2^{\mathsf{O}(n^k)}$ for some fixed $k \in \mathbb{N}$: let $\mathcal{S}$ be defined according to Definition 5.8. By Lemma 5.9, $\mathcal{R} \cup \mathcal{S}$ is completely defined, and moreover, $\mathrm{rc}^{\mathsf{i}}_{\mathcal{R}}(n) \leqslant \mathrm{rc}^{\mathsf{i}}_{\mathcal{R}\cup\mathcal{S}}(n)$. Note that $\max\{|r| \mid l \to r \in \mathcal{R} \cup \mathcal{S}\} = \max\{|r| \mid l \to r \in \mathcal{R}\}$ is well-defined. Further $(\mathcal{R} \cup \mathcal{S}) \subseteq >_{\mathsf{epo}\star}$ follows by assumption and definition of $\mathcal{S}$. Hence all assumptions of Theorem 5.7 are fulfilled, and we conclude $\mathrm{rc}^{\mathsf{i}}_{\mathcal{R}}(n) \leqslant \mathrm{rc}^{\mathsf{i}}_{\mathcal{R}\cup\mathcal{S}}(n) \leqslant 2^{\mathsf{O}(n^k)}$ for some $k \in \mathbb{N}$. ◀

6 Characterising Exponential Time Computation

In this section we present one application of EPO* in the context of *implicit computational complexity (ICC)*. Following [11, 5] we give semantics to TRS $\mathcal{R}$ as follows:

▶ **Definition 6.1.** Let $\mathcal{V}\mathrm{al} := \mathcal{T}(\mathcal{C}, \mathcal{V})$ denote the set of *values*. Further, let $\mathcal{P} \subseteq \mathcal{V}\mathrm{al}$ be a finite set of *non-accepting patterns*. We call a term t *accepting* (with respect to $\mathcal{P}$) if there exists no $p \in \mathcal{P}$ 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 $\mathcal{P}$ if there exists $\mathsf{f} \in \mathcal{D}$ such that for all $s, t \in \mathcal{V}\mathrm{al}$,

$$s\ R\ t \qquad \Leftrightarrow \qquad \mathsf{f}(s) \xrightarrow{\mathsf{i}}{}^{!}_{\mathcal{R}}\ t \text{ and } t \text{ is accepting}\,.$$

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 $\mathcal{P}$ 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 aims to eliminate by-products of the computation that should not be considered as part of the computed relation R.

As a consequence of Theorem 3.4 we derive our soundness result. Following [14, 5] we employ *graph rewriting* (c.f. [24]) to efficiently compute normal forms.

▶ **Theorem 6.2** (Soundness). *Suppose $\mathcal{R}$ is a constructor TRS compatible with $>_{\mathsf{epo}\star}$. The relations computed by $\mathcal{R}$ are computable in nondeterministic time $2^{\mathsf{O}(n^k)}$ for some $k \in \mathbb{N}$. In particular, if $\mathcal{R}$ is confluent then $f \in \mathsf{FEXP}$ for each function f computed by $\mathcal{R}$.*

Proof. We sketch the implementation of the relation R_f (function f) on a Turing machine M_f.

Single out the corresponding defined function symbol f, and consider some arbitrary input $v \in \mathcal{V}\mathrm{al}$. First writing $\mathsf{f}(v)$ on a dedicated working tape, the machine M_f iteratively rewrites $\mathsf{f}(v)$ to normal form in an innermost fashion. For non-confluent TRSs $\mathcal{R}$, the choice of the redex is performed nondeterministically, otherwise some innermost redex is computed deterministically.

By the assumption $\mathcal{R} \subseteq >_{\mathsf{epo}\star}$, Theorem 3.4 provides an upper bound $2^{|\mathsf{f}(v)|^{c_1}}$ on the number of iterations for some $c_1 \in \mathbb{N}$, i.e., the machine performs at most exponentially many iterations in the size of the input v. Thus the theorem follows if we can prove that each iteration is computable in time exponential in $|v|$.

To investigate into the complexity of a single iteration, consider the i-th iteration with t_i written on the working tape (where $\mathsf{f}(v) \to^i_{\mathcal{R}} t_i$). We want to compute some t_{i+1} with $t_i \xrightarrow{\mathsf{i}}_{\mathcal{R}} t_{i+1}$. Observe that in the presence of duplicating rules, $|t_i|$ might be exponential in i (and $|v|$). As we can only assume $i \leqslant 2^{|\mathsf{f}(v)|^{c_1}}$, we cannot hope to construct t_{i+1} from t_i in time exponential in $|v|$ if we use a representation of terms that is linear in size in the number of symbols.

Instead, we employ the machinery of [5]. By taking sharing into account, [5] achieves an encoding of t_i that is bounded in size polynomially in $|v|$ and i. Hence in particular t_i is encoded in size $2^{|s|^{c_2}}$ for some $c_2 \in \mathbb{N}$ depending only on $\mathcal{R}$. Further, a single step is computable in polynomial time (in the encoding size). And so t_{i+1} is computable from t_i in time $2^{|s|^{c_3}}$ for some $c_3 \in \mathbb{N}$ depending only on $\mathcal{R}$. Overall, we conclude that normal forms are computable in time $2^{|v|^{c_1}} \cdot 2^{|v|^{c_3}} = 2^{\mathsf{O}(|v|^k)}$ for some $k \in \mathbb{N}$ worst case.

After the final iteration, the machine M_f checks whether the computed normal form t_l is accepting and either accepts or rejects the computation. Using the machinery of [5] pattern matching is polynomial the encoding size of t_l, by the above bound on encoding sizes the operation is exponential in $|v|$. As v was chosen arbitrarily and k depends only on $\mathcal{R}$, we conclude the theorem. ◀

▶ **Example 6.3.** [Example 6.3 continued]. Since $\mathcal{R}_{\mathsf{fib}} \subseteq >_{\mathsf{epo}\star}$, Theorem 6.2 yields that the function $f_{\mathsf{fib}} : \mathcal{T}(\{\mathsf{0},\mathsf{s}\},\mathcal{V}) \to \mathcal{T}(\{\mathsf{0},\mathsf{s}\},\mathcal{V})$ computed by $\mathcal{R}_{\mathsf{fib}}$ is computable in exponential time.

In correspondence to Theorem 6.2, EPO* is also complete in the sense that every exponential time function is computable by a TRS compatible with EPO*. To prove completeness we use the characterisation of the exponential time computable functions $\mathcal{N}$ given by Arai and the second author [1], or more precisely the resulting *term rewriting characterisation* $\mathcal{R}_\mathcal{N}$ presented in [15].

Similar to the definition of EPO*, the class $\mathcal{N}$ relies on a syntactic separation of argument positions into *normal* and *safe* ones. To highlight this separation, we again write $f(\vec{x};\vec{y})$ instead of $f(\vec{x},\vec{y})$ for normal arguments $\vec{x}$ and safe arguments $\vec{y}$. The class $\mathcal{N}$ is defined as the least class containing certain initial functions that is closed under the scheme of *weak safe composition* (*WSC* for short) and *safe nested recursion on notation* (*SNRN* for short). In [1] it has been shown that $\mathcal{N}$ coincides with the class of exponential time functions FEXP. Below we give a brief definition of the above mentioned term rewriting characterisation of $\mathcal{N}$. Essentially, all the equations defining the functions from $\mathcal{N}$ are oriented from left to right, resulting in an *infinite* set of rewrite rules $\mathcal{R}_\mathcal{N}$.

For $k,l \in \mathbb{N}$, the signature $\mathcal{F}$ underlying $\mathcal{R}_\mathcal{N}$ is partitioned into sets $\mathcal{F}^{k,l}$, collecting function symbols with k normal and l safe arguments. To express natural numbers, the constructor $\mathsf{0} \in \mathcal{F}^{0,0}$, and dyadic successors $\mathsf{S}_1,\mathsf{S}_2 \in \mathcal{F}^{0,1}$ are used. Terms formed from $\mathsf{0},\mathsf{S}_1$ and S_2 are called *numerals*. A numeral u encodes the natural number $\bar{u}$ as follows: $\bar{\mathsf{0}} := 0$, and $\bar{\mathsf{S}_i}(;u) := 2 \cdot \bar{u} + i$.

▶ **Definition 6.4.** The system $\mathcal{R}_\mathcal{N}$ contains the following rewrite rules, encoding the initial functions of $\mathcal{N}$:

$$
\begin{array}{lll}
\mathsf{O}^{k,l}(\vec{x};\vec{y}) \to \mathsf{0} & \text{for } k,l \in \mathbb{N} & \mathsf{P}(;\mathsf{0}) \to \mathsf{0} \\
\mathsf{I}^{k,l}_r(\vec{x};\vec{y}) \to x_r & \text{for } k,l \in \mathbb{N} \text{ and } r \in \{1,\dots,k\} & \mathsf{P}(;\mathsf{S}_i(;x)) \to x \quad \text{for } i \in \{1,2\} \\
\mathsf{I}^{k,l}_r(\vec{x};\vec{y}) \to y_{r-k} & \text{for } k,l \in \mathbb{N} \text{ and} & \mathsf{C}(;\mathsf{0},y_1,y_2) \to y_1 \\
 & r \in \{k+1,\dots,l+k\} & \mathsf{C}(;\mathsf{S}_i(;x),y_1,y_2) \to y_i \quad \text{for } i \in \{1,2\}
\end{array}
$$

Here $\vec{x} = x_1,\dots,x_k$ and $\vec{y} = y_1,\dots,y_l$ are supposed to be distinct variables.

Suppose the mapping safe is defined according to the definition of the rules above. Then each rule is oriented by an instance of EPO* regardless of the precedence used.

The scheme WSC is captured in the following definition.

▶ **Definition 6.5.** Suppose $g \in \mathcal{F}^{m,n}$ and $\vec{h} = h_1, \ldots, h_n \in \mathcal{F}^{k,l}$. Then for each sequence of indices $1 \leqslant i_1, \ldots, i_m \leqslant k$, the signature contains a fresh function symbol $\mathsf{SUB}[g, i_1, \ldots, i_m, h_1, \ldots, h_n] \in \mathcal{F}^{k,l}$. This symbol denotes the composition of functions g and $\vec{h}$ according to the rule

$$\mathsf{SUB}[g, i_1, \ldots, i_m, h_1, \ldots, h_n](\vec{x}; \vec{y}) \to g(x_{i_1}, \ldots, x_{i_m}; \vec{h}(\vec{x}; \vec{y})) \,.$$

Here we use $\vec{h}(\vec{x}; \vec{y})$ to abbreviate $h_1(\vec{x}; \vec{y}), \ldots, h_n(\vec{x}; \vec{y})$, and we use $\vec{x} = x_1, \ldots, x_k$ and $\vec{y} = y_1, \ldots, y_l$ for distinct variables.

Note that the above rule can be oriented by EPO*. For that we can employ any precedence that complies with $\mathsf{SUB}[g, i_1, \ldots, i_m, h_1, \ldots, h_n] > g, \vec{h}$. The scheme reflects that the class of exponential time functions is *not* closed under composition in general. However, we are allowed to substitute function calls $h_i(\vec{x}; \vec{y})$ in safe argument positions of g.

It remains to define the rules for the scheme SNRN. For that we make use of the following restriction of the lexicographic order.

▶ **Definition 6.6.** Let $\vec{u} = u_1, \ldots, u_n$ and $\vec{v} = v_1, \ldots, v_n$ be vectors of (possibly non-ground) numerals. We define $\vec{u} >^n_{\mathsf{lex}'} \vec{v}$ if there exists $k \in \{1, \ldots, n\}$ such that i) $u_1, \ldots, u_{k-1} = v_1, \ldots, v_{k-1}$, ii) u_k is a binary successor of v_k (i.e., $u_k = \mathsf{S}_i(; v_k)$ for some $i \in \{1, 2\}$), and iii) for each $j \in \{k+1, \ldots, n\}$ there exists $i \in \{1, \ldots, n\}$ such that $u_i = v_j$ or u_i is a binary successor of v_j.

Clearly the predecessor with respect to $>^n_{\mathsf{lex}'}$ is not unique. To precisely explain the relationship between arguments of the function and arguments replaced in recursive calls, we introduce the notion of a $>^n_{\mathsf{lex}'}$*-function* p.

The function p computes a suitable $>^n_{\mathsf{lex}'}$-predecessor of the normal arguments $\vec{u}$. We make use of the *type* $\tau(\vec{u})$ of $\vec{u}$, which is a ternary string over $\Sigma := \{0, 1, 2\}$: for single numeral, we set $\tau(\mathsf{0}) := 0$, $\tau(\mathsf{S}_1(; u)) := 1$ and $\tau(\mathsf{S}_2(; u)) := 2$. We extend τ to sequences of numerals $\tau(u_1, \ldots, u_n) := \tau(u_1) \cdots \tau(u_n)$.

Thus roughly, $\tau(\vec{u})$ corresponds to the vector of most significant bits of $\vec{u}$. Abusing notation, let $\mathsf{S}_0(; u)$ denote $\mathsf{0}$. Then $\vec{u}$ with type $\tau(\vec{u}) = w_1 \cdots w_n = w$ is expressible as $\mathsf{S}_{w_1}(; v_1), \ldots, \mathsf{S}_{w_n}(; v_n)$, or short $\mathsf{S}_w(; \vec{v})$, for some numerals $\vec{v} = v_1, \ldots, v_n$. The use of $>^n_{\mathsf{lex}'}$-functions relies on the following projection-function: for $n \in \mathbb{N}$ and $j \in \{1, \ldots, 2n\}$

$$J^n_j(u_1, \ldots, u_n) := \begin{cases} u_j & \text{if } 1 \leqslant j \leqslant n, \text{ and} \\ v & \text{if } n+1 \leqslant j \leqslant 2n \text{ and } u_{j-n} = \mathsf{S}_i(; v) \ (i \in \Sigma) \,. \end{cases}$$

Further, consider a function $p : \{1, \ldots, n\} \times \Sigma^n \to \{1, \ldots, 2n\}$. Based on p we extend the above function J^n, returning sequences of arguments as follows:

$$J^n_p(u_1, \ldots, u_n) := J^n_{p(1,\tau(\vec{u}))}(u_1, \ldots, u_n), \ldots, J^n_{p(n,\tau(\vec{u}))}(u_1, \ldots, u_n) \,.$$

Finally, the next definition provides the notion of a $>^n_{\mathsf{lex}'}$*-function* p.

▶ **Definition 6.7.** A function $p : \{1, \ldots, n\} \times \Sigma^n \to \{1, \ldots, 2n\}$ is called a $>^n_{\mathsf{lex}'}$*-function* if for all vectors of numerals $u_1, \ldots, u_n \neq \mathsf{0}, \ldots, \mathsf{0}$ we have $u_1, \ldots, u_n >_{\mathsf{lex}'} J^n_p(u_1, \ldots, u_n)$.

We complete the definition of $\mathcal{R}_\mathcal{N}$. We abbreviate $\Sigma^k \setminus \{\mathsf{0} \cdots \mathsf{0}\}$ as Σ^k_0.

▶ **Definition 6.8.** Suppose $g \in \mathcal{F}^{k,l}$ and $r_w, \vec{s_w}, \vec{t_w} \in \mathcal{F}^{k+k',l+1}$ for each type $w \in \Sigma^k_0$. Then for each triple $\vec{p} = p_1, p_2, p_3$ of $>^k_{\mathsf{lex}'}$-functions, the signature contains a fresh function symbol

$$\mathsf{f} = \mathsf{SNRN}_{\vec{p}}[g, [r_w \mid w \in \Sigma^k_0], [\vec{s_w} \mid w \in \Sigma^k_0], [\vec{t_w} \mid w \in \Sigma^k_0]] \in \mathcal{F}^{k+k',l} \,,$$

or more briefly $\mathsf{SNRN}_{\vec{p}}[g, r_w, \vec{s_w}, \vec{t_w}(w \in \Sigma_0^k)]$. This symbol denotes the function defined by SNRN according to the following set of rules (here the second rule is present for all $w \in \Sigma_0^k$).

$$\begin{aligned}\mathsf{f}(\vec{0}, \vec{x}; \vec{y}) &\to g(\vec{x}; \vec{y})\\ \mathsf{f}(\mathsf{S}_w(;\vec{z}), \vec{x}; \vec{y}) &\to r_w(\vec{v_1}, \vec{x}; \vec{y}, \mathsf{f}(\vec{v_1}, \vec{x}; \vec{s}_w(\vec{v_2}, \vec{x}; \vec{y}, \mathsf{f}(\vec{v_2}, \vec{x}; \vec{t}_w(\vec{v_3}, \vec{x}; \vec{y}, \mathsf{f}(\vec{v_3}, \vec{x}; \vec{y}))))))\end{aligned}$$

Here, $\vec{x}$, $\vec{y}$ and $\vec{z}$ are distinct variables and $\vec{v_i} = J^k_{p_i}(\mathsf{S}_w(\vec{;}z))$ $(i \in \{1,2,3\})$ are the predecessors of normal arguments as given by p_i.

The system $\mathcal{R}_\mathcal{N}$ consists of all the rules mentioned in Definition 6.4, Definition 6.5 and Definition 6.8. It is not difficult to see that for each function $f \in \mathcal{N}$, there is a *finite* restriction $\mathcal{R}(f) \subseteq \mathcal{R}_\mathcal{N}$ that computes the function f, c.f. [15]. Hence to prove our completeness theorem, it suffices to orient each finite restriction of $\mathcal{R}_\mathcal{N}$ by an instance of EPO*.

In the proof below, we use the following auxiliary function $\mathsf{h} : \mathcal{F} \to \mathbb{N}$ that computes the height of the definition tree of functions in $\mathcal{N}$. For $\vec{f} = f_1, \ldots, f_n$, we write $\max\{\mathsf{h}(\vec{f})\}$ instead of $\max\{\mathsf{h}(f_1), \ldots, \mathsf{h}(f_n)\}$.

$$\mathsf{h}(f) := \begin{cases} 0 & \text{if } f \in \{\mathsf{O}^{k,l}, \mathsf{I}^{k,l}_r, \mathsf{P}, \mathsf{C}, \mathsf{S}_1, \mathsf{S}_2, 0\},\\ 1 + \max\{\mathsf{h}(g), \max\{\mathsf{h}(\vec{h})\}\} & \text{if } f = \mathsf{SUB}[g, i_1, \ldots, i_m, \vec{h}],\\ 1 + \max\big\{\mathsf{h}(g), & \text{if } f = \mathsf{SNRN}_{\vec{p}}[g, r_w, \vec{s_w}, \vec{t_w}(w \in \Sigma_0^k)].\\ \quad \max\{\mathsf{h}(r_w), \max\{\mathsf{h}(\vec{s_w})\}, \max\{\mathsf{h}(\vec{t_w})\} \mid w \in \Sigma_0^k\}\big\} \end{cases}$$

▶ **Theorem 6.9** (Completeness). *Suppose $f \in$ FEXP. Then there exists a confluent, constructor TRS $\mathcal{R}(f)$ computing f that is compatible with some exponential path order $>_{\mathsf{epo}\star}$.*

Proof. Consider some arbitrary function $f \in \mathsf{FEXP}$ and the corresponding TRS $\mathcal{R}(f) \subseteq \mathcal{R}_\mathcal{N}$ computing f. Note that $\mathcal{R}(f)$ is a non-overlapping (hence confluent) constructor TRS. Let $\mathcal{F}(f)$ be the (finite) signature consisting of function symbols appearing in $\mathcal{R}(f)$.

For function symbols $g, h \in \mathcal{F}(f)$, we define the (admissible) precedence $>$ by setting $g > h$ if and only if $\mathsf{h}(g) > \mathsf{h}(h)$. Furthermore, define the safe mapping safe as indicated by the system $\mathcal{R}_\mathcal{N}$. We verify $\mathcal{R}(f) \subseteq >_{\mathsf{epo}\star}$ for $>_{\mathsf{epo}\star}$ induced by the precedence $>$ and the mapping safe. For brevity, we consider the most interesting case, the rules representing the schema SNRN, cf. Definition 6.8.

Abbreviate $\mathsf{SNRN}_{\vec{p}}[g, r_w, \vec{s_w}, \vec{t_w}(w \in \Sigma_0^k)]$ as f and fix some $w \in \Sigma_0^k$. Using Definition 3.2.2, employing $\mathsf{f} > g$, it is easy to check that $\mathsf{f}(\vec{0}, \vec{x}; \vec{y}) >_{\mathsf{epo}\star} g(\vec{x}; \vec{y})$ holds. We show

$$\mathsf{f}(\mathsf{S}_w(;\vec{z}), \vec{x}; \vec{y}) >_{\mathsf{epo}\star} r_w(\vec{v_1}, \vec{x}; \vec{y}, \mathsf{f}(\vec{v_1}, \vec{x}; \vec{s}_w(\vec{v_2}, \vec{x}; \vec{y}, \mathsf{f}(\vec{v_2}, \vec{x}; \vec{t}_w(\vec{v_3}, \vec{x}; \vec{y}, \mathsf{f}(\vec{v_3}, \vec{x}; \vec{y})))))) \quad (8)$$

First, consider the recursion-parameters $\vec{v_i} = J^k_{p_i}(\mathsf{S}_w(;\vec{z}))$ with $i \in \{1,2,3\}$. Let $\mathsf{S}_w(;\vec{z}) = \mathsf{S}_{w_1}(;z_1), \ldots, \mathsf{S}_{w_k}(;z_k)$ and let $\vec{v_i} = v_{i,1}, \ldots, v_{i,k}$. According to Definition 6.7 we have $\mathsf{S}_w(;\vec{z}) >^k_{\mathsf{lex}'} \vec{v_i}$. That is, there exists index $m \in \{1, \ldots, k\}$ such that $\mathsf{S}_{w_j}(;z_j) = v_{i,j}$ for $j \in \{1, \ldots, m-1\}$, $\mathsf{S}_{w_m}(;z_m)$ is a binary successor of $v_{i,m}$, and for $j \in \{m+1, \ldots, k\}$ there exists some $n \in \{1, \ldots, k\}$ with $\mathsf{S}_{w_n}(;z_n)$ equal to or a binary successor of $v_{i,j}$. This immediately gives

1) $\mathsf{S}_{w_j}(;z_j) \stackrel{s}{\sim} v_{i,j}$ for $j \in \{1, \ldots, m-1\}$,
2) $\mathsf{S}_{w_m}(;z_1) \sqsupset_{\mathsf{epo}\star} v_{i,m}$, and
3) $\mathsf{f}(\mathsf{S}_w(;\vec{z}), \vec{x}; \vec{y}) \sqsupset_{\mathsf{epo}\star} v_{i,j}$ for all $j \in \{m+1, \ldots, k\}$.

Clearly $\mathsf{f}(\mathsf{S}_w(;\vec{z}),\vec{x};\vec{y}) \sqsupseteq_{\mathsf{epo}\star} x_j$ for all $x_j \in \vec{x}$ by Definition 3.1, and similar Definition 3.2.1 gives $\mathsf{f}(\mathsf{S}_w(;\vec{z}),\vec{x};\vec{y}) >_{\mathsf{epo}\star} y_j$ for $y_j \in \vec{y}$. Using (1) – (3) with respect to $i = 3$, we conclude $\mathsf{f}(\mathsf{S}_w(;\vec{z}),\vec{x};\vec{y}) >_{\mathsf{epo}\star} \mathsf{f}(\vec{v_3},\vec{x};\vec{y})$ through an application of Definition 3.2.3.

Consider an arbitrary function symbol $t_{w,j} \in \vec{t_w}$. By definition, $\mathsf{f} > t_{w,j}$ in the precedence. Note that the above observations (1) – (3) imply $\mathsf{f}(\mathsf{S}_w(;\vec{z}),\vec{x};\vec{y}) \sqsupseteq_{\mathsf{epo}\star} v_{3,j}$ for all $v_{3,j} \in \vec{v_3}$. Further, using $\mathsf{f}(\mathsf{S}_w(;\vec{z}),\vec{x};\vec{y}) >_{\mathsf{epo}\star} y_j$ (for all $y_j \in \vec{y}$) and the above established inequality $\mathsf{f}(\mathsf{S}_w(;\vec{z}),\vec{x};\vec{y}) >_{\mathsf{epo}\star} \mathsf{f}(\vec{v_3},\vec{x};\vec{y})$ we see that $\mathsf{f}(\mathsf{S}_w(;\vec{z}),\vec{x};\vec{y}) >_{\mathsf{epo}\star} \vec{t}_{w,j}(\vec{v_3},\vec{x};\vec{y},\mathsf{f}(\vec{v_3},\vec{x};\vec{y}))$ follows by Definition 3.2.2.

By instantiating observations (1) – (3) with $i = 1, 2$, and repeated application of Definition 3.2.3 and Definition 3.2.2 exactly as above, it is tedious but straight forward to prove (8). ◀

7 Conclusion

In this paper we present the *exponential path order* EPO*. Suppose a term rewrite system $\mathcal{R}$ is compatible with EPO*, then the runtime complexity of $\mathcal{R}$ is bounded from above by an exponential function. Further, EPO* is sound and complete for the class of functions computable in exponential time on a Turing machine. We have implemented EPO* in the complexity tool TcT.[1] TcT can automatically prove exponential runtime complexity of our motivating example $\mathcal{R}_{\mathsf{fib}}$. Due to Theorem 6.2 we thus obtain through an automatic analysis that the computation of the Fibonacci number is exponential.

Based on our characterisation of the class of exponential time function through the order EPO*, it is a natural question whether this approach easily generalises to any super-exponential function 2_k, for $k \in \mathbb{N}$. We studied the possiblity for such generalisations, for instance to the class of double-exponential time functions to some extent. We soon realised that any sound generalisation of EPO* to this class quickly becomes technically much more involved, if possible at all.

References

1 T. Arai and N. Eguchi. A New Function Algebra of EXPTIME Functions by Safe Nested Recursion. *TOCL*, 10(4):24:1–24:19, 2009.

2 T. Arai and G. Moser. Proofs of Termination of Rewrite Systems for Polytime Functions. In *Proc. of 15th FSTTCS*, volume 3821 of *LNCS*, pages 529–540, 2005.

3 M. Avanzini, N. Eguchi, and G. Moser. A Path Order for Rewrite Systems that Compute Exponential Time Functions. *CoRR*, cs/CC/1010.1128, 2010. Available at `http://www.arxiv.org/`.

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

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

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

7 P. Baillot, J.-Y. Marion, and S. R. D. Rocca. Guest editorial: Special Issue on Implicit Computational Complexity. *TOCL*, 10(4), 2009.

[1] See `http://cl-informatik.uibk.ac.at/software/tct/`, the experimental data for our implementation is available here: `http://cl-informatik.uibk.ac.at/software/tct/experiments/epostar`.

8 S. Bellantoni and S. Cook. A new Recursion-Theoretic Characterization of the Polytime Functions. *CC*, 2(2):97–110, 1992.

9 A. Ben-Amram, N. D. Jones, and L. Kristiansen. Linear, Polynomial or Exponential? Complexity Inference in Polynomial Time. In *Proc. of 4th CiE*, volume 5028 of *LNCS*, pages 67–76, 2008.

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

11 G. Bonfante, J.-Y. Marion, and J.-Y. Moyen. Quasi-interpretations: A Way to Control Resources. *TCS*, 2009. To appear.

12 C. Choppy, S. Kaplan, and M. Soria. Complexity Analysis of Term-Rewriting Systems. *TCS*, 67(2–3):261–282, 1989.

13 A. Cichon and P. Lescanne. Polynomial Interpretations and the Complexity of Algorithms. In *Proc. of 11th CADE*, volume 607 of *LNCS*, pages 139–147, 1992.

14 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, 2009.

15 N. Eguchi. A Lexicographic Path Order with Slow Growing Derivation Bounds. *MLQ*, 55(2):212–224, 2009.

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

17 M. F. Ferreira. *Termination of Term Rewriting. Well-foundedness, Totality and Transformations.* PhD thesis, University of Utrecht, Faculty for Computer Science, 1995.

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

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

20 D. C. Kozen. *Theory of Computation.* Springer Verlag, first edition, 2006.

21 D. Leivant. Stratified functional programs and computational complexity. In *Proc. of 20th POPL*, pages 325–333, 1993.

22 J.-Y. Marion. Analysing the implicit complexity of programs. *IC*, 183:2–18, 2003.

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

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

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

Modes of Convergence for Term Graph Rewriting

Patrick Bahr

Department of Computer Science, University of Copenhagen
Universitetsparken 1, 2100 Copenhagen, Denmark
paba@diku.dk

Abstract

Term graph rewriting provides a simple mechanism to finitely represent restricted forms of infinitary term rewriting. The correspondence between infinitary term rewriting and term graph rewriting has been studied to some extent. However, this endeavour is impaired by the lack of an appropriate counterpart of infinitary rewriting on the side of term graphs. We aim to fill this gap by devising two modes of convergence based on a partial order resp. a metric on term graphs. The thus obtained structures generalise corresponding modes of convergence that are usually studied in infinitary term rewriting. We argue that this yields a common framework in which both term rewriting and term graph rewriting can be studied. In order to substantiate our claim, we compare convergence on term graphs and on terms. In particular, we show that the resulting infinitary calculi of term graph rewriting exhibit the same correspondence as we know it from term rewriting: Convergence via the partial order is a conservative extension of the metric convergence.

1998 ACM Subject Classification F.4.2, F.1.1

Keywords and phrases term graphs, partial order, metric, infinitary rewriting, graph rewriting

Digital Object Identifier 10.4230/LIPIcs.RTA.2011.139

Category Regular Research Paper

Introduction

Infinitary term rewriting [15] extends the theory of term rewriting by giving a meaning to transfinite reductions instead of dismissing them as undesired and meaningless artifacts. *Term graphs*, on the other hand, allow to explicitly represent and reason about sharing and recursion [2] by dropping the restriction to a tree structure that we have for terms. Apart from that, term graphs also provide a finite representation of certain infinite terms, viz. *rational terms*. As Kennaway et al. [14, 16] have shown, this can be leveraged in order to finitely represent restricted forms of infinitary term rewriting using *term graph rewriting*.

However, in order to benefit from this, we need to know for which class of term rewriting systems the set of rational terms is closed under (normalising) reductions. One such class of systems – a rather restrictive one – is the class of *regular equation systems* [9] which consist of rules having only constants on their left-hand side. Having an understanding of infinite reductions over term graphs could help to investigate closure properties of rational terms in the setting of infinitary term rewriting.

By studying infinitary calculi of term graph rewriting, we can also expect to better understand calculi with explicit sharing and/or recursion. Due to the lack of finitary confluence of these systems, Ariola and Blom [1] resort to a notion of skew confluence in order to be able to define infinite normal forms. An appropriate infinitary calculus could provide a direct approach to define infinite normal forms.

22nd International Conference on Rewriting Techniques and Applications (RTA'11).
Editor: M. Schmidt-Schauß; pp. 139–154



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

In this paper, we devise a partial order on term graphs generalising the partial order that is employed to formalise convergence in infinitary term rewriting [6]. We show that the partial order forms a complete semilattice on term graphs. Equipped with this, we shall formalise an infinitary calculus of term graph rewriting.

Historically, the theory of infinitary term rewriting is, however, mostly based on the metric space of terms [3]. Its notion of convergence captures "well-behaved" transfinite reductions. In order to replicate this on term graphs, we derive from the partial order a complete metric on term graphs generalising the metric on terms. Similar to the term rewriting case [6], we show that the metric calculus of infinitary term graph rewriting is the *total fragment* of the partial order calculus of infinitary term graph rewriting.

To our knowledge, this is the very first formalisation of infinitary term graph rewriting. We illustrate the adequacy of our formalisation as well as its relation to rational term rewriting on a number of examples. Due to space constraints not all proofs are given here. Full proofs of all theorems in this paper can be found in the author's master's thesis [4].

1 Infinitary Term Rewriting

We assume the reader to be familiar with the basic theory of ordinal numbers, orders and topological spaces [12], as well as term rewriting [18]. In the following, we give a brief outline of infinitary rewriting on terms [15, 6].

Given two sequences S, T, we write $S \cdot T$ to denote their concatenation and $S \leq T$ (resp. $S < T$) if S is a (proper) prefix of T. For a set A, we write A^* to denote the set of finite sequences over A. For a finite sequence $(a_i)_{i<n} \in A^*$, we also write $\langle a_0, a_1, \ldots, a_{n-1}\rangle$. In particular, $\langle\rangle$ denotes the empty sequence.

We consider the set of (possibly infinite) *terms* $\mathcal{T}^\infty(\Sigma, \mathcal{V})$ over a signature Σ and a set of variables $\mathcal{V}$. A *signature* Σ is a countable set of symbols. Each symbol f is associated with its arity $\mathsf{ar}(f)$, and we write $\Sigma^{(n)}$ for the set of symbols in Σ which have arity n.

Kennaway [13] and Bahr [5] investigated abstract models of infinitary rewriting based on metric spaces resp. partially ordered sets. Both models have been applied to term rewriting [15, 6, 8]. In the following, we summarise the resulting theory of infinitary term rewriting.

The metric $\mathbf{d}$ on terms that is used in this setting is 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* [3]. A metric $\mathbf{d}$ is called an *ultrametric* if it satisfies the stronger triangle inequality $\mathbf{d}(x,z) \leq \max\{\mathbf{d}(x,y), \mathbf{d}(y,z)\}$; it is called *complete* if each of its non-empty Cauchy sequences converges.

A *transfinite reduction* in a term rewriting system $\mathcal{R}$, i.e. a transfinite sequence $(t_\iota \to_\mathcal{R} t_{\iota+1})_{\iota<\alpha}$ of rewriting steps in $\mathcal{R}$, is said to *m-converge* to t iff the sequence of terms $(t_\iota)_{\iota<\widehat{\alpha}}$ is continuous, i.e. $\lim_{\iota\to\lambda} t_\iota = t_\lambda$ for each limit ordinal $\lambda < \alpha$, and $(t_\iota)_{\iota<\widehat{\alpha}}$ converges to t, i.e. $\lim_{\iota\to\widehat{\alpha}} t_\iota = t$, where $\widehat{\alpha} = \alpha$ if α is a limit ordinal and $\widehat{\alpha} = \alpha + 1$ otherwise.

▶ **Example 1.1.** Consider the term rewriting system $\mathcal{R}$ containing the rule $a : x \to b : a : x$, where $:$ is a binary symbol that we write infix and assume to associate to the right. That is, the right-hand side of the rule is parenthesised as $b : (a : x)$. Think of the $:$ symbol as the list constructor *cons*. In $\mathcal{R}$, we have the infinite reduction sequence

$$S\colon\ a : c \to b : a : c \to b : b : a : c \to \ldots$$

The position at which two consecutive terms differ moves deeper and deeper during the reduction S. Hence, S m-converges to the infinite term s satisfying the equation $s = b : s$, i.e. $s = b : b : b : \ldots$.

The partial order $\leq_\bot$ is defined on *partial terms*, i.e. terms over signature $\Sigma_\bot = \Sigma \uplus \{\bot\}$, with $\bot$ a nullary symbol. It is characterised as follows: $s \leq_\bot t$ iff t can be obtained from s by replacing each occurrence of $\bot$ by some partial term. The pair $(\mathcal{T}^\infty(\Sigma_\bot, \mathcal{V}), \leq_\bot)$ forms a *complete semilattice* [10]. A partially ordered set $(A, \leq)$ is called a *complete partial order* (*cpo*) if it has a *least element* and every *directed subset* D of A has a *least upper bound* (*lub*) $\bigsqcup D$ in A. If, additionally, every *non-empty* subset B of A has a *greatest lower bound* (*glb*) $\bigsqcap B$, then $(A, \leq)$ is called a *complete semilattice*. This means that for complete semilattices the *limit inferior* $\liminf_{\iota\to\alpha} a_\iota = \bigsqcup_{\beta<\alpha} \left(\bigsqcap_{\beta\leq\iota<\alpha} a_\iota\right)$ of a sequence $(a_\iota)_{\iota<\alpha}$ is always defined.

In the partial order model of infinitary rewriting, convergence is modelled by the limit inferior: A transfinite reduction $(t_\iota \to_\mathcal{R} t_{\iota+1})_{\iota<\alpha}$ of *partial terms* in $\mathcal{R}$ is said to *p-converge* to t if it is continuous in the sense that $\liminf_{\iota<\lambda} t_\iota = t_\lambda$ for each limit ordinal $\lambda < \alpha$, and $\liminf_{\iota<\widehat{\alpha}} t_\iota = t$. The distinguishing feature of this model is that, given a complete semilattice, each continuous reduction also converges. This provides a conservative extension to m-convergence that allows rewriting modulo *meaningless terms* [6] by essentially mapping those parts of the reduction to $\bot$ that are divergent according to the metric model.

Intuitively, p-convergence on terms describes an approximation process. To this end, the partial order $\leq_\bot$ captures a notion of *information preservation*, i.e. $s \leq_\bot t$ iff t contains at least the same information as s does but potentially more. A monotonic sequence of terms $t_0 \leq_\bot t_1 \leq_\bot \dots$ thus approximates the information contained in $\bigsqcup_{i<\omega} t_i$. Given this reading of $\leq_\bot$, the glb $\bigsqcap T$ of a set of terms T captures the common (non-contradicting) information of the terms in T. Leveraging this, a sequence that is not necessarily monotonic can be turned into a monotonic sequence $t_j = \bigsqcap_{j\leq i<\omega} s_j$ such that each t_j contains exactly the information that remains stable in $(s_i)_{i<\omega}$ from j onwards. Hence, the limit inferior $\liminf_{i\to\omega} s_i = \bigsqcup_{j<\omega} \bigsqcap_{j\leq i<\omega} s_i$ is the term that contains the accumulated information that eventually remains stable in $(s_i)_{i<\omega}$. This is expressed as an approximation of the monotonically increasing information that remains stable from some point on.

▶ **Example 1.2.** Reconsider the system from Example 1.1. The reduction S also p-converges to s. Its sequence of stable information $\bot : \bot \leq_\bot b : \bot : \bot \leq_\bot b : b : \bot : \bot \leq_\bot \dots$ approximates s. Now consider the system with the *additional* rule $b : x \to a : b : x$. Starting with the same term, but applying the two rules alternately at the root, we obtain the reduction sequence

$$T\colon\ a : c \to b : a : c \to a : b : a : c \to b : a : b : a : c \to \dots$$

Now the differences between two consecutive terms occur right below the root symbol "$:$". Hence, T does not m-converge. This, however, only affects the left argument of "$:$". Following the right argument position, the bare list structure becomes eventually stable. The sequence of stable information $\bot : \bot \leq_\bot \bot : \bot : \bot \leq_\bot \bot : \bot : \bot : \bot \leq_\bot \dots$ approximates the term $t = \bot : \bot : \bot : \dots$. Hence, T p-converge to t.

The relation between m- and p-convergence illustrated in the examples above is characteristic: p-convergence is a conservative extension of m-convergence [5]. A reduction m-converges to a term t iff it totally p-converges to t, i.e. over terms without $\bot$. The goal of this paper is to generalise both the metric and the partial order on terms to term graphs while maintaining the properties presented here in order to instantiate the abstract models of infinitary rewriting and thereby obtain models for infinitary term graph rewriting.

2 Term Graphs

The notion of term graphs we are using is taken from Barendregt et al. [7]. Also our generalised notion of homomorphisms, which is crucial for the definition of the partial order

on term graphs, follows the general idea of Barendregt et al.

▶ **Definition 2.1.** Let Σ be a signature. A Σ-*graph* (or simply *graph*) is a tuple $g = (N, \mathsf{lab}, \mathsf{suc})$ consisting of a set N (of *nodes*), a *labelling function* $\mathsf{lab}\colon N \to \Sigma$, and a *successor function* $\mathsf{suc}\colon N \to N^*$ such that $|\mathsf{suc}(n)| = \mathsf{ar}(\mathsf{lab}(n))$ for each node $n \in N$, i.e. a node labelled with a k-ary symbol has precisely k successors. If $\mathsf{suc}(n) = \langle n_0, \dots, n_{k-1}\rangle$, then we write $\mathsf{suc}_i(n)$ for n_i. Moreover, we use the abbreviation $\mathsf{ar}_g(n)$ for the arity $\mathsf{ar}(\mathsf{lab}(n))$ of n.

▶ **Definition 2.2.** Let $g = (N, \mathsf{lab}, \mathsf{suc})$ be a Σ-graph and $n, n' \in N$.

(i) A *path* in g from n to n' is a finite sequence $(p_i)_{i<l}$ in $\mathbb{N}$ such that either
- $n = n'$ and $(p_i)_{i<l}$ is empty, i.e. $l = 0$, or
- $0 \le p_0 < \mathsf{ar}_g(n)$ and the suffix $(p_i)_{1\le i<l}$ is a path in g from $\mathsf{suc}_{p_0}(n)$ to n'.

(ii) If there exists a path from n to n' in g, we say that n' is *reachable* from n in g.

▶ **Definition 2.3.** Given a signature Σ, a *term graph* g over Σ is a tuple $(N, \mathsf{lab}, \mathsf{suc}, r)$ consisting of an *underlying* Σ-graph $(N, \mathsf{lab}, \mathsf{suc})$ whose nodes are all reachable from the *root node* $r \in N$. The class of all term graphs over Σ is denoted $\mathcal{G}^\infty(\Sigma)$. We use the notation N^g, lab^g, suc^g and r^g to refer to the respective components N,lab, suc and r of g.

Paths in a graph are not absolute but relative to a starting node. In term graphs, however, we have a distinguished root node from which each node is reachable. Paths relative to the root node are central for dealing with term graphs:

▶ **Definition 2.4.** Let $g \in \mathcal{G}^\infty(\Sigma)$ and $n \in N$.

(i) An *occurrence* of n is a path in the underlying graph of g from r^g to n. The set of all occurrences in g is denoted $\mathcal{P}(g)$; the set of all occurrences of n in g is denoted $\mathcal{P}_g(n)$.[1]

(ii) The *depth* of n in g, denoted $\mathsf{depth}_g(n)$, is the minimum of the lengths of the occurrences of n in g, i.e. $\mathsf{depth}_g(n) = \min\{|\pi| \mid \pi \in \mathcal{P}_g(n)\}$.

(iii) Let $\Delta \subseteq \Sigma$. The Δ-*depth* of g, denoted $\Delta\text{-}\mathsf{depth}(g)$, is the minimal depth of a Δ-node, i.e., a node labelled with a symbol in Δ, or ∞ if no such node exists in g:

$$\Delta\text{-}\mathsf{depth}(g) = \min\left\{\mathsf{depth}_g(n) \,\middle|\, n \in N, \mathsf{lab}^g(n) \in \Delta\right\} \cup \{\infty\}$$

If Δ is a singleton set $\{\sigma\}$, we also write $\sigma\text{-}\mathsf{depth}(g)$ instead of $\{\sigma\}\text{-}\mathsf{depth}(g)$.

(iv) For an occurrence $\pi \in \mathcal{P}(g)$, we write $\mathsf{node}_g(\pi)$ for the unique node $n \in N^g$ with $\pi \in \mathcal{P}_g(n)$ and $g(\pi)$ for its symbol $\mathsf{lab}^g(n)$.

(v) An occurrence $\pi \in \mathcal{P}(g)$ is called *cyclic* if there are paths $\pi_1 < \pi_2 \le \pi$ with $\mathsf{node}_g(\pi_1) = \mathsf{node}_g(\pi_2)$. The non-empty path π' with $\pi_1 \cdot \pi' = \pi_2$ is then called a *cycle* of $\mathsf{node}_g(\pi_1)$. An occurrence that is not cyclic is called *acyclic*.

(vi) The term graph g is called a *term tree* if each node in g has exactly one occurrence.

Note that the labelling function of graphs – and thus term graphs – is *total*. In contrast, Barendregt et al. [7] considered *open* (term) graphs with a *partial* labelling function such that unlabelled nodes denote holes or variables. This is reflected in their notion of homomorphisms in which the homomorphism condition is suspended for unlabelled nodes.

Instead of a partial labelling function, we chose a *syntactic* approach that is closer to the representation in terms: Variables, holes and "bottoms" are represented as distinguished

[1] The notion/notation of occurrences is borrowed from terms: Every occurrence π of a node n corresponds to the subterm represented by n occurring at position π in the unravelling of the term graph to a term.

syntactic entities. We achieve this on term graphs by making the notion of homomorphisms dependent on a distinguished set of constant symbols Δ for which the homomorphism condition is suspended:

▶ **Definition 2.5.** Let Σ be a signature, $\Delta \subseteq \Sigma^{(0)}$, and $g, h \in \mathcal{G}^\infty(\Sigma)$.

(i) A function $\phi\colon N^g \to N^h$ is called *homomorphic* in $n \in N^g$ if the following holds:

$$\mathsf{lab}^g(n) = \mathsf{lab}^h(\phi(n)) \qquad \text{(labelling)}$$
$$\phi(\mathsf{suc}^g_i(n)) = \mathsf{suc}^h_i(\phi(n)) \quad \text{for all } 0 \leq i < \mathsf{ar}_g(n) \qquad \text{(successor)}$$

(ii) A Δ-*homomorphism* ϕ from g to h, denoted $\phi\colon g \to_\Delta h$, is a function $\phi\colon N^g \to N^h$ that is homomorphic in n for all $n \in N^g$ with $\mathsf{lab}^g(n) \notin \Delta$ and satisfies $\phi(r^g) = r^h$.

It should be obvious that we get the usual notion of homomorphisms on term graphs if $\Delta = \emptyset$. The Δ-nodes can be thought of as holes in the term graphs which can be filled with other term graphs. For example, if we have a distinguished set of variable symbols $\mathcal{V} \subseteq \Sigma^{(0)}$, we can use $\mathcal{V}$-homomorphisms to formalise the matching step of term graph rewriting which requires the instantiation of variables.

▶ **Proposition 2.6.** *The Δ-homomorphisms on $\mathcal{G}^\infty(\Sigma)$ form a category which is a preorder. That is, there is at most one Δ-homomorphism from one term graph to another.*

As a consequence, each Δ-homomorphism is both monic and epic, and whenever there are two Δ-homomorphisms $\phi\colon g \to_\Delta h$ and $\psi\colon h \to_\Delta g$, they are inverses of each other, i.e. Δ-*isomorphisms*. If two term graphs are Δ-*isomorphic*, we write $g \cong_\Delta h$.

Note that injectivity is in general different from both being monic and the existence of left-inverses. The same holds for surjectivity and being epic resp. having right-inverses. However, each Δ-homomorphism is a Δ-isomorphism iff it is bijective.

For the two special cases $\Delta = \emptyset$ and $\Delta = \{\sigma\}$, we write $\phi\colon g \to h$ resp. $\phi\colon g \to_\sigma h$ instead of $\phi\colon g \to_\Delta h$ and call ϕ a homomorphism resp. σ-homomorphism. The same convention applies to Δ-isomorphisms.

3 Canonical Term Graphs

In this section, we introduce a canonical representation of isomorphism classes of term graphs. We use a well-known trick to achieve this [17]. As we shall see at the end of this section, this will also enable us to construct term graphs modulo isomorphism very easily.

▶ **Definition 3.1.** A term graph g is called *canonical* if $n = \mathcal{P}_g(n)$ holds for each $n \in N^g$. That is, each node is the set of its occurrences in the term graph. The set of all canonical term graphs over Σ is denoted $\mathcal{G}^\infty_\mathcal{C}(\Sigma)$.

This structure allows a convenient characterisation of Δ-homomorphisms:

▶ **Lemma 3.2.** *For $g, h \in \mathcal{G}^\infty_\mathcal{C}(\Sigma)$, a function $\phi\colon N^g \to N^h$ is a Δ-homomorphism $\phi\colon g \to_\Delta h$ iff the following holds for all $n \in N^g$:*

(a) $n \subseteq \phi(n)$, *and* *(b)* $\mathsf{lab}^g(n) = \mathsf{lab}^h(\phi(n))$ *whenever* $\mathsf{lab}^g(n) \notin \Delta$.

Proof. Straightforward. ◀

By Proposition 2.6, there is at most one Δ-homomorphism between two term graphs. The lemma above uniquely defines this Δ-homomorphism: If there is a Δ-homomorphism from g to h, it is defined by $\phi(n) = n'$, where n' is the unique node $n' \in N^h$ with $n \subseteq n'$.

▶ **Remark 3.3.** Note that the lemma above is also applicable to non-canonical term graphs. It simply has to be rephrased such that instead of just referring to a node n, its set of occurrences $\mathcal{P}_g(n)$ is referred to whenever the "inner structure" of n is used.

The set of nodes in a canonical term graph forms a partition of the set of occurrences. Hence, it defines an equivalence relation on the set of occurrences. For a canonical term graph g, we write $\sim_g$ for this equivalence relation on $\mathcal{P}(g)$. According to Remark 3.3, we can extend this to arbitrary term graphs: $\pi_1 \sim_g \pi_2$ iff $\mathsf{node}_g(\pi_1) = \mathsf{node}_g(\pi_2)$. The characterisation of Δ-homomorphisms can thus be recast to obtain the following lemma that characterises the *existence* of Δ-homomorphisms:

▶ **Lemma 3.4.** *Given $g, h \in \mathcal{G}^\infty(\Sigma)$, there is a Δ-homomorphism $\phi\colon g \to_\Delta h$ iff, for all $\pi, \pi' \in \mathcal{P}(g)$, we have*

$$\textit{(a)}\ \pi \sim_g \pi' \implies \pi \sim_h \pi',\ \textit{and} \qquad \textit{(b)}\ g(\pi) = h(\pi) \quad \textit{whenever} \quad g(\pi) \notin \Delta.$$

Intuitively, (a) means that h has at least as much sharing of nodes as g has, whereas (b) means that h has at least the same non-Δ-symbols as g.

▶ **Corollary 3.5.** *Given $g, h \in \mathcal{G}^\infty(\Sigma)$, the following holds:*

(i) $\phi\colon N^g \to N^h$ is a Δ-isomorphism iff for all $n \in N^g$
(a) $\mathcal{P}_h(\phi(n)) = \mathcal{P}_g(n)$, and (b) $\mathsf{lab}^g(n) = \mathsf{lab}^h(\phi(n))$ or $\mathsf{lab}^g(n), \mathsf{lab}^h(\phi(n)) \in \Delta$.
(ii) $g \cong_\Delta h$ iff (a) $\sim_g = \sim_h$, and (b) $g(\pi) = h(\pi)$ or $g(\pi), h(\pi) \in \Delta$.

Proof. Immediate consequence of Lemma 3.2 resp. Lemma 3.4 and Proposition 2.6. ◀

From (ii) we immediately obtain the following equivalence:

▶ **Corollary 3.6.** *Given $g, h \in \mathcal{G}^\infty(\Sigma)$ and $\sigma \in \Sigma^{(0)}$, we have $g \cong h$ iff $g \cong_\sigma h$.*

Now we can revisit the notion of canonical term graphs using the above characterisation of Δ-isomorphisms. We will define a function $\mathcal{C}(\cdot)\colon \mathcal{G}^\infty(\Sigma) \to \mathcal{G}^\infty_\mathcal{C}(\Sigma)$ that maps a term graph to its canonical representation. To this end, let $g = (N, \mathsf{lab}, \mathsf{suc}, r)$ be a term graph and define $\mathcal{C}(g) = (N', \mathsf{lab}', \mathsf{suc}', r')$ as follows:

$$N' = \{\mathcal{P}_g(n) \mid n \in N\} \qquad r' = \mathcal{P}_g(r)$$
$$\mathsf{lab}'(\mathcal{P}_g(n)) = \mathsf{lab}(n) \qquad \mathsf{suc}'_i(\mathcal{P}_g(n)) = \mathcal{P}_g(\mathsf{suc}_i(n)) \qquad \text{for all } n \in N, 0 \le i < \mathsf{ar}_g(n)$$

$\mathcal{C}(g)$ is obviously a well-defined canonical term graph. With this definition we indeed capture the idea of a canonical representation of isomorphism classes:

▶ **Proposition 3.7.** *Given $g \in \mathcal{G}^\infty(\Sigma)$, the term graph $\mathcal{C}(g)$ canonically represents the equivalence class $[g]_\cong$. More precisely, it holds that*

$$\textit{(i)}\ [g]_\cong = [\mathcal{C}(g)]_\cong,\ \textit{and} \qquad \textit{(ii)}\ [g]_\cong = [h]_\cong \quad \textit{iff} \quad \mathcal{C}(g) = \mathcal{C}(h).$$

In particular, we have, for all canonical term graphs g, h, that $g = h$ iff $g \cong h$.

Proof. Straightforward consequence of Corollary 3.5. ◀

▶ **Remark 3.8.** Δ-homomorphisms can be naturally lifted to $\mathcal{G}^\infty(\Sigma)/_\cong$: We say that two Δ-homomorphisms $\phi\colon g \to_\Delta h$, $\phi'\colon g' \to_\Delta h'$, are isomorphic, written $\phi \cong \phi'$ iff there are isomorphisms $\psi_1\colon g \xrightarrow{\sim} g'$ and $\psi_2\colon h' \xrightarrow{\sim} h$ such that $\phi = \psi_2 \circ \phi' \circ \psi_1$. Given a Δ-homomorphism $\phi\colon g \to_\Delta h$ in $\mathcal{G}^\infty(\Sigma)$, $[\phi]_\cong\colon [g]_\cong \to_\Delta [h]_\cong$ is a Δ-homomorphism in $\mathcal{G}^\infty(\Sigma)/_\cong$. These Δ-homomorphisms then form a category which can easily be show to be isomorphic to the category of Δ-homomorphisms on $\mathcal{G}^\infty_\mathcal{C}(\Sigma)$ via the mapping $[\cdot]_\cong$.

Corollary 3.5 has shown that term graphs can be characterised up to isomorphism by only giving the equivalence $\sim_g$ and the labelling $g(\cdot)\colon\ \pi \mapsto g(\pi)$. This observation gives rise to the following definition:

▶ **Definition 3.9.** A *labelled quotient tree* over signature Σ is a triple $(P, l, \sim)$ consisting of a non-empty set $P \subseteq \mathbb{N}^*$, a function $l\colon\ P \to \Sigma$, and an equivalence relation $\sim$ on P that satisfies the following conditions for all $\pi, \pi' \in P$ and $i \in \mathbb{N}$:

$$\pi \cdot i \in P \quad \Longrightarrow \quad \pi \in P \quad \text{and} \quad i < \mathsf{ar}(l(\pi)) \qquad \text{(reachability)}$$

$$\pi \sim \pi' \quad \Longrightarrow \quad \begin{cases} l(\pi) = l(\pi') & \text{and} \\ \pi \cdot j \sim \pi' \cdot j & \text{for all } \ j < \mathsf{ar}(l(\pi)) \end{cases} \qquad \text{(congruence)}$$

The following lemma confirms that labelled quotient trees uniquely characterise any term graph up to isomorphism:

▶ **Lemma 3.10.** *Each term graph $g \in \mathcal{G}^\infty(\Sigma)$ induces a* canonical labelled quotient tree *$(\mathcal{P}(g), g(\cdot), \sim_g)$ over Σ. Vice versa, for each labelled quotient tree $(P, l, \sim)$ over Σ there is a unique canonical term graph $g \in \mathcal{G}^\infty_{\mathcal{C}}(\Sigma)$ whose canonical labelled quotient tree is $(P, l, \sim)$, i.e. $\mathcal{P}(g) = P$, $g(\pi) = l(\pi)$ for all $\pi \in P$, and $\sim_g\ = \sim$.*

Proof. The first part is trivial: $(\mathcal{P}(g), g(\cdot), \sim_g)$ satisfies the conditions from Definition 3.9.

Let $(P, l, \sim)$ be a labelled quotient tree. Define the term graph $g = (N, \mathsf{lab}, \mathsf{suc}, r)$ by

$$N = P/_\sim \qquad \mathsf{lab}(n) = f \quad \text{iff} \quad \exists \pi \in n.\ l(\pi) = f$$

$$r = n \quad \text{iff} \quad \langle\rangle \in n \qquad \mathsf{suc}_i(n) = n' \quad \text{iff} \quad \exists \pi \in n.\ \pi \cdot i \in n'$$

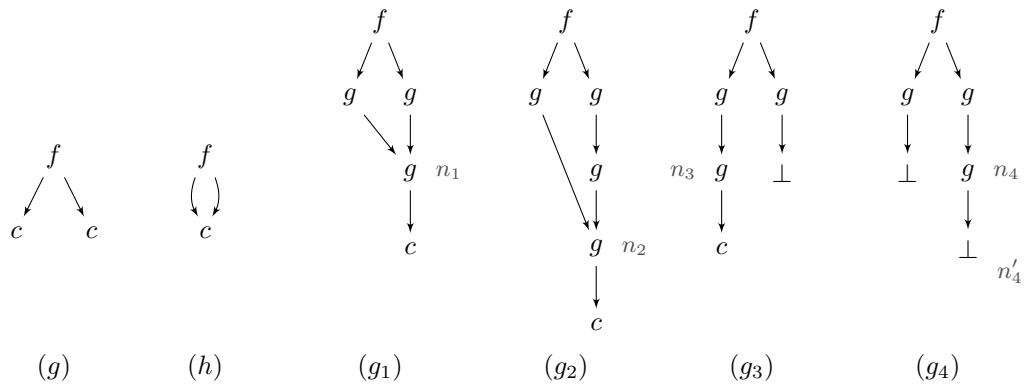
The functions lab and suc are well-defined due to the congruence condition satisfied by $(P, l, \sim)$. Since P is non-empty and closed under prefixes, it contains $\langle\rangle$. Hence, r is well-defined. Moreover, by the reachability condition, each node in N is reachable from the root node. An easy induction proof shows that $\mathcal{P}_g(n) = n$ for each node $n \in N$. Thus, g is a well-defined canonical term graph. The canonical labelled quotient tree of g is obviously $(P, l, \sim)$. Whenever there are two canonical term graphs with labelled quotient tree $(P, l, \sim)$, they are isomorphic due to Corollary 3.5 and, therefore, have to be identical by Proposition 3.7. ◀

Labelled quotient trees provide a valuable tool for constructing canonical term graphs. Nevertheless, the original graph representation remains convenient for practical purposes as it allows a straightforward formalisation of term graph rewriting and provides a finite representation of finite cyclic term graphs which induce an infinite labelled quotient tree.

Before we continue, it is instructive to make the correspondence between terms and term graphs clear. Note, that there is an obvious one-to-one correspondence between canonical term *trees* and terms. For example, the term tree g depicted in Figure 1a corresponds to the term $f(c, c)$. We thus consider the set of terms $\mathcal{T}^\infty(\Sigma)$ to be the subset of canonical term trees of $\mathcal{G}^\infty_{\mathcal{C}}(\Sigma)$. The *unravelling* of a term graph g is the unique term t such that there is a homomorphism $\phi\colon\ t \to g$. For example, g is the unravelling of h in Figure 1a. The unravelling of cyclic term graphs yields infinite terms, e.g. in Figure 3 on page 152, the term h_ω is the unravelling of the term graph g_2.

4 Partial Order on Term Graphs

In this section, we want to develop a partial order suitable for formalising convergence of a sequences of canonical term graphs similarly to p-convergence on terms.



(a) Term graphs $g \leq^1_\bot h$. **(b)** Term graphs g_1, g_2 with maximal lower bounds g_3, g_4 w.r.t. $\leq^3_\bot$.

Figure 1 Alternative partial orders on term graphs.

To get started, we use the correspondence between terms and canonical term trees, in order to characterise the partial order $\leq_\bot$ on $\mathcal{T}^\infty(\Sigma_\bot)$ via $\bot$-homomorphisms: Given $s, t \in \mathcal{T}^\infty(\Sigma_\bot)$, we have $s \leq_\bot t$ iff there is a $\bot$-homomorphism $\phi\colon s \to_\bot t$. The $\bot$-homomorphism formalises the intuition that t can be obtained from s by replacing occurrences of $\bot$ by terms. Let us generalise this to canonical term graphs: Given $g, h \in \mathcal{G}^\infty_\mathcal{C}(\Sigma_\bot)$, define $g \leq^1_\bot h$ iff there is a $\bot$-homomorphism $\phi\colon g \to_\bot h$. This definition indeed yields a complete semilattice $(\mathcal{G}^\infty_\mathcal{C}(\Sigma_\bot), \leq^1_\bot)$. Yet, as we will explain below, $\leq^1_\bot$ does not provide an adequate foundation for p-convergence on term graphs.

Recall that p-convergence on terms is based on the ability of the partial order $\leq_\bot$ to capture *information preservation* between terms. The limit inferior – and thus p-convergence – comprises the accumulated information that eventually remains stable. Following the approach on terms, a partial order $\leq^\mathcal{G}_\bot$ suitable as a basis for convergence for term graph rewriting, has to capture an appropriate notion of information preservation as well. However, term graphs encode an additional dimension of information through *sharing* of nodes, i.e. nodes with multiple occurrences. This rules out the straightforward partial order $\leq^1_\bot$ defined above. At first glance, $\bot$-homomorphisms capture information preservation as they allow to replace '$\bot$'s. Unfortunately, $\bot$-homomorphisms also allow to introduce sharing by mapping different nodes to the same target node: Considering the term graphs in Figure 1a, we have $g \leq^1_\bot h$, even though g and h contain contradicting information. Moreover, we get the counterintuitive situation that a total term graph such as g can be non-maximal w.r.t. $\leq^1_\bot$.

In order to avoid the introduction of sharing, we need to consider $\bot$-homomorphisms that preserve the structure of term graphs. Recall that by Lemma 3.10, the structure of a term graph is essentially given by the occurrences of nodes and their labelling. Labellings are already taken into consideration by $\bot$-homomorphisms. Thus, we can define a partial order $\leq^2_\bot$ that preserves the structure of term graphs by: $g \leq^2_\bot h$ iff there is a $\bot$-homomorphism $\phi\colon g \to_\bot h$ with $\mathcal{P}(\phi(n)) = \mathcal{P}(n)$ for all $n \in N^g$ with $\mathsf{lab}^g(n) \neq \bot$. While this would again yield a complete semilattice, it is unfortunately too restrictive. For example, we would not have $g|2 \leq^2_\bot g$ for the term graphs depicted in Figure 2a. The problem of $\leq^2_\bot$ is that it also considers sharing that originates from *below* a node. The fact that the node n (as well as r) has different occurrences in g and $g|2$ is solely caused by the edge from n to r that comes from *below* and thus closes a *cycle*. Even though the edge occurs below n and r, it affects their occurrences. Cutting off that edge, as in $g|2$, changes the sharing. As a

consequence, in the complete semilattice $(\mathcal{G}^\infty_\mathcal{C}(\Sigma_\bot), \leq^2_\bot)$, we do not obtain the intuitively expected convergence behaviour depicted in Figure 3c.

This observation suggests that we should only consider the *upward structure* of each node, ignoring the sharing that is caused by edges occurring *bellow* a node. By restricting our attention to *acyclic occurrences*, we can obtain the desired properties for a partial order on term graphs.

Recall that an occurrence π in a term graph g is called cyclic iff there are occurrences π_1, π_2 with $\pi_1 < \pi_2 \leq \pi$ such that $\mathsf{node}_g(\pi_1) = \mathsf{node}_g(\pi_2)$. Otherwise it is called acyclic. We will use the notation $\mathcal{P}^a(g)$ for the set of all acyclic occurrences in g, and $\mathcal{P}^a_g(n)$ for the set of all acyclic occurrences of a node n in g.

▶ **Definition 4.1.** Let Σ be a signature, $\Delta \subseteq \Sigma^{(0)}$ and $g, h \in \mathcal{G}^\infty(\Sigma)$ such that $\phi\colon g \to_\Delta h$.

(i) Given $n \in N^g$, ϕ is said to *preserve the sharing* of n if it satisfies the equation

$$\mathcal{P}^a_g(n) = \mathcal{P}^a_h(\phi(n)) \qquad \text{(preservation of sharing)}$$

(ii) ϕ is called *strong* if it preserves the sharing of all $n \in N^g$ with $\mathsf{lab}^g(n) \notin \Delta$.

▶ **Proposition 4.2.** *The strong Δ-homomorphisms on $\mathcal{G}^\infty(\Sigma)$ form a subcategory of the category of Δ-homomorphisms on $\mathcal{G}^\infty(\Sigma)$. Each Δ-isomorphism is a strong Δ-homomorphism.*

Proof. Straightforward. ◀

It is obvious from its definition that $\mathcal{P}^a_g(n)$ is the set of minimal elements of $\mathcal{P}_g(n)$ w.r.t. the prefix order. Strong $\bot$-homomorphisms thus preserve the upward structure of each non-$\bot$-node and, therefore, provide the desired structure for a partial order that captures information preservation on term graphs:

▶ **Definition 4.3.** For every $g, h \in \mathcal{G}^\infty(\Sigma_\bot)$, define $g \leq^\mathcal{G}_\bot h$ iff there is a strong $\bot$-homomorphism $\phi\colon g \to_\bot h$.

▶ **Proposition 4.4.** *The relation $\leq^\mathcal{G}_\bot$ is a partial order on $\mathcal{G}^\infty_\mathcal{C}(\Sigma_\bot)$.*

Proof. Reflexivity and transitivity of $\leq^\mathcal{G}_\bot$ follow immediately from Proposition 4.2. For antisymmetry, assume $g \leq^\mathcal{G}_\bot h$ and $h \leq^\mathcal{G}_\bot g$. By Proposition 2.6, this implies $g \cong_\bot h$. Corollary 3.6 then yields that $g \cong h$. Hence, according to Proposition 3.7, $g = h$. ◀

Similarly to Lemma 3.4, we can characterise strong Δ-homomorphisms by looking only at the occurrences' equivalence and labelling:

▶ **Lemma 4.5.** *Given $g, h \in \mathcal{G}^\infty(\Sigma)$, a Δ-homomorphism $\phi\colon g \to_\Delta h$ is strong iff*

$$\pi \sim_h \pi' \implies \pi \sim_g \pi' \quad \text{for all } \pi \in \mathcal{P}(g) \text{ with } g(\pi) \notin \Delta \text{ and } \pi' \in \mathcal{P}^a(h).$$

From this we can derive the following compact characterisation of $\leq^\mathcal{G}_\bot$:

▶ **Corollary 4.6.** *Let $g, h \in \mathcal{G}^\infty_\mathcal{C}(\Sigma_\bot)$. Then $g \leq^\mathcal{G}_\bot h$ iff the following conditions are met:*

(a) $\pi \sim_g \pi' \implies \pi \sim_h \pi'$ *for all* $\pi, \pi' \in \mathcal{P}(g)$
(b) $\pi \sim_h \pi' \implies \pi \sim_g \pi'$ *for all* $\pi \in \mathcal{P}(g)$ *with* $g(\pi) \in \Sigma$ *and* $\pi' \in \mathcal{P}^a(h)$
(c) $g(\pi) = h(\pi)$ *for all* $\pi \in \mathcal{P}(g)$ *with* $g(\pi) \in \Sigma$.

Proof. This follows immediately from Lemma 3.4 and Lemma 4.5. ◀

Note that for term trees (b) is always true and (a) follows from (c). Hence, on term trees, $\leq_\bot^{\mathcal{G}}$ can be characterised by (c). This shows that $\leq_\bot^{\mathcal{G}}$ restricted to canonical term trees is isomorphic to $\leq_\bot$ on terms. That is, $\leq_\bot^{\mathcal{G}}$ is indeed a generalisation of $\leq_\bot$ and we can use $\leq_\bot$ to refer to $\leq_\bot^{\mathcal{G}}$ without ambiguity, which we will do from now on.

▶ **Theorem 4.7.** *The pair $(\mathcal{G}_{\mathcal{C}}^{\infty}(\Sigma_\bot), \leq_\bot)$ forms a cpo.*

Proof (sketch). The least element of $\leq_\bot$ is obviously $\bot$. Assuming a directed subset G of $\mathcal{G}_{\mathcal{C}}^{\infty}(\Sigma_\bot)$, we define a canonical term graph $\overline{g}$ by giving a labelled quotient tree $(P, l, \sim)$ with

$$P = \bigcup_{g\in G} \mathcal{P}(g) \qquad \sim = \bigcup_{g\in G} \sim_g \qquad l(\pi) = \begin{cases} f & \text{if } f \in \Sigma \text{ and } \exists g \in G.\ g(\pi) = f \\ \bot & \text{otherwise} \end{cases}$$

From its construction it is easy to show that $(P, l, \sim)$ is a well-defined labelled quotient tree. Using the characterisation of $\leq_\bot$ provided by Corollary 4.6 one can then show that the thus defined term graph $\overline{g}$ is indeed the lub of G. ◀

For showing that $\leq_\bot$ is a complete semilattice, we use the following result from Kahn and Plotkin [11]:

▶ **Proposition 4.8.** *A cpo is a complete semilattice iff every pair of elements having an upper bound also has a least upper bound.*

This reduces the proof that $\leq_\bot$ is a complete semilattice to the following lemma:

▶ **Lemma 4.9.** *If $\{g_1, g_2\} \subseteq \mathcal{G}_{\mathcal{C}}^{\infty}(\Sigma_\bot)$ has an upper bound, then it has a least upper bound.*

Proof (sketch). Since $\{g_1, g_2\}$ is not necessarily directed, its lub might have occurrences that are neither in g_1 or g_2. Therefore, we have to employ a different construction here: Following Remark 3.8 we can define an order $\leq_\bot$ on $\mathcal{G}^{\infty}(\Sigma_\bot)/_{\cong}$ which is isomorphic to the order $\leq_\bot$ on $\mathcal{G}_{\mathcal{C}}^{\infty}(\Sigma_\bot)$: $[g]_\cong \leq_\bot [h]_\cong$ iff there is a strong $\bot$-homomorphism $\phi\colon\ g \to_\bot h$. Since the orders are isomorphic, showing the above property for the order on $\mathcal{G}^{\infty}(\Sigma_\bot)/_{\cong}$ suffices. To this end, we will construct a term graph $\overline{g}$ such that $[\overline{g}]_\cong$ is the lub of $\{[g_1]_\cong, [g_2]_\cong\}$.

Intuitively, $\overline{g}$ is constructed by forming the disjoint union of g_1 and g_2. For each occurrence π common to g_1 and g_2 the two nodes $\mathsf{node}_{g_1}(\pi)$ and $\mathsf{node}_{g_2}(\pi)$ are equated in $\overline{g}$. For the labelling of the resulting node, we prefer non-$\bot$-labels over $\bot$-labels.

Let $g_j = (N^j, \mathsf{suc}^j, \mathsf{lab}^j, r^j)$, $j = 1, 2$. As we are dealing with isomorphism classes, we can assume w.l.o.g. that nodes in g_j are of the form n^j for $j = 1, 2$. That is, given $\overline{M} = N^1 \cup N^2$ and $n^j \in \overline{M}$, we have $n^j \in N^k$ iff $j = k$. Hence, we can define a relation $\sim$ on $\overline{M}$ as follows:

$$n^j \sim m^k \quad \text{iff} \quad \mathcal{P}_{g_j}(n^j) \cap \mathcal{P}_{g_k}(m^k) \neq \emptyset$$

$\sim$ is clearly reflexive and symmetric. Hence, its transitive closure $\sim^+$ is an equivalence relation on $\overline{M}$. Now define the term graph $\overline{g} = (\overline{N}, \overline{\mathsf{lab}}, \overline{\mathsf{suc}}, \overline{r})$ as follows:

$$\overline{N} = \overline{M}/_{\sim^+} \qquad \overline{\mathsf{lab}}(N) = \begin{cases} f & \text{if } f \in \Sigma, \exists n^j \in N.\ \mathsf{lab}^j(n^j) = f \\ \bot & \text{otherwise} \end{cases}$$

$$\overline{r} = [r^1]_{\sim^+} \qquad \overline{\mathsf{suc}}_i(N) = N' \quad \text{iff} \quad \exists n^j \in N.\ \mathsf{suc}_i^j(n^j) \in N'$$

For the remainder of the proof it is crucial that $\{[g_1]_\cong, [g_2]_\cong\}$ has an upper bound. That is, there are two strong $\bot$-homomorphisms $\phi_j\colon\ g_j \to_\bot \widehat{g}$, $j = 1, 2$, for some term graph $\widehat{g}$.

It still remains to be shown that $\overline{g}$ is a well-defined term graph. Next it has to be shown that $[g_1]_\cong, [g_2]_\cong \leq_\bot [\overline{g}]_\cong$ by providing two strong $\bot$-homomorphisms $\psi_j\colon\ g_j \to_\bot \overline{g}$, $j = 1, 2$. And finally, to show that $[\overline{g}]_\cong$ is a lub, one has to construct a strong $\bot$-homomorphism $\psi\colon\ \overline{g} \to_\bot \widehat{g}'$ for each pair of strong $\bot$-homomorphisms $\phi'_j\colon\ g_j \to_\bot \widehat{g}'$, $j = 1, 2$. ◀

▶ **Theorem 4.10.** *The pair* $(\mathcal{G}^\infty_{\mathcal{C}}(\Sigma_\bot), \leq_\bot)$ *forms a complete semilattice.*

Proof. This is, by Proposition 4.8, a consequence of Theorem 4.7 and Lemma 4.9. ◀

5 Metric on Term Graphs

In this section, we want to derive a metric on canonical term graphs using the partial order $\leq_\bot$ introduced in the previous section. We will define this metric in a fashion similar to the metric on terms. All we need is an appropriate measure for the minimal depth of differences between two distinct term graphs. The partial order $\leq_\bot$ provides a tool for that as the glb $g \sqcap_\bot h$ of two term graphs g, h tells us on which parts g and h agree. The minimal depth at which g and h disagree is then simply the minimal depth of $\bot$-nodes in $g \sqcap_\bot h$:

▶ **Definition 5.1.** Given $g, h \in \mathcal{G}^\infty_{\mathcal{C}}(\Sigma)$ and any fresh nullary symbol $\bot \notin \Sigma$, the *similarity* $\mathsf{sim}(g, h)$ of g and h is the least depth of a $\bot$-node in $g \sqcap_\bot h$, i.e. $\bot\text{-}\mathsf{depth}(g \sqcap_\bot h)$. We define the distance function $\mathbf{d}$ on $\mathcal{G}^\infty_{\mathcal{C}}(\Sigma)$ by $\mathbf{d}(g, h) = 2^{-\mathsf{sim}(g,h)}$, where we interpret $2^{-\infty}$ as 0.

In order to show that $\mathbf{d}$ is a metric on $\mathcal{G}^\infty_{\mathcal{C}}(\Sigma)$, we use an idea similar to that of Arnold and Nivat [3]: We define the *truncation* $g|d$ of a term graph g at depth d, which removes certain nodes from g of depth at least d and fills the resulting holes with fresh $\bot$-nodes. This will provide an alternative characterisation of the metric $\mathbf{d}$ on term graphs.

▶ **Definition 5.2.** Let $g \in \mathcal{G}^\infty(\Sigma_\bot)$ and $d \in \mathbb{N}$.

(i) Given $n, m \in N^g$, m is an *acyclic predecessor* of n in g if there is an acyclic occurrence $\pi \cdot i \in \mathcal{P}^a_g(n)$ with $\pi \in \mathcal{P}_g(m)$. The set of acyclic predecessors of n in g is denoted $\mathsf{Pre}^a_g(n)$.

(ii) The set of *retained nodes* of g at d, denoted $N^g_{<d}$, is the least subset M of N^g satisfying the following conditions for all $n \in N^g$:

$$\text{(T1) } \mathsf{depth}_g(n) < d \implies n \in M \qquad \text{(T2) } n \in M \implies \mathsf{Pre}^a_g(n) \subseteq M$$

(iii) For each $n \in N^g$ and $i \in \mathbb{N}$, we use n^i to denote a fresh node, i.e. $\{n^i \mid n \in N^g, i \in \mathbb{N}\}$ is a set of pairwise distinct nodes not occurring in N^g. The set of *fringe nodes* of g at d, denoted $N^g_{=d}$, is defined as the singleton set $\{r^g\}$ if $d = 0$, and otherwise as the set

$$\left\{ n^i \;\middle|\; n \in N^g_{<d}, 0 \leq i < \mathsf{ar}_g(n) \text{ with } \begin{array}{rl} & \mathsf{suc}^g_i(n) \notin N^g_{<d} \\ \text{or} & \mathsf{depth}_g(n) \geq d-1, n \notin \mathsf{Pre}^a_g(\mathsf{suc}^g_i(n)) \end{array} \right\}$$

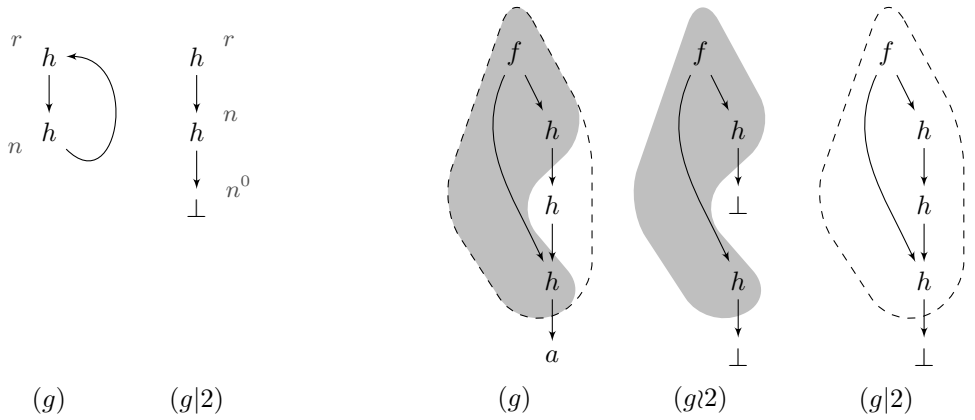
(iv) The *truncation* of g at d, denoted $g|d$, is the term graph defined by

$$N^{g|d} = N^g_{<d} \uplus N^g_{=d} \qquad r^{g|d} = r^g$$

$$\mathsf{lab}^{g|d}(n) = \begin{cases} \mathsf{lab}^g(n) & \text{if } n \in N^g_{<d} \\ \bot & \text{if } n \in N^g_{=d} \end{cases} \qquad \mathsf{suc}^{g|d}_i(n) = \begin{cases} \mathsf{suc}^g_i(n) & \text{if } n^i \notin N^g_{=d} \\ n^i & \text{if } n^i \in N^g_{=d} \end{cases}$$

Additionally, we define $g|\infty$ to be the term graph g itself.

Before discussing the intuition behind this definition of truncation, let us have a look at the rôle of retained and fringe nodes: The set of retained nodes $N^g_{<d}$ contains the nodes that are *preserved* by the truncation. All other nodes in $N^g \setminus N^g_{<d}$ are cut off. The "holes" that are thus created are filled by the fringe nodes in $N^g_{=d}$. This is expressed in the condition $\mathsf{suc}^g_i(n) \notin N^g_{<d}$ which, if satisfied, yields a fringe node n^i. That is, a fresh fringe node is inserted for each successor of a retained node that is not a retained node itself.



(a) Truncation of loops. **(b)** Comparison to strict truncation.

Figure 2 Examples of truncations.

But there is another circumstance that can give rise to a fringe node: If $\mathsf{depth}_g(n) \geq d-1$ and $n \notin \mathsf{Pre}^a_g(\mathsf{suc}^g_i(n))$, we also get a fringe node n^i. This condition is satisfied whenever an outgoing edge from a retained node closes a cycle. The lower bound for the depth is chosen such that a successor node of n is not necessarily retained node. An example is depicted in Figure 2a. For depth $d = 2$, the node n in the term graph g is just above the fringe, i.e. satisfies $\mathsf{depth}_g(n) \geq d - 1$. Moreover, it has an edge to the node r that closes a cycle. Hence, the truncation $g|2$ contains the fringe node n^0 which is now the 0-th successor of n.

We chose this admittedly complicated notion of truncation in order to make it "compatible" with the partial order $\leq_\bot$ and thus the metric $\mathbf{d}$: First of all, the truncation of a term graph is supposed to yield a smaller term graph w.r.t. $\leq_\bot$, viz. $g|d \leq_\bot g$. Hence, whenever a node is kept as a retained node, also its acyclic occurrences have to be kept in order to preserve its upward structure. To achieve this, with each node also its *acyclic ancestors* have to be retained. The closure condition (T2) is enforced exactly for this purpose.

To see this, consider Figure 2b. It shows a term graph g and its truncation at depth 2, once without the closure condition (T2), $g⟅2$, and once including (T2), $g|2$. The grey area highlights the nodes that are at depth smaller than 2, i.e. the nodes contained in $N^g_{<2}$ due to (T1) only. The nodes within the area surrounded by a dashed line are all the nodes in $N^g_{<2}$. One can observe that with the *strict truncation* $g⟅d$ without (T2), we do not have $g⟅2 \leq_\bot g$.

If the truncation construction is applied to term trees, then the result is also a term tree and is equal to the *truncation of terms* employed by Arnold and Nivat [3].

The most important property of the truncation of term graphs is that it allows the following alternative characterisation of similarity:

▶ **Proposition 5.3.** *Let* $g, h \in \mathcal{G}^\infty_\mathcal{C}(\Sigma)$. *Then* $\mathsf{sim}(g,h) = \max\{d \in \mathbb{N} \cup \{\infty\} \mid g|d \cong h|d\}$.

Apart from being indispensable in the subsequent proofs concerning the distance measure $\mathbf{d}$ on term graphs, the above proposition also reveals the close relationship to the metric $\mathbf{d}$ on terms which is essentially defined as characterised in the proposition above [3].

▶ **Proposition 5.4.** *The pair* $(\mathcal{G}^\infty_\mathcal{C}(\Sigma), \mathbf{d})$ *constitutes an ultrametric space.*

Proof. Using Proposition 5.3, the proof is the same as for the metric on terms [3]. ◀

With the following proposition we will be able to derive completeness of the metric space $(\mathcal{G}^\infty_\mathcal{C}(\Sigma), \mathbf{d})$ from the completeness of the semilattice $(\mathcal{G}^\infty_\mathcal{C}(\Sigma_\bot), \leq_\bot)$:

▶ **Proposition 5.5.** *Let $\Sigma_\bot$ be a signature and $(g_\iota)_{\iota<\alpha}$ a non-empty Cauchy sequence in the metric space $(\mathcal{G}^\infty_\mathcal{C}(\Sigma), \mathbf{d})$. Then $\lim_{\iota\to\alpha} g_\iota = \liminf_{\iota\to\alpha} g_\iota$.*

Proof (sketch). The term graph $\overline{g} = \liminf_{\iota\to\alpha} g_\iota$ is well-defined by Theorem 4.10. Since $(g_\iota)_{\iota<\alpha}$ is Cauchy, we obtain for each $d \in \mathbb{N}$ some $\beta < \alpha$ such that $g_\beta|d \cong g_\iota|d$ for each $\beta \leq \iota < \alpha$. From this we then obtain that for each $d \in \mathbb{N}$, there is some $\beta < \alpha$ such that $g_\beta|d \leq_\bot \overline{g}$. Hence, $\overline{g}$ is total, i.e. in $\mathcal{G}^\infty_\mathcal{C}(\Sigma)$. Moreover, $g_\beta|d \leq_\bot \overline{g}$ implies that $g_\beta|d \cong \overline{g}|d$. Therefore, we find for each $d \in \mathbb{N}$ some $\beta < \alpha$ with $\mathsf{sim}(\overline{g}, g_\beta) \geq 0$. Hence, we find for each $\varepsilon \in \mathbb{R}^+$ some $\beta < \alpha$ with $\mathbf{d}(\overline{g}, g_\beta) < \varepsilon$. That is, $(g_\iota)_{\iota<\alpha}$ converges to $\overline{g}$. ◀

▶ **Theorem 5.6.** *The metric space $(\mathcal{G}^\infty_\mathcal{C}(\Sigma), \mathbf{d})$ is complete.*

Proof. Immediate consequence of Proposition 5.5 and Theorem 4.10. ◀

Additionally, we can obtain that the notion of convergence provided by the partial order is a conservative extension of the one provided by the metric:

▶ **Proposition 5.7.** *Let $\Sigma_\bot$ be a signature, $(g_\iota)_{\iota<\alpha}$ a non-empty sequence in $\mathcal{G}^\infty_\mathcal{C}(\Sigma)$, and $\overline{g} = \liminf_{\iota\to\alpha} g_\iota$. If $\overline{g} \in \mathcal{G}^\infty_\mathcal{C}(\Sigma)$, then $\lim_{\iota\to\alpha} g_\iota = \overline{g}$.*

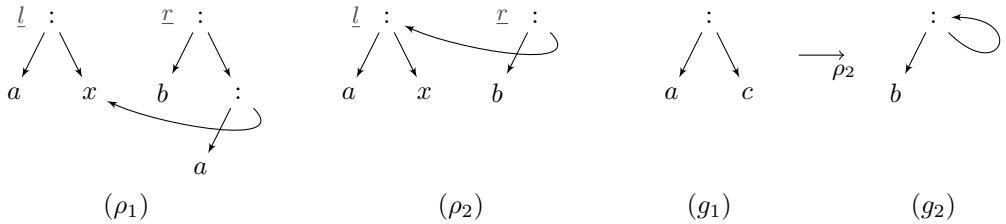
Proof (sketch). This can be derived from Proposition 5.5 by showing that $(g_\iota)_{\iota<\alpha}$ is Cauchy whenever $\overline{g} \in \mathcal{G}^\infty_\mathcal{C}(\Sigma)$: Assume that $(g_\iota)_{\iota<\alpha}$ is not Cauchy. Then we find some $d \in \mathbb{N}$ such that for each $\beta < \alpha$ there are $\beta \leq \gamma, \iota < \alpha$ with $\mathsf{sim}(g_\gamma, g_\iota) \leq d$, i.e. $\bot\text{-}\mathsf{depth}(g_\gamma \sqcap_\bot g_\iota) \leq d$. Let $h_\beta = \bigsqcap^\bot_{\beta\leq\iota<\alpha} g_\iota$. Since $\bot\text{-}\mathsf{depth}(h_\beta) \leq \bot\text{-}\mathsf{depth}(g_\gamma \sqcap_\bot g_\iota)$ for all $\beta \leq \gamma, \iota < \alpha$, we find some $d \in \mathbb{N}$ such that for each $\beta < \alpha$ there is some $\pi \in \mathcal{P}(h_\beta)$ with $|\pi| \leq d$ and $h_\beta(\pi) = \bot$. Because there are only finitely many relevant positions of length at most d, we thus obtain some position π^* such that for each $\beta < \alpha$ there is some $\beta \leq \gamma < \alpha$ with $h_\gamma(\pi^*) = \bot$. Since $(h_\iota)_{\iota<\alpha}$ is a $\leq_\bot$-chain, we know that $h_\beta(\pi^*) = \bot$ for any $\beta < \alpha$ with $\pi^* \in \mathcal{P}(h_\beta)$. But then we obtain that $\overline{g}(\pi^*) = \bot$, which contradicts the assumption that $\overline{g} \in \mathcal{G}^\infty_\mathcal{C}(\Sigma)$. ◀

6 Infinitary Term Graph Rewriting

Having obtained a complete semilattice and, from that, a complete metric, we can now instantiate the abstract models of infinitary rewriting [5] for term graphs. To this end, we adopt the term graph rewriting framework by Barendregt et al. [7].

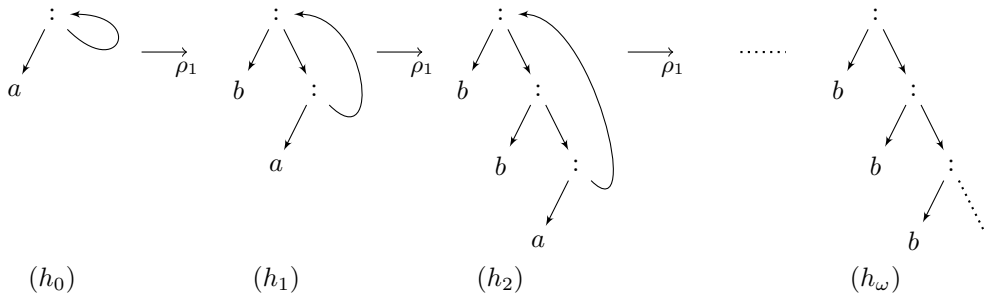
Without going into the details, a *term graph rewriting system* (GRS) $\mathcal{R}$ is a pair (Σ, R) consisting of a signature Σ and a set of rewrite rules R over Σ. A GRS $\mathcal{R}$ gives rise to a notion of rewriting steps $g \to_\mathcal{R} h$ on canonical term graphs. Figure 3a illustrates two term graph rules that both unravel to the term rule $a : x \to b : a : x$ from Example 1.1. A rule consists of a graph with two root nodes that represent the left- resp. right-hand side of the rule (indicated by $\underline{l}$ resp. $\underline{r}$). The right-hand side can refer to variables on the left-hand side only via sharing. This can occur as immediate sharing, i.e. by directly pointing to the variable as in ρ_1, or by mediate sharing as in ρ_2.

The application of a rewrite rule ρ (with root nodes l and r) to a term graph g is performed in four steps: At first a suitable sub-term graph of g rooted in some node n of g is *matched* against the left-hand side of ρ. This amounts to finding a $\mathcal{V}$-homomorphism ϕ from the term graph rooted in l to the sub-term graph rooted in n, the *redex*. Here, $\mathcal{V}$ is a set of variables. The $\mathcal{V}$-homomorphism ϕ thus replaces variables with term graphs. In the



(a) Term graph rules that unravel to $a : x \to b : a : x$.

(b) A single ρ_2-step.



(c) An m-convergent term graph reduction over ρ_1.

Figure 3 Term graph rules and their reductions.

second step, nodes and edges in ρ that are not reachable from l are copied into g, such that edges pointing to nodes in the term graph rooted in l are redirected to the image under ϕ. In the last two steps, all edges pointing to n are redirected to (the copy of) r and all nodes not reachable from the root of (the modified version of) g are removed. Examples for term graph rewriting steps are shown in Figure 3. We revisit them in more detail in Example 6.2 below.

▶ **Definition 6.1.** Let $\mathcal{R}$ be a GRS.

(i) A *transfinite reduction* in $\mathcal{R}$ is a sequence $(g_\iota \to_\mathcal{R} g_{\iota+1})_{\iota<\alpha}$ of rewriting steps in $\mathcal{R}$.

(ii) A transfinite reduction $S = (g_\iota \to_\mathcal{R} g_{\iota+1})_{\iota<\alpha}$ *m-converges* to $g \in \mathcal{G}^\infty_\mathcal{C}(\Sigma)$ in $\mathcal{R}$, written $S\colon\ g_0 \xrightarrow{m}_\mathcal{R} g$, if $(g_\iota)_{\iota<\widehat{\alpha}}$ is continuous and converges to g in the metric space.

(iii) Let $\mathcal{R}_\bot$ be the GRS $(\Sigma_\bot, R)$ over the extended signature $\Sigma_\bot$. A transfinite reduction $S = (g_\iota \to_{\mathcal{R}_\bot} g_{\iota+1})_{\iota<\alpha}$ *p-converges* to $g \in \mathcal{G}^\infty_\mathcal{C}(\Sigma_\bot)$ in $\mathcal{R}$, written $S\colon\ g_0 \xrightarrow{p}_\mathcal{R} g$, if $\liminf_{\iota<\lambda} g_i = g_\lambda$ for each limit ordinal $\lambda < \alpha$, and $\liminf_{\iota<\widehat{\alpha}} g_i = g$.

Note that we have to extend the signature of $\mathcal{R}$ to $\Sigma_\bot$ for the definition of p-convergence. However, we can obtain the total fragment of p-convergence if we restrict ourselves to total term graphs in $\mathcal{G}^\infty_\mathcal{C}(\Sigma)$: A transfinite reduction $(g_\iota \to_{\mathcal{R}_\bot} g_{\iota+1})_{\iota<\alpha}$ p-converging to g is called *total* if g as well as each g_ι is total, i.e. an element of $\mathcal{G}^\infty_\mathcal{C}(\Sigma)$.

▶ **Example 6.2.** Consider the term graph rule ρ_1 in Figure 3a that unravels to the term rule $a : x \to b : a : x$ from Example 1.1. Starting with the term tree $a : c$, depicted as g_1 in Figure 3b, we obtain the same transfinite reduction as in Example 1.1:

$$S\colon\ a : c \to_{\rho_1} b : a : c \to_{\rho_1} b : b : a : c \to_{\rho_1} \ \ldots\ h_\omega$$

Also in this setting, S both m- and p-converges to the term tree h_ω shown in Figure 3c. Similarly, we can reproduce the p-converging but not m-converging reduction T from Example 1.2. Notice that h_ω is a rational term tree as it can be obtained by unravelling the

finite term graph g_2 depicted in Figure 3b. In fact, if we use the rule ρ_2, we can immediately rewrite g_1 to g_2. In ρ_2, not only the variable x is shared but the whole left-hand side of the rule. This causes each redex of ρ_2 to be *captured* by the right-hand side.

Figure 3c indicates a transfinite reduction starting with a cyclic term graph h_0 that unravels to the rational term $t = a : t$. This reduction both m- and p-converges to the rational term tree h_ω as well. Again, by using ρ_2 instead of ρ_1, we can rewrite h_0 to the cyclic term graph g_2 in one step.

The following theorem shows that the total fragment of p-converging reductions is in fact equivalent to the m-converging reductions:

▶ **Theorem 6.3.** *Let S be a transfinite reduction in a GRS $\mathcal{R}_\perp$. Then*

$$S\colon g \overset{p}{\to}_{\mathcal{R}} h \text{ is total} \quad \text{iff} \quad S\colon g \overset{m}{\to}_{\mathcal{R}} h.$$

Proof. Follows straightforwardly from Proposition 5.7. ◀

An analogous result was also shown for infinitary term rewriting [6, 4]. In the setting of term rewriting, however, it also holds for the so-called strong convergence. The notion of convergence considered here is the weak convergence and we do not know whether the theorem above can be transferred to strong convergence as well.

7 Alternative Approaches and Future Work

While exhibiting the desired properties, the structures that we have investigated here seem quite intricate. This concerns both the partial order and the notion of truncation that provides an alternative characterisation for the metric. It is therefore advisable to further scrutinise these structures as well as possible alternatives.

The two partial orders $\leq^1_\perp$ and $\leq^2_\perp$, which we briefly discussed in Section 4, are not suited for formalising convergence as they capture too much sharing resp. too little. Instead, we took a middle ground, based on strong $\perp$-homomorphisms, yielding the order $\leq^{\mathcal{G}}_\perp$. However, injective $\perp$-homomorphisms provide a much more natural generalisation of strong $\perp$-homomorphisms: A $\perp$-homomorphism $\phi\colon g \to_\perp h$ is *injective* if $\phi(n) = \phi(m)$ implies $n = m$ for all non-$\perp$-nodes in g. Unfortunately, the thus obtained order $\leq^3_\perp$ has a quirk: In general, it does not even admit the glb of a finite number of term graphs. Figure 1b shows two term graphs with two maximal lower bounds w.r.t. $\leq^3_\perp$. Even though this means that $\leq^3_\perp$ does not provide a complete semilattice, it might still be appealing for other purposes, as it forms a cpo.

While we defined the metric $\mathbf{d}$ on term graphs using the glb induced by the partial order $\leq^{\mathcal{G}}_\perp$, we also provided a characterisation via the truncation $g|d$. We can take this as a starting point to define a metric in the style of Proposition 5.3 but with a simpler notion of truncation: Consider the *strict truncation* $g\wr d$, sketched in Figure 2b, that simply removes all nodes at depth d or below. Conceptionally, the thus induced metric $\mathbf{d}_\wr$ is considerably simpler. This is also manifested by its invariance under some minor changes to its definition: In the definition of the truncation $g|d$, we had to be very careful in defining the fringe nodes which have to have at most one predecessor and also have to be introduced for each edge at sufficient depth that closes a cycle. Changing these intricate details of the definition change the induced topology of the corresponding metric space. This is not the case for the metric $\mathbf{d}_\wr$. Regardless of how we deal with fringe nodes in the strict truncation, as long as they are labelled with $\perp$, the induced topology of the resulting metric space is the same. Moreover,

$(\mathcal{G}^{\infty}_{\mathcal{C}}(\Sigma), \mathbf{d}_{\wr})$ is also a complete ultrametric space. It is, however, unknown to us whether there is a complete semilattice that is compatible with it in the sense of Proposition 5.7.

Acknowledgement

I would like to thank Bernhard Gramlich for his constant support during the work on my master's thesis which made this work possible. I am also indebted to the anonymous referees whose comments and suggestions greatly helped to improve the presentation of this paper.

References

1 Z.M. Ariola and S. Blom. Skew confluence and the lambda calculus with letrec. *Annals of Pure and Applied Logic*, 117(1-3):95–168, 2002.

2 Z.M. Ariola and J.W. Klop. Lambda calculus with explicit recursion,. *Information and Computation*, 139(2):154 – 233, 1997.

3 A. Arnold and M. Nivat. The metric space of infinite trees. Algebraic and topological properties. *Fundamenta Informaticae*, 3(4):445–476, 1980.

4 P. Bahr. Infinitary rewriting - theory and applications. Master's thesis, Vienna University of Technology, Vienna, 2009.

5 P. Bahr. Abstract models of transfinite reductions. In C. Lynch, editor, *RTA 2010*, volume 6 of *LIPIcs*, pages 49–66. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2010.

6 P. Bahr. Partial order infinitary term rewriting and böhm trees. In C. Lynch, editor, *RTA 2010*, volume 6 of *LIPIcs*, pages 67–84. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2010.

7 H.P. Barendregt, M.C.J.D. van Eekelen, J.R.W. Glauert, R. Kennaway, M.J. Plasmeijer, and M.R. Sleep. Term graph rewriting. In Philip C. Treleaven Jaco de Bakker, A. J. Nijman, editor, *PARLE 1987*, volume 259 of *LNCS*, pages 141–158. Springer, 1987.

8 S. Blom. An approximation based approach to infinitary lambda calculi. In Vincent van Oostrom, editor, *RTA 2004*, volume 3091 of *LNCS*, pages 221–232. Springer, 2004.

9 B. Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25(2):95–169, 1983.

10 J.A. Goguen, J.W. Thatcher, E.G. Wagner, and J.B. Wright. Initial algebra semantics and continuous algebras. *Journal of the ACM*, 24(1):68–95, 1977.

11 G. Kahn and G.D. Plotkin. Concrete domains. *Theoretical Computer Science*, 121(1-2):187–277, 1993.

12 J.L. Kelley. *General Topology*, volume 27 of *Graduate Texts in Mathematics*. Springer-Verlag, 1955.

13 R. Kennaway. On transfinite abstract reduction systems. Technical report, CWI (Centre for Mathematics and Computer Science), Amsterdam, 1992.

14 R. Kennaway. Infinitary rewriting and cyclic graphs. *Electronic Notes in Theoretical Computer Science*, 2:153–166, 1995. SEGRAGRA '95.

15 R. Kennaway and F.-J. de Vries. Infinitary rewriting. In Terese [18], chapter 12, pages 668–711.

16 R. Kennaway, J.W. Klop, M.R. Sleep, and F.-J. de Vries. On the adequacy of graph rewriting for simulating term rewriting. *ACM Transactions on Programming Languages and Systems*, 16(3):493–523, 1994.

17 D. Plump. Term graph rewriting. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2, pages 3–61. World Scientific Publishing Co., Inc., 1999.

18 Terese. *Term Rewriting Systems*. Cambridge University Press, 1st edition, 2003.

Modular Termination Proofs of Recursive Java Bytecode Programs by Term Rewriting*

Marc Brockschmidt, Carsten Otto, and Jürgen Giesl

LuFG Informatik 2, RWTH Aachen University, Germany

Abstract

In [5, 15] we presented an approach to prove termination of non-recursive Java Bytecode (JBC) programs automatically. Here, JBC programs are first transformed to finite *termination graphs* which represent all possible runs of the program. Afterwards, the termination graphs are translated to term rewrite systems (TRSs) such that termination of the resulting TRSs implies termination of the original JBC programs. So in this way, existing techniques and tools from term rewriting can be used to prove termination of JBC automatically. In this paper, we improve this approach substantially in two ways:

(1) We extend it in order to also analyze *recursive* JBC programs. To this end, one has to represent call stacks of arbitrary size.

(2) To handle JBC programs with several methods, we *modularize* our approach in order to re-use termination graphs and TRSs for the separate methods and to prove termination of the resulting TRS in a modular way.

We implemented our approach in the tool AProVE. Our experiments show that the new contributions increase the power of termination analysis for JBC significantly.

1998 ACM Subject Classification D.1.5 - Object-oriented Programming, D.2.4 - Software/Program Verification, D.3.3 - Language Constructs and Features, F.3 - Logics and Meanings of Programs, F.4.2 - Grammars and Other Rewriting Systems, I.2.2 - Automatic Programming

Keywords and phrases termination, Java Bytecode, term rewriting, recursion

Digital Object Identifier 10.4230/LIPIcs.RTA.2011.155

Category Regular Research Paper

1 Introduction

While termination of TRSs and logic programs was studied for decades, recently there have also been many results on termination of *imperative programs* (e.g., [3, 6, 7, 8]). However, these methods do not re-use the many existing termination techniques for TRSs and declarative languages. Therefore, in [5, 15] we presented the first rewriting-based approach for proving termination of a real imperative object-oriented language, viz. Java Bytecode [14].

We only know of two other automated methods to analyze JBC termination, implemented in the tools COSTA [2] and Julia [16]. They transform JBC into a constraint logic program by abstracting objects of dynamic data types to integers denoting their path-length (e.g., list objects are abstracted to their length). While this fixed mapping from objects to integers leads to high efficiency, it also restricts the power of these methods.

In contrast, in [5, 15] we represent data objects not by integers, but by *terms* which express as much information as possible about the objects. For example, list objects are represented by terms of the form $\mathsf{List}(t_1, \mathsf{List}(t_2, \ldots \mathsf{List}(t_n, \mathsf{null}) \ldots))$. In this way, we benefit from the fact

* Supported by the DFG grant GI 274/5-3 and the G.I.F. grant 966-116.6.

22nd International Conference on Rewriting Techniques and Applications (RTA'11).
Editor: M. Schmidt-Schauß; pp. 155–170

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

that rewrite techniques can automatically generate well-founded orders comparing arbitrary forms of terms. Moreover, by using TRSs with built-in integers [9], our approach is not only powerful for algorithms on user-defined data structures, but also for algorithms on pre-defined data types like integers. To obtain TRSs that are suitable for termination analysis, our approach first transforms a JBC program into a *termination graph* which represents all possible runs of the program. These graphs handle all aspects of JBC that cannot easily be expressed in term rewriting (e.g., side effects, cyclic data objects, object-orientation, etc.). Afterwards, a TRS is generated from the termination graph. As proved in [5, 15], termination of this TRS implies termination of the original JBC program.

We implemented this approach in our tool AProVE [10] and in the *International Termination Competitions*,[1] AProVE achieved competitive results compared to Julia and COSTA.

However, a significant drawback was that (in contrast to techniques that abstract objects to integers [2, 8, 16]), our approach in [5, 15] could not deal with *recursion*. The problem is that for recursive methods, the size of the call stack usually depends on the input arguments. Hence, to represent all possible runs, this would lead to termination graphs with infinitely many states (since [5, 15] used no abstraction on call stacks). An abstraction of call stacks is non-trivial due to possible aliasing between references in different stack frames.

In the current paper, we solve these problems. Instead of directly generating a termination graph for the whole program as in [5, 15], in Sect. 2 we construct a separate termination graph for each method. These graphs can be combined afterwards. Similarly, one can also combine the TRSs resulting from these "method graphs" (Sect. 3). As demonstrated by our implementation in AProVE (Sect. 4), our new approach has two main advantages over [5, 15]:

(1) We can now analyze *recursive* methods, since our new approach can deal with call stacks that may grow unboundedly due to method calls.

(2) We obtain a *modular* approach, because one can re-use a method graph (and the rewrite rules generated from it) whenever the method is called. So in contrast to [5, 15], now we generate TRSs that are amenable to modular termination proofs.

See [4] for all proofs, and see [1] for experimental details and our previous papers [5, 15].

2 From Recursive JBC to Modular Termination Graphs

To analyze termination of a set of desired initial (concrete) program states, we represent this set by a suitable *abstract state* which is the initial node of the termination graph. Then this state is *evaluated symbolically*, which leads to its child nodes in the termination graph.

Our approach is restricted to verified[2] sequential JBC programs. To simplify the presentation in this paper, we exclude arrays, static class fields, interfaces, and exceptions. We also do not describe the annotations introduced in [5, 15] to handle complex sharing effects. With such annotations one can for example also model "unknown" objects with arbitrary sharing behavior as well as cyclic objects. Extending our approach to such constructs is easily possible and has been done for our implementation in the termination prover AProVE. However, currently our implementation has only minimal support for features like floating point arithmetic, strings, static initialization of classes, instances of `java.lang.Class`, reflection, etc.

Sect. 2.1 presents our notion of *states*. Sect. 2.2 introduces *termination graphs* for one method and Sect. 2.3 shows how to re-use these graphs for programs with many methods.

[1] See `http://www.termination-portal.org/wiki/Termination_Competition`.

[2] The bytecode verifier of the JVM [14] ensures certain properties of the code that are useful for our analysis, e.g., that there is no overflow or underflow of the operand stack.

2.1 States

```
final class List {
 List n;
 public void appE(int i) {
  if (n == null) {
   if (i <= 0) return;
   n = new List();
   i--;
  }
  n.appE(i);
 }}
```

```
00: aload_0       // load this to opstack
01: getfield n    // load this.n to opstack
04: ifnonnull 26  // jump to 26 if n is not null
07: iload_1       // load i to opstack
08: ifgt 12       // jump to 12 if i > 0
11: return        // return (without value)
12: aload_0       // load this to opstack
13: new List      // create new List object
16: dup           // duplicate top stack entry
17: invokespecial <init> // invoke constructor
20: putfield n    // write new List to field n
23: iinc 1, -1    // decrement i by 1
26: aload_0       // load this to opstack
27: getfield n    // load this.n to opstack
30: iload_1       // load i to opstack
31: invokevirtual appE   // recursive call
34: return        // return (without value)
```

Consider the recursive method `appE` (presented in both Java and JBC). We use a class `List` where the field `n` points to the next list element. For brevity, we omitted a field for the value of a list element. The method `appE` recursively traverses the list to its end, where it attaches `i` fresh elements (if `i > 0`).

Fig. 1 displays an abstract state of `appE`. A state consists of a sequence of *stack frames* and the *heap*, i.e., States = SFrames* ×Heap. The state in Fig. 1 has just a single stack frame "o_1, i_3 | `0` | `t`:o_1,`i`:i_3 | ε" which consists of four components. Its first component o_1, i_3 are the *input arguments*, i.e., those objects that are "visible" from outside the analyzed method. This component is new compared to [5, 15] and it is needed to denote later on which of these objects have been modified by side effects during the execution of the method. In our example, `appE` has two input arguments, viz. the implicit formal parameter `this` (whose value is o_1) and the formal parameter `i` with value i_3. In contrast to JBC, we also represent integers by references and adapt the semantics of all instructions to handle this correctly. So $o_1, i_3 \in$ Refs, where Refs is an infinite set of names for addresses on the heap.

o_1, i_3 | `0` | `t`:o_1,`i`:i_3 | ε
o_1:`List`(`n`=o_2) i_3:$\mathbb{Z}$
o_2:`List`(?)

Figure 1 State

The second component `0` of the stack frame is the *program position* (from ProgPos), i.e., the index of the next instruction. So `0` means that evaluation continues with `aload_0`.

The third component is the list of values of *local variables*, i.e., LocVar = Refs*. To ease readability, we do not only display the values, but also the variable names. For example, the name of the first local variable `this` is shortened to `t` and its value is o_1.

The fourth component is the *operand stack* to store temporary results, i.e., OpStack = Refs*. Here, ε is the empty stack and "o_8, o_1" denotes a stack with o_8 on top.

So the set of all *stack frames* is SFrames = InpArgs × ProgPos × LocVar × OpStack. As mentioned, the *call stack* of a state can consist of several stack frames. If a method calls another method, then a new frame is put on top of the call stack.

In addition to the call stack, a state contains information on the *heap*. The heap is a partial function mapping references to their value, i.e., Heap = Refs → Integers ∪ Instances ∪ Unknown ∪ {`null`}. We depict a heap by pairs of a reference and a value, separated by ":".

Integers are represented by intervals, i.e., Integers = $\{\{x \in \mathbb{Z} \mid a \leq x \leq b\} \mid a \in \mathbb{Z} \cup \{-\infty\}, b \in \mathbb{Z} \cup \{\infty\}, a \leq b\}$. We abbreviate $(-\infty, \infty)$ by $\mathbb{Z}$, $[1, \infty)$ by $[> 0]$, etc. So "$i_3 : \mathbb{Z}$" means that any integer can be at the address i_3. Since current TRS tools cannot handle 32-bit `int`-numbers, we treat all numeric types like `int` as the infinite set of all integers.

To represent Instances (i.e., objects) of some class, we store their type and the values of their fields, i.e., Instances = Classnames ×(FieldIDs → Refs). Classnames contains the names of all classes. FieldIDs is the set of all field names. To prevent ambiguities, in general the FieldIDs also include the respective class name. For all $(cl, f) \in$ Instances, the function f is defined for all fields of cl and of its superclasses. Thus, "o_1 : `List`(`n` = o_2)" means that at the address o_1, there is a `List` object whose field `n` has the value o_2.

UNKNOWN = CLASSNAMES ×{?} represents `null` and all tree-shaped objects for which we only have type information. In particular, UNKNOWN objects are acyclic and do not share parts of the heap with any objects at the other references in the state. For example, "o_2 : `List`(?)" means that o_2 is `null` or an instance of `List` (or a subtype of `List`).

Every *input argument* has a boolean flag, where *false* indicates that it may have been modified (as a side effect) by the current method. Moreover, we store which formal parameter of the method corresponds to this input argument. So in Fig. 1, the full input arguments are $(o_1, \text{LV}_{0,0}, \mathit{true})$ and $(i_3, \text{LV}_{0,1}, \mathit{true})$. Here, $\text{LV}_{i,j}$ is the *position* of the j-th local variable in the i-th stack frame. When the top stack frame (i.e., frame 0) is at program position `0` of a method, then its 0-th and 1-st local variables (at positions $\text{LV}_{0,0}$ and $\text{LV}_{0,1}$) correspond to the first and second formal parameter of the method. Formally, $\text{INPARGS} = 2^{\text{REFS} \times \text{SPOS} \times \mathbb{B}}$.

A *state position* $\pi \in \text{SPOS}(s)$ is a sequence starting with $\text{LV}_{i,j}$, $\text{OS}_{i,j}$ (for operand stack entries), or $\text{IN}_{i,\tau}$ (for input arguments (r, τ, b) in the i-th stack frame), followed by a sequence of FIELDIDS. This sequence indicates how to access a particular object.

▶ **Definition 2.1** (State Positions). Let $s = (\langle fr_0, \ldots, fr_n \rangle, h) \in \text{STATES}$ where $fr_i = (in_i, pp_i, lv_i, os_i)$. Then $\text{SPOS}(s)$ is the smallest set containing all the following sequences π:

- $\pi = \text{LV}_{i,j}$ where $0 \le i \le n$, $lv_i = \langle l_0, \ldots, l_m \rangle$, $0 \le j \le m$. Then $s|_\pi$ is l_j.
- $\pi = \text{OS}_{i,j}$ where $0 \le i \le n$, $os_i = \langle o_0, \ldots, o_k \rangle$, $0 \le j \le k$. Then $s|_\pi$ is o_j.
- $\pi = \text{IN}_{i,\tau}$ where $0 \le i \le n$ and $(r, \tau, b) \in in_i$. Then $s|_\pi$ is r.
- $\pi = \pi' v$ for some $v \in \text{FIELDIDS}$ and some $\pi' \in \text{SPOS}(s)$ where $h(s|_{\pi'}) = (cl, f) \in \text{INSTANCES}$ and where $f(v)$ is defined. Then $s|_\pi$ is $f(v)$.

The *references in the state* s are defined as $\mathit{Ref}(s) = \{s|_\pi \mid \pi \in \text{SPOS}(s)\}$.

So for the state s in Fig. 1, we have $s|_{\text{LV}_{0,0}} = s|_{\text{IN}_{0,\text{LV}_{0,0}}} = o_1$, $s|_{\text{LV}_{0,0}\,\texttt{n}} = s|_{\text{IN}_{0,\text{LV}_{0,0}}\,\texttt{n}} = o_2$, etc.

2.2 Termination Graphs for a Single Method

In Fig. 2, we construct the termination graph of `appE`. The state in Fig. 1 is its initial state A, i.e., we analyze termination of `appE` for acyclic lists of arbitrary length and any integer.

In A, `aload_0` loads the value of the 0-th local variable `this` on the operand stack. So A is connected by an *evaluation edge* to a state with program position `1` (omitted from Fig. 2 due to space reasons, i.e., dotted arrows abbreviate *several* steps). Then "`getfield n`" replaces o_1 on the operand stack by the value o_2 of its field `n`, resulting in state B. The value `List`(?) of o_2 does not provide enough information to evaluate `ifnonnull`. Thus, we perform an *instance refinement* [5, Def. 5] resulting in C and D, i.e., a case analysis whether o_2's value is `null`. *Refinement edges* are denoted by dashed lines. In C, we assume that o_2's value is not `null`. Thus, we replace o_2 by a fresh[3] reference o_4, which points to `List`(`n` = o_5). Hence, we can now evaluate `ifnonnull` and jump to instruction `26` in state M.

In D, we assume that o_2's value is `null`, i.e., "o_1 : `List`(`n` = o_2)" and "o_2 : `null`". To ease the presentation, in such states we simply replace all occurrences of o_2 with `null`. After evaluating the instruction "`ifnonnull 26`", in the next state (which we omitted from Fig. 2 for space reasons), the instruction "`iload_1`" loads the value of `i` on the operand stack. This results in state E. Now again we do not have enough information to evaluate `ifgt`. Thus, we perform an *integer refinement* [5, Def. 1], leading to states F (if `i <= 0`) and H.

[3] We rename references that are refined to ease the formal definition of the refinements, cf. [5].

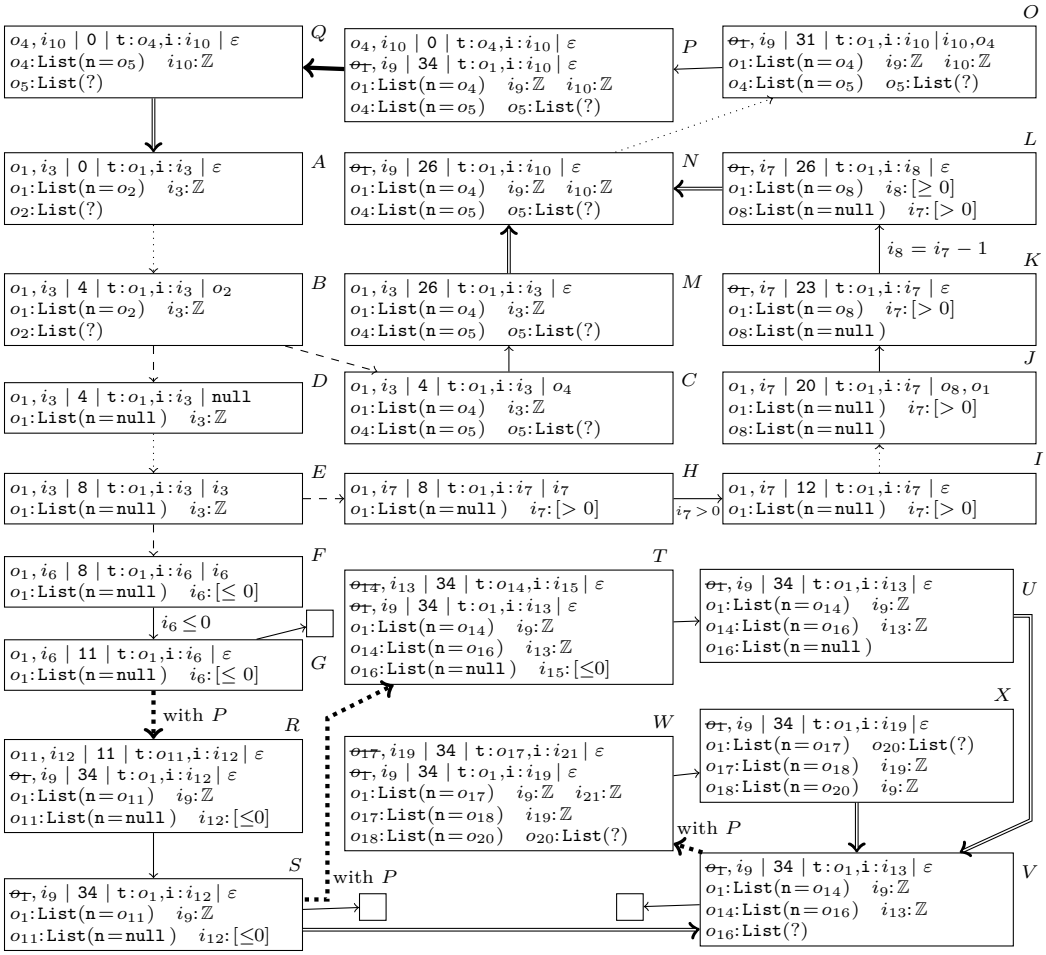


Figure 2 Termination Graph of `appE`

In F, we evaluate `ifgt`, leading to G. We label the edge from F to G with the condition $i_6 \leq 0$ of this case. This label will be used when generating a TRS from the termination graph. States like G that have only a single stack frame which is at a `return` position are called *return states*. Thus, we reach a *program end*, denoted by $\square$. From H, we jump to instruction `12` in I and label the edge with $i_7 > 0$. In I, o_1 is pushed on the operand stack. Afterwards, we create another list element o_8, where we skipped the constructor call in Fig. 2. In K, o_8 has been written to the field `n` of o_1. This is a *side effect* on an object that is visible from outside the method (since o_1 is an input argument). Hence, in K we set the boolean flag for o_1 to *false* (depicted by crossing out the input argument o_1).

In L, the value of the 1-st local variable `i` is decremented by 1. In contrast to JBC, we represent primitive data types by references. Hence, we introduce a fresh reference i_8, pointing to the adapted value. Since i_7's value did not change, i_7 is not crossed out.

State L is similar to the state M we obtained from the other branch of our first refinement. To simplify the graph, we create a *generalized* state N, which represents a superset of all concrete states represented by L or M. N is almost like M (up to renaming of references) and only differs in the information about input arguments, which is taken from L. We draw *instance edges* (double arrows) from L and M to N and only consider N in the remainder.

In O, we have loaded `this.n` and `i` on the operand stack and invoke `appE` on these values.

So in P, a second stack frame is pushed on top of the previous one. States like P that contain at least two frames where the top frame is at the start of a method are *call states*.

We now introduce a new approach to represent call stacks of arbitrary size by *splitting up* call stacks. Otherwise, for recursive methods the call stack could grow unboundedly and we would obtain an infinite termination graph. So P has a *call edge* (thick arrow) to Q which only contains P's top stack frame. Since Q is identical to A (modulo renaming), we do not have to analyze `appE` again, but simply draw an instance edge from Q to A.

Up to now A only represented concrete states where `appE` was called "directly". However, now A can also be reached from a "method call" in P. Hence, now A and the other abstract states s of `appE`'s termination graph also represent states where `appE` was called "recursively", i.e., where below the stack frames of s, one has the stack frames of P (only P's top frame is replaced by the frames of s).[4] For each *return state* we now consider two cases: Either there are no further frames below the top frame (then one reaches a leaf of the termination graph) or else, there are further frames below the top (which result from the method call in P). Hence, for every return state like G, we now create an additional successor state R (the *context concretization of G with P*), connected by a *context concretization edge* (a thick dotted arrow). R has the same stack frame as G (up to renaming), but below we add the call stack of P (without P's top frame that corresponded to the method call).

In R, `appE`'s recursive call has just reached the `return` statement at index `11`. Here, we identified o_1 and i_6 from state G with o_4 and i_{10} from P and renamed them to o_{11} and i_{12}. We now consider which information we have about R's heap. According to state G, the input arguments of `appE`'s recursive call were not modified during the execution of this recursive call. Thus, for the input arguments o_{11} and i_{12} in R, we can use *both* the information on o_1 and i_6 in G and on o_4 and i_{10} in P. According to G, o_1 is a list of length 1 and $i_6 \leq 0$. According to P, o_4 has at least length 1 and i_{10} is arbitrary. Hence, in R we can take the *intersection* of this information and deduce that o_{11} has length 1 and $i_{12} \leq 0$. (So in this example, the intersection of G's and P's information coincides with the information in G.)

When constructing termination graphs, context concretization is only needed for return states. But to formulate Thm. 2.3 on the soundness of termination graphs later on, in Def. 2.2 we introduce context concretization for arbitrary states $s = (\langle fr_0, \ldots, fr_n \rangle, h)$. So s results from evaluating the method in the bottom frame fr_n (i.e., fr_{n-1} was created by a call in fr_n, fr_{n-2} was created by a call in fr_{n-1}, etc.). Context concretization of s with a call state $\overline{s} = (\langle \overline{fr}_0, \ldots, \overline{fr}_m \rangle, \overline{h})$ means that we consider the case where fr_n results from a call in $\overline{fr}_1$. Thus, the top frame $\overline{fr}_0$ of $\overline{s}$ is at the start of some method and the bottom frame fr_n of s must be at an instruction of the *same* method. Moreover, for all input arguments $(\overline{r}, \tau, \overline{b})$ in $\overline{fr}_0$ there must be a *corresponding* input argument (r, τ, b) in fr_n.[5] To ease the formalization, let $Ref(s)$ and $Ref(\overline{s})$ be disjoint. For instance, if s is G and $\overline{s}$ is P, we can mark the references by ^{G} and ^{P} to achieve disjointness (e.g., $o_1^G \in Ref(G)$ and $o_1^P \in Ref(P)$).

Then we add the frames $\overline{fr}_1, \ldots, \overline{fr}_m$ of the call state $\overline{s}$ below the call stack of s to obtain a new state $\tilde{s}$ with the call stack $\langle fr_0\sigma, \ldots, fr_n\sigma, \overline{fr}_1\sigma, \ldots, \overline{fr}_m\sigma \rangle$. The *identification substitution* σ identifies every input argument $\overline{r}$ of $\overline{fr}_0$ with the corresponding input argument r of fr_n. If the boolean flag for the input argument r in s is *false*, then this object may have changed during the evaluation of the method and in $\tilde{s}$, we should only use the information

[4] For example, A now represents all states with call stacks $\langle fr^A, fr_1^P, fr_1^P, \ldots, fr_1^P \rangle$ where fr^A is A's stack frame and $fr_1^P, fr_1^P, \ldots, fr_1^P$ are copies of P's bottom frame (in which references may have been renamed). So A represents states where `appE` was called within an arbitrary high context of recursive calls.

[5] This obviously holds for all input arguments corresponding to formal parameters of the method, but Sect. 2.3 will illustrate that sometimes $\overline{fr}_0$ may have additional input arguments.

from s. But if the flag is *true*, then the object did not change. Then, both the information in s and in $\overline{s}$ about this object is correct and for $\tilde{s}$, we take the intersection of this information. In our example, $\sigma(o_1^G) = \sigma(o_4^P) = o_{11}^R$ and $\sigma(i_6^G) = \sigma(i_{10}^P) = i_{12}^R$. Since the flags of the input arguments o_1^G and i_6^G are *true*, for o_{11}^R and i_{12}^R, we intersect the information from G and P.

If we identify r and $\overline{r}$, and both point to INSTANCES, then we may also have to identify the references in their fields. To this end, we define an equivalence relation $\equiv \subseteq \text{REFS} \times \text{REFS}$ where "$r \equiv \overline{r}$" means that r and $\overline{r}$ are identified. Let $r \equiv \overline{r}$ and let r be no input argument in s with the flag *false*. If r points to (cl, f) in s and $\overline{r}$ points to $(cl, \overline{f})$ in $\overline{s}$, then all references in the fields v of cl and its superclasses also have to be identified, i.e., $f(v) \equiv \overline{f}(v)$.

To illustrate this in our example, note that we abbreviated the information on G's heap in Fig. 2. In reality we have "$o_1^G : \texttt{List}(\texttt{n} = o_2^G)$", "$o_2^G : \texttt{null}$", and "$i_6^G : [\leq 0]$". Hence, we do not only obtain $i_6^G \equiv i_{10}^P$ and $o_1^G \equiv o_4^P$, but since o_1^G's boolean flag is not *false*, we also have to identify the references at the field `n` of the object, i.e., $o_2^G \equiv o_5^P$.

Let ρ be an injective function that maps each $\equiv$-equivalence class to a fresh reference. We define the *identification substitution* σ as $\sigma(r) = \rho([r]_\equiv)$ for all $r \in \mathit{Ref}(s) \cup \mathit{Ref}(\overline{s})$. So we map equivalent references to the same new reference and we map non-equivalent references to different references. To construct $\tilde{s}$, if $r \in \mathit{Ref}(s)$ points to an object which was not modified by side effects during the execution of the called method (i.e., where the flag is not *false*), we intersect all information in s and $\overline{s}$ on the references in $[r]_\equiv$. For all other references in $\mathit{Ref}(s)$ resp. $\mathit{Ref}(\overline{s})$, we only take the information from s resp. $\overline{s}$ and apply σ.

In our example, we have the equivalence classes $\{o_1^G, o_4^P\}$, $\{o_2^G, o_5^P\}$, $\{i_6^G, i_{10}^P\}$, $\{o_1^P\}$, and $\{i_9^P\}$. For these classes we choose the new references o_{11}^R, o_2^R, i_{12}^R, o_1^R, i_9^R, and obtain $\sigma = \{o_1^G/o_{11}^R, o_4^P/o_{11}^R, o_2^G/o_2^R, o_5^P/o_2^R, i_6^G/i_{12}^R, i_{10}^P/i_{12}^R, o_1^P/o_1^R, i_9^P/i_9^R\}$. The information for o_{11}^R, o_2^R, and i_{12}^R is obtained by intersecting the respective information from G and P. The information for o_1^R and i_9^R is taken over from P (by applying σ).

Def. 2.2 also introduces the concept of *intersection* formally. If $r \in \mathit{Refs}(s)$, $\overline{r} \in \mathit{Refs}(\overline{s})$, and h resp. $\overline{h}$ are the heaps of s resp. $\overline{s}$, then intuitively, $h(r) \cap \overline{h}(\overline{r})$ consists of those values that are represented by both $h(r)$ and $\overline{h}(\overline{r})$. For example, if $h(r) = [\geq 0] = (-1, \infty)$ and $\overline{h}(\overline{r}) = [\leq 0] = (-\infty, 1)$, then the intersection is $(-1, 1) = [0, 0]$. Similarly, if $h(r)$ or $\overline{h}(\overline{r})$ is `null`, then their intersection is again `null`. If $h(r), \overline{h}(\overline{r})$ are UNKNOWN instances of classes cl_1, cl_2, then their intersection is an UNKNOWN instance of the more special class $\min(cl_1, cl_2)$. Here, $\min(cl_1, cl_2) = cl_1$ if cl_1 is a (not necessarily proper) subtype of cl_2 and $\min(cl_1, cl_2) = cl_2$ if cl_2 is a subtype of cl_1. Otherwise, cl_1 and cl_2 are called *orthogonal*. If $h(r) \in$ UNKNOWN and $\overline{h}(\overline{r}) \in$ INSTANCES, then their intersection is from INSTANCES using the more special type. Finally, if both $h(r), \overline{h}(\overline{r}) \in$ INSTANCES with the same type, then their intersection is again from INSTANCES. For the references in its fields, we use the identification substitution σ that renames equivalent references to the same new reference.

Note that one may also have to identify different references in the *same* state. For example, $\overline{s}$ could have the input arguments $(\overline{r}, \tau_1, \overline{b})$ and $(\overline{r}, \tau_2, \overline{b})$ with the corresponding input arguments (r_1, τ_1, b_1) and (r_2, τ_2, b_2) in s. Then $\overline{r} \equiv r_1 \equiv r_2$. Note that if $r_1 \neq r_2$ are references from the *same* state where $h(r_1) \in$ INSTANCES, then they point to different objects (i.e., then $h(r_1) \cap h(r_2)$ is empty). Similarly, if $h(r_1), h(r_2) \in$ UNKNOWN, then they also point to different objects or to `null` (i.e., then $h(r_1) \cap h(r_2)$ is `null`).

▶ **Definition 2.2** (Context Concretization). Let $s = (\langle fr_0, \ldots, fr_n \rangle, h)$ and let $\overline{s} = (\langle \overline{fr}_0, \ldots, \overline{fr}_m \rangle, \overline{h})$ be a call state where fr_n and $\overline{fr}_0$ correspond to the same method. (So $\overline{fr}_0$ is at the start of the method and fr_n can be at any position of the method.) Let in_n resp. $\overline{in}_0$ be the input arguments of fr_n resp. $\overline{fr}_0$, and let $\mathit{Ref}(s) \cap \mathit{Ref}(\overline{s}) = \varnothing$. For every input argument $(\overline{r}, \tau, \overline{b}) \in \overline{in}_0$ there must be a *corresponding* input argument $(r, \tau, b) \in in_n$ (i.e., with the same

position τ), otherwise there is no context concretization of s with $\overline{s}$. Let $\equiv \subseteq \text{REFS} \times \text{REFS}$ be the smallest equivalence relation which satisfies the following two conditions:

- if $(\overline{r}, \tau, \overline{b}) \in \overline{in}_0$ and $(r, \tau, b) \in in_n$, then $r \equiv \overline{r}$.
- if $r \in \mathit{Ref}(s)$, $\overline{r} \in \mathit{Ref}(\overline{s})$, $r \equiv \overline{r}$, and there is no $(r, \tau, \mathit{false}) \in in_n$, then $h(r) = (cl, f)$ and $\overline{h}(\overline{r}) = (cl, \overline{f})$ implies that $f(v) \equiv \overline{f}(v)$ holds for all fields v of cl and its superclasses.

Let $\rho : \text{REFS}/_{\equiv} \to \text{REFS}$ be an injective mapping to fresh references $\notin \mathit{Ref}(s) \cup \mathit{Ref}(\overline{s})$ and let $\sigma(r) = \rho([r]_{\equiv})$ for all $r \in \mathit{Ref}(s) \cup \mathit{Ref}(\overline{s})$. Then the *context concretization of s with $\overline{s}$* is the state $\tilde{s} = (\langle \mathit{fr}_0\sigma, \ldots, \mathit{fr}_n\sigma, \overline{\mathit{fr}}_1\sigma, \ldots, \overline{\mathit{fr}}_m\sigma \rangle, \tilde{h})$. Here, we define $\tilde{h}(\sigma(r))$ to be

- $h(r_1) \cap \ldots \cap h(r_k) \cap \overline{h}(\overline{r}_1) \cap \ldots \cap \overline{h}(\overline{r}_d)$, if $[r]_{\equiv} \cap \mathit{Ref}(s) = \{r_1, \ldots, r_k\}$, $[r]_{\equiv} \cap \mathit{Ref}(\overline{s}) = \{\overline{r}_1, \ldots, \overline{r}_d\}$, and there is no input argument $(r_i, \tau, \mathit{false}) \in in_n$
- $h(r_1) \cap \ldots \cap h(r_k)$, if $[r]_{\equiv} \cap \mathit{Ref}(s) = \{r_1, \ldots, r_k\}$, and there is an $(r_i, \tau, \mathit{false}) \in in_n$

If the intersection is empty, then there is no concretization of s with $\overline{s}$. Moreover, whenever there is an input argument $(\overline{r}, \tau, \overline{b}) \in \overline{in}_0$ with corresponding input argument $(r, \tau, \mathit{false}) \in in_n$, then for all input arguments $(\overline{r}', \tau', \overline{b}')$ in lower stack frames of $\overline{s}$ where $\overline{r}'$ reaches[6] $\overline{r}$ in $\overline{h}$, the flag $\overline{b}'$ must be replaced by *false* when creating the context concretization $\tilde{s}$. In other words, in the lower stack frame of $\tilde{s}$, we then have the input argument $(\overline{r}'\sigma,\ \tau',\ \mathit{false})$.

Finally, for all $s_1, \ldots, s_k \in \{s, \overline{s}\}$ where h_i is the heap of s_i, and for all pairwise different references $r_1, \ldots, r_k$ with $r_i \in \mathit{Ref}(s_i)$ where $r_1 \equiv \ldots \equiv r_k$, we define $h_1(r_1) \cap \ldots \cap h_k(r_k)$ to be $h_1(r_1)\sigma$ if $k = 1$. Otherwise, $h_1(r_1) \cap \ldots \cap h_k(r_k)$ is

- $(\max(a_1, \ldots, a_k), \min(b_1, \ldots, b_k))$, if all $h_i(r_i) = (a_i, b_i) \in \text{INTEGERS}$ and $\max(a_1, \ldots, a_k) + 1 < \min(b_1, \ldots, b_k)$
- `null`, if all $h_i(r_i) \in \text{UNKNOWN} \cup \{\texttt{null}\}$ and at least one of them is `null`
- `null`, if all $h_i(r_i) \in \text{UNKNOWN}$ and there are $j \neq j'$ with $s_j = s_{j'}$
- `null`, if $k = 2$, $h_1(r_1) = (cl_1, ?)$, $h_2(r_2) = (cl_2, ?)$ and cl_1, cl_2 are orthogonal
- $(\min(cl_1, cl_2), ?)$, if $k = 2$, $s_1 \neq s_2$, $h_1(r_1) = (cl_1, ?)$, $h_2(r_2) = (cl_2, ?)$, and cl_1, cl_2 are not orthogonal
- (cl, f), if $k = 2$, $s_1 \neq s_2$, $h_1(r_1) = (cl, f_1), h_2(r_2) = (cl, f_2) \in \text{INSTANCES}$. Here, $f(v) = \sigma(f_1(v)) = \sigma(f_2(v))$ for all fields v of cl and its superclasses.
- $(\min(cl_1, cl_2), f)$, if $k = 2$, $s_1 \neq s_2$, $h_1(r_1) = (cl_1, ?)$, $h_2(r_2) = (cl_2, f_2)$, and cl_1, cl_2 are not orthogonal. Here, $f(v) = \sigma(f_2(v))$ for all fields v of cl_2 and its superclasses. If cl_1 is a subtype of cl_2, then for those fields v of cl_1 and its superclasses where f_2 is not defined, $f(v)$ returns a fresh reference r_v where $\tilde{h}(r_v) = (-\infty, \infty)$ if the field v has an integer type and $\tilde{h}(r_v) = (cl_v, ?)$ if the type of the field v is some class cl_v. The case where $h_1(r_1) \in \text{INSTANCES}$ and $h_2(r_2) \in \text{UNKNOWN}$ is analogous.

In all other cases, $h_1(r_1) \cap \ldots \cap h_k(r_k)$ is empty.

We continue with constructing `appE`'s termination graph. When evaluating R, the top frame is removed from the call stack and due to the lower stack frame, we now reach a new return state S. As above, for every return state, we have to create a new context concretization T which is like the call state P, but where P's top stack frame is replaced by the stack frame of the return state S. We use an identification substitution σ which maps o_1^S and o_4^P to o_{14}^T, i_9^S and i_{10}^P to i_{13}^T, i_{12}^S to i_{15}^T, o_{11}^S to o_{16}^T, o_1^P to o_1^T, and i_9^P to i_9^T. The value of o_{14}^T (i.e., o_1^S and o_4^P) may have changed during the execution of the top frame (as o_1^S is

[6] We say that $\overline{r}'$ *reaches* $\overline{r}$ in $\overline{h}$ iff there is a position $\pi_1\,\pi_2 \in \text{SPOS}(\overline{s})$ such that $\overline{s}|_{\pi_1} = \overline{r}'$ and $\overline{s}|_{\pi_1\,\pi_2} = \overline{r}$.

crossed out). Hence, we only take the value from S, i.e., o_{14}^T is a list of length 2. For i_{13}^T, we intersect the information on i_9^S and on i_{10}^P. The information on i_{15}^T is taken from i_{12}^S and the information on o_1^T resp. i_9^T is taken from o_1^P resp. i_9^P (where σ is applied).

When evaluating T, the top frame is removed and we reach a new return state U. If we continued in this way, we would perform context concretization on U again, etc. Then the construction would not finish and we would get an infinite termination graph.

To obtain finite graphs, we use the heuristic to generalize all return states with the same program position to one common state, i.e., only one of them may have no outgoing instance edge. Then this generalized state can be used instead of the original ones. In S, `this` is a list of length 2, whereas in U, `this` has length 3. Moreover, `i` ≤ 0 in S, whereas `i` is arbitrary in U. Therefore, we generalize S and U to a new state V where `this` has length ≥ 2 and `i` is arbitrary. Now T and U are not needed anymore and could be removed.

As V is a return state, we have to create a new successor W by context concretization, which is like the call state P, but where P's top frame is replaced by V's frame (analogous to the construction of T). Evaluating W leads to X, which is an instance of V. Thus, we draw an instance edge from X to V and the termination graph construction is finished.

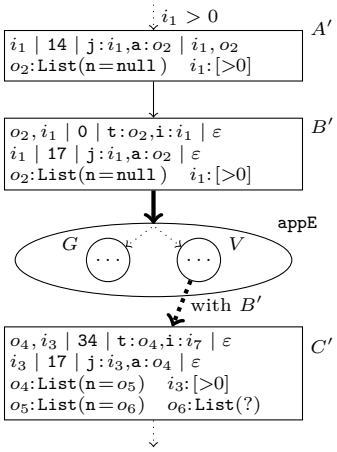
In general, a state s' is an *instance* of a state s (denoted $s' \sqsubseteq s$) if all concrete states represented by s' are also represented by s. For a formal definition of "$\sqsubseteq$", we refer to [5, Def. 3] and [15, Def. 2.3]. The only condition that has to be added to this definition is that for every input argument (r', τ, b') in the i-th frame of s', there must also be a corresponding input argument (r, τ, b) in the i-th frame of s, where $b' = \mathit{false}$ implies $b = \mathit{false}$.

However in [5, 15], $s' \sqsubseteq s$ only holds if s' and s have the same call stack size. In contrast, we now also allow larger call stacks in s' and define $s' \sqsubseteq s$ iff a state $\tilde{s}$ can be obtained by repeated context concretization from s, where s' and $\tilde{s}$ have the same call stack size and $s' \sqsubseteq \tilde{s}$. For example, $P \sqsubseteq A$, although P has two and A only has one stack frame, since context concretization of A (with P) yields a state $\tilde{A}$ which is a renaming of P (thus, $P \sqsubseteq \tilde{A}$).

2.3 Termination Graphs for Several Methods

```
static void cappE(int j) {
 List a = new List();
 if (j > 0) {
  a.appE(j);
  while (a.n == null) {}
 }}
```

Termination graphs for a method can be re-used whenever the method is called. To illustrate this, consider a method `cappE` which calls `appE`. It constructs a new `List a`, checks if the formal parameter `j` is `> 0`, and calls `a.appE(j)` to append `j` elements to `a`. Then, if `a.n` is `null`, one enters a non-terminating loop. But as `j > 0`, our analysis can detect that after the call `a.appE(j)`, the list `a.n` is not `null`. Hence, the loop is never executed and `cappE` is terminating.



In `cappE`'s termination graph, after constructing the new `List` and checking `j > 0`, one reaches A'. The call of `appE` leads to the call state B', whose top frame is at position `0` of `appE`. As in the step from P to Q in Fig. 2, we now split the call stack. The resulting state (with only B''s top frame) is connected by an instance edge to the initial state A of `appE`'s termination graph, i.e., we re-use the graph of Fig. 2. Recall that for every call state $\overline{s}$ that calls `appE` and each return state s in `appE`'s termination graph, we perform context concretization of s with $\overline{s}$. In fact, one can restrict this to return states $\overline{s}$ without outgoing instance edges (i.e., to G and V).

Now we have another call state B' which calls `appE`. G has no context concretization with B', as the second input

argument is ≤ 0 in G and > 0 in B' (i.e., the intersection is empty). Context concretization of V with B' yields state C'. Here, $i_3^{C'}$ results from intersecting i_9^V and $i_1^{B'}$, whereas $o_4^{C'}$ is taken over from o_1^V (thus in C', `a.n` is not `null` and hence, the `while` loop is not executed).[7]

To define termination graphs formally, in [5, Def. 6] we extended JBC-evaluation to abstract states, i.e., "$s \overset{SyEv}{\longrightarrow} s'$" means that s *symbolically evaluates* to s'. We now extend [5, Def. 6] to handle *input arguments*. Input arguments remain unchanged by symbolic evaluation, except when evaluating `putfield` or invoking a method. If evaluation of a `putfield` instruction changes an object at a position $\text{IN}_{i,\tau}\ \pi$, then we set the boolean flag b of the input argument (r, τ, b) in the i-th stack frame to *false* (cf. $J \overset{SyEv}{\longrightarrow} K$ in Fig. 2).

Now we explain how to create the input arguments for new stack frames which are generated when invoking a method. In general, one may need more input arguments than the method's formal parameters. To see this, consider a variant of `cappE`, where before the call of `appE`, we add the instruction "`List b = a.n = new List();`". Thus, now `a` is a list of length 2 and `b` also points to `a`'s second element. Hence, in state A' we now have the local variables "$\texttt{j}{:}i_1$, $\texttt{a}{:}o_2$, $\texttt{b}{:}o_3$" where "$o_2 : \texttt{List}(\texttt{n} = o_3)$" and "$o_3 : \texttt{List}(\texttt{n} = \texttt{null})$". As before, `appE` is called with the arguments o_2 and i_1 and its execution modifies the object at o_2 as a side effect. However, due to this, the object at o_3 is modified *as well*. We have to take this into account, because after the execution of `appE`, the object at o_3 is still accessible via the local variable `b`. So here the execution of a called method has a side effect on objects that are visible from lower frames of the call stack.

Recall that the purpose of the *input arguments* is to describe which objects may have changed (as a side effect) during the execution of the method. Therefore in B', we now have to add o_3 as an additional input argument when calling `appE`. More precisely, the three input arguments of B' would be $(o_2, \text{LV}_{0,0}, \mathit{true})$, $(i_1, \text{LV}_{0,1}, \mathit{true})$, and $(o_3, \text{LV}_{0,0}\ \texttt{n}, \mathit{true})$ (corresponding to the field `n` of `appE`'s first formal parameter).

Consequently, we now have to re-process the termination graph of `appE` to obtain a variant where the states have three input arguments. The stack frame of V would then be "$\sout{o_1}, i_9, \sout{o_{14}}, \mid 34 \mid \texttt{t}{:}o_1, \texttt{i}{:}i_{13} \mid \varepsilon$". Hence, in the context concretization of V with B' (where o_{14}^V is identified with $o_3^{B'}$), the information on $o_3^{B'}$ is longer valid, but instead one has to use o_{14}^V. Thus in C', the value of `b` is no longer "$o_3 : \texttt{List}(\texttt{n} = \texttt{null})$", but "$o_5 : \texttt{List}(\texttt{n} = o_6^{C'})$", where $o_6^{C'}$'s value is a copy of V's value for o_{16}^V, i.e., `List`(?).

So for any call state[8] $\overline{s}$, if there is a number i and a $\tau \in \text{FieldIDs}^*$ such that $\overline{s}|_{\text{LV}_{0,i}\,\tau} = r$, then $(r, \text{LV}_{0,i}\,\tau, \mathit{true})$ should be included in the input arguments of the top stack frame. The only exception are references r that are no *top references* and where all *predecessors* of r can also be reached from some formal parameter $\overline{s}|_{\text{LV}_{0,j}}$ of the called method. The reason is that then r is only reachable from other input arguments of $\overline{s}$ and hence, their flags suffice to indicate whether the object at r has changed. Here, r is a *top reference* iff $\overline{s}|_\pi = r$ holds for some position π with $|\pi| = 1$ (i.e., π has the form $\text{LV}_{i,j}$, $\text{OS}_{i,j}$, or $\text{IN}_{i,\tau}$). A reference r' is a *predecessor* of r iff $\overline{s}|_\pi = r'$ and $\overline{s}|_{\pi\,v} = r$ for some $\pi \in \text{SPos}(\overline{s})$ and some $v \in \text{FieldIDs}$.

For P in Fig. 2, o_4, i_{10}, and o_5 are at positions of the form $\text{LV}_{0,i}\,\tau$. However, only o_4 and

[7] When methods modify objects as a side effect, the exact result of this modification is often not expressible if objects are abstracted to integers. Therefore tools like Julia and COSTA often do not try to express such modifications and fail if this would have been crucial for the termination proof. Indeed, for `cappE`'s termination, one needs information about the object `a` *after* it was modified by `a.appE(j)`. Therefore, while Julia and COSTA can prove termination of `appE`, they fail on `cappE` (although in this example, the effect of the modification would even be expressible when using the path-length abstraction to integers).

[8] In fact, this requirement also has to be imposed for initial states of method graphs, i.e., states with just one stack frame and program position `0` (i.e., at the start of a method).

i_{10} must be input arguments (o_5 is not at a top position and its only predecessor is o_4).

Finally, we can explain how to construct termination graphs in general:

- Each call state $(\langle \overline{fr}_0, \ldots, \overline{fr}_m \rangle, \overline{h})$ is connected to $(\langle \overline{fr}_0 \rangle, \overline{h})$ by a *call edge*.
- Each return state $s = (\langle fr \rangle, h)$ has an edge to the *program end* (ε, h) and *context concretization edges* to all context concretizations of s with call states of the termination graph.
- For all other states s, if $s \xrightarrow{SyEv} s'$, then we connect s to s' by an *evaluation edge*.
- If evaluation is impossible, we use integer or instance refinement (using *refinement edges*).
- To get finite graphs,[9] we use a heuristic which sometimes introduces more general states (e.g., when a program position is visited twice). If $s' \sqsubseteq s$, then s' can be connected to s by an *instance edge*. However, all cycles of the graph must contain an evaluation edge.
- In a termination graph, all nodes except *program ends* must have outgoing edges.

In [5, Thm. 10] we proved that on *concrete* states, our notion of symbolic evaluation $\xrightarrow{SyEv}$ is equivalent to evaluation in JBC. Thm. 2.3 shows that symbolic evaluation of *abstract* states correctly simulates the evaluation of concrete states (and hence, of JBC).

▶ **Theorem 2.3** (Soundness of Termination Graphs). *Let c, c' be concrete states where c can be evaluated to c' (i.e., $c \xrightarrow{SyEv} c'$). If a termination graph contains an abstract state s which represents c (i.e., $c \sqsubseteq s$), then the graph has a path from s to a state s' with $c' \sqsubseteq s'$.*

Paths in the termination graph that correspond to repeated evaluations of concrete states are called *computation paths*. Note that Thm. 2.3 can be used to prove the soundness of our approach: Suppose there is an infinite JBC-computation, i.e., an infinite evaluation of concrete states $c_1 \xrightarrow{SyEv} c_2 \xrightarrow{SyEv} \ldots$ If c_1 is represented in the termination graph, then by Thm. 2.3 there is an infinite computation path in the termination graph. In Thm. 3.3, we will show that then the TRS resulting from the termination graph is not terminating.

3 From Modular Termination Graphs to Term Rewriting

We now transform termination graphs into *integer term rewrite system* (*ITRSs*) [9]. These are conditional TRSs where the booleans, integers, standard arithmetic operations $\mathcal{A}rith\mathcal{O}p$ like $+, -, *, /, \ldots$, and standard relations $\mathcal{R}el\mathcal{O}p$ like $>, <, \ldots$ are pre-defined by an infinite set of rules $\mathcal{PD}$. For example, $\mathcal{PD}$ contains $4 + 2 \to 6$ and $2 < 3 \to \mathsf{true}$. The *rewrite relation* $\hookrightarrow_{\mathcal{R}}$ of an ITRS $\mathcal{R}$ is defined as the *innermost* rewrite relation of $\mathcal{R} \cup \mathcal{PD}$, where all variables (including extra variables in conditions or right-hand sides of rules) may only be instantiated by normal forms. So if $\mathcal{R}$ contains "$\mathsf{f}(x) \to \mathsf{g}(x, y) \mid x > 2$", then $\mathsf{f}(4 + 2) \hookrightarrow_{\mathcal{R}} \mathsf{f}(6) \hookrightarrow_{\mathcal{R}} \mathsf{g}(6, 23)$. TRS termination techniques can easily be adapted to ITRSs as well [9].

As in [15, Def. 3.2], a reference r in a state s with heap h is transformed into a term by the function $\mathrm{tr}(s, r)$. If $h(r) \in$ Unknown or $h(r)$ is an integer interval of several numbers, then $\mathrm{tr}(s, r)$ is a variable with the name r. If $h(r)$ is a concrete integer like $[5, 5]$, then $\mathrm{tr}(s, r)$ is the corresponding constant 5. If $h(r) = \mathtt{null}$, then $\mathrm{tr}(s, r)$ is the constant null.

The main advantage of our rewrite-based approach becomes obvious when transforming data objects into terms (i.e., when $h(r) \in$ Instances). The reason is that such data objects essentially *are* terms and hence, our transformation can keep their structure. We use the class names as function symbols, and the arguments of these symbols represent the values of

[9] Indeed, our implementation uses heuristics which guarantee that we automatically generate a finite termination graph for any JBC program.

fields. So to represent objects of the class `List`, we use a unary function symbol List whose argument corresponds to the value of the field `n`. Thus, o_1 in P from Fig. 2 is transformed into the term $\mathrm{tr}(P, o_1) = \mathsf{List}(\mathsf{List}(o_5))$.[10] However, references r pointing to *cyclic* objects are transformed to a variable r in order to represent an "arbitrary unknown" object.

Now we show how to transform states into terms. In [5, 15], for each state s we used a function symbol f_s which had one argument for each top position in the state. In contrast, to model the call and return of methods, we now encode each stack frame on its own. Then a state is represented by nesting the terms for its stack frames.

To encode a stack frame (in, pp, lv, os) of s to a term, we use a function symbol $f_{s,pp}$ whose arguments correspond to the top positions in this frame. To represent the call stack, $f_{s,pp}$ gets an additional first argument, which contains the encoding of the frame *above* the current one, or eos (for "end of stack") if there is no such frame. So the top stack frame is always at an innermost position of the form $1\,1\ldots1$. Thus, state P is encoded as the term

$$\mathrm{ts}(P) \;=\; \mathsf{f}_{P,34}(\,\mathsf{f}_{P,0}(\mathsf{eos},\, \underbrace{\mathsf{List}(o_5)}_{o_4},\, i_{10},\, \underbrace{\mathsf{List}(o_5)}_{o_4},\, i_{10}),\quad \underbrace{\mathsf{List}(\mathsf{List}(o_5))}_{o_1},\, i_9,\, \underbrace{\mathsf{List}(\mathsf{List}(o_5))}_{o_1},\, i_{10})$$

In Def. 3.1, for any sequence $\langle r_1, ..., r_k\rangle$, "$\mathrm{tr}(s, \langle r_1, ..., r_k\rangle)$" stands for "$\mathrm{tr}(s, r_1), \ldots, \mathrm{tr}(s, r_k)$".

▶ **Definition 3.1** (Transforming States)**.** Let $s = (\langle fr_0, \ldots, fr_n\rangle, h)$ with $fr_i = (in_i, pp_i, lv_i, os_i)$ and $in_i = \{(r_{i,0}, \tau_{i,0}, b_{i,0}), \ldots, (r_{i,k_i}, \tau_{i,k_i}, b_{i,k_i})\}$, for all i. We define $\mathrm{ts}(s) = \overline{\mathrm{ts}}(s, n)$, where

$$\overline{\mathrm{ts}}(s,i) = \begin{cases} \mathsf{f}_{s,pp_i}\left(\,\overline{\mathrm{ts}}(s, i-1),\ \mathrm{tr}(s, \langle r_{i,0}\ldots r_{i,k_i}\rangle),\ \mathrm{tr}(s, lv_i),\ \mathrm{tr}(s, os_i)\,\right), & \text{if } i \geq 0 \\ \mathsf{eos}, & \text{otherwise} \end{cases}$$

As in [15], the instance relation on states is related to the matching relation on the corresponding terms. If $s' \sqsubseteq s$ and the call stack of s has size n, then $\mathrm{ts}(s)$ matches the subterm of $\mathrm{ts}(s')$ that encodes the upper n frames of the call stack. Hence, if one generates rewrite rules to evaluate $\mathrm{ts}(s)$, then they can also be applied to $\mathrm{ts}(s')$. Here, one of course has to label the function symbols in $\mathrm{ts}(s)$ and $\mathrm{ts}(s')$ in the same way. To this end, let $\mathrm{ts}_s(s')$ be a copy of $\mathrm{ts}(s')$ where all symbols are labeled by s instead of s'. Consider Fig. 2, where $P \sqsubseteq A$ and where the call stacks of P and A have size 2 and 1, respectively. Here, $\mathrm{ts}(A) = \mathsf{f}_{A,0}(\mathsf{eos}, \mathsf{List}(o_2), i_3, \mathsf{List}(o_2), i_3)$ matches $\mathrm{ts}_A(P)|_1 = \mathsf{f}_{A,0}(\mathsf{eos}, \mathsf{List}(o_5), i_{10}, \mathsf{List}(o_5), i_{10})$.

To ease presentation,[11] we assume that frames of the same method refer to the "same" input arguments. More precisely, let $fr = (pp, in, lv, os)$ and $fr' = (pp', in', lv', os')$ be frames with pp and pp' in the same method. If $in = \{(r_1, \tau_1, b_1), \ldots, (r_k, \tau_k, b_k)\}$, then we assume that $in' = \{(r'_1, \tau_1, b'_1), \ldots, (r'_k, \tau_k, b'_k)\}$ for the *same* positions $\tau_1, \ldots, \tau_k$. When encoding fr and fr' to terms t and t' in Def. 3.1, we fix a total order on positions $\tau_1, \ldots, \tau_k$. Then the argument positions that correspond to (r_i, τ_i, b_i) in t and to (r'_i, τ_i, b'_i) in t' are the same.

▶ **Lemma 3.2.** *Let $s' \sqsubseteq s$ and let $i = |s'| - |s|$ be the difference of their call stack sizes. Then there is a substitution σ with $\mathrm{ts}(s)\sigma = \mathrm{ts}_s(s')|_{1^i}$. Here, "$1^i$" means "$1\,1\ldots1$" (i times).*

Now we construct ITRSs whose termination implies termination of the original programs. To this end, we transform the edges of the termination graph into rewrite rules.

[10] In general, tr also takes the class hierarchy into account. To simplify the presentation, we refer to [15, Def. 3.3] for details and use the above representation in the illustrating examples.

[11] Without this assumption, $s' \sqsubseteq s$ would not imply that $\mathrm{ts}(s)$ matches a subterm of $\mathrm{ts}_s(s')$. Instead, one first would have to *expand* $\mathrm{ts}_s(s')$ by the additional input arguments of s that are missing in s'. The remaining construction and Thm. 3.3 are easily adapted accordingly (but it complicates the presentation).

If there is an *evaluation edge* from s to $\tilde{s}$, then we generate the rule $\mathrm{ts}(s) \to \mathrm{ts}(\tilde{s})$ which rewrites any instance of s to the corresponding instance of $\tilde{s}$. As in [15], if this edge is labeled with $o_1 = o_2 \circ o_3$ where $\circ \in \mathcal{A}rith\mathcal{O}p$, then in $\mathrm{ts}(\tilde{s})$ we replace o_1 by $\mathrm{tr}(s, o_2) \circ \mathrm{tr}(s, o_3)$. If the edge is labeled by $o_1 \circ o_2$ where $\circ \in \mathcal{R}el\mathcal{O}p$, then we add the condition $\mathrm{tr}(s, o_1) \circ \mathrm{tr}(s, o_2)$ to the generated rule. So the edge from H to I in Fig. 2 results in

$$\mathsf{f}_{H,8}(\mathsf{eos}, \mathsf{List}(\mathsf{null}), i_7, \mathsf{List}(\mathsf{null}), i_7, i_7) \;\to\; \mathsf{f}_{I,12}(\mathsf{eos}, \mathsf{List}(\mathsf{null}), i_7, \mathsf{List}(\mathsf{null}), i_7) \quad | \quad i_7 > 0$$

If there is an *instance edge* from s to $\tilde{s}$, then in the resulting rule we keep all information that we already have for the specialized state s and continue rewriting with the rules we already created for $\tilde{s}$. So instead of $\mathrm{ts}(s) \to \mathrm{ts}(\tilde{s})$, we generate the rule $\mathrm{ts}(s) \to \mathrm{ts}_{\tilde{s}}(s)$. For example, for the instance edge from L to N, we generate the rule

$$\mathsf{f}_{L,26}(\mathsf{eos}, \mathsf{List}(\mathsf{List}(\mathsf{null})), i_7, \mathsf{List}(\mathsf{List}(\mathsf{null})), i_8) \;\to\; \mathsf{f}_{N,26}(\mathsf{eos}, \mathsf{List}(\mathsf{List}(\mathsf{null})), i_7, \mathsf{List}(\mathsf{List}(\mathsf{null})), i_8)$$

Similarly, if there is a *refinement edge* from s to $\tilde{s}$, then $\tilde{s}$ is a specialized version of s. These edges represent a case analysis and hence, some instances of s are also instances of $\tilde{s}$, but others are no instances of $\tilde{s}$. By Lemma 3.2, we can use pattern matching to perform the necessary case analysis. Thus, instead of $\mathrm{ts}(s) \to \mathrm{ts}(\tilde{s})$ we generate the rule $\mathrm{ts}_s(\tilde{s}) \to \mathrm{ts}(\tilde{s})$. As an example, the instance refinement from B to D results in the rule

$$\mathsf{f}_{B,4}(\mathsf{eos}, \mathsf{List}(\mathsf{null}), i_3, \mathsf{List}(\mathsf{null}), i_3, \mathsf{null}) \;\to\; \mathsf{f}_{D,4}(\mathsf{eos}, \mathsf{List}(\mathsf{null}), i_3, \mathsf{List}(\mathsf{null}), i_3, \mathsf{null})$$

If there is a *call edge* from s to $\tilde{s}$, then $\tilde{s}$ only contains the top frame of the call stack of s. Here, we also generate the rule $\mathrm{ts}_s(\tilde{s}) \to \mathrm{ts}(\tilde{s})$. So for the edge from P to Q, we get

$$\mathsf{f}_{P,0}(\mathsf{eos}, \mathsf{List}(o_5), i_{10}, \mathsf{List}(o_5), i_{10}) \;\to\; \mathsf{f}_{Q,0}(\mathsf{eos}, \mathsf{List}(o_5), i_{10}, \mathsf{List}(o_5), i_{10})$$

Now this rule and the other appE-rules can be applied in terms like $\mathsf{f}_{P,34}(\underline{\mathsf{f}_{P,0}(\mathsf{eos}, \ldots)}, \ldots)$ to rewrite the underlined subterm that represents a recursive call of appE. By applying all rules corresponding to the edges from Q up to P, one then obtains $\mathsf{f}_{P,34}(\mathsf{f}_{P,34}(\mathsf{f}_{P,0}(\mathsf{eos}, \ldots), \ldots), \ldots)$. So the rules resulting from a termination graph can create call stacks of arbitrary size.

For a *context concretization edge* from s to $\tilde{s}$ with the call-state $\overline{s}$, the left-hand side of the corresponding rule should essentially represent the state where the method in the top frame of $\overline{s}$ has been called and its execution reached the `return` statement in s. So the left-hand side should be like $\mathrm{ts}(\overline{s})$, but the subterm at position $\pi = 1^{|\overline{s}|-1}$ (which encodes the top stack frame of $\overline{s}$) is replaced by $\mathrm{ts}(s)$. Hence, we obtain $\mathrm{ts}(\overline{s})[\mathrm{ts}(s)]_\pi$. Note that in the new state $\tilde{s}$, we used the identification substitution σ for the references from s and $\overline{s}$, cf. Def. 2.2. Therefore, in the corresponding rewrite rule, we should use the new names of these references not only on the right-hand side of the rule (which results from encoding $\tilde{s}$), but also on the left-hand side. In other words, we create the rule $(\mathrm{ts}(\overline{s})[\mathrm{ts}(s)]_\pi)\sigma \to \mathrm{ts}(\tilde{s})$.

As an example, let s be the return state V, $\overline{s}$ be the call state P, and $\tilde{s}$ be the context concretization W. We abbreviate "List" by "L". Then for the edge from V to W, we get

$$\begin{array}{l}\mathsf{f}_{P,34}(\,\mathsf{f}_{V,34}(\mathsf{eos}, \mathsf{L}(\mathsf{L}(o_{20}^W)), i_{19}^W, \mathsf{L}(\mathsf{L}(o_{20}^W)), i_{21}^W),\;\; \mathsf{L}(\mathsf{L}(o_5^W)),\quad i_9^W,\; \mathsf{L}(\mathsf{L}(o_5^W)),\quad i_{19}^W) \;\to \\ \mathsf{f}_{W,34}(\,\mathsf{f}_{W,34}(\mathsf{eos}, \mathsf{L}(\mathsf{L}(o_{20}^W)), i_{19}^W, \mathsf{L}(\mathsf{L}(o_{20}^W)), i_{21}^W),\; \mathsf{L}(\mathsf{L}(\mathsf{L}(o_{20}^W))),\; i_9^W,\; \mathsf{L}(\mathsf{L}(\mathsf{L}(o_{20}^W))),\; i_{19}^W)\end{array}$$

Note that on the left-hand side of this rule, for the lower stack frames of P, we still have the values *before* the execution of the method (then, o_1 had the value $\mathsf{L}(\mathsf{L}(o_5))$ in P). The reason is that when simulating the evaluation of states via term rewriting, our rules only modify the subterm corresponding to the top stack frame, until the method of the top frame

reaches a `return`. At that point, we perform all side effects that were caused by the executed method and modify the objects in lower stack frames accordingly. Therefore, the above rule performs the side effect of changing the object at o_1 from $\mathsf{L}(\mathsf{L}(o_5))$ to $\mathsf{L}(\mathsf{L}(\mathsf{L}(o_{20})))$.[12]

As explained in [15], to simplify the resulting TRS, one can often *merge* rules (where essentially, a rule $\ell \to r \mid b$ is used to narrow all right-hand sides where it is applicable and afterwards, the rule is removed). In this way, the termination graph for `appE` of Fig. 2 is transformed into the following ITRS. The rules correspond to the paths from state A via D and F to G (rule (1)), from A via D, H, and P back to A (rule (2)), from A via C and P back to A (rule (3)), from G to V (rule (4)), and from V via W back to V (rule (5)). To ease readability, we omitted "eos" and the arguments for local variables and operand stack entries from the rules. Moreover, we abbreviated "null" by "n".

$$\mathsf{f}_{A,0}(\mathsf{L}(\mathsf{n}),\, i_6) \to \mathsf{f}_{G,11}(\mathsf{L}(\mathsf{n}),\, i_6) \qquad |i_6 \le 0 \quad (1)$$

$$\mathsf{f}_{A,0}(\mathsf{L}(\mathsf{n}),\, i_7) \to \mathsf{f}_{P,34}(\,\mathsf{f}_{A,0}(\mathsf{L}(\mathsf{n}), i_7 - 1),\, \mathsf{L}(\mathsf{L}(\mathsf{n})),\, i_7) \;|i_7 > 0 \quad (2)$$

$$\mathsf{f}_{A,0}(\mathsf{L}(\mathsf{L}(o_5)),\, i_3) \to \mathsf{f}_{P,34}(\,\mathsf{f}_{A,0}(\mathsf{L}(o_5), i_3),\, \mathsf{L}(\mathsf{L}(o_5)),\, i_3) \quad (3)$$

$$\mathsf{f}_{P,34}(\,\mathsf{f}_{G,11}(\mathsf{L}(\mathsf{n}), i_{12}),\, \mathsf{L}(\mathsf{L}(\mathsf{n})),\, i_9) \to \mathsf{f}_{V,34}(\mathsf{L}(\mathsf{L}(\mathsf{n})),\, i_9) \quad (4)$$

$$\mathsf{f}_{P,34}(\,\mathsf{f}_{V,34}(\mathsf{L}(\mathsf{L}(o_{20})), i_{19}),\, \mathsf{L}(\mathsf{L}(o_5)),\, i_9) \to \mathsf{f}_{V,34}(\mathsf{L}(\mathsf{L}(\mathsf{L}(o_{20}))),\, i_9) \quad (5)$$

These rules are a natural representation of the original JBC algorithm as a TRS. Rules (1) and (2) handle the case where the length of the input list is 1 (i.e., `n == null`). If the integer parameter `i` is `<= 0`, then we immediately return (rule (1)). Otherwise, in rule (2) a new element is attached to the input list (i.e., now the input list is $\mathsf{L}(\mathsf{L}(\mathsf{n}))$, and the algorithm is called recursively with the tail of the list (i.e., again with $\mathsf{L}(\mathsf{n})$) and with `i - 1`. In rule (3), the input list has length ≥ 2. Here, the algorithm is called recursively with the tail of the list, whereas the integer parameter is unchanged. Rules (4) and (5) state that after the execution of the recursive call `n.appE(i)`, the list that results from this recursive call (e.g., $\mathsf{L}(\mathsf{L}(o_{20}))$ in rule (5)) is written to the field `n` of the current list as a side effect (e.g., in rule (5), the subterm $\mathsf{L}(o_5)$ in the current list $\mathsf{L}(\underline{\mathsf{L}(o_5)})$ is replaced by $\mathsf{L}(\mathsf{L}(o_{20}))$).

Termination of this ITRS can easily be proved automatically. In the only recursive rules (2) and (3), either the number in the second argument or the length of the list in the first argument of $\mathsf{f}_{A,0}$ decreases. As mentioned before, termination of `appE` can also be proved by Julia and COSTA, because here it suffices to compare arguments by their path-length. However, if lists or other data objects have to be compared in a different way, tools like Julia and COSTA fail, whereas rewrite techniques can compare arbitrary forms of terms, cf. Sect. 4.

Note that in [5, 15], JBC was transformed into TRSs where defined symbols (except pre-defined operations on integers and booleans) only occur on root positions. So instead of a term like $\mathsf{f}_{P,34}(\underline{\mathsf{f}_{A,0}(\mathsf{L}(\mathsf{n}), i_7 - 1)},\, \mathsf{L}(\mathsf{L}(\mathsf{n})),\, i_7)$ on the right-hand side of rule (2), we would generate a term $\mathsf{f}_{PA}(\mathsf{L}(\mathsf{n}), i_7 - 1,\, \mathsf{L}(\mathsf{L}(\mathsf{n})),\, i_7)$ for a new symbol f_{PA}. The disadvantage is that then it is not possible to re-use TRSs and their termination proofs for auxiliary methods that are called in the current method (i.e., one cannot prove termination in a *modular* way).

So for `cappE` from Sect. 2.3, with our new approach the rule for the call of `appE` is $\mathsf{f}_{A',14}(\ldots) \to \mathsf{f}_{B',17}(\underline{\mathsf{f}_{A,0}(\mathsf{L}(\mathsf{n}), i_1)}, \ldots)$ and the rule for its return is $\mathsf{f}_{B',17}(\underline{\mathsf{f}_{V,34}(\mathsf{L}(\mathsf{L}(o_6)), i_3)}, \ldots)$

[12] So for objects that were changed during the execution of the method, the information from $\tilde{s}$ may not be used on the left-hand side of the resulting rewrite rule. However, one could improve the generation of the left-hand-sides by allowing to use the information from $\tilde{s}$ for those references which were not changed by the method (i.e., where the information in $\tilde{s}$ results from the intersection of the corresponding information in s and $\overline{s}$). Then for the edge from G to R, one would obtain a rule where instead of the left-hand side $\mathsf{f}_{P,34}(\mathsf{f}_{G,11}(\ldots), \mathsf{L}(\mathsf{L}(o_2^R)), i_9^R, \mathsf{L}(\mathsf{L}(o_2^R)), i_{12}^R)$ one has the left-hand side $\mathsf{f}_{P,34}(\mathsf{f}_{G,11}(\ldots), \mathsf{L}(\mathsf{L}(\mathsf{null})), i_9^R, \mathsf{L}(\mathsf{L}(\mathsf{null})), i_{12}^R)$. We used this improvement in rule (4) above.

$\to \mathsf{f}_{C',17}(\mathsf{f}_{C',34}(\ldots),\ldots)$. The rules for $\mathsf{f}_{A,0}$ and the other function symbols from `appE` remain unchanged and can be re-used. Hence, their (innermost) termination proof can also be re-used. Since the remaining rules for `cappE` have no recursion, termination of the `cappE`-TRS trivially follows from termination of the `appE`-TRS. This illustrates the advantages of our modular approach which leads to TRSs that form *hierarchical combinations*. Hence, one can benefit from termination methods like the *dependency pair* technique that prove innermost termination of hierarchical combinations in a modular way, cf. [11, 12, 13]. Note that while COSTA and Julia can prove termination of `appE`, they fail on `cappE`.

Using Lemma 3.2, we can now prove that every computation path in a termination graph can be simulated by a rewrite sequence with the corresponding ITRS.

▶ **Theorem 3.3** (Soundness of ITRS Translation). *If the ITRS corresponding to a termination graph G is terminating, then G has no infinite computation path.*

As explained at the end of Sect. 2.3, by combining Thm. 3.3 with Thm. 2.3, we obtain that termination of the resulting ITRS implies termination of the original JBC program for all concrete states represented in the termination graph. Of course, the converse does not hold, i.e., our approach cannot be used to prove non-termination of JBC. Future work will be concerned with using our termination graphs also for non-termination analysis, as well as for other analyses like absence of null pointer exceptions and side effect freeness.

4 Experiments and Conclusion

We presented a new approach to prove termination of JBC programs automatically. In contrast to our earlier work [5, 15], we introduced a technique (based on *context concretizations*) that abstracts from the exact form of the call stack. In this way, we can now also analyze *recursive* methods, which were excluded in [5, 15]. Moreover, we obtain a *modular* approach, since one can now generate termination graphs for different methods separately and re-use them whenever a method is called. In contrast to [5, 15], we now also synthesize TRSs from the termination graphs whose termination can be proved in a modular way.

We implemented our new approach in the termination tool AProVE [10] and evaluated it on a collection of 83 recursive and 133 non-recursive JBC programs. These examples contain the 172 JBC programs from the *Termination Problem Data Base* (used in the *International Termination Competition*)[13] as well as a number of additional typical recursive programs.[14] Below, we compare AProVE 2011 (which contains all contributions of this paper), AProVE 2010 (which implements [5, 15]),[15] Julia [16], and COSTA [2]. We used a runtime of 2 minutes for each example. "**Y**es" indicates how many examples could be proved, "**F**ail" states how often the tool failed in less than 2 minutes, "**T**" indicates how many examples led to a **T**ime-out, and "**R**" gives the average **R**untime in seconds for each example.

	recursion				no recursion			
	Y	**F**	**T**	**R**	**Y**	**F**	**T**	**R**
AProVE 2011	67	0	16	30	108	0	25	27
AProVE 2010	15	3	65	96	103	13	17	23
Julia	57	26	0	3	96	37	0	2
COSTA	47	35	1	6	73	60	0	5

So due to our new modular approach, AProVE 2011 yields the most precise results for the

[13] We removed one controversial example whose termination depends on the handling of integer overflows.
[14] Of course, we also included `appE` and `cappE`, and AProVE 2011 easily proves termination of them.
[15] In addition, whenever a recursive method is called with *fixed* inputs, AProVE 2010 tries to evaluate it. But it cannot prove termination of recursive method for (infinite) *sets* of possible inputs.

recursive JBC programs in the collection. (However, there are also several examples where Julia or COSTA succeed whereas AProVE fails.) On non-recursive programs, AProVE 2010 was already powerful (but the modularity of our new approach helps in large examples). Of course, Julia and COSTA are significantly faster than AProVE. This is because Julia and COSTA use a *fixed* abstraction from objects to integers, whereas AProVE applies rewrite techniques to generate (potentially different) suitable well-founded orders in every termination proof. Nevertheless, the experiments clearly show that rewrite techniques are not only powerful, but also efficient enough for termination of JBC. So a fruitful approach for the future could be to couple the rewrite-based approach of AProVE with the technique of Julia and COSTA to combine their respective advantages. To experiment with our implementation via a web interface and for details on the experiments, we refer to [1].

Acknowledgement. We are grateful to F. Spoto and S. Genaim for help with the experiments and to the referees for many helpful suggestions.

References

1 `http://aprove.informatik.rwth-aachen.de/eval/JBC-Recursion/`.
2 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.
3 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.
4 M. Brockschmidt, C. Otto, and J. Giesl. Modular termination proofs of recursive Java Bytecode programs by term rewriting. Technical Report AIB 2011-02, RWTH Aachen, 2011. Available at [1] and at `http://aib.informatik.rwth-aachen.de`.
5 M. Brockschmidt, C. Otto, C. von Essen, and J. Giesl. Termination graphs for Java Bytecode. In *Verification, Induction, Termination Analysis*, LNCS 6463, pages 17–37, 2010. Extended version (with proofs) available at [1].
6 M. Colón and H. Sipma. Practical methods for proving program termination. In *Proc. CAV '02*, LNCS 2404, pages 442–454, 2002.
7 B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *Proc. PLDI '06*, pages 415–426. ACM Press, 2006.
8 B. Cook, A. Podelski, and A. Rybalchenko. Summarization for termination: No return! *Formal Methods in System Design*, 35(3):369–387, 2009.
9 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.
10 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.
11 J. Giesl, T. Arts, and E. Ohlebusch. Modular termination proofs for rewriting using dependency pairs. *Journal of Symbolic Computation*, 34(1):21–58, 2002.
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 N. Hirokawa and A. Middeldorp. Automating the dependency pair method. *Information and Computation*, 199(1,2):172–199, 2005.
14 T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Prentice Hall, 1999.
15 C. Otto, M. Brockschmidt, C. von Essen, and J. Giesl. Automated termination analysis of Java Bytecode by term rewriting. In *Proc. RTA '10*, LIPIcs 6, pages 259–276, 2010. Extended version (with proofs) available at [1].
16 F. Spoto, F. Mesnard, and É. Payet. A termination analyser for Java Bytecode based on path-length. *ACM TOPLAS*, 32(3), 2010.

Rewriting-based Quantifier-free Interpolation for a Theory of Arrays

Roberto Bruttomesso[1], Silvio Ghilardi[2], and Silvio Ranise[3]

1 **Università della Svizzera Italiana, Lugano, Switzerland**
2 **Università degli Studi di Milano, Milan, Italy**
3 **FBK (Fondazione Bruno Kessler), Trento, Italy**

Abstract

The use of interpolants in model checking is becoming an enabling technology to allow fast and robust verification of hardware and software. The application of encodings based on the theory of arrays, however, is limited by the impossibility of deriving quantifier-free interpolants in general. In this paper, we show that, with a minor extension to the theory of arrays, it is possible to obtain quantifier-free interpolants. We prove this by designing an interpolating procedure, based on solving equations between array updates. Rewriting techniques are used in the key steps of the solver and its proof of correctness. To the best of our knowledge, this is the first successful attempt of computing quantifier-free interpolants for a theory of arrays.

Digital Object Identifier 10.4230/LIPIcs.RTA.2011.171

Category Regular Research Paper

1 Introduction

After the seminal work of McMillan (see, e.g., [20]), Craig's interpolation [9] has become an important technique in verification. For example, the importance of computing *quantifier-free* interpolants to over-approximate the set of reachable states for model checking has been observed. Unfortunately, Craig's interpolation theorem does not guarantee that it is always possible to compute quantifier-free interpolants. Even worse, for certain first-order theories, it is known that quantifiers must occur in interpolants of quantifier-free formulae [15]. As a consequence, a lot of effort has been put in designing efficient procedures for the computation of quantifier-free interpolants for first-order theories which are relevant for verification (e.g., uninterpreted functions and fragments of Presburger arithmetics). Despite these efforts, so far, only the negative result in [15] is available for the computation of interpolants in the theory of arrays with extensionality, axiomatized by the following three sentences: $\forall y, i, e.rd(wr(y, i, e), i) = e$, $\forall y, i, j, e.i \neq j \Rightarrow rd(wr(y, i, e), j) = rd(y, j)$, and

$$\forall x, y.x \neq y \Rightarrow (\exists i.\ rd(x, i) \neq rd(y, i)),$$

where rd and wr are the usual operations for reading and updating arrays, respectively. This theory is important for both hardware and software verification, and a procedure for computing quantifier-free interpolants "*would extend the utility of interpolant extraction as a tool in the verifier's toolkit*" [20]. Indeed, the endeavour of designing such a procedure would be bound to fail (according to [15]) if we restrict ourselves to the original theory. To circumvent the problem, we replace the third axiom above with its Skolemization, i.e.,

$$\forall x, y.x \neq y \Rightarrow rd(x, \texttt{diff}(x, y)) \neq rd(y, \texttt{diff}(x, y))),$$

so that the Skolem function `diff` is supposed to return an index at which the elements stored in two distinct arrays are different. This variant of the theory of arrays admits quantifier-free interpolants for quantifier-free formulae. The main contribution of the paper is to prove

22nd International Conference on Rewriting Techniques and Applications.
Editor: M. Schmidt-Schauß; pp. 171–186
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

this by designing an **algorithm for the generation of quantifier-free interpolants from finite sets** (intended conjunctively) **of literals in the theory of arrays with** `diff`. The algorithm uses as a sub-module a satisfiability procedure for sets of literals of the theory, based on a sequence of syntactic transformations organized in several groups. The most important group of such transformations is a Knuth-Bendix completion procedure (see, e.g., [2]) extended to solve an equation $a = wr(b, i, e)$ for b when this is required by the ordering defined on terms. The goal of these transformations is to produce a "modularized" constraint for which it is trivial to establish satisfiability. To compute interpolants, the satisfiability procedure is invoked on two mutually unsatisfiable sets A and B of literals. While running, the two instances of the procedure exchange literals on the common signature of A and B (similarly to the Nelson and Oppen combination method, see, e.g., [21]) and perform some additional operations. At the end of the computation, the execution trace is examined and the desired interpolant is built by applying simple rules manipulating Boolean combinations of literals in the common signature of A and B.

The paper is organized as follows. In §2, we recall some background notions and introduce the notation. In §3, we give the notion of modularized constraint and state its key properties. In §4, we describe the satisfiability solver for the theory of arrays with `diff` and extend it to produce interpolants in §5. Finally, we discuss the related work and conclude in §6. All proofs can be found in [5].

2 Background and Preliminaries

We assume the usual syntactic (e.g., signature, variable, term, atom, literal, formula, and sentence) and semantic (e.g., structure, truth, satisfiability, and validity) notions of first-order logic. The equality symbol "=" is included in all signatures considered below. For clarity, we shall use "≡" in the meta-theory to express the syntactic identity between two symbols or two strings of symbols.

A *theory* T is a pair (Σ, Ax_T), where Σ is a signature and Ax_T is a set of Σ-sentences, called the axioms of T (we shall sometimes write directly T for Ax_T). The Σ-structures in which all sentences from Ax_T are true are the *models* of T. A Σ-formula ϕ is *T-satisfiable* if there exists a model $\mathcal{M}$ of T such that ϕ is true in $\mathcal{M}$ under a suitable assignment $\mathtt{a}$ to the free variables of ϕ (in symbols, $(\mathcal{M}, \mathtt{a}) \models \phi$); it is *$T$-valid* (in symbols, $T \vdash \varphi$) if its negation is T-unsatisfiable or, equivalently, iff φ is provable from the axioms of T in a complete calculus for first-order logic. A formula φ_1 *T-entails* a formula φ_2 if $\varphi_1 \to \varphi_2$ is *T-valid*; the notation used for such T-entailment is $A \vdash_T B$ or simply $A \vdash B$, if T is clear from the context. The *satisfiability modulo the theory* T ($SMT(T)$) *problem* amounts to establishing the T-satisfiability of quantifier-free Σ-formulae.

Let T be a theory in a signature Σ; a *T-constraint* (or, simply, a constraint) A is a set of ground literals in a signature Σ' obtained from Σ by adding a set of free constants. Taking conjunction, we can see a finite constraint A as a single formula; thus, when we say that a constraint A is *T-satisfiable* (or just "satisfiable" if T is clear from the context), we mean that the associated formula (also called A) is satisfiable in a Σ'-structure which is a model of T. We have two notions of equivalence between constraints, which are summarized in the next definition:

▶ **Definition 2.1.** Let A and B be finite constraints (or, more generally, first order sentences) in an expanded signature. We say that A and B are *logically equivalent* (modulo T) iff $T \vdash A \leftrightarrow B$; on the other hand, we say that they are *$\exists$-equivalent* (modulo T) iff $T \vdash A^{\exists} \leftrightarrow B^{\exists}$, where $A^{\exists}$ (and similarly $B^{\exists}$) is the formula obtained from A by replacing free constants

with variables and then existentially quantifying them out.

Logical equivalence means that the constraints have the same semantic content (modulo T); $\exists$-equivalence is also useful because we are mainly interested in T-satisfiability of constraints and it is trivial to see that $\exists$-equivalence implies equi-satisfiability (again, modulo T). As an example, if we take a constraint A, we replace all occurrences of a certain term t in it by a fresh constant a and add the equality $a = t$, called the *(explicit) definition (of t)*, the constraint A' we obtain in this way is $\exists$-equivalent to A. As another example, suppose that $A \vdash_T a = t$, that a does not occur in t, and that A' is obtained from A by replacing a by t everywhere; then the following four constraints are $\exists$-equivalent

$$A, \quad A \cup \{a = t\}, \quad A' \cup \{a = t\}, \quad A'$$

(the first three are also pairwise logically equivalent). The above examples show how explicit definitions can be introduced and removed from constraints while preserving $\exists$-equivalence.

Theories of Arrays. In this paper, we consider a variant of a three-sorted theory of arrays defined as follows. The McCarthy *theory of arrays* $\mathcal{AX}$ [17] has three sorts `ARRAY`, `ELEM`, `INDEX` (called "array", "element", and "index" sort, respectively) and two function symbols rd and wr of appropriate arities; its axioms are:

$$\forall y, i, e. \quad rd(wr(y, i, e), i) = e \tag{1}$$

$$\forall y, i, j, e. \quad i \neq j \Rightarrow rd(wr(y, i, e), j) = rd(y, j). \tag{2}$$

The theory of *arrays with extensionality* $\mathcal{AX}_{\text{ext}}$ has the further axiom $\forall x, y. x \neq y \Rightarrow (\exists i.\ rd(x, i) \neq rd(y, i))$ (called the 'extensionality' axiom). To build the *theory of arrays with* `diff` $\mathcal{AX}_{\texttt{diff}}$, we need a further function symbol `diff` in the signature and we replace the extensionality axiom by its Skolemization

$$\forall x, y. \quad x \neq y \Rightarrow rd(x, \texttt{diff}(x, y)) \neq rd(y, \texttt{diff}(x, y)). \tag{3}$$

As it is evident from axiom (3), the new symbol `diff` is a binary function of sort `INDEX` taking two arguments of sort `ARRAY`: its semantics is a function producing an index where the input arguments differ (it has an arbitrary value in case the input arguments are equal).

We introduce here some notational conventions which are specific for constraints in our theory $\mathcal{AX}_{\texttt{diff}}$. We use $a, b, \ldots$ to denote free constants of sort `ARRAY`, $i, j, \ldots$ for free constants of sort `INDEX`, and $d, e, \ldots$ for free constants of sort `ELEM`; $\alpha, \beta, \ldots$ stand for free constants of any sort. Below, we shall introduce non-ground rewriting rules involving (universally quantified) variables of sort `ARRAY`: for these variables, we shall use the symbols $x, y, z, \ldots$. We make use of the following abbreviations.

- [Nested write terms] By $wr(a, I, E)$ we indicate a nested write on the array variable a, where indexes are represented by the free constants list $I \equiv i_1, \ldots, i_n$ and elements by the free constants list $E \equiv e_1, \ldots, e_n$; more precisely, $wr(a, I, E)$ abbreviates the term $wr(wr(\cdots wr(a, i_1, e_1) \cdots), i_n, e_n)$. Notice that, whenever the notation $wr(a, I, E)$ is used, the lists I and E must have the same length; for empty I, E, the term $wr(a, I, E)$ conventionally stands for a.
- [Multiple read literals] Let a be a constant of sort `ARRAY`, $I \equiv i_1, \ldots, i_n$ and $E \equiv e_1, \ldots, e_n$ be lists of free constants of sort `INDEX` and `ELEM`, respectively; $rd(a, I) = E$ abbreviates the formula $rd(a, i_1) = e_1 \wedge \cdots \wedge rd(a, i_n) = e_n$.
- [Multiple equalities] If $L \equiv \alpha_1, \ldots, \alpha_n$ and $L' \equiv \alpha'_1, \ldots, \alpha'_n$ are lists of constants of the same sort, by $L = L'$ we indicate the formula $\bigwedge_{i=1}^{n} \alpha_i = \alpha'_i$.

Refl	$wr(a,I,E) = a \leftrightarrow rd(a,I) = E$ *Proviso*: $Distinct(I)$
Symm	$(wr(a,I,E) = b \wedge rd(a,I) = D) \leftrightarrow (wr(b,I,D) = a \wedge rd(b,I) = E)$ *Proviso*: $Distinct(I)$
Trans	$(a = wr(b,I,E) \wedge b = wr(c,J,D)) \leftrightarrow (a = wr(c, J \cdot I, D \cdot E) \wedge b = wr(c,J,D))$
Confl	$b = wr(a, I \cdot J, E \cdot D) \wedge b = wr(a, I \cdot H, E' \cdot F) \leftrightarrow$ $\leftrightarrow (b = wr(a,I,E) \wedge E = E' \wedge rd(a,J) = D \wedge rd(a,H) = F)$ *Proviso*: $Distinct(I \cdot J \cdot H)$
Red	$(a = wr(b,I,E) \wedge rd(b,i_k) = e_k) \leftrightarrow (a = wr(b, I-k, E-k) \wedge rd(b,i_k) = e_k)$ *Proviso*: $Distinct(I)$

Legenda: a and b are constants of sort `ARRAY`; $I \equiv i_1, \ldots, i_n$, $J \equiv j_1, \ldots, j_m$ and $H \equiv h_1, \ldots, h_l$ are lists of constants of sort `INDEX`; $E \equiv e_1, \ldots, e_n$, $E' \equiv e'_1, \ldots, e'_n$, $D \equiv d_1, \ldots, d_m$, and $F \equiv f_1, \ldots, f_l$ are lists of constants of sort `ELEM`.

Figure 1 Key properties of write terms

- [Multiple distinctions] If $L \equiv \alpha_1, \ldots, \alpha_n$ is a list of constants of the same sort, by $Distinct(L)$ we abbreviate the formula $\bigwedge_{i \neq j} \alpha_i \neq \alpha_j$.
- [Juxtaposition and subtraction] If $L \equiv \alpha_1, \ldots, \alpha_n$ and $L' \equiv \alpha'_1, \ldots, \alpha'_m$ are lists of constants, by $L \cdot L'$ we indicate the list $\alpha_1, \ldots, \alpha_n, \alpha'_1, \ldots, \alpha'_m$; for $1 \leq k \leq n$, the list $L - k$ is the list $\alpha_1, \ldots, \alpha_{k-1}, \alpha_{k+1}, \ldots, \alpha_n$.

Some key properties of equalities involving write terms are stated in the following lemma (see also Figure 1).

▶ **Lemma 2.2** (Key properties of write terms). *The formulae in Figure 1 are all $\mathcal{AX}_{\texttt{diff}}$-valid under the assumption that their provisoes - if any - hold (when we say that a formula ϕ is $\mathcal{AX}_{\texttt{diff}}$-valid under the proviso π, we just mean that $\pi \vdash_{\mathcal{AX}_{\texttt{diff}}} \phi$).*

A (ground) *flat* literal is a literal of the form $a = wr(b,I,E), rd(a,i) = e, \texttt{diff}(a,b) = i, \alpha = \beta, \alpha \neq \beta$. Notice that replacing a sub-term t with a fresh constant α in a constraint A and adding the corresponding defining equation $\alpha = t$ to A always produces an $\exists$-equivalent constraint; by repeatedly applying this method, one can show that every constraint is $\exists$-equivalent to a *flat* constraint, i.e., to one containing only flat literals. We split a flat constraint A into two parts, the *index* part A_I and the *main* part A_M: A_I contains the literals of the form $i = j, i \neq j, \texttt{diff}(a,b) = i$, whereas A_M contains the remaining literals, i.e., those of the form $a = wr(b,I,E), a \neq b, rd(a,i) = e, e = d, e \neq d$ (atoms $a = b$ are identified with literals $a = wr(b, \emptyset, \emptyset)$). We write $A =< A_I, A_M >$ to indicate the two parts of the constraint A.

3 Constraints combination

We shall need basic term rewriting system terminology and results: the reader is referred to [2] for the required background. In the main part of a constraint, positive literals will be treated as rewrite rules; to get a suitable orientation, we use a *lexicographic path ordering* with a total precedence $>$ such that $a > wr > rd > \texttt{diff} > i > e$, for all a, i, e of the corresponding sorts. This choice orients equalities $a = wr(b,I,E)$ *from left to right* when $a > b$; equalities like $a = wr(b,I,E)$ for $a < b$ or $a \equiv b$ will be called *badly orientable* equalities. Our plan to derive a quantifier-free interpolation procedure for $\mathcal{AX}_{\texttt{diff}}$ relies on

the notion of "modularized constraint": after introducing such constraints, we show that their satisfiability can be easily recognized (Lemma 3.2) and that they can be combined in a modular way (Proposition 3.3).

▶ **Definition 3.1.** A constraint $A =< A_I, A_M >$ is said to be *modularized* iff it is flat and the following conditions are satisfied (we let $\tilde{I}, \tilde{E}$ be the sets of free constants of sort `INDEX` and `ELEM` occurring in A):

(o) no positive index literal $i = j$ occurs in A_I;

(i) no negative array literal $a \neq b$ occurs in A_M;

(ii) A_M does not contain badly orientable equalities;

(iii) the rewriting system A_R given by the oriented positive literals of A_M joined with the rewriting rules

$$rd(wr(x,i,e),j) \to rd(x,j) \qquad \text{for } i,j \in \tilde{I},\ e \in \tilde{E},\ i \not\equiv j \tag{4}$$

$$rd(wr(x,i,e),i) \to e \qquad \text{for } i \in \tilde{I},\ e \in \tilde{E} \tag{5}$$

$$wr(wr(x,i,e),j,d) \to wr(wr(x,j,d),i,e) \qquad \text{for } i,j \in \tilde{I},\ e,d \in \tilde{E},\ i > j \tag{6}$$

$$wr(wr(x,i,e),i,d) \to wr(x,i,d). \qquad \text{for } i \in \tilde{I},\ e,d \in \tilde{E} \tag{7}$$

is confluent and ground irreducible;[1]

(iv) if $a = wr(b,I,E) \in A_M$ and i, e are in the same position in the lists I, E, respectively, then $rd(b,i) \not\downarrow_{A_R} e$ (we use $\downarrow_{A_R}$ for joinability of terms);

(v) $\{\mathtt{diff}(a,b) = i, \mathtt{diff}(a',b') = i'\} \subseteq A_I$ and $a \downarrow_{A_R} a'$ and $b \downarrow_{A_R} b'$ imply $i \equiv i'$;

(vi) $\mathtt{diff}(a,b) = i \in A_I$ and $rd(a,i) \downarrow_{A_R} rd(b,i)$ imply $a \downarrow_{A_R} b$.

▶ Remark. Condition (o) means that the index constants occurring in a modularized constraint are implicitly assumed to denote distinct objects. This is confirmed also by the proof of Lemma 3.2 below: from which, it is evident that the addition of all the negative literals $i \neq j$ (for $i, j \in \tilde{I}$ with $i \not\equiv j$) does not compromise the satisfiability of a modularized constraint, precisely because such negative literals are implicitly part of the constraint.

In Condition (i), negative array literals $a \neq b$ are not allowed because they can be replaced by suitable literals involving fresh constants and the `diff` operation (see axiom (3)).

Rules (4) and (5) mentioned in condition (iii) reduce read-over-writes and rules (6) and (7) sort indexes in flat terms $wr(a,I,E)$ in ascending order. In addition, condition (iv) prevents further redundancies in our rules.

Conditions (v) and (vi) deal with `diff`: in particular, (v) says that `diff` is "well defined" and (vi) is a "conditional" translation of the contraposition of axiom (3).

▶ Remark. The non-ground rules from Definition 3.1(iii) form a convergent rewrite system (critical pairs are confluent): this can be checked manually (and can be confirmed also by tools like SPASS or MAUDE). Ground rules from A_R are of the form

$$a \to wr(b,I,E), \tag{8}$$

$$rd(a,i) \to e, \tag{9}$$

$$e \to d. \tag{10}$$

Only rules of the form (10) can overlap with the non-ground rules (4)-(7), but the resulting critical pairs are trivially confluent. Thus, in order to check confluence of A_M, *only overlaps*

[1] The latter means that no rule can be used to reduce the left-hand or the right-hand side of another ground rule. Notice that ground rules from A_R are precisely the rules obtained by orienting an equality from A_M (rules (4)-(7) are not ground as they contain one *variable*, namely the array variable x).

between ground rules (8)-(10) *need to be considered* (this is the main advantage of our choice to orient equalities $a = wr(b, I, E)$ from left to right instead of right to left).

▶ **Lemma 3.2.** *A modularized constraint A is $\mathcal{AX}_{\mathtt{diff}}$-satisfiable iff for no negative element equality $e \neq d$ from A_M, we have that $e \downarrow_{A_R} d$.*

Let A, B be two constraints in the signatures Σ^A, Σ^B obtained from the signature Σ by adding some free constants and let $\Sigma^C := \Sigma^A \cap \Sigma^B$. Given a term, a literal or a formula φ we call it:

- *AB-common* iff it is defined over Σ^C;
- *A-local* (resp. *B-local*) if it is defined over Σ^A (resp. Σ^B);
- *A-strict* (resp. *B-strict*) iff it is A-local (resp. B-local) but not AB-common;
- *AB-mixed* if it contains symbols in both $(\Sigma^A \setminus \Sigma^C)$ and $(\Sigma^B \setminus \Sigma^C)$;
- *AB-pure* if it does not contain symbols in both $(\Sigma^A \setminus \Sigma^C)$ and $(\Sigma^B \setminus \Sigma^C)$.

(Notice that, sometimes in the literature about interpolation, "A-local" and "B-local" are used to denote what we call here "A-strict" and "B-strict"). The following modularity result is crucial for establishing interpolation in $\mathcal{AX}_{\mathtt{diff}}$:

▶ **Proposition 3.3.** *Let $A = \langle A_I, A_M \rangle$ and $B = \langle B_I, B_M \rangle$ be constraints in expanded signatures Σ^A, Σ^B as above (here Σ is the signature of $\mathcal{AX}_{\mathtt{diff}}$); let A, B be both consistent and modularized. Then $A \cup B$ is consistent and modularized, in case all the following conditions hold:*

(O) *an AB-common literal belongs to A iff it belongs to B;*

(I) *every rewrite rule in $A_M \cup B_M$ whose left-hand side is AB-common has also an AB-common right-hand side;*

(II) *if a, b are both AB-common and $\mathtt{diff}(a, b) = i \in A_I \cup B_I$, then i is AB-common too;*

(III) *if a rewrite rule of the kind $a \to wr(c, I, E)$ is in $A_M \cup B_M$ and the term $wr(c, I, E)$ is AB-common, so is the constant a.*

4 A Solver for Arrays with `diff`

In this section we present a solver for the theory $\mathcal{AX}_{\mathtt{diff}}$. The idea underlying our algorithm is to separate the "index" part (to be treated by guessing) of a constraint from the "array" and "elem" parts (to be treated with rewriting techniques). The problem is how, given a *finite* constraint A, to determine whether it is satisfiable or not by transforming it into a modularized $\exists$-equivalent constraint.

4.1 Preprocessing

In order to establish the satisfiability of a constraint A, we first need a pre-processing phase, consisting of the following sequential steps:

Step 1 Flatten A, by replacing sub-terms with fresh constants and by adding the related defining equalities.

Step 2 Replace array inequalities $a \neq b$ by the following literals (i, e, d are fresh)

$$\mathtt{diff}(a, b) = i, \quad rd(b, i) = e, \quad rd(a, i) = d, \quad d \neq e.$$

Step 3 Guess a partition of index constants, i.e., for any pair of indexes i, j add either $i = j$ or $i \neq j$ (but not both of them); then remove the positive literals $i = j$ by replacing i by j everywhere (if $i > j$ according to the symbol precedence, otherwise replace j by

i); if an inconsistent literal $i \neq i$ is produced, try with another guess (and if all guesses fail, report `unsat`).

Step 4 For all a, i such that $rd(a, i) = e$ does not occur in the constraint, add such a literal $rd(a, i) = e$ with fresh e.

At the end of the preprocessing phase, we get a finite set of flat constraints; *the disjunction of these constraints is $\exists$-equivalent to the original constraint.* For each of these constraints, go to the completion phase: *if the transformations below can be exhaustively applied (without failure) to at least one of the constraints, report* `sat`, *otherwise report* `unsat`.

The reason for inserting Step 4 above is just to simplify Orientation and Gaussian completion below. Notice that, even if rules $rd(a, i) \to e$ can be removed during completion, the following **invariant** is maintained: *terms $rd(a, i)$ always reduce to constants of sort* `ELEM`.

4.2 Completion

The completion phase consists in various transformations that should be non-deterministically executed until no rule or a failure instruction applies. For clarity, we divide the transformations into five groups.

(I) Orientation. This group contains a single instruction: get rid of badly orientable equalities, by using the equivalences **Refl**exivity and **Symm**etry of Figure 1; a badly orientable equality $a = wr(b, I, E)$ (with $a < b$) is replaced by an equality of the kind $b = wr(a, I, D)$ and by the equalities $rd(a, I) = E$ (all "read literals" required by the left-hand side of **Symm** comes from the above invariant). A badly orientable equality $a = wr(a, I, E)$ is removed and replaced by read literals only (or by nothing if I, E are empty).

(II) Gaussian completion. We now take care of the confluence of A_R (i.e., point (iii) of Definition 3.1). To this end, we consider all the critical pairs that may arise among our rewriting rules (8)-(10) (recall that, by Remark 3, there is no need to examine overlaps involving the non ground rules (4)-(7)). To treat the relevant critical pairs, we combine standard Knuth-Bendix completion for congruence closure with a specific method ("Gaussian completion") based on equivalences **Symm**etry, **Trans**itivity and **Confl**ict of Figure 1.[2] The critical pairs are listed below. Two preliminary observations are in order. First, we normalize a critical pair by using $\to_*$ before recovering convergence by adding a suitably oriented equality and removing the parent equalities (the symbol $\to_*$ denotes the reflexive and transitive closure of the rewrite relation $\to$ induced by the rewrite rules $A_R \cup \{(4)-(7)\}$). Second, the provisoes of all the equivalences in Figure 1 used below (i.e., **Symm**, **Trans**, and **Confl**) are satisfied because of the pre-processing Step 3 above.

(C1) $wr(b_1, I_1, E_1) \;{}_*\!\leftarrow wr(b'_1, I'_1, E'_1) \leftarrow a \to wr(b'_2, I'_2, E'_2) \to_* wr(b_2, I_2, E_2)$

with $b_1 > b_2$. We proceed in two steps. First, we use **Symm** (from right to left) to replace the parent rule $a \to wr(b'_1, I'_1, E'_1)$ with

$$wr(a, I_1, F) = b_1 \wedge rd(a, I_1) = E_1$$

for a suitable list F of constants of sort `ELEM` (notice that the equalities $rd(b_1, I_1) = F$, which are required to apply **Symm**, are already available because terms of the form $rd(b_1, j)$ for j in I_1 always reduce to constants of sort `ELEM` by the invariant resulting from the application of Step 4 in the pre-processing phase). Then, we apply **Trans** to

[2] The name "Gaussian" is due to the analogy with Gaussian elimination in Linear Arithmetic (see [1,4] for a generalization to the first-order context).

the previously derived equality $b_1 = wr(a, I_1, F)$ and to the normalized second equality of the critical pair (i.e., $a = wr(b_2, I_2, E_2)$) and we derive

$$b_1 = wr(b_2, I_2 \cdot I_1, E_2 \cdot F) \wedge a = wr(b_2, I_2, E_2). \tag{11}$$

Hence, we are entitled to replace $b_1 = wr(a, I_1, F)$ with the rule $b_1 \to wr(b_2, J, D)$, where J and D are lists obtained by normalizing the right-hand-side of the first equality of (11) with respect to the non-ground rules (6) and (7). To summarize: the parent rules are removed and replaced by the rules

$$b_1 \to wr(b_2, J, D), \quad a \to wr(b_2, I_2, E_2)$$

and a bunch of new equalities of the form $rd(a, i) = e$, giving rise, in turn, to rules of the form $rd(b_2, i) \to e$ or to rewrite rules of the form (10) after normalization of their left members.

(C2) $\boxed{wr(b, I_1, E_1) \ {}_*\!\leftarrow wr(b'_1, I'_1, E'_1) \leftarrow a \to wr(b'_2, I'_2, E'_2) \to_* wr(b, I_2, E_2)}$

Since identities like $wr(c, H, G) = wr(c, \pi(H), \pi(G))$ are $\mathcal{AX}_{\texttt{diff}}$-valid for every permutation π (under the proviso $Distinct(H)$), it is harmless to suppose that the set of index variables $I := I_1 \cap I_2$ coincides with the common prefix of the lists I_1 and I_2; hence we have $I_1 \equiv I \cdot J$ and $I_2 \equiv I \cdot H$ for suitable disjoint lists J and H. Then, let E and E' be the prefixes of E_1 and E_2, respectively, of length equal to that of I; and let $E_1 \equiv E \cdot D$ and $E_2 \equiv E' \cdot F$ for suitable lists D and F. At this point, we can apply **Confl** to replace both parent rules forming the critical pair with

$$a = wr(b, I, E) \wedge E = E' \wedge rd(b, J) = D \wedge rd(b, H) = F,$$

where the first equality is oriented from left to right (i.e., $a \to wr(b, I, E)$).

(III) Knuth-Bendix completion. The remaining critical pairs are treated by standard completion methods for congruence closure.

(C3) $\boxed{d \ {}_*\!\leftarrow rd(wr(b, I, E), i) \leftarrow rd(a, i) \to e' \to_* e}$

Remove the parent rule $rd(a, i) \to e'$ and, depending on whether $d > e, e > d$, or $d \equiv e$, add the rule $d \to e$, $e \to d$, or do nothing. (Notice that terms of the form $rd(b, j)$ are always reducible because of the invariant of Step 4 in the pre-processing phase; hence, $rd(wr(b, I, E), i)$ always reduces to some constant of sort `ELEM`.)

(C4) $\boxed{e \ {}_*\!\leftarrow e' \leftarrow rd(a, i) \to d' \to_* d}$

Orient the critical pair (if e and d are not identical), add it as a new rule and remove one parent rule.

(C5) $\boxed{d \ {}_*\!\leftarrow d' \leftarrow e \to d'_1 \to_* d_1}$

Orient the critical pair (if d and d_1 are not identical), add it as a new rule and remove one parent rule.

(IV) Reduction. The instructions in this group simplify the current rewrite rules.

(R1) If the right-hand side of a current ground rewrite rule can be reduced, reduce it as much as possible, remove the old rule, and replace it with the newly obtained reduced rule. Identical equations like $t = t$ are also removed.

(R2) For every rule $a \to wr(b, I, E) \in A_M$, exhaustively apply **Red**uction in Figure 1 from left to right (this amounts to do the following: if there are i, e in the same position k in the lists I, E such that $rd(b, i) \downarrow_{A_R} e$, replace $a = wr(b, I, E)$ with $a = wr(b, I{-}k, E{-}k)$).

(R3) If $\texttt{diff}(a, b) = i \in A_I$, $rd(a, i) \downarrow_{A_R} rd(b, i)$ and $a > b$, add the rule $a \to b$; replace also $\texttt{diff}(a, b) = i$ by $\texttt{diff}(b, b) = i$ (this is needed for termination, it prevents the rule for being indefinitely applied).

(V) Failure. The instructions in this group aim at detecting inconsistency.

(U1) If for some negative literal $e \neq d \in A_M$ we have $e \downarrow_{A_R} d$, report `failure` and backtrack to Step 3 of the pre-processing phase.

(U2) If $\{\mathtt{diff}(a,b) = i, \mathtt{diff}(a',b') = i'\} \subseteq A_I$ and $a \downarrow_{A_R} a'$ and $b \downarrow_{A_R} b'$ for $i \not\equiv i'$, report `failure` and backtrack to Step 3 of the pre-processing phase.

Notice that the instructions in the last two groups may require a confluence test $\alpha \downarrow_{A_R} \beta$ that can be effectively performed in case the instructions from groups (II)-(III) have been exhaustively applied, because then all critical pairs have been examined and the rewrite system A_R is confluent. If this is not the case, one may pragmatically compute and compare any normal form of α and β, keeping in mind that the test has to be repeated when all completion instructions (II)-(III) have been exhaustively applied.

▶ **Theorem 4.1.** *The above procedure decides constraint satisfiability in* $\mathcal{AX}_{\mathtt{diff}}$.

5 The Interpolation Algorithm

In the literature one can roughly distinguish two approaches to the problem of computing interpolants. In the former (see e.g. [3, 19]), an interpolating calculus is obtained from a standard calculus by adding decorations so as to enable the recursive construction of an interpolating formula from a proof; in the latter (see, e.g., [7, 11, 23]), the focus is on how to extend an available decision procedure to return interpolants. Our methodology is similar to the second approach, since we add the capability of computing interpolants to the satisfiability procedure in Section 4. However, we do this by designing a flexible and abstract framework, relying on the identification of *basic operations* that can be performed independently from the method used by the underlying satisfiability procedure to derive a refutation.

5.1 Interpolating Metarules

Let now A, B be constraints in signatures Σ^A, Σ^B expanded with free constants and $\Sigma^C := \Sigma^A \cap \Sigma^B$; we shall refer to the definitions of AB-common, A-local, B-local, A-strict, B-strict, AB-mixed, AB-pure terms, literals and formulae given in Section 3. Our goal is to produce, in case $A \wedge B$ is $\mathcal{AX}_{\mathtt{diff}}$-unsatisfiable, a ground AB-common sentence ϕ such that $A \vdash_{\mathcal{AX}_{\mathtt{diff}}} \phi$ and $\phi \wedge B$ is $\mathcal{AX}_{\mathtt{diff}}$-unsatisfiable.

Let us examine some of the transformations to be applied to A, B. Suppose for instance that the literal ψ is AB-common and such that $A \vdash_{\mathcal{AX}_{\mathtt{diff}}} \psi$; then we can transform B into $B' := B \cup \{\psi\}$. Suppose now that we got an interpolant ϕ for the pair A, B': clearly, we can derive an interpolant for the original pair A, B by taking $\phi \wedge \psi$. The idea is to collect some useful transformations of this kind. Notice that these transformations can also modify the signatures Σ^A, Σ^B. For instance, suppose that t is an AB-common term and that c is a fresh constant: then we can put $A' := A \cup \{c = t\}$, $B' := B \cup \{c = t\}$: in fact, if ϕ is an interpolant for A', B', then $\phi(t/c)$ is an interpolant for A, B.[3] The transformations we need are called *metarules* and are listed in Table 1 below (in the Table and more generally in this Subsection, we use the notation $\phi \vdash \psi$ for $\phi \vdash_{\mathcal{AX}_{\mathtt{diff}}} \psi$).

[3] Notice that the fresh constant c is now a shared symbol, because Σ^A is enlarged to $\Sigma^A \cup \{c\}$, Σ^B is enlarged to $\Sigma^B \cup \{c\}$ and hence $(\Sigma^A \cup \{c\}) \cap (\Sigma^B \cup \{c\}) = \Sigma^C \cup \{c\}$.

An *interpolating metarules refutation* for A, B is a labeled tree having the following properties: (i) nodes are labeled by pairs of finite sets of constraints; (ii) the root is labeled by A, B; (iii) the leaves are labeled by a pair A, B such that $\bot \in A \cup B$; (iv) each non-leaf node is the conclusion of a rule from Table 1 and its successors are the premises of that rule. The crucial properties of the metarules are summarized in the following two Propositions.

▶ **Proposition 5.1.** *The unary metarules* $\frac{A \mid B}{A' \mid B'}$ *from Table 1 have the property that* $A \wedge B$ *is* $\exists$*-equivalent to* $A' \wedge B'$*; similarly, the n-ary metarules* $\frac{A_1 \mid B_1 \quad \cdots \quad A_n \mid B_n}{A \mid B}$ *from Table 1 have the property that* $A \wedge B$ *is* $\exists$*-equivalent to* $\bigvee_{k=1}^{n}(A_k \wedge B_k)$.

▶ **Proposition 5.2.** *If there exists an interpolating metarules refutation for* A, B *then there is a quantifier-free interpolant for* A, B *(namely there exists a quantifier-free* AB*-common sentence* ϕ *such that* $A \vdash \phi$ *and* $B \wedge \phi \vdash \bot$*). The interpolant* ϕ *is recursively computed applying the relevant interpolating instructions from Table 1.*

5.2 The Interpolation Solver

The metarules are complete, i.e. if $A \wedge B$ is $\mathcal{AX}_{\mathtt{diff}}$-unsatisfiable, then (since we shall prove that an interpolant exists) a single application of (Propagate1) and (Close2) gives an interpolating metarules refutation. This observation shows that metarules are by no means better than the brute force enumeration of formulae to find interpolants. However, metarules are useful to design an algorithm manipulating pairs of constraints based on transformation instructions. In fact, each of the transformation instructions can be *justified* by a metarule (or by a sequence of metarules): in this way, if our instructions form a complete and terminating algorithm, we can use Proposition 5.2 to get the desired interpolants. The main advantage of using metarules as justifications is that we just need to take care of the completeness and termination of the algorithm, and not about interpolants anymore. Here "completeness" means that our transformations should be able to bring a pair (A, B) of constraints into a pair (A', B') that either matches the requirements of Proposition 3.3 or is explicitly inconsistent, in the sense that $\bot \in A' \cup B'$. The latter is obviously the case whenever the original pair (A, B) is $\mathcal{AX}_{\mathtt{diff}}$-unsatisfiable and it is precisely the case leading to an interpolating metarules refutation.

The basic idea is that of invoking the algorithm of Section 4 on A and B separately and to propagate equalities involving AB-common terms. We shall assume *an ordering precedence making* AB*-common constants smaller than* A*-strict or* B*-strict constants of the same sort.* However, this is not sufficient to prevent the algorithm of Section 4 from generating literals and rules violating one or more of the hypotheses of Proposition 3.3: this is why the extra correcting instructions of group (γ) below are needed. Our interpolating algorithm has a pre-processing and a completion phase, like the algorithm from Section 4.

Pre-processing. In this phase the four Steps of Section 4.1 are performed both on A and on B; to justify these steps we need metarules (Define0,1,2), (Redplus1,2), (Redminus1,2), (Disjunction1,2), (ConstElim0,1,2), and (Propagate1,2) - the latter because if i, j are AB-common, the guessing of $i = j$ versus $i \neq j$ in Step 3 can be done, say, in the A-component and then propagated to the B-component. At the end of the preprocessing phase, the following properties (to be maintained as invariants afterwards) hold:

(i1) A (resp. B) contains $i \neq j$ for all A-local (resp. B-local) constants i, j of sort `INDEX` occurring in A (resp. in B);

(i2) if a, i occur in A (resp. in B), then $rd(a, i)$ reduces to an A-local (resp. B-local) constant of sort `ELEM`.

Close1	Close2	Propagate1	Propagate2
$\dfrac{}{A \mid B}$	$\dfrac{}{A \mid B}$	$\dfrac{A \mid B \cup \{\psi\}}{A \mid B}$	$\dfrac{A \cup \{\psi\} \mid B}{A \mid B}$
Prv.: A is unsat. *Int.*: $\phi' \equiv \bot$.	*Prv.*: B is unsat. *Int.*: $\phi' \equiv \top$.	*Prv.*: $A \vdash \psi$ and ψ is AB-common. *Int.*: $\phi' \equiv \phi \wedge \psi$.	*Prv.*: $B \vdash \psi$ and ψ is AB-common. *Int.*: $\phi' \equiv \psi \rightarrow \phi$.

Define0	Define1	Define2
$\dfrac{A \cup \{a = t\} \mid B \cup \{a = t\}}{A \mid B}$	$\dfrac{A \cup \{a = t\} \mid B}{A \mid B}$	$\dfrac{A \mid B \cup \{a = t\}}{A \mid B}$
Prv.: t is AB-common, a fresh. *Int.*: $\phi' \equiv \phi(t/a)$.	*Prv.*: t is A-local and a is fresh. *Int.*: $\phi' \equiv \phi$.	*Prv.*: t is B-local and a is fresh. *Int.*: $\phi' \equiv \phi$.

Disjunction1	Disjunction2
$\dfrac{\cdots \quad A \cup \{\psi_k\} \mid B \quad \cdots}{A \mid B}$	$\dfrac{\cdots \quad A \mid B \cup \{\psi_k\} \quad \cdots}{A \mid B}$
Prv.: $\bigvee_{k=1}^{n} \psi_k$ is A-local and $A \vdash \bigvee_{k=1}^{n} \psi_k$. *Int.*: $\phi' \equiv \bigvee_{k=1}^{n} \phi_k$.	*Prv.*: $\bigvee_{k=1}^{n} \psi_k$ is B-local and $B \vdash \bigvee_{k=1}^{n} \psi_k$. *Int.*: $\phi' \equiv \bigwedge_{k=1}^{n} \phi_k$.

Redplus1	Redplus2	Redminus1	Redminus2
$\dfrac{A \cup \{\psi\} \mid B}{A \mid B}$	$\dfrac{A \mid B \cup \{\psi\}}{A \mid B}$	$\dfrac{A \mid B}{A \cup \{\psi\} \mid B}$	$\dfrac{A \mid B}{A \mid B \cup \{\psi\}}$
Prv.: $A \vdash \psi$ and ψ is A-local. *Int.*: $\phi' \equiv \phi$.	*Prv.*: $B \vdash \psi$ and ψ is B-local. *Int.*: $\phi' \equiv \phi$.	*Prv.*: $A \vdash \psi$ and ψ is A-local. *Int.*: $\phi' \equiv \phi$.	*Prv.*: $B \vdash \psi$ and ψ is B-local. *Int.*: $\phi' \equiv \phi$.

ConstElim1	ConstElim2	ConstElim0
$\dfrac{A \mid B}{A \cup \{a = t\} \mid B}$	$\dfrac{A \mid B}{A \mid B \cup \{b = t\}}$	$\dfrac{A \mid B}{A \cup \{c = t\} \mid B \cup \{c = t\}}$
Prv.: a is A-strict and does not occur in A, t. *Int.*: $\phi' \equiv \phi$.	*Prv.*: b is B-strict and does not occur in B, t. *Int.*: $\phi' \equiv \phi$.	*Prv.*: c, t are AB-common, c does not occur in A, B, t. *Int.*: $\phi' \equiv \phi$.

Table 1 Interpolating Metarules: each rule has a proviso *Prv.* and an instruction *Int.* for recursively computing the new interpolant ϕ' from the old one(s) $\phi, \phi_1, \ldots, \phi_k$.

Completion. Some groups of instructions to be executed non-deterministically constitute the completion phase. There is however an important difference here with respect to the completion phase of Section 4.2: it may happen that we need some *guessing* also inside the completion phase (only the instructions from group (γ) below may need such guessings). Each instruction can be easily justified by suitable metarules (we omit the details for lack of space). The groups of instructions are the following:

(α) Apply to A or to B any instruction from the completion phase of Section 4.2.

(β) If there is an AB-common literal that belongs to A but not to B (or vice versa), copy it in B (resp. in A).

(γ) Replace *undesired literals*, i.e., those violating conditions (I)-(II)-(III) from Proposition 3.3.

To avoid trivial infinite loops with the (β) instructions, rules in (α) deleting an AB-common literal should be performed *simultaneously* in the A- and in the B-components (it can be easily checked [5] that this is always possible, for instance if rules in (β) and (γ) are given higher priority). Instructions (γ) need to be described in more details. Preliminarily, we introduce a technique that we call *Term Sharing*. Suppose that the A-component contains a literal $\alpha = t$, where the term t is AB-common but the free constant α is only A-local. Then it is possible to "make α AB-common" in the following way. First, introduce a fresh AB-common constant α' with the explicit definition $\alpha' = t$ (to be inserted both in A and in B, as justified by metarule (Define0)); then replace the literal $\alpha = t$ by $\alpha = \alpha'$ and replace α by α' everywhere else in A; finally, delete $\alpha = \alpha'$ too. The result is a pair (A, B) where basically nothing has changed but α has been renamed to an AB-common constant α'. Notice that the above transformations can be justified by metarules (Define0), (Redplus1), (Redminus1), (ConstElim1). We are now ready to explain instructions (γ) in details. First, consider undesired literals corresponding to the rewrite rules of the form

$$rd(c, i) \to d \tag{12}$$

in which the left-hand side is AB-common and the right-hand side is, say, A-strict. If we apply Term Sharing, we can solve the problem by renaming d to an AB-common fresh constant d'. We can apply a similar procedure to the rewrite rules

$$a \to wr(c, I, E) \tag{13}$$

in case the right-hand side is AB-common and the left-hand side is not; when we rename a to some fresh AB-common constant c', we must arrange the precedence so that $c' > c$ to orient the renamed literal as $c' \to wr(c, I, E)$. Then, consider the literals of the form

$$\texttt{diff}(a, b) = k \tag{14}$$

in which the left-hand side is AB-common and the right-hand side is, say, A-strict. Again, we can rename k to some AB-common constant k' by Term Sharing. Notice that k' is AB-common, whereas k was only A-local: this implies that we might need to perform some guessing to maintain the invariant (i1). Basically, we need to repeat Step 3 from Section 4.1 till invariant (i1) is restored (k' must be compared for equality with the other B-local constants of sort `INDEX`). The last undesired literals to take care of are the rules of the form[4]

$$c \to wr(c', I, E) \tag{15}$$

having an AB-common left-hand side but, say, only an A-local right-hand side. Notice that from the fact that c is AB-common, it follows (by our choice of the precedence) that c' is AB-common too. We can freely suppose that I and E are split into sublists I_1, I_2 and E_1, E_2, respectively, such that $I \equiv I_1 \cdot I_2$ and $E \equiv E_1 \cdot E_2$, where I_1, E_1 are AB-common, $I_2 \equiv i_1, \ldots, i_n$, $E_2 \equiv e_1, \ldots, e_n$ and for each $k = 1, \ldots, n$ at least one from i_k, e_k is not AB-common. This n (measuring essentially the number of non AB-common symbols in (15)) is called the *degree* of the undesired literal (15): in the following, we shall see how to eliminate (15) or to replace it with a smaller degree literal. We first make a guess (see metarule (Disjunction1)) about the truth value of the literal $c = wr(c', I_1, E_1)$. In the first

[4]Literals $d = e$ are automatically oriented in the right way by our choice of the precedence.

case, we add the positive literal to the current constraint; as a consequence, we get that the literal (15) is equivalent to $c = wr(c, I_2, E_2)$ and also to $rd(c, I_2) = E_2$ (see **Red** in Figure 1). In conclusion, in this case, the literal (15) is replaced by the AB-common rewrite rule $c \to wr(c', I_1, E_1)$ and by the literals $rd(c, I_2) = E_2$. In the second case, we guess that the negative literal $c \neq wr(c', I_1, E_1)$ holds; we introduce a fresh AB-common constant c'' together with the defining AB-common literal[5]

$$c'' \to wr(c', I_1, E_1) \tag{16}$$

(see metarule (Define0)). The literal (15) is replaced by the literal

$$c \to wr(c'', I_2, E_2). \tag{17}$$

We show how to make the degree of (17) smaller than n. In addition, we eliminate the negative literal $c \neq c''$ coming from our guessing (notice that, according to (16), c'' renames $wr(c', I_1, E_1)$). This is done as follows: we introduce fresh AB-common constants i, d, d'' together with the AB-common defining literals

$$\mathtt{diff}(c, c'') = i, \quad rd(c, i) \to d, \quad rd(c'', i) \to d'' \tag{18}$$

(see metarule (Define0)). Now it is possible to replace $c \neq c''$ by the literal $d \neq d''$ (see axiom (3)). Under the assumption $Distinct(I_2)$, the following statement is $\mathcal{AX}_{\mathtt{diff}}$ valid:

$$c = wr(c'', I_2, E_2) \wedge rd(c'', i) = d'' \wedge rd(c, i) = d \wedge d \neq d'' \to \bigvee_{k=1}^{n} (i = i_k \wedge d = e_k).$$

Thus, we get n alternatives (see metarule (Disjunction1)). In the k-th alternative, we can remove the constants i_k, e_k from the constraint, by replacing them with the AB-common terms i, d respectively (see metarules (Redplus1), (Redplus2), (Redminus1), (Redminus2),(ConstElim1),(ConstElim0)); notice that it might be necessary to complete the index partition. In this way, the degree of (17) is now smaller than n.

In conclusion, if we apply exhaustively Pre-Processing and Completion instructions above, starting from an initial pair of constraints (A, B), we can produce a tree, whose nodes are labelled by pairs of constraints (the successor nodes are labelled by pairs of constraints that are obtained by applying an instruction). We call such a tree an *interpolating tree* for (A, B). The following result shows that we obtained an interpolation algorithm for $\mathcal{AX}_{\mathtt{diff}}$:

▶ **Theorem 5.3.** *Any interpolation tree for (A, B) is finite; moreover, it is an interpolating metarules refutation (from which an interpolant can be recursively computed according to Proposition 5.2) precisely iff $A \wedge B$ is $\mathcal{AX}_{\mathtt{diff}}$-unsatisfiable.*

From the above Theorem it immediately follows that:

▶ **Theorem 5.4.** *The theory $\mathcal{AX}_{\mathtt{diff}}$ admits quantifier-free interpolants (i.e., for every quantifier free formulae ϕ, ψ such that $\psi \wedge \phi$ is $\mathcal{AX}_{\mathtt{diff}}$-unsatisfiable, there exists a quantifier free formula θ such that: (i) $\psi \vdash_{\mathcal{AX}_{\mathtt{diff}}} \theta$; (ii) $\theta \wedge \phi$ is not $\mathcal{AX}_{\mathtt{diff}}$-satisfiable: (iii) only variables occurring both in ψ and in ϕ occur in θ).*

In [5], we also give a direct (although non-constructive) proof of this theorem by using the model-theoretic notion of amalgamation.

[5] We put $c > c'' > c'$ in the precedence. Notice that invariant (i2) is maintained, because all terms $rd(c'', h)$ normalize to an element constant. In case I_1 is empty, one can directly take c' as c''.

5.3 An Example

To illustrate our method, we describe the computation of an interpolant for the mutually unsatisfiable sets $A \equiv \{a = wr(b,i,d)\}, B \equiv \{rd(a,j) \neq rd(b,j), rd(a,k) \neq rd(b,k), j \neq k\}$. Notice that i, d are A-strict constants, j, k are B-strict constants, and a, b are AB-common constants with precedence $a > b$. We first apply Pre-Processing instructions to obtain $A \equiv \{a = wr(b,i,d), rd(a,i) = e_5, rd(b,i) = e_6\}, B \equiv \{rd(a,j) = e_1, rd(b,j) = e_2, rd(a,k) = e_3, rd(b,k) = e_4, e_1 \neq e_2, e_3 \neq e_4, j \neq k\}$. Since $a = wr(b,i,d)$ is an undesired literal of the kind (15), we generate the two subproblems $\Pi_1 \equiv (A \cup \{rd(b,i) = d, a = b\}, B)$ and $\Pi_2 \equiv (A \cup \{a \neq b\}, B)$.[6]

Let us consider Π_1 first. Notice that $A \vdash a = b$, and $a = b$ is AB-common. Therefore we send $a = b$ to B, and we may derive the new equality $e_1 = e_2$ from the critical pair (C3) $e_1 \leftarrow rd(a,j) \rightarrow rd(b,j) \rightarrow e_2$, thus obtaining $A \equiv \{a = b, rd(b,i) = d, rd(a,i) = e_5, rd(b,i) = e_6\}$, $B \equiv \{rd(b,j) = e_2, rd(a,k) = e_3, rd(b,k) = e_4, e_1 \neq e_2, e_3 \neq e_4, j \neq k, a = b, e_1 = e_2\}$. Now B is inconsistent. The interpolant for Π_1 can be computed with the *interpolating instructions* of the metarules (Close1,Propagate1,Redminus1,Redplus1) resulting in $\varphi_1 \equiv (\top \wedge a = b) \equiv a = b$.

Then, let us consider branch Π_2. Recall that this branch originates from the attempt of removing the undesired rule $a \rightarrow wr(b,i,d)$. We introduce the AB-common defining literals $\mathtt{diff}(a,b) = l, rd(a,l) = f_1, rd(b,l) = f_2$, and $f_1 \neq f_2$, in order to remove $a \neq b$ from A. These are immediately propagated to B: $A \equiv \{a = wr(b,i,d), rd(a,i) = e_5, rd(b,i) = e_6, \mathtt{diff}(a,b) = l, rd(a,l) = f_1, rd(b,l) = f_2, f_1 \neq f_2\}, B \equiv \{rd(a,j) = e_1, rd(b,j) = e_2, rd(a,k) = e_3, rd(b,k) = e_4, e_1 \neq e_2, e_3 \neq e_4, j \neq k, \mathtt{diff}(a,b) = l, rd(a,l) = f_1, rd(b,l) = f_2, f_1 \neq f_2\}$. Since $a = wr(b,i,d)$ contains only the index i, we do not have a real case split. Therefore we replace i with l, and d with f_1. At last, we propagate the AB-common literal $a = wr(b,l,f_1)$ to B. After all these steps we obtain: $A \equiv \{a = wr(b,l,f_1), rd(a,l) = e_5, rd(b,i) = e_6, \mathtt{diff}(a,b) = l, rd(a,l) = f_1, rd(b,l) = f_2, f_1 \neq f_2\}$, $B \equiv \{rd(a,j) = e_1, rd(b,j) = e_2, rd(a,k) = e_3, rd(b,k) = e_4, e_1 \neq e_2, e_3 \neq e_4, j \neq k, \mathtt{diff}(a,b) = l, rd(a,l) = f_1, rd(b,l) = f_2, f_1 \neq f_2, a = wr(b,l,f_1)\}$. Since we have one more AB-common index constant l, we complete the current index constant partition, namely $\{k\}$ and $\{j\}$: we have three alternatives, to let l stay alone in a new class, or to add l to one of the two existing classes. In the first alternative, because of the following critical pair (C3) $e_1 \leftarrow rd(a,j) \rightarrow rd(wr(b,l,f_1),j) \rightarrow e_2$, we add $e_1 = e_2$ to B, which becomes trivially unsatisfiable. The other two alternatives yield similar outcomes. For each subproblem the interpolant, reconstructed by reverse application of the interpolating instructions of (Define0) and (Propagate1), is $\varphi_2' \equiv \{(a = wr(b, \mathtt{diff}(a,b), rd(a, \mathtt{diff}(a,b))) \wedge rd(a, \mathtt{diff}(a,b)) \neq rd(b, \mathtt{diff}(a,b)))\}$. The interpolant φ_2 for the branch Π_2 has to be computed by combining with (Disjunction2) three copies of φ_2', and so $\varphi_2 \equiv \varphi_2'$.

The final interpolant is computed by combining the interpolants for Π_1 and Π_2 by means of (Disjunction1), yielding $\varphi \equiv \varphi_1 \vee \varphi_2 \equiv (a = b \vee (a = wr(b, \mathtt{diff}(a,b), rd(a, \mathtt{diff}(a,b))) \wedge rd(a, \mathtt{diff}(a,b)) \neq rd(b, \mathtt{diff}(a,b))))$, i.e. $a = wr(b, \mathtt{diff}(a,b), rd(a, \mathtt{diff}(a,b)))$.

6 Related work and Conclusions

There is a series of papers devoted to building satisfiability procedures for the theory of arrays with or without extensionality. The interested reader is pointed to, e.g., [10, 12] for

[6] Notice that this is precisely the case in which there is no need of an extra AB-common constant c''.

an overview. In the following, for lack of space, we discuss the papers more closely related to interpolation for the theory of arrays.

After McMillan's seminal work on interpolation for model checking [18,20], several papers appeared whose aim was to design techniques for the efficient computation of interpolants in first-order theories of interest for verification, mainly uninterpreted function symbols, fragments of Linear Arithmetic, or their combination. An interpolating theorem prover is described in [19], where a sequent-like calculus is used to derive interpolants from proofs in propositional logic, equality with uninterpreted functions, linear rational arithmetic, and their combinations. In [15], a method to compute interpolants in data structures theories, such as sets and arrays (with extensionality), by axiom instantiation and interpolant computation in the theory of uninterpreted functions is described. It is also shown that the theory of arrays with extensionality does not admit quantifier-free interpolation. The "split" prover in [13] applies a sequent calculus for the synthesis of interpolants along the lines of that in [19] and is tuned for predicate abstraction [22]. The "split" prover can handle a combination of theories among which also the theory of arrays without extensionality is considered. In [13], it is pointed out that the theory of arrays poses serious problems in deriving quantifier-free interpolants because it entails an infinite set of quantifier-free formulae, which is indeed problematic when interpolants are to be used for predicate abstraction. To overcome the problem, [13] suggests to constrain array valued terms to occur in equalities of the form $a = wr(a, I, E)$ in the notation of this paper. It is observed that this corresponds to the way in which arrays are used in imperative programs. Further limitations are imposed on the symbols in the equalities in order to obtain a complete predicate abstraction procedure. In [14], the method described in [13] is specialized to apply CEGAR techniques [8] for the verification of properties of programs manipulating arrays. The method of [13] is extended to cope with range predicates which allow one to describe unbounded array segments which permit to formalize typical programming idioms of arrays, yielding property-sensitive abstractions. In [16], a method to derive quantified invariants for programs manipulating arrays and integer variables is described. A resolution-based prover is used to handle an *ad hoc* axiomatization of arrays by using predicates. Neither McCarthy's theory of arrays nor one of its extensions are considered in [16]. The invariant synthesis method is based on the computation of interpolants derived from the proofs of the resolution-based prover and constraint solving techniques to handle the arithmetic part of the problem. The resulting interpolants may contain even alternation of quantifiers.

To the best of our knowledge, the interpolation procedure presented in this paper is the first to compute quantifier-free interpolants for a natural variant of the theory of arrays with extensionality. In fact, the variant is obtained by replacing the extensionality axiom with its Skolemization which should be sufficient when the procedure is used to detect unsatisfiability of formulae as it is the case in standard model checking methods for infinite state systems. Because our method is not based on a proof calculus, we can avoid the burden of generating a large proof before being able to extract interpolants. The implementation of our procedure is currently being developed in the SMT-solver OpenSMT [6] and preliminary experiments are encouraging. An extensive experimental evaluation is planned for the immediate future.
Acknowledgements. We wish to thank an anonymuous referee for many useful criticisms that helped improving the quality of the paper.

References

1 F. Baader, S. Ghilardi, and C. Tinelli. A new combination procedure for the word problem that generalizes fusion decidability results in modal logics. *Inform. and Comput.*,

204(10):1413–1452, 2006.

2 F. Baader and T. Nipkow. *Term rewriting and all that*. Cambridge University Press, Cambridge, 1998.

3 A. Brillout, D. Kroening, P. Rümmer, and W. Thomas. An Interpolating Sequent Calculus for Quantifier-Free Presburger Arithmetic . In *IJCAR*, 2010.

4 R. Bruttomesso. *Problemi di combinazione nella dimostrazione automatica e nella verifica del software*. Università degli Studi di Milano, 2004. Master Thesis.

5 R. Bruttomesso, S. Ghilardi, and S. Ranise. Rewriting-based Quantifier-free Interpolation for a Theory of Arrays. Technical Report RI 334-10, Dip. Scienze dell'Informazione, Univ. di Milano, 2010.

6 R. Bruttomesso, E. Pek, N. Sharygina, and A. Tsitovich. The OpenSMT Solver. In *TACAS*, pages 150–153, 2010.

7 A. Cimatti, A. Griggio, and R. Sebastiani. Efficient Interpolation Generation in Satisfiability Modulo Theories. *ACM Trans. Comput. Logic*, 12:1–54, 2010.

8 E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In *CAV*, pages 154–169, 2000.

9 W. Craig. Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. *J. Symb. Log.*, pages 269–285, 1957.

10 L. de Moura and N. Bjørner. Generalized, Efficient Array Decision Procedures. In *FMCAD*, pages 45–52, 2009.

11 A. Fuchs, A. Goel, J. Grundy, S. Krstić, and C. Tinelli. Ground Interpolation for the Theory of Equality. In *TACAS*, pages 413–427, 2009.

12 S. Ghilardi, E. Nicolini, S. Ranise, and D. Zucchelli. Decision procedures for extensions of the theory of arrays. *Annals of Mathematics and Artificial Intelligence*, 50:231–254, 2007.

13 R. Jhala and K. L. McMillan. A Practical and Complete Approach to Predicate Refinement. In *TACAS*, pages 459–473, 2006.

14 R. Jhala and K. L. McMillan. Array Abstractions from Proofs. In *CAV*, pages 193–206, 2007.

15 D. Kapur, R. Majumdar, and C. Zarba. Interpolation for Data Structures. In *SIGSOFT'06/FSE-14*, pages 105–116, 2006.

16 L. Kovács and A. Voronkov. Finding Loop Invariants for Programs over Arrays Using a Theorem Prover. In *FASE*, pages 470–485, 2009.

17 J. McCarthy. Towards a Mathematical Science of Computation. In *IFIP Congress*, pages 21–28, 1962.

18 K. L. McMillan. Interpolation and SAT-Based Model Checking. In *CAV*, pages 1–13, 2003.

19 K. L. McMillan. An Interpolating Theorem Prover. *Theor. Comput. Sci.*, 345(1):101–121, 2005.

20 K. L. McMillan. Applications of Craig Interpolation to Model Checking. In *TACAS*, pages 1–12, 2005.

21 S. Ranise, C. Ringeissen, and D. Tran. Nelson-Oppen, Shostak and the Extended Canonizer: A Family Picture with a Newborn. In *ICTAC*, pages 372–386, 2004.

22 H. Saidi and S. Graf. Construction of abstract state graphs with PVS. In *CAV*, pages 72–83, 1997.

23 G. Yorsh and M. Musuvathi. A Combination Method for Generating Interpolants. In *CADE*, pages 353–368, 2005.

Improved Functional Flow and Reachability Analyses Using Indexed Linear Tree Grammars

Jonathan Kochems and Luke Ong

Oxford University Computing Laboratory

Abstract

The collecting semantics of a program defines the strongest static property of interest. We study the analysis of the collecting semantics of higher-order functional programs, cast as left-linear term rewriting systems. The analysis generalises functional flow analysis and the reachability problem for term rewriting systems, which are both undecidable. We present an algorithm that uses *indexed linear tree grammars* (ILTGs) both to describe the input set and compute the set that approximates the collecting semantics. ILTGs are equi-expressive with pushdown tree automata, and so, strictly more expressive than regular tree grammars. Our result can be seen as a refinement of Jones and Andersen's procedure, which uses regular tree grammars. The main technical innovation of our algorithm is the use of indices to capture (sets of) substitutions, thus enabling a more precise binding analysis than afforded by regular grammars. We give a simple proof of termination and soundness, and demonstrate that our method is more accurate than other approaches to functional flow and reachability analyses in the literature.

Keywords and phrases Flow analysis, reachability, collecting semantics, higher-order program, term rewriting, indexed linear tree grammar

Digital Object Identifier 10.4230/LIPIcs.RTA.2011.187

Category Regular Research Paper

1 Introduction

In program analysis, the *collecting semantics* of a program maps a given program point to the collection of all states attainable by a run of the program when control reaches that point. Thus the collecting semantics defines the strongest static property of interest [3]. A good method of analysing the collecting semantics of programs is the basis of a useful *generic* tool; it can be employed, *a fortiori*, to analyse such practically important computational properties as reachability and control flow.

A higher-order functional program with pattern-matching algebraic data types may be viewed[1] as a (left-linear) term rewriting system, namely, a set $\mathcal{P} = \{ l_i \to r_i \mid 1 \le i \le p \}$ of rewrite rules where each l_i and r_i are elements of the term algebra of a given signature, generated from a set of variables. The one-step rewrite relation, $\to_{\mathcal{P}}$, is standard: if a ground term t has a subterm u that matches the pattern l_i (i.e. $t = C[u]$ for some context $C[\text{-}]$, and $\sigma\, l_i = u$ for some ground substitution σ) then $t = C[\sigma\, l_i]$ rewrites in one step to $C[\sigma\, r_i]$, written $C[\sigma\, l_i] \to_{\mathcal{P}} C[\sigma\, r_i]$. What then is the collecting semantics of $\mathcal{P}$? Following Jones and Andersen [10], we take the program points of $\mathcal{P}$ to be the rewrite rules, and the

[1] The idea goes back to Reynolds' *defunctionalization* [20]. A higher-order lambda-term can be systematically "lambda-lifted" to an equivalent term rewriting system that has explicit binary application operators, systematically replacing closed higher-order lambda-terms by named combinators; thus every function is treated as "curried". See, for example, Jones and Andersen's account of the translation [10].

22nd International Conference on Rewriting Techniques and Applications.
Editor: M. Schmidt-Schauß; pp. 187–202
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

states of $\mathcal{P}$ to be the ground substitutions. The *collecting semantics* of $\mathcal{P}$ over a set I of input terms is then the tuple $(Z_0, Z_1, \ldots, Z_p)$, where

$$Z_0 \coloneqq \{\, (Id, t) \mid \exists s \in I \,.\, s \to_{\mathcal{P}}^* t \,\}$$
$$Z_i \coloneqq \{\, (\sigma, t) \mid \exists s \in I \,.\, s \to_{\mathcal{P}}^* C[\sigma\, l_i] \;\wedge\; \sigma\, r_i \to_{\mathcal{P}}^* t \,\} \quad \text{for } 1 \le i \le p$$

In words, Z_i consists of all pairs (σ, t) where σ is a substitution that matches the LHS of rule i in a $\mathcal{P}$-computation which is reachable from I (we call σ an *I-reachable substitution*), and t is a *result term* which is reachable from the RHS of rule i when instantiated by σ. (In a functional computation, control flow is determined by a sequence of function calls, possibly unknown at compile time. Thus, the flow analysis of $\mathcal{P}$ amounts to approximating the values that may be substituted for the program variables of each rewrite rule during a run of $\mathcal{P}$. It follows that one can use collecting semantics to flow-analyse functional programs.) Since a precise characterisation of the collecting semantics is uncomputable, our goal is to build over-approximations of the Z_is.

The key to our construction is a binding analysis that uses indices to explicitly model (sets of) I-reachable substitutions in the setting of *indexed linear tree grammars* (ILTGs). The rules of an ILTG rewrite term-trees in which leaf-nodes may be labelled by a non-terminal annotated with a sequence of indices; indices propagate from the root to the leaves of a term-tree. ILTGs are strictly more expressive than regular tree grammars; in fact, they are equi-expressive with pushdown tree automata. We give an algorithm which takes three input arguments (namely, a program $\mathcal{P}$ as before, a set of input terms defined by an ILTG $\mathcal{G}_0$, and an accuracy parameter $n \ge 0$) and constructs an ILTG $\mathcal{G}^n$ that approximates the collecting semantics of $\mathcal{P}$ on input given by $\mathcal{G}_0$. Precisely, the ILTG $\mathcal{G}^n$, which is equipped with distinguished non-terminals

- R_i (denoting the "results of the $\mathcal{P}$-rule, $l_i \to r_i$"), for each $1 \le i \le p$, and
- X, for each program *variable* X,

over-approximates Z_i (for each $0 \le i \le p$) in the following sense:

> *local safety*: for every $(\sigma, t) \in Z_i$, there is (constructively) an index sequence δ_σ such that $R_i\, \delta_\sigma \to_{\mathcal{G}^n}^* t$ and $X\, \delta_\sigma \to_{\mathcal{G}^n}^* \sigma X$ for each variable X that occurs in r_i

where σX denotes the term obtained by applying the substitution σ to the *variable* X, and $\to_{\mathcal{G}^n}$ is the one-step rewrite relation of $\mathcal{G}^n$. The superscript n of $\mathcal{G}^n$ controls the accuracy of approximation: if $n < n'$ then the ILTG $\mathcal{G}^{n'}$ offers at least as accurate an approximation as $\mathcal{G}^n$. Happily, the ILTG $\mathcal{G}^n$ is not prohibitively large: its size is polynomial in the number of rules in $\mathcal{P}$ and $\mathcal{G}_0$, and exponential in n and the number[2] of program variables in $\mathcal{P}$.

As far as we know, our algorithm gives the first completion procedure for indexed linear tree grammars (equivalently, pushdown tree automata). It extends Jones and Andersen's safe approximation of collecting semantics by admitting a strictly larger class of input sets. Even when restricted to regular input sets, for each $n \ge 0$, our algorithm builds an ILTG $\mathcal{G}^n$ which is at least as accurate as the result of Jones and Andersen's method.

Since the 2-projection of Z_0 is the set $\mathit{Reach}_{\mathcal{P}}(I)$ of terms that are reachable from the input set I under rewriting by $\mathcal{P}$, our analysis of collecting semantics may also be viewed as a contribution to the reachability problem for left-linear term rewriting systems (TRS) (i.e. given an input set I and a left-linear TRS $\mathcal{R}$, construct a set that over-approximates $\mathit{Reach}_{\mathcal{R}}(I)$) [9, 5, 2]. To the best of our knowledge, none of the over-approximation results in the literature can admit an arbitrary pushdown tree language as the input set.

---

[2] This can be improved to $\max_{1 \le i \le p} \#\mathit{vars}(l_i)$ where $\#\mathit{vars}(l_i)$ is the number of variables in l_i.

Outline In Section 2, after fixing notations we introduce indexed linear tree grammars and the notion of minimally reachable match. The ILTG completion algorithm for over-approximating the collecting semantics of a program on an input set is presented in Section 3. The termination proof and soundness proof are given in Sections 4 and 5 respectively. In the concluding section, we evaluate our result and set out a number of further directions. Note, a long version of the paper is available [13] containing the proofs that are omitted from the main text together with two complete worked examples which illustrate the workings of our algorithm.

The work reported here is based on preliminary results first presented in the first author's MSc dissertation [12].

2 Preliminaries

Term Rewriting Systems and Programs Let Σ be a ranked alphabet equipped with a function ar giving the arity of each symbol in Σ. We write $\Sigma_n := ar^{-1}(n)$ and use letters f, g, h, a and b to denote members of Σ. The *free algebra* over an arbitrary set $\mathcal{X}$, written $\mathcal{T}_\Sigma(\mathcal{X})$, is the smallest set such that $\mathcal{X} \subseteq \mathcal{T}_\Sigma(\mathcal{X})$, and if $f \in \Sigma_n$ and $t_1, \ldots, t_n \in \mathcal{T}_\Sigma(\mathcal{X})$ then $f(t_1, \ldots, t_n) \in \mathcal{T}_\Sigma(\mathcal{X})$. Elements of $\mathcal{T}_\Sigma(\mathcal{X})$ are called *terms*, denoted by letters s and t; and we write $\mathcal{T}_\Sigma := \mathcal{T}_\Sigma(\varnothing)$ for the set of *ground terms*.

Let $\mathcal{V} = \{ X, Y, Z, \cdots \}$ be a set of variables, and $\Sigma = \Delta \uplus \Gamma$ be a ranked alphabet that is partitioned into disjoint sets Δ and Γ. Symbols in Δ and Γ are called *defined operators* and *constructors* respectively. A *call* is a term of the form $f(t_1, \ldots, t_n)$ with $f \in \Delta$; a *pattern* is a call in which every variable occurs at most once. A *term rewriting system (TRS) over* Σ is a finite set of rewrite rules $\mathcal{P} = \{l_i \to r_i \mid l_i, r_i \in \mathcal{T}_\Sigma(\mathcal{V}); 1 \le i \le p\}$ such that for every i, l_i is a call, and every variable that occurs in r_i also occurs in l_i; further $\mathcal{P}$ is *left-linear* just if for each i, l_i is a pattern. The (one-step) *rewrite relation* $\to_\mathcal{P} \subseteq (\mathcal{T}_\Sigma(\mathcal{X}))^2$ of a TRS $\mathcal{P}$ is defined as

$$C[\sigma\, l] \to_\mathcal{P} C[\sigma\, r]$$

with $C[\text{-}]$ ranging over one-hole contexts, $\sigma : \mathcal{V} \to \mathcal{T}_\Sigma$ ranging over substitutions, and $l \to r$ ranging over rules in $\mathcal{P}$. The n-step rewrite relation, $\to^n_\mathcal{P}$, and the reflexive, transitive closure, $\to^*_\mathcal{P}$, of $\to_\mathcal{P}$ are defined in the usual way.

Henceforth we fix a ranked alphabet $\Sigma = \Delta \uplus \Gamma$. By a *program* we mean a pair $(\Sigma, \mathcal{P})$ where $\mathcal{P}$ is a left-linear TRS over Σ. (We can think of the defined operators and constructors respectively as the non-terminals and terminals of the program.) An *input* of $\mathcal{P}$ is a set $I \subseteq \mathcal{T}_\Sigma$ of ground terms. We are interested in analysing (*inter alia*) the set $\mathit{Reach}_\mathcal{P}(I) := \{t \mid s \in I, s \to^*_\mathcal{P} t\}$ of *$\mathcal{P}$-reachable terms from I*.

▸ **Example 2.1** (Running Example)**.** Consider the program $\mathcal{P}$ with $\Delta_1 = \{\mathsf{counter}, \mathsf{genh}, \mathsf{genk}\}$, $\Gamma_0 = \{\mathsf{0}, \mathsf{a}, \mathsf{b}\}$, $\Gamma_1 = \{\mathsf{S}, \mathsf{h}, \mathsf{k}\}$, $\Gamma_2 = \{\mathsf{f}\}$, and rules as follows:

1. $\mathsf{counter}(\mathsf{x}) \to \mathsf{counter}(\mathsf{S}(\mathsf{x}))$
2. $\mathsf{counter}(\mathsf{x}) \to \mathsf{f}(\mathsf{genh}(\mathsf{x}), \mathsf{genk}(\mathsf{x}))$
3. $\mathsf{genh}(\mathsf{0}) \to \mathsf{a}$
4. $\mathsf{genh}(\mathsf{S}(\mathsf{x})) \to \mathsf{h}(\mathsf{genh}(\mathsf{x}))$
5. $\mathsf{genk}(\mathsf{0}) \to \mathsf{b}$
6. $\mathsf{genk}(\mathsf{S}(\mathsf{x})) \to \mathsf{k}(\mathsf{genk}(\mathsf{x}))$

Let $I = \{\, \mathsf{counter}(\mathsf{0}) \,\}$. The set of reachable constructor-terms from I, namely, $\mathit{Reach}_\mathcal{P}(I) \cap \mathcal{T}_\Gamma$, is $\{\, \mathsf{f}(\mathsf{h}^n(\mathsf{a}), \mathsf{k}^n(\mathsf{b})) \mid n \ge 0 \,\}$.[3]

[3] There is a simpler program that gives the same set of reachable constructor terms, namely,

Indexed Linear Tree Grammar (ILTG) Jones and Andersen's algorithm constructs a regular tree grammar to over-approximate the collecting semantics of a given program $\mathcal{P}$ on input I. Our refinement works in a similar fashion but builds an indexed linear tree grammar instead. Let us introduce ILTG with an example.

▸ **Example 2.2.** Consider the grammar with non-terminal alphabet $\mathcal{N} = \{\mathsf{S},\mathsf{S'},\mathsf{A},\mathsf{B}\}$, terminal alphabet $\Sigma = \{\mathsf{f},\mathsf{h},\mathsf{k},\mathsf{a},\mathsf{b}\}$ and index set $\mathcal{F} = \{\alpha,\beta\}$. The rewrite rules are as follows.

1. $\mathsf{S} \to \mathsf{S'}\,\beta$
2. $\mathsf{S'} \to \mathsf{S'}\,\alpha$
3. $\mathsf{S'} \to \mathsf{f}(\mathsf{A},\mathsf{B})$
4. $\mathsf{A}\alpha \to \mathsf{h}(\mathsf{A})$
5. $\mathsf{A}\beta \to \mathsf{a}$
6. $\mathsf{B}\alpha\,\alpha \to \mathsf{k}(\mathsf{k}(\mathsf{B}))$
7. $\mathsf{B}\alpha\,\beta \to \mathsf{k}(\mathsf{b})$
8. $\mathsf{B}\beta \to \mathsf{b}$

This ILTG rewrites terms in $\mathcal{T}_\Sigma(\mathcal{N}\mathcal{F}^*)$. For example, the rule $\mathsf{B}\alpha\,\alpha \to \mathsf{k}(\mathsf{k}(\mathsf{B}))$ allows us to replace the term $\mathsf{B}\alpha\,\alpha\,\nu_1 \ldots \nu_n$ by the term $\mathsf{k}(\mathsf{k}(\mathsf{B}\nu_1 \ldots \nu_n))$ where $\nu_1 \ldots \nu_n \in \mathcal{F}^*$ —note the propagation of $\nu_1 \ldots \nu_n$ from the root of the term-tree $\mathsf{k}(\mathsf{k}(\mathsf{B}))$ to its leaf. Similarly Rule 4 allows the term $\mathsf{A}\alpha\,\beta$ to rewrite to $\mathsf{h}(\mathsf{A}\beta)$. A possible rewrite using the above rules is

$$\mathsf{S'}\,\alpha\,\beta \to \mathsf{S'}\,\alpha\,\alpha\,\beta \to \mathsf{f}(\mathsf{A}\alpha\,\alpha\,\beta\,,\mathsf{B}\alpha\,\alpha\,\beta\,) \to \mathsf{f}(\mathsf{A}\alpha\,\alpha\,\beta\,,\mathsf{k}(\mathsf{k}(\mathsf{B}\beta))).$$

In this ILTG, the set of reachable ground terms (in $\mathcal{T}_\Sigma$) from input $\{S\}$ of the ILTG is $\{\mathsf{f}(\mathsf{h}^n(\mathsf{a}),\mathsf{k}^n(\mathsf{b})) \mid n \geq 0\}$ which is the same as that of the TRS in Example 2.1. Note that the set is not regular; it follows that ILTGs are strictly more expressive than regular tree grammars. Note also the similarities between the computation of this ILTG and that of the TRS in Example 2.1.

We now give a formal definition of indexed linear tree grammars.

▸ **Definition 2.3** (ILTG)**.** An *indexed linear tree grammar (ILTG)* is a 5-tuple $(\Sigma,\mathcal{N},\mathcal{F},S,\mathcal{G})$ (of finite objects) where

- Σ is a ranked alphabet of terminal symbols (ranged over by f,g,a,b, etc.)
- $\mathcal{N}$ is an alphabet of nullary non-terminal symbols (ranged over by A,B,C, etc.), and $S \in \mathcal{N}$ is a distinguished *start* symbol
- $\mathcal{F}$ is a set of index symbols (ranged over by α,β, etc.)
- $\mathcal{G}$ a set of rewrite rules of the form $A\,\gamma \to_\mathcal{G} t$, where $A \in \mathcal{N}$, $\gamma \in \mathcal{F}^*$ and $t \in \mathcal{T}_\Sigma(\mathcal{N}\mathcal{F}^*)$.

By abuse of notation, we shall refer to the ILTG as $\mathcal{G}$. The (one-step) *rewrite relation* of $\mathcal{G}$, $\to_\mathcal{G} \subseteq (\mathcal{T}_\Sigma(\mathcal{N}\mathcal{F}^*))^2$, is defined as

$$C[A\,\gamma\,\delta] \to_\mathcal{G} C[\mathrm{dist}_\delta(t)]$$

with $C[\text{-}]$ ranging over (one-holed) contexts, δ over $\mathcal{F}^*$ and $A\,\gamma \to_\mathcal{G} t$ over rules in $\mathcal{G}$, and $\mathrm{dist}_\delta(t)$ is defined as $\mathrm{dist}_\delta(A\,\gamma) := A\,\gamma\,\delta$ and $\mathrm{dist}_\delta(f(t_1,\ldots,t_n)) := f(\mathrm{dist}_\delta(t_1),\ldots,\mathrm{dist}_\delta(t_n))$, where $n \geq 0$, and f, A and γ range over $\Sigma,\mathcal{N}$ and $\mathcal{F}^*$ respectively. For example,

$$\mathrm{dist}_{\alpha_1\alpha_2}(f(A\beta_1, g(B\beta_2\beta_3, a))) = f(A\beta_1\alpha_1\alpha_2, g(B\beta_2\beta_3\alpha_1\alpha_2, a)).$$

We denote the reflexive, transitive closure and the n-step rewrite relation of $\to_\mathcal{G}$ by $\to_\mathcal{G}^*$ and $\to_\mathcal{G}^n$ respectively.

$\mathsf{gen}(\mathsf{x},\mathsf{y}) \to \mathsf{gen}(\mathsf{h}(\mathsf{x}),\mathsf{k}(\mathsf{y})) \mid \mathsf{f}(\mathsf{x},\mathsf{y})$ with input $I = \{\mathsf{gen}(\mathsf{a},\ \mathsf{b})\}$.

Take an ILTG $(\Sigma, \mathcal{N}, \mathcal{F}, S, \mathcal{G})$, and let $I \subseteq \mathcal{T}_\Sigma(\mathcal{N}\mathcal{F}^*)$. We define $Reach_\mathcal{G}(I) := \{t \mid s \in I, s \to_\mathcal{G}^* t\}$; for singleton sets we omit set braces e.g. $Reach_\mathcal{G}(t)$ means $Reach_\mathcal{G}(\{t\})$; and if S is $\mathcal{G}$'s start symbol we write $Reach_\mathcal{G}$ for $Reach_\mathcal{G}(S)$. If $Reach_\mathcal{G}$ is well-defined we write $Reach_\mathcal{P}(\mathcal{G}) := Reach_\mathcal{P}(Reach_\mathcal{G})$ and set $Reach^o_\mathcal{G}(I) := Reach_\mathcal{G}(I) \cap \mathcal{T}_\Sigma$. Note that elements of $\mathcal{T}_\Sigma(\mathcal{N}\mathcal{F}^*)$ are term-trees: they can be viewed as (finite) ranked trees, whose internal nodes are labelled by symbols in Σ (of non-zero arities), and whose leaves are labelled by symbols in $\mathcal{N}\mathcal{F}^* \cup \Sigma_0$. Thus we define the *language of (finite) Σ-labelled trees generated by $\mathcal{G}$* to be $Reach^o_\mathcal{G}$.

▸ **Proposition 2.4.** *ILTGs are equi-expressive with pushdown tree automata [8] as generators of Σ-labelled tree languages. Thus ILTGs generate precisely the level-1 trees*[4] *of the hierarchy of (collapsible) pushdown trees [18].*

The idea is that the non-terminals and indices of an ILTG correspond respectively to the states and stack symbols of a pushdown tree automaton. See the full paper for a proof.

▸ Remark 2.5. (i) ILTGs are similar to Aho's indexed grammars [1] but there are important differences. First Aho's grammars are generators of word languages which are equi-expressive with second-order pushdown word automata; they define level 2 of the Maslov Hierarchy [16, 18]. Secondly, the linearity constraint on ILTG (each leaf of the RHS of a rule has at most one occurrence of a non-terminal) has the effect of reining in the power of indices, so that the branch language of a tree generated by an ILTG is context-free. (ii) Engelfriet and Vogler [4] introduced regular tree grammars with pushdown store which are equi-expressive with context-free tree languages [8], and with ILTGs. (iii) ILTGs define a class of trees with rich algorithmic properties, which make them highly suitable for verification (for example, their emptiness problem is in EXPTIME). In fact, all trees in the hierarchy of (collapsible) pushdown trees have decidable MSO theories [17].

ILTGs are a concise formalism; when rewriting, indices propagate from the root of a term-tree to its leaves, just like substitutions. We think that ILTGs are an attractive vehicle for the presentation of our algorithm.

Minimally Reachable Match Minimally reachable match is a concept due to Jones and Andersen [10]; it formalises the idea of rewriting a term by as many steps as necessary—but no more—in order to achieve a match against a pattern. Here we generalise it to the setting of indexed grammar, and consider the substitution that witnesses a minimally reachable match of a *uniformly indexed* term against a pattern. A term $t \in \mathcal{T}_\Sigma(\mathcal{N}\mathcal{F}^*)$ is said to be *uniformly indexed* (or simply *uniform*) just if $t = \mathrm{dist}_\delta(s)$ for some $\delta \in \mathcal{F}^*$ and $s \in \mathcal{T}_\Sigma(\mathcal{N})$ i.e. every non-terminal in t is annotated with the same index sequence.

▸ **Definition 2.6** (Minimally Reachable Match)**.** Let $\mathcal{G} = (\Sigma, \mathcal{N}, \mathcal{F}, S, \mathcal{G})$ be an ILTG, $s \in \mathcal{T}_\Sigma(\mathcal{V})$ be a pattern, and $\mathrm{dist}_{\alpha_1 \ldots \alpha_n} t$ be a uniform term (thus $t \in \mathcal{T}_\Sigma(\mathcal{N})$). We say that a substitution σ is a *minimally reachable match of* $\mathrm{dist}_{\alpha_1 \ldots \alpha_n}(t)$ *against* s just if there exists $m \geq 0$ such that

(i) $\mathrm{dist}_{\alpha_1 \ldots \alpha_n}(t) \to_\mathcal{G}^m \sigma s$, and
(ii) $\neg(\exists \sigma'. \mathrm{dist}_{\alpha_1 \ldots \alpha_n}(t) \to_\mathcal{G}^{m-1} \sigma' s \to \sigma s)$, and
(iii) $\neg(\exists \sigma'. \exists k < n. (\mathrm{dist}_{\alpha_1 \ldots \alpha_k}(t) \to_\mathcal{G}^m \sigma' s$ and $\sigma = \mathrm{dist}_{\alpha_{k+1} \ldots \alpha_n}(\sigma')))$.

[4] See, for example, the survey [18] for an introduction to the hierarchies of finite and infinite ranked trees. Note that ILTGs are generators of (languages of) both finite and infinite trees.

I.e. m is the minimal number of reduction steps from $\mathrm{dist}_{\alpha_1 \ldots \alpha_n} t$, and $\alpha_1 \ldots \alpha_n$ is a corresponding minimal index sequence, that are required to achieve a match against the pattern s. We shall sometimes refer to σ as a *minimally reachable substitution*.

Condition (iii) means that every index in $\alpha_1 \ldots \alpha_n$ must be "consumed" in the construction of σ. For example, take the rules in Example 2.2 and the pattern $s = \mathsf{f(g(X),h(h(Y)))}$ then $\mathsf{f(A}\alpha\,\alpha\mathsf{,B}\alpha\,\alpha\mathsf{)} \to^* \mathsf{f(g(A}\alpha\mathsf{),h(h(B)))} = \sigma s$ is a minimal derivation, where $\sigma = \{\mathsf{X} \mapsto \mathsf{A}\alpha, \mathsf{Y} \mapsto \mathsf{B}\}$. However the derivation $\mathsf{f(A}\alpha\,\alpha\,\beta\mathsf{,B}\alpha\,\alpha\,\beta\mathsf{)} \to^* \mathsf{f(g(A}\alpha\,\beta\mathsf{),h(h(B}\beta\mathsf{)))} = \mathrm{dist}_\beta(\sigma s)$ is not minimal, because β is superfluous. Note that in the absence of indices (i.e. in a regular tree grammar), n is necessarily 0, and so, the notion of minimally reachable match here coincides with that of Jones and Andersen's.

We say that an ILTG is *uniform* if the RHS of every rule is uniform. In a uniform ILTG, a minimally reachable match of a uniform term t against a pattern p has the nice property that the substitution in question will only replace a variable with a subterm of t (which is uniform) or an indexed subterm of the RHS of a rule (which is also uniform). In the following we write $r' \leq r$ to mean that r' is a subterm of r.

▸ **Proposition 2.7.** *Let $\mathcal{G}$ be an ILTG, $t \in \mathcal{T}_\Sigma(\mathcal{N})$ and $p \in \mathcal{T}_\Sigma(\mathcal{V})$. If σ is the substitution of a minimally reachable match of $\mathrm{dist}_\gamma(t)$ against p then for all X in $Vars(p)$, $\sigma X \leq \mathrm{dist}_\gamma(t)$, or $\sigma X = \mathrm{dist}_{\gamma'}(g)$ for some index sequence γ' and some subterm g of the right-hand-side (RHS) of a $\mathcal{G}$-rule.*

▸ Notational Convention 2.8. Henceforth we assume a program $\mathcal{P} = \{\, l_i \to r_i \mid 1 \leq i \leq p \,\}$ and a set I of input terms over a ranked alphabet $\Sigma = \Delta \uplus \Gamma$, where Δ consists of defined-operator symbols and Γ consists of constructor symbols. We further assume that the input $I := \mathit{Reach}^o_{\mathcal{G}_0}$ where $\mathcal{G}_0$ is a (uniform) ILTG with start symbol R_0. We aim to over-approximate the collecting semantics of $\mathcal{P}$ on I by means of ILTGs, ranged over by $\mathcal{G}$, that are defined over a terminal alphabet which is set to be Σ, and a non-terminal alphabet $\mathcal{N}$ that satisfies

$$\mathcal{N} \supseteq \{\, X \mid X \in \mathit{Vars}(l_i), 1 \leq i \leq p \,\} \cup \{\, R_0, R_1, \ldots, R_p \,\}$$

where R_0 is the start symbol. Note that every symbol of $\mathcal{P}$ (whether user-defined or constructor) is a terminal symbol of $\mathcal{G}$; and every program variable of $\mathcal{P}$ is a non-terminal of $\mathcal{G}$. For each $1 \leq i \leq p$, the non-terminal R_i (read "the results of the rule $l_i \to r_i$") is intended to generate a superset of all the terms that are reachable from l_i in a rewriting sequence that originates from a term in I.

We make precise what it means for an ILTG to be a safe over-approximation of the collecting semantics of $\mathcal{P}$ on I, and distinguish two versions of safety.

▸ **Definition 2.9** (Safety)**.** Let $(\Sigma, \mathcal{N}, \mathcal{F}, S, \mathcal{G})$ be an ILTG.

(i) $\mathcal{G}$ is *globally safe* for $\mathcal{P}$ on I just if there are terms (i.e. elements of $\mathcal{T}_\Sigma(\mathcal{N}\mathcal{F}^*)$)

- $\widetilde{R_0}, \widetilde{R_1}, \ldots, \widetilde{R_p}$, and
- $\widetilde{X}$, for each $X \in \mathit{Vars}(r_i)$, each $1 \leq i \leq p$

such that for every i and $(\sigma, g) \in Z_i$, $\widetilde{R_i} \to^*_{\mathcal{G}} g$; and $\widetilde{X} \to^*_{\mathcal{G}} \sigma X$ for each $X \in \mathit{Vars}(r_i)$.

(ii) $\mathcal{G}$ is *locally safe* for $\mathcal{P}$ on I just if for every i and $(\sigma, g) \in Z_i$, there is an index sequence δ_σ such that $R_i\,\delta_\sigma \to^*_{\mathcal{G}} g$; and $X\,\delta_\sigma \to^*_{\mathcal{G}} \sigma X$ for each $X \in \mathit{Vars}(r_i)$.

Note that regular tree grammars are ILTGs with an empty index set. Both versions of safety subsume Jones and Andersen's (which assumes regularity of both the input and the approximating grammar).

3 The Grammar Completion Algorithm

Suppose we want to investigate the effect the program $\mathcal{P}$ in Example 2.1 has on the set $Reach_{\mathcal{G}}$ where the ILTG $\mathcal{G}$ is defined in Example 3.1.

▸ **Example 3.1.** $\mathcal{N} = \{\mathsf{R}_0,\mathsf{R}_1,\mathsf{R}_1^{\sigma_1},\mathsf{R}_1^{\sigma_2},\mathsf{R}_2,\mathsf{R}_2^{\sigma_1},\mathsf{R}_2^{\sigma_2}\}$, $\mathcal{F} = \{\sigma_1,\sigma_2\}$, $\Sigma_0 = \{0\}$, $\Sigma_1 = \{\mathsf{counter},\mathsf{genh},\mathsf{genk},\mathsf{S},\mathsf{h},\mathsf{k}\}$, $\Sigma_2 = \{\mathsf{f}\}$

$$\begin{array}{ll}
\mathsf{R}_0 \to \mathsf{counter}(0) \mid \mathsf{R}_1^{\sigma_1} \mid \mathsf{R}_2^{\sigma_1} & \mathsf{R}_1 \to \mathsf{counter}(\mathsf{S}(\mathsf{X})) \mid \mathsf{R}_1^{\sigma_2} \mid \mathsf{R}_2^{\sigma_2} \\
\mathsf{R}_1^{\sigma_1} \to \mathsf{R}_1\sigma_1 & \mathsf{R}_2 \to \mathsf{f}(\mathsf{genh}(\mathsf{X}),\mathsf{genk}(\mathsf{X})) \\
\mathsf{R}_1^{\sigma_2} \to \mathsf{R}_1\sigma_2 & \mathsf{X}\sigma_1 \to 0 \\
\mathsf{R}_2^{\sigma_1} \to \mathsf{R}_2\sigma_1 & \mathsf{X}\sigma_2 \to \mathsf{S}(\mathsf{X}) \\
\mathsf{R}_2^{\sigma_2} \to \mathsf{R}_2\sigma_2 &
\end{array}$$

If we rewrite the term $\mathsf{R}_2\sigma_2\sigma_1$ (which is easily seen to be in $Reach_{\mathcal{G}}$)

$$\mathsf{R}_2\sigma_2\sigma_1 \to_{\mathcal{G}}^* \mathsf{f}(\mathsf{genh}(\mathsf{S}(\mathsf{X}\sigma_1)),\mathsf{genk}(\mathsf{X}\sigma_2\sigma_1)) \to_{\mathcal{P}} \mathsf{f}(\mathsf{h}(\mathsf{genh}(\mathsf{X}\sigma_1)),\mathsf{genk}(\mathsf{X}\sigma_2\sigma_1)) =: t$$

then we observe that t is $\mathcal{P}$ reachable from a term in $Reach_{\mathcal{G}}$, but t is not in $Reach_{\mathcal{G}}$ which thus can be seen not to be invariant under $\mathcal{P}$. However, if we place the rules of Example 3.4 and $\mathcal{G}$ in a new ILTG $\mathcal{G}'$ then using the latter we can rewrite

$$\mathsf{R}_2\sigma_2\sigma_1 \to^* \mathsf{f}(\mathsf{R}_4\sigma_3\sigma_1),\mathsf{genk}(\mathsf{X}\sigma_2\sigma_1)) \to^* \mathsf{f}(\mathsf{h}(\mathsf{genh}(\mathsf{X}\sigma_1)),\mathsf{genk}(\mathsf{X}\sigma_2\sigma_1)).$$

Thus $\mathcal{G}'$ can be seen as a partial completion of $\mathcal{G}$ with respect to $\mathcal{P}$; the former is however still not complete w.r.t. $\mathcal{P}$. In fact, $\mathcal{G}$ is an intermediate result of our algorithm to build an over-approximation of $\mathcal{P}$ on the input $Reach_{\mathcal{G}_0}$ where $\mathcal{G}_0$ is the ILTG with a single rule $\mathsf{R}_0 \to \mathsf{counter}(0)$ and start symbol R_0 .

We aim to over-approximate the collecting semantics of $\mathcal{P}$ on I by means of ILTGs $\mathcal{G}$ that conform to Notational Convention 2.8, using an operator on ITLG, $\delta^n_{\mathcal{P}}(\text{-})$, which we will introduce shortly. First we define an auxiliary operation that takes a rule of an ILTG and returns a set of rules.

▸ **Definition 3.2** (Ext-Operator)**.** Let $\mathcal{G}$ be an ILTG, $n \geq 0$ and $A\gamma \to C[\mathrm{dist}_{\alpha_1\ldots\alpha_k}(g')] \in \mathcal{G}$ where $g' \in \mathcal{T}_\Sigma(\mathcal{N})$. The set $\mathrm{Ext}^n_{\mathcal{P},\mathcal{G}}(A\gamma \to C[\mathrm{dist}_{\alpha_1\ldots\alpha_k}(g')])$ contains (only) the following rules

$$\begin{cases}
A\gamma \to C[R_i^\sigma \alpha_1 \ldots \alpha_k] & \text{(I)} \\
R_i^\sigma \beta_1 \ldots \beta_{\min(m,n)} \to \begin{cases} R_i\,\sigma & \text{if } m \leq n \\ R_i\,\sigma\,\top & \text{otherwise} \end{cases} & \text{(II)} \\
R_i \to r_i & \text{(III)} \\
X\sigma \to \mathrm{aprx}_n(\sigma_0 X), \text{for each } X \in \mathit{Vars}(r_i) & \text{(IV)}
\end{cases}$$

whenever there exist an index sequence $\overline{\beta} = \beta_1 \ldots \beta_m$, a substitution σ_0, and an $1 \leq i \leq n$ such that

(1) $\overline{\beta}$ and $\overline{\alpha} = \alpha_1 \ldots \alpha_k$ are compatible sequences (i.e. one is a prefix of the other), and
(2) σ_0 is a minimally reachable match of $\mathrm{dist}_{\beta_1\ldots\beta_m}(g')$ against l_i, and
(3) σ is the index (*qua* substitution) defined by $\sigma X := \mathit{erase}(\sigma_0 X)$ for each variable X, where $\mathit{erase}(t)$ erases every index sequence, and every superscript of every non-terminal, that occur in $t \in \mathcal{T}_\Sigma(\mathcal{N}\mathcal{F}^*)$; for example

$$\mathit{erase}(g(f(R_1^\tau \alpha_1\alpha_2, h(a, X\beta_1\beta_2\beta_3)))) = g(f(R_1, h(a, X)))$$

$$\text{where } \mathrm{aprx}_i(A\,\alpha_1 \dots \alpha_k) := \begin{cases} A\,\top & \text{if } i = 0 \\ A\,\alpha_1 \dots \alpha_k & \text{if } i > 0 \wedge k \le i \\ A\,\alpha_1 \dots \alpha_i\,\top & \text{if } i > 0 \wedge k > i. \end{cases}$$

The function aprx_i is extended to terms and substitutions in the natural way.

▸ Remark 3.3. (i) In rule of type (IV) above, X on the LHS is a non-terminal; the expression $\mathrm{aprx}_n(\sigma_0 X)$ on the RHS denotes the term obtained by applying the *substitution* $\mathrm{aprx}_n(\sigma_0)$ to the *variable* X (of $\mathcal{P}$). (ii) Several minimally reachable matches σ_0 may give rise to the same substitution σ. The equation in condition (3) is intended to "merge" indices so that only a finite number of indices is eventually generated. (iii) In case $m > n$, the type-(II) rule is added: $R_i^\sigma \alpha_1 \dots \alpha_n \to R_i\,\sigma\,\top$. The operation $\mathrm{Ext}^n_{\mathcal{P},\mathcal{G}}(\text{-})$ does not produce a rule with $R_i\,\sigma$ or $R_i\,\sigma\,\top$ on the LHS. The point of the distinguished index $\top$—as indicated in the definition of $\top(\mathcal{G})$ in the following—is to introduce a measure of non-determinacy, conflating all possible instantiations of a variable X by substitutions introduced during the construction. (iv) Every rule of $\mathrm{Ext}^n_{\mathcal{P},\mathcal{G}}(A\,\gamma \to C[\mathrm{dist}_{\alpha_1 \dots \alpha_k}(g')])$ has index sequences of length at most $n+1$, assuming that $A\,\gamma \to C[\mathrm{dist}_{\alpha_1 \dots \alpha_k}(g')]$ itself has index sequences of length at most n.

▸ **Example 3.4.** The following are the rules in $\mathrm{Ext}^1_{\mathcal{G},\mathcal{P}}(\mathsf{R}_2{\to}\mathsf{f}([\mathsf{genh}(\mathsf{X})],\mathsf{genk}(\mathsf{X})))$ where $\mathcal{G}$ and $\mathcal{P}$ are as before.

$$\begin{array}{lll} \mathsf{R}_2 \to \mathsf{f}(\mathsf{R}_4^{\sigma_3},\mathsf{genk}(\mathsf{X})) & \mathsf{R}_3^{\sigma_4}\sigma_1 \to \mathsf{R}_3\sigma_4 & \mathsf{R}_4^{\sigma_3}\sigma_2 \to \mathsf{R}_4\sigma_3 \\ \mathsf{R}_2 \to \mathsf{f}(\mathsf{R}_3^{\sigma_4},\mathsf{genk}(\mathsf{X})) & \mathsf{R}_3 \to \mathsf{a} & \mathsf{R}_4 \to \mathsf{h}(\mathsf{genh}(\mathsf{X})) \\ & \mathsf{X}\sigma_3 \to \mathsf{X} & \end{array}$$

The rules are the result of the two minimally reachable matches σ_3 (which is the substitution $\{\mathsf{X} \mapsto \mathsf{X}\}$) of $\mathsf{genh}(\mathsf{X}\sigma_2)$ against $\mathsf{genh}(\mathsf{S}(\mathsf{X}))$ and σ_4 (which is the empty substitution) of $\mathsf{genh}(\mathsf{X}\sigma_1)$ against $\mathsf{genh}(\mathsf{0})$. The two patterns are the LHSs of $\mathcal{P}$'s 4th and 3rd rule.

An ILTG completion algorithm Let $n \ge 0$. Using the operation $\mathrm{Ext}^n_{\mathcal{P},\mathcal{G}}(\text{-})$ on $\mathcal{G}$-rules, we first define an operator $\delta^n_{\mathcal{P}}(\text{-})$ on ILTGs, and then construct a sequence of ILTGs by iterating it.

$$\delta^n_{\mathcal{P}}(\mathcal{G}) := \bigcup_{A\,\gamma \to C[\mathrm{dist}_{\alpha_1 \dots \alpha_k}(g')] \in \mathcal{G}} \mathrm{Ext}^n_{\mathcal{P},\mathcal{G}}(A\,\gamma \to C[\mathrm{dist}_{\alpha_1 \dots \alpha_k}(g')]) \;\cup\; \top(\mathcal{G})$$

where (writing $\gamma' < \gamma$ to mean "γ' is a proper prefix of γ")

$$\top(\mathcal{G}) := \bigcup_{\substack{A \ne R_i^\sigma \\ A\gamma \to t \in \mathcal{G}}} \bigcup_{\gamma' < \gamma} \{A\gamma'\top \to \mathrm{aprx}_0(t)\} \;\cup \bigcup_{R_i^\sigma \gamma \to t \in \mathcal{G}} \bigcup_{\gamma' < \gamma} \{R_i^\sigma \gamma'\top \to \mathrm{aprx}_1(t)\}$$

The construction then proceeds as follows. Starting off with an ILTG $\mathcal{G}_0$, we inductively construct a sequence of ILTGs by $\mathcal{G}^n_0 := \mathcal{G}_0$ and $\mathcal{G}^n_{i+1} := \mathcal{G}^n_i \cup \delta^n_{\mathcal{P}}(\mathcal{G}^n_i)$. For each $n \ge 0$, this gives an increasing sequence of ILTGs (*qua* sets of rules): $\mathcal{G}^n_0 \subseteq \mathcal{G}^n_1 \subseteq \mathcal{G}^n_2 \subseteq \mathcal{G}^n_3 \subseteq \dots$

▸ Remark 3.5. The seed, $\mathcal{G}_0$, is an arbitrary uniform ILTG (conforming to Notational Convention 2.8). Note that uniformity is not a real restriction since every ILTG can be transformed to an equivalent uniform ILTG. It follows from Proposition 2.4 that the input set of our algorithm can be an arbitrary pushdown-tree language.

▸ **Example 3.6.** Combining the rules below and Examples 3.1 and 3.4 we get the result of our procedure to approximate the collecting semantics of the program $\mathcal{P}$ in Example 2.1 on the input grammar $\mathcal{G}_0 = \{\mathsf{R}_0 \to \mathsf{counter}(\mathsf{0})\}$.

//... the rules of Examples 3.1 and 3.4
$\mathsf{R}_2 \to \mathsf{f}(\mathsf{R}_4^{\sigma_3}, \mathsf{R}_5^{\sigma_4}) \mid \mathsf{f}(\mathsf{R}_4^{\sigma_3}, \mathsf{R}_6^{\sigma_3})$
$\mathsf{R}_2 \to \mathsf{f}(\mathsf{R}_3^{\sigma_4}, \mathsf{R}_5^{\sigma_4}) \mid \mathsf{f}(\mathsf{R}_3^{\sigma_4}, \mathsf{R}_6^{\sigma_3})$
$\mathsf{R}_4 \to \mathsf{h}(\mathsf{R}_3^{\sigma_4}) \mid \mathsf{h}(\mathsf{R}_4^{\sigma_3})$
$\mathsf{R}_5^{\sigma_4}\sigma_1 \to \mathsf{R}_5\sigma_4$
$\mathsf{R}_6^{\sigma_3}\sigma_2 \to \mathsf{R}_6\sigma_3$
$\mathsf{R}_5 \to \mathsf{b}$
$\mathsf{R}_6 \to \mathsf{k}(\mathsf{genk}(\mathsf{X})) \mid \mathsf{k}(\mathsf{R}_5^{\sigma_4}) \mid \mathsf{k}(\mathsf{R}_6^{\sigma_3})$

Note that the results introduced by the operator $\mathsf{T}(-)$ have been omitted. It can be easily seen that the reachable ground terms of the resulting ILTG are precisely the terms reachable by $\mathcal{P}$ from $\mathcal{G}_0$. Further the ground terms reachable from $\mathsf{R}_2\sigma_2^n\sigma_1$ are

$$\mathsf{f}(\mathsf{h}^i(\mathsf{genh}^{n-i}(\mathsf{a}),\mathsf{k}^j(\mathsf{genk}^{n-j}(\mathsf{b}))))$$

which are the result terms bound to the substitution $\sigma_2^n \circ \sigma_1$ in Z_2 in $\mathcal{P}$'s collecting semantics (where we write $f^n = f \circ \ldots \circ f$ n-times), letting n range over all n we get precisely Z_2.[5] Moreover, the set of reachable ground constructor terms from R_0 is precisely $\{\, f(h^n(a), k^n(b)) \mid n \geq 0 \,\}$. Note that the tree language generated by $\mathcal{P}$ from $\mathcal{G}_0$ is non-regular. Thus, we can see that our algorithm makes use of the greater expressivity provided by ILTGs to describe this set.

4 Termination

There are two key ideas in the termination proof of the ILTG completion algorithm. The first, due to Jones and Andersen, concerns minimally reachable match. By considering only these substitutions when constructing $\mathrm{Ext}^n_{\mathcal{P},\mathcal{G}_i}(\text{-})$, the RHS of every rule that is generated is guaranteed to be a *variant* of a subterm of the RHS of either a $\mathcal{P}$-rule or a $\mathcal{G}_0$-rule; and the set of such variants is bounded. Secondly, the merging of substitutions via *erase* ensures that only finitely many indices are generated eventually.

Variants Let $\mathcal{X}$ be a set of terms. A $\mathcal{X}$-*variant* of t is a term obtained from t by replacing one or more subterms by an element of $\mathcal{X}$. For example, the term $f(g(A\alpha\beta, B\alpha\alpha\alpha), a)$ is a $\mathcal{N}'\mathcal{F}'^{\leq 3}$-variant of the term $t = f(g(h(A), f(A', b, c), a)$, where $\mathcal{N}' = \{A, B\}$ and $\mathcal{F}' = \{\alpha, \beta\}$ and we write $\mathcal{A}^{\leq k}$ for the set of sequences of elements of $\mathcal{A}$ of length no more than k. Note that the term t trivially is a $\mathcal{T}_\Sigma(\mathcal{N}\mathcal{F}^*)$ -variant of itself. The following definition makes the variant relation precise.

▸ **Definition 4.1** ($\mathcal{X}$-Variant)**.** Let $\mathcal{X} \subseteq \mathcal{T}_\Sigma(\mathcal{N}\mathcal{F}^*)$. We define the $\mathcal{X}$-variant relation $\sqsubseteq_\mathcal{X} \subseteq \mathcal{T}_\Sigma(\mathcal{N}\mathcal{F}^*) \times \mathcal{T}_\Sigma(\mathcal{N}\mathcal{F}^*)$ by induction over the following rules:

- $t \sqsubseteq_\mathcal{X} t$ for $t \in \mathcal{T}_\Sigma(\mathcal{N}\mathcal{F}^*)$
- $s \sqsubseteq_\mathcal{X} t$ for $s \in \mathcal{X}$ and $t \in \mathcal{T}_\Sigma(\mathcal{N}\mathcal{F}^*)$
- if $t_i \sqsubseteq_\mathcal{X} t'_i$ for each $1 \leq i \leq n$ then $f(t_1, \ldots, t_n) \sqsubseteq_\mathcal{X} f(t'_1, \ldots, t'_n)$

If $s \sqsubseteq_\mathcal{X} t$ we say s is an $\mathcal{X}$-*variant* of t.

We note at this point that for a fixed $\mathcal{X}$ the relation $\sqsubseteq_\mathcal{X}$ is reflexive and transitive.

The next proposition captures the observation that our flow analysis procedure does not really add "new information". It turns out that all RHSs of rules in $\mathcal{G}_i^n$ are in fact variants of subterms of terms t where t ranges over the RHSs of rules in $\mathcal{G}_0$ and $\mathcal{P}$.

For $i, n \geq 0$, define $\mathcal{F}_{i,n}$ be the index set of $\mathcal{G}_i^n$ and write $\mathcal{N}_{i,n}$ ($\mathcal{N}_0$) for the set of non-terminals occurring in $\mathcal{G}_i^n$ ($\mathcal{G}_0$).

[5] Owing to the "merging" of substitutions, it is not always possible to identify indices with substitutions.

▸ **Proposition 4.2.** *Assume that $\mathcal{G}_0$ is uniform. For each $i, n \geq 0$ and each rule $A\,\gamma \to t$ in $\mathcal{G}_i^n$, there exist a rule $l \to r$ in $\mathcal{P} \cup \mathcal{G}_0$ and $r' \leq r$ such that $t \sqsubseteq_{\mathcal{X}} r'$ where $\mathcal{X} = \mathcal{N}_{i,n}\mathcal{F}_{i,n}^{\leq \max(n+1,2)}$, and t is uniform.*

In the following we write $\mathcal{V}_{\mathcal{P}} = \bigcup_{i=1}^{p} Vars(l_i)$ and $\mathcal{N}_{\mathcal{P}} = \{ R_i \mid 1 \leq i \leq p \}$.

▸ **Lemma 4.3.** *For each $i, n \geq 0$, $\mathcal{F}_{i,n} \subseteq \mathcal{F}_0 \cup \{ \top \} \cup \{ \sigma \mid \sigma$ has type $\mathcal{V}_{\mathcal{P}} \to \mathcal{Y} \}$ where*

$$\mathcal{Y} \coloneqq \{ t \in \mathcal{T}_\Sigma(\mathcal{N}\mathcal{F}^*) \mid \text{there exist } l \to r \in \mathcal{G}_0 \cup \mathcal{P} \text{ and } r' \leq r \text{ such that } t \sqsubseteq_{\mathcal{N}_{\mathcal{P}} \cup \mathcal{V}_{\mathcal{P}}} r' \}$$

and $\mathcal{F}_0$ is the index set of $\mathcal{G}_0$ (which is defined to be $\varnothing$ in case $\mathcal{G}_0$ is regular). Hence there is some $m \geq 0$ such that for all $i, n \geq 0$, $|\mathcal{F}_{i,n}| \leq m$.

Termination of our completion procedure is an immediate consequence of Proposition 4.2 and Lemma 4.3.

▸ **Theorem 4.4** (Termination)**.** *For each $n \geq 0$, there is some $i \geq 0$ such that $\mathcal{G}_i^n = \mathcal{G}_{i+1}^n$. I.e. the algorithm terminates.*

We will refer to the fixpoint ILTG by $\mathcal{G}^n$ from now on.

We can give the following size bound for $\mathcal{G}^n$.

$$|\mathcal{G}^n| = O\left((\text{size}(\mathcal{P}) + \text{size}(\mathcal{G}_0))^{2|\mathcal{V}_{\mathcal{P}}| m(n,\mathcal{D})} \times \text{size}(\mathcal{P})^{\mathcal{D}|\mathcal{V}_{\mathcal{P}}| m(n,\mathcal{D}) + \mathcal{D} + 1} \right)$$

where $m(n, \mathcal{D}) = \mathcal{D}(\max(n+1, 2) + 1) + n + 1$, $\text{size}(\mathcal{P})$ is linear in $|\mathcal{P}|$, $|\mathcal{V}_{\mathcal{P}}|$ and the number of subterms of the largest RHS of $\mathcal{P}$, and $\text{size}(\mathcal{G}_0)$ is linear in $|\mathcal{G}_0|$, the number of $\mathcal{G}_0's$ non-terminals, indices and LHSs and number of subterms of the largest RHS of $\mathcal{G}_0$. Note that $\mathcal{D}$ is greatest number $(k+1)^d$ such that $d = \text{depth}(r)$ and k is the arity of a Σ symbol occurring in r, where r ranges over the RHSs of $\mathcal{P}$ and $\mathcal{G}_0$. Thus, we can see the size of $\mathcal{G}^n$ is polynomial in the number of rules in $\mathcal{P}$ and $\mathcal{G}_0$, and exponential in n, $\mathcal{D}$ and the number of variables.[6] For comparison, a similar analysis yields that the size of the fixpoint grammar of Jones and Andersen's procedure is $O(\text{size}(\mathcal{G}_0) + \text{size}(\mathcal{P}))^3 \times \text{size}(\mathcal{P})^{\mathcal{D}})$.

5 Soundness: Local and Global Safety

A program $\mathcal{P}$ (or an ILTG $\mathcal{G}$) determines a transition graph whose vertices are terms and whose edge-set is the rewrite relation of the program (or grammar). In such a setting it is natural to consider *simulation*. A key insight of our soundness proof is that reachability under ILTG $\mathcal{G}^n$ is *invariant under $\mathcal{P}$-transition, modulo simulation*. I.e. whenever a term $t \in \mathcal{T}_\Sigma(\mathcal{N}\mathcal{F}^*)$, which is reachable from s under rewriting by $\mathcal{G}^n$, can make a $\mathcal{P}$-transition to t', then there is a term, which is reachable from s under rewriting by $\mathcal{G}^n$, that simulates t'.

The main technical result (Theorem 5.12) is that $\mathcal{G}^n$ is locally safe, from which global safety follows. We organise our proof as follows. First, we identify a crucial property of ILTGs that are candidates for approximating the collecting semantics of $\mathcal{P}$ on I, called *emulation* (Definition 5.5). We show that emulation implies invariance under $\mathcal{P}$-transition, modulo simulation (Proposition 5.6), which implies local safety (Proposition 5.7). It then remains to show that $\mathcal{G}^n$ satisfies emulation (Theorem 5.10).

[6] Note that $\mathcal{D}$ can be made into a constant by enforcing a constraint on the depth of RHSs. A program can be transformed into a conforming program by introducing "subroutines" to reduce the depth of RHSs. The increase in rules is then polynomial in the length of $\mathcal{P}$.

Simulation Fix $\Sigma, \mathcal{N}, \mathcal{F}$ and a program $\mathcal{P} = \{ l_i \to r_i \mid 1 \leq i \leq p \}$ (over Σ) and let the meta-variable $\mathcal{R}$ range over $\mathcal{P}$ and ILTGs $\mathcal{G}$.

▸ **Definition 5.1** (Σ-equal $\mathcal{R}$-simulation)**.** We call a relation $R \subseteq (\mathcal{T}_\Sigma(\mathcal{N}\mathcal{F}^*))^2$ a Σ-*equal pre-$\mathcal{R}$-simulation* just if $R \subseteq F_\mathcal{R}(R)$, where $F_\mathcal{R}(R)$ is defined to be the set of pairs $(t_1, t_2) \in (\mathcal{T}_\Sigma(\mathcal{N}\mathcal{F}^*))^2$ satisfying

(i) $\widehat{t_1} = \widehat{t_2}$, where $\widehat{t} \in \mathcal{T}_\Sigma(*)$ is obtained from t by replacing every $\mathcal{N}\mathcal{F}^*$-subterm of t by a special symbol $*$, which is assumed not to be a member of $\Sigma \cup \Delta$; and
(ii) for every t_1' if $t_1 \to_\mathcal{R} t_1'$ then there exists t_2' such that $t_2 \to_\mathcal{R} t_2'$ and $(t_1', t_2') \in R$.

Since $F_\mathcal{R}$ is a monotone function, by Knaster-Tarski Fixpoint Theorem, it has a greatest fixpoint, which we write as $\preceq_\mathcal{R}$ and refer to as Σ-*equal $\mathcal{R}$-simulation* (or simply *$\mathcal{R}$-simulation*). We read $s \preceq_\mathcal{R} t$ as "t $\mathcal{R}$-simulates s".

We denote the intersection of the $\mathcal{G}$-simulation and $\mathcal{P}$-simulation by $\preceq_{\mathcal{G},\mathcal{P}}$ i.e. $\preceq_{\mathcal{G},\mathcal{P}} := \preceq_\mathcal{G} \cap \preceq_\mathcal{P}$. Whenever it is clear from the context, we drop the subscript and write it simply as $\preceq$.

▸ **Proposition 5.2.** *Let $\mathcal{G}$ be an ILTG, then the relation $\preceq$ has the following properties.*

(i) Whenever two terms are related by $\preceq$, if one of them is ground (i.e. a member of $\mathcal{T}_\Sigma$) then so is the other, and they are equal terms.
(ii) $\preceq$ is a pre-congruence i.e. if $t \preceq t'$ then $C[t] \preceq C[t']$.
(iii) $\preceq$ is a preorder on $\mathcal{T}_\Sigma(\mathcal{N}\mathcal{F}^*)$.
(iv) If $t_1 \preceq t_2$ and $t_1 \to^* t_1'$ then there is a term t_2' such that $t_2 \to^* t_2'$ and $t_1' \preceq t_2'$
(v) If $t \preceq t'$ and $t \to^* s$ for some $s \in \mathcal{T}_\Sigma$, then $t' \to^* s$.
(vi) If $t \preceq t'$ then $Reach^o_\mathcal{G}(t) \subseteq Reach^o_\mathcal{G}(t')$.

▸ **Proposition 5.3.** *Let t_0 be a term such that if $t \in Reach_\mathcal{G}(t_0)$ and $t \to_\mathcal{P} t'$ then there exists $t'' \in Reach_\mathcal{G}(t_0)$ such that $t' \preceq t''$. Then*

(i) for each $n \geq 0$, if $t \in Reach_\mathcal{G}(t_0)$ and $t \to^n_\mathcal{P} t'$ then there exists $t'' \in Reach_\mathcal{G}(t_0)$ such that $t' \preceq t''$.
(ii) $Reach^o_\mathcal{P}(Reach_\mathcal{G}(t_0)) \subseteq Reach^o_\mathcal{G}(t_0)$

Emulation The collecting semantics of a program $\mathcal{P}$ on an input set I can be characterised as follows [10, Lemma 2.6]. We aim to introduce a notion (called emulation) that mimicks the property.

▸ **Lemma 5.4** (Jones and Andersen 2007)**.** *The collecting semantics of $\mathcal{P}$ on I is the smallest (ordered point-wise) tuple of sets of pairs, $(Z_0, Z_1, \ldots, Z_p)$, that satisfies:*

(1) If $g \in I$ then $(id, g) \in Z_0$.
(2) If $(\sigma, C[\sigma' \, l_i]) \in Z_j$ then $(\sigma', \sigma' \, r_i) \in Z_i$ for $0 \leq i, j \leq p$.
(3) If $(\sigma, C[\sigma' \, l_i]) \in Z_j$ and $(\sigma', g') \in Z_i$, then $(\sigma, C[g']) \in Z_j$ for $0 \leq i, j \leq p$.

In the definition to follow, we assume that ITLGs $\mathcal{G}_0$ and $\mathcal{G}$ that satisfy the Notational Convention 2.8, with the input to $\mathcal{P}$ set to $Reach_{\mathcal{G}_0}$. Let the meta-variable A range over the following subset of non-terminals (of $\mathcal{G}_0$ and $\mathcal{G}$)

$$\{ R_0, R_1, \ldots, R_p \} \cup \{ X \mid X \in Vars(r_i), 1 \leq i \leq p \}. \tag{1}$$

▸ **Definition 5.5** (Emulation)**.** An ILTG $\mathcal{G}$ *emulates* the collecting semantics of $\mathcal{P}$ on input $Reach^o_{\mathcal{G}_0}$ just if

(i) $Reach_{\mathcal{G}_0} \subseteq Reach_{\mathcal{G}}(R_0)$
(ii) whenever $C[\sigma\, l_i] \in Reach_{\mathcal{G}}(A\,\gamma)$ then there are $\delta_\sigma \in \mathcal{F}^*$ and substitution σ' such that
  (a) $\sigma'\, r_i \in Reach_{\mathcal{G}}(R_i\,\delta_\sigma)$, $\sigma' Z \in Reach_{\mathcal{G}}(Z\,\delta_\sigma)$ for each $Z \in Vars(r_i)$, and $\sigma \preceq \sigma'$ (i.e. $\sigma\, X \preceq \sigma'\, X$ for all $X \in \mathcal{V}$)
  (b) if $t \in Reach_{\mathcal{G}}(R_i\,\delta_\sigma)$ then $C[t] \in Reach_{\mathcal{G}}(A\,\gamma)$.

▸ **Proposition 5.6.** *Let $\mathcal{G}$ be an ILTG that emulates the collecting semantics of $\mathcal{P}$ on $Reach^o_{\mathcal{G}_0}$. Then, with A ranging over the set (1) of non-terminals, and δ ranging over $\mathcal{F}^*$*

(i) if $t \in Reach_{\mathcal{G}}(A\,\delta)$ and $t \to^*_{\mathcal{P}} t'$ then there exists $t'' \in Reach_{\mathcal{G}}(A\,\delta)$ such that $t' \preceq t''$.
(ii) $Reach^o_{\mathcal{P}}(Reach_{\mathcal{G}}(A\delta)) \subseteq Reach^o_{\mathcal{G}}(A\delta)$.

Proof. (i) Let $t \in Reach_{\mathcal{G}}(A\delta)$. We assume that $t \to_{\mathcal{P}} t'$; the general case of $t \to^*_{\mathcal{P}} t'$ then follows from Proposition 5.3. By assumption, $t = C[\sigma l_i]$ and $t' = C[\sigma r_i]$ for some i. Thus by assumption of emulation there exist σ' and δ_σ such that $\sigma' r_i \in Reach_{\mathcal{G}}(R_i\delta_\sigma)$ and $\sigma \preceq \sigma'$. So $C[\sigma r_i] \preceq C[\sigma' r_i]$ as $\preceq$ is a pre-congruence. Also since $\sigma' r_i \in Reach_{\mathcal{G}}(R_i\delta_\sigma)$, it follows from the emulation assumption that $C[\sigma' r_i] \in Reach_{\mathcal{G}}(A\delta)$.

(ii) Follows from Proposition 5.3 when combined with (i).

◂

▸ **Proposition 5.7.** *Let $\mathcal{G}$ be an ILTG that emulates the collecting semantics of $\mathcal{P}$ on $Reach^o_{\mathcal{G}_0}$. Then $\mathcal{G}$ is locally safe for $\mathcal{P}$ on $Reach^o_{\mathcal{G}_0}$.*

Proof. Let $(\sigma, g) \in Z_i$, then there exists $w \in Reach^o_{\mathcal{G}_0}$ and context $C[\,\text{-}\,]$ such that $w \to^*_{\mathcal{P}} C[\sigma\, l_i]$ and $\sigma\, r_i \to^*_{\mathcal{P}} g$. Note that $C[\sigma\, l_i]$, $\sigma\, r_i$ and g are elements of $\mathcal{T}_\Sigma$. Thus $C[\sigma l_i] \in Reach^o_{\mathcal{P}}(Reach_{\mathcal{G}_0}) \subseteq Reach^o_{\mathcal{P}}(Reach_{\mathcal{G}}(R_0)) \subseteq Reach^o_{\mathcal{G}}(R_0)$, the second inclusion follows from Proposition 5.6(ii). Thus by definition there exist σ' and δ_σ such that $\sigma'\, r_i \in Reach_{\mathcal{G}}(R_i\,\delta_\sigma)$, $\sigma'\, X \in Reach_{\mathcal{G}}(X\,\delta_\sigma)$ for each $X \in Vars(r_i)$, and $\sigma \preceq \sigma'$.

Since $\mathrm{range}(\sigma) \subseteq \mathcal{T}_\Sigma$, $\sigma \preceq \sigma'$ and $\preceq$ is Σ-equal, we have that $\sigma = \sigma'$. Hence we can conclude that in fact $\sigma\, r_i \in Reach_{\mathcal{G}}(R_i\,\delta_\sigma)$ and $\sigma X \in Reach_{\mathcal{G}}(X\,\delta_\sigma)$. Now since $\sigma\, r_i \to^*_{\mathcal{P}} g$, it follows from Proposition 5.6(ii) that $g \in Reach^o_{\mathcal{P}}(Reach_{\mathcal{G}}(R_i\,\delta_\sigma)) \subseteq Reach^o_{\mathcal{G}}(R_i\,\delta_\sigma)$. Thus $R_i\,\delta_\sigma \to^* g$ and $X\,\delta_\sigma \to^* \sigma X$.

◂

▸ Remark 5.8. Define a simulation relation over index sequences by $\gamma \preceq \gamma'$ just if $A\,\gamma \preceq A\,\gamma'$ for all $A \in \mathcal{N}$. If an ILTG is locally safe for $\mathcal{P}$ on $Reach^o_{\mathcal{G}_0}$ and has an index $\top$ (say) that simulates every index (i.e. $\alpha \preceq \top$ for all $\alpha \in \mathcal{F}$), then it is also globally safe, for we can set $\widetilde{R_i} := R_i\,\top$ and $\widetilde{X} := X\,\top$.

$\mathcal{G}^n$ emulates the collecting semantics For our soundness argument it remains to show that for each n, the fixpoint ILTG $\mathcal{G}^n$ emulates the collecting semantics. We begin by stating a technical lemma.

▸ **Lemma 5.9.** *For each $n \geq 0$, if $A\gamma \to^*_{\mathcal{G}^n} C[\sigma l_i]$ then there are σ_0, σ'_0 and δ such that*

(i) $A\gamma \to^*_{\mathcal{G}^n} C[\sigma_0 l_i]$ and $\sigma_0 r_i \to^*_{\mathcal{G}^n} \sigma r_i$
(ii) $A\gamma \to^*_{\mathcal{G}^n} C[R_i\delta]$
(iii) $X\,\delta \to^*_{\mathcal{G}^n} \sigma'_0 X$, for each $X \in Vars(r_i)$
(iv) $\sigma_0 \preceq \sigma'_0$

Emulation is a straightforward consequence of the above result.

▸ **Theorem 5.10.** *For each $n \geq 0$, $\mathcal{G}^n$ emulates the collecting semantics of $\mathcal{P}$ on $Reach^o_{\mathcal{G}_0}$.*

Proof. We will now show that the conditions (i), (iia) and (iib) of Definition 5.5 hold for $\mathcal{G}^n$. By construction it is the case that $\mathcal{G}^n \supseteq \mathcal{G}_0$, thus $Reach_{\mathcal{G}_0} = Reach_{\mathcal{G}_0}(R_0) \subseteq Reach_{\mathcal{G}^n}(R_0)$. Therefore condition (i) is satisfied. For condition (iia) and (iib), suppose that $C[\sigma l_i] \in Reach_{\mathcal{G}^n}(A\gamma)$. By applying Lemma 5.9 to $A\gamma \to^*_{\mathcal{G}^n} C[\sigma l_i]$, it follows that for some δ_σ,σ_0,σ'_0, we have $A\gamma \to^* C[R_i\delta_\sigma]$, $X\delta_\sigma \to^* \sigma'_0 X$ for every X that occurs in r_i, $\sigma_0 r_i \to^*_{\mathcal{G}^n} \sigma r_i$ and $\sigma_0 \preceq \sigma'_0$. Since $\sigma_0 \preceq \sigma'_0$, $\sigma_0 X \to^*_{\mathcal{G}^n} \sigma X$, there is a term $\sigma' X$ such that $\sigma'_0 X \to^*_{\mathcal{G}^n} \sigma' X$ and $\sigma X \preceq \sigma' X$ for all $X \in Vars(r_i)$. Thus we can infer that $\sigma \preceq \sigma'$ and $\sigma'_0 r_i \to^*_{\mathcal{G}^n} \sigma' r_i$. Hence $X\delta_\sigma \to^*_{\mathcal{G}^n} \sigma'_0 X \to^*_{\mathcal{G}^n} \sigma' X$. Further $R_i\delta_\sigma \to \mathrm{dist}_{\delta_\sigma} r_i \to^* \sigma' r_i$ by rewriting all non-terminals $X \in Vars(r_i)$ to $\sigma' X$. We can thus infer that $\sigma' r_i \in Reach_{\mathcal{G}^n}(R_i\delta_\sigma)$, $\sigma' X \in Reach_{\mathcal{G}^n}(X\delta_\sigma)$ where $\sigma \preceq \sigma'$. Hence condition (iia) holds. For condition (iib) further suppose that $t \in Reach_{\mathcal{G}^n}(R_i\delta_\sigma)$, then since $A\gamma \to^* C[R_i\delta_\sigma]$ and $R_i\delta_\sigma \to^* t$ we can rewrite $A\gamma \to^* C[t]$, i.e. $C[t] \in Reach_{\mathcal{G}^n}(A\gamma)$. ◂

Thus it follows from Proposition 5.7 that $\mathcal{G}^n$ is locally safe for each $n \geq 0$. To prove global safety, we combine the following lemma with Remark 5.8.

▸ **Proposition 5.11.** *For all $\gamma_0, \gamma_1, \gamma_2 \in \mathcal{F}^*$ we have $\gamma_0\gamma_1 \preceq \gamma_0 \top \gamma_2$ in $\mathcal{G}^n$.*

To summarise

▸ **Theorem 5.12.** *For each $n \geq 0$, the fixpoint ILTG $\mathcal{G}^n$ is both locally and globally safe for program $\mathcal{P}$ on input $Reach_{\mathcal{G}_0}$.*

6 Related Work

TRS Reachability Problem There is a line of work in the rewriting community devoted to the construction of $Reach_{\mathcal{R}}(I)$ where I is a regular set and $\mathcal{R}$ is a TRS satisfying various restrictions. This is an interesting problem because for a regular input set I, $Reach_{\mathcal{R}}(I)$ is not necessarily regular, even if $\mathcal{R}$ is a confluent and terminating linear TRS [7].

Based on earlier work by Genet [6], Feuillade et al. [5] proposed a tree-automaton completion algorithm for over-approximating $Reach_{\mathcal{R}}(I)$, for a given (left-linear) TRS $\mathcal{R}$ and a regular input set I. The algorithm constructs a sequence of tree automata and is parametrised by an abstraction function that maps terms to states of the automaton. This method is quite versatile as the completion procedure can be fine-tuned by choosing the appropriate abstraction function. However, the approach is not fully automatic; further not every abstraction function is guaranteed to lead to a fixpoint automaton.

Building on this work, Boichut et al. [2] introduced a semi-algorithm which automatically chooses abstraction functions for the completion procedure. Their aim is to obtain a more conclusive analysis of whether a term t is reachable from a given input set. The abstraction function is refined in order to obtain either a fixpoint automaton which does not accept t or an under-approximation of $Reach_{\mathcal{P}}(I)$ which does include t. However, this approach is not guaranteed to terminate.

Model Checking Functional Programs Kobayashi et al. [11] introduced a type-based model-checking method for an extension of higher-order recursion schemes called *higher-order multi-parameter tree transducers* (HMTT). They gave an algorithm for checking if the tree generated by a given HMTT satisfies a given output specification, provided the input trees conform to a given input specification. It is not easy to compare our work with theirs, but two aspects stand out. First their specifications are restricted to regular tree

languages. Secondly patterns (for matching) in their framework are required to satisfy a rigid type constraint; for example, their method cannot handle our Example 2.1 (unless certain invariants on intermediate data structures are provided by the programmer [21]).

Ong and Ramsay [19] recently introduced *pattern-matching recursion schemes* (PMRS) as an accurate model of computation for functional programs that manipulate algebraic data types. They present a verification method that, given an order-n PMRS $\mathcal{P}$ and an input set I generated by a regular tree grammar, constructs an order-n weak PMRS which over-approximates the set of terms reachable from I under rewriting from $\mathcal{P}$. Their construction uses a binding analysis à la Jones and Andersen to over-approximate only the first-order pattern-matching behaviour, whilst remaining completely faithful to the higher-order control flow. We believe that their binding analysis can be refined by using a variant of our ILTG-based completion algorithm, thus giving a more accurate over-approximation of pattern-matching in their framework.

XML Type Checking The extensible markup language XML is the standard format for exchanging structured data. Central to XML processing is the *type checking problem*: given an input type, an output type and a transformation f, does f transform every input that conforms to the input type to an output that conforms to the output type? Since the XML type checking problem is undecidable, general solutions are necessarily approximate. However, by restricting types and transformations appropriately, type checking can be made decidable. A recent direction [14, 15] considers types given by languages recognisable by finite-state automata, and transformations specified by (versions of) stay macro tree transducers (SMTTs). SMTTs are first-order functional programs that generate output trees by top-down pattern matching its first (tree) argument, while possibly accumulating intermediate results in the other (tree) parameters: they are essentially first-order pattern-matching recursion schemes [19]. It would be interesting to understand the approach based on SMTTs, as it seems likely that there are connections with our work.

7 Evaluation, Conclusion and Further Directions

Evaluation A few remarks by way of comparison with related work. (i) Our algorithm can take an arbitrary pushdown tree language as the input set. To our knowledge, all published over-approximation results for the reachability problem for left-linear TRS assume a regular input set. (ii) For each fixed $n \geq 0$, our completion method is at least as accurate as Jones and Andersen's [10] (because after erasing all references to indices, our method yields the same result as Jones and Andersen's, modulo some superfluous rules). (iii) A source of inaccuracy in Jones and Andersen's approach is the decoupling of the pairing $(\sigma, t) \in Z_i$ (in the collecting semantics) between a reachable substitution σ and the associated result term t. The notion of emulation allows us to introduce a weak form of coupling between σ and t in the sense of local safety. The definition of emulation also applies to Jones and Andersen's fixpoint regular tree grammar. However, in the absence of indices, $\delta_\sigma = \epsilon$. Thus, we can see that the coupling of program results and program states (substitutions) is stronger in our fixpoint ILTG. (iv) It is not straightforward to compare our result with related work from the rewriting community [9, 5, 2]. We can see from Example 2.1 that our approach accurately captures a non-regular set of reachable terms. Because Feuillade et al. and Boichut et al. use finite tree automata, such accuracy is beyond their reach. However, there is an example in the full version of this paper [13] for which our algorithm produces a strict over-approximation which is regular, whereas, by a judicious choice of parameters,

the algorithm of Feuillade et al. can yield a better over-approximation. On the other hand, our algorithm is fully automatic and guaranteed to terminate for each fixed n (as is Jones and Andersen's), neither of which is true of the methods of Feuillade et al. and of Boichut et al..

Conclusion Using indices to capture (sets of) substitutions, we have presented a completion algorithm which, given a left-linear TRS $\mathcal{P}$, an input set described by an ILTG $\mathcal{G}_0$ and $n \geq 0$, constructs an ILTG $\mathcal{G}^n$ that safely over-approximates the collecting semantics of $\mathcal{P}$ on $\mathcal{G}_0$. To our knowledge, this is the first completion procedure for pushdown tree automata, and it yields a strong approximation result for the left-linear TRS reachability problem.

Further Directions (i) A priority is to construct an implementation of our completion algorithm for empirical evaluation. (ii) A key idea of our approach is to merge substitutions, σ and σ', just when $\mathit{erase}(\sigma) = \mathit{erase}(\sigma')$. One way to improve our algorithm is to refine the merge operation. For example, one could define $\mathit{erase}^1(A\alpha_1 \dots \alpha_n) := A\,\mathit{erase}(\alpha_1) \dots \mathit{erase}(\alpha_n)$. A similar argument to our current termination algorithm should apply. One could envisage an inductive definition in the same style for erase^2, erase^3, ... (iii) It would be interesting to identify sufficient conditions for the ILTG completion algorithm to be accurate for reachability i.e. for which class of programs does the fixpoint ILTG generate precisely the set of reachable terms?

Acknowledgements Financial support by EPSRC (research grant EP/F036361/1 and OUCL DTG Account doctoral studentship for the first author) is gratefully acknowledged. We would like to thank Damien Sereni and Steven Ramsay for helpful discussions and insightful comments, and the anonymous reviewers for their detailed reports.

References

1 Alfred V. Aho. Indexed grammars - an extension of context-free grammars. *Journal of the ACM (JACM)*, 15(4):647–671, 1968.

2 Yohan Boichut, Roméo Courbis, Pierre-Cyrille Héam, and Olga Kouchnarenko. Finer is better: Abstraction refinement for rewriting approximations. In *Proceedings of Rewriting Techniques and Applications (RTA)*, volume 5117 of *Lecture Notes in Computer Science*, pages 48–62. Springer, 2008.

3 Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, pages 238–252. ACM, 1977.

4 Joost Engelfriet and Heiko Vogler. Pushdown machines for the macro tree transducer. *Theoretical Computer Science*, 42:251–368, 1986.

5 Guillaume Feuillade, Thomas Genet, and Valérie Viet Triem Tong. Reachability Analysis over Term Rewriting Systems. *Journal of Automated Reasoning*, 33(3-4):341–383, 2005.

6 Thomas Genet. Decidable approximations of sets of descendants and sets of normal forms. In *Proceedings of Rewriting Techniques and Applications (RTA)*, volume 1379 of *Lecture Notes in Computer Science*, pages 151–165. Springer, 1998.

7 Rémi Gilleron and Sophie Tison. Regular tree languages and rewrite systems. *Fundamenta Informaticae*, 24(1/2):157–174, 1995.

8 Irène Guessarian. Pushdown tree automata. *Mathematical Systems Theory*, 16(4):237–263, 1983.

9 Florent Jacquemard. Decidable approximations of term rewriting systems. In *Proceedings of Rewriting Techniques and Applications (RTA)*, volume 1103 of *Lecture Notes in Computer Science*, pages 362–376. Springer, 1996.

10 Neil D. Jones and Nils Andersen. Flow analysis of lazy higher-order functional programs. *Theoretical Computer Science*, 375(1-3):120–136, 2007.

11 Naoki Kobayashi, Naoshi Tabuchi, and Hiroshi Unno. Higher-order multi-parameter tree transducers and recursion schemes for program verification. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, pages 495–508. ACM, 2010.

12 Jonathan Kochems. Approximating reachable terms of functional programs. Oxford University MMathsCS 4th-year Project Report, 2010.

13 Jonathan Kochems and C.-H. Luke Ong. Improved functional flow and reachability analyses using indexed linear tree grammars. Long version, 2010.

14 Sebastian Maneth, Alexandru Berlea, Thomas Perst, and Helmut Seidl. Xml type checking with macro tree transducers. In *Proceedings of the Symposium on Principles of Database Systems (PODS)*, pages 283–294. ACM, 2005.

15 Sebastian Maneth, Thomas Perst, and Helmut Seidl. Exact xml type checking in polynomial time. In *Proceedings of the International Conference on Database Theory (ICDT)*, volume 4353 of *Lecture Notes in Computer Science*, pages 254–268. Springer, 2007.

16 A. N. Maslov. Multilevel stack automata. *Problems of Information Transmission*, 12:38–43, 1976.

17 C.-H. Luke Ong. On model-checking trees generated by higher-order recursion schemes. In *Proceedings of Symposium on Logic in Computer Science (LICS)*, pages 81–90. IEEE Computer Society, 2006.

18 C.-H. Luke Ong. Models of Higher-Order Computation: Recursion Schemes and Collapsible Pushdown Automata. In J. Esparza, B. Spanfelner, and O. Grumberg, editors, *Logics and Languages for Reliability and Security*, pages 263–300. IOS Press, 2010. NATO Science for Peace and Security Series, D: Information and Communication Security - Vol. 25.

19 C.-H. Luke Ong and Steven James Ramsay. Verifying Higher-Order Functional Programs with Pattern-Matching Algebraic Data Types. In *Proceedings of the Symposium on Principles of Programming Languages (POPL)*, pages 587–598. ACM, 2011.

20 John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998.

21 Hiroshi Unno, Naoshi Tabuchi, and Naoki Kobayashi. Verification of tree-processing programs via higher-order model checking. In *Proceedings of the Asian Symposium on Programming Languages and Systems (APLAS)*, volume 6461 of *Lecture Notes in Computer Science*, pages 312–327. Springer, 2010.

Higher Order Dependency Pairs for Algebraic Functional Systems

Cynthia Kop[1] **and Femke van Raamsdonk**[1]

1 Faculty of Sciences, VU, De Boelelaan 1081a, 1081 HV Amsterdam

Abstract

We extend the termination method using dynamic dependency pairs to higher order rewriting systems with beta as a rewrite step, also called Algebraic Functional Systems (AFSs). We introduce a variation of usable rules, and use monotone algebras to solve the constraints generated by dependency pairs. This approach differs in several respects from those dealing with higher order rewriting modulo beta (e.g. HRSs).

Keywords and phrases higher order rewriting, termination, dynamic dependency pairs

Digital Object Identifier 10.4230/LIPIcs.RTA.2011.203

Category Regular Research Paper

1 Introduction

An important method to (automatically) prove termination of first order term rewriting is the dependency pair approach by Arts and Giesl [3]. This approach transforms a rewrite system into groups of ordering constraints, such that rewriting is terminating if and only if the groups of constraints are (separately) solvable. Various optimizations of the method have been studied, see for example [7, 6].

This paper contributes to the study of dependency pairs for higher order rewriting. Higher order rewriting comes in different shapes. First, there is rewriting modulo $\alpha\beta\eta$ as in the higher order rewrite systems (HRSs) defined by Nipkow [20]; Klop's CRSs [13] and Khasidashvili's ERSs [12] are in some aspects similar. Various definitions of dependency pairs, often with optimizations, have been given for HRSs [23, 22, 17, 15, 24]. Second, applicative term rewriting systems with functional variables but no abstraction are sometimes considered as a (restricted) form of higher order rewriting. Also in this setting several definitions of dependency pairs exist [16, 18, 19, 1, 2, 8]. The aim of the present paper is to study dependency pairs for a third variant of higher order rewriting: *algebraic functional systems* (AFSs), introduced by Jouannaud and Okada [10]. In AFSs we consider simply typed terms, which are rewritten both using specific rewrite rules and β-reduction, with matching modulo α. While higher order versions of the recursive path ordering are commonly studied in the setting of AFSs [11, 5], there is little work on dependency pairs for this formalism.

We briefly discuss the ideas from studies of dependency pairs for HRSs and for applicative systems in Section 2; we also explain why those approaches do not quite, or not at all apply to the setting with AFSs. We define dependency pairs for AFSs in the so-called *dynamic* style, where functional variables in the right-hand side of a rewrite rule may give rise to dependency pairs. We study the notions of dependency chains, dependency graphs and reduction orders for AFSs with dynamic dependency pairs. To demonstrate that the dynamic approach has adequate strength even without restrictions, we also define a variant of usable rules and apply van de Pol's monotone algebra approach [21] to solve constraints generated by the method. The result is a method to prove termination (a complete method for left-linear

22nd International Conference on Rewriting Techniques and Applications.
Editor: M. Schmidt-Schauß; pp. 203–218

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

systems), which may serve as a basis for further definitions – for example static dependency pairs, or dynamic pairs with restrictions that allow us to drop the subterm property.

2 Background and Related Work

The extension of dependency pairs to the higher order case is not entirely straightforward and thus many variations exist. This work can roughly be split along two axes. On the one axis, the higher order formalism (we distinguish between applicative rewriting, rewriting modulo β (HRSs), and with β as a separate step (AFSs)), on the other the style of dependency pairs (with the common styles being *dynamic* and *static*). Figure 1 gives an overview.

	Applicative	**HRS**	**AFS**
Dynamic	[16]	[23] [15]	*this paper*
Static	[18] [19]	[4] [22] [17] [24]	[4]
Other	[1] [2] [8]	–	–

Figure 1 References on Higher Order Dependency Pairs

The dynamic and static approach differ in the treatment of leading variables in the right-hand sides of rules (subterms $x \cdot s_1 \cdots s_n$ with $n > 0$ and x a free variable). In the dynamic approach, such subterms lead to a dependency pair; in the static approach they do not. Consequently, first order techniques like argument filterings and usable rules are easier to extend to a static approach, while equivalence results tend to be limited to the dynamic style. Static dependency pairs can only be applied on systems satisfying certain restrictions.

Dependency pairs for applicative term rewriting We first say some words about applicative term rewriting. In applicative systems, terms are built from variables, constants and a binary application operator. Functional variables may be present, as in $x \cdot a$, but there is no abstraction, as in $\lambda x.\, x$. There are various styles of applicative rewriting.

A dynamic approach was defined both for untyped and simply-typed applicative systems in [16], along with a definition of argument filterings. A first static approach appears in [18] and is improved in [19]; the method is restricted to 'plain function passing' systems where, intuitively, leading variables are harmless. Due to the lack of binders, it is also possible to eliminate leading variables by instantiating them, as is done for simply typed systems in [1, 2]; in [8], an uncurrying transformation from untyped applicative systems to normal first order systems is used. These techniques have no parallel in rewriting with binders.

Unfortunately, they are not directly useful in the setting of AFSs, since termination may be lost by adding λ-abstraction and β-reduction. For example, the simply typed applicative system $\mathsf{app} \cdot (\mathsf{abs} \cdot F) \cdot x \to F \cdot x$, with $F : \iota \Rightarrow \iota$ a functional variable, $x : \iota$ a variable, and app, abs constants, is terminating because in every step the size of a term decreases. However, adding λ-abstraction and β-reduction spoils this property: with $\omega = \mathsf{abs} \cdot (\lambda x.\, \mathsf{app} \cdot x \cdot x)$ we have $\mathsf{app} \cdot \omega \cdot \omega = \mathsf{app} \cdot (\mathsf{abs} \cdot (\lambda x.\, \mathsf{app} \cdot x \cdot x)) \cdot \omega \to (\lambda x.\, \mathsf{app} \cdot x \cdot x) \cdot \omega \to \mathsf{app} \cdot \omega \cdot \omega$.

Dynamic Dependency Pairs for HRSs A first, very natural, definition of dependency pairs for HRSs is given in [23]. Here termination is not equivalent to the absence of infinite dependency chains, and a term is required to be greater than its subterms (the *subterm property*), which makes many optimizations impossible. Consequently, most of the focus since has been on the static approach. However, with restrictions on the rules the subterm property may be weakened, as discussed in [15] (extended abstract).

Static Dependency Pairs for HRSs The static approach in [18] is moved to the setting of HRSs in [17], and extended with argument filterings and usable rules in [24]. The static

approach omits dependency pairs $f^\#(\vec{l}) \rightsquigarrow x(\vec{r})$ with x a variable, which avoids the need of a subterm property. The technique is restricted to *plain function passing* HRSs; for example the (terminating) rule $\mathsf{foo}(\mathsf{bar}(\lambda x.\, F(x))) \to F(\mathsf{a})$ cannot be handled. In addition, bound variables may become free in a dependency pair. For instance, the rule $\mathsf{I}(\mathsf{s}(n)) \to \mathsf{twice}(\lambda x.\, I(x), n)$ generates a pair $\mathsf{I}^\#(\mathsf{s}(n)) \rightsquigarrow \mathsf{I}^\#(x)$ which admits an infinite dependency chain.

The definitions for HRSs [23, 17] do not immediately carry over to AFSs, since AFSs may have rules of functional type and β-reduction is a separate rewrite step. A short paper by Blanqui [4] introduces static dependency pairs on a form of rewriting which includes AFSs, but it restricts to base-type rules. The present work considers dynamic dependency pairs and is most related to [23], but is adaptated for the different formalism. Our method conservatively extends the one for first order rewriting and provides a characterization of termination for left-linear AFSs. We have chosen for a dynamic rather than a static approach because, although the static approach is stronger when applicable, the dynamic definitions can be given without restrictions. It would be nice for future work to integrate the two approaches; for the moment they co-exist with each their own advantages and disadvantages.

3 Preliminaries

We consider higher order rewriting as defined by Jouannaud and Okada, also called Algebraic Functional Systems (AFSs). Terms are built from simply typed variables, abstraction and application (as in simply typed λ-calculus), and in addition function symbols which take a fixed number of typed arguments. Terms and matching are modulo α, and β is a rewrite step. We follow roughly the definitions in [25, Chapter 11], as recalled below.

Types and Terms The set of *simple types* (or just *types*) is generated from a given set $\mathcal{B}$ of *base types* and the binary type constructor $\Rightarrow$, which is right-associative. Types are denoted by σ, τ and base types by ι, κ. A type with at least one occurrence of $\Rightarrow$ is called a *functional type*. A *type declaration* is an expression of the form $(\sigma_1 \times \ldots \times \sigma_n) \Rightarrow \tau$; if $n = 0$ this is written as just τ. Type declarations are not types, but are used for typing purposes.

We assume a set $\mathcal{V}$, consisting of infinitely many typed variables for each type, and a set $\mathcal{F}$ disjoint from $\mathcal{V}$, consisting of function symbols each equipped with a type declaration. Variables are denoted by x, y, z and function symbols by f, g, h or using more suggestive notation. To stress the type (declaration) of a symbol a we may write $a : \sigma$. *Terms* over $\mathcal{F}$ are those expressions s for which we can infer $s : \sigma$ for some type σ using the clauses:

(var)	$x : \sigma$	if $x : \sigma \in \mathcal{V}$
(app)	$s \cdot t : \tau$	if $s : \sigma \Rightarrow \tau$ and $t : \sigma$
(abs)	$\lambda x.\, s : \sigma \Rightarrow \tau$	if $x : \sigma \in \mathcal{V}$ and $s : \tau$
(fun)	$f(s_1, \ldots, s_n) : \tau$	if $f : (\sigma_1 \times \ldots \times \sigma_n) \Rightarrow \tau \in \mathcal{F}$ and $s_1 : \sigma_1, \ldots, s_n : \sigma_n$

Note that a function symbol $f : (\sigma_1 \times \ldots \times \sigma_n) \Rightarrow \tau$ takes exactly n arguments, and τ is not necessarily a base type. λ binds occurrences of variables as in the λ-calculus. Terms are considered modulo α-conversion; bound variables are renamed if necessary. The set of variables of s which are not bound is denoted $FV(s)$. Application is left-associative.

A *substitution* $[\vec{x} := \vec{s}]$, with $\vec{x}$ and $\vec{s}$ non-empty finite vectors of equal length, is the homomorphic extension of the type-preserving mapping $\vec{x} \mapsto \vec{s}$ from variables to terms. Substitutions are denoted γ, δ, and the result of applying γ to a term s is denoted $s\gamma$. The *domain* $\mathsf{dom}(\gamma)$ of $\gamma = [\vec{x} := \vec{s}]$ is $\{\vec{x}\}$. Substituting does not capture free variables.

We assume a fresh symbol $\Box_\sigma : \sigma$ for every type σ. A *context* $C[]$ is a term with a single occurrence of some $\Box_\sigma$. The result of replacing $\Box_\sigma$ in $C[]$ by a term s of type σ is denoted

$C[s]$. Free variables may be captured; if $C[] = \lambda x.\, \square_\sigma$ then $C[x] = \lambda x.\, x$. If $s = C[t]$ we say t is a *subterm* of s, notation $s \trianglerighteq t$, or $s \triangleright t$ (strict subterm) if $C[]$ is not the empty context $\square$.

Rules and Rewriting A *rewrite rule* is a pair of terms $l \to r$ such that l and r are terms of the same type and do not contain a subterm of the form $(\lambda x.\, s) \cdot t$, all free variables of r also occur in l, and l has the form $f(l_1, \ldots, l_n) \cdot l_{n+1} \cdots l_m$ (with $m \geq n \geq 0$). Given a set of rules $\mathcal{R}$, the *rewrite* or *reduction relation* $\to_\mathcal{R}$ on terms is given by the following clauses:

`(rule)`	$C[l\gamma]$	$\to_\mathcal{R}$	$C[r\gamma]$	with $l \to r \in \mathcal{R}$, C a context, γ a substitution
`(beta)`	$C[(\lambda x.\, s) \cdot t]$	$\to_\mathcal{R}$	$C[s[x := t]]$	

We sometimes use the notation $s \to_\beta t$ for a rewrite step using `(beta)`. An *algebraic functional system* (AFS) is the combination of a set of terms and a rewrite relation on this set, and is usually specified by a set of rules (perhaps with function symbols). A function symbol f is a *defined symbol* of an AFS if there is a rule with left-hand side $f(l_1, \ldots, l_n) \cdot l_{n+1} \cdots l_m$. A function symbol that is not a defined symbol is a *constructor symbol*. The sets of defined or constructor symbols are denoted by $\mathcal{D}$ or $\mathcal{C}$ respectively. A rewrite rule $l \to r$ is *left-linear* if every free variable occurs at most once in l; an AFS is left-linear if all its rewrite rules are.

▶ **Example 3.1.** Throughout this paper, we will consider as an example the AFS twice. It has four function symbols, $\mathsf{o} : \mathsf{nat}$, $\mathsf{s} : (\mathsf{nat}) \Rightarrow \mathsf{nat}$, $\mathsf{I} : (\mathsf{nat}) \Rightarrow \mathsf{nat}$, $\mathsf{twice} : (\mathsf{nat} \Rightarrow \mathsf{nat}) \Rightarrow \mathsf{nat} \Rightarrow \mathsf{nat}$, and three rewrite rules:

$$\begin{array}{rcl@{\qquad\qquad}rcl} \mathsf{I}(\mathsf{o}) & \to & \mathsf{o} & \mathsf{twice}(F) & \to & \lambda y.\, F \cdot (F \cdot y) \\ \mathsf{I}(\mathsf{s}(n)) & \to & \mathsf{s}(\mathsf{twice}(\lambda x.\, \mathsf{I}(x)) \cdot n) & & & \end{array}$$

An example reduction: $\mathsf{I}(\mathsf{s}(\mathsf{o})) \to \mathsf{s}(\mathsf{twice}(\lambda x.\, \mathsf{I}(x)) \cdot \mathsf{o}) \to \mathsf{s}((\lambda y.\, (\lambda x.\, \mathsf{I}(x)) \cdot ((\lambda x.\, \mathsf{I}(x)) \cdot y)) \cdot \mathsf{o}) \to_\beta \mathsf{s}((\lambda x.\, \mathsf{I}(x)) \cdot ((\lambda x.\, \mathsf{I}(x)) \cdot \mathsf{o})) \to_\beta \mathsf{s}((\lambda x.\, \mathsf{I}(x)) \cdot \mathsf{I}(\mathsf{o})) \to \mathsf{s}((\lambda x.\, \mathsf{I}(x)) \cdot \mathsf{o}) \to_\beta \mathsf{s}(\mathsf{I}(\mathsf{o})) \to \mathsf{s}(\mathsf{o})$.

The symbol I represents the identity function, and therefore no infinite reduction exists. However, this is not trivial to prove; neither orderings like HORPO [11] nor a static dependency pair approach can handle the second rule, due to the subterm $\mathsf{I}(x)$. The static approach gives a requirement $\mathsf{I}^\#(\mathsf{s}(n)) > \mathsf{I}^\#(x)$, where the right-hand side contains a variable which does not occur in the left-hand side. Since $>$ must be closed under substitution, this is impossible to satisfy, as $\mathsf{s}(n)$ might be substituted for x. Applying HORPO leads to a similar problem.

4 Dependency Pairs

An intuition behind the dependency pair approach is to identify those parts of the right-hand sides of rewrite rules which may give rise to an infinite reduction. These are subterms headed by a defined symbol (as in first order term rewriting), and also subterms headed by a free variable, because such a variable can be instantiated by a defined symbol or abstraction. The latter is typical for the *dynamic* approach to higher order dependency pairs.

In this section we will extend the concepts of dependency pairs and dependency chains to AFSs. We show that an AFS is terminating if it does not have an infinite dependency chain, and that absence of dependency chains characterizes termination for left-linear AFSs.

Completed Rules An AFS is *completed* by adding for each rule of the form $l \to \lambda x_1 \ldots x_n.\, r$ with $n > 0$ and r not an abstraction the n new rules $l \cdot x_1 \to \lambda x_2 \ldots x_n.\, r, \ldots, l \cdot x_1 \cdots x_n \to r$. We do this to avoid creating dependency pairs containing a β-redex. Completing does not affect termination. For example, the system twice is completed by adding $\mathsf{twice}(F) \cdot m \to F \cdot (F \cdot m)$. In the remainder of the paper, we work with completed AFSs.

Candidate terms The definition of a dependency pair uses the notion of candidate terms, intuitively those subterms which might cause non-termination. Subterms that cannot be reduced at the root are omitted, because they are not a minimal starting point of an infinite reduction. Bound variables that become free by taking a subterm are replaced by fresh constants. We denote by $\mathbb{C}$ the set consisting of infinitely many fresh symbols c_x with c_x the same type as x, where x in c_x is not bound and is not subject to α-conversion.

▶ **Definition 4.1.** We say $t[x_1 := c_{x_1}, \dots, x_n := c_{x_n}]$ is a *candidate term* of s if $s \trianglerighteq t$, and $x_1, \dots, x_n$ are the variables which occur bound in s but free in t, and either $t = f(t_1, \dots, t_n) \cdot t_{n+1} \cdots t_m$ with f a defined symbol and $m \geq n \geq 0$, or $t = x \cdot t_1 \cdots t_n$ with x free in s and $n > 0$. We denote the set of candidate terms of s by $Cand(s)$.

In the AFS twice we have $Cand(F \cdot (F \cdot m)) = \{F \cdot (F \cdot m),\ F \cdot m\}$ and $Cand(\mathsf{s}(\mathsf{twice}(\lambda x.\, \mathsf{I}(x)) \cdot n)) = \{\mathsf{twice}(\lambda x.\, \mathsf{I}(x)) \cdot n, \mathsf{twice}(\lambda x.\, \mathsf{I}(x)),\ \mathsf{I}(c_x)\}$. Note that for example $x \cdot y$ is not a candidate term of $g(\lambda x.\, x \cdot y)$ because x occurs only bound.

Dependency Pairs The definition of dependency pair also uses marked function symbols as in the first order case. Let $\mathcal{F}^\# = \mathcal{F} \cup \{f^\# : \sigma \mid f : \sigma \in \mathcal{D}\}$, so $\mathcal{F}$ extended with a marked version for every defined symbol, having the same type declaration. The marked counterpart of a term s, notation $s^\#$, is $f^\#(s_1, \dots, s_n)$ if $s = f(s_1, \dots, s_n)$ with f in $\mathcal{D}$, and just s otherwise. For example, $(\mathsf{twice}(F))^\# = \mathsf{twice}^\#(F)$ and $(\mathsf{twice}(F) \cdot m)^\# = \mathsf{twice}(F) \cdot m$.

▶ **Definition 4.2** (Dependency Pair)**.** The set of *dependency pairs* of a rewrite rule $l \to r$, notation $\mathsf{DP}(l \to r)$, consists of:

- all pairs $l^\# \rightsquigarrow p^\#$ with $p \in Cand(r)$,
- all pairs $l \cdot y_1 \cdots y_k \rightsquigarrow r \cdot y_1 \cdots y_k$ with $1 \leq k \leq n$ if $r : \sigma_1 \Rightarrow \ldots \Rightarrow \sigma_n \Rightarrow \iota$, and either $r = x \cdot r_1 \cdots r_m$ with $m \geq 0$ or $r = f(r_1, \dots, r_i) \cdot r_{i+1} \cdots r_m$ with $m \geq i \geq 0$ and $f \in \mathcal{D}$.

We use $\mathsf{DP}(\mathcal{R})$ (or just DP) for the set of all dependency pairs of rewrite rules of an AFS $\mathcal{R}$.

▶ **Example 4.3.** The set of dependency pairs of the AFS twice consists of:

$$\begin{array}{lcllcl}
\mathsf{I}^\#(\mathsf{s}(n)) & \rightsquigarrow & \mathsf{twice}(\lambda x.\, \mathsf{I}(x)) \cdot n & \mathsf{twice}^\#(F) & \rightsquigarrow & F \cdot (F \cdot c_y) \\
\mathsf{I}^\#(\mathsf{s}(n)) & \rightsquigarrow & \mathsf{twice}^\#(\lambda x.\, \mathsf{I}(x)) & \mathsf{twice}^\#(F) & \rightsquigarrow & F \cdot c_y \\
\mathsf{I}^\#(\mathsf{s}(n)) & \rightsquigarrow & \mathsf{I}^\#(c_x) & \mathsf{twice}(F) \cdot m & \rightsquigarrow & F \cdot (F \cdot m) \\
 & & & \mathsf{twice}(F) \cdot m & \rightsquigarrow & F \cdot m
\end{array}$$

The last two dependency pairs originate from the rule added by completion.

To illustrate the second form of dependency pair, consider the system with function symbols $\mathsf{app} : (o) \Rightarrow o \Rightarrow o$ and $\mathsf{abs} : (o \Rightarrow o) \Rightarrow o$, and one rewrite rule: $\mathsf{app}(\mathsf{abs}(x)) \to x$. This system has no dependency pairs of the first form, but does admit a two-step loop: $s := \mathsf{app}(\mathsf{abs}(\lambda x.\, \mathsf{app}(x) \cdot x)) \cdot \mathsf{abs}(\lambda x.\, \mathsf{app}(x) \cdot x) \to (\lambda x.\, \mathsf{app}(x) \cdot x) \cdot \mathsf{abs}(\lambda x.\, \mathsf{app}(x) \cdot x) \to_\beta s$.

Comparing our approach to static dependency pairs as defined in [17], the two main differences are that we avoid bound variables becoming free, and that we include dependency pairs where the right-hand side is headed by a variable. We call such pairs *collapsing*.

Dependency Chains We can now investigate termination by means of *dependency chains*:

▶ **Definition 4.4.** A *dependency chain* is a sequence $[(\rho_i, s_i, t_i) \mid i \in \mathbb{N}]$ such that for all i:

1. $\rho_i \in \mathsf{DP} \cup \{\mathtt{beta}\}$,
2. if $\rho_i = l_i \rightsquigarrow p_i \in \mathsf{DP}$ there exists γ with domain $FV(l_i)$ such that $s_i = l_i\gamma$ and $t_i = p_i\gamma$

3. if $\rho_i = \mathtt{beta}$ then $s_i = (\lambda x.\, u) \cdot v \cdot w_1 \cdots w_k$ and either
   a. $k > 0$ and $t_i = u[x := v] \cdot w_1 \cdots w_k$, or
   b. $k = 0$ and there is w such that $u \trianglerighteq w$ and $x \in FV(w)$ and $w^\#[x := v] = t_i$, but $w \neq x$
4. $t_i \to_{in}^* s_{i+1}$

A step $\to_{in}$ is obtained by rewriting some s_i inside a term of the form $f(s_1, \ldots, s_n) \cdot s_{n+1} \cdots s_m$.

▶ **Theorem 4.5.** *If $\mathcal{R}$ is non-terminating there is an infinite dependency chain over $\mathsf{DP}(\mathcal{R})$.*

Proof Sketch. Say a term s is *minimally non-terminating* (MNT) if s is terminating but all its subterms are not. Let u_{-1} be any MNT term, and subsequently for every $i \in \mathbb{N}$, given an MNT term u_{i-1}, we define $\rho_i \in \mathsf{DP} \cup \{\mathtt{beta}\}$ and terms s_i and t_i. Note (**): *if an MNT term is reduced at any other position than the top, the result is also MNT, or terminating.*

If $u_{i-1} = (\lambda x.\, s) \cdot t$ then $s[x := t]$ is also non-terminating (because eventually a topmost step must be done, and we can see that $s[x := t]$ reduces to the result); let u_i be an MNT subterm of $s[x := t]$ and define $\rho_i, s_i, t_i := \mathtt{beta}, u_{i-1}, u_i^\#$. If $u_{i-1} = (\lambda x.\, s) \cdot t \cdot v_0 \cdots v_k$ then by (**) $u_i := s[x := t] \cdot v_0 \cdots v_k$ is also MNT, so choose $\rho_i, s_i, t_i := \mathtt{beta}, u_{i-1}, u_i$. Otherwise $u_{i-1} = f(v_1, \ldots, v_n) \cdot v_{n+1} \cdots v_m$; then $u_{i-1} \to_{in}^*$ some term $l\gamma \cdot w_1 \cdots w_k$, with $r\gamma \cdot \vec{w}$ still non-terminating. If $k > 0$ then by (**) $r\gamma \cdot \vec{w}$ is MNT, so choose $u_i := r\gamma \cdot \vec{w}$ and $\rho_i, s_i, t_i := l \cdot x_1 \cdots x_k \rightsquigarrow r \cdot x_1 \cdots x_k,\ l\gamma \cdot \vec{w}, r\gamma \cdot \vec{w}$. Otherwise let r' be the smallest subterm of r such that $p := r'\delta$ is still non-terminating, where δ replaces the newly free variables x_i by c_{x_i}. Then some analysis shows that p is a candidate of r and $p\gamma$ is also MNT; choose $u_i := p\gamma$ and $\rho_i, s_i, t_i := l^\# \rightsquigarrow p^\#, l^\#\gamma, p^\#\gamma$. This process generates a dependency chain. ◀

The converse of Theorem 4.5 does not hold. Consider the AFS with rules:

$$f(x, y, \mathsf{s}(z)) \to g(h(x, y), \lambda u.\, f(u, x, z)) \quad \text{and} \quad h(x, x) \to f(x, \mathsf{s}(x), \mathsf{s}(\mathsf{s}(x)))$$

This system has the following dependency pairs:

$$\begin{array}{lcllcl} f^\#(x, y, \mathsf{s}(z)) & \rightsquigarrow & h^\#(x, y) & h^\#(x, x) & \rightsquigarrow & f^\#(x, \mathsf{s}(x), \mathsf{s}(\mathsf{s}(x))) \\ f^\#(x, y, \mathsf{s}(z)) & \rightsquigarrow & f^\#(c_u, x, z) & & & \end{array}$$

There is an infinite dependency chain: $f^\#(c_u, \mathsf{s}(c_u), \mathsf{s}(\mathsf{s}(c_u))) \rightsquigarrow f^\#(c_u, c_u, \mathsf{s}(c_u)) \rightsquigarrow h^\#(c_u, c_u) \rightsquigarrow f^\#(c_u, \mathsf{s}(c_u), \mathsf{s}(\mathsf{s}(c_u))) \rightsquigarrow \ldots$ However, the AFS is terminating, intuitively because the bound variable destroys matching possibilities. The crucial point of the example is the combination of bound variables and non-left-linear rules. Theorem 4.6 shows that for left-linear AFSs, the absence of infinite dependency chains actually characterizes termination.

▶ **Theorem 4.6.** *A left-linear AFS $\mathcal{R}$ is terminating if and only if it does not admit an infinite dependency chain.*

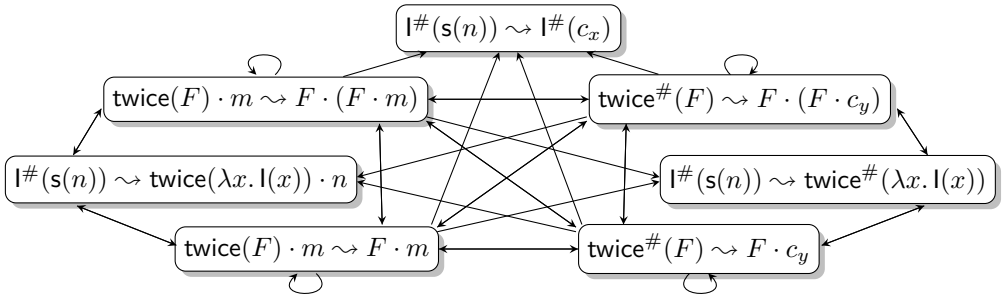
Proof Sketch. In a left-linear system replacing variables by a function symbol that doesn't occur in any rule has no effect on applicability of $\to_{\mathcal{R}}$. Thus a dependency chain effectively produces an infinite reduction $|s_i| \to_{\mathcal{R}} \cdot \rhd |t_i| \to_{\mathcal{R}}^* |s_{i+1}|$ (where $|\cdot|$ replaces any $f^\#$ by its unmarked counterpart), and this implies the existence of an infinite $\to_{\mathcal{R}}$ reduction. ◀

5 The Dependency Graph

As in the first order case, we use a dependency graph to organize the dependency pairs. The definition of a dependency graph is typical for our setting here, namely AFSs with dynamic dependency pairs, but the other notions we use are similar to the first order ones.

The *dependency graph* of an AFS $\mathcal{R}$ has the dependency pairs of $\mathcal{R}$ as nodes, and an edge from node $l \rightsquigarrow p$ to node $l' \rightsquigarrow p'$ if there is a finite dependency chain $[(l \rightsquigarrow p, s_1, t_1), (\texttt{beta}, s_2, t_2), \ldots, (\texttt{beta}, s_{k-1}, t_{k-1}), (l' \rightsquigarrow p', s_k, t_k)]$ with all but the first and the last elements `beta`.

▶ **Example 5.1.** The dependency graph of the AFS twice:



A *cycle* is a non-empty set $\mathcal{C}$ of dependency pairs such that between every two pairs $\rho, \pi \in \mathcal{C}$ there is a non-empty path in the graph using only nodes in $\mathcal{C}$. A cycle that is not contained in any other cycle is called a *strongly connected component* (SCC). To prove termination we must show that cycles in a dependency graph are in some sense well-behaved (see Theorem 6.2). Due to clause 3b in Definition 4.4, there is an edge from any node of the form $l \rightsquigarrow x \cdot r_1 \ldots \cdot r_n$ with x a variable to all other nodes. Hence a rule with a functional variable in its right-hand side gives rise to many cycles. Here, exactly, lies the appeal of the static approach, which eliminates the need for such pairs. However, this barrier is not impossible to overcome, and as discussed, the dynamic approach can deal with systems where the static approach fails.

A set $D \subseteq \mathsf{DP}$ is *looping* if there is an infinite dependency chain using only dependency pairs from D and `beta`. By termination of simply typed β-reduction, $\emptyset$ is not looping.

Because the dependency graph cannot be computed in general, one uses *approximations* of the dependency graph, which have the same nodes but possibly more edges. A brute method to find an approximation of the dependency graph is to have an edge between $l \rightsquigarrow p$ and $l' \rightsquigarrow p'$ as soon as the head of p is a variable, or if p and l' both have the form $f(s_1, \ldots, s_n) \cdot s_{n+1} \cdots s_m$ for some function symbol f and some $m \geq n \geq 0$. It is interesting to study more sophisticated methods to find approximations, but this is left for future work.

In the remainder of this paper, we will assume that dependency graphs (and hence also their approximations) have only finitely many nodes. This is the case if the AFS under consideration has finitely many rewrite rules. However, note that also for infinite AFSs (arising for example by instantiation of polymorphic rewrite rules) we can work with finite dependency graphs, if (infinite) sets of dependency pairs are represented by a single node.

▶ **Lemma 5.2.** *Let G be an approximation of the dependency graph of an AFS $\mathcal{R}$. Suppose that every cycle in G is non-looping. Then $\mathcal{R}$ is terminating.*

Proof Sketch. Given an infinite dependency chain, there must be a dependency pair ρ_i which occurs infinitely often (by the finiteness assumption). Then $\{\rho_j \mid j > i\}$ is a cycle. ◀

▶ **Example 5.3.** The dependency graph (approximation) of twice from Example 5.1 admits many cycles, such as $\{\mathsf{twice}(F) \cdot n \rightsquigarrow F \cdot (F \cdot n)\}$ or the following cycle $\mathcal{C}_{\mathsf{twice}}$:

$$\left\{ \begin{array}{rcl@{\qquad}rcl} \mathsf{I}^\#(\mathsf{s}(n)) & \rightsquigarrow & \mathsf{twice}(\lambda x.\mathsf{I}(x)) \cdot n & \mathsf{twice}^\#(F) & \rightsquigarrow & F \cdot (F \cdot c_y) \\ \mathsf{I}^\#(\mathsf{s}(n)) & \rightsquigarrow & \mathsf{twice}^\#(\lambda x.\mathsf{I}(x)) & \mathsf{twice}^\#(F) & \rightsquigarrow & F \cdot c_y \\ \mathsf{twice}(F) \cdot m & \rightsquigarrow & F \cdot (F \cdot m) & \mathsf{twice}(F) \cdot m & \rightsquigarrow & F \cdot m \end{array} \right\}$$

$\mathcal{C}_{\mathsf{twice}}$ is an SCC and includes all cycles. Therefore twice is terminating if $\mathcal{C}_{\mathsf{twice}}$ is non-looping.

6 Reduction Orders

The challenge, then, is to prove the absence of looping cycles. We use the following definition:

▶ **Definition 6.1.** A *reduction triple* consists of a well-founded ordering $>$, a quasi-ordering $\geq$ and a sub-relation $\geq_1$ of $\geq$, such that:

1. $>$ and $\geq$ are *compatible*: either $> \cdot \geq \subseteq >$ or $\geq \cdot > \subseteq >$;
2. $>$, $\geq$ and $\geq_1$ are all *stable* (that is, closed under substitution);
3. $\geq_1$ is *monotonic*: (that is, if $s \geq_1 t$ with s, t sharing a type, then $C[s] \geq_1 C[t]$);
4. $\geq_1$ contains `beta` (that is, always $(\lambda x.\, s) \cdot t \geq_1 s[x := t]$).

A *reduction pair* is a pair $(>, \geq)$ such that $(>, \geq, \geq)$ is a reduction triple; this corresponds with the original (first order) notion of reduction pair. The reduction triple is a generalisation of this notion, where $\geq$ itself is not required to be monotonic; we will need a non-monotonic $\geq$ in Section 6.1 to compare terms with different types. To deal with subterm reduction in dependency chains, an additional definition is needed. We say $\geq$ has the *limited subterm property* if: for all x, s, t, u such that $s \trianglerighteq u$ and u is neither an abstraction nor a single variable, there is a substitution γ such that $(\lambda x.\, s) \cdot t \geq (u^{\#})\gamma[x := t]$. Intuitively, the substitution γ is used to replace free variables in u that are bound in s by fresh constants c_x. However, we will also use a more liberal replacement of those variables, hence the general γ.

The following theorem shows how reduction triples can be used with dependency pairs.

▶ **Theorem 6.2.** *A set $D = D_1 \uplus D_2$ of dependency pairs is non-looping if D_2 is non-looping, and there is a reduction triple $(>, \geq, \geq_1)$ such that*

- *$l > p$ for all $l \rightsquigarrow p \in D_1$,*
- *$l \geq p$ for all $l \rightsquigarrow p \in D_2$,*
- *$l \geq_1 r$ for all $l \to r \in \mathcal{R}$,*
- *either D is non-collapsing or $\geq$ satisfies the limited subterm property.*

Proof Sketch. If D is looping it has an infinite chain which (as D_2 is non-looping) contains infinitely many pairs in D_1. If D is non-collapsing we can find such a chain without `beta` steps, and have $s_i \geq t_i \geq s_{i+1}$ for all i, and if $\rho_i \in D_1$ even $s_i > t_i$, contradicting well-foundedness of $>$. If D is collapsing then let $[(\rho_i, s_i, t_i) | i \in \mathbb{N} | i \geq j]$ be an infinite dependency chain over D; if $\rho_j \in D_1$ then $s_j > t_j \geq s_{j+1}$, if $\rho_j \in D_2$ then $s_j \geq t_j \geq s_{j+1}$ and if $\rho_j =$ `beta` then there is some substitution δ such that $s_j \geq t_j\delta \geq s_{j+1}\delta$. Since $[(\rho_i, s_i\delta, t_i\delta) | i \in \mathbb{N} | i \geq j+1]$ is also a dependency chain we can continue this reasoning recursively, obtaining a decreasing $\geq$ sequence with infinitely many $>$ steps, contradicting well-foundedness. ◀

Theorem 6.2 can be used to prove that every cycle in the dependency graph approximation of an AFS is non-looping; termination follows with Lemma 5.2. See also Section 9 for an algorithm. For left-linear AFSs, we even have a characterization of termination.

▶ **Theorem 6.3.** *A left-linear AFS with dependency graph approximation G is terminating if and only if for every cycle in G the requirements of Theorem 6.2 are satisfied.*

▶ **Example 6.4.** Termination of twice is proved if there is a reduction triple $(>, \geq, \geq_1)$ with the limited subterm property, such that $l \geq_1 r$ for all rules, and $l > p$ for every dependency pair in $\mathcal{C}_{\mathsf{twice}}$ from Example 5.3 (choosing $D_2 = \emptyset$, which is non-looping).

6.1 Type Changing

The situation so far is not completely satisfactory, because both $>$ and $\geq$ may have to compare terms of different types. Consider for example the dependency pair $\mathsf{twice}^\#(F) \rightsquigarrow F \cdot c_y$ from twice where the two sides have a different type. Moreover, the comparison in the definition of limited subterm property may concern terms of different types. This is problematic because term orderings do not usually compare terms of arbitrary different types; neither any version of the higher order path ordering [11, 5] nor monotone algebras [21] are equipped for this.

A solution is to manipulate the ordering requirements. Let $(\succ, \succeq)$ be a reduction pair (so a pair such that $(\succ, \succeq, \succeq)$ is a reduction triple). Define $>$, $\geq$, and $\geq_1$ as follows:

- $s > t$ if there are fresh variables $x_1, \ldots, x_n$ and terms $u_1, \ldots, u_m$ such that $s \cdot x_1 \cdots x_n \succ t \cdot u_1 \cdots u_m$ and both sides have some base type;
- $s \geq t$ if there are fresh variables $x_1, \ldots, x_n$ and terms $u_1, \ldots, u_m$ such that $s \cdot x_1 \cdots x_n \; R \; t \cdot u_1 \cdots u_m$ and both sides have some base type, where R is $\succeq \cup \succ \cdot \succeq \cup \succeq \cdot \succ$;
- $s \geq_1 t$ if $s \succeq t$ and s, t have the same type.

▶ **Lemma 6.5.** *$(>, \geq, \geq_1)$ as generated from a reduction pair $(\succ, \succeq)$ is a reduction triple.*

Proof. This is easy, noting: (1) if $s \geq_1 t$ then by monotonicity $s\vec{x} \succeq t\vec{x}$, (2) if $s > t$ then for any $\vec{u}$ there are $\vec{v}$ such that $s \cdot \vec{u} \succ t \cdot \vec{v}$ (by stability of $\succ$), (3) similar for $\geq$. ◀

The relations $>$ and $\geq$ are not necessarily computable, but we will not need to work with them directly. To prove some set of dependency pairs D non-looping, we can choose for every pair $l \rightsquigarrow p \in D$ a corresponding base-type pair $\overline{l} \rightsquigarrow \overline{p}$, and prove either $\overline{l} \succ \overline{p}$ or $\overline{l} \succeq \overline{p}$. For example, we could assign $\overline{l} := l \cdot x_1 \cdots x_n$ and $\overline{p} := p \cdot c_{y_1} \cdots c_{y_m}$, where the c_{y_i} are chosen arbitrarily. This is the choice we will use in examples in this paper. Other choices for $\overline{p}$, for instance made in such a way as to duplicate existing requirements, are also possible.

To make sure that $\geq$ satisfies the limited subterm property, we consider a base-type version of subterm reduction, which is strongly related to β-reduction.

▶ **Definition 6.6.** $\rhd^!$ is the relation on base-type terms (and $\trianglerighteq^!$ its reflexive closure) generated by the following clauses:

- $(\lambda x.\, s) \cdot t_0 \cdots t_n \rhd^! u$ if $s[x := t_0] \cdot t_1 \cdots t_n \trianglerighteq^! u$
- $f(s_1, \ldots, s_m) \cdot t_1 \cdots t_n \trianglerighteq^! u$ if $s_i \cdot \vec{c} \trianglerighteq^! u$
- $s \cdot t_1 \cdots t_n \trianglerighteq^! u$ if $t_i \cdot \vec{c} \trianglerighteq^! u$ (s may have any form)

Here, $s \cdot \vec{c}$ is a term s applied to constants c_y of the right type. Note that if $s \rhd t$ and s has base type, there are terms $u_1, \ldots, u_n$ and substitution γ such that $s \rhd^! t\gamma \cdot u_1 \cdots u_n$. Consequently, $\geq$ satisfies the limited subterm property if $\succ \cup \succeq$ contains $\rhd^!$ and $f(\vec{x}) \succeq f^\#(\vec{x})$ for all $f \in \mathcal{D}$ (the *marking property*). We can derive the following theorem.

▶ **Theorem 6.7.** *A set of dependency pairs $D = D_1 \uplus D_2$ is non-looping if D_2 is non-looping and there is a reduction pair $(\succ, \succeq)$ such that:*

1. *$\overline{l} \succ \overline{p}$ for all $l \rightsquigarrow p \in D_1$;*
2. *$\overline{l} \succeq \overline{p}$ for all $l \rightsquigarrow p \in D_2$;*
3. *$l \succeq r$ for all $l \to r \in \mathcal{R}$;*
4. *if D is collapsing, then $\succ \cup \succeq$ contains $\rhd^!$, and $f(\vec{x}) \succeq f^\#(\vec{x})$ for all $f \in \mathcal{D}$.*

▶ **Example 6.8.** To prove that $\mathcal{C}_{\mathsf{twice}}$ is non-looping it suffices to find a reduction pair $(\succ, \succeq)$ such that $l \succeq r$ for all rules, $\succeq$ satisfies the subterm and marking properties, and furthermore:

$$\begin{array}{rcl@{\qquad}rcl}
\mathsf{I}^\#(\mathsf{s}(n)) & \succ & \mathsf{twice}(\lambda x.\, \mathsf{I}(x)) \cdot n & \mathsf{twice}^\#(F) \cdot x & \succ & F \cdot (F \cdot c_y) \\
\mathsf{I}^\#(\mathsf{s}(n)) & \succ & \mathsf{twice}^\#(\lambda x.\, \mathsf{I}(x)) \cdot c_z & \mathsf{twice}^\#(F) \cdot x & \succ & F \cdot c_y \\
\mathsf{twice}(F) \cdot m & \succ & F \cdot (F \cdot m) & \mathsf{twice}(F) \cdot m & \succ & F \cdot m
\end{array}$$

This completes the basis of dynamic dependency pairs for AFSs. But is this approach any easier than proving $l > r$ for all rewrite rules? Unless the dependency graph has no cycles we still have to prove $l \geq r$ for all rules and with an ordering like HORPO [11] this is barely an improvement. In Section 7 we will therefore discuss a variation of usable rules, which allows us to drop a number of ordering requirements. In Section 8 we will define a variation of the monotone algebra approach that is especially suited to dependency pairs.

7 Formative Rules

In the first order setting, the result corresponding with Theorem 6.2 is optimized: it is sufficient to consider for a cycle only its *usable rules* instead of all rules. The definition of usable rules cannot easily be extended to our setting, because we admit collapsing dependency pairs. Therefore we take a different approach with the same goal of restricting attention to rules which are in some way relevant to a set of dependency pairs. Where usable rules are defined from the right-hand sides of dependency pairs, our *formative rules* are based on the left-hand sides. We will use the notion of *simple terms*:

▶ **Definition 7.1.** A term s is *simple* if:

- it is linear,
- it has no subterm of the form $x \cdot s_1 \cdots s_n$ with $n > 0$ and x a free variable,
- there is no occurrence of a free variable below an abstraction.

Many examples of AFSs, such as rules from functional programming, have a simple left-hand side. The intuition behind formative rules is that, for rewrite rules with a simple left-hand side, only the formative rules can contribute to the creation of its pattern.

▶ **Definition 7.2.** For β-normal terms s, let $Symb(s)$ be recursively defined as follows:

$$\begin{array}{rcl} Symb(\lambda x.\, s : \sigma) & = & \{\langle ABS, \sigma\rangle\} \cup Symb(s) \\ Symb(f(s_1, \ldots, s_n) \cdot s_{n+1} \cdots s_m : \sigma) & = & \{\langle f, \sigma\rangle\} \cup Symb(s_1) \cup \ldots \cup Symb(s_m) \\ Symb(x \cdot s_1 \cdots s_n : \sigma) & = & \{\langle VAR, \sigma\rangle\} \cup Symb(s_1) \cup \ldots \cup Symb(s_n) \ (n > 0) \\ Symb(x) & = & \emptyset \end{array}$$

The formative symbols and rules of any term are defined by a (possibly) infinite process:

- the starting point: $FS_0(s) = Symb(s)$
- for all $n \geq 0$, the set $FR_n(s)$ consists of rules $l \cdot x_1 \cdots x_k \to r \cdot x_1 \cdots x_k$ if $l \to r \in \mathcal{R}$ and
  - $k = 0,\ r = \lambda x.\, r' : \sigma$ and $\langle ABS, \sigma\rangle \in FS_n(s)$, or
  - $r = f(\vec{u}) \cdot \vec{v} : \sigma_1 \Rightarrow \ldots \Rightarrow \sigma_k \Rightarrow \tau$ and $\langle f, \tau\rangle \in FS_n(s)$, or
  - $r = x \cdot \vec{v} : \sigma_1 \Rightarrow \ldots \Rightarrow \sigma_k \Rightarrow \tau$ and $\langle f, \tau\rangle \in FS_n(s)$ for some $f \in \mathcal{F} \cup \{ABS, VAR\}$; $|\vec{v}| \geq 0$
- $FS_{n+1}(s) = FS_n(s) \cup \bigcup_{l \to r \in FR_n(s)} Symb(l)$

Now $FR(s)$ is defined as the union of all $FR_n(s)$ (this is a finite union for finite AFSs) in the case that both s is simple and all rules in this union have a simple left-hand side. Otherwise, $FR(s) = \mathcal{R}$. The set of formative rules of a dependency pair, $FR(f(l_1, \ldots, l_n) \cdot l_{n+1} \cdots l_m \rightsquigarrow p)$, is defined as $\bigcup_{1 \leq i \leq m} FR(l_i)$. For a set D of dependency pairs, $FR(D) = \bigcup_{l \rightsquigarrow p \in D} FR(l \rightsquigarrow p)$.

Note that $FR_n(s)$ and $FS_{n+1}(s)$ can easily be calculated (automatically) from $FS_n(s)$; to compute $FR(s)$ a tool would simply repeat this process until either a rule with a non-simple left-hand-side is included (in which case $FR(s) = \mathcal{R}$), or until no new symbols are added.

▶ **Example 7.3.** Recall the rules for the (completed) system twice:

$$\begin{array}{llcl@{\qquad}llcl} (A) & \mathsf{I}(\mathsf{o}) & \to & \mathsf{o} & (C) & \mathsf{twice}(F) & \to & \lambda y.\, F \cdot (F \cdot y) \\ (B) & \mathsf{I}(\mathsf{s}(n)) & \to & \mathsf{s}(\mathsf{twice}(\lambda x.\, \mathsf{I}(x)) \cdot n) & (D) & \mathsf{twice}(F) \cdot m & \to & F \cdot (F \cdot m) \end{array}$$

In this context, let $l = \mathsf{s}(n)$. Then

$$\begin{array}{lcll@{\qquad}lcll} FS_0(l) & = & \{\langle \mathsf{s}, \mathsf{nat}\rangle\} & & FS_1(l) & = & \{\langle \mathsf{s}, \mathsf{nat}\rangle, \langle \mathsf{twice}, \mathsf{nat}\rangle, \langle \mathsf{I}, \mathsf{nat}\rangle\} & \\ FR_0(l) & = & \{(B),\ (D)\} & & FR_1(l) & = & \{(B),\ (D)\} = FR_0(l) & \end{array}$$

We have $FR(\mathsf{I}^{\#}(\mathsf{s}(n)) \rightsquigarrow p) = FR(\mathsf{s}(n)) = \{(B),\ (D)\}$ for any p. Note that for a dependency pair with left-hand side $\mathsf{twice}(F) \cdot n$ or $\mathsf{twice}^{\#}(F)$ the set of formative rules is empty (since $Symb(F) = Symb(n) = \emptyset$). Therefore, the formative rules of the SCC $\mathcal{C}_{\mathsf{twice}}$ are (B) and (D).

Using formative rules Formative rules are constructed in such a way that to reduce to a term of the form $l\gamma$ we only need its formative rules:

▶ **Lemma 7.4.** *If s is terminating and $s \to^*_{\mathcal{R}} l\gamma$, then there exists a substitution δ on the same domain as γ such that each $\delta(x) \to^*_{\mathcal{R}} \gamma(x)$ and $s \to^*_{FR(l)} l\delta$.*

Proof Sketch. We assume l is simple and not a variable (otherwise this is trivial). Transform the reduction $s \to^*_{\mathcal{R}} l\gamma$ into a reduction without any headmost steps with a rule $l' \to \lambda x.\, r'$ (this is possible because the rules have been completed). Then perform induction on s first, using $\to_{\mathcal{R}} \cup \rhd$, the length of the reduction second. If s is headed by a beta-redex we can start with a β-step because $l\gamma$ is not (and complete with IH1), if s reduces to $l\gamma$ without any headmost steps we use the $\rhd$ part of IH1 (variable capture is not an issue because γ can be assumed to have empty domain if l is an abstraction) and if $s \to^*_{\mathcal{R}} l'\gamma' \cdot t_1 \cdots t_n \to_{\mathcal{R}} r'\gamma' \cdot t_1 \cdots t_n \to^*_{\mathcal{R}'} l\gamma$ with either r' headed by a variable or the latter part not using any headmost steps, then $l'' := l' \cdot x_1 \cdots x_n \to r' \cdot x_1 \cdots x_n =: r''$ is a formative rule of l and can be assumed simple, so we use the second induction hypothesis to get $s \to^*_{FR(l'')} l''\delta' \to_{\mathcal{R}} r''\delta'$ and the first induction hypothesis to have $r''\delta' \to^*_{FR(l)} l\delta$; this suffices because $FR(l'') \subseteq FR(l)$. ◀

With this we can strengthen the definition of dependency chains, and adapt Theorem 4.5:

▶ **Lemma 7.5.** *If $\mathcal{R}$ is non-terminating, there is an infinite dependency chain over $\mathsf{DP}(\mathcal{R})$ such that for all i: $t_i \to^*_{in} s_{i+1}$ using only rules from $FR(l_{i+1})$.*

Thus, we can restrict attention to dependency chains using only formative rules, and adapt the definition of *looping* and the results of Sections 5 and 6 accordingly. We obtain:

▶ **Theorem 7.6** (Complete Result)**.** *A set of dependency pairs $D = D_1 \uplus D_2$ is non-looping if D_2 is non-looping and there is a reduction triple such that:*

1. *$l > p$ for $l \rightsquigarrow p \in D_1$,*
2. *$l \geq p$ for $l \rightsquigarrow p \in D_2$,*
3. *$l \geq_1 r$ for $l \to r \in FR(D)$,*
4. *If D is collapsing, then $\geq$ additionally satisfies the limited subterm property.*

Also $\emptyset$ is non-looping. An AFS with rules $\mathcal{R}$ and dependency graph approximation G is terminating if all cycles in G are non-looping, which holds if all SCCs are non-looping.

In requirement (3) in Theorem 6.7 we can also restrict attention to the formative rules of D instead of considering all rules. It remains to find a suitable reduction triple or pair.

8 Monotone Algebras

A semantical method to prove termination of rewriting is to interpret terms in a well-founded algebra, and show that whenever $s \to t$ their interpretations decrease: $[\![s]\!] > [\![t]\!]$. For TRSs, such an algebra is called a termination model if $[\![l]\!] > [\![r]\!]$ for all rules $l \to r$ and some additional properties guarantee that this implies $[\![C[l\gamma]]\!] > [\![C[r\gamma]]\!]$ for all contexts C and substitutions γ. A TRS is terminating if and only if it has a termination model [9, 26]. Van de Pol [21] generalizes this approach to HRSs, with higher order rewriting modulo $\alpha\beta\eta$, and shows that a HRS is terminating if it has a termination model; the converse does not hold.

Here we consider interpretations of AFS terms in a monotone algebra, and use the orderings to solve dependency pair constraints. Since $>$ does not have to be monotonic when using dependency pairs, the theory of [21] can be significantly simplified. We interpret all base types with the same algebra to avoid problems with comparing differently-typed terms.

▶ **Definition 8.1** (Weakly Monotonic Functionals)**.** Let $\mathcal{A}$ be an algebra with a well-founded partial order $>$ and minimum element 0. We assume there is a binary operator $\vee$ on $\mathcal{A}$ such that $x \vee y \geq x, y$ for all $x, y \in \mathcal{A}$ and $x \vee 0 = x$. Terms will be interpreted by elements of, and weakly monotonic functionals over, $\mathcal{A}$. Intuitively, a functional f is weakly monotonic if $f(x) \geq f(y)$ whenever $x \geq y$; however, f only needs to be defined on weakly monotonic input. We inductively define the weakly monotonic functionals for all types, and relations $\sqsupset_{wm}$ and $\sqsupseteq_{wm}$ on these functionals:

- the interpretation for base types: $\mathcal{WM}_\iota = \mathcal{A}$ for all $\iota \in \mathcal{B}$,
- the orderings on $\mathcal{WM}_\iota$ (with $\iota \in \mathcal{B}$): $\sqsupset_{wm}$ equals $>$, and $\sqsupseteq_{wm}$ is its reflexive closure,
- the interpretation for functional types: $\mathcal{WM}_{\sigma\Rightarrow\tau}$ consists of the functions mapping elements of $\mathcal{WM}_\sigma$ to elements of $\mathcal{WM}_\tau$, such that $\sqsupseteq_{wm}$ is preserved (that is, if $x \sqsupseteq_{wm} y$ in $\mathcal{WM}_\sigma$ then $f(x) \sqsupseteq_{wm} f(y)$ in $\mathcal{WM}_\tau$),
- the orderings on $\mathcal{WM}_{\sigma\Rightarrow\tau}$: we have $f \sqsupset_{wm} g$ iff $f(x) \sqsupset_{wm} g(x)$ for all $x \in \mathcal{WM}_\sigma$, and $f \sqsupseteq_{wm} g$ iff $f(x) \sqsupseteq_{wm} g(x)$ for all $x \in \mathcal{WM}_\sigma$.

$\sqsupset_{wm}$ and $\sqsupseteq_{wm}$ are an order and quasi-order respectively, and strongly compatible. If either $x \sqsupset_{wm} y$ or $x = y$ then $x \sqsupseteq_{wm} y$, but the converse implication does not hold.

Constant functions are weakly monotonic functionals: for $n \in \mathcal{A}$ and $\sigma = \tau_1 \Rightarrow \ldots \Rightarrow \tau_k \Rightarrow \iota$ (note that ι always refers to a base type), let $n_\sigma = \boldsymbol{\lambda} x_1 \ldots x_k.n$ (the function in $\mathcal{WM}_\sigma$ taking k arguments and returning n). The function $\boldsymbol{\lambda} f.f(\vec{0})$ is also in $\mathcal{WM}_{\sigma\Rightarrow\iota}$, where $f(\vec{0})$ is short for $f(0_{\tau_1}, \ldots, 0_{\tau_k})$. A weakly monotonic functional not defined in [21], but which will be needed to deal with term application, is *max*:

$$\begin{array}{rcll} \max_\iota(x,y) & = & x \vee y & (\text{for } x, y \in \mathcal{A}) \\ \max_{\sigma\Rightarrow\tau}(f,y) & = & \boldsymbol{\lambda} x.\, \max_\tau(f(x), y) & (\text{for } f \in \mathcal{WM}_{\sigma\Rightarrow\tau}, y \in \mathcal{A}) \end{array}$$

Using induction on the type of the first argument, it is easy to see that $\max_\sigma \in \mathcal{WM}_{\sigma\Rightarrow\iota\Rightarrow\sigma}$.

Term Interpretation. Using an interpretation $\mathcal{J}$ of function symbols, van de Pol associates to each closed term a weakly monotonic functional. Although the definition in [21] considers terms modulo $\alpha\beta\eta$, this is not a significant blockade because we can handle application as a function symbol. The following is our own adaptation of the translation in [21]:

▶ **Definition 8.2.** For all function symbols $f : (\sigma_1 \times \ldots \times \sigma_n) \Rightarrow \tau$ let $\mathcal{J}_f \in \mathcal{WM}_\sigma$, where σ is $\sigma_1 \Rightarrow \ldots \sigma_n \Rightarrow \tau$. A valuation is a function α with a finite domain of variables, such that $\alpha(x) \in \mathcal{WM}_\sigma$ for $x : \sigma$ in its domain. For any AFS-term s and valuation α whose domain contains all $x \in FV(s)$, let $[\![s]\!]_{\mathcal{J},\alpha}$ be the weakly monotonic functional defined as follows:

$$\begin{array}{lcl}
[\![x]\!]_{\mathcal{J},\alpha} & = & \alpha(x) \text{ if } x \in \mathcal{V} \\
[\![f(s_1,\ldots,s_n)]\!]_{\mathcal{J},\alpha} & = & \mathcal{J}_f([\![s_1]\!],\ldots,[\![s_n]\!]) \\
[\![\lambda x.\, s]\!]_{\mathcal{J},\alpha} & = & \boldsymbol{\lambda} n.[\![s]\!]_{\mathcal{J},\alpha\cup\{x\mapsto n\}} \text{ if } x \notin \mathsf{dom}(\alpha) \\
[\![s \cdot t]\!]_{\mathcal{J},\alpha} & = & \max([\![s]\!]_{\mathcal{J},\alpha}([\![t]\!]_{\mathcal{J},\alpha}), [\![t]\!]_{\mathcal{J},\alpha}(\vec{0}))
\end{array}$$

▶ **Example 8.3.** In our running example, consider an interpretation into the natural numbers Say $\mathcal{J}_{\mathsf{I}} = \boldsymbol{\lambda} n.n$ and $\mathcal{J}_{\mathsf{s}} = \boldsymbol{\lambda} n.n+1$. Then $[\![\mathsf{I}(\mathsf{s}(x))]\!]_{\mathcal{J},\alpha} = \alpha(x)+1$.

Reduction Pair Since this definition uses weak rather than strict monotonicity it cannot be used directly like in first order rewriting: $[\![l]\!] \sqsupset_{wm} [\![r]\!]$ does not in general imply $[\![C[l\gamma]]\!] \sqsupset_{wm} [\![C[r\gamma]]\!]$. This issue (which van de Pol works around by defininining an additional relation) disappears in the context of dependency pairs. Using Theorem 6.7 we obtain a number of requirements $[\![l]\!] \sqsupset_{wm} [\![r]\!]$ or $[\![l]\!] \sqsupseteq_{wm} [\![r]\!]$, and additionally, for collapsing D, the subterm and marking properties must be satisfied. The latter is a simple restriction, the former holds if the value of a function is always greater than or equal to the value of its arguments.

▶ **Theorem 8.4.** *Let $\mathcal{J}$ be a symbol interpretation such that:*

- *$\mathcal{J}_f \sqsupseteq_{wm} \mathcal{J}_{f^\#}$ for all $f \in \mathcal{D}$*
- *$\mathcal{J}$ maps each c_x to the appropriate 0_σ*
- *for all $f : (\sigma_1 \times \ldots \times \sigma_n) \Rightarrow \tau_1 \Rightarrow \ldots \Rightarrow \tau_m \Rightarrow \iota \in \mathcal{F}$, all $1 \leq i \leq n$ and all $n \in \mathcal{WM}_{\sigma_i}$: $\mathcal{J}_f(0_{\sigma_1},\ldots,n,\ldots,0_{\sigma_n},0_{\tau_1},\ldots,0_{\tau_m}) \sqsupseteq_{wm} n(\vec{0})$.*

Define $s \succ t$ if $[\![s]\!]_{\mathcal{J},\alpha} \sqsupset_{wm} [\![t]\!]_{\mathcal{J},\alpha}$ for all valuations α and $s \succeq t$ if $[\![s]\!]_{\mathcal{J},\alpha} \sqsupseteq_{wm} [\![t]\!]_{\mathcal{J},\alpha}$ for all valuations α. Then $(\succ,\succeq)$ is a reduction pair which satisfies the subterm and marking properties from Theorem 6.7.

Proof. Compatibility is evident, weak monotonicity holds by a simple case distinction and stability by the substitution Lemma [21, Theorem 3.2.1]. By the interpretation of application also $\to_\beta$ is contained in $\succeq$, and subterm reduction is included by an inductive argument which uses the last two requirements. The marking property is given by the first requirement. ◀

It is not immediately obvious how to use monotone algebras automatically; a lot will depend on the chosen interpretation for the function symbols. Common first order methods, like polynomial or matrix interpretations, are not likely to be succesful in the presence of functional variables. However, it is very likely that higher order parallels exist, such as an interpretation with primitive recursive functions. While a proper study of such methods is beyond the scope of this paper, the example below might give some initial ideas.

▶ **Example 8.5.** Suppose we have to satisfy a requirement $\mathsf{map}(F,\mathsf{cons}(x,y)) \succeq \mathsf{cons}(F \cdot x, \mathsf{map}(F,y))$, where $F : \mathsf{nat} \Rightarrow \mathsf{nat}$. We consider an interpretation in the natural numbers (with standard $>$, 0, and $\vee$ giving the highest of two numbers) using primitive recursive functions. Let $G(f,m,n)$ be the recursive function defined by: $G(f,m,0) = \max(f(m),m)$ and $G(f,m,n+1) = f(n+1, 2G(f,m,n))$. This function is weakly monotonic in each of f, m and n, and moreover $G(f,m,n+k) \geq G(f,m,n)+G(f,m,k)$ for all $n,k > 0$. Also $G(f,m,n) \geq m$, and $G(f,m,n) \geq f(0)$ if f is weakly monotonic. Choose $\mathcal{J}_{\mathsf{cons}} = \boldsymbol{\lambda} nm.n+m+1$ and $\mathcal{J}_{\mathsf{map}} = \boldsymbol{\lambda} fn.G(f,n,n+1)$, and let $\alpha = \{F \mapsto f, x \mapsto n, y \mapsto m\}$ be a valuation. Then:

$$\begin{array}{lcl}
[\![\mathsf{map}(F,\mathsf{cons}(x,y))]\!]_{\mathcal{J},\alpha} & = & G(f,n+m+1,n+m+2) \\
& \sqsupseteq_{wm} & G(f,n+m+1,n+1) + G(f,n+m+1,m+1) \\
& = & f(n+1) + 2G(f,n+m+1,n) + G(f,n+m+1,m+1) \\
& \sqsupseteq_{wm} & f(n) + 2(n+m+1) + G(f,m,m+1) \\
& \sqsupset_{wm} & \max(f(n),n) + G(f,m,m+1) + 1 \\
& = & [\![\mathsf{cons}(F \cdot x, \mathsf{map}(F,y))]\!]_{\mathcal{J},\alpha}
\end{array}$$

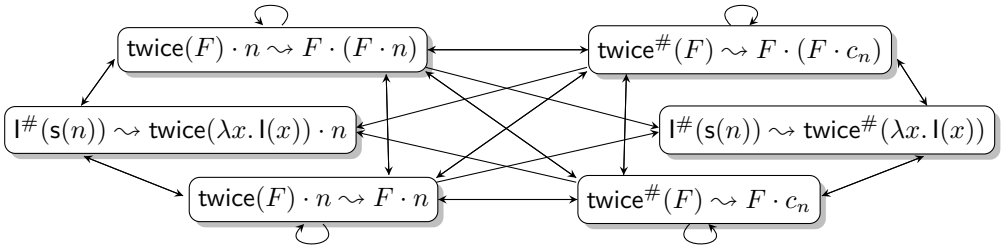
9 Conclusion

A Termination Algorithm The combination of Theorem 7.6 and Section 6.1 provides an algorithm to prove termination of an AFS. First calculate the system's dependency pairs and take an approximation of the (finite) dependency graph. Then:

1. remove all nodes from G which are not on a cycle;
2. if G is empty return `terminating`; otherwise find an SCC $\mathcal{C}$;
3. determine a partition in $\mathcal{C} = \mathcal{C}_1 \uplus \mathcal{C}_2$ and find a reduction pair $(\succ, \succeq)$ such that $\overline{l} \succ \overline{p}$ for $l \rightsquigarrow p \in \mathcal{C}_1$, $\overline{l} \succeq \overline{p}$ for $l \rightsquigarrow p \in \mathcal{C}_2$, $l \succeq r$ for $l \to r \in FR(\mathcal{C})$ and either $\mathcal{C}$ is non-collapsing, or $\succ \cup \succeq$ contains $\rhd^!$ and $f(\vec{x}) \succeq f^\#(\vec{x})$ for all $f \in \mathcal{D}$; if this step fails, return `fail`;
4. remove all pairs in $\mathcal{C}_1$ from the graph, since any cycle $\mathcal{C}'$ which includes such a pair is a subcycle of $\mathcal{C}$ and thus also proved non-looping by $(\succ, \succeq)$; continue with (1).

The algorithm iterates over a graph approximation, simplifying SCCs until none remain; note that this moves in the direction of the dependency pair framework as defined in [6].

▶ **Example 9.1.** Consider our running example twice, whose dependency graph was shown in Example 5.1. As instructed in step (1) of the algorithm, we remove nodes not on a cycle.



In step (2) we choose the SCC of all pairs in the graph; its formative rules are calculated in Example 7.3. For step (3) let $\mathcal{C}_1 := \{\mathsf{I}^\#(\mathsf{s}(n)) \rightsquigarrow \mathsf{twice}(\lambda x.\mathsf{I}(x)) \cdot n, \mathsf{I}^\#(\mathsf{s}(n)) \rightsquigarrow \mathsf{twice}^\#(\lambda x.\mathsf{I}(x))\}$ and $\mathcal{C}_2$ the set containing the other pairs. We have the following proof obligations:

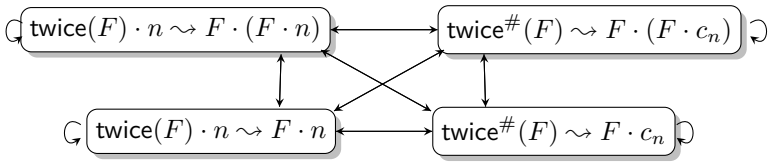
$A.$	$\mathsf{I}^\#(\mathsf{s}(n))$	$\succ$	$\mathsf{twice}(\lambda x.\mathsf{I}(x)) \cdot n$	$E.$	$\mathsf{twice}^\#(F) \cdot x$	$\succeq$	$F \cdot (F \cdot c_y)$
$B.$	$\mathsf{I}^\#(\mathsf{s}(n))$	$\succ$	$\mathsf{twice}^\#(\lambda x.\mathsf{I}(x)) \cdot c_z$	$F.$	$\mathsf{twice}^\#(F) \cdot x$	$\succeq$	$F \cdot c_y$
$C.$	$\mathsf{twice}(F) \cdot m$	$\succeq$	$F \cdot (F \cdot m)$	$G.$	$\mathsf{I}(\mathsf{s}(n))$	$\succeq$	$\mathsf{s}(\mathsf{twice}(\lambda x.\mathsf{I}(x)) \cdot n)$
$D.$	$\mathsf{twice}(F) \cdot m$	$\succeq$	$F \cdot m$	$H.$	$\mathsf{twice}(F) \cdot m$	$\succeq$	$F \cdot (F \cdot m)$

Requirement (H) is a duplicate of (C). Using an interpretation in functionals over the natural numbers where each $\mathcal{J}_{c_x} = 0$, and assuming $\mathcal{J}_{\mathsf{twice}} = \mathcal{J}_{\mathsf{twice}^\#}$, (B) is implied by (A), and (E) by (C), and (F) by (D). The remaining requirements are satisfied with $\mathcal{J}_{\mathsf{I}^\#} = \mathcal{J}_{\mathsf{I}} = \boldsymbol{\lambda} n.n$ and $\mathcal{J}_{\mathsf{s}} = \boldsymbol{\lambda} n.n + 1$ and $\mathcal{J}_{\mathsf{twice}^\#} = \mathcal{J}_{\mathsf{twice}} = \boldsymbol{\lambda} f.\boldsymbol{\lambda} n.f(f(n))$:

$A.$	$n+1$	$>$	$\max((\boldsymbol{\lambda} n.n)((\boldsymbol{\lambda} n.n)n), n) = \max(n, n) = n$
$C.$	$\max(F(F(n)), n)$	$\geq$	$\max(F(\max(F(n), n)), \max(F(n), n))$
$D.$	$\max(F(F(n)), n)$	$\geq$	$\max(F(n), n)$
$G.$	$n+1$	$\geq$	$\max(n, n) + 1 = n + 1$

The calculations for (A) and (G) are obvious. With some reasoning (distinguishing the cases $n > F(n)$, and $F(n) \geq n$ and noting that $F(n) \geq n$ implies $F(F(n)) \geq F(n)$ by weak monotonicity), (C) and (D) also hold.

Thus we move on to step (4) and remove the two nodes in $\mathcal{C}_1$ from the graph:



All nodes are still interconnected, so we continue with the SCC of all pairs. Interestingly, $FR(\mathcal{C}) = \emptyset$. Therefore it suffices to find a reduction pair with the usual properties and:

$$\begin{array}{lcllcl} \mathsf{twice}(F) \cdot n & \succ & F \cdot (F \cdot n) & \mathsf{twice}^{\#}(F) \cdot n & \succ & F \cdot (F \cdot c_y) \\ \mathsf{twice}(F) \cdot n & \succ & F \cdot n & \mathsf{twice}^{\#}(F) \cdot n & \succ & F \cdot c_y \end{array}$$

This is satisfied with an algebra interpretation with $\mathcal{J}_{\mathsf{twice}^{\#}} = \mathcal{J}_{\mathsf{twice}} = \boldsymbol{\lambda} fn. \max(f(f(n)), n) + 1$. Thus we remove the final four nodes from the graph, and conclude that twice is terminating.

Summary and Future Work We have defined a first basic dependency pair method for AFSs, with a variation of usable rules which takes into account the possible presence of collapsing dependency pairs. We have explained that besides orderings such as HORPO also monotone algebras can be used to solve the ordering constraints.

We intend to further study dependency pairs for AFSs with restrictions. For example, if function symbols have a base output type we can drop requirements, yielding an easier method. If we restrict to rules without abstractions in the left-hand sides, we may weaken the subterm property to obtain a stronger method, and define for instance argument filterings (in the extended abstract [15] a first step in this direction is given for HRSs).

A preliminary version of the dependency pair method with argument filterings is implemented in the tool WANDA v1.0 [14]. We intend to improve the implementation by taking into account also the dependency graph, strongly connected components and formative rules.

This work aims to contribute to the larger goal of understanding dependency pairs for higher order rewriting, and creating tools to automatically prove termination in this setting.

Acknowledgments. We are very grateful for the constructive comments of the referees that helped to improve the paper. We also gratefully acknowledge remarks from Jan Willem Klop.

References

1 T. Aoto and Y. Yamada. Dependency pairs for simply typed term rewriting. In J. Giesl, editor, *Proceedings of RTA 2005*, volume 3467 of *LNCS*, pages 120–134, Nara, Japan, April 2005. Springer.

2 T. Aoto and Y. Yamada. Argument filterings and usable rules for simply typed dependency pairs. In S. Ghilardi and R. Sebastiani, editors, *Proceedings of FroCoS 2009*, volume 5749 of *LNAI*, pages 117–132, Trento, Italy, September 2009. Springer.

3 T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236(1-2):133–178, 2000.

4 F. Blanqui. Higher-order dependency pairs. In *Proceedings of WST 2006*, Seattle, USA, August 2006.

5 F. Blanqui, J.-P. Jouannaud, and A. Rubio. The computability path ordering: The end of a quest. In *CSL 2008*, volume 5213 of *LNCS*, pages 1–14, Bertinoro, Italy, July 2008. Springer.

6 J. Giesl, R. Thiemann, and P. Schneider-Kamp. The dependency pair framework: Combining techniques for automated termination proofs. In *Proceedings of LPAR 2004*, volume 3452 of *Lecture Notes in Computer Science*, pages 301–331. Springer, 2005.

7 N. Hirokawa and A. Middeldorp. Automating the dependency pair method. *Information and Computation*, 205(4):474–511, 2007.

8 Nao Hirokawa, Aart Middeldorp, and Harald Zankl. Uncurrying for termination. In *LPAR 2008*, volume 5330 of *LNAI*, pages 667–681, Doha, 2008. Springer-Verlag.

9 G. Huet and D.C. Oppen. Equations and rewrite rules: a survey. In R.V Book, editor, *Formal Language Theory: Perspectives and Open Problems*, pages 349–405. Academic Press, London, 1980.

10 J.-P. Jouannaud and M. Okada. A computation model for executable higher-order algebraic specification languages. In *LICS 1991*, pages 350–361, Amsterdam, The Netherlands, July 1991. IEEE Computer Society Press.

11 J.-P. Jouannaud and A. Rubio. The higher-order recursive path ordering. In *LICS 1999*, pages 402–411, Trento, Italy, July 1999.

12 Z. Khasidashvili. Expression Reduction Systems. In *Proceedings of I. Vekua Institute of Applied Mathematics*, volume 36, pages 200–220, Tblisi, Georgia, 1990.

13 J.W. Klop. *Combinatory Reduction Systems*, volume 127 of *Mathematical Centre Tracts*. CWI, Amsterdam, The Netherlands, 1980. PhD Thesis.

14 C. Kop. Wanda. `http://www.few.vu.nl/~kop/code.html`.

15 C. Kop and F. van Raamsdonk. Higher-order dependency pairs with argument filterings. In *Proceedings of WST 2010*, Edinburgh, UK, July 2010. `http://www.few.vu.nl/~kop/wst10.pdf`.

16 K. Kusakari. On proving termination of term rewriting systems with higher-order variables. *IPSJ Transactions on Programming*, 42(SIG 7 PRO11):35–45, 2001.

17 K. Kusakari, Y. Isogai, M. Sakai, and F. Blanqui. Static dependency pair method based on strong computability for higher-order rewrite systems. *IEICE Transactions on Information and Systems*, 92(10):2007–2015, 2009.

18 K. Kusakari and M. Sakai. Enhancing dependency pair method using strong computability in simply-typed term rewriting. *AAECC*, 18(5):407–431, 2007.

19 K. Kusakari and M. Sakai. Static dependency pair method for simply-typed term rewriting and related techniques. *IEICE Transactions*, 2(92-D):235–247, 2009.

20 T. Nipkow. Higher-order critical pairs. In *LICS 1991*, pages 342–349, Amsterdam, The Netherlands, July 1991.

21 J.C. van de Pol. *Termination of Higher-order Rerwite Systems*. PhD thesis, University of Utrecht, 1996.

22 M. Sakai and K. Kusakari. On dependency pair method for proving termination of higher-order rewrite systems. *IEICE Transactions on Information and Systems*, E88-D(3):583–593, 2005.

23 M. Sakai, Y. Watanabe, and T. Sakabe. An extension of the dependency pair method for proving termination of higher-order rewrite systems. *IEICE Transactions on Information and Systems*, E84-D(8):1025–1032, 2001.

24 S. Suzuki, K. Kusakari, and F. Blanqui. Argument filterings and usable rules in higher-order rewrite systems. *IPSJ Transactions on Programming*, 4(2):1–12, 2011. To appear.

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

26 H. Zantema. Termination of term rewriting: interpretation and type elimination. *Journal of Symbolic Computation*, 17:23–50, 1994.

Anti-Unification for Unranked Terms and Hedges

Temur Kutsia[1], Jordi Levy[2], and Mateu Villaret[3]

1 **Research Institute for Symbolic Computation (RISC)**
Johannes Kepler University, Linz, Austria
kutsia@risc.jku.at
2 **Artificial Intelligence Research Institute (IIIA)**
Spanish Council for Scientific Research (CSIC)
Barcelona, Spain
levy@iiia.csic.es
3 **Departament d'Informàtica i Matemàtica Aplicada (IMA)**
Universitat de Girona (UdG), Girona, Spain
villaret@ima.udg.edu

Abstract

We study anti-unification for unranked terms and hedges that may contain term and hedge variables. The anti-unification problem of two hedges $\tilde{s}_1$ and $\tilde{s}_2$ is concerned with finding their generalization, a hedge $\tilde{q}$ such that both $\tilde{s}_1$ and $\tilde{s}_2$ are instances of $\tilde{q}$ under some substitutions. Hedge variables help to fill in gaps in generalizations, while term variables abstract single (sub)terms with different top function symbols. First, we design a complete and minimal algorithm to compute least general generalizations. Then, we improve the efficiency of the algorithm by restricting possible alternatives permitted in the generalizations. The restrictions are imposed with the help of a rigidity function that is a parameter in the improved algorithm and selects certain common subsequences from the hedges to be generalized. Finally, we indicate a possible application of the algorithm in software engineering.

1998 ACM Subject Classification F.4.2 [Theory of Computation]: Mathematical Logic and Formal Languages—Grammars and Other Rewriting Systems, F.2.2 [Theory of Computation]: Analysis of Algorithms and Problem Complexity—Nonnumerical Algorithms and Problems, D.2.7 [Software]: Software Engineering—Distribution, Maintenance, and Enhancement.

Keywords and phrases Anti-unification, generalization, unranked terms, hedges, software clones.

Digital Object Identifier 10.4230/LIPIcs.RTA.2011.219

Category Regular Research Paper

1 Introduction

The anti-unification problem of two terms t_1 and t_2 is concerned with finding their generalization, a term t such that both t_1 and t_2 are instances of t under some substitutions. The problem has a trivial solution, a fresh variable, that is the most general generalization of the given terms. Interesting generalizations are the least general ones. The purpose of anti-unification algorithms is to compute such least general generalizations. Plotkin [27] and Reynolds [28] pioneered research on anti-unification, designing generalization algorithms for ranked terms (where function symbols have a fixed arity) in the syntactic case. Since then, a number of algorithms and their modifications have been developed, addressing the problem in various theories (e.g., [1, 2, 4, 9, 15, 26]) and from different application points of view (e.g., [3, 8, 12, 17, 25, 31]). Applications come from the areas such as reasoning by analogy,

22nd International Conference on Rewriting Techniques and Applications (RTA'11).
Editor: M. Schmidt-Schauß; pp. 219–234

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


machine learning, inductive logic programming, software engineering, program synthesis, analysis, transformation, verification, just to name a few.

Unranked terms differ from the ranked ones by not having fixed arity for function symbols. Hedges are finite sequences of such terms. They are flexible structures, popular in representing semistructured data. To take the advantage of variadicity, unranked terms and hedges use two kinds of variables: term variables that stand for a single term and hedge variables that stand for hedges. Solving techniques over unranked terms and hedges mostly address unification and matching problems, see, e.g., [11, 18, 19, 20, 21, 23, 24]. Anti-unification for these structures practically has not been studied. The only exceptions, to the best of our knowledge, are [3, 33], where anti-unification of feature terms, and a special case of so called simple hedges are considered, respectively.

We address this shortcoming, presenting algorithms to compute least general generalizations for unranked terms/hedges. Hedge variables help to fill in gaps in generalizations, while term variables abstract single (sub)terms with different top function symbols. First, we develop a complete and minimal algorithm. Next, we improve its efficiency by restricting possible alternatives permitted in the generalizations. The restrictions are imposed with the help of a rigidity function that is a parameter in the improved algorithm. At each step, the algorithm decides which subsequence of terms of the given hedges is to be (structurally) retained in the generalization. This gives more efficient, yet pretty general algorithm for a generic rigidity function. Instantiating the parameter with specific rigidity functions, we obtain various special instances.

Finally, we discuss a possible application in software code clone detection. Our results open a possibility to address the problem of searching XML clones by means of anti-unification. Rigid hedge generalizations provide several advantages for this, combining fast textual and precise structural techniques.

2 Preliminaries

Given pairwise disjoint countable sets of unranked function symbols F (symbols without fixed arity), term variables $\mathcal{V}_\mathsf{T}$, and hedge variables $\mathcal{V}_\mathsf{H}$, we define *unranked terms* (terms in short) and *hedges* (sequences of terms or hedge variables) over F and $\mathsf{V} = \mathcal{V}_\mathsf{T} \cup \mathcal{V}_\mathsf{H}$ by the grammar: $t ::= x \mid f(\tilde{s}),\ s ::= t \mid X,\ \tilde{s} ::= s_1, \ldots, s_n$, where $x \in \mathcal{V}_\mathsf{T}$, $f \in \mathsf{F}$, $X \in \mathcal{V}_\mathsf{H}$, and $n \geq 0$. With this definition, terms are singleton hedges. Not all singleton hedges are terms: some may be hedge variables. If $\tilde{s} = s_1, \ldots, s_n$ and $\tilde{s}' = s'_1, \ldots, s'_m$, then we write $\tilde{s}, \tilde{s}'$ for $s_1, \ldots, s_n, s'_1, \ldots, s'_m$. We denote by $\tilde{s}|_i$ the ith element of $\tilde{s}$. We denote by $\tilde{s}|_i^j$, where $i < j$, the subsequence between positions i and j excluding them, i.e., the subsequence $\tilde{s}|_{i+1}, \ldots, \tilde{s}|_{j-1}$. The length of a sequence $\tilde{s}$, denoted $|\tilde{s}|$, is the number of elements in it.

The set of terms (resp., the set of hedges) over F and V is denoted by $\mathsf{T}(\mathsf{F}, \mathcal{V}_\mathsf{T}, \mathcal{V}_\mathsf{H})$ (resp., by $\mathsf{H}(\mathsf{F}, \mathcal{V}_\mathsf{T}, \mathcal{V}_\mathsf{H})$). We use the letters f, g, h, a, b, c, and d for function symbols, x, y, and z for term variables, X, Y, Z, U, and V for hedge variables, χ for a term variable or a hedge variable, t, l, and r for terms, s and q for a hedge variable or a term, and $\tilde{s}$ and $\tilde{q}$ for hedges. The empty hedge is denoted by ϵ. The terms of the form $a(\epsilon)$ are written as just a.

The *size* of a term t is the number of occurrences of symbols (from $\mathsf{F} \cup \mathsf{V}$) in it and is denoted by $\mathit{size}(t)$. We denote by $\mathit{var}(t)$ the *set of variables* of a term. These definitions are generalized for any syntactic object.

A *substitution* is a mapping from hedge variables to hedges and from term variables to terms, which is identity almost everywhere. We will use the traditional finite set representation of substitutions, writing, e.g., $\{x \mapsto f(a), X \mapsto \epsilon,\ Y \mapsto x, g(a, Z)\}$ for the substitution that

maps every variable to itself except x, X, and Y that are mapped respectively to $f(a)$, to ϵ, and to $x, g(a, Z)$. The lower case Greek letters are used to denote substitutions, with the exception of the identity substitution for which we write Id.

Substitutions can be applied to terms and hedges using the congruences

$$\sigma(f(s_1, \ldots, s_n)) = f(\sigma(s_1), \ldots, \sigma(s_n)), \qquad \sigma(s_1, \ldots, s_n) = \sigma(s_1), \ldots, \sigma(s_n).$$

We call $\sigma(s)$ and $\sigma(\tilde{s})$ the *instances* of respectively s and $\tilde{s}$ and use postfix notation to denote them, writing $s\sigma$ and $\tilde{s}\sigma$. We also say that $\tilde{s}$ is *more general* than $\tilde{q}$ if $\tilde{q}$ is an instance of $\tilde{s}$ and denote this fact by $\tilde{s} \preceq \tilde{q}$. If $\tilde{s} \preceq \tilde{q}$ and $\tilde{q} \preceq \tilde{s}$, then we write $\tilde{s} \simeq \tilde{q}$. If $\tilde{s} \preceq \tilde{q}$ and $\tilde{s} \not\simeq \tilde{q}$, then we say that $\tilde{s}$ is *strictly more general* than $\tilde{q}$ and write $\tilde{s} \prec \tilde{q}$. The set $dom(\sigma) = \{\chi \in \mathsf{V} \mid \chi\sigma \neq \chi\}$ is called the *domain* of σ.

The *composition* of two substitutions σ and ϑ, written as $\sigma\vartheta$, is defined as the composition of two mappings: We have $s(\sigma\vartheta) = (s\sigma)\vartheta$ for all s. A substitution σ_1 is *more general* than σ_2 with respect to a set of variables $\mathcal{X} \subseteq \mathsf{V}$, written $\sigma_1 \preceq_{\mathcal{X}} \sigma_2$, if there exists ϑ such that $\chi\sigma_1\vartheta = \chi\sigma_2$, for each $\chi \in \mathcal{X}$. The relations $\simeq$ and $\prec$ are extended to substitutions: $\sigma_1 \simeq_{\mathcal{X}} \sigma_2$ means $\sigma_1 \preceq_{\mathcal{X}} \sigma_2$ and $\sigma_2 \preceq_{\mathcal{X}} \sigma_1$, and $\sigma_1 \prec_{\mathcal{X}} \sigma_2$ means $\sigma_1 \preceq_{\mathcal{X}} \sigma_2$ and $\sigma_1 \not\simeq_{\mathcal{X}} \sigma_2$.

The *top symbol* of a term is defined as $top(x) = x$ and $top(f(\tilde{s})) = f$. We extend this notion to hedges, defining it as the sequence of symbols as follows: $top(\epsilon) = \epsilon$, $top(X, \tilde{s}) = X\,top(\tilde{s})$, and $top(t, \tilde{s}) = top(t)top(\tilde{s})$. Notice that we write these sequences as words, e.g., $top(f(a), a, X, x) = faXx$. The letter w will be used for those words.

A hedge $\tilde{s}$ is called a *generalization* or an *anti-instance* of two hedges $\tilde{s}_1$ and $\tilde{s}_2$ if $\tilde{s} \preceq \tilde{s}_1$ and $\tilde{s} \preceq \tilde{s}_2$. That means, there exist substitutions σ_1 and σ_2 such that $\tilde{s}_1 = \tilde{s}\sigma_1$ and $\tilde{s}_2 = \tilde{s}\sigma_2$. We say that a hedge $\tilde{s}$ is a *least general generalization* (lgg in short), aka a *most specific anti-instance*, of $\tilde{s}_1$ and $\tilde{s}_2$ if $\tilde{s}$ is a generalization of $\tilde{s}_1$ and $\tilde{s}_2$ and there is no generalization $\tilde{q}$ of $\tilde{s}_1$ and $\tilde{s}_2$ that satisfies $\tilde{s} \prec \tilde{q}$. That means, there are no generalizations of $\tilde{s}_1$ and $\tilde{s}_2$ that are strictly less general than their least general generalization.

An *anti-unification problem* (or equation), AUP in short, is a triple $\chi : \tilde{s}_1 \triangleq \tilde{s}_2$, where χ does not occur in $\tilde{s}_1$ and $\tilde{s}_2$. Intuitively, χ is a variable that stands for the most general generalization of $\tilde{s}_1$ and $\tilde{s}_2$. An *anti-unifier* of $\chi : \tilde{s}_1 \triangleq \tilde{s}_2$ is a substitution σ such that $dom(\sigma) \subseteq \{\chi\}$ and $\chi\sigma$ is a generalization of both $\tilde{s}_1$ and $\tilde{s}_2$. An anti-unifier σ of an AUP $\chi : \tilde{s}_1 \triangleq \tilde{s}_2$ is *least general (or most specific)* if there is no anti-unifier of ϑ of the same problem that satisfies $\sigma \prec_{\mathcal{X}} \vartheta$. Obviously, if σ is a least general anti-unifier of an AUP $\chi : \tilde{s}_1 \triangleq \tilde{s}_2$, then $\chi\sigma$ is a least general generalization of $\tilde{s}_1$ and $\tilde{s}_2$.

A *complete set of generalizations* of two hedges $\tilde{s}_1$ and $\tilde{s}_2$ is a set G of hedges that satisfies the properties:
Soundness: Each $\tilde{q} \in G$ is a generalization of both $\tilde{s}_1$ and $\tilde{s}_2$.
Completeness: For each generalization $\tilde{s}$ of $\tilde{s}_1$ and $\tilde{s}_2$, there exists $\tilde{q} \in G$ such that $\tilde{s} \preceq \tilde{q}$.

G is a *minimal complete set of generalizations* of $\tilde{s}_1$ and $\tilde{s}_2$ if it, in addition to soundness and completeness, satisfies also the following property:
Minimality: For each $\tilde{q}_1, \tilde{q}_2 \in G$, if $\tilde{q}_1 \preceq \tilde{q}_2$ then $\tilde{q}_1 = \tilde{q}_2$.

▶ **Lemma 2.1.** *For any three hedges $\tilde{s}_1$, $\tilde{s}_2$ and $\tilde{q}$, and any pair of substitutions σ_1 and σ_2 satisfying $\tilde{s}_1 = \tilde{q}\sigma_1$ and $\tilde{s}_2 = \tilde{q}\sigma_2$, if $size(\tilde{q}) \geq size(\tilde{s}_1) + size(\tilde{s}_2)$ then there exists a hedge variable X occurring in $\tilde{q}$ such that $X\sigma_1 = X\sigma_2 = \epsilon$.*

Proof. The hedge $\tilde{q}$ can not contain more function symbols than $\tilde{s}_1$ and $\tilde{s}_2$ do. It also can not contain more term variables than there are subterms in $\tilde{s}_1$ or $\tilde{s}_2$. Violation of any of these conditions would forbid $\tilde{s}_1$ or $\tilde{s}_2$ to be an instance of $\tilde{q}$. Hence, the only reason why $size(\tilde{q}) \geq size(\tilde{s}_1) + size(\tilde{s}_2)$ is that $\tilde{q}$ may contain extra hedge variables that are mapped to ϵ by both σ_1 and σ_2. ◀

▶ **Lemma 2.2.** *For any hedges $\tilde{s}_1$ and $\tilde{s}_2$ there exists their minimal complete set of generalizations that, modulo $\simeq$, is unique and finite.*

Proof. For classical first-order anti-unification this property is trivial, because instantiation does not decrease the size of terms. This means that anti-unifiers of two terms are smaller than each of those terms, hence finite modulo variable renaming. For hedges the property is not so simple to prove because instantiating a hedge variable by ϵ, the size of a term may decrease. However, by Lemma 2.1 we have that for any anti-unifier $\tilde{q}$ of $\tilde{s}_1$ and $\tilde{s}_2$ with $size(\tilde{q}) \geq size(\tilde{s}_1) + size(\tilde{s}_2)$ there exists another anti-unifier less general than $\tilde{q}$ (that we can obtain by replacing those extra hedge variables in $\tilde{q}$ by ϵ). The set of anti-unifiers smaller than the sum of the sizes of both hedges is a complete set of anti-unifiers, and it is finite and unique modulo $\simeq$. ◀

We denote the minimal complete set of generalizations of $\tilde{s}_1$ and $\tilde{s}_2$ by $mcg(\tilde{s}_1, \tilde{s}_2)$. Its elements are lggs of $\tilde{s}_1$ and $\tilde{s}_2$.

Like unification problems, anti-unification problems may be classified as unitary (if minimal complete sets of generalizations always exist and are singletons), finitary (if they always exist, are finite, and the problem is not unitary), infinitary (if they always exist and may be infinite), and nullary (if they may not exist). Hence, Lemma 2.2 implies that hedge anti-unification is finitary.

An anti-unification problem always has an anti-unifier. The empty substitution is a trivial example that represents the most general generalization. Our goal is to compute less general generalizations. In the next section, we design an algorithm that computes (the set of anti-unifiers that represents) the *mcg* of a given AUP. It requires some care to properly address the issues that arise because of hedge variables in generalizations.

Quiz 1: *Given two hedges $\tilde{s} = f(a), f(a)$ and $\tilde{q} = f(a), f$, what is the set $mcg(\tilde{s}, \tilde{q})$? Hint: There are three elements in $mcg(\tilde{s}, \tilde{q})$.*

Below we assume that the hedges to be generalized are variable disjoint.

3 Complete and Minimal Algorithm

We present our anti-unification algorithm as a rule-based algorithm that works on triples $A; S; \sigma$. Here A is a set of AUPs of the form $\{X_1 : \tilde{s}_1 \triangleq \tilde{q}_1, \ldots, X_n : \tilde{s}_n \triangleq \tilde{q}_n\}$ where each X_i occurs in the problem only once, S is a set of already solved anti-unification equations (the store), and σ is a substitution (computed so far)[1]. We call such a triple a *system*. The rules transform systems into systems:

T-H: **Trivial Hedge**

$$\{X : \epsilon \triangleq \epsilon\} \cup A;\ S;\ \sigma \Longrightarrow A;\ S;\ \sigma\{X \mapsto \epsilon\}.$$

Dec-T: **Decomposition for Terms**

$$\{X : f(\tilde{s}) \triangleq f(\tilde{q})\} \cup A;\ S;\ \sigma \Longrightarrow \{Y : \tilde{s} \triangleq \tilde{q}\} \cup A;\ S;\ \sigma\{X \mapsto f(Y)\}$$

where Y is a fresh variable.

Dec1-H: **Decomposition 1 for Hedges**

$$\{X : s, \tilde{s} \triangleq q, \tilde{q}\} \cup A;\ S;\ \sigma \Longrightarrow \{Y : s \triangleq q, Z : \tilde{s} \triangleq \tilde{q}\} \cup A;\ S;\ \sigma\{X \mapsto Y, Z\},$$

where $U : s, \tilde{s} \triangleq q, \tilde{q} \notin S$ for all $U \in \mathcal{V}_{\mathsf{H}}$, the variables Y and Z are fresh, and $\tilde{s} \neq \epsilon$ or $\tilde{q} \neq \epsilon$.

[1] Such a representation was first proposed in [1] for equational anti-unification.

Dec2-H: **Decomposition 2 for Hedges**

$$\{X : s, \tilde{s} \triangleq \tilde{q}\} \cup A;\ S;\ \sigma \Longrightarrow \{Y : s \triangleq \epsilon, Z : \tilde{s} \triangleq \tilde{q}\} \cup A;\ S;\ \sigma\{X \mapsto Y, Z\},$$

where $\chi : s, \tilde{s} \triangleq \tilde{q} \notin S$ for all χ, the variables Y and Z are fresh, and $\tilde{s} \neq \epsilon$ or $\tilde{q} \neq \epsilon$.

Dec3-H: **Decomposition 3 for Hedges**

$$\{X : \tilde{s} \triangleq q, \tilde{q}\} \cup A;\ S;\ \sigma \Longrightarrow \{Y : \epsilon \triangleq q, Z : \tilde{s} \triangleq \tilde{q}\} \cup A;\ S;\ \sigma\{X \mapsto Y, Z\},$$

where $\chi : \tilde{s} \triangleq q, \tilde{q} \notin S$ for all χ, the variables Y and Z are fresh, and $\tilde{s} \neq \epsilon$ or $\tilde{q} \neq \epsilon$.

Sol1-H: **Solve 1 for Hedges**

$$\{X : s \triangleq \epsilon\} \cup A;\ S;\ \sigma \Longrightarrow A;\ \{X : s \triangleq \epsilon\} \cup S;\ \sigma, \quad \text{if } Y : s \triangleq \epsilon \notin S \text{ for all } Y.$$

Sol2-H: **Solve 2 for Hedges**

$$\{X : \epsilon \triangleq q\} \cup A;\ S;\ \sigma \Longrightarrow A;\ \{X : \epsilon \triangleq q\} \cup S;\ \sigma, \quad \text{if } Y : \epsilon \triangleq q \notin S \text{ for all } Y.$$

Sol3-H: **Solve 3 for Hedges**

$$\{X : s \triangleq q\} \cup A;\ S;\ \sigma \Longrightarrow A;\ \{X : s \triangleq q\} \cup S;\ \sigma,$$

if $s \neq q$, $s \in \mathcal{V}_{\mathsf{H}}$ or $q \in \mathcal{V}_{\mathsf{H}}$, and $Y : s \triangleq q \notin S$ for all Y.

Sol-T: **Solve for Terms**

$$\{X : l \triangleq r\} \cup A;\ S;\ \sigma \Longrightarrow A;\ \{y : l \triangleq r\} \cup S;\ \sigma\{X \mapsto y\},$$

if $top(l) \neq top(r)$, $\chi : l \triangleq r \notin S$ for all χ, and y is fresh.

Rec: **Recover**

$$\{X : \tilde{s} \triangleq \tilde{q}\} \cup A;\ \{\chi : \tilde{s} \triangleq \tilde{q}\} \cup S;\ \sigma \Longrightarrow A;\ \{\chi : \tilde{s} \triangleq \tilde{q}\} \cup S;\ \sigma\{X \mapsto \chi\}.$$

The idea of the store is to keep track of already solved AUPs in order to generalize the same pair of hedges with the same variable, as it is illustrated in the Rec rule: The already solved AUP $\chi : \tilde{s} \triangleq \tilde{q}$ from the store helps to reuse χ instead of X as a generalization of $\tilde{s}$ and $\tilde{q}$. This is important, since we aim at computing lggs.

In the condition of Dec1-H we use a hedge variable U while in Dec2-H and Dec3-H in the same role χ appears. The reason is that in Dec1-H, the hedge $s, \tilde{s}$ or the hedge $q, \tilde{q}$ is not a term and, hence, we can not have a term variable in place of U. On the other hand, in Dec2-H and Dec3-H it can happen that the AUP in the condition is between terms with χ being a term variable.

Notice that there is no rule for AUPs of the form $X : x \triangleq x$. This is because we assume the hedges to be generalized are variable disjoint and, hence, such problems do not appear.

To compute generalizations for hedges $\tilde{s}$ and $\tilde{q}$, the procedure starts with $\{X : \tilde{s} \triangleq \tilde{q}\}; \emptyset; Id$ where X is a fresh hedge variable and applies the rules on each selected anti-unification equation in all possible ways. We denote this procedure by $\mathfrak{G}$. To show that the process terminates, we define a complexity measure of the triple $A; S; \sigma$ as a multiset $M(A) := \{size(\tilde{s} \triangleq \tilde{q}) + 1 \mid X : \tilde{s} \triangleq \tilde{q} \in A\}$. We order complexity measures by the multiset extension $>_m$ of the standard ordering on natural numbers. It is easy to check that the theorem below holds, which immediately implies termination:

▶ **Theorem 3.1.** *If* $A_1; S_1; \sigma_1 \Longrightarrow A_2; S_2; \sigma_2$ *in* $\mathfrak{G}$, *then* $M(A_1) >_m M(A_2)$.

Hence, starting from $\{X : \tilde{s} \triangleq \tilde{q}\}; \emptyset; Id$, each sequence of transformations by $\mathfrak{G}$ necessarily terminates with a triple of the form $\emptyset; S; \sigma$.

▶ **Theorem 3.2** (Soundness of $\mathfrak{G}$). *If* $\{X : \tilde{s} \triangleq \tilde{q}\}; \emptyset; Id \Longrightarrow^* \emptyset; S; \sigma$ *is a derivation in* $\mathfrak{G}$, *then* $X\sigma \preceq \tilde{s}$ *and* $X\sigma \preceq \tilde{q}$.

Proof. The theorem follows from the straightforward fact that if $X\sigma \preceq \tilde{s}$ and $X\sigma \preceq \tilde{q}$ and $\{X : \tilde{s} \triangleq \tilde{q}\} \cup A; S; \sigma \Longrightarrow A', S', \sigma'$ is a transformation step in $\mathfrak{G}$, then $X\sigma' \preceq \tilde{s}$ and $X\sigma' \preceq \tilde{q}$. ◀

If $\{X : \tilde{s} \triangleq \tilde{q}\}; \emptyset; Id \Longrightarrow^* \emptyset; S; \sigma$ is a derivation in $\mathfrak{G}$, then we say that

- σ is a *substitution computed by* $\mathfrak{G}$ *for* $X : \tilde{s} \triangleq \tilde{q}$*;*
- the restriction of σ on X, denoted by $\sigma|_X$, is a *anti-unifier of* $X : \tilde{s} \triangleq \tilde{q}$ *computed by* $\mathfrak{G}$*;*
- the hedge $X\sigma$ is a *generalization of* $\tilde{s}$ *and* $\tilde{q}$ *computed by* $\mathfrak{G}$*.*

The proof of completeness of the algorithm requires auxiliary definitions and lemmas. We start generalizing the notion of anti-unifier for sets of equations.

▶ **Definition 3.3.** A set of AUPs is a set $A = \{\chi_1 : \tilde{s}_1 \triangleq \tilde{q}_1, \ldots, \chi_n : \tilde{s}_n \triangleq \tilde{q}_n\}$, where each of the variables $\chi_1, \ldots, \chi_n$ does not occur more than once. We define the set of generalization variables $gvar(A) = \{\chi_1, \ldots, \chi_n\}$.

▶ **Definition 3.4.** A substitution σ is called an anti-unifier of a set of AUPs A, if $dom(\sigma) \subseteq gvar(A)$ and for each $(\chi : \tilde{s} \triangleq \tilde{q}) \in A$, $\chi\sigma$ is a generalization of both $\tilde{s}$ and $\tilde{q}$.

Similarly, least general anti-unifiers are also generalized for sets of AUPs.

▶ **Definition 3.5.** We say that a set of AUPs A is unsimplifiable if any anti-unifier of A is equal to Id modulo variable renaming.

Notice that if A is unsimplifiable then A cannot contain equations with pairs of terms with the same top symbol $x : f(\tilde{s}) \triangleq f(\tilde{q})$, equations between equal sequences $\chi : \tilde{s} \triangleq \tilde{s}$, equations between terms $X : f(\tilde{s}) \triangleq g(\tilde{q})$ where $X \in \mathcal{V}_\mathsf{H}$, nor pairs of identical equations $\chi : \tilde{s} \triangleq \tilde{q}, \chi' : \tilde{s} \triangleq \tilde{q}$.

▶ **Lemma 3.6.** *Let A be a set of AUPs satisfying $gvar(A) \subseteq \mathcal{V}_\mathsf{H}$. Let S be an unsimplifiable set of AUPs. Let ϑ be an anti-unifier of A. Then, there exists a sequence of transformations $A; S; Id \Longrightarrow^* \emptyset; S'; \sigma$ where $\vartheta \preceq_{gvar(A)} \sigma$.*

This lemma is crucial for showing completeness of $\mathfrak{G}$. Its proof is quite long and proceeds by structural induction on A and by detailed case analysis on the form of a selected AUP in transformations. The interested reader can find it in the technical report [22].

▶ **Theorem 3.7** (Completeness of $\mathfrak{G}$)**.** *Let ϑ be an anti-unifier of $X : \tilde{s} \triangleq \tilde{q}$. Then $\mathfrak{G}$ computes a substitution σ such that $X\vartheta \preceq X\sigma$.*

Proof. Immediate consequence of Lemma 3.6 with $A = \{X : \tilde{s} \triangleq \tilde{q}\}$ and $S = \emptyset$. ◀

Hence, collecting all the hedges $X\sigma$ such that $\{X : \tilde{s} \triangleq \tilde{q}\}; \emptyset; Id \Longrightarrow^* \emptyset; S; \sigma$, we obtain a finite complete set of generalizations of $\tilde{s}$ and $\tilde{q}$. In general, this set is not minimal. Even for such a simple input as $\{X : f(a) \triangleq f(b)\}; \emptyset; Id$, the algorithm $\mathfrak{G}$ produces five generalizations: two hedges Y_1, Y_2 and Z_1, Z_2 and three terms $f(U_1, U_2)$, $f(V_1, V_2)$, and $f(x)$. The last term is an instance of the other four generalizations.

Nevertheless, this redundancy is not trivially avoidable because rules allowing apparently useless alignments are needed for completeness:

Answer to Quiz 1. *Besides the "expected" lgg $f(a), f(X)$, the set $mcg(\tilde{s}, \tilde{q})$ for $\tilde{s} = f(a), f(a)$ and $\tilde{q} = f(a), f$ also contains two less obvious ones: $f(X, Y), f(X)$ and $f(X, Y), f(Y)$.*

We need a minimization step to keep only least general generalizations. Minimization involves a matchability test between two hedges. If two hedges $\tilde{s}$ and $\tilde{q}$ are in the set we are going to minimize, then we proceed as follows:

- If $\tilde{s} \simeq \tilde{q}$, then we delete one of them and keep the other (e.g., with the smaller size).
- If one of them is strictly more general than the other one, we delete the more general hedge and keep the more specific one.

For matchability, one could, in principle, use the hedge matching algorithm from [18], but there is a subtlety one should take into account: The hedges that are to be matched, in general, are not ground. Therefore, when trying to match, e.g., $\tilde{s} = X, X$ to $\tilde{q} = X, a$, we should rename X in $\tilde{q}$ into a new constant. Furthermore, we should introduce a restriction that no term variable matches such new constants. Thus, the matchability test should fail for the problems like $X, X \ll X, a$ and $x \ll X$.

Hence, combining $\mathfrak{G}$ with minimization, we can compute $mcg(\tilde{s}_1, \tilde{s}_2)$ for each $\tilde{s}_1$ and $\tilde{s}_2$.

▶ **Example 3.8.** For the terms $f(g(a, X), a, X, b)$ and $f(g(b), b)$, $\mathfrak{G}$ computes the *mcg*: $\{f(g(x, Y), Z, Y, b),\ f(g(x, Y), x, Y, Z),\ f(g(U, Y, Z), Y, Z, b),\ f(g(U, Y, Z), U, Y, b)\}$. These four lggs are selected from 169 generalizations computed in the first step of the algorithm.

The drawback of the algorithm $\mathfrak{G}$ is that it is highly nondeterministic. It computes $O(3^n)$ generalizations[2], where n is the size of the input. (For instance, for $f(a_1, a_2, a_3, a_4, a_5)$ and $f(b_1, b_2, b_3, b_4, b_5)$ it computes 11685 generalizations, most of them several times, until it selects a single one, e.g., $f(x_1, x_2, x_3, x_4, x_5)$, on the minimization step.) The minimization step involves NP-complete hedge matching algorithm (see [18, 21]) performed on the pairs of elements of the generalization set. Hence, this algorithm is only of theoretical interest and falls short of being practically useful. Our goal is to impose requirements on the set of generalizations such that, on the one hand, it is still "interesting", on the other hand, it can be computed faster in many cases. This leads us to the notion of rigid generalization, described in the next section.

4 Computing Rigid Generalizations

The main intuition behind rigid generalizations is to capture the structure (modulo a given rigidity property) of as many nonvariable terms in the input hedges as possible. It is parameterized by a binary *rigidity function* $\mathcal{R}$ that computes a finite set of *alignments* for strings, defined as follows:

▶ **Definition 4.1** (Alignment and Rigidity Function)**.** Let w_1 and w_2 be strings of symbols. Then the sequence $a_1[i_1, j_1] \cdots a_n[i_n, j_n]$, for $n \geq 0$, is an alignment if

- i's and j's are positive integers such that $i_1 < \cdots < i_n$ and $j_1 < \cdots < j_n$, and
- $a_k = w_1|_{i_k} = w_2|_{j_k}$, for all $1 \leq k \leq n$.

A rigidity function $\mathcal{R}$ is a function that returns, for every pair of strings of symbols w_1 and w_2, a set of alignments of w_1 and w_2.

▶ **Example 4.2.** We give some examples of rigidity functions. Here and below, instead of saying that the rigidity function $\mathcal{R}$ returns "the set of alignments of ...", we just say that it returns "the set of ...".

- $\mathcal{R}$ returns the set of all longest common subsequences of its arguments: $\mathcal{R}(abc, dd) = \{\epsilon\}$, $\mathcal{R}(abcda, bcad) = \{b[2, 1]c[3, 2]a[5, 3], b[2, 1]c[3, 2]d[4, 4]\}$.
- $\mathcal{R}$ returns the set of all those longest common subsequences whose length is at least 4: $\mathcal{R}(abcda, bca) = \emptyset$, $\mathcal{R}(abcda, bcacda) = \{a[1, 3]c[3, 4]d[4, 5]a[5, 6], b[2, 1]c[3, 4]d[4, 5]a[5, 6]\}$.

[2] Notice that the hedge decomposition rule has three non-deterministic choices.

- $\mathcal{R}$ returns the set of all longest common substrings of its arguments: $\mathcal{R}(abcda, bcad) = \{b[2,1]c[3,2]\}$, $\mathcal{R}(abcda,\ bcada) = \{b[2,1]c[3,2], d[4,4]a[5,5]\}$, $\mathcal{R}(abc, dd) = \{\epsilon\}$.

▶ **Definition 4.3** ($\mathcal{R}$-Generalization)**.** Given two (variable disjoint) hedges $\tilde{s}_1$ and $\tilde{s}_2$ and the rigidity function $\mathcal{R}$, we say that a hedge $\tilde{s}$ that generalizes both $\tilde{s}_1$ and $\tilde{s}_2$ is their *generalization with respect to* $\mathcal{R}$, or, in short, an $\mathcal{R}$-generalization, if either $\mathcal{R}(top(\tilde{s}_1), top(\tilde{s}_2)) = \emptyset$ and $\tilde{s}$ is a hedge variable, or there exists an alignment $f_1[i_1, j_1] \cdots f_n[i_n, j_n] \in \mathcal{R}(top(\tilde{s}_1), top(\tilde{s}_2))$, such that the following conditions are fulfilled:

1. The sequence $\tilde{s}$ does not contain pairs of consecutive hedge variables.
2. If we remove all hedge variables that occur as elements of $\tilde{s}$, we get a sequence of the form $f_1(\tilde{q}_1), \ldots, f_n(\tilde{q}_n)$.
3. For every $1 \leq k \leq n$, there exists a pair of sequences $\tilde{s}'_1$ and $\tilde{s}'_2$ such that $\tilde{s}_1|_{i_k} = f_k(\tilde{s}'_1)$, $\tilde{s}_2|_{j_k} = f_k(\tilde{s}'_2)$ and $\tilde{q}_k$ is an $\mathcal{R}$-generalization of $\tilde{s}'_1$ and $\tilde{s}'_2$.

Under this definition, $\mathcal{R}$-generalizations do not contain term variables. The minimal complete set of $\mathcal{R}$-generalizations of $\tilde{s}_1$ and $\tilde{s}_2$ is denoted by $mcg_{\mathcal{R}}(\tilde{s}_1, \tilde{s}_2)$. An *$\mathcal{R}$-anti-unifier* of $X : \tilde{s}_1 \triangleq \tilde{s}_2$ is a substitution σ such that $X\sigma$ is an $\mathcal{R}$-generalization of $\tilde{s}_1$ and $\tilde{s}_2$.

▶ **Example 4.4.** Let $\mathcal{R}(w_1, w_2)$ be the set of all longest common subsequences of w_1 and w_2.

- The terms $t_1 = f(g(a, X), a, X, b)$ and $t_2 = f(g(b), b)$ have a single least general $\mathcal{R}$-generalization $f(g(Y), Z, b)$. Note that this term does not belong to $mcg(t_1, t_2)$ computed in Example 3.8.
- $f(g(a, a), a, X, b)$ and $f(g(b, b), g(Y), b)$ have two $\mathcal{R}$-generalizations: $f(g(U), Z, b)$ and $f(V, g(U), Z, b)$. The first one is less general than the second one.
- The hedges a, b and b, c have a single $\mathcal{R}$-generalization: X, b, Y.

▶ **Example 4.5.** Let $\mathcal{R}(w_1, w_2)$ be the set of all longest common substrings of w_1 and w_2.

- The least general $\mathcal{R}$-generalization of $a, a, b, f, f, f(a, a, b)$ and $a, a, c, f, f, f(a, a, c)$ is the hedge $X, f, f, f(a, a, Y)$.
- The least general $\mathcal{R}$-generalization of $a, a, b, b, f, f, f(a, a, b, b)$ and $a, a, c, f, f, f(a, a, c)$ is the hedge $X, f, f, f(a, a, Y)$.

Quiz 2: *What is the $mcg_{\mathcal{R}}(\tilde{s}, \tilde{q})$ for two identical hedges $\tilde{s} = f(a, b, c), g(a), h(a)$ and $\tilde{q} = f(a, b, c), g(a), h(a)$, where $\mathcal{R}$ is a function that computes the set of all common subsequences of the minimal length 3 of its arguments?*

Our goal is to compute a minimal complete set of $\mathcal{R}$-generalizations. For this, we design a new set of transformation rules. It consists of only four rules shown below:

$\mathcal{R}$-Dec-H: **$\mathcal{R}$-Rigid Decomposition for Hedges**

$$\{X : \tilde{s} \triangleq \tilde{q}\} \cup A;\ S;\ \sigma \Longrightarrow$$
$$\{Z_k : \tilde{s}_k \triangleq \tilde{q}_k \mid 1 \leq k \leq n\} \cup A;$$
$$\{Y_0 : \tilde{s}|_0^{i_1} \triangleq \tilde{q}|_0^{j_1}\} \cup \{Y_k : \tilde{s}|_{i_k}^{i_{k+1}} \triangleq \tilde{q}|_{j_k}^{j_{k+1}} \mid 1 \leq k \leq n-1\} \cup \{Y_n : \tilde{s}|_{i_n}^{|\tilde{s}|+1} \triangleq \tilde{q}|_{j_n}^{|\tilde{q}|+1}\} \cup S;$$
$$\sigma\{X \mapsto Y_0, f_1(Z_1), Y_1, \ldots, Y_{n-1}, f_n(Z_n), Y_n\},$$

if $\mathcal{R}(top(\tilde{s}), top(\tilde{q}))$ contains a sequence $f_1[i_1, j_1] \cdots f_n[i_n, j_n]$ such that for all $1 \leq k \leq n$, $\tilde{s}|_{i_k} = f_k(\tilde{s}_k)$, $\tilde{q}|_{j_k} = f_k(\tilde{q}_k)$, and Y_0, Y_k's and Z_k's are fresh.

$\mathcal{R}$-S-H: **$\mathcal{R}$-Rigid Solve for Hedges**

$$\{X : \tilde{s} \triangleq \tilde{q}\} \cup A;\ S;\ \sigma \Longrightarrow A;\ \{X : \tilde{s} \triangleq \tilde{q}\} \cup S;\ \sigma,$$

if $\mathcal{R}(top(\tilde{s}), top(\tilde{q})) = \emptyset$. (Notice that this transformation is equivalent to rule $\mathcal{R}$-Dec-H where $\mathcal{R}(top(\tilde{s}), top(\tilde{q})) = \{\epsilon\}$).

$\mathcal{R}$-CS1: **$\mathcal{R}$-Rigid Clean Store 1**

$$A; \{X_1 : \tilde{s} \triangleq \tilde{q}, X_2 : \tilde{s} \triangleq \tilde{q}\} \cup S; \sigma \Longrightarrow A; \{X_1 : \tilde{s} \triangleq \tilde{q}\} \cup S; \sigma\{X_2 \mapsto X_1\}, \quad \text{if } X_1 \neq X_2.$$

$\mathcal{R}$-CS2: **$\mathcal{R}$-Rigid Clean Store 2**

$$A;\ \{X : \epsilon \triangleq \epsilon\} \cup S;\ \sigma \Longrightarrow A;\ S;\ \sigma\{X \mapsto \epsilon\}$$

To compute $\mathcal{R}$-generalizations of $\tilde{s}$ and $\tilde{q}$, we start with $\{X : \tilde{s} \triangleq \tilde{q}\}; \emptyset; Id$ and apply the rules on the selected anti-unification equation(s) in all possible ways. The obtained procedure is denoted by $\mathfrak{G}(\mathcal{R})$. To show that it terminates, we define the complexity measure for $A; S; \sigma$ as a pair $(M(A), M(S))$, where M is defined as in the termination proof of $\mathfrak{G}$. The measures are compared lexicographically. Each rule strictly reduces it, therefore there can be no infinite transformation chains. All the rules, except $\mathcal{R}$-Dec-H, transform the selected equation(s) uniquely. $\mathcal{R}$-Dec-H can introduce only finitely many branchings, because $\mathcal{R}$ returns a finite set. Hence, the following theorem holds:

▶ **Theorem 4.6.** *The procedure $\mathfrak{G}(\mathcal{R})$ terminates on any input and produces a system $\emptyset; S; \sigma$ where S is irreducible with respect to the store cleaning rules.*

The intuition behind the $\mathcal{R}$-Dec-H rule is that, once $\mathcal{R}$ gives the set of alignments of the strings $top(\tilde{s})$ and $top(\tilde{q})$, we choose one alignment from it, and rigid decomposition is not permitted to be performed on the equations formed by the remaining subsequences of $\tilde{s}$ and $\tilde{q}$ (i.e, the ones that are generalized by Y's in $\mathcal{R}$-Dec-H). Otherwise, the generalization might violate the restrictions of Definition 4.3. Therefore, we move these equations to the store where the decomposition and solve rules do not apply. However, it may introduce certain redundancies in the store. These redundancies are dealt with the store cleaning rules. Another interesting observation is that $\mathfrak{G}(\mathcal{R})$ never introduces in the set A or S equations of the form $x : l \triangleq r$ where x is a term variable.

Since we generalize variable disjoint hedges, the strings in $\mathcal{R}(top(\tilde{s}), top(\tilde{q}))$ (that are common subsequences of $top(\tilde{s})$ and $top(\tilde{q})$) do not contain variables. After application of the rule $\mathcal{R}$-Dec-H, each hedge variable in the anti-unifier gets separated from the other variables by a nonvariable term, to obey the restriction 1 of Definition 4.3.

We did not have the store cleaning rules in our previous algorithm $\mathfrak{G}$, because the AUPs they are dealing with would never appear in the store the rules in $\mathfrak{G}$ are operating on.

Proving soundness of $\mathfrak{G}(\mathcal{R})$ is quite involved, because we should show that the output of $\mathfrak{G}(\mathcal{R})$ satisfies properties of $\mathcal{R}$-generalizations. We need a couple of lemmas for that:

▶ **Lemma 4.7.** *Let $A; S; \vartheta \Longrightarrow_{\mathsf{R}_1} A_1; S_1; \vartheta\sigma_1 \Longrightarrow_{\mathsf{R}_2} A_2; S_2; \vartheta\sigma_1\sigma_2$ be a sequence of transformations where $\mathsf{R}_1 \in \{\mathcal{R}\text{-CS1}, \mathcal{R}\text{-CS2}\}$ and $\mathsf{R}_2 \in \{\mathcal{R}\text{-Dec-H}, \mathcal{R}\text{-S-H}\}$. Then there exists a transformation sequence $A; S; \vartheta \Longrightarrow_{\mathsf{R}_2} A'_1; S'_1; \vartheta\sigma_2 \Longrightarrow_{\mathsf{R}_1} A'_2; S'_2; \vartheta\sigma_2\sigma_1$ such that $A'_2 = A_2$, $S'_2 = S_2$, and $\vartheta\sigma_1\sigma_2 = \vartheta\sigma_2\sigma_1$.*

Proof. Since R_1 does not affect the first component in the system, we have $A_1 = A$ and $A'_2 = A'_1$. We perform the step R_2 in the second transformation sequence exactly in the same way as in the first one, choosing the same rule, the same AUP in A, the same alignment, and the same fresh variables. Then $A'_1 = A_2$ and, hence, $A'_2 = A_2$. As for the stores, S_2 consists of all the AUPs in S except those deleted by R_1 and R_2 and, in addition, it contains the AUPs introduced by R_2. In the second sequence, S'_1 consists of all the AUPs in S except the one deleted by R_2 and the ones introduced by R_2. In the last step, we delete from S'_1 exactly the same AUP that was deleted from S_1 by R_1. Therefore, we get $S'_2 = S_2$. Finally, σ_1 and σ_2 commute, because their domains and ranges are disjoint. Hence, $\vartheta\sigma_1\sigma_2 = \vartheta\sigma_2\sigma_1$. ◀

▶ **Lemma 4.8.** *If $A; S_1; \vartheta \Longrightarrow^* \emptyset; S_2; \vartheta\sigma$ is a derivation in $\mathfrak{G}(\mathcal{R})$ using only $\mathcal{R}$-Dec-H and $\mathcal{R}$-S-H, then for all $(X : \tilde{s} \triangleq \tilde{q}) \in A$, the hedge $X\sigma$ is an $\mathcal{R}$-generalization of $\tilde{s}$ and $\tilde{q}$.*

Proof. We proceed by induction on the length of the derivation. If it is 1, then the derivation has the form $\{X : \tilde{s} \triangleq \tilde{q}\}; S; \vartheta \Longrightarrow^* \emptyset; \{X : \tilde{s} \triangleq \tilde{q}\} \cup S; \vartheta\sigma$, where $\sigma = \{X \mapsto Y_0\}$ for a fresh Y_0 if the used rule is $\mathcal{R}$-Dec-H, and $\sigma = Id$ if the used rule is $\mathcal{R}$-S-H. Since $\mathcal{R}(top(\tilde{s}), top(\tilde{q})) \subseteq \{\epsilon\}$, $X\sigma$ is an $\mathcal{R}$-generalization of $\tilde{s}$ and $\tilde{q}$.

Now we assume that the lemma holds for all derivations with the length less than $m > 1$ and prove it for m. Let the system to be transformed be $\{X : \tilde{s} \triangleq \tilde{q}\} \cup A'; S; \vartheta$. If it is transformed by the rule $\mathcal{R}$-S-H then $\mathcal{R}(top(\tilde{s}), top(\tilde{q})) = \emptyset$, $\sigma = Id$, and we obtain a new system $A'; \{X : \tilde{s} \triangleq \tilde{q}\} \cup S; \vartheta$. By the induction hypothesis, $X'\sigma$ is an $\mathcal{R}$-generalization of $\tilde{s}'$ and $\tilde{q}'$ for all $(X' : \tilde{s}' \triangleq \tilde{q}') \in A'$. By the definition of $\mathcal{R}$-generalization, the same holds for $X\sigma$, $\tilde{s}$, and $\tilde{q}$ because $\mathcal{R}(top(\tilde{s}), top(\tilde{q})) = \emptyset$.

If the rule $\mathcal{R}$-Dec-H is used to transform $\{X : \tilde{s} \triangleq \tilde{q}\} \cup A'; S; \vartheta$, then the new system is $\{Z_k : \tilde{s}_k \triangleq \tilde{q}_k \mid 1 \leq k \leq n\} \cup A'; S'; \vartheta\sigma'$, where $\sigma' = \{X \mapsto Y_0, f_1(Z_1), Y_1, \ldots, Y_{n-1}, f_n(Z_n), Y_n\}$ and the conditions of $\mathcal{R}$-Dec-H are satisfied. By the induction hypothesis, We have a derivation $\{Z_k : \tilde{s}_k \triangleq \tilde{q}_k \mid 1 \leq k \leq n\} \cup A'; S'; \vartheta\sigma' \Longrightarrow^* \emptyset; S''; \vartheta\sigma'\sigma''$ using only the rules $\mathcal{R}$-Dec-H and $\mathcal{R}$-S-H such that for all $(X' : \tilde{s}' \triangleq \tilde{q}') \in \{Z_k : \tilde{s}_k \triangleq \tilde{q}_k \mid 1 \leq k \leq n\} \cup A'$, the hedge $X'\sigma''$ is an $\mathcal{R}$-generalization of $\tilde{s}'$ and $\tilde{q}'$. In particular, this holds for Z's. Therefore, $X\sigma'\sigma''$ is an $\mathcal{R}$-generalization of $\tilde{s}$ and $\tilde{q}$. This finishes the proof. ◀

▶ **Lemma 4.9.** *If $\{X : \tilde{s}_1 \triangleq \tilde{s}_2\}; \emptyset; Id \Longrightarrow^* \emptyset; S_1; \vartheta \Longrightarrow_{\mathsf{R}} \emptyset; S_2; \vartheta\sigma$ is a derivation in $\mathfrak{G}(\mathcal{R})$ such that $X\vartheta$ is an $\mathcal{R}$-generalization of $\tilde{s}_1$ and $\tilde{s}_2$ and $\mathsf{R} \in \{\mathcal{R}\text{-CS1}, \mathcal{R}\text{-CS2}\}$. Then $X\vartheta\sigma$ is an $\mathcal{R}$-generalization of $\tilde{s}_1$ and $\tilde{s}_2$.*

Proof. Let R be $\mathcal{R}$-CS1, transforming $\{X_1 : \tilde{s}'_1 \triangleq \tilde{s}'_2, X_2 : \tilde{s}'_1 \triangleq \tilde{s}'_2\} \subseteq S_1$ into $\{X_1 : \tilde{s}'_1 \triangleq \tilde{s}'_2\} \subseteq S_2$ with the substitution $\sigma = \{X_2 \mapsto X_1\}$. The hedges $\tilde{s}'_1$ and $\tilde{s}'_2$ occur in $\tilde{s}_1$ and $\tilde{s}_2$, respectively, so that the corresponding occurrences are abstracted by the same variable in $X\vartheta$. This variable for some pairs of occurrences of $\tilde{s}'_1$ and $\tilde{s}'_2$ is X_1 and for some others X_2. Hence, if we replace X_2 with X_1 in $X\vartheta$, the obtained hedge $X\vartheta\sigma$ will be a generalization of $\tilde{s}_1$ and $\tilde{s}_2$.

To prove that after this replacement we still have an $\mathcal{R}$-generalization of $\tilde{s}_1$ and $\tilde{s}_2$, we proceed by induction on the maximal depth d of the occurrences of X_2 in $X\vartheta$. It is enough to show that replacing only one occurrence of X_2 with X_1 retains the $\mathcal{R}$-generalization property.

Let first $\mathsf{d} = 0$. Then $X\vartheta$ has a form $\tilde{q}_1, X_2, \tilde{q}_2$. Replacing X_2 with X_1 gives $\tilde{q}_1, X_1, \tilde{q}_2$, that keeps the same alignment from $\mathcal{R}(top(\tilde{s}_1), top(\tilde{s}_2))$ that was in $X\vartheta$ and satisfies all three conditions of the definition of $\mathcal{R}$-generalization. Hence, $\tilde{q}_1, X_1, \tilde{q}_2$ is an $\mathcal{R}$-generalization of $\tilde{s}_1$ and $\tilde{s}_2$.

Now assume that $\mathsf{d} > 0$. It means that there exists a term $f(\tilde{q})$ in $X\vartheta$, such that X_2 occurs at depth $\mathsf{d} - 1$ in $\tilde{q}$. Then there are terms $f(\tilde{s}''_1)$ in $\tilde{s}_1$ and $f(\tilde{s}''_2)$ in $\tilde{s}_2$ such that $\tilde{q}$ is an $\mathcal{R}$-generalization of $\tilde{s}''_1$ and $\tilde{s}''_2$. By the induction hypothesis, replacing an occurrence of X_2 in $\tilde{q}$ with X_1 gives a hedge that is again an $\mathcal{R}$-generalization of $\tilde{s}''_1$ and $\tilde{s}''_2$. Hence, the hedge obtained from $X\vartheta$ by replacing one occurrence of X_2 with X_1 is an $\mathcal{R}$-generalization of $\tilde{s}_1$ and $\tilde{s}_2$, because we just showed that the third condition of the definition of $\mathcal{R}$-generalization is satisfied, while the other two conditions were not affected.

Repeating the process of replacement of one occurrence of X_2 by X_1 iteratively until there are no more X_2's in $X\vartheta$, we prove that $X\vartheta\sigma$ is an $\mathcal{R}$-generalization of $\tilde{s}_1$ and $\tilde{s}_2$.

Proof for $\mathsf{R} = \mathcal{R}$-CS2 is straightforward. ◀

Now we can prove the soundness theorem for $\mathfrak{G}(\mathcal{R})$:

▶ **Theorem 4.10** (Soundness of $\mathfrak{G}(\mathcal{R})$). *If* $\{X : \tilde{s}_1 \triangleq \tilde{s}_2\}; \emptyset; Id \Longrightarrow^* \emptyset; S; \sigma$ *is a derivation in* $\mathfrak{G}(\mathcal{R})$*, then* $X\sigma$ *is an* $\mathcal{R}$*-generalization of* $\tilde{s}_1$ *and* $\tilde{s}_2$*.*

Proof. By Lemma 4.7, every derivation in $\mathfrak{G}(\mathcal{R})$ can be reordered so that first only the rules $\mathcal{R}$-Dec-H and $\mathcal{R}$-S-H are applied until the set of AUPs becomes empty, and then the store is cleaned. The substitutions computed by the original derivation and by the reordered derivation coincide. Let σ' be the substitution obtained at the end of the subderivation with $\mathcal{R}$-Dec-H and $\mathcal{R}$-S-H. By Lemma 4.8, $X\sigma'$ is an $\mathcal{R}$-generalization of $\tilde{s}_1$ and $\tilde{s}_2$. By Lemma 4.9, substitutions introduced by the store cleaning rules keep the $\mathcal{R}$-generalization property. Hence, $X\sigma$ is an $\mathcal{R}$-generalization of $\tilde{s}_1$ and $\tilde{s}_2$. ◀

The algorithm $\mathfrak{G}(\mathcal{R})$ is complete, as the following theorem shows.

▶ **Theorem 4.11** (Completeness of $\mathfrak{G}(\mathcal{R})$). *Let* $\tilde{q}$ *be an* $\mathcal{R}$*-generalization of* $\tilde{s}_1$ *and* $\tilde{s}_2$*. Then* $\mathfrak{G}(\mathcal{R})$ *computes an* $\mathcal{R}$*-anti-unifier* σ *for* $X : \tilde{s}_1 \triangleq \tilde{s}_2$ *such that* $\tilde{q} \preceq X\sigma$*.*

The proof of this theorem is rather long, proceeding by induction on the size of $\tilde{q}$ and by case analysis on its form. It can be found in the technical report [22].

We may prune the search space of the algorithm $\mathfrak{G}(\mathcal{R})$, giving priority to the rules $\mathcal{R}$-CS1 and $\mathcal{R}$-CS2. If they are applicable to a system, no other rule should apply to it. It can prevent re-computing equivalent $\mathcal{R}$-generalizations on different branches without violating completeness. In addition, we may forbid the rule $\mathcal{R}$-Dec-H to add to the set A the AUPs of the form $Z_k : \epsilon \triangleq \epsilon$ for $1 \leq k \leq n$, and to the set S the AUPs of the form $Y_m : \epsilon \triangleq \epsilon$ for $0 \leq m \leq n$. Respectively, such Z_k's and Y_m's are replaced by ϵ in the substitution computed by $\mathcal{R}$-Dec-H. These simplifications can be justified by the fact that those AUPs, anyway, eventually will be eliminated by the $\mathcal{R}$-CS2 rule. Therefore, they do not affect completeness. In the examples below we assume $\mathfrak{G}(\mathcal{R})$ to be optimized in such ways. The length of each derivation under the optimized $\mathfrak{G}(\mathcal{R})$ does not exceed the size of the input problem.

To compute minimal complete set of $\mathcal{R}$-generalizations, we still need to perform the minimization step, unless the cardinality of the set that $\mathcal{R}$ computes is not greater than 1. In the latter case the $\mathfrak{G}(\mathcal{R})$ computes a single $\mathcal{R}$-generalization of the input hedges.

Hence, combining $\mathfrak{G}(\mathcal{R})$ with the minimization step, we can compute $mcg_{\mathcal{R}}(\tilde{s}_1, \tilde{s}_2)$ for any hedges $\tilde{s}_1, \tilde{s}_2$, and the rigidity function $\mathcal{R}$.

▶ **Example 4.12.** Given two terms $f(g(a,a), a, X, b)$ and $f(g(b,b), g(Y), b)$, and $\mathcal{R}$ being the function computing the set of all longest common subsequences, the algorithm $\mathfrak{G}(\mathcal{R})$ gives two $\mathcal{R}$-generalizations: $f(V, g(U), Z, b)$ and $f(g(U), Z, b)$. After the minimization step, only the last one is retained. We illustrate how $\mathfrak{G}(\mathcal{R})$ computes $f(g(U), Z, b)$:

$$\{X_0 : f(g(a,a), a, X, b) \triangleq f(g(b,b), g(Y), b)\}; \emptyset; Id \Longrightarrow_{\mathcal{R}\text{-Dec-H}}$$
$$\{X_1 : g(a,a), a, X, b \triangleq g(b,b), g(Y), b\}; \emptyset; \{X_0 \mapsto f(X_1)\}.$$

This problem is transformed by the $\mathcal{R}$-Dec-H rule with the alignment $g[1,1]b[4,3]$:

$$\{U : a, a \triangleq b, b,\ U' : \epsilon \triangleq \epsilon\}; \{Z : a, X \triangleq g(Y)\}; \{X_0 \mapsto f(g(U), Z, b(U')), \ldots\} \Longrightarrow_{\mathcal{R}\text{-S-H}}$$
$$\{U' : \epsilon \triangleq \epsilon\}; \{U : a, a \triangleq b, b,\ Z : a, X \triangleq g(Y)\}; \{X_0 \mapsto f(g(U), Z, b(U')), \ldots\} \Longrightarrow_{\mathcal{R}\text{-Dec-H}}$$
$$\emptyset; \{U : a, a \triangleq b, b,\ Z : a, X \triangleq g(Y)\}; \{X_0 \mapsto f(g(U), Z, b), \ldots\}.$$

From the final state one can get not only the $\mathcal{R}$-anti-unifier $\{X_0 \mapsto f(g(U), Z, b)\}$ and the corresponding $\mathcal{R}$-generalization $f(g(U), Z, b)$, but also the substitutions that show how the original terms are obtained from the $\mathcal{R}$-generalization. These substitutions can be extracted

from the store: $\sigma_1 = \{U \mapsto a, a,\ Z \mapsto a, X\}$ with $f(g(U), Z, b)\sigma_1 = f(g(a, a), a, X, b)$ and $\sigma_2 = \{U \mapsto b, b,\ Z \mapsto g(Y)\}$ with $f(g(U), Z, b)\sigma_2 = f(g(b, b), g(Y), b)$. In this way, we can also say that the store gives us the difference of the input terms.

▶ **Example 4.13.** Let $\mathcal{R}$ compute the set of all longest common substrings of its arguments and let $a, a, b, f, f, f(a, a, b)$ and $a, a, c, f, f, f(a, a, c)$ be the input hedges. Their only $\mathcal{R}$-generalization $X, f, f, f(a, a, Y)$ can be computed by $\mathfrak{G}(\mathcal{R})$ performing the following steps:

$$\{X_0 : a, a, b, f, f, f(a, a, b) \triangleq a, a, c, f, f, f(a, a, c)\}; \emptyset; Id \Longrightarrow_{\mathcal{R}\text{-Dec-H}}$$
$$\{Y' : a, a, b \triangleq a, a, c\}; \{X : a, a, b \triangleq a, a, c\}; \{X_0 \mapsto X, f, f, f(Y')\} \Longrightarrow_{\mathcal{R}\text{-Dec-H}}$$
$$\emptyset; \{X : a, a, b \triangleq a, a, c,\ Y : b \triangleq c\}; \{X_0 \mapsto X, f, f, f(a, a, Y), \ldots\}.$$

Answer to the Quiz 2: *Given two identical hedges $\tilde{s} = f(a, b, c), g(a), h(a)$ and $\tilde{q} = f(a, b, c), g(a), h(a)$ and $\mathcal{R}$ computing the set of all common subsequences of the minimal length 3 of its arguments, $mcg_{\mathcal{R}}(\tilde{s}, \tilde{q}) = \{f(a, b, c), g(X), h(X)\}$. One might expect the lgg to be the hedge $f(a, b, c), g(a), h(a)$ itself, but it violates the condition 3 of Definition 4.3.*

The example in the Quiz 2 makes it clear why among the $\mathcal{R}$-generalization rules, we do not have the one that would generalize two identical terms with the same term (the so called Trivial Terms rule). It would simply make the $\mathfrak{G}(\mathcal{R})$ algorithm unsound.

Our approach generalizes existing works on word anti-unification. To extend the word anti-unification algorithm from [10] to hedges, one can just take as $\mathcal{R}$ the function that generates the singleton set consisting of the maximal variable-free subsequence in the unique generalization of two words computed in [10]. Similarly, ϵ-free anti-unification for words [6] can be extended to hedges by taking $\mathcal{R}$ as the function that computes the set of all maximal variable-free subsequences of ϵ-free generalizations of the input words.

Precision of rigid anti-unification can be improved, permitting term variables to occur in rigid generalizations. The idea is to generalize AUPs between two term sequences of equal length not with a single hedge variable, but with a sequence of term variables of the same length. This refinement would give $f(x_1, x_2, x_3, x_4, x_5)$ (instead of $f(X)$) as a generalization of $f(a_1, a_2, a_3, a_4, a_5)$ and $f(b_1, b_2, b_3, b_4, b_5)$. This can be achieved by a relatively little computational overhead compared to the $\mathfrak{G}(\mathcal{R})$ algorithm. The details can be found in [22]. Here we only remark that the standard anti-unification over ranked terms [27, 28] can be modeled by such a refinement of $\mathcal{R}$-generalization, choosing $\mathcal{R}$ as the function that returns a singleton set $\mathcal{R}(w_1, w_2) = \{a_1[i_1, i_1] \cdots a_n[i_n, i_n]\}$, where $a_1 \cdots a_n$ is the longest common subsequence of w_1 and w_2 such that all a_is occur at the same positions in w_1 and w_2.

5 Application in Clone Detection

In this section we outline a possible application of $\mathcal{R}$-generalization in software code clone detection. Clone detection is an active research topic since clones are considered to be a significantly problematic issue for software maintenance. Studies show that from 5 to 20% of software systems can contain duplicated code. Due to various complications such duplicated pieces cause, it is widely agreed and the clones should be detected. The survey papers [29, 30] give a detailed characterization of code duplication reasons and drawbacks, introduce clone types, describe and evaluate clone detection process and techniques, and list open problems in clone detection research. The proposed classification distinguishes four types of clones:

Type I: Identical code fragments except for variations in whitespace, layout and comments.

Type II: Syntactically identical fragments except for variations in identifiers, literals, types, whitespace, layout and comments.

Type III: Copied fragments with further modifications such as changed, added or removed statements, in addition to variations in identifiers, literals, types, whitespace, layout and comments.

Type IV: Two or more code fragments that perform the same computation but are implemented by different syntactic variants.

Complexity and sophistication in detecting such clones increases from Type I through Type IV with Type IV being the highest. (Although it does not mean that Type IV contains other types as special cases.) $\mathcal{R}$-generalizations can help in detecting clones of types I-III. We illustrate the idea on an example of a clone of type III.

▶ **Example 5.1.** Type III clones from [29]:

```
if (a >= b)  {                    if (m >= n)
    c = d + b; // Comment1           { // Comment1'
    d = d + 1; }                       y = x + n;
else                                   z = 1; // Added statement
    c = d - a; //Comment2              x = x + 5; //Comment3 }
                                  else
                                     y = x - m; //Comment2'
```

Some clone detection techniques are based on tree representation of the code, like parse trees, abstract syntax trees, or an XML form of abstract syntax trees; see, e.g., [5, 13, 16, 34, 32], for some of the works that follow this approach. We assume that the code is represented in a structural form that can be encoded with unranked terms (or hedges). We keep the representation abstract, without specifying what exactly this structural form is.

Usually, clone detection tools first preprocess the code, then find potential clone candidates, and, finally, analyze them to detect actual clones. One can employ the $\mathcal{R}$-generalization algorithm in the process of finding potential code clones. Further analysis can be based on various measures, like, e.g., on the size of the generalization, or on the maximal length of a nonvariable hedge in the generalization, etc. Although we are not concerned with this part, by choosing appropriate $\mathcal{R}$s we can anticipate this last filtering process. The choice of the $\mathcal{R}$ depends on what is considered as interesting clone.

▶ **Example 5.2.** Unranked term form for the pieces of code in Example 5.1:

$$if(\geq(a,b), then(=(c,+(d,b)),=(d,+(d,1))), else(=(c,-(d,a))))$$
$$if(\geq(m,n), then(=(y,+(x,n)),=(z,1),=(x,+(x,5))), else(=(y,-(x,m))))$$

Let $\mathcal{R}$ be the relation of longest common subsequence. We choose it to capture the idea that the clones have a lot in common. Such an $\mathcal{R}$ is supposed to draw out from two pieces of code as much common statements as possible. Then (the refinement with term variables for) $\mathcal{R}$-generalization of these terms returns three generalizations as clone candidates:

$$if(\geq(x_1,x_2), then(X,=(x_3,x_4),=(x_5,+(x_5,x_6))), else(x_7,-(x_5,x_1))),$$
$$if(\geq(y_1,y_2), then(=(y_3,+(y_4,y_2)),Y,=(y_4,+(y_4,y_5))), else(y_3,-(y_4,y_1))),$$
$$if(\geq(z_1,z_2), then(=(z_3,+(z_4,z_2)),=(z_5,z_6),Z), else(z_3,-(z_4,z_1))).$$

Among them, we say that the second one is the best generalization of the clone pieces, because it preserves the common structure better that the other two (has a bigger size compared to them).

6 Discussion

The standard anti-unification [27, 28] has already been considered for computing software clones in [8, 7], detecting mostly clones of types I and II. However, we think that parameterized anti-unification over unranked terms offers more flexibility in finding clone candidates. First of all, it helps to detect inserted or deleted pieces of code, which is necessary for clones of type III. Besides, if we are interested in clones whose length (as a sequence of program statements) is greater than a predefined threshold, we can include this measure in the definition of the relation $\mathcal{R}$, considering only sequences that are longer than the threshold number. Another advantage of this approach is that it is modular, where most of the computations are performed on strings. It may combine advantages of fast textual and precise structural techniques. For many interesting string relations (e.g., longest common subsequence, longest common substring, sequence alignment, etc.), there exist efficient algorithms that also scale well for large data [14]. Hence, one can take advantage using these off-the-shelf methods when computing clone candidates by $\mathcal{R}$-generalization.

Yet another advantage of using $\mathcal{R}$-generalizations in clone detection is that it works on unranked terms that are natural abstractions of XML documents. How to detect clones well in generated XML/HTML is mentioned as one of the open problems in clone detection research in [29]. A detection technique that uses $\mathcal{R}$-generalization would be an interesting approach to this problem.

Moreover, from the clones computed by $\mathcal{R}$-generalization (anti-unification, in general) one can extract a procedure. This process has a use in code refactoring. The clones can be replaced by the procedure calls, properly instantiated by the substitution that gives from the computed $\mathcal{R}$-generalization the clone it generalizes. As we saw in Example 4.12, these substitutions are easily extracted from the store. In general, while anti-unifiers reflect similarities between two inputs, the data in the store can be used to identify differences between them (i.e., between inputs). This provides for unranked trees a functionality similar, for instance, to one of the well-known comparison utilities (e.g., diff, cmp, fc) that compare the contents of files, finding common contents and differences in them.

The emphasis of this paper is not, however, on clone detection by anti-unification. It can be a topic of separate research where (a) one shows that the $\mathcal{R}$-generalization approach can cover a wide range of currently existing techniques to find similarities between different pieces of code, and (b) presents a clone detection method (and its implementation) fully, starting from code preprocessing till returning the actual interesting clones, where $\mathcal{R}$-generalization performs the task of selecting clone candidates. In this paper we presented the $\mathcal{R}$-anti-unification itself from the theoretical point of view and just tried to indicate some possibilities of its application in clone detection.

We proved properties of $\mathcal{R}$-generalization for a generic $\mathcal{R}$, i.e., for the entire class of rigidity functions. Specializing $\mathcal{R}$ with a particular function, we obtain a specific instance of $\mathcal{R}$-generalization. We saw how certain known generalization problems fall into the class of specific instances of $\mathcal{R}$-generalization in this way.

$\mathcal{R}$-generalization can be made more precise by permitting to generalize term sequences of equal length with a sequence of term variables of the same length, instead of abstracting the original sequences by a single hedge variable. One could think of another extension, to allow a kind of recursive rigid generalization, extending the scope of rigid decomposition rule to the hedges that we currently move to the store, whenever possible. It would require an appropriate revision of the definition of rigid anti-unification.

7 Final Comments

We have presented anti-unification algorithms for unranked terms and hedges, starting from a minimal complete one and then designing a more efficient and flexible version for computing only rigid anti-unifiers. We indicated possible applications of this technique in software code clone detection.

There are a couple of possible directions in future work. One option is to bring in certain higher-order features that can help to further improve the precision of rigid generalizations. An example of such a higher-order extension would be the introduction of function variables. With their help, the algorithm can compute generalizations of the arguments of terms whose heads are distinct. Other interesting directions would be to perform unranked anti-unification in a sorted setting or on compressed terms.

Acknowledgments

This research has been partially supported by the CICyT projects SuRoS (ref. TIN2008-04547/TIN) and TASSAT (ref. TIN2010-20967-C04-01) and by the EC FP6 for Integrated Infrastructures Initiatives under the project SCIEnce (contract No. 026133). The authors thank the anonymous referees for helpful comments.

References

1 M. Alpuente, S. Escobar, J. Meseguer, and P. Ojeda. A modular equational generalization algorithm. In M. Hanus, editor, *LOPSTR*, volume 5438 of *Lecture Notes in Computer Science*, pages 24–39. Springer, 2008.

2 M. Alpuente, S. Escobar, J. Meseguer, and P. Ojeda. Order-sorted generalization. *Electr. Notes Theor. Comput. Sci.*, 246:27–38, 2009.

3 E. Armengol and E. Plaza. Bottom-up induction of feature terms. *Machine Learning*, 41(3):259–294, 2000.

4 F. Baader. Unification, weak unification, upper bound, lower bound, and generalization problems. In R. V. Book, editor, *RTA*, volume 488 of *Lecture Notes in Computer Science*, pages 86–97. Springer, 1991.

5 I. D. Baxter, A. Yahin, L. M. de Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *ICSM*, pages 368–377, 1998.

6 A. Biere. Normalisation, unification and generalisation in free monoids. Master's thesis, University of Karlsruhe, 1993. (in German).

7 P. Bulychev and M. Minea. An evaluation of duplicate code detection using anti-unification. In *Proc. 3rd International Workshop on Software Clones*, 2009.

8 P. E. Bulychev, E. V. Kostylev, and V. A. Zakharov. Anti-unification algorithms and their applications in program analysis. In A. Pnueli, I. Virbitskaite, and A. Voronkov, editors, *Ershov Memorial Conference*, volume 5947 of *Lecture Notes in Computer Science*, pages 413–423. Springer, 2009.

9 J. Burghardt. E-generalization using grammars. *Artif. Intell.*, 165(1):1–35, 2005.

10 I. Cicekli and N. K. Cicekli. Generalizing predicates with string arguments. *Appl. Intell.*, 25(1):23–36, 2006.

11 H. Cirstea, C. Kirchner, R. Kopetz, and P.-E. Moreau. Anti-patterns for rule-based languages. *J. Symb. Comput.*, 45(5):523–550, 2010.

12 A. L. Delcher and S. Kasif. Efficient parallel term matching and anti-unification. *J. Autom. Reasoning*, 9(3):391–406, 1992.

13 W. S. Evans, C. W. Fraser, and F. Ma. Clone detection via structural abstraction. *Software Quality Journal*, 17(4):309–330, 2009.

14 D. Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology*. Cambridge University Press, 1997.

15 G. Huet. *Résolution d'équations dans des langages d'ordre* $1, 2, \ldots, \omega$. PhD thesis, Université Paris VII, September 1976.

16 R. Koschke, R. Falke, and P. Frenzel. Clone detection using abstract syntax suffix trees. In *WCRE*, pages 253–262. IEEE Computer Society, 2006.

17 U. Krumnack, A. Schwering, H. Gust, and K.-U. Kühnberger. Restricted higher-order anti-unification for analogy making. In M. A. Orgun and J. Thornton, editors, *Australian Conference on Artificial Intelligence*, volume 4830 of *Lecture Notes in Computer Science*, pages 273–282. Springer, 2007.

18 T. Kutsia. Solving equations with sequence variables and sequence functions. *J. Symb. Comput.*, 42(3):352–388, 2007.

19 T. Kutsia. Flat matching. *J. Symb. Comput.*, 43(12):858–873, 2008.

20 T. Kutsia, J. Levy, and M. Villaret. Sequence unification through currying. In F. Baader, editor, *RTA*, volume 4533 of *Lecture Notes in Computer Science*, pages 288–302. Springer, 2007.

21 T. Kutsia, J. Levy, and M. Villaret. On the relation between context and sequence unification. *J. Symb. Comput.*, 45(1):74–95, 2010.

22 T. Kutsia, J. Levy, and M. Villaret. Anti-Unification for Unranked Terms and Hedges. Technical Report 11-03, RISC Report Series, University of Linz, Austria, 2011.

23 T. Kutsia and M. Marin. Matching with regular constraints. In G. Sutcliffe and A. Voronkov, editors, *LPAR*, volume 3835 of *Lecture Notes in Computer Science*, pages 215–229. Springer, 2005.

24 T. Kutsia and M. Marin. Order-sorted unification with regular expression sorts. In C. Lynch, editor, *RTA*, volume 6 of *LIPIcs*, pages 193–208. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010.

25 J. Lu, J. Mylopoulos, M. Harao, and M. Hagiya. Higher order generalization and its application in program verification. *Ann. Math. Artif. Intell.*, 28(1-4):107–126, 2000.

26 F. Pfenning. Unification and anti-unification in the calculus of constructions. In *LICS*, pages 74–85. IEEE Computer Society, 1991.

27 G. D. Plotkin. A note on inductive generalization. *Machine Intelligence*, 5(1):153–163, 1970.

28 J. C. Reynolds. Transformational systems and the algebraic structure of atomic formulas. *Machine Intelligence*, 5(1):135–151, 1970.

29 C. K. Roy and J. R. Cordy. A survey of software clone detection research. Technical report, School of Computing, Queen's University at Kingston, Ontario, Canada, 2007.

30 C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Sci. Comput. Program.*, 74(7):470–495, 2009.

31 U. Schmid. *Inductive Synthesis of Functional Programs, Universal Planning, Folding of Finite Programs, and Schema Abstraction by Analogical Reasoning*, volume 2654 of *Lecture Notes in Computer Science*. Springer, 2003.

32 V. Wahler, D. Seipel, J. W. von Gudenberg, and G. Fischer. Clone detection in source code by frequent itemset techniques. In *SCAM*, pages 128–135. IEEE Computer Society, 2004.

33 A. Yamamoto, K. Ito, A. Ishino, and H. Arimura. Modelling semi-structured documents with hedges for deduction and induction. In C. Rouveirol and M. Sebag, editors, *ILP*, volume 2157 of *Lecture Notes in Computer Science*, pages 240–247. Springer, 2001.

34 W. Yang. Identifying syntactic differences between two programs. *Softw., Pract. Exper.*, 21(7):739–755, 1991.

Termination Proofs in the Dependency Pair Framework May Induce Multiple Recursive Derivational Complexity*

Georg Moser[1] **and Andreas Schnabl**[1]

1 **Institute of Computer Science**
University of Innsbruck, Austria
{georg.moser,andreas.schnabl}@uibk.ac.at

Abstract

We study the complexity of rewrite systems shown terminating via the dependency pair framework using processors for reduction pairs, dependency graphs, or the subterm criterion. The complexity of such systems is bounded by a multiple recursive function, provided the complexity induced by the employed base techniques is at most multiple recursive. Moreover this upper bound is tight.

1998 ACM Subject Classification F.2.2, F.4.1, D.2.4, D.2.8

Keywords and phrases Complexity, DP Framework, Multiple Recursive Functions

Digital Object Identifier 10.4230/LIPIcs.RTA.2011.235

Category Regular Research Paper

1 Introduction

Several notions to assess the complexity of a terminating term rewrite system (TRS) have been proposed in the literature, compare [3, 13, 4, 11]. The conceptually simplest one was suggested by Hofbauer and Lautemann in [13]: the *derivational complexity function* with respect to a terminating TRS $\mathcal{R}$ relates the maximal derivation height to the size of the initial term. We adopt this notion as our central definition of the complexity of a TRS. However, we emphasise that our results immediately carry over to other complexity measures of TRSs (compare Section 5). Hence our results are conceivable as an investigation into *implicit computational complexity theory* (see [2] for an overview). To motivate our study consider the following example.

▶ **Example 1.1.** Let $\mathcal{R}_1$ be the TRS defined by the following rules:

$$\mathsf{Ack}(0,y) \to \mathsf{S}(y) \qquad \mathsf{Ack}(\mathsf{S}(x),\mathsf{S}(y)) \to \mathsf{Ack}(x,\mathsf{Ack}(\mathsf{S}(x),y))$$
$$\mathsf{Ack}(\mathsf{S}(x),0) \to \mathsf{Ack}(x,\mathsf{S}(0))$$

The $\mathcal{R}_1$ encodes the Ackermann function. Hence its derivational complexity function grows faster than any primitive recursive functions. Furthermore it is easy to see that the derivational complexity with respect to $\mathcal{R}_1$ is bounded by a multiple recursive function.

* This research is supported by FWF (Austrian Science Fund) project P20133 and a grant of the University of Innsbruck.

22nd International Conference on Rewriting Techniques and Applications (RTA'11).
Editor: M. Schmidt-Schauß; pp. 235–250

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

We show termination of $\mathcal{R}_1$ by an application of the *dependency pair framework* (*DP framework* for short), compare [9, 23]. (All used notions will be defined in Section 2.) The set of dependency pairs $\mathsf{DP}(\mathcal{R}_1)$ with respect to $\mathcal{R}_1$ is given below:

$$\mathsf{Ack}^\sharp(\mathsf{S}(x),0) \to \mathsf{Ack}^\sharp(x,\mathsf{S}(0)) \qquad \mathsf{Ack}^\sharp(\mathsf{S}(x),\mathsf{S}(y)) \to \mathsf{Ack}^\sharp(x,\mathsf{Ack}(\mathsf{S}(x),y))$$
$$\mathsf{Ack}^\sharp(\mathsf{S}(x),\mathsf{S}(y)) \to \mathsf{Ack}^\sharp(\mathsf{S}(x),y)$$

These six rules together constitute the DP problem $(\mathsf{DP}(\mathcal{R}_1),\mathcal{R}_1)$. We apply the subterm criterion processor Φ^{SC} with respect to the simple projection π_1 that projects the first argument of the dependency pair symbol $\mathsf{Ack}^\sharp$. Thus $\Phi^{\mathsf{SC}}((\mathsf{DP}(\mathcal{R}_1),\mathcal{R}_1))$ consists of the single DP problem $(\mathcal{P},\mathcal{R}_1)$, where $\mathcal{P} = \{\mathsf{Ack}^\sharp(\mathsf{S}(x),\mathsf{S}(y)) \to \mathsf{Ack}^\sharp(\mathsf{S}(x),y))\}$. Another application of Φ^{SC}, this time with the simple projection π_2 projecting on the second argument of $\mathsf{Ack}^\sharp$ yields the DP problem $(\varnothing,\mathcal{R}_1)$, which is trivially finite. Thus termination of $\mathcal{R}_1$ follows.

For termination proofs by direct methods a considerable number of results establish essentially optimal upper bounds on the growth rate of the derivational complexity function. For example [25] studies the derivational complexity induced by the lexicographic path order (LPO). LPO induces multiple recursive derivational complexity. In recent years the research focused on automatable proof methods that induce polynomially bounded (derivational) complexity (see for example [26, 18, 24]). The focus and the re-newed interest in this area was partly triggered by the integration of a dedicated complexity category in the annual international termination competition.[1].

None of these results can be applied to our motivating example. Abstracting from Example 1.1 suppose termination of a given TRS $\mathcal{R}$ can be shown via the DP framework in conjunction with a well-defined set of processors. Furthermore assume that all *base techniques* employed in the termination proof (employed within a processor) induce at most multiple recursive derivational complexity. Kindly note that this assumption is rather weak as for all termination techniques whose complexity has been analysed a multiple recursive upper bound exists. Then we show that the derivational complexity with respect to $\mathcal{R}$ is bounded by a multiply recursive function. We restrict our attention to simple DP processors like the *reduction pair processor*, the *dependency graph processor* or the *subterm criterion processor*. Furthermore we show that this upper bound is tight, even if we restrict to base techniques that induce linear derivational complexity. This result can be understood as a negative result: using the above mentioned DP processors, it is theoretically impossible to prove termination of any TRS whose (derivational) complexity is not bounded by a multiply recursive function. One famous example of such a TRS is Dershowitz's system `TRS/D33-33`, aka the Hydra battle rewrite system (see [6, 14]). On the other hand, our result immediately turns termination provers into automatic complexity provers, albeit rather weak ones. Furthermore it provides the basis for further investigations into termination techniques that induced more feasible (derivational) complexities.

The rest of this paper is organised as follows. In Section 2 we present basic notions and starting points of the paper. Section 3 states our main result and provides suitable examples to show that the multiple recursive bound presented is tight. The technical construction is given in Section 4. Finally, we conclude in Section 5. Due to space restriction, some technical proofs have been replaced by sketches. For the full proofs, we refer to the extended version of this paper [17].

[1] `http://termcomp.uibk.ac.at`

2 Preliminaries

We assume familiarity with term rewriting (see [22]) and in particular with the DP method and the DP framework (see [9, 10, 23]). Let $\mathcal{V}$ denote a countably infinite set of variables and $\mathcal{F}$ a signature. Without loss of generality we assume that $\mathcal{F}$ contains at least one constant. The set of (ground) terms over $\mathcal{F}$ and $\mathcal{V}$ is denoted by $\mathcal{T}(\mathcal{F},\mathcal{V})$ ($\mathcal{T}(\mathcal{F})$). The (proper) subterm relation is denoted as $\trianglelefteq$ ($\triangleleft$) and the subterm of t at position p is denoted as $t|_p$. Let t be a term. The *root symbol* of t is denoted as $\mathsf{rt}(t)$; the *positions* of t are denoted as $\mathsf{Pos}(t)$; the *size* (*depth*) of t is denoted as $|t|$ ($\mathsf{dp}(t)$). Let $\mathcal{R}$ and $\mathcal{S}$ be finite TRSs over $\mathcal{F}$. We write $\to_\mathcal{R}$ (or simply $\to$) for the induced rewrite relation. Let $s \xrightarrow{\epsilon}_\mathcal{R} t$ ($s \xrightarrow{\succ\epsilon}_\mathcal{R} t$) denote rewrite steps at (below) the root. We recall the notion of *relative rewriting* [8]. Let $\mathcal{R}$ and $\mathcal{S}$ be finite TRSs over $\mathcal{F}$. The *relative rewrite relation* $\to_{\mathcal{R}/\mathcal{S}}$ is defined as $\to_\mathcal{S}^* \cdot \to_\mathcal{R} \cdot \to_\mathcal{S}^*$. We use $\mathsf{NF}(\mathcal{R})$ to denote the set of normal-forms of $\mathcal{R}$, and $\mathsf{NF}(\mathcal{R}/\mathcal{S})$ for the set of normal forms of $\to_{\mathcal{R}/\mathcal{S}}$. The n-fold composition of $\to$ is denoted as $\to^n$ and the *derivation height* of a term s with respect to a finitely branching, well-founded binary relation $\to$ on terms is defined as $\mathsf{dh}(s,\to) := \max\{n \mid \exists t\ s \to^n t\}$. The *derivational complexity function* of $\mathcal{R}$ is defined as: $\mathsf{dc}_\mathcal{R}(n) = \max\{\mathsf{dh}(t,\to_\mathcal{R}) \mid |t| \leqslant n\}$. Let $\mathcal{R}$ be a TRS and M a termination method. We say M *induces* a certain derivational complexity, if $\mathsf{dc}_\mathcal{R}$ is bounded by a function of this complexity, whenever termination of $\mathcal{R}$ follows by M.

Let t be a term. We set $t^\sharp = t$ if $t \in \mathcal{V}$, and $t^\sharp = f^\sharp(t_1,\dots,t_n)$ if $t = f(t_1,\dots,t_n)$. Here $f^\sharp$ is a new n-ary function symbol called *dependency pair symbol*. The set $\mathsf{DP}(\mathcal{R})$ of *dependency pairs* of $\mathcal{R}$ is defined as $\{l^\sharp \to u^\sharp \mid l \to r \in \mathcal{R}, u \trianglelefteq r$, but $u \not\trianglelefteq l$, and $\mathsf{rt}(u)$ is defined$\}$.[2] A *DP problem* is a pair $(\mathcal{P},\mathcal{R})$, where $\mathcal{P}$ and $\mathcal{R}$ are sets of rewrite rules.[3] It is *finite* if there exists no infinite sequence of rules $s_1 \to t_1, s_2 \to t_2, \dots$ from $\mathcal{P}$ such that for all $i > 0$, t_i is terminating with respect to $\mathcal{R}$, and there exist substitutions σ and τ with $t_i\sigma \to_\mathcal{R}^* s_{i+1}\tau$. A DP problem of the form $(\varnothing,\mathcal{R})$ is trivially finite. We recall the following (well-known) characterisation of termination of a TRS. A TRS $\mathcal{R}$ is terminating if and only if the DP problem $(\mathsf{DP}(\mathcal{R}),\mathcal{R})$ is finite. A *DP processor* is a mapping from DP problems to sets of DP problems. A DP processor Φ is *sound* if for all DP problems $(\mathcal{P},\mathcal{R})$, $(\mathcal{P},\mathcal{R})$ is finite whenever all DP problems in $\Phi((\mathcal{P},\mathcal{R}))$ are finite. A *reduction pair* $(\succcurlyeq,\succ)$ consists of a preorder $\succcurlyeq$ which is closed under contexts and substitutions, and a compatible well-founded order $\succ$ which is closed under substitutions. Here compatibility means the inclusion $\succcurlyeq \cdot \succ \cdot \succcurlyeq \subseteq \succ$. Recall that any well-founded weakly monotone algebra $(\mathcal{A},\succcurlyeq)$ gives rise to a reduction pair $(\succcurlyeq_\mathcal{A},\succ_\mathcal{A})$.

▶ **Proposition 2.1** ([9]). *Let $(\succcurlyeq,\succ)$ be a reduction pair. Then the following DP processor (*reduction pair processor*) Φ^{RP} is sound:*

$$\Phi^{\mathsf{RP}}((\mathcal{P},\mathcal{R})) = \begin{cases} \{(\mathcal{P}',\mathcal{R})\} & \text{if } \mathcal{P}' \cup \mathcal{R} \subseteq \succcurlyeq \text{ and } \mathcal{P} \setminus \mathcal{P}' \subseteq \succ \\ \{(\mathcal{P},\mathcal{R})\} & \text{otherwise .} \end{cases}$$

The *dependency graph* of a DP problem $(\mathcal{P},\mathcal{R})$ (denoted by $\mathsf{DG}(\mathcal{P},\mathcal{R})$) is a graph whose nodes are the elements of $\mathcal{P}$. It contains an edge from $s \to t$ to $u \to v$ whenever there exist substitutions σ and τ such that $t\sigma \to_\mathcal{R}^* u\tau$. A *strongly connected component* (*SCC* for short) of $\mathsf{DG}(\mathcal{P},\mathcal{R})$ is a maximal subset of nodes such that for each pair of nodes $s \to t$, $u \to v$, there exists a path (possibly empty) from $s \to t$ to $u \to v$. Note that this is the

[2] The observation that pairs $l^\sharp \to u^\sharp$ such that $u \triangleleft l$ need not be considered is due to Dershowitz.
[3] We use a simpler definition of DP problems than [9, 23], which suffices in our context.

standard definition of SCC from graph theory (cf. [5], for instance), which slightly differs from the definition that is often used in the termination literature. We call an SCC *trivial* if it consists of a single node $s \to t$ such that the only path from that node to itself is the empty path. All other SCCs are called *nontrivial*.

▶ **Proposition 2.2** ([9]). *The following DP processor (*dependency graph processor*)* Φ^{DG} *is sound:* $\Phi^{\mathsf{DG}}((\mathcal{P},\mathcal{R})) = \{(\mathcal{P}',\mathcal{R}) \mid \mathcal{P}' \text{ is a nontrivial SCC of } \mathsf{DG}(\mathcal{P},\mathcal{R})\}$.

A *simple projection* is an argument filtering π such that for each function symbol $f \in \mathcal{F}$ of arity n, we have $\pi(f) = [1,\dots,n]$, and for each dependency pair symbol $f \in \mathcal{F}^\sharp \setminus \mathcal{F}$, $\pi(f) = i$ for some $1 \leqslant i \leqslant n$.

▶ **Proposition 2.3** ([10, 23]). *Let* π *be a simple projection. Then the following DP processor (*subterm criterion processor*)* Φ^{SC} *is sound:*

$$\Phi^{\mathsf{SC}}((\mathcal{P},\mathcal{R})) = \begin{cases} \{(\mathcal{P}',\mathcal{R})\} & \text{if } \pi(\mathcal{P}') \subseteq \trianglerighteq \text{ and } \pi(\mathcal{P}\setminus\mathcal{P}') \subseteq \triangleright \\ \{(\mathcal{P},\mathcal{R})\} & \text{otherwise .} \end{cases}$$

Let $\mathcal{R}$ be a TRS. A *proof tree* of $\mathcal{R}$ is a tree satisfying the following: the nodes are DP problems, the root is $(\mathsf{DP}(\mathcal{R}),\mathcal{R})$, each leaf is a DP problem of the shape $(\varnothing,\mathcal{R})$, and for each inner node $(\mathcal{P},\mathcal{R}')$, there exists a sound DP processor Φ such that each element of $\Phi((\mathcal{P},\mathcal{R}'))$ is a child of $(\mathcal{P},\mathcal{R}')$, and each of the edges from $(\mathcal{P},\mathcal{R}')$ to the elements of $\Phi((\mathcal{P},\mathcal{R}'))$ is labelled by Φ.

▶ **Theorem 2.4.** *Let* $\mathcal{R}$ *be a TRS such that there exists a proof tree* PT *of* $\mathcal{R}$*. Suppose that each edge label of that proof tree is a reduction pair, dependency graph, or subterm criterion processor. Then* $\mathcal{R}$ *is terminating.*

Furthermore, we recall some essentials of recursion theory, compare [20, 21]. We call the following functions over $\mathbb{N}$ *initial functions*: the constant zero function $z_n(x_1,\dots,x_n) = 0$ of all arities, the unary successor function $s(x) = x + 1$, and all projection functions $\pi^i_n(x_1,\dots,x_n) = x_i$ for $1 \leqslant i \leqslant n$. A class $\mathcal{C}$ of functions over $\mathbb{N}$ is *closed under composition* if for all $f\colon \mathbb{N}^m \to \mathbb{N}$ and $g_1,\dots,g_m\colon \mathbb{N}^n \to \mathbb{N}$ in $\mathcal{C}$, the function $h(x_1,\dots,x_n) = f(g_1(x_1,\dots,x_n),\dots,g_m(x_1,\dots,x_n))$ is in $\mathcal{C}$, as well. It is *closed under primitive recursion* if for all $f\colon \mathbb{N}^n \to \mathbb{N}$ and $g\colon \mathbb{N}^{n+2} \to \mathbb{N}$, the function h defined by $h(0,x_1,\dots,x_n) = f(x_1,\dots,x_n)$ and $h(y+1,x_1,\dots,x_n) = f(y,h(y,x_1,\dots,x_n),x_1,\dots,x_n)$ is contained in $\mathcal{C}$, as well. The *k-ary Ackermann function* A_k for $k \geqslant 2$ is defined recursively as follows:

$$\begin{aligned} A_k(0,\dots,0,x_k) &= x_k + 1 \\ A_k(x_1,\dots,x_{k-2},x_{k-1}+1,0) &= A_k(x_1,\dots,x_{k-1},1) \\ A_k(x_1,\dots,x_{k-2},x_{k-1}+1,x_k+1) &= A_k(x_1,\dots,x_{k-1},A_k(x_1,\dots,x_{k-2},x_{k-1}+1,x_k)) \\ A_k(x_1,\dots,x_{i-1},x_i+1,0,\dots,0,x_k) &= A_k(x_1,\dots,x_i,x_k,0,\dots,0,x_k) \end{aligned}$$

Here, the last equation is a schema instantiated for all $1 \leqslant i \leqslant k-2$. The set of *primitive recursive functions* is the smallest set of functions over $\mathbb{N}$ which contains all initial functions and is closed under composition and primitive recursion. The set of *multiply recursive functions* is the smallest set of functions over $\mathbb{N}$ which contains all initial functions and k-ary Ackermann functions, and is closed under composition and primitive recursion.

▶ **Proposition 2.5** ([21], Chapter 1). *For every multiply recursive function* f*, there exists a* k *such that* A_k *asymptotically dominates* f*.*

3 Main Theorem

In this short section we show that there exist TRSs whose termination is shown via Theorem 2.4 such that the derivational complexity cannot be bounded by a primitive recursive function. Furthermore we state our main result in precise terms.

▶ **Example 3.1.** Consider the following TRS $\mathcal{R}_2$, taken from [13, 12]: $\mathsf{i}(x) \circ (y \circ z) \to x \circ (\mathsf{i}(\mathsf{i}(y)) \circ z)$ and $\mathsf{i}(x) \circ (y \circ (z \circ w)) \to x \circ (z \circ (y \circ w))$. It is shown in [12] that $\mathsf{dc}_{\mathcal{R}_2}$ is not primitive recursive as the system encodes the Ackermann function.

Following Endrullis et al. [7, Example 11] we show termination of $\mathcal{R}_2$ employing Theorem 2.4. The dependency pairs with respect to $\mathcal{R}_2$ are:

$$\begin{array}{lll} 1\colon & \mathsf{i}(x) \circ^\sharp (y \circ z) \to x \circ^\sharp (\mathsf{i}(\mathsf{i}(y)) \circ z) & 2\colon \quad \mathsf{i}(x) \circ^\sharp (y \circ z) \to \mathsf{i}(\mathsf{i}(y)) \circ^\sharp z \\ 3\colon & \mathsf{i}(x) \circ^\sharp (y \circ (z \circ w)) \to x \circ^\sharp (z \circ (y \circ w)) & 4\colon \quad \mathsf{i}(x) \circ^\sharp (y \circ (z \circ w)) \to z \circ^\sharp (y \circ w) \\ 5\colon & \mathsf{i}(x) \circ^\sharp (y \circ (z \circ w)) \to y \circ^\sharp w & \end{array}$$

First, consider the reduction pair induced by the polynomial algebra $\mathcal{A}$ defined as follows: $\circ^\sharp_{\mathcal{A}}(x, y) = y$, $\circ_{\mathcal{A}}(x, y) = y + 1$, and $\mathsf{i}_{\mathcal{A}}(x) = 0$. An application of the processor Φ^{RP} removes the dependency pairs $\{2, 4, 5\}$. Next, we apply the reduction pair induced by the polynomial algebra $\mathcal{B}$ with $\circ^\sharp_{\mathcal{B}}(x, y) = x$, $\circ_{\mathcal{B}}(x, y) = 0$, and $\mathsf{i}_{\mathcal{B}}(x) = x + 1$, which removes the remaining pairs $\{1, 3\}$. Hence we conclude termination of $\mathcal{R}$.

As a corollary we see that the derivational complexity function $\mathsf{dc}_{\mathcal{R}_2}$ is bounded from below by a function that grows faster than any primitive recursive function. On the other hand the complexity induced by the base techniques is linear. Let $\mathcal{P}_1$ denote the set of dependency pairs $\{2, 4, 5\}$ and let $\mathcal{P}_2 = \{1, 3\}$. It is easy to infer from the polynomial algebras $\mathcal{A}$ and $\mathcal{B}$ employed in the two applications of Φ^{RP} that the derivation height functions $\mathsf{dh}(t^\sharp, \to_{\mathcal{P}_1/(\mathcal{P}_2 \cup \mathcal{R})})$ and $\mathsf{dh}(t^\sharp, \to_{\mathcal{P}_2/\mathcal{R}})$ are linear in $|t|$. In order to obtain a tight lower bound we generalise Example 1.1

▶ **Example 3.2.** Let $k \geqslant 2$ and consider the following schematic rewrite rules, denoted as $\mathcal{R}_3^k$. It is easy to see that for fixed k, the TRS $\mathcal{R}_3^k$ encodes the k-ary Ackermann function:

$$\begin{aligned} \mathsf{Ack}_k(0, \ldots, 0, n) &\to \mathsf{S}(n) \\ \mathsf{Ack}_k(l_1, \ldots, l_{k-2}, \mathsf{S}(m), 0) &\to \mathsf{Ack}_k(l_1, \ldots, l_{k-2}, m, \mathsf{S}(0)) \\ \mathsf{Ack}_k(l_1, \ldots, l_{k-2}, \mathsf{S}(m), \mathsf{S}(n)) &\to \mathsf{Ack}_k(l_1, \ldots, l_{k-2}, m, \mathsf{Ack}_k(l_1, \ldots, l_{k-2}, \mathsf{S}(m), n)) \\ \mathsf{Ack}_k(l_1, \ldots, l_{i-1}, \mathsf{S}(l_i), 0, \ldots, 0, n) &\to \mathsf{Ack}_k(l_1, \ldots, l_i, n, 0, \ldots, 0, n) \end{aligned}$$

Here, the last rule is a schema instantiated for all $1 \leqslant i \leqslant k - 2$.

Following of the pattern of the termination proof of $\mathcal{R}_1$, we show termination of $\mathcal{R}_3^k$ by k applications of processor Φ^{SC}. The next lemma is a direct consequence of Proposition 2.5 and the above considerations.

▶ **Lemma 3.3.** *For any multiple recursive function f, there exists a TRS $\mathcal{R}$ whose derivational complexity function $\mathsf{dc}_{\mathcal{R}}$ majorises f. Furthermore termination of $\mathcal{R}$ follows by an application of Theorem 2.4.*

Lemma 3.3 shows that the DP *framework* admits much higher derivational complexities than the basic DP method. In [15, 16] we show that the derivational complexity induced by the DP method is *primitive recursive* in the complexity induced by the base technique, even

if standard refinements like *usable rules* or *dependency graphs* are considered. Examples 3.1 and 3.2 show that we cannot hope to achieve such a bound in the context of the DP framework. In the remainder of this paper we show that jumping to the next function class, the multiple recursive functions, suffices to bound the induced derivational complexities.

▶ **Definition 3.4.** Let $\mathcal{R}$ be a TRS whose termination is shown via Theorem 2.4, and PT the proof tree employed by the theorem. Let k be the maximum number of SCCs in any dependency graph employed by any instance of Φ^{DG} occurring in PT, and let $g\colon \mathbb{N} \to \mathbb{N}$ denote a monotone function such that:

$$g(n) \geqslant \max(\{k\} \cup \{\mathsf{dh}(t^\sharp, \to_{(\mathcal{P}\setminus\mathcal{Q})/(\mathcal{R}\cup\mathcal{Q})}) \mid \text{there exists an edge from } (\mathcal{P},\mathcal{R}) \text{ to } (\mathcal{Q},\mathcal{R}) \text{ in } \mathsf{PT} \text{ labelled by an instance of } \Phi^{\mathsf{RP}} \text{ and } |t| \leqslant n\}) \,.$$

Then g is called a *reduction pair function* of $\mathcal{R}$ with respect to PT.

Note that some reduction pair function can often be computed just by inspection of the employed instances of Φ^{RP}. Moreover, for most of the known reduction pairs (in particular, for virtually all reduction pairs currently applied by automatic termination provers), it is easily possible to compute a multiply recursive reduction pair function.

▶ **Theorem 3.5** (Main Theorem)**.** *Let $\mathcal{R}$ be a TRS whose termination is shown via Theorem 2.4 and let the reduction pair function g of $\mathcal{R}$ be multiple recursive. Then the derivational complexity function $\mathsf{dc}_\mathcal{R}$ with respect to $\mathcal{R}$ is bounded by a multiple recursive function. Furthermore this upper bound is tight.*

The proof of Theorem 3.5 makes use of a combinatorial argument, and is given in the next section. Here we present the proof plan. In proving the theorem we essentially use three different ideas. First, we exploit the given proof tree PT. We observe that, if we restrict our attention to termination of terms, we can focus on specific branches of the proof tree. Secondly, we define a TRS $\mathcal{S}$ simulating the initial TRS $\mathcal{R}$: $s \to_\mathcal{R} t$ implies $\mathsf{tr}(s) \to_\mathcal{S}^+ \mathsf{tr}(t)$. Here tr denotes a suitable interpretation of terms into the signature of the simulating TRS $\mathcal{S}$, compare Definition 4.11. The term $\mathsf{tr}(t)$ aggregates the termination arguments for t given by the DP processors in part of the proof tree which has been identified as particularly relevant for t in the first step. Finally, $\mathcal{S}$ will be simple enough to be compatible with a LPO so that we can employ Weiermann's result in [25] to deduce a multiple recursive upper bound on the derivational complexity with respect to $\mathcal{S}$ and conclusively with respect to $\mathcal{R}$.

Note that our proof technique is conceptually simpler than the technique we used in [15, 16] to show a triple exponential upper complexity bound on the most basic version of the dependency pair method: in order to establish the much lower bound in [15, 16], we constructed the whole proof argument from scratch, while in this paper we exploit Weiermann's analysis of the derivational complexity induced by LPO (see [25]). Still the here presented application of Weiermann's result is non-trivial and requires some preparation.

4 Proof of the Main Result

In this section we prove our main result, Theorem 3.5. We start with some preliminary definitions. Let $\mathcal{R}$ denote a TRS. We assume without loss of generality for each considered termination proof that whenever a DP processor Φ is applied to a DP problem $(\mathcal{P},\mathcal{R})$, then $\Phi((\mathcal{P},\mathcal{R})) \neq \{(\mathcal{P},\mathcal{R})\}$. For each of the DP processors Φ considered in this paper, the following facts are obvious: $(\mathcal{P}',\mathcal{R}') \in \Phi((\mathcal{P},\mathcal{R}))$ implies $\mathcal{P}' \subset \mathcal{P}$ and $\mathcal{R}' = \mathcal{R}$. Therefore, we assume throughout the rest of this paper that for each DP problem $(\mathcal{P},\mathcal{R})$, $\mathcal{P} \subseteq \mathsf{DP}(\mathcal{R})$.

In particular, each rule in $\mathcal{P}$ has the shape $s^\sharp \to t^\sharp$ for some $s,t \in \mathcal{T}(\mathcal{F},\mathcal{V})$. Moreover, $(\mathcal{P}',\mathcal{R}) \in \Phi((\mathcal{P},\mathcal{R}))$, $(\mathcal{P}'',\mathcal{R}) \in \Phi((\mathcal{P},\mathcal{R}))$, and $\mathcal{P}' \neq \mathcal{P}''$ imply $\mathcal{P}' \cap \mathcal{P}'' = \varnothing$. Therefore, each dependency pair can only appear in a single branch of a proof tree.

Let $\mathcal{G}$ be a dependency graph; we order the SCCs of $\mathcal{G}$ by assigning a *rank* to each of them. Let $\mathcal{P}$, $\mathcal{Q}$ denote two distinct SCCs of $\mathcal{G}$. We call $\mathcal{Q}$ *reachable* from $\mathcal{P}$ if there exist nodes $u \in \mathcal{P}$, $v \in \mathcal{Q}$ and a path in $\mathcal{G}$ from u to v. Let k be the number of SCCs in $\mathcal{G}$. Consider a bijection $\mathsf{rk}(\mathcal{G},\cdot)$ from the set of SCCs of $\mathcal{G}$ to $\{1,\dots,k\}$ such that $\mathsf{rk}(\mathcal{G},\mathcal{P}) > \mathsf{rk}(\mathcal{G},\mathcal{Q})$ whenever $\mathcal{Q}$ is reachable from $\mathcal{P}$ in $\mathcal{G}$. We call $\mathsf{rk}(\mathcal{G},\mathcal{P})$ the *rank* of an SCC $\mathcal{P}$ in $\mathcal{G}$. The rank of a dependency pair $l \to r$, denoted by $\mathsf{rk}(\mathcal{G}, l \to r)$, is the rank of $\mathcal{P}$ in $\mathcal{G}$ such that $l \to r \in \mathcal{P}$. Finally, the rank of a term t such that $t^\sharp \notin \mathsf{NF}(\mathcal{P}/\mathcal{R})$ for some SCC $\mathcal{P}$ of $\mathcal{G}$ is defined by $\mathsf{rk}(\mathcal{G},t) := \max\{\mathsf{rk}(\mathcal{G}, l \to r) \mid \exists\sigma\ t^\sharp \to_{\mathcal{R}}^* l\sigma\}$. Observe that $\mathsf{rk}(\mathcal{G},t)$ need not be defined, although t has a redex at the root position. This is due to the fact that this redex need not be governed by a dependency pair. On the other hand observe that if $t \notin \mathsf{NF}(\mathcal{P}/\mathcal{R})$ for some SCC $\mathcal{P}$ of $\mathcal{G}$, then $\mathsf{rk}(\mathcal{G},t)$ is defined. Furthermore in this case $\mathsf{rk}(\mathcal{G},t) > 0$ and $\mathsf{dh}(t^\sharp, \to_{\mathcal{P}/\mathcal{R}}) > 0$.

We now change the definition of proof trees to better suit our needs. The main change is that for Φ^{DG}, all SCCs of the respective dependency graph are taken into account (not just, as usual, the nontrivial ones). While termination of trivial SCCs follows trivially, they might still form a bridge between nontrivial SCCs in a dependency graph thus crucially increasing the length of derivations. Example 4.2 below illustrates this. Moreover, as mentioned above, we use the proof tree to track its currently relevant part with respect to showing termination of a given term. This relevant part may very well include the DP problem belonging to a trivial SCC of a dependency graph.

▶ **Definition 4.1.** We redefine *proof trees*. A proof tree PT of $\mathcal{R}$ is a tree satisfying:

1) The nodes of PT are DP problems and $(\mathsf{DP}(\mathcal{R}),\mathcal{R})$ is the root of PT.
2) For every inner node $(\mathcal{P},\mathcal{R})$ in PT, there exists a sound DP processor Φ such that for each DP problem $(\mathcal{Q},\mathcal{R}) \in \Phi((\mathcal{P},\mathcal{R}))$, there exists an edge from $(\mathcal{P},\mathcal{R})$ to $(\mathcal{Q},\mathcal{R})$ in PT labelled by Φ.
3) Further, suppose $\Phi = \Phi^{\mathsf{DG}}$. Then there exists an edge from $(\mathcal{P},\mathcal{R})$ to a leaf $(\mathcal{Q},\mathcal{R})$ (labelled by Φ) for every trivial SCC $\mathcal{Q}$ of $\mathsf{DG}(\mathcal{P},\mathcal{R})$. Moreover the successors of $(\mathcal{P},\mathcal{R})$ are ordered from left to right in decreasing order with respect to the function rk.

The positions of nodes in PT are defined as usual as finite sequences of numbers. We write Greek letters for positions in PT. It is easy to verify that there is a one-to-one correspondence between proof trees according to Section 2 and Definition 4.1.

▶ **Example 4.2.** Consider the TRS $\mathcal{R}_4$ given by the rewrite rules: $\mathsf{d}(\mathsf{0}) \to \mathsf{0}$, $\mathsf{d}(\mathsf{S}(x)) \to \mathsf{S}(\mathsf{S}(\mathsf{d}(x)))$, $\mathsf{e}(\mathsf{S}(x),y) \to \mathsf{e}(x,\mathsf{d}(y))$, and $\mathsf{sexp}(\mathsf{S}(x),\mathsf{e}(\mathsf{0},y)) \to \mathsf{sexp}(x,\mathsf{e}(y,\mathsf{S}(\mathsf{0})))$. The dependency pairs $\mathsf{DP}(\mathcal{R}_4)$ of $\mathcal{R}_4$ are:

$$1\colon\ \mathsf{d}^\sharp(\mathsf{S}(x)) \to \mathsf{d}^\sharp(x)$$
$$2\colon\ \mathsf{e}^\sharp(\mathsf{S}(x),y) \to \mathsf{d}^\sharp(y) \qquad 3\colon\ \mathsf{e}^\sharp(\mathsf{S}(x),y) \to \mathsf{e}^\sharp(x,\mathsf{d}(y))$$
$$4\colon\ \mathsf{sexp}^\sharp(\mathsf{S}(x),\mathsf{e}(\mathsf{0},y)) \to \mathsf{e}^\sharp(y,\mathsf{S}(\mathsf{0})) \qquad 5\colon\ \mathsf{sexp}^\sharp(\mathsf{S}(x),\mathsf{e}(\mathsf{0},y)) \to \mathsf{sexp}^\sharp(x,\mathsf{e}(y,\mathsf{S}(\mathsf{0})))$$

We start with the dependency graph processor Φ^{DG}. The dependency graph of the initial DP problem $(\mathsf{DP}(\mathcal{R}_4),\mathcal{R}_4)$ contains three nontrivial SCCs $\{1\}$, $\{3\}$, and $\{5\}$, and two trivial SCCs $\{2\}$ and $\{4\}$. Finiteness of each of the nontrivial SCCs can be shown by the reduction pair processor Φ^{RP} employing the following linear polynomial algebra $\mathcal{A}$: $\mathsf{S}_{\mathcal{A}}(x) = x+1$, $\mathsf{0}_{\mathcal{A}} = 0$, $\mathsf{d}_{\mathcal{A}}(x) = 2x$, $\mathsf{e}_{\mathcal{A}}(x,y) = 0$, $\mathsf{sexp}_{\mathcal{A}}(x,y) = 0$, and $\mathsf{d}^\sharp_{\mathcal{A}}(x) = \mathsf{e}^\sharp_{\mathcal{A}}(x,y) = \mathsf{sexp}^\sharp_{\mathcal{A}}(x,y) = x$.

Figure 1 shows the proof tree PT of this termination proof, where we make use of a simplified notation for edge labels. The nodes at positions 11, 31, and 51 are leaves in this proof tree because they are labelled by the DP problem $(\varnothing, \mathcal{R}_4)$, which is trivially finite. The nodes at positions 2 and 4 are leaves because $\{4\}$ and $\{2\}$ are trivial SCCs of the dependency graph employed. The following derivation illustrates the importance of trivial SCCs in our analysis:

$$\mathsf{e}^\sharp(\mathsf{S}^n(0), \mathsf{S}(0)) \to^*_{\{3\}\cup\mathcal{R}_4} \mathsf{e}^\sharp(\mathsf{S}(0), \mathsf{S}^{2^n}(0)) \to_{\{2\}} \mathsf{d}^\sharp(\mathsf{S}^{2^n}(0)) \to^*_{\{1\}\cup\mathcal{R}_4} \mathsf{S}^{2^{n+1}}(0)$$

Observe that the step within the trivial SCC $\{2\}$ connects two subderivations using the (otherwise unconnected) SCCs $\{3\}$ and $\{1\}$, thus increasing the length of the total derivation. In order to capture this behaviour, we keep track of trivial SCCs in our proof trees.

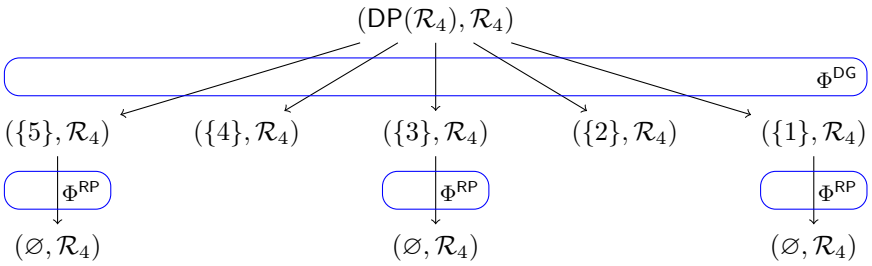


Figure 1 A proof tree of $\mathcal{R}_4$

For the remainder of this section, we assume that termination of $\mathcal{R}$ is shown Theorem 2.4 employing a proof tree PT. Further suppose that there exists a multiply recursive reduction pair function of $\mathcal{R}$, and fix such a function g. Let d be the depth of PT plus one. As stated in the proof plan, we now determine which part of the termination proof is active with respect to a given term. Intuitively, for many terms, only a part of PT is relevant for showing termination of that particular term. More specifically, for any term t, only a certain subset of the dependency pairs can be used for rewriting $t^\sharp$. Of these dependency pairs, we view the one occurring in the leftmost positions of PT (with respect to the order of PT) as the *current dependency pair*. We call the set of positions in which the current dependency pair occurs, the *current path of t in* PT.

▶ **Example 4.3** (continued from Example 4.2)**.** Consider the terms $t_1 = \mathsf{sexp}(\mathsf{S}(0), \mathsf{e}(0, \mathsf{S}(0)))$, $t_2 = \mathsf{sexp}(0, \mathsf{e}(\mathsf{S}(0), \mathsf{S}(0)))$, and $t_3 = \mathsf{e}(\mathsf{S}(0), \mathsf{S}(0))$. We obtain the following derivations: $t_1^\sharp \to_{\mathsf{DP}(\mathcal{R}_4)} t_2^\sharp$ and $t_1^\sharp \to_{\mathsf{DP}(\mathcal{R}_4)} t_3^\sharp$. Hence the term $t_1^\sharp$ is not a normal form with respect to $\{5\}/\mathcal{R}_4$ nor with $\{4\}/\mathcal{R}_4$. Similarly $t_3^\sharp$ is not a normal form with respect to $\{3\}/\mathcal{R}_4$ and $\{2\}/\mathcal{R}_4$. Therefore, the parts of PT highlighted in Figure 2 are particularly relevant for t_1 and t_3, respectively. The term $t_2^\sharp$ is a normal form with respect to $\mathsf{DP}(\mathcal{R}_4)/\mathcal{R}_4$, therefore no part of the proof tree is relevant to show termination for t_2.

The next definition formalises the relevant parts of a proof tree. As mentioned above it suffices to restrict the notion to a single path.

▶ **Definition 4.4.** The *current path* $\mathsf{PT}(t)$ of a term t in PT is defined as follows. If $t^\sharp \in \mathsf{NF}(\mathsf{DP}(\mathcal{R})/\mathcal{R})$, then $\mathsf{PT}(t)$ is the empty path, denoted as $()$. Otherwise, for each dependency pair $l \to r$ such that $t^\sharp \notin \mathsf{NF}(\{l \to r\}/\mathcal{R})$, consider the set of nodes whose label contains $l \to r$. By previous observations, each of these sets forms a path starting at the root node

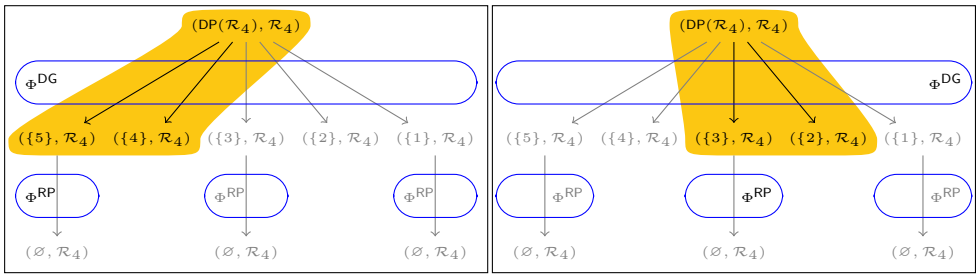


Figure 2 The relevant parts of the proof tree for two terms

of PT. The set of positions forming the leftmost of these paths is $\mathsf{PT}(t)$. We use $\mathsf{PT}_i(t)$ to project on single elements of $\mathsf{PT}(t) = (\alpha_1 = \epsilon, \alpha_2, \ldots, \alpha_n)$: if $i > n$, then $\mathsf{PT}_i(t) = \bot$, otherwise $\mathsf{PT}_i(t) = \alpha_i$.

▶ **Example 4.5** (continued from Example 4.3)**.** The current paths of t_1, t_2, and t_3 are the following: we have $\mathsf{PT}(t_1) = (\epsilon, 1)$, $\mathsf{PT}(t_2) = ()$, and $\mathsf{PT}(t_3) = (\epsilon, 3)$. For t_1, the projections on the single elements of the path are the following: $\mathsf{PT}_1(t_1) = \epsilon$, $\mathsf{PT}_2(t_1) = 1$, and $\mathsf{PT}_i(t_1) = \bot$ for $i > 2$.

Using the DP processors applied to the nodes of $\mathsf{PT}(t)$, we now define the complexity measure $\mathsf{norm}(t)$ assigned to t. For each DP processor, we use whatever value is naturally decreasing in the termination argument of that processor in order to get the associated bound. Given the reduction pair function g, $\mathsf{norm}(t)$ is easily computable.

▶ **Definition 4.6.** We define the mapping $\mathsf{norm}_i\colon \mathcal{T}(\mathcal{F},\mathcal{V}) \to \mathbb{N} \cup \mathcal{T}(\mathcal{F},\mathcal{V}) \cup \{\bot\}$ for $i \in \mathbb{N}$ as follows: let $t \in \mathcal{T}(\mathcal{F},\mathcal{V})$ and $\alpha = \mathsf{PT}_i(t)$.

1) If $\alpha = \bot$, we set $\mathsf{norm}_i(t) = 0$ if $\mathsf{rt}(t)$ is a defined symbol, and $\mathsf{norm}_i(t) = \bot$ otherwise.
2) If $\alpha \neq \bot$ and $(\mathcal{P},\mathcal{R})$ denotes the node at position α in PT such that $(\mathcal{P},\mathcal{R})$ is a leaf, then either $\mathcal{P} = \varnothing$, or $\mathcal{P}$ is a trivial SCC of a dependency graph. In both cases, we set $\mathsf{norm}_i(t) = \mathsf{dh}(t^\sharp, \to_{\mathcal{P}/\mathcal{R}})$.
3) If $\alpha \neq \bot$ and $(\mathcal{P},\mathcal{R})$ denotes the node at position α in PT such that $(\mathcal{P},\mathcal{R})$ is an inner node, then suppose Φ labels each edge starting from $(\mathcal{P},\mathcal{R})$:
   - If Φ is Φ^{RP} with $\Phi((\mathcal{P},\mathcal{R})) = \{(\mathcal{Q},\mathcal{R})\}$, then we set $\mathsf{norm}_i(t) = \mathsf{dh}(t^\sharp, \to_{(\mathcal{P}\setminus\mathcal{Q})/(\mathcal{Q}\cup\mathcal{R})})$.
   - If Φ is Φ^{DG} using a dependency graph $\mathcal{G}$, then we set $\mathsf{norm}_i(t) = \mathsf{rk}(\mathcal{G}, t)$.
   - If Φ is Φ^{SC} using a simple projection π, then we set $\mathsf{norm}_i(t) = \pi(t^\sharp)$.

We extend the mappings norm_i to the *norm* of a term: $\mathsf{norm}(t) = (\mathsf{norm}_1(t), \ldots, \mathsf{norm}_d(t))$.

The central idea behind the complexity measures used in the mapping norm is that rewrite steps induce lexicographical decreases in the norm of the considered term.

▶ **Definition 4.7.** We define the following order $\sqsupset$ on $\mathbb{N} \cup \mathcal{T}(\mathcal{F},\mathcal{V}) \cup \{\bot\}$. We have $a \sqsupset b$ if and only if one of the following properties holds: (i) $a \in \mathbb{N}$, $b \in \mathbb{N}$, and $a > b$, (ii) $a \in \mathcal{T}(\mathcal{F},\mathcal{V})$, $b \in \mathcal{T}(\mathcal{F},\mathcal{V})$, and $a(\to_{\mathcal{R}} \cup \rhd)^+ b$, and (iii) $a \in \mathcal{T}(\mathcal{F},\mathcal{V})$ and $b = 0$, or $a \in \mathcal{T}(\mathcal{F},\mathcal{V}) \cup \mathbb{N}$ and $b = \bot$.

We define $\sqsupseteq$ to be the reflexive closure of $\sqsupset$. We write $\sqsupset^{\mathrm{lex}}$ and $\sqsupseteq^{\mathrm{lex}}$ for the lexicographic extensions of $\sqsupset$ and $\sqsupseteq$, respectively. Note that termination of $\mathcal{R}$ implies well-foundedness of $(\to_{\mathcal{R}} \cup \rhd)^+$, hence $\sqsupset$ is well-founded.

▶ **Lemma 4.8.** *Let s and t be terms such that $s \xrightarrow{\geq\epsilon}_{\mathcal{R}} t$. For all $1 \leqslant i < d$, if $\mathsf{PT}_i(s) = \mathsf{PT}_i(t)$ and $\mathsf{norm}_i(s) = \mathsf{norm}_i(t)$, then either $\mathsf{PT}_{i+1}(t) = \bot$, or $\mathsf{PT}_{i+1}(s) = \mathsf{PT}_{i+1}(t)$.*

Proof Sketch. Straightforward case distinction on the node at position $\mathsf{PT}_i(s)$. ◀

▶ **Lemma 4.9.** *For any terms s and t such that $s \xrightarrow{\geq\epsilon}_{\mathcal{R}} t$, we have $\mathsf{norm}(s) \sqsupseteq^{\mathrm{lex}} \mathsf{norm}(t)$.*

Proof. We can assume that $\mathsf{rt}(t)$ is a defined symbol. Otherwise, $\mathsf{norm}_i(t) = \bot$ for all $1 \leqslant i \leqslant d$, and hence $\mathsf{norm}(t) = (\bot, \ldots, \bot)$, so the lemma would be trivial. As $\mathsf{rt}(s) = \mathsf{rt}(t)$, $\mathsf{rt}(s)$ is also defined. Hence, $s^\sharp \to_{\mathcal{R}} t^\sharp$.

We now show the following by induction on $d-i$: if for all $1 \leqslant j < i$, $\mathsf{norm}_j(s) = \mathsf{norm}_j(t)$, then $(\mathsf{norm}_i(s), \ldots, \mathsf{norm}_d(s)) \sqsupseteq^{\mathrm{lex}} (\mathsf{norm}_i(t), \ldots, \mathsf{norm}_d(t))$. Clearly, this claim implies the lemma, so the remainder of this proof is devoted to it. Applying Lemma 4.8 $i-1$ times reveals that $\mathsf{PT}_i(t)$ is either $\bot$ or the same as $\mathsf{PT}_i(s)$. We perform case distinction on $\mathsf{PT}_i(t)$. We restrict our attention to the interesting case that $\mathsf{PT}_i(s) = \mathsf{PT}_i(t) = \alpha$, $\alpha \neq \bot$, and $(\mathcal{P}, \mathcal{R})$ denotes the node at α in PT such that $(\mathcal{P}, \mathcal{R})$ is an inner node.

Suppose Φ labels the edges starting from $(\mathcal{P}, \mathcal{R})$. If Φ is Φ^{RP} with $\Phi((\mathcal{P}, \mathcal{R})) = \{(\mathcal{Q}, \mathcal{R})\}$, then because of $s^\sharp \to_{\mathcal{R}} t^\sharp$, the inequality $\mathsf{dh}(s^\sharp, \to_{(\mathcal{P}\setminus\mathcal{Q})/(\mathcal{Q}\cup\mathcal{R})}) \geqslant \mathsf{dh}(t^\sharp, \to_{(\mathcal{P}\setminus\mathcal{Q})/(\mathcal{Q}\cup\mathcal{R})})$. Thus $\mathsf{norm}_i(s) \sqsupseteq \mathsf{norm}_i(t)$ holds. If Φ is Φ^{DG} using a dependency graph $\mathcal{G}$, then for each SCC $\mathcal{P}_j$ in $\mathcal{G}$, $s^\sharp$ can only be a normal form of $\mathcal{P}_j/\mathcal{R}$ if $t^\sharp$ is one, as well. Therefore, we have $\mathsf{norm}_i(s) \sqsupseteq \mathsf{norm}_i(t)$ in that case, too. If Φ is Φ^{SC} with simple projection π, then $\mathsf{norm}_i(s) = \pi(s^\sharp) \to_{\mathcal{R}}^{=} \pi(t^\sharp) = \mathsf{norm}_i(t)$, and hence $\mathsf{norm}_i(s) \sqsupseteq \mathsf{norm}_i(t)$.

So, regardless of the processor Φ, we have $\mathsf{norm}_i(s) \sqsupseteq \mathsf{norm}_i(t)$. If $\mathsf{norm}_i(s) \sqsupset \mathsf{norm}_i(t)$, then the claim trivially follows. On the other hand, if $\mathsf{norm}_i(s) = \mathsf{norm}_i(t)$, then the claim holds by induction hypothesis. ◀

The following lemma extends Lemma 4.9 to root steps $s \xrightarrow{\epsilon}_{\mathcal{R}} t$. However, in this case, we do not consider only the root position of t, but all positions that were "created" by the rewrite step. So essentially, we show that such a step causes a decrease in $\sqsupset^{\mathrm{lex}}$ from s to subterms of t. The restriction on positions p below takes care of the Dershowitz condition in the definition of dependency pairs and the substitution of the applied rewrite rule.

▶ **Lemma 4.10.** *For any terms s and t such that $s \xrightarrow{\epsilon}_{\mathcal{R}} t$, we have $\mathsf{norm}(s) \sqsupset^{\mathrm{lex}} \mathsf{norm}(t|_p)$ for all $p \in \mathsf{Pos}(t)$ such that $t|_p \not\trianglelefteq s$.*

Proof. For this proof, we fix p, and let $u = t|_p$. We can assume that $\mathsf{rt}(u)$ is a defined symbol. Otherwise, $\mathsf{norm}(u) = (\bot, \ldots, \bot)$, but $\mathsf{norm}(s) \sqsupseteq^{\mathrm{lex}} (0, \ldots, 0)$ (note that $\mathsf{rt}(s)$ is defined), so the lemma would be immediate. Hence, we have $s^\sharp \to_{\mathsf{DP}(\mathcal{R})} u^\sharp$ using some dependency pair $l \to r$. Let j be the greatest number between 1 and d such that $\mathsf{PT}_j(s) \neq \bot$, the node at $\mathsf{PT}_j(s)$ is $(\mathcal{Q}, \mathcal{R})$, and $\mathcal{Q}$ contains $l \to r$. Note that such a number exists: since $s^\sharp \to_{\mathsf{DP}(\mathcal{R})} u^\sharp$, we have $\mathsf{PT}_1(s) = \epsilon$, which denotes $(\mathsf{DP}(\mathcal{R}), \mathcal{R})$, and $\mathsf{DP}(\mathcal{R})$ contains $l \to r$. Let $\alpha = \mathsf{PT}_j(s)$. We distinguish whether $\mathsf{PT}_j(u) = \alpha$. This determines whether the strict part of the lexicographic decrease must happen at index j or at an earlier index.

Suppose $\mathsf{PT}_j(u) = \alpha$. Then we show that for all $1 \leqslant i \leqslant j$, $\mathsf{norm}_i(s) \sqsupseteq \mathsf{norm}_i(u)$ holds, and $\mathsf{norm}_j(s) \sqsupset \mathsf{norm}_j(u)$. From these two properties, the lemma follows. In order to show them, we fix some $1 \leqslant i \leqslant j$. Let $\beta = \mathsf{PT}_i(s) = \mathsf{PT}_i(u)$.

1) If the node at position β is a leaf of PT, then $i = j$, and $\mathcal{Q}$ is a trivial SCC of a dependency graph. By assumption, $l \to r \in \mathcal{Q}$. Therefore, $\mathsf{dh}(s^\sharp, \to_{\mathcal{Q}/\mathcal{R}}) > \mathsf{dh}(u^\sharp, \to_{\mathcal{Q}/\mathcal{R}})$, and thus $\mathsf{norm}_i(s) \sqsupset \mathsf{norm}_i(u)$.
2) If the node $(\mathcal{P}, \mathcal{R})$ at position β is an inner node of PT, let Φ be the label of each edge starting from $(\mathcal{P}, \mathcal{R})$. Obviously, $\mathcal{Q} \subseteq \mathcal{P}$, and therefore $l \to r \in \mathcal{P}$. For all possibilities of Φ, the semantics of Φ imply that $\mathsf{norm}_i(s) \sqsupseteq \mathsf{norm}_i(u)$. Moreover, if $i = j$, then $\mathsf{norm}_i(s) \sqsupset \mathsf{norm}_i(u)$ follows. In more detail: If Φ is Φ^{RP}, then let $\{(\mathcal{P}', \mathcal{R})\} = \Phi((\mathcal{P}, \mathcal{R}))$. Since $l \to r \in \mathcal{P}$, it follows that $\mathsf{dh}(s^\sharp, \to_{(\mathcal{P}\setminus\mathcal{P}')/(\mathcal{P}'\cup\mathcal{R})}) \geqslant \mathsf{dh}(u^\sharp, \to_{(\mathcal{P}\setminus\mathcal{P}')/(\mathcal{P}'\cup\mathcal{R})})$, and

thus $\mathsf{norm}_i(s) \sqsupseteq \mathsf{norm}_i(u)$. If $i = j$, then by definition of j, $l \to r$ is contained in $\mathcal{P} \setminus \mathcal{P}'$. Therefore, $\mathsf{norm}_i(s) \sqsupset \mathsf{norm}_i(u)$ in that case. If Φ is Φ^{DG} using a dependency graph $\mathcal{G}$, then by definition of SCCs in a dependency graph, $\mathsf{rk}(\mathcal{G}, s) \geqslant \mathsf{rk}(\mathcal{G}, l \to r) \geqslant \mathsf{rk}(\mathcal{G}, u)$, hence $\mathsf{norm}_i(s) \sqsupseteq \mathsf{norm}_i(u)$. If $i = j$, then by definition of j, $\mathsf{rk}(\mathcal{G}, s) \neq \mathsf{rk}(\mathcal{G}, l \to r)$. Thus, $\mathsf{norm}_i(s) \sqsupset \mathsf{norm}_i(u)$ in that case. If Φ is Φ^{SC} with $\Phi((\mathcal{P}, \mathcal{R})) = (\mathcal{P}', \mathcal{R})$ and simple projection π, then $\pi(s^\sharp) \trianglerighteq \pi(u^\sharp)$, and hence $\mathsf{norm}_i(s) \sqsupseteq \mathsf{norm}_i(u)$. If $i = j$, then by definition of j, $l \to r \in \mathcal{P} \setminus \mathcal{P}'$, and hence $\pi(s^\sharp) \triangleright \pi(u^\sharp)$ and $\mathsf{norm}_i(s) \sqsupset \mathsf{norm}_i(u)$ in that case.

In all cases, it follows that for all $1 \leqslant i \leqslant j$, $\mathsf{norm}_i(s) \sqsupseteq \mathsf{norm}_i(u)$ holds, and $\mathsf{norm}_j(s) \sqsupset \mathsf{norm}_j(u)$.

Now suppose $\mathsf{PT}_j(u) \neq \alpha$. Then let i be the greatest number between 1 and j such that $\mathsf{PT}_i(s) = \mathsf{PT}_i(u) = \beta$. As β is a prefix of α, the node $(\mathcal{P}, \mathcal{R})$ at β is an inner node of PT. Let Φ be the label of each edge starting from $(\mathcal{P}, \mathcal{R})$. Using the arguments from above, we see that $\mathsf{norm}_{i'}(s) \sqsupseteq \mathsf{norm}_{i'}(u)$ for all $1 \leqslant i' \leqslant i$. We now show that $\mathsf{norm}_i(s) \sqsupset \mathsf{norm}_i(u)$ or $\mathsf{norm}_{i+1}(s) \sqsupset \mathsf{norm}_{i+1}(u)$ holds.

1) If Φ is Φ^{RP} or Φ^{SC}, then by our assumptions, $\mathsf{PT}_{i+1}(s) = \beta 1$. Since β has only one child in this case, this implies $\mathsf{PT}_{i+1}(u) = \bot$. Thus, $\mathsf{norm}_{i+1}(s) > 0 = \mathsf{norm}_{i+1}(u)$.

2) If Φ is Φ^{DG}, then $\mathsf{norm}_i(s) \neq \mathsf{norm}_i(u)$, since $\mathsf{PT}_{i+1}(s) \neq \mathsf{PT}_{i+1}(u)$ by assumption. Thus $\mathsf{norm}_i(s) \sqsupset \mathsf{norm}_i(u)$.

In both cases, it follows that $\mathsf{norm}(s) \sqsupset^{\mathrm{lex}} \mathsf{norm}(u)$, which is what we wanted to show. ◀

Up to now, we have shown norm decreases under rewriting. For rewrite steps whose redex position is at the root, this decrease is even strict. In order to turn this into an upper bound on derivational complexities, we still have to do some work: we also have to consider the norm of all proper subterms of a considered term, and the range of norm is not suitable for direct complexity measures yet. We now solve these problems by lifting the range of norm to the term level and simulating derivations of $\mathcal{R}$ at that level.

For the rest of this section let A be the maximum arity of any function symbol occurring in $\mathcal{R}$, and $C := \max\{\mathsf{dp}(r) \mid l \to r \in \mathcal{R}\}$. Depending on PT, d, A, C, and g, we now define a TRS $\mathcal{S}$ which simulates $\mathcal{R}$ and is compatible with LPO. The simulating TRS $\mathcal{S}$ is based on a mapping tr (see Definition 4.13) such that $s \to_{\mathcal{R}} t$ implies $\mathsf{tr}(s) \to_{\mathcal{S}}^{+} \mathsf{tr}(t)$. Given a term t, tr employs the $d + A$-ary function symbol f. The first d arguments of f are used to represent $\mathsf{norm}(t)$; the last a arguments of f are used to represent $\mathsf{tr}(t')$ for each direct subterm t' of t. In the simulation, we often have to recalculate the first d arguments of each f. Due to the definition of norm, we know that for each term t and $1 \leqslant i \leqslant d$, either $\mathsf{norm}_i(t) \in \mathbb{N}$ and $\mathsf{norm}_i(t) \leqslant g(|t|)$, or $\mathsf{norm}_i(t) \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ and $\mathsf{norm}_i(t) \trianglelefteq t$, or $\mathsf{norm}_i(t) = \bot$. We use a unary function symbol choice such that $\mathsf{choice}(\mathsf{tr}(t))$ rewrites to the representations of $g(|t|)$, $\mathsf{tr}(t')$ for each subterm t' of t, and $\bot$. In particular we often we use terms of the shape $\mathsf{choice}(\mathsf{f}(\mathsf{0}, \dots, \mathsf{0}, x_1, \dots, x_A))$ in the definition of $\mathcal{S}$, so we use the abbreviation $N(x_1, \dots, x_A)$ for this.

The main tool for achieving the simulation of a root rewrite step $s \xrightarrow{\epsilon}_{\mathcal{R}} t$ are rules which build the new f symbols for the positions in t "created" by the step. These are at most A^{C+1} many new positions, and each proper subterm of s may be duplicated at most that many times. As a very simple example, if $d = 3$, $A = 1$, and $C = 1$, this behaviour is simulated

by rules of the following shape:

$$\begin{aligned}
\mathsf{f}(u_1, \mathsf{S}(u_2), u_3, x) &\to \mathsf{f}(u_1, u_2, N(x), \mathsf{f}(u_1, u_2, N(x), x)) \\
\mathsf{f}(u_1, \mathsf{f}(v_1, v_2, v_3, y), u_3, x) &\to \mathsf{f}(u_1, y, N(x), \mathsf{f}(u_1, y, N(x), x)) \\
\mathsf{f}(u_1, \mathsf{f}(v_1, v_2, v_3, y), u_3, x) &\to \mathsf{f}(u_1, \mathsf{0}, N(x), \mathsf{f}(u_1, \mathsf{0}, N(x), x)) \\
\mathsf{f}(u_1, \mathsf{0}, u_3, x) &\to \mathsf{f}(u_1, \bot, N(x), \mathsf{f}(u_1, \bot, N(x), x))
\end{aligned}$$

We use similar rules for decreases of u_1 or u_3 with respect to the ordering $\sqsupset$. In order to write down these rules concisely for arbitrary A and C, we make use of the following abbreviation M_i^k (for $i \in \{1, \dots, d\}$ and $k \in \mathbb{N}$):

$$\begin{aligned}
&M_i^0(u_1, \dots, u_i, x_1, \dots, x_A) = \mathsf{f}(u_1, \dots, u_i, \overline{N(x_1, \dots, x_A)}, x_1, \dots, x_A) \\
&M_i^{k+1}(u_1, \dots, u_i, x_1, \dots, x_A) \\
&\quad = \mathsf{f}(u_1, \dots, u_i, \overline{N(\overline{M_i^k(u_1, \dots, u_i, x_1, \dots, x_A)})}, \overline{M_i^k(u_1, \dots, u_i, x_1, \dots, x_A)})
\end{aligned}$$

Here u_i ($i \in \{1, \dots, i\}$) and x_j ($j \in \{1, \dots, A\}$) denote variables and $\bar{t}$ is an abbreviation of $t, \dots, t$, where the number of repetitions of t follows from the context.

Consider the reduction pair function g of $\mathcal{R}$. Since g is assumed to be a multiple recursive function, it is an easy exercise to define a TRS $\mathcal{S}'$ (employing the constructors S, $\mathsf{0}$) that computes the function g: one can simply define g using only initial functions, composition, primitive recursion, and k-ary Ackermann functions, and directly turn the resulting definition of g into rewrite rules. That is, there exists a TRS $\mathcal{S}'$ and a defined function symbol g such that $\mathsf{g}(\mathsf{S}^n(\mathsf{0})) \to^*_{\mathcal{S}'} \mathsf{S}^{g(n)}(\mathsf{0})$. Here we use $\mathsf{S}^n(\mathsf{0})$ to denote $\mathsf{S}(\dots(\mathsf{S}(\mathsf{0}))\dots)$, where S is repeated n times. Moreover, $\mathcal{S}'$ is compatible with a LPO such that the precedence $\succ$ of the LPO includes $\mathsf{g} \succ \mathsf{S} \succ \mathsf{0}$.

▶ **Definition 4.11.** Consider the following (schematic) TRS $\mathcal{S}$, where $1 \leqslant i \leqslant d$ and $1 \leqslant j \leqslant A$. Here we use $\vec{x}$ as a shorthand for $x_1, \dots, x_A$.

$$\begin{aligned}
1_i&: && \mathsf{f}(u_1, \dots, u_{i-1}, \mathsf{S}(u_i), u_{i+1}, \dots, u_d, \vec{x}) \to M_i^C(u_1, \dots, u_i, \vec{x}) \\
2_{i,j}&: && \mathsf{f}(u_1, \dots, u_{i-1}, \mathsf{f}(v_1, \dots, v_d, \vec{y}), u_{i+1}, \dots, u_d, \vec{x}) \to M_i^C(u_1, \dots, u_{i-1}, y_j, \vec{x}) \\
3_i&: && \mathsf{f}(u_1, \dots, u_{i-1}, \mathsf{f}(v_1, \dots, v_d, \vec{y}), u_{i+1}, \dots, u_d, \vec{x}) \to M_i^C(u_1, \dots, u_{i-1}, \mathsf{0}, \vec{x}) \\
4_i&: && \mathsf{f}(u_1, \dots, u_{i-1}, \mathsf{0}, u_{i+1}, \dots, u_d, \vec{x}) \to M_i^C(u_1, \dots, u_{i-1}, \bot, \vec{x}) \\
5_j&: && \mathsf{size}(\mathsf{f}(u_1, \dots, u_d, \vec{x})) \to \times_A(\mathsf{size}(x_j)) \\
6&: && \mathsf{size}(\mathsf{c}) \to \mathsf{S}(\mathsf{0}) \\
7&: && \times_A(\mathsf{S}(x)) \to \mathsf{S}^A(\times_A(x)) \\
8&: && \times_A(\mathsf{0}) \to \mathsf{0} \\
9&: && \mathsf{f}(u_1, \dots, u_d, \vec{x}) \to \mathsf{c} \\
10_j&: && \mathsf{f}(u_1, \dots, u_d, \vec{x}) \to x_j \\
11&: && \mathsf{h}(x) \to \mathsf{f}(\overline{N(\overline{x})}, \overline{x}) \\
12&: && \mathsf{z} \to \mathsf{f}(\overline{N(\overline{\mathsf{c}})}, \overline{\mathsf{c}}) \\
13_j&: && \mathsf{choice}(\mathsf{f}(u_1, \dots, u_d, \vec{x})) \to x_j \\
14&: && \mathsf{choice}(x) \to \mathsf{g}(\mathsf{size}(x)) \\
15&: && \mathsf{choice}(x) \to \bot
\end{aligned}$$

These rules are augmented by $\mathcal{S}'$ defining the function symbol g. The signatures of $\mathcal{S}'$ and $\mathcal{S} \setminus \mathcal{S}'$ are disjoint with the exception of g and the constructors S and $\mathsf{0}$.

Note that $\mathcal{S}$ depends only on the constants d, A, C, and the reduction pair function g. The rules 1_i–4_i are the main rules for the simulation of the effects of a single rewrite step $s \xrightarrow{\epsilon}_{\mathcal{R}} t$ in $\mathcal{S}$. These rules employ that $\mathsf{norm}_i(s^\sharp) \sqsupset \mathsf{norm}_i((t|_p)^\sharp)$ for all $p \in \mathsf{Pos}(t)$ such that $t|_p \not\trianglelefteq s$, and $\mathsf{norm}_{i'}(s^\sharp) \sqsupseteq \mathsf{norm}_{i'}((t|_p)^\sharp)$ for all $1 \leqslant i' \leqslant i$. They are also responsible for creating the at most A^{C+1} many new positions and copies of each subterm of s in t. The rules 5_j–8 define a function symbol size, that is, $\mathsf{size}(s)$ reduces to a numeral $\mathsf{S}^n(\mathsf{0})$ such that $n \geqslant |s|$. The rules 9–10_j make sure that any superfluous positions and copies of subterms created by the rules of type 1_i–4_i can be deleted. The rules 11 and 12 guarantee that the simulating derivation can be started with a small enough initial term. The rules 13_j–15 define the function symbol choice introduced in the abbreviations M_i^C, and N. Loosely speaking, $\mathsf{choice}(t)$ is an upper bound of all $\mathsf{norm}_i(t)$ with respect to $\sqsupseteq$.

The next lemma essentially follows from Weiermann's result that LPO induces multiple recursive derivational complexity.

▶ **Lemma 4.12.** *The function $\mathsf{dc}_{\mathcal{S}}$ is multiply recursive.*

Proof. The TRS $\mathcal{S}'$ computing g can be shown terminating using an LPO such that the precedence $\succ$ of the LPO contains $\mathsf{g} \succ \mathsf{S} \succ \mathsf{0}$. It is easy to check that extending this precedence by

$$\mathsf{h}, \mathsf{z} \succ \mathsf{f} \succ \mathsf{choice} \succ \mathsf{g} \qquad \mathsf{size} \succ \mathsf{S} \succ \times, \mathsf{0}, \mathsf{c}, \bot\,,$$

makes the whole TRS $\mathcal{S}$ compatible with this LPO. By [25], termination of a finite TRS by an LPO implies that the derivational complexity of that TRS is multiple recursive. Note that the definition of multiple recursion used in this paper and the definition given in [25] coincide by [19]. Thus, $\mathsf{dc}_{\mathcal{S}}$ is a multiple recursive function. ◀

For the remainder of this section, let $\mathcal{H}$ denote the signature of $\mathcal{S}$. We now show that the TRS $\mathcal{S}$ indeed simulates $\mathcal{R}$ as requested. Since the proofs of the lemmas in this part of the paper are rather straightforward, but very technical. for them.

▶ **Definition 4.13.** The mapping $\mathsf{tr}\colon \mathcal{T}(\mathcal{F}) \to \mathcal{T}(\mathcal{H})$ is defined by the equation $\mathsf{tr}(t) = \mathsf{f}((\mathsf{norm}_1(t))^*, \dots, (\mathsf{norm}_d(t))^*, \mathsf{tr}(t_1), \dots, \mathsf{tr}(t_n), \mathsf{c}, \dots, \mathsf{c})$, where $t = f(t_1, \dots, t_n)$ and the operator $(\cdot)^*$ is defined for a term s as follows:

$$u^* := \begin{cases} \bot & \text{if } u = \bot \\ \mathsf{S}^u(\mathsf{0}) & \text{if } u \in \mathbb{N} \\ \mathsf{tr}(u) & \text{if } u \in \mathcal{T}(\mathcal{F}, \mathcal{V}) \end{cases}$$

We define an equivalence $s \approx t$ on $\mathcal{T}(\mathcal{H})$. If $s = \mathsf{c}$, then $t = \mathsf{c}$. Otherwise if $s = \mathsf{f}(u_1, \dots, u_d, s_1, \dots, s_A)$, then $t = \mathsf{f}(v_1, \dots, v_d, t_1, \dots, t_A)$ such that $s_i \approx t_i$ for all $1 \leqslant i \leqslant A$.

▶ **Lemma 4.14.** *For all ground terms s with $t \approx \mathsf{tr}(s)$, $\mathsf{size}(t) \to_{\mathcal{S}}^+ \mathsf{S}^n(\mathsf{0})$ where $n \geqslant |s|$.*

Proof Sketch. The proof uses induction on $|s|$. The main ingredient of the inductive argument is straightforward application of the rules 5_j–8 of $\mathcal{S}$. ◀

▶ **Lemma 4.15.** *The following properties of $\mathcal{S}$ hold:*

1) *If $s = \mathsf{f}(u_1^*, \dots, u_d^*, \vec{s})$, $t = \mathsf{tr}(t') = \mathsf{f}(v_1^*, \dots, v_d^*, \vec{s})$, and $(u_1, \dots, u_d) \sqsupset^{\mathrm{lex}} (v_1, \dots, v_d)$, then $s \to_{\mathcal{S}}^+ t$.*
2) *For any ground terms $s = \mathsf{tr}(s')$ and $t = \mathsf{tr}(t')$, $s' \xrightarrow{\epsilon}_{\mathcal{R}} t'$ implies $s \to_{\mathcal{S}}^+ t$.*

3) *If* $a \to_{\mathcal{R}} b$ *and* $\mathsf{tr}(a) \to_{\mathcal{S}}^{+} \mathsf{tr}(b)$*, then for any* n*-ary function symbol* $f \in \mathcal{F}$*, we have* $s = \mathsf{tr}(f(t_1, \ldots, a, \ldots, t_n)) \to_{\mathcal{S}}^{+} \mathsf{tr}(f(t_1, \ldots, b, \ldots, t_n))$.

Proof Sketch. The proof uses mutual induction on $\mathsf{dh}(s, \to_{\mathcal{S}} \cup \rhd)$. Note that by Lemma 4.12, $\mathcal{S}$ terminates, and hence $\to_{\mathcal{S}} \cup \rhd$ is well-founded. The following are the central parts of the inductive arguments for the three properties of the lemma:

1) Property 1 states that all lexicographic decreases in the norm of a term occurring during a simulation can indeed be handled by $\mathcal{S}$. The main ingredient for this part of the proof is the application of the rules 1_i–4_i of $\mathcal{S}$.
2) Property 2 states that the simulation of a root rewrite step can indeed be done within $\mathcal{S}$. The outline of this part of the proof is the following: The first step of the simulation is an application of one of the rules 1_i–4_i. This yields a rewrite step of the shape

   $$s \to_{\mathcal{S}} M_i^C((\mathsf{norm}_1(s'))^*, \ldots, (\mathsf{norm}_{i-1}(s'))^*, v_i'^*, s_1, \ldots, s_n)$$

   such that $v_i' \sqsupseteq \mathsf{norm}_i(s')$. The proof then proceeds to show by side induction on $\mathsf{dp}(u)$ the following for subterms u of t', which concludes Property 2 of the lemma:

   $$M_i^{\mathsf{dp}(u)}((\mathsf{norm}_1(s'))^*, \ldots, (\mathsf{norm}_{i-1}(s'))^*, v_i'^*, s_1, \ldots, s_n) \to_{\mathcal{S}}^{*} \mathsf{tr}(u) \ .$$

   The most important argument within the side induction is the application of Lemma 4.10.
3) Property 3 essentially states that Property 2 is closed under contexts. The main ingredient for this part of the proof is the application of Lemma 4.9.

◀

The next lemma is an easy consequence of Lemma 4.15(2) and (3).

▶ **Lemma 4.16.** *For any ground terms* s *and* t*,* $s \to_{\mathcal{R}} t$ *implies* $\mathsf{tr}(s) \to_{\mathcal{S}}^{+} \mathsf{tr}(t)$.

Lemma 4.16 yields that the length of any derivation in $\mathcal{R}$ can be estimated by the maximal derivation height with respect to $\mathcal{S}$. To extend Lemma 4.16 so that the derivational complexity function $\mathsf{dc}_{\mathcal{R}}$ can be measured via the function $\mathsf{dc}_{\mathcal{S}}$ we make use of the following lemma.

▶ **Lemma 4.17.** *For any ground term* t*, we have* $\mathsf{h}^{\mathsf{dp}(t)}(\mathsf{z}) \to_{\mathcal{S}}^{+} \mathsf{tr}(t)$.

Proof Sketch. The proof uses induction on $\mathsf{dp}(t)$. The main ingredients of the inductive argument are rules 11–12 of $\mathcal{S}$ and $\mathsf{choice}(t')$ essentially being an upper bound for all $\mathsf{norm}_i(t')$. ◀

We recall our main result, Theorem 3.5.

▶ **Theorem** (Main Theorem)**.** *Let* $\mathcal{R}$ *be a TRS whose termination is shown via Theorem 2.4 and let the reduction pair function* g *of* $\mathcal{R}$ *be multiple recursive. Then the derivational complexity function* $\mathsf{dc}_{\mathcal{R}}$ *with respect to* $\mathcal{R}$ *is bounded by a multiple recursive function. Furthermore this upper bound is tight.*

Proof. Let $\mathcal{S}$ be the simulating TRS for $\mathcal{R}$, as defined over the course of this section. Due to Lemma 4.12, $\mathsf{dc}_{\mathcal{S}}$ is multiply recursive. Let t be a term. Without loss of generality, we assume that t is ground. Due to Lemmas 4.16 and 4.17, we have the following inequalities:

$$\mathsf{dh}(t, \to_{\mathcal{R}}) \leqslant \mathsf{dh}(\mathsf{tr}(t), \to_{\mathcal{S}}) \leqslant \mathsf{dh}(\mathsf{h}^{\mathsf{dp}(t)}(\mathsf{z}), \to_{\mathcal{S}}) \ .$$

Note that $|\mathsf{h}^{\mathsf{dp}(t)}(\mathsf{z})| \leqslant |t|$. Hence for all $n \in \mathbb{N}$: $\mathsf{dc}_{\mathcal{R}}(n) \leqslant \mathsf{dc}_{\mathcal{S}}(n)$. Thus, $\mathsf{dc}_{\mathcal{R}}$ is multiply recursive. Tightness of the bound follows by Lemma 3.3: for any multiply recursive function f, there exists a k such that $\mathsf{dc}_{\mathcal{R}_3^k}$ dominates f, and $\mathsf{dc}_{\mathcal{R}_3^k}$ terminates by Theorem 2.4. Moreover, the proof tree induced by the termination proof shown in Example 3.2 admits a constant reduction pair function. ◀

5 Conclusion and Future Work

In this paper we established that the derivational complexity of any TRS whose termination can be shown within the DP framework is bounded by a multiple recursive function, whenever the set of processors used is suitably restricted.

As briefly mentioned in the introduction the *derivational complexity* is not the only measure of the complexity of a TRS suggested in the literature. In particular, alternative approaches have been suggested by Choppy et al. [3], Cichon and Lescanne [4], Hirokawa and the first author [11]. In [11] the *runtime complexity* with respect to a TRS is defined as a refinement of the derivational complexity, by restricting the set of admitted initial terms. This notion has first been suggested in [3], where it is augmented by an *average case* analysis. Finally [4] studies the complexity of the functions *computed* by a given TRS. This latter notion is extensively studied within *implicit computational complexity theory* (see [2] for an overview).

While we have presented our results for derivational complexity, it is easy to see that the same result holds for runtime complexity (as defined in [11]) and also for the complexity of the functions computed by the TRS (as suggested in [4]). For the former it suffices to observe that runtime complexity is a restriction of derivational complexity and that Example 3.2 provides a non-primitive recursive lower bound also in the context of runtime complexity. With respect to the second claim it suffices to observe that any function computed by a TRS that admits at most multiple recursive runtime complexity is computable on a Turing machine in multiple recursive time (compare also [1]). We assume that a similar result holds for the more involved notion proposed in [3]. However, this requires further work.

Thus our results indicate that the DP framework may induce multiple recursive complexity. This constitutes a first, but important, step towards the analysis of the complexity induced by the DP framework *in general*. Note that for all termination technique whose complexity has been analysed a multiple recursive upper bound exists. This leads us to the following conjecture.

▶ **Conjecture.** *Let $\mathcal{R}$ be a TRS whose termination can be proved with the DP framework using any DP processors, whose induced complexity does not exceed the class of multiple recursive functions. Then the derivational complexity with respect to $\mathcal{R}$ is multiple recursive.*

Should this conjecture be true, then for instance, none of the existing automated termination techniques would in theory be powerful enough to prove termination of Dershowitz's system `TRS/D33-33`, aka the Hydra battle rewrite system, whose complexity is not a provable recursive function of Peano Arithmetic (see [6, 14]). Hence far beyond multiple-recursion.

References

1 M. Avanzini and G. Moser. Closing the gap between runtime complexity and polytime computability. In *Proc. 21st RTA*, volume 6 of *LIPIcs*, pages 33–48, 2010.

2 P. Baillot, J.-Y. Marion, and S. R. D. Rocca. Guest editorial: Special issue on implicit computational complexity. *ACM Trans. Comput. Log.*, 10(4), 2009.

3 C. Choppy, S. Kaplan, and M. Soria. Complexity analysis of term-rewriting systems. *TCS*, 67(2–3):261–282, 1989.

4 E.-A. Cichon and P. Lescanne. Polynomial interpretations and the complexity of algorithms. In *Proc. 11th CADE*, volume 607 of *LNCS*, pages 139–147, 1992.

5 T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

6 N. Dershowitz and G. Moser. The Hydra battle revisited. In *Rewriting, Computation and Proof*, volume 4600 of *LNCS*, pages 1–27, 2007.

7 J. Endrullis, J. Waldmann, and H. Zantema. Matrix interpretations for proving termination of term rewriting. *JAR*, 40(3):195–220, 2008.

8 A. Geser. *Relative Termination*. PhD thesis, Universität Passau, 1990.

9 J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Mechanizing and improving dependency pairs. *JAR*, 37(3):155–203, 2006.

10 N. Hirokawa and A. Middeldorp. Tyrolean termination tool: Techniques and features. *IC*, 205:474–511, 2007.

11 N. Hirokawa and G. Moser. Automated complexity analysis based on the dependency pair method. In *Proc. 4th IJCAR*, volume 5195 of *LNAI*, pages 364–380, 2008.

12 D. Hofbauer. *Termination Proofs and Derivation Lengths in Term Rewriting Systems*. PhD thesis, Technische Universität Berlin, 1992.

13 D. Hofbauer and C. Lautemann. Termination proofs and the length of derivations. In *Proc. 3rd RTA*, volume 355 of *LNCS*, pages 167–177, 1989.

14 G. Moser. The Hydra Battle and Cichon's Principle. *AAECC*, 20(2):133–158, 2009.

15 G. Moser and A. Schnabl. The derivational complexity induced by the dependency pair method. In *Proc. 20th RTA*, volume 5595 of *LNCS*, pages 255–269, 2009.

16 G. Moser and A. Schnabl. The derivational complexity induced by the dependency pair method. *LMCS*, 2011. To appear. Available at `http://arxiv.org/abs/0904.0570`.

17 G. Moser and A. Schnabl. Termination proofs in the dependency pair framework may induce multiple recursive derivational complexity. *CoRR*, abs/1103.5082, 2011.

18 F. Neurauter, H. Zankl, and A. Middeldorp. Revisiting matrix interpretations for polynomial derivational complexity of term rewriting. In *Proc. 17th LPAR*, volume 6397 of *LNCS (ARCoSS)*, pages 550–564, 2010.

19 R. Péter. Zusammenhang der mehrfachen und transfiniten Rekursionen. *JSL*, 15(4):248–272, 1950.

20 R. Péter. *Recursive Functions*. Academic Press, 1967.

21 H. E. Rose. *Subrecursion - function and hierarchies*. Clarendon Press, 1984.

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

23 R. Thiemann. *The DP Framework for Proving Termination of Term Rewriting*. PhD thesis, University of Aachen, 2007.

24 J. Waldmann. Polynomially bounded matrix interpretations. In *Proc. 21st RTA*, volume 6 of *LIPIcs*, pages 357–372, 2010.

25 A. Weiermann. Termination proofs for term rewriting systems with lexicographic path orderings imply multiply recursive derivation lengths. *TCS*, 139(1,2):355–362, 1995.

26 H. Zankl and M. Korp. Modular complexity analysis via relative complexity. In *Proc. 21st RTA*, volume 6 of *LIPIcs*, pages 385–400, 2010.

Revisiting Matrix Interpretations for Proving Termination of Term Rewriting

Friedrich Neurauter* **and Aart Middeldorp**

Institute of Computer Science
University of Innsbruck, Austria
{friedrich.neurauter,aart.middeldorp}@uibk.ac.at

Abstract

Matrix interpretations are a powerful technique for proving termination of term rewrite systems, which is based on the well-known paradigm of interpreting terms into a domain equipped with a suitable well-founded order, such that every rewrite step causes a strict decrease. Traditionally, one uses vectors of non-negative numbers as domain, where two vectors are in the order relation if there is a strict decrease in the respective first components and a weak decrease in all other components. In this paper, we study various alternative well-founded orders on vectors of non-negative numbers based on vector norms and compare the resulting variants of matrix interpretations to each other and to the traditional approach. These comparisons are mainly theoretical in nature. We do, however, also identify one of these variants as a proper generalization of traditional matrix interpretations as a stand-alone termination method, which has the additional advantage that it gives rise to a more powerful implementation.

1998 ACM Subject Classification F.4.2 Grammars and Other Rewriting Systems, F.4.1 Mathematical Logic: Computational logic.

Keywords and phrases term rewriting, termination, matrix interpretations

Digital Object Identifier 10.4230/LIPIcs.RTA.2011.251

Category Regular Research Paper

1 Introduction

As far as research on termination of term rewrite systems is concerned, in recent years a lot of effort has been put on proving termination automatically. Indeed, many powerful techniques for establishing termination of term rewrite systems that have been developed in the course of time have been automated successfully, as is evident in the results of the (annual) international competition for termination tools.[1] In particular, the method of matrix interpretations greatly contributes to the success of these tools.

Matrix interpretations were originally introduced by Hofbauer and Waldmann in the context of string rewriting [8, 9], allowing them to solve hard termination problems like $\{\mathsf{aa} \to \mathsf{bc}, \mathsf{bb} \to \mathsf{ac}, \mathsf{cc} \to \mathsf{ab}\}$, Problem #104 in the RTA list of open problems.[2] One particular instance of the matrix method of [9] has been extended to term rewriting by Endrullis *et al.* in [4]. The method is based on the well-known paradigm of interpreting terms (strings) into a domain equipped with a suitable well-founded order, such that every

* Friedrich Neurauter is supported by a grant of the University of Innsbruck.
[1] http://termcomp.uibk.ac.at.
[2] http://rtaloop.mancoosi.univ-paris-diderot.fr.

22nd International Conference on Rewriting Techniques and Applications (RTA'11).
Editor: M. Schmidt-Schauß; pp. 251–266

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


rewrite step causes a strict decrease with respect to this order. In [4], the authors consider the set of vectors of natural numbers as domain and equip it with a well-founded order that is not total, such that two vectors are in the order relation if there is a strict decrease in the respective first components and a weak decrease in all other components. Function symbols are interpreted by suitable linear mappings represented by square matrices all of whose entries are natural numbers. In [1, 5, 15], the method of Endrullis *et al.* was lifted to the non-negative rational (real) numbers using the same technique that was already used to lift polynomial interpretations from the naturals to the rationals (reals) (cf. [7]). Recently, another generalization appeared in [3] that employs matrices of natural numbers (instead of vectors) as underlying domain and associates to each function symbol a linear matrix polynomial. In principle, this approach also allows for non-linear matrix polynomials.

In this paper, we re-examine the basics of the method of Endrullis *et al.*, especially focusing on the actual role of the well-founded order on vectors of natural numbers it is based on. Obviously, there are many other orders on vectors of natural numbers having the desired properties, so why the choice of this particular order? In [4], the justification is as follows (in addition to the very convincing fact that the resulting termination method is very powerful):

> *Of course other orders on vectors could have been chosen, too, but many of them are not suitable for our purpose. For instance, choosing a lexicographic order fails because then multiplication by a constant matrix is not monotone in general.*

But still the question remains whether there exist other orders inducing variants of matrix interpretations that are also useful for proving termination of term rewrite systems. To this end, we study various alternative well-founded orders on vectors of (natural) numbers based on vector norms. The underlying idea is that every rewrite step is supposed to decrease the "length" of the associated vectors. This leads directly to the notion of normed vector spaces, norms being a suitable measure of the length or magnitude of a vector. Basically, we consider two classes of orders, weakly decreasing orders, where two vectors are comparable only if there is a weak decrease in every single component, and orders without this property. The conclusion is that the latter kind of orders induces only weak forms of matrix interpretations that are no more powerful than linear polynomial interpretations. For weakly decreasing orders (like the order in [4]), however, the situation is different. That is to say that some of them do indeed induce matrix interpretations that are useful for proving termination. In particular, one of these variants subsumes traditional matrix interpretations and has the additional advantage that it gives rise to a more powerful implementation.

The remainder of this paper is organized as follows. In Section 2, we introduce some preliminary definitions and terminology concerning matrix interpretations. In Section 3, we present the orders on vectors of natural numbers considered in this paper. Sections 4 and 5 are dedicated to matrix interpretations over weakly decreasing orders and the comparison between them, while Section 6 features matrix interpretations over non-weakly decreasing orders. In Section 7 we present a generalization of traditional matrix interpretations, before concluding in Section 8.

2 Preliminaries

As usual, we denote by $\mathbb{N}$ and $\mathbb{Z}$ the sets of natural and integer numbers. Given $N \in \{\mathbb{N}, \mathbb{Z}\}$, $>_N$ denotes the standard order of the respective domain. The *cardinality* of a (finite) set S is denoted by $|S|$. For any ring R, we denote the ring of all n-dimensional square matrices over R by $R^{n\times n}$. A matrix is *non-negative* if all its entries are non-negative. Abusing notation, we

denote the set of all non-negative n-dimensional square matrices of $\mathbb{Z}^{n\times n}$ by $\mathbb{N}^{n\times n}$. As usual, we denote the *transpose* of a matrix (vector) M by M^T. For any vector $\vec{x} = (x_1, \dots, x_n)^T$, $(\vec{x})_i$ denotes its i-th component. Likewise, M_{ij} denotes the entry in the i-th row and j-th column of a matrix M, and M_{j-} (M_{-j}) refers to the j-th row (column). A *zero column* is a column, where all entries are zero. For $M \in R^{n\times n}$ and $\mathcal{I} \subseteq \{1, \dots, n\}$, $(M)_{\mathcal{I}}$ denotes the *submatrix* of M formed by the rows and columns whose indices are in the index set $\mathcal{I}$. Finally, a *permutation matrix* is a square matrix whose entries are all 0's and 1's, with exactly one 1 in each row and exactly one 1 in each column.

We assume familiarity with the basics of term rewriting [2, 14]. Let $\mathcal{V}$ denote a countably infinite set of variables and $\mathcal{F}$ a signature, that is, a set of function symbols equipped with fixed arities. The set of *terms* over $\mathcal{F}$ and $\mathcal{V}$ is denoted by $\mathcal{T}(\mathcal{F}, \mathcal{V})$. By $\mathcal{V}\mathrm{ar}(t)$ we denote the set of variables occurring in a term t, and $|t|_x$ denotes the number of occurrences of the variable x. A *rewrite rule* is a pair of terms (ℓ, r), conveniently written as $\ell \to r$, such that ℓ is not a variable and all variables in r are contained in ℓ. A *term rewrite system* $\mathcal{R}$ (TRS for short) over $\mathcal{T}(\mathcal{F}, \mathcal{V})$ is a set of rewrite rules. The rewrite relation induced by $\to$ is denoted by $\to_{\mathcal{R}}$. As usual, $\to^*_{\mathcal{R}}$ denotes the reflexive transitive closure of $\to_{\mathcal{R}}$ and $\to^n_{\mathcal{R}}$ its n-th iterate. For notational convenience, we sometimes drop the subscript $\mathcal{R}$ if it is clear from the context.

An important concept for establishing termination of TRSs is the notion of well-founded monotone algebras. An *$\mathcal{F}$-algebra* $\mathcal{A}$ consists of a non-empty carrier A and interpretation functions $f_{\mathcal{A}} \colon A^n \to A$ for every n-ary function symbol $f \in \mathcal{F}$. By $[\alpha]_{\mathcal{A}}(\cdot) \colon \mathcal{T}(\mathcal{F}, \mathcal{V}) \to A$ we denote the usual evaluation function of $\mathcal{A}$ with respect to a variable assignment $\alpha \colon \mathcal{V} \to A$. An interpretation $f_{\mathcal{A}} \colon A^n \to A$ is monotone with respect to a binary relation $>_A$ on A if $f_A(a_1, \dots, a_i, \dots, a_n) >_A f_A(a_1, \dots, b, \dots, a_n)$ for all $a_1, \dots, a_n, b \in A$ and $i \in \{1, \dots, n\}$ with $a_i >_A b$. A *well-founded monotone $\mathcal{F}$-algebra* is a pair $(\mathcal{A}, >_A)$, where $\mathcal{A}$ is an $\mathcal{F}$-algebra and $>_A$ is a well-founded order on A, such that every $f_{\mathcal{A}}$ is monotone with respect to $>_A$. It is well-known that a TRS $\mathcal{R}$ is terminating if and only if it is *compatible* with a well-founded monotone algebra $(\mathcal{A}, >_A)$, where compatibility means that for every rewrite rule $\ell \to r \in \mathcal{R}$, $[\alpha]_{\mathcal{A}}(\ell) >_A [\alpha]_{\mathcal{A}}(r)$ for all variable assignments $\alpha \colon \mathcal{V} \to A$.

3 Well-founded Orders on Vectors of Natural Numbers

In this section we introduce several well-founded orders on vectors of natural numbers serving as foundation for alternative kinds of matrix interpretations. We consider two classes of orders on $\mathbb{N}^n$, $n \geqslant 1$, weakly decreasing orders and non-weakly decreasing ones.

3.1 Weakly Decreasing Orders

We call a (partial) order $>$ on $\mathbb{N}^n$ *weakly decreasing* if $(x_1, \dots, x_n)^T > (y_1, \dots, y_n)^T$ implies $x_i \geqslant_{\mathbb{N}} y_i$ for all $i \in \{1, \dots, n\}$. The component-wise (partial) order on $\mathbb{N}^n$ induced by $\geqslant_{\mathbb{N}}$ is denoted by $\geqslant^w$.

▶ **Definition 3.1.** Let $\mathcal{I} \subseteq \{1, \dots, n\}$ be a non-empty index set, and let $\vec{x} = (x_1, \dots, x_n)^T$ and $\vec{y} = (y_1, \dots, y_n)^T$ be vectors in $\mathbb{N}^n$. We define relations $>^w_{\mathcal{I}}$, $>^w_{\Sigma}$, $>^w_{\ell}$ and $>^w_m$ on $\mathbb{N}^n$ as follows:

- Weak decrease + strict decrease in some component(s):

$$\vec{x} >^w_{\mathcal{I}} \vec{y} : \iff \vec{x} \geqslant^w \vec{y} \wedge \exists\, j \in \mathcal{I} \colon x_j >_{\mathbb{N}} y_j$$

- Weak decrease + strict decrease in sum of components:

$$\vec{x} >^w_\Sigma \vec{y} :\iff \vec{x} \geqslant^w \vec{y} \wedge \sum_{i=1}^{n} x_i >_\mathbb{N} \sum_{i=1}^{n} y_i$$

- Weak decrease + strict decrease in Euclidean length:

$$\vec{x} >^w_\ell \vec{y} :\iff \vec{x} \geqslant^w \vec{y} \wedge \sum_{i=1}^{n} x_i^2 >_\mathbb{N} \sum_{i=1}^{n} y_i^2$$

- Weak decrease + strict decrease in maximum component:

$$\vec{x} >^w_m \vec{y} :\iff \vec{x} \geqslant^w \vec{y} \wedge \max_i x_i >_\mathbb{N} \max_i y_i$$

It is routine to verify that all these relations are in fact well-founded orders on vectors of natural numbers.

▶ **Lemma 3.2.** *The relations $>^w_\mathcal{I}$, $>^w_\Sigma$, $>^w_\ell$ and $>^w_m$ are well-founded orders on $\mathbb{N}^n$.* ◀

The relations listed above are not the only well-founded orders on $\mathbb{N}^n$. Numerous variations exist. Some of these (like parameterizing $>^w_\Sigma$ or $>^w_\ell$ by an index set $\mathcal{I}$) are implicitly covered because of the lemma below, while others (like demanding a strict decrease in all components specified by an index set $\mathcal{I}$) proved to be impractical.

Intuitively, the order $>^w_\mathcal{I}$ is a generalization of $>^w_1$, the order used in [4], where the strict decrease is not necessarily fixed to one specific component; in particular, $>^w_1 = >^w_\mathcal{I}$ for $\mathcal{I} = \{1\}$. Moreover, its extension to matrices yields the main order considered in [3]. As to the remaining three orders, two vectors being in relation means that there is a strict decrease in the lengths of the vectors with respect to the Manhattan, Euclidean or maximum norm, respectively [10]. The relationship between these orders is described in the following lemma.

▶ **Lemma 3.3.** *Let $\mathcal{I}$, $\mathcal{J}$ and $\mathcal{K}$ be non-empty index sets, such that $\mathcal{I} = \{1, \ldots, n\}$ and $\mathcal{J} \subseteq \mathcal{K} \subseteq \mathcal{I}$. Then the following statements hold:*

1. *$>^w_\mathcal{J} \subseteq >^w_\mathcal{K}$ and $>^w_1 = >^w_{\{1\}}$,*
2. *$>^w_1$ and $>^w_m$ are incomparable for $n \geqslant 2$, identical otherwise,*
3. *$>^w_m \subset >^w_\mathcal{I} = >^w_\Sigma = >^w_\ell$ for $n \geqslant 2$, all identical otherwise, and*
4. *$>^w_\mathcal{I}$ is the strict part of $\geqslant^w$.*

The last item gives rise to the following corollary stressing an important aspect of some of the orders considered above.

▶ **Corollary 3.4.** *For $\mathcal{I} = \{1, \ldots, n\}$, $>^w_\mathcal{I}$ is the most general of the weakly decreasing proper orders on $\mathbb{N}^n$ (in the sense that it subsumes any other such order).* ◀

3.2 Non-weakly Decreasing Orders

Taking a closer look at Definition 3.1, one observes that weak decreasingness is not the essential property for obtaining well-founded orders on vectors of natural numbers, which is all we need to build matrix interpretations upon. That is to say that the last three orders remain well-founded orders on $\mathbb{N}^n$ even after dropping this property. We denote the corresponding orders by $>_\Sigma$, $>_\ell$ and $>_m$, respectively. Concerning $>^w_\mathcal{I}$, one must be careful when dropping weak decreasingness because the resulting relation $>_\mathcal{I}$ is an order only if the index set $\mathcal{I}$ is a singleton set, in which case $>_\mathcal{I}$ is also well-founded. In the remainder of this paper this is implicitly assumed whenever we refer to $>_\mathcal{I}$. Finally, we note that all four orders coincide in the one-dimensional case ($n = 1$), all being equal to $>_\mathbb{N}$. For $n \geqslant 2$, however, all these orders are pairwise incomparable (for all singleton sets $\mathcal{I}$).

4 Matrix Interpretations and Weakly Decreasing Orders

In this section we take the orders introduced in Definition 3.1 and build matrix interpretations on top of them. According to Lemma 3.3 (item 3), we only have to consider the family of orders $(>^w_{\mathcal{I}})_{\mathcal{I}}$ parametrized by some non-empty index set $\mathcal{I} \subseteq \{1,\dots,n\}$ and $>^w_m$, the order induced by the maximum norm. We shall see, however, that the latter kind of matrix interpretation is subsumed by an instance of the former.

Before we can go about formally defining matrix interpretations over $>^w_{\mathcal{I}}$ ($>^w_m$), we have to have an understanding of when a linear function is monotone with respect to the orders $\geqslant^w$ and $>^w_{\mathcal{I}}$ ($>^w_m$). We consider linear functions of the form $f(\vec{x}_1,\dots,\vec{x}_k) = \sum_{i=1}^k F_i\vec{x}_i + \vec{f}$, where $\vec{f} \in \mathbb{N}^n$ and $F_i \in \mathbb{N}^{n\times n}$ for all $i \in \{1,\dots,k\}$. Obviously, all such functions are monotone with respect to $\geqslant^w$. Concerning monotonicity with respect to $>^w_{\mathcal{I}}$, we give necessary and sufficient conditions in the lemma below. A similar lemma, showing sufficiency of the conditions, appeared in [3].

▶ **Lemma 4.1.** *Let $\mathcal{I} \subseteq \{1,\dots,n\}$ be a non-empty index set. The function $f(\vec{x}_1,\dots,\vec{x}_k) = \sum_{i=1}^k F_i\vec{x}_i + \vec{f}$ is monotone with respect to $>^w_{\mathcal{I}}$ if and only if for each $(F_i)_{\mathcal{I}}$, $i = 1,\dots,k$, all column sums are at least one.*

Proof. Let $\vec{x}_1,\dots,\vec{x}_k$ and $\vec{y}$ be arbitrary vectors in $\mathbb{N}^n$, such that $\vec{x}_i >^w_{\mathcal{I}} \vec{y}$ for some argument position $i \in \{1,\dots,k\}$. Then there exist a vector $\vec{d} \in \mathbb{N}^n$ and an index $j \in \mathcal{I}$, such that $\vec{x}_i = \vec{y}+\vec{d}$ and $d_j >_{\mathbb{N}} 0$. Now $f(\dots,\vec{x}_i,\dots) >^w_{\mathcal{I}} f(\dots,\vec{y},\dots)$ holds if and only if $F_i\,\vec{x}_i >^w_{\mathcal{I}} F_i\,\vec{y}$, which is equivalent to $F_i\,\vec{d} >^w_{\mathcal{I}} 0$. If all column sums of $(F_i)_{\mathcal{I}}$ are at least one, then we have $(F_i)_{-j} >^w_{\mathcal{I}} 0$, which yields $F_i\,\vec{d} >^w_{\mathcal{I}} 0$ because of $F_i\,\vec{d} \geqslant^w (F_i)_{-j} \cdot d_j \geqslant^w (F_i)_{-j}$.

Conversely, if $(F_i)_{\mathcal{I}}$ has a zero column, then let $j' \in \mathcal{I}$ denote the index of the column of F_i it originates from, and let $\vec{x}_i$ be zero everywhere except for its j'-th component, which we set to one. Then $\vec{x}_i >^w_{\mathcal{I}} 0$ but $f(\dots,0,\vec{x}_i,0,\dots) = (F_i)_{-j'} + \vec{f} \not>^w_{\mathcal{I}} \vec{f} = f(0,\dots,0)$. ◀

We are now ready to formally define matrix interpretations over instances of $>^w_{\mathcal{I}}$ (cf. also the E-compatible matrix interpretations in [3]).

▶ **Definition 4.2.** Let $\mathcal{F}$ denote a signature and $\mathcal{I} \subseteq \{1,\dots,n\}$ a non-empty index set. An n-dimensional *matrix interpretation $\mathcal{M}_{>^w_{\mathcal{I}}}$ over $>^w_{\mathcal{I}}$* is an $\mathcal{F}$-algebra with carrier $\mathbb{N}^n$ together with the well-founded order $>^w_{\mathcal{I}}$, where each k-ary function symbol $f \in \mathcal{F}$ is interpreted by a function $f_{\mathcal{M}}\colon (\mathbb{N}^n)^k \to \mathbb{N}^n, (\vec{x}_1,\dots,\vec{x}_k) \mapsto \sum_{i=1}^k F_i\vec{x}_i + \vec{f}$ with $\vec{f} \in \mathbb{N}^n$ and $F_i \in \mathbb{N}^{n\times n}$ for all $i \in \{1,\dots,k\}$, such that $f_{\mathcal{M}}$ is monotone with respect to $>^w_{\mathcal{I}}$.

Clearly, such matrix interpretations are well-founded monotone algebras. Moreover, the notion of matrix interpretations of Endrullis *et al.* [4] is included in Definition 4.2 by choosing the special index set $\mathcal{I} = \{1\}$. In order to use a matrix interpretation $\mathcal{M}$ over $>^w_{\mathcal{I}}$ to establish termination of a TRS, one should be able to check whether $[\alpha]_{\mathcal{M}}(\ell) >^w_{\mathcal{I}} [\alpha]_{\mathcal{M}}(r)$ for all $\alpha\colon \mathcal{V} \to \mathbb{N}^n$ and rules $\ell \to r$. The following well-known lemma is helpful for this purpose.

▶ **Lemma 4.3.** *Let $\mathcal{M}$ be an $\mathcal{F}$-algebra with carrier $\mathbb{N}^n$ as in Definition 4.2 and t a term with $\mathcal{V}\mathrm{ar}(t) = \{x_1,\dots,x_m\}$. Then there exist matrices $T_1,\dots,T_m \in \mathbb{N}^{n\times n}$ and a vector $\vec{t} \in \mathbb{N}^n$, such that for any assignment $\alpha\colon \mathcal{V} \to \mathbb{N}^n$, $[\alpha]_{\mathcal{M}}(t) = T_1\alpha(x_1) + \cdots + T_m\alpha(x_m) + \vec{t}$.* ◀

Therefore, the compatibility checks $[\alpha]_{\mathcal{M}}(\ell) >^w_{\mathcal{I}} [\alpha]_{\mathcal{M}}(r)$ and $[\alpha]_{\mathcal{M}}(\ell) \geqslant^w [\alpha]_{\mathcal{M}}(r)$ boil down to the comparison of such linear functions, which is decidable according to the next lemma. Here, $\geqslant$ denotes the component-wise (partial) order on $\mathbb{N}^{n\times n}$ induced by $\geqslant_{\mathbb{N}}$.

▶ **Lemma 4.4.** *Let $L_1, \dots, L_m$, $R_1, \dots, R_m$ and $\vec{l}$, $\vec{r}$ correspond to a rewrite rule $\ell \to r$ as in Lemma 4.3. Then, for $\triangleright \in \{>^w_{\mathcal{I}}, \geqslant^w\}$, $[\alpha]_{\mathcal{M}}(\ell) \triangleright [\alpha]_{\mathcal{M}}(r)$ for all variable assignments $\alpha \colon \mathcal{V} \to \mathbb{N}^n$ if and only if $\vec{l} \triangleright \vec{r}$ and $L_i \geqslant R_i$ for $i = 1, \dots, m$.* ◀

We close this section with the treatment of matrix interpretations over $>^w_m$. In particular, we show that they are subsumed by the instance of matrix interpretations over $>^w_{\mathcal{I}}$ one obtains by choosing $\mathcal{I} = \{1, \dots, n\}$, which is assumed to be the case in the rest of this section. According to Lemma 3.3, we have $>^w_m \subseteq >^w_{\mathcal{I}}$ for all dimensions $n \geqslant 1$. However, this does not directly imply that the same inclusion also holds for matrix interpretations based on these two orders because of the monotonicity requirement that all interpretation functions have to satisfy. If the monotonicity conditions with respect to $>^w_{\mathcal{I}}$ are more strict than the ones for $>^w_m$, then the set of potential interpretation functions is smaller, and it is therefore very well conceivable that the inclusion on the base orders does not propagate to the notions of matrix interpretations built on top of them. However, this is not the case for the two orders considered here.

▶ **Lemma 4.5.** *Let $f(\vec{x}_1, \dots, \vec{x}_k) = \sum_{i=1}^k F_i \vec{x}_i + \vec{f}$, where $\vec{f} \in \mathbb{N}^n$ and $F_1, \dots, F_k \in \mathbb{N}^{n \times n}$. Then monotonicity of f with respect to $>^w_m$ implies monotonicity with respect to $>^w_{\mathcal{I}}$.*

Proof. This can be shown using contraposition. Assume that f is not monotone with respect to $>^w_{\mathcal{I}}$. According to Lemma 4.1 this means that (at least) one of its matrices has a zero column. Without loss of generality, let the j-th column of some F_i, $i \in \{1, \dots, k\}$, be a zero column and let $\vec{x}_i$ be zero everywhere except for its j-th component. Then $\vec{x}_i >^w_m 0$ but $f(\dots, 0, \vec{x}_i, 0, \dots) = \vec{f} \not>^w_m \vec{f} = f(0, \dots, 0)$, i.e., f is not monotone with respect to $>^w_m$. ◀

Hence, if $\mathcal{M}_{>^w_m}$ is a matrix interpretation over $>^w_m$, consisting of a set of interpretation functions that are monotone with respect to $>^w_m$, then the same functions together with $>^w_{\mathcal{I}}$ constitute a matrix interpretation $\mathcal{M}_{>^w_{\mathcal{I}}}$ over $>^w_{\mathcal{I}}$ that is able to orient all rules orientable by $\mathcal{M}_{>^w_m}$ because of the inclusion $>^w_m \subseteq >^w_{\mathcal{I}}$. In other words, $\mathcal{M}_{>^w_{\mathcal{I}}}$ subsumes $\mathcal{M}_{>^w_m}$.

5 Comparing Matrix Interpretations over Weakly Decreasing Orders

After the discussion in the previous section the family of orders $(>^w_{\mathcal{I}})_{\mathcal{I}}$ parametrized by some non-empty index set $\mathcal{I} \subseteq \{1, \dots, n\}$ remains as a potentially interesting foundation for matrix interpretations. It includes the traditional order $>^w_1$ as well as $>^w_\Sigma = >^w_\ell$, the most general of the weakly decreasing orders on $\mathbb{N}^n$ (cf. Lemma 3.3 and Corollary 3.4). Now the purpose of this chapter is to compare the resulting variants of matrix interpretations to each other and thus also to the traditional approach.

First, we remark that we do not have to consider all possible index sets since matrix interpretations are invariant under permutations. For example, matrix interpretations over $>^w_{\{1\}}$ are equivalent to matrix interpretations over $>^w_{\{j\}}$, $j \in \{2, \dots, n\}$, with respect to termination proving power. The relevant property is that there is a strict decrease in a single fixed vector component, it is not important which component. All that matters is the cardinality of the index set $\mathcal{I}$. Hence, for n-dimensional matrix interpretations, we are left with n different index sets, and, without loss of generality, we can restrict to the sets $\mathcal{I}_d = \{1, \dots, d\}$ for $d = 1, 2, \dots, n$. By definition, the following inclusions hold: $>^w_{\mathcal{I}_1} \subset >^w_{\mathcal{I}_2} \subset \cdots \subset >^w_{\mathcal{I}_n}$. However, as explained at the end of the previous section, from this we cannot immediately conclude that the same inclusions also hold for matrix interpretations based on these orders because for $\mathcal{I} \subset \mathcal{J}$, monotonicity of a function with respect to $>^w_{\mathcal{I}}$ does not imply monotonicity with respect to $>^w_{\mathcal{J}}$ according to Lemma 4.1. In fact, the situation turns out to be a bit more intricate. To begin with, let us consider the following example.

▶ **Example 5.1.** Consider the TRS $\mathcal{R}_1 = \{\, \mathsf{f}(\mathsf{a}) \to \mathsf{f}(\mathsf{g}(\mathsf{a})), \mathsf{g}(\mathsf{b}) \to \mathsf{g}(\mathsf{f}(\mathsf{b})) \,\}$. Termination of this system can be shown with the following 2-dimensional matrix interpretation over $>^w_{\{1,2\}}$:

$$\mathsf{f}_{\mathcal{M}}(\vec{x}) = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \vec{x} \qquad \mathsf{g}_{\mathcal{M}}(\vec{x}) = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \vec{x} \qquad \mathsf{a}_{\mathcal{M}} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \qquad \mathsf{b}_{\mathcal{M}} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

However, one can show that there is no compatible 2-dimensional matrix interpretation over $>^w_{\{1\}}$. In the same vein, one can establish termination of the TRS $\mathcal{R}_2$

$$\begin{array}{lll} \mathsf{f}(\mathsf{g}(x)) \to \mathsf{f}(\mathsf{a}(\mathsf{g}(\mathsf{g}(\mathsf{f}(x))), \mathsf{g}(\mathsf{g}(\mathsf{f}(x))))) & \mathsf{a}(x,x) \to \mathsf{h}(x) & \mathsf{f}(x) \to x \\ \mathsf{h}(\mathsf{h}(x)) \to \mathsf{c}(\mathsf{h}(x)) & \mathsf{c}(x) \to x & \mathsf{g}(x) \to x \end{array}$$

via the following 2-dimensional matrix interpretation over $>^w_{\{1\}}$

$$\mathsf{a}_{\mathcal{M}}(\vec{x},\vec{y}) = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \vec{x} + \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \vec{y} + \begin{pmatrix} 3 \\ 0 \end{pmatrix} \qquad \mathsf{c}_{\mathcal{M}}(\vec{x}) = \begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix} \vec{x} + \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

$$\mathsf{f}_{\mathcal{M}}(\vec{x}) = \begin{pmatrix} 1 & 3 \\ 0 & 1 \end{pmatrix} \vec{x} + \begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad \mathsf{g}_{\mathcal{M}}(\vec{x}) = \begin{pmatrix} 1 & 0 \\ 1 & 2 \end{pmatrix} \vec{x} + \begin{pmatrix} 1 \\ 3 \end{pmatrix} \quad \mathsf{h}_{\mathcal{M}}(\vec{x}) = \begin{pmatrix} 2 & 0 \\ 0 & 0 \end{pmatrix} \vec{x} + \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

and show that there is no compatible 2-dimensional matrix interpretation over $>^w_{\{1,2\}}$.

The bottom line of this example is that if we fix the dimension, then matrix interpretations over $>^w_{\{1\}}$ are incomparable to matrix interpretations over $>^w_{\{1,2\}}$. (We are not aware of a general construction that works for any dimension.) However, without this restriction the situation is altogether different. That is to say that for dimension 3, for example, there is a compatible matrix interpretation over $>^w_{\{1\}}$ for the TRS $\mathcal{R}_1$. Likewise, there is a compatible 3-dimensional matrix interpretation over $>^w_{\{1,2\}}$ for the TRS $\mathcal{R}_2$. Indeed, that is no coincidence as will be shown in the remainder of this section. In particular, we shall see that in some sense the various instances of matrix interpretations over $>^w_{\mathcal{I}}$ are all equivalent with respect to termination proving power, no matter what the index set $\mathcal{I}$ looks like. To get to the bottom of this phenomenon, we need a couple of transformations on matrix interpretations.

As to the first transformation, let $P \in \mathbb{N}^{n\times n}$ be a nonsingular matrix and $\mathcal{M}$ some matrix interpretation consisting of a collection of interpretation functions $\{f_{\mathcal{M}}\}_{f\in\mathcal{F}}$, such that each k-ary function symbol f in the signature is interpreted by a function $f_{\mathcal{M}}(\vec{x}_1,\dots,\vec{x}_k) = \sum_{i=1}^k F_i\vec{x}_i + \vec{f}$, where $\vec{f} \in \mathbb{N}^n$ and $F_i \in \mathbb{N}^{n\times n}$ for all $i \in \{1,\dots,k\}$. Then we associate with $\mathcal{M}$ a matrix interpretation $\Phi_P(\mathcal{M})$, where each k-ary function symbol f is interpreted by a function $f_{\Phi_P(\mathcal{M})}(\vec{x}_1,\dots,\vec{x}_k) = \sum_{i=1}^k PF_iP^{-1}\vec{x}_i + P\vec{f}$.

▶ Remark. Note that in general PF_iP^{-1} is not a non-negative matrix, even if P and F_i are non-negative. As we need this property in our context, we must be careful when applying this transformation, unless P happens to be a (generalized) permutation matrix. In the remainder of this paper, non-negativity of PF_iP^{-1} is assumed or explicitly stated.

According to Lemma 4.3, the interpretation of a term with respect to $\mathcal{M}$ and a variable assignment α can be written as $[\alpha]_{\mathcal{M}}(t) = T_1\alpha(x_1) + \cdots + T_m\alpha(x_m) + \vec{t}$. By construction of $\Phi_P(\mathcal{M})$, we obtain the following lemma.

▶ **Lemma 5.2.** *Let $T_1, \dots, T_m$ and $\vec{t}$ correspond to a term t as described in Lemma 4.3. Then $[\alpha]_{\Phi_P(\mathcal{M})}(t) = PT_1P^{-1}\alpha(x_1) + \cdots + PT_mP^{-1}\alpha(x_m) + P\vec{t}$ for any assignment α.* ◀

▶ **Corollary 5.3.** *For every ground term t, $[\alpha]_{\Phi_P(\mathcal{M})}(t) = P \cdot [\alpha]_{\mathcal{M}}(t)$.* ◀

Our next transformation associates with an n-dimensional matrix interpretation $\mathcal{M}$ (as above) an $(n+1)$-dimensional matrix interpretation $\Psi(\mathcal{M})$, where each k-ary function symbol f is interpreted by a function $f_{\Psi(\mathcal{M})}(\vec{x}_1,\dots,\vec{x}_k) = \sum_{i=1}^{k} F_i'\vec{x}_i + \vec{f}'$, such that for all $i \in \{1,\dots,k\}$,

$$\vec{f}' = \begin{pmatrix} 0 \\ \vec{f} \end{pmatrix} \text{ and } F_i' = \begin{pmatrix} f_i & 0 \\ 0 & F_i \end{pmatrix} \text{ for some } f_i \in \mathbb{N} \setminus \{0\}.$$

Moreover, we associate with $\mathcal{M}$ (resp. $\Psi(\mathcal{M})$) a linear polynomial interpretation $\mathcal{P}(\mathcal{M})$, where each k-ary function symbol f is interpreted by a linear polynomial $f_{\mathcal{P}(\mathcal{M})}(x_1,\dots,x_k) = \sum_{i=1}^{k} f_i x_i$ (with the f_i's of $\Psi(\mathcal{M})$).

▶ **Lemma 5.4.** *Let t be an arbitrary term. Then for all variable assignments $\alpha\colon \mathcal{V} \to \mathbb{N}^n$ and $\beta\colon \mathcal{V} \to \mathbb{N}$, the following statement holds:*

$$[\gamma]_{\Psi(\mathcal{M})}(t) = \begin{pmatrix} [\beta]_{\mathcal{P}(\mathcal{M})}(t) \\ [\alpha]_{\mathcal{M}}(t) \end{pmatrix} \text{ for the variable assignment } \gamma\colon \mathcal{V} \to \mathbb{N}^{n+1},\ x \mapsto \begin{pmatrix} \beta(x) \\ \alpha(x) \end{pmatrix}. \quad \blacktriangleleft$$

▶ **Corollary 5.5.** *For every ground term t, $[\gamma]_{\Psi(\mathcal{M})}(t) = \begin{pmatrix} 0 \\ [\alpha]_{\mathcal{M}}(t) \end{pmatrix}$.* ◀

Again, by Lemma 4.3, $[\alpha]_{\mathcal{M}}(t)$ can be written as $[\alpha]_{\mathcal{M}}(t) = T_1\alpha(x_1) + \cdots + T_m\alpha(x_m) + \vec{t}$. Likewise, the interpretation of t with respect to $\mathcal{P}(\mathcal{M})$ and some variable assignment β can be written as $[\beta]_{\mathcal{P}(\mathcal{M})}(t) = t_1\beta(x_1) + \cdots + t_m\beta(x_m)$, where $t_1,\dots,t_m \in \mathbb{N}$. Plugging these expressions into Lemma 5.4, we obtain the following lemma.

▶ **Lemma 5.6.** *Let $T_1,\dots,T_m$, $t_1,\dots,t_m$ and $\vec{t}$ correspond to a term t as described above. Then, in the situation of Lemma 5.4, the following statement holds:*

$$[\gamma]_{\Psi(\mathcal{M})}(t) = \sum_{i=1}^{m} \begin{pmatrix} t_i & 0 \\ 0 & T_i \end{pmatrix} \gamma(x_i) + \begin{pmatrix} 0 \\ \vec{t} \end{pmatrix} \quad \blacktriangleleft$$

Moreover, if all the f_i's introduced by $\Psi(\mathcal{M})$ are one, then each t_i in $[\beta]_{\mathcal{P}(\mathcal{M})}(t)$ corresponds to the number of occurrences of the associated variable x_i.

▶ **Lemma 5.7.** *Let t be an arbitrary term with $\mathcal{V}\mathrm{ar}(t) = \{x_1,\dots,x_m\}$, and let all interpretation functions in $\mathcal{P}(\mathcal{M})$ have the shape $f_{\mathcal{P}(\mathcal{M})}(x_1,\dots,x_k) = \sum_{i=1}^{k} x_i$ (for each k-ary function symbol f). Then for any variable assignment β, $[\beta]_{\mathcal{P}(\mathcal{M})}(t) = \sum_{i=1}^{m} |t|_{x_i}\beta(x_i)$.* ◀

We are now ready to present the main results of this section comparing matrix interpretations over various instances of $>_{\mathcal{I}}^{w}$ with respect to proving (direct) termination of TRSs. In what follows, for a given TRS $\mathcal{R}$, $\mathcal{R} \subseteq \succcurlyeq_{\mathcal{P}(\mathcal{M})}$ abbreviates $[\beta]_{\mathcal{P}(\mathcal{M})}(\ell) \geqslant_{\mathbb{N}} [\beta]_{\mathcal{P}(\mathcal{M})}(r)$ for all variable assignments $\beta\colon \mathcal{V} \to \mathbb{N}$ and all rewrite rules $\ell \to r \in \mathcal{R}$.

▶ **Lemma 5.8.** *Let $\mathcal{M}$ be an n-dimensional matrix interpretation over $>_{\mathcal{I}}^{w}$, $\mathcal{I} \subseteq \{1,\dots,n\}$, and let $\mathcal{R}$ be a TRS satisfying $\mathcal{R} \subseteq \succcurlyeq_{\mathcal{P}(\mathcal{M})}$. Then compatibility of $\mathcal{R}$ with $\mathcal{M}$ implies compatibility with an $(n+1)$-dimensional matrix interpretation over $>_{\mathcal{J}}^{w}$, where $|\mathcal{J}| = |\mathcal{I}| + 1$, $\mathcal{J} \subseteq \{1,\dots,n+1\}$.*

Proof. Assuming that $\mathcal{M}$ is compatible with $\mathcal{R}$, we show that $\Psi(\mathcal{M})$ is compatible as well. To this end, we let $\mathcal{J} = \{1\} \cup \{\, x+1 \mid x \in \mathcal{I} \,\}$ and reason as follows. By assumption, all interpretation functions of $\mathcal{M}$ are monotone with respect to $>_{\mathcal{I}}^{w}$, that is, for each matrix $M \in \mathcal{M}$, all column sums of $(M)_{\mathcal{I}}$ are at least one according to Lemma 4.1. By construction

of $\Psi(\mathcal{M})$, this implies that for each matrix $M' \in \Psi(\mathcal{M})$, all column sums of $(M')_{\mathcal{J}}$ are also at least one. Hence, all interpretation functions of $\Psi(\mathcal{M})$ are monotone with respect to $>^w_{\mathcal{J}}$. As to compatibility of $\Psi(\mathcal{M})$ with $\mathcal{R}$, for any rewrite rule $\ell \to r$, $[\gamma]_{\Psi(\mathcal{M})}(\ell) >^w_{\mathcal{J}} [\gamma]_{\Psi(\mathcal{M})}(r)$ holds for all variable assignments γ if and only if

$$\begin{pmatrix} [\beta]_{\mathcal{P}(\mathcal{M})}(\ell) \\ [\alpha]_{\mathcal{M}}(\ell) \end{pmatrix} >^w_{\mathcal{J}} \begin{pmatrix} [\beta]_{\mathcal{P}(\mathcal{M})}(r) \\ [\alpha]_{\mathcal{M}}(r) \end{pmatrix} \text{ for all variable assignments } \alpha \text{ and } \beta \text{ (cf. Lemma 5.4).}$$

By definition of $>^w_{\mathcal{J}}$, it remains to show that there is a weak decrease in every single component and a strict decrease in some component with index $j \in \mathcal{J}$. By compatibility of $\mathcal{M}$ with $\mathcal{R}$, we have $[\alpha]_{\mathcal{M}}(\ell) >^w_{\mathcal{I}} [\alpha]_{\mathcal{M}}(r)$ for all assignments α, which immediately establishes the latter requirement and, together with the assumption $\mathcal{R} \subseteq \succcurlyeq_{\mathcal{P}(\mathcal{M})}$, also the former. ◀

With the help of Lemma 5.7 one can replace the semantic condition $\mathcal{R} \subseteq \succcurlyeq_{\mathcal{P}(\mathcal{M})}$ by a (more familiar) syntactic condition.

▶ **Corollary 5.9.** *Let $\mathcal{R}$ be a non-duplicating TRS. Then compatibility of $\mathcal{R}$ with an n-dimensional matrix interpretation over $>^w_{\mathcal{I}}$, $\mathcal{I} \subseteq \{1, \dots, n\}$, implies compatibility with an $(n+1)$-dimensional matrix interpretation over $>^w_{\mathcal{J}}$, where $|\mathcal{J}| = |\mathcal{I}| + 1$, $\mathcal{J} \subseteq \{1, \dots, n+1\}$.*

Proof. Setting all the f_i's introduced by $\Psi(\mathcal{M})$ to one, the condition $\mathcal{R} \subseteq \succcurlyeq_{\mathcal{P}(\mathcal{M})}$ becomes equivalent to $\mathcal{R}$ being non-duplicating according to Lemma 5.7. ◀

▶ **Example 5.10.** Consider the TRS $\mathcal{R}_3 = \{\, \mathsf{f}(x) \to \mathsf{g}(\mathsf{h}(x,x)), \mathsf{g}(\mathsf{a}) \to \mathsf{f}(\mathsf{a}) \,\}$. Termination can be shown with the following 2-dimensional matrix interpretation over $>^w_{\{1\}}$:

$$\mathsf{f}_{\mathcal{M}}(\vec{x}) = \begin{pmatrix} 3 & 0 \\ 2 & 0 \end{pmatrix} \vec{x} + \begin{pmatrix} 2 \\ 0 \end{pmatrix} \qquad \mathsf{g}_{\mathcal{M}}(\vec{x}) = \begin{pmatrix} 1 & 1 \\ 0 & 0 \end{pmatrix} \vec{x} + \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

$$\mathsf{h}_{\mathcal{M}}(\vec{x}, \vec{y}) = \begin{pmatrix} 2 & 0 \\ 0 & 0 \end{pmatrix} \vec{x} + \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \vec{y} \qquad \mathsf{a}_{\mathcal{M}} = \begin{pmatrix} 0 \\ 3 \end{pmatrix}$$

Moreover, the following linear polynomial interpretation orients all rules weakly:

$$\mathsf{f}_{\mathcal{P}(\mathcal{M})}(x) = 2x \qquad \mathsf{g}_{\mathcal{P}(\mathcal{M})}(x) = x \qquad \mathsf{h}_{\mathcal{P}(\mathcal{M})}(x, y) = x + y \qquad \mathsf{a}_{\mathcal{P}(\mathcal{M})} = 0$$

Hence, by (the proof of) Lemma 5.8, there exists a compatible 3-dimensional matrix interpretation over $>^w_{\{1,2\}}$.

Next we argue that the precondition $\mathcal{R} \subseteq \succcurlyeq_{\mathcal{P}(\mathcal{M})}$ in Lemma 5.8 is only technical in nature (to satisfy the formal definition of a well-founded monotone algebra); when it comes to termination proving power, it is actually superfluous.

▶ Remark. In the situation of Lemma 5.8, if $\mathcal{M}$ is compatible with $\mathcal{R}$, then, no matter whether $\mathcal{R} \subseteq \succcurlyeq_{\mathcal{P}(\mathcal{M})}$ or not, $\Psi(\mathcal{M})$ always establishes termination of $\mathcal{R}$ by proving the absence of infinite rewrite sequences of ground terms (assuming that the associated signature contains at least one constant symbol). This follows from Corollary 5.5 and compatibility of $\mathcal{M}$ with $\mathcal{R}$. Also note that in the proof of Lemma 5.8 the only purpose of the condition $\mathcal{R} \subseteq \succcurlyeq_{\mathcal{P}(\mathcal{M})}$ is to ensure a weak decrease in the first components of the vectors associated with some rewrite rule. However, for ground terms, we always have a weak decrease by Corollary 5.5.

Finally, we remark that by doubling the dimension of $\mathcal{M}$ these technicalities can be resolved.

▶ **Lemma 5.11.** *Compatibility of a TRS $\mathcal{R}$ with an n-dimensional matrix interpretation over $>^w_{\mathcal{I}}$, $\mathcal{I} \subseteq \{1,\dots,n\}$, implies compatibility with a $2n$-dimensional matrix interpretation over $>^w_{\mathcal{J}}$, where $|\mathcal{J}| = 2 \cdot |\mathcal{I}|$, $\mathcal{J} \subseteq \{1,\dots,2n\}$.*

In order to prove Lemma 5.11, we introduce a construction that combines two matrix interpretations $\mathcal{M}$ and $\mathcal{N}$ (not necessarily of the same dimension) to a matrix interpretation $\Pi(\mathcal{M},\mathcal{N})$ as follows. Assuming $\mathcal{M}$ consists of interpretation functions $f_{\mathcal{M}}(\vec{x}_1,\dots,\vec{x}_k) = \sum_{i=1}^k F_i\vec{x}_i + \vec{f}$ and $\mathcal{N}$ of interpretation functions $f_{\mathcal{N}}(\vec{x}_1,\dots,\vec{x}_k) = \sum_{i=1}^k \widetilde{F}_i\vec{x}_i + \tilde{f}$, the $\Pi(\mathcal{M},\mathcal{N})$-interpretation of each k-ary function symbol f is

$$f_{\Pi(\mathcal{M},\mathcal{N})}(\vec{x}_1,\dots,\vec{x}_k) = \sum_{i=1}^{k} \begin{pmatrix} F_i & 0 \\ 0 & \widetilde{F}_i \end{pmatrix} \vec{x}_i + \begin{pmatrix} \vec{f} \\ \tilde{f} \end{pmatrix}$$

▶ **Lemma 5.12.** *Let t be an arbitrary term. Then for all variable assignments α and β,*

$$[\gamma]_{\Pi(\mathcal{M},\mathcal{N})}(t) = \begin{pmatrix} [\alpha]_{\mathcal{M}}(t) \\ [\beta]_{\mathcal{N}}(t) \end{pmatrix} \text{ for the variable assignment } \gamma(x) = \begin{pmatrix} \alpha(x) \\ \beta(x) \end{pmatrix}.$$

Proof. By induction on the structure of t. ◀

Proof of Lemma 5.11. Assuming that $\mathcal{M}$ is an n-dimensional matrix interpretation over $>^w_{\mathcal{I}}$ compatible with $\mathcal{R}$, we show that $\Pi(\mathcal{M},\mathcal{M})$ is compatible as well. To this end, we let $\mathcal{J} = \mathcal{I} \cup \{\, x+n \mid x \in \mathcal{I} \,\}$ and reason as follows. By assumption, all interpretation functions of $\mathcal{M}$ are monotone with respect to $>^w_{\mathcal{I}}$, that is, for each matrix $M \in \mathcal{M}$, all column sums of $(M)_{\mathcal{I}}$ are at least one according to Lemma 4.1. By construction of $\Pi(\mathcal{M},\mathcal{M})$, this implies that for each matrix $M' \in \Pi(\mathcal{M},\mathcal{M})$, all column sums of $(M')_{\mathcal{J}}$ are also at least one. Hence, all interpretation functions of $\Pi(\mathcal{M},\mathcal{M})$ are monotone with respect to $>^w_{\mathcal{J}}$. As to compatibility of $\Pi(\mathcal{M},\mathcal{M})$ with $\mathcal{R}$, for any rewrite rule $\ell \to r$, $[\gamma]_{\Pi(\mathcal{M},\mathcal{M})}(\ell) >^w_{\mathcal{J}} [\gamma]_{\Pi(\mathcal{M},\mathcal{M})}(r)$ holds for all variable assignments γ if and only if

$$\begin{pmatrix} [\alpha]_{\mathcal{M}}(\ell) \\ [\beta]_{\mathcal{M}}(\ell) \end{pmatrix} >^w_{\mathcal{J}} \begin{pmatrix} [\alpha]_{\mathcal{M}}(r) \\ [\beta]_{\mathcal{M}}(r) \end{pmatrix} \text{ for all variable assignments } \alpha \text{ and } \beta \text{ (cf. Lemma 5.12).}$$

But this follows directly from compatibility of $\mathcal{M}$ with $\mathcal{R}$ since $[\alpha]_{\mathcal{M}}(\ell) >^w_{\mathcal{I}} [\alpha]_{\mathcal{M}}(r)$ for all assignments α. ◀

Summarizing the above results, every TRS that can be proved terminating by a matrix interpretation over $>^w_{\mathcal{I}}$, for some index set $\mathcal{I}$, can also be proved terminating by a matrix interpretation over $>^w_{\mathcal{J}}$, for a larger index set $\mathcal{J}$, at the expense of an increased dimension.

Next we elaborate on the converse of this statement. To this end, let us consider some TRS $\mathcal{R}$ and a compatible n-dimensional matrix interpretation $\mathcal{M}$ over $>^w_{\mathcal{I}}$, where $|\mathcal{I}| > 1$, consisting of interpretation functions $f_{\mathcal{M}}(\vec{x}_1,\dots,\vec{x}_k) = \sum_{i=1}^k F_i\vec{x}_i + \vec{f}$ for each k-ary function symbol f in the signature. Our aim is to show that $\mathcal{R}$ is also compatible with a matrix interpretation over $>^w_{\{1\}}$ (or more generally, $>^w_{\mathcal{I}}$ for a singleton index set $\mathcal{I}$), albeit with a higher dimension.

First, we transform $\mathcal{M}$ into $\mathcal{M}' := \Psi(\mathcal{M})$, which is in turn transformed into $\mathcal{M}'' := \Phi_P(\mathcal{M}')$ for $P = I + U$, where I is the identity matrix and U is all zero except for the entries $U_{1,i+1} = 1$, for all $i \in \mathcal{I}$. As $P^{-1} = I - U$ is not non-negative, we have to ensure well-definedness of $\mathcal{M}''$, that is, make sure that all its matrices are non-negative. Now for any matrix (vector) M, PM is equal to M except for the first row, which is the sum of the rows of M with indices in $\{1\} \cup \{\, i+1 \mid i \in \mathcal{I} \,\}$. Hence,

$$P \begin{pmatrix} f_i & 0 \\ 0 & F_i \end{pmatrix} = \begin{pmatrix} f_i & \sum_{c\in\mathcal{I}}(F_i)_{c1} & \cdots & \sum_{c\in\mathcal{I}}(F_i)_{cn} \\ 0 & (F_i)_{-1} & \cdots & (F_i)_{-n} \end{pmatrix}$$

Multiplying this matrix by P^{-1} from the right has the effect of subtracting its first column from the columns with indices in $\{\, i+1 \mid i \in \mathcal{I} \,\}$, thus replacing $\sum_{c\in\mathcal{I}}(F_i)_{cj}$ by $\sum_{c\in\mathcal{I}}(F_i)_{cj} - f_i$ for all indices $j \in \mathcal{I}$ in the above representation. As these are the only entries that may eventually be negative, $\sum_{c\in\mathcal{I}}(F_i)_{cj} - f_i \geqslant 0$ for all $j \in \mathcal{I}$ implies well-definedness of $\mathcal{M}''$. Note, however, that if all the f_i's introduced by the transformation Ψ are one, then the latter condition is satisfied without further ado because, by assumption, all interpretation functions of $\mathcal{M}$ are monotone with respect to $>^w_{\mathcal{I}}$; hence, for all $j \in \mathcal{I}$, $\sum_{c\in\mathcal{I}}(F_i)_{cj}$ is at least one according to Lemma 4.1. Moreover, note that the top-left entry of each matrix occurring in $\mathcal{M}''$ is positive since $f_i > 0$. Consequently, all interpretation functions of $\mathcal{M}''$ are monotone with respect to $>^w_{\{1\}}$.

As to compatibility of $\mathcal{M}''$ with $\mathcal{R}$, let $\ell \to r$ be an arbitrary rule in $\mathcal{R}$, and let $[\alpha]_{\mathcal{M}}(\ell) = L_1\alpha(x_1) + \cdots + L_m\alpha(x_m) + \vec{l}$ and $[\alpha]_{\mathcal{M}}(r) = R_1\alpha(x_1) + \cdots + R_m\alpha(x_m) + \vec{r}$, where $x_1, \ldots, x_m$ are the variables occurring in ℓ and r. Likewise, let $[\beta]_{\mathcal{P}(\mathcal{M})}(\ell) = l_1\beta(x_1) + \cdots + l_m\beta(x_m)$, where $l_1, \ldots, l_m \in \mathbb{N}$, and similarly for $[\beta]_{\mathcal{P}(\mathcal{M})}(r)$. By compatibility of $\mathcal{M}$, we have $\vec{l} >^w_{\mathcal{I}} \vec{r}$ and $L_i \geqslant R_i$ for $i = 1, \ldots, m$ (cf. Lemma 4.4). Moreover, by Lemmata 5.2 and 5.6,

$$[\gamma]_{\mathcal{M}''}(\ell) = \sum_{i=1}^{m} P \begin{pmatrix} l_i & 0 \\ 0 & L_i \end{pmatrix} P^{-1}\gamma(x_i) + P \begin{pmatrix} 0 \\ \vec{l} \end{pmatrix} \text{ for } \gamma\colon \mathcal{V} \to \mathbb{N}^{n+1},\ x \mapsto \begin{pmatrix} \beta(x) \\ \alpha(x) \end{pmatrix}.$$

Therefore, $[\gamma]_{\mathcal{M}''}(\ell) >^w_{\{1\}} [\gamma]_{\mathcal{M}''}(r)$ holds for all variable assignments γ if and only if

$$P \begin{pmatrix} 0 \\ \vec{l} \end{pmatrix} >^w_{\{1\}} P \begin{pmatrix} 0 \\ \vec{r} \end{pmatrix} \text{ and } P \begin{pmatrix} l_i - r_i & 0 \\ 0 & L_i - R_i \end{pmatrix} P^{-1} \geqslant 0 \text{ for } i = 1, \ldots, m.$$

The first condition follows directly from $\vec{l} >^w_{\mathcal{I}} \vec{r}$ and the shape of P. Concerning the second condition, we first rewrite the corresponding matrix to

$$\begin{pmatrix} l_i - r_i & \sum_{c\in\mathcal{I}}(L_i - R_i)_{c1} & \cdots & \sum_{c\in\mathcal{I}}(L_i - R_i)_{cn} \\ 0 & (L_i - R_i)_{-1} & \cdots & (L_i - R_i)_{-n} \end{pmatrix} P^{-1}.$$

Using the fact that $L_i \geqslant R_i$, the entire matrix is non-negative if and only if, for $i = 1, \ldots, m$, $l_i \geqslant r_i$ and $\sum_{c\in\mathcal{I}}(L_i - R_i)_{cj} \geqslant l_i - r_i$ for all $j \in \mathcal{I}$.

Based on these observations, we establish the following lemma.

▶ **Lemma 5.13.** *Let $\mathcal{M}$ be an n-dimensional matrix interpretation over $>^w_{\mathcal{I}}$, $\mathcal{I} \subseteq \{1, \ldots, n\}$, such that $|\mathcal{I}| > 1$, and let $\mathcal{R}$ be a TRS satisfying $\mathcal{R} \subseteq \succcurlyeq_{\mathcal{P}(\mathcal{M})}$. Moreover, assume that for each k-ary function symbol f, all column sums of each $(F_i)_{\mathcal{I}}$ are greater than or equal to f_i for all $i \in \{1, \ldots, k\}$, and that for each $\ell \to r \in \mathcal{R}$, all column sums of each $(L_i - R_i)_{\mathcal{I}}$ are greater than or equal to $l_i - r_i$ for all $i \in \{1, \ldots, m\}$. Then compatibility of $\mathcal{R}$ with $\mathcal{M}$ implies compatibility with an $(n+1)$-dimensional matrix interpretation over $>^w_{\mathcal{J}}$, where $|\mathcal{J}| = 1$, $\mathcal{J} \subseteq \{1, \ldots, n+1\}$.* ◀

▶ **Corollary 5.14.** *Let $\mathcal{M}$ be an n-dimensional matrix interpretation over $>^w_{\mathcal{I}}$, $\mathcal{I} \subseteq \{1, \ldots, n\}$, such that $|\mathcal{I}| > 1$, and let $\mathcal{R}$ be a non-duplicating TRS. Moreover, assume that for each $\ell \to r \in \mathcal{R}$, all column sums of each $(L_i - R_i)_{\mathcal{I}}$ are greater than or equal to $l_i - r_i$ for all $i \in \{1, \ldots, m\}$. Then compatibility of $\mathcal{R}$ with $\mathcal{M}$ implies compatibility with an $(n+1)$-dimensional matrix interpretation over $>^w_{\mathcal{J}}$, where $|\mathcal{J}| = 1$, $\mathcal{J} \subseteq \{1, \ldots, n+1\}$.*

Proof. Setting all the f_i's introduced by $\Psi(\mathcal{M})$ to one, the condition $\mathcal{R} \subseteq \succcurlyeq_{\mathcal{P}(\mathcal{M})}$ becomes equivalent to $\mathcal{R}$ being non-duplicating according to Lemma 5.7. Moreover, all column sums of each $(F_i)_{\mathcal{I}}$ are greater than or equal to $f_i = 1$ because all interpretation functions of $\mathcal{M}$ are monotone with respect to $>^w_{\mathcal{I}}$. ◀

By restricting the class of non-duplicating TRSs further, we can get rid of the condition that all column sums of $(L_i - R_i)_{\mathcal{I}}$ are greater than or equal to $l_i - r_i$.

▶ **Corollary 5.15.** *Let $\mathcal{R}$ be a TRS, such that for all $\ell \to r \in \mathcal{R}$, $|\ell|_x = |r|_x$ for all variables x. Then compatibility of $\mathcal{R}$ with an n-dimensional matrix interpretation over $>^w_{\mathcal{I}}$, where $|\mathcal{I}| > 1$ and $\mathcal{I} \subseteq \{1, \ldots, n\}$, implies compatibility with an $(n+1)$-dimensional matrix interpretation over $>^w_{\mathcal{J}}$, where $|\mathcal{J}| = 1$, $\mathcal{J} \subseteq \{1, \ldots, n+1\}$.* ◀

▶ Remark. Let all the f_i's introduced by $\Psi(\mathcal{M})$ be one. Then, in the situation of Lemma 5.13, if $\mathcal{M}$ is compatible with $\mathcal{R}$, then, no matter whether the other preconditions mentioned in the lemma are satisfied or not, $\mathcal{M}''$ always establishes termination of $\mathcal{R}$ by proving the absence of infinite rewrite sequences of ground terms. This can be seen as follows. Assume to the contrary that $t_1 \to t_2 \to t_3 \to \cdots$ is such an infinite sequence. By Corollaries 5.3 and 5.5, we have

$$[\gamma]_{\mathcal{M}''}(t_i) = P \cdot [\gamma]_{\Psi(\mathcal{M})}(t_i) = P \cdot \begin{pmatrix} 0 \\ [\alpha]_{\mathcal{M}}(t_i) \end{pmatrix} = \begin{pmatrix} \sum_{c \in \mathcal{I}} ([\alpha]_{\mathcal{M}}(t_i))_c \\ [\alpha]_{\mathcal{M}}(t_i) \end{pmatrix}$$

From this and from compatibility of $\mathcal{M}$ with $\mathcal{R}$, we conclude that $[\gamma]_{\mathcal{M}''}(t_i) >^w_{\{1\}} [\gamma]_{\mathcal{M}''}(t_{i+1})$ holds for all $i \in \mathbb{N} \setminus \{0\}$ because of $[\alpha]_{\mathcal{M}}(t_i) >^w_{\mathcal{I}} [\alpha]_{\mathcal{M}}(t_{i+1})$. However, this contradicts well-foundedness of $>^w_{\{1\}}$.

6 Matrix Interpretations and Non-weakly Decreasing Orders

In this section we investigate the usefulness of the orders $>_\Sigma$, $>_\ell$, $>_m$ and $>_{\mathcal{I}}$ (where $\mathcal{I}$ is a singleton set) introduced in Subsection 3.2 for building matrix interpretations on top of them. As these orders originated from the orders introduced in Definition 3.1 by dropping the property of weak decreasingness, each of them obviously subsumes its ancestor, e.g., $>^w_\Sigma \subset >_\Sigma$, so that one is tempted to believe that these more general base orders would induce more powerful kinds of matrix interpretations. However, as already mentioned at the end of Section 4, an inclusion like $>^w_\Sigma \subset >_\Sigma$ does not necessarily propagate to the corresponding notions of matrix interpretations because of the monotonicity requirement all interpretation functions have to satisfy. Indeed, it turns out that the monotonicity conditions with respect to $>_\Sigma$, $>_\ell$, $>_m$ and $>_{\mathcal{I}}$ are much stronger than the ones associated with their respective weakly decreasing counterparts, ultimately resulting in weaker notions of matrix interpretations. In particular, we will see that matrix interpretations over $>_{\mathcal{I}}$ and $>_\Sigma$ are equivalent to linear polynomial interpretations.

As already mentioned in Subsection 3.2, all four orders are equal to $>_{\mathbb{N}}$ when the dimension n is one. Hence, matrix interpretations based on them are at least as powerful as linear polynomial interpretations. Next we show that matrix interpretations over $>_{\mathcal{I}}$ and $>_\Sigma$ are no more powerful than linear polynomial interpretations. Since matrix interpretations are invariant under permutations, we consider the index set $\mathcal{I} = \{1\}$ without loss of any generality.

▶ **Lemma 6.1.** *Let $f(\vec{x}_1, \ldots, \vec{x}_k) = \sum_{i=1}^k F_i \vec{x}_i + \vec{f}$, where $\vec{f} \in \mathbb{N}^n$ and $F_1, \ldots, F_k \in \mathbb{N}^{n \times n}$. Then $f(\vec{x}_1, \ldots, \vec{x}_k)$ is monotone with respect to $>_{\{1\}}$ if and only if for each F_i, $i = 1, \ldots, k$, $(F_i)_{11} \geqslant 1$ and $(F_i)_{12} = \cdots = (F_i)_{1n} = 0$.* ◀

Intuitively, this means that the first component of a function application $f(\vec{x}_1, \ldots, \vec{x}_k)$ only depends on the respective first components of its arguments, not on the other components. Based on this observation and the fact that for comparisons with $>_{\{1\}}$ only the first

components matter, we associate the following linear polynomial interpretation $\mathcal{P}$ with a given matrix interpretation $\mathcal{M}$ over $>_{\{1\}}$. For each k-ary function symbol f, if $f_{\mathcal{M}}(\vec{x}_1, \dots, \vec{x}_k) = \sum_{i=1}^{k} F_i\vec{x}_i + \vec{f}$ is its interpretation in $\mathcal{M}$, with all matrices satisfying the conditions of Lemma 6.1, then we define its $\mathcal{P}$-interpretation as $f_{\mathcal{P}}(x_1, \dots, x_k) = \sum_{i=1}^{k} (F_i)_{11}\, x_i + (\vec{f})_1$, which is monotone because $(F_i)_{11} \geqslant 1$. By construction, the $\mathcal{P}$-interpretation of an arbitrary term coincides with the first component of its $\mathcal{M}$-interpretation. The straightforward induction proof is omitted.

▶ **Lemma 6.2.** *Let $\mathcal{M}$ be a matrix interpretation over $>_{\{1\}}$ of dimension n, $\mathcal{P}$ the associated linear polynomial interpretation as described above and t an arbitrary term. Then for any variable assignment $\alpha\colon \mathcal{V} \to \mathbb{N}^n$, $\pi_1([\alpha]_{\mathcal{M}}(t)) = [\pi_1 \circ \alpha]_{\mathcal{P}}(t)$, where π_1 projects a vector to its first component.* ◀

Therefore, any rewrite rule $\ell \to r$ that is orientable by $\mathcal{M}$, is also orientable by $\mathcal{P}$ (since $\pi_1 \circ \alpha$ covers all assignments $\mathcal{V} \to \mathbb{N}$), which shows that matrix interpretations over $>_{\{1\}}$ ($>_{\mathcal{I}}$) are no more powerful than linear polynomial interpretations. The following lemma states that this is also the case for matrix interpretations over $>_{\Sigma}$.

▶ **Lemma 6.3.** *Let $f(\vec{x}_1, \dots, \vec{x}_k) = \sum_{i=1}^{k} F_i\vec{x}_i + \vec{f}$, where $\vec{f} \in \mathbb{N}^n$ and $F_1, \dots, F_k \in \mathbb{N}^{n\times n}$. Then $f(\vec{x}_1, \dots, \vec{x}_k)$ is monotone with respect to $>_{\Sigma}$ if and only if for each F_i, $i = 1, \dots, k$, all column sums are equal and at least one.* ◀

In analogy to the treatment of matrix interpretations over $>_{\{1\}}$, given an n-dimensional matrix interpretation $\mathcal{M}$ over $>_{\Sigma}$, we again associate a linear polynomial interpretation $\mathcal{P}$ with $\mathcal{M}$ as follows. For each k-ary function symbol f, if $f_{\mathcal{M}}(\vec{x}_1, \dots, \vec{x}_k) = \sum_{i=1}^{k} F_i\vec{x}_i + \vec{f}$ is its interpretation in $\mathcal{M}$, with all matrices satisfying the conditions of Lemma 6.3, then its $\mathcal{P}$-interpretation is defined as $f_{\mathcal{P}}(x_1, \dots, x_k) = \sum_{i=1}^{k} F_i^{\Sigma} x_i + \sum_{j=1}^{n} (\vec{f})_j$, where F_i^{Σ} denotes the column sum of F_i, which is equal for all columns of F_i and at least one; hence, $f_{\mathcal{P}}$ is monotone. By construction, the $\mathcal{P}$-interpretation of an arbitrary term coincides with the sum of the components of its $\mathcal{M}$-interpretation.

▶ **Lemma 6.4.** *Let $\mathcal{M}$ be a matrix interpretation over $>_{\Sigma}$ of dimension n, $\mathcal{P}$ the associated linear polynomial interpretation as described above and t an arbitrary term. Then for any variable assignment $\alpha\colon \mathcal{V} \to \mathbb{N}^n$, $\sum_{j=1}^{n} ([\alpha]_{\mathcal{M}}(t))_j = [\alpha']_{\mathcal{P}}(t)$, where $\alpha'(x) = \sum_{j=1}^{n} (\alpha(x))_j$ for all $x \in \mathcal{V}$.* ◀

So, if a rewrite rule $\ell \to r$ is orientable by $\mathcal{M}$, i.e., $[\alpha]_{\mathcal{M}}(\ell) >_{\Sigma} [\alpha]_{\mathcal{M}}(r)$ for all variable assignments α, then it is also orientable by $\mathcal{P}$ (since α' covers all assignments $\mathcal{V} \to \mathbb{N}$), which shows that matrix interpretations over $>_{\Sigma}$ are no more powerful than linear polynomial interpretations.

Finally, concerning matrix interpretations over $>_m$ and $>_\ell$, the situation is similar as for $>_{\Sigma}$ and $>_{\mathcal{I}}$. That is to say that the respective monotonicity conditions are too strong, thus reducing the set of potential interpretation functions down to a size that renders matrix interpretations over $>_m$ and $>_\ell$ useless. For example, one can show that for monotonicity of a function $A\vec{x} + \vec{b}$ with respect to $>_m$, it is necessary that the matrix A satisfies the conditions of Lemma 6.3, that is, all column sums of A must be equal and at least one; e.g., by considering vectors $\vec{x}$ and $\vec{y}$, such that all components of $\vec{y}$ are equal to some $y \in \mathbb{N}$ and $\vec{x}$ is zero everywhere except for its j-th component, $j \in \{1, \dots, n\}$, which contains the value $y + 1$. Similarly, one can show that for monotonicity of $A\vec{x} + \vec{b}$ with respect to $>_\ell$, it is necessary that all column vectors of A are non-zero and have the same (Euclidean) length; e.g., for dimension 2 and higher, by considering vectors $\vec{x} = (y \mp 1, y \pm 1, 0, \dots, 0)^T$ and

$\vec{y} = (y, y, 0, \ldots, 0)^T$, where $y \in \mathbb{N} \setminus \{0\}$. However, these conditions are not sufficient. Even if A is the identity matrix, $A\vec{x} + \vec{b}$ is not necessarily monotone with respect to $>_\ell$.

7 Improved Matrix Interpretations

According to the results presented in Section 5, in theory the various instances of matrix interpretations over $>^w_{\mathcal{I}}$ are all equivalent *somehow* with respect to termination proving power if there is no bound on the dimension of the matrices. In practice, however, due to computational restrictions the dimension is limited. But then the various instances of matrix interpretations over $>^w_{\mathcal{I}}$ are incomparable as witnessed by Example 5.1 and by the experiments we performed. Therefore, an implementation should try all instances (cf. also [3]). Apart from parallelization, one could try to combine the constraints associated with each instance into a single disjunctive constraint and let the constraint solver figure out which instance to pursue. This approach was chosen in [3]. However, according to our experiments, it does not yield an efficient implementation (cf. experimental results below). Therefore, we propose a different approach, which generalizes traditional matrix interpretations.

Given some signature $\mathcal{F}$, we define an $\mathcal{F}$-algebra $\mathcal{M}$ with carrier $\mathbb{N}^n$, where each k-ary function symbol $f \in \mathcal{F}$ is interpreted by a linear function as in Definition 4.2 (without the monotonicity requirement). Concerning monotonicity, we demand that

- all functions are monotone with respect to $>^w_{\mathcal{I}_1}$, or
- all functions are monotone with respect to $>^w_{\mathcal{I}_2}$, or
- $\vdots$
- all functions are monotone with respect to $>^w_{\mathcal{I}_n}$.

Compatibility with a given TRS $\mathcal{R}$ is established by demanding that for every rewrite rule $\ell \to r \in \mathcal{R}$, $[\alpha]_{\mathcal{M}}(\ell) >^w_{\mathcal{I}_1} [\alpha]_{\mathcal{M}}(r)$ for all variable assignments α; i.e., every rewrite rule gives rise to a strict decrease in the first components of the vectors associated with it.

Clearly, if all interpretation functions of $\mathcal{M}$ are monotone with respect to $>^w_{\mathcal{I}_1}$, then $\mathcal{M}$ corresponds to a traditional matrix interpretation [4]. More generally, $\mathcal{M}$ always is a matrix interpretation over $>^w_{\mathcal{I}_d}$, $d \in \{1, \ldots, n\}$, in the sense of Definition 4.2 because of the inclusions $>^w_{\mathcal{I}_1} \subset >^w_{\mathcal{I}_2} \subset \cdots \subset >^w_{\mathcal{I}_n}$.

Next we provide some experimental data. We implemented the variants of matrix interpretations considered in this paper in the termination prover TTT2 [11] and analyzed their performance on TPDB[3] version 7.0.2. All tests have been performed on a laptop equipped with 2 GB of main memory and one dual-core INTEL® Core 2 Duo T7500 processor running at a clock rate of 2.2 GHz with a time limit of 60 seconds per system.[4]

Table 1 summarizes our results for establishing direct termination (using matrix interpretations as a stand-alone method). We searched for matrix interpretations of dimensions two and three by encoding the constraints as an SMT problem (quantifier-free non-linear arithmetic), which is solved by bit-blasting. The table lists the number of bits used to represent matrix/vector coefficients, the number of bits for intermediate results is one higher than that. The entry $>^{\text{ext}}_{\{1\}}$ in the first column refers to the notion of matrix interpretations presented above, whereas the entry [3] refers to the approach proposed in [3]. For the experiments presented in the table the time limit was hardly ever consumed. Typically, a termination proof is obtained in about 2 (5) seconds for dimension 2 (3). For dimensions 4

[3] Termination Problems Data Base, `http://termcomp.uibk.ac.at`.
[4] For full details see `http://colo6-c703.uibk.ac.at/ttt2/fn/matrix`.

Table 1 Experimental results for various matrix interpretations.

method	dimension	# bits	SCORE
$>^{w}_{\mathcal{I}_1}$	2	3	242
$>^{w}_{\mathcal{I}_2}$	2	3	247
$>^{\text{ext}}_{\{1\}}$	2	3	254
[3]	2	3	250

method	dimension	# bits	SCORE
$>^{w}_{\mathcal{I}_1}$	3	2|3	266|285
$>^{w}_{\mathcal{I}_2}$	3	2|3	252|264
$>^{w}_{\mathcal{I}_3}$	3	2|3	249|269
$>^{\text{ext}}_{\{1\}}$	3	2|3	276|287
[3]	3	2|3	267|270

and higher, however, there are many more timeouts, resulting in inferior performance scores; e.g., for matrix interpretations over $>^{w}_{\mathcal{I}_1}$ of dimension 4 (with 3 bits) one loses more than 40 of the 285 systems for dimension 3.

8 Conclusion and Future Work

In this paper we studied various alternative well-founded orders on vectors of natural numbers based on vector norms. Most of them turned out to be equivalent to or subsumed by an instance of $>^{w}_{\mathcal{I}}$, an order which already appeared in [3]. In this respect, our main contribution are the theoretical comparisons presented in Section 5, as well as the variant of matrix interpretations introduced in Section 7. We do note, however, that the situation is altogether different when switching from the natural numbers to the rationals and reals. Then it is not the case anymore that almost all of the orders of Section 3 (suitably adapted) are equivalent. In particular, one could imagine interpretation functions, all of whose matrices have entries less than one, but which are still monotone. We leave this issue for the near future. In this context, we also mention the recent work of Lucas [12] where an attempt is made to simulate matrix interpretation over the rationals by an interpretation over the naturals.

We also plan to investigate on the ramifications of the kinds of matrix interpretations proposed in this paper with respect to recent results on the derivational complexity of TRSs [13]. For example, if a matrix has a diagonal of all zeros, then its trace, the sum of the diagonal entries, is also zero. As the trace of a matrix is the sum of its eigenvalues, which have been shown to be the determining factor for the derivational complexity of TRSs, a lower trace might be beneficial in this context.

In the near future work we will address alternative matrix interpretations in the context of the DP framework [6], where it suffices to consider weakly monotone algebras. A *well-founded weakly monotone $\mathcal{F}$-algebra* $(\mathcal{A}, >, \gtrsim)$ is an $\mathcal{F}$-algebra $\mathcal{A}$ equipped with two relations $>, \gtrsim$ on A, such that $>$ is well-founded, $> \cdot \gtrsim \subseteq >$, and for every $f \in \mathcal{F}$, $f_{\mathcal{A}}$ is monotone with respect to $\gtrsim$. If, in addition, $f_{\mathcal{A}}$ is also monotone with respect to $>$ (for every $f \in \mathcal{F}$), then we obtain an *extended monotone $\mathcal{F}$-algebra* [4], the analogon of well-founded monotone algebras in the context of relative termination.

Based on the results of the previous sections, the following instances of a weakly monotone algebra $(\mathbb{N}^n, >, \gtrsim)$, where $\gtrsim = \geqslant^{w}$ and (1) $> = >^{w}_{\mathcal{I}}$ for $\mathcal{I} = \{1, \ldots, n\}$, (2) $> = >_{\Sigma}$, (3) $> = >_{m}$, or (4) $> = >_{\ell}$ need to be considered. As to the first instance, we note that $>$ is the strict part of $\gtrsim$ according to Lemma 3.3. Yet this is exactly the case that is considered in [4], apart from a refinement that reduces the search space in an implementation. Moreover, by Corollary 3.4, no other weakly decreasing orders need to be considered for $>$. However, observing that weak decreasingness is not really needed to obtain a weakly monotone algebra, one might as well drop it, thus obtaining a weakly monotone algebra,

where $> = >_\Sigma$ (instance (2) above), which is a proper generalization of the first one since $>^w_{\mathcal{I}} = >^w_\Sigma \subset >_\Sigma$ (cf. Lemma 3.3). Similarly, one can use the non-weakly decreasing orders $>_m$ and $>_\ell$ to obtain other instances of weakly monotone algebras. They are all incomparable since $>_\Sigma$, $>_m$ and $>_\ell$ are so.

Acknowledgements We thank Bertram Felgenhauer for his helpful comments in the early stages of this work.

References

1 B. Alarcón, S. Lucas, and R. Navarro-Marset. Proving termination with matrix interpretations over the reals. In *Proc. 10th International Workshop on Termination (WST 2009)*, pages 12–15, 2009.

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

3 P. Courtieu, G. Gbedo, and O. Pons. Improved matrix interpretation. In *Proc. 36th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM 2010)*, volume 5901 of *LNCS*, pages 283–295, 2010.

4 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.

5 A. Gebhardt, D. Hofbauer, and J. Waldmann. Matrix evolutions. In *Proc. 9th International Workshop on Termination (WST 2007)*, pages 4–8, 2007.

6 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 11)*, volume 3452 of *LNAI*, pages 301–331, 2005.

7 D. Hofbauer. Termination proofs by context-dependent interpretations. In *Proc. 12th International Conference on Rewriting Techniques and Applications (RTA 2001)*, volume 2051 of *LNCS*, pages 108–121, 2001.

8 D. Hofbauer and J. Waldmann. Termination of $\{aa \to bc, bb \to ac, cc \to ab\}$. *Information Processing Letters*, 98(4):156–158, 2006.

9 D. Hofbauer and J. Waldmann. Termination of string rewriting with matrix interpretations. In *Proc. 17th International Conference on Rewriting Techniques and Applications (RTA 2006)*, volume 4098 of *LNCS*, pages 328–342, 2006.

10 R. A. Horn and C. R. Johnson. *Matrix Analysis.* Cambridge University Press, 1990.

11 M. Korp, C. Sternagel, H. Zankl, and A. Middeldorp. Tyrolean Termination Tool 2. In *Proc. 20th International Conference on Rewriting Techniques and Applications (RTA 2009)*, volume 5595 of *LNCS*, pages 295–304, 2009.

12 S. Lucas. From matrix interpretations over the rationals to matrix interpretations over the naturals. In *Proc. 10th International Conference on Artificial Intelligence and Symbolic Computation (AISC 2010)*, volume 6167 of *LNCS*, pages 116–131, 2010.

13 F. Neurauter, H. Zankl, and A. Middeldorp. Revisiting matrix interpretations for polynomial derivational complexity of term rewriting. In *Proc. 17th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 17)*, volume 6397 of *LNCS*, pages 550–564, 2010.

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

15 H. Zankl and A. Middeldorp. Satisfiability of non-linear (ir)rational arithmetic. In *Proc. 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 16)*, volume 6355 of *LNCS*, 2010.

Soundness of Unravelings for Deterministic Conditional Term Rewriting Systems via Ultra-Properties Related to Linearity

Naoki Nishida[1], Masahiko Sakai[1], and Toshiki Sakabe[1]

1 **Graduate School of Information Science, Nagoya University**
Furo-cho, Chikusa-ku, 4648603 Nagoya, Japan
{nishida,sakai,sakabe}@is.nagoya-u.ac.jp

Abstract

Unravelings are transformations from a conditional term rewriting system (CTRS, for short) over an original signature into an unconditional term rewriting systems (TRS, for short) over an extended signature. They are not *sound* for every CTRS *w.r.t. reduction*, while they are complete w.r.t. reduction. Here, soundness w.r.t. reduction means that every reduction sequence of the corresponding unraveled TRS, of which the initial and end terms are over the original signature, can be simulated by the reduction of the original CTRS. In this paper, we show that an optimized variant of Ohlebusch's unraveling for deterministic CTRSs is sound w.r.t. reduction if the corresponding unraveled TRSs are left-linear or both right-linear and non-erasing. We also show that soundness of the variant implies that of Ohlebusch's unraveling.

1998 ACM Subject Classification F.4.2 Grammars and Other Rewriting Systems

Keywords and phrases conditional term rewriting, program transformation

Digital Object Identifier 10.4230/LIPIcs.RTA.2011.267

Category Regular Research Paper

1 Introduction

Unravelings are transformations from a conditional term rewriting system (CTRS, for short) over an original signature into an unconditional term rewriting system (TRS, for short) over an extended signature, that are complete for every CTRS w.r.t. the simulation of reduction sequences of the CTRS [11], i.e., every reduction sequence of the CTRS can be simulated by the reduction of the corresponding unraveled TRS. The unraveled TRSs are approximations of the original CTRSs and they are useful in analyzing properties of the CTRSs, such as syntactic properties, modularity and operational termination, since TRSs are much easier to handle than CTRSs. Marchiori has proposed unravelings for *join* and *normal* CTRSs in order to analyze *ultra-properties* and *modularity* of the CTRSs [11], and he has also proposed an unraveling for *deterministic* CTRSs (DCTRS, for short) [12]. Ohlebusch has presented an improved variant of Marchiori's unraveling for DCTRSs in order to analyze termination of logic programs [20]. Termination of the unraveled TRSs is a practical sufficient-condition for proving *operational termination* of the original CTRSs [10]. A variant of Ohlebusch's unraveling for DCTRSs has been proposed in [14] and [4] (cf. [19, 18, 3]). This variant is sometimes called *optimized* in the sense that the variable-carrying arguments of *U symbols* introduced via the application of the unraveling are optimized.

Although the mechanism of unconditional rewriting is much simpler than that of conditional rewriting, the reduction of the unraveled TRSs has never been used as an alternative

22nd International Conference on Rewriting Techniques and Applications (RTA'11).
Editor: M. Schmidt-Schauß; pp. 267–282

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

to that of the original CTRSs in order to simulate reduction sequences of the CTRSs. This is because unravelings are not *sound* for every CTRS *w.r.t. reduction* [11, 20]. Here, *soundness w.r.t. reduction* (simply, *soundness*) means that every reduction sequence of the unraveled TRSs, of which the initial and end terms are over the original signatures, can be simulated by the reduction of the original CTRSs [11]. It has been shown that unravelings are sound if the unraveled TRSs satisfy some syntactic properties or if appropriate reduction strategies are introduced to the reduction of the unraveled TRSs. Marchiori has shown in [11] that his unravelings for join and normal CTRSs are sound for left-linear ones, and he has also shown in [12] that his unraveling for DCTRSs is sound for *semi-linear* DCTRSs. Nishida et al. have shown in [19] that the combined reduction restriction of the *membership* [25] and *context-sensitive* [9] conditions that are determined via the application of the optimized unraveling is sufficient for soundness. Schernhammer and Gramlich have shown in [23, 22] that a similar context-sensitive restriction is sufficient for soundness of Ohlebusch's unraveling. Gmeiner et al. have shown in [5] that Marchiori's unraveling for normal CTRSs is sound for confluent, non-erasing or weakly left-linear ones. They have also given a discussion what properties are necessary or sufficient for soundness.

In this paper, we show two sufficient syntactic-conditions of DCTRSs for soundness of the optimized unraveling. One is *ultra-left-linearity w.r.t. the unraveling*, i.e., that the unraveled TRSs are left-linear. The other is the combination of *ultra-right-linearity* and *ultra-non-erasingness w.r.t. the unraveling*, i.e., that the unraveled TRSs are right-linear and non-erasing. We also provide necessary and sufficient syntactic-conditions of DCTRSs in which the corresponding unraveled TRSs are left-linear, right-linear and non-erasing, respectively. Finally, we show that soundness of the optimized unraveling implies that of Ohlebusch's unraveling, i.e., if the optimized one is sound for a DCTRS, then Ohlebusch's one is also sound for the DCTRS. A main difference to the preliminary version [17] is the result on the relationship with Ohlebusch's unraveling.

The optimized unraveling in this paper is employed in the *inversion compilers* for constructor TRSs [14, 19, 18]. The compilers transform a constructor TRS into a DCTRS defining inverses of functions defined in the constructor TRS and then unravel it into a TRS (see Example 3.3). The resulting TRS may have extra variables since the intermediate DCTRS may have extra variables that occur in the right-hand side but not in the conditional part. For this reason, this paper allows TRSs to have extra variables (called *EV-TRS*). It is allowed to instantiate extra variables with arbitrary terms in applying rewrite rules. Since many instantiated terms of extra variables are meaningless and sometimes cause non-termination, we focus on meaningful derivations by giving a restriction to reduction sequences of the resulting TRS. The restriction, called *EV-basicness* [16, 14, 17], is a relaxed variant of the *basicness* property [7, 13] of reduction sequences: if a TRS has extra variables, then any redex introduced by extra variables is not reduced anywhere in reduction sequences. Roughly speaking, in applying the inversion compilers, the resulting TRS is often right-linear (left-linear, resp.) if the input constructor TRS is left-linear (right-linear, resp.). Moreover, the resulting TRS is usually non-erasing if the target function is injective. Note that injective functions are the most interesting targets of program inversion. For these reasons, the sufficient conditions shown in this paper are very practical because they guarantee that the resulting TRS is definitely an inverse of the given constructor TRS.

As described above, Ohlebusch's unraveling is sound for any DCTRS if we introduce the particular context-sensitive restriction to the reduction of the corresponding unraveled TRSs. However, characterizing sufficient syntactic-properties for soundness without the restriction to the reduction is important for the use of the unraveled TRSs instead of the

original CTRSs since the context-sensitivity makes the reduction more complicated than the ordinary reduction. Moreover, if the unraveling is sound for the resulting TRS obtained by the inversion compilers [14, 19, 18] without context-sensitivity, then we can apply the restricted *completion* [15] to the resulting TRS in order to make it convergent. Note that when the target of the inversion compilers are injective functions, convergence of the resulting TRSs is desirable. For these reasons, soundness of unravelings without any restriction to the reduction is important in employing the reduction of the unraveled TRSs instead of that of the original CTRSs.

Finally, we briefly describe a related work that is not mentioned above. Serbanuta and Rosu have proposed a sound and complete transformation of left-linear or ground-confluent DCTRSs into TRSs where function symbols in the original signatures are completely extended, increasing their arities [24]. Their transformation is based on Viry's approach [26] that is another direction of developing transformations of CTRSs to TRSs. Rules produced by this transformation are much more complicated than those produced by unravelings. Moreover, it is not easy to know if DCTRSs are ground-confluent.

This paper is organized as follows. In Section 2, we review basic notions and notations of term rewriting. In Section 3, we review unravelings for DCTRSs and syntactic properties related to DCTRSs and the corresponding unraveled TRSs. In Section 4, we show the main results of this paper, i.e., the optimized unraveling for DCTRSs is sound if the corresponding unraveled TRSs are left-linear or both right-linear and non-erasing. We also show that these results hold for Ohlebusch's unraveling. In Section 5, we briefly describe future work on soundness of unravelings.

2 Preliminaries

In this section, we review basic notions and notations of term rewriting [2, 21].

Throughout the paper, we use $\mathcal{V}$ as a countably infinite of *variables*. Let $\mathcal{F}$ be a *signature*, a finite set of *function symbols* each of which has its own fixed arity that is denoted by $\mathsf{ar}(f)$ for a function symbol f. The set of *terms* over $\mathcal{F}$ and $\mathcal{V}$ is denoted by $\mathcal{T}(\mathcal{F},\mathcal{V})$, and the set of variables appearing in any of terms $t_1,\cdots,t_n$ is denoted by $\mathcal{V}ar(t_1,\cdots,t_n)$. The *identity* of terms s and t is written by $s \equiv t$. A term is called *linear* if any variable occurs in the term at most once. The set of *positions* of a term t is denoted by $\mathcal{P}os(t)$. The sets of positions for function symbols and for variables in t are denoted by $\mathcal{P}os_{\mathcal{F}}(t)$ and $\mathcal{P}os_{\mathcal{V}}(t)$, respectively. For a term t and a position p of t, the notation $t|_p$ represents the *subterm* of t at the position p. The function symbol at the *root* position ε of term t is denoted by $\mathsf{root}(t)$. Given an n-hole *context* $C[\ ]$ with parallel positions $p_1,\cdots,p_n$, the notation $C[t_1,\cdots,t_n]_{p_1,\cdots,p_n}$ represents the term obtained by replacing each occurrence of hole $\Box$ at position p_i with term t_i for all $1 \leq i \leq n$. We may omit the subscription $_{p_1,\cdots,p_n}$. For positions p and p' of a term, we write $p' \geq p$ if p is a prefix of p' (i.e., there exists a q' such that $pq = p'$). Moreover, we write $p' > p$ if p is a proper prefix of p'.

The *domain* and *range* of a *substitution* σ are denoted by $\mathcal{D}om(\sigma)$ and $\mathcal{R}an(\sigma)$, respectively. We may denote σ by $\{x_1 \mapsto t_1,\cdots,x_n \mapsto t_n\}$ if $\mathcal{D}om(\sigma) = \{x_1,\cdots,x_n\}$ and $\sigma(x_i) \equiv t_i$ for all $1 \leq i \leq n$. For a signature $\mathcal{F}$, the set of *substitutions* whose domains are over $\mathcal{F}$ and $\mathcal{V}$ is denoted by $\mathcal{S}ub(\mathcal{F},\mathcal{V})$: $\mathcal{S}ub(\mathcal{F},\mathcal{V}) = \{\sigma \mid \mathcal{R}an(\sigma) \subseteq \mathcal{T}(\mathcal{F},\mathcal{V})\}$. The application $\sigma(t)$ of a substitution σ to a term t is abbreviated to $t\sigma$. Given a set X of variables, $\sigma|_X$ denotes the restricted substitution of σ w.r.t. X: $\sigma|_X = \{x \mapsto x\sigma \mid x \in \mathcal{D}om(\sigma) \cap X\}$. The *composition* $\sigma\theta$ of substitutions σ and θ is defined as $x\sigma\theta = (x\sigma)\theta$.

An *oriented conditional rewrite rule* over a signature $\mathcal{F}$ is a triple (l,r,c), denoted by

$l \to r \Leftarrow c$, such that the *left-hand side* l is a non-variable term in $\mathcal{T}(\mathcal{F},\mathcal{V})$, the *right-hand side* r is a term in $\mathcal{T}(\mathcal{F},\mathcal{V})$, and the *conditional part* c is a sequence $s_1 \twoheadrightarrow t_1; \cdots ; s_k \twoheadrightarrow t_k$ ($k \geq 0$) where all of $s_1, t_1 \cdots, s_k, t_k$ are terms in $\mathcal{T}(\mathcal{F},\mathcal{V})$. In particular, the rewrite rule is called *unconditional* if the conditional part is the empty sequence (i.e., $k = 0$), and we may abbreviate it to $l \to r$. The rewrite rule is called *extended* if the condition "$l \notin \mathcal{V}$" is not imposed. We sometimes attach a unique label ρ to the rewrite rule $l \to r \Leftarrow c$ by denoting $\rho : l \to r \Leftarrow c$, and we use the label to refer to the rewrite rule. The set of variables in c and in ρ are denoted by $\mathcal{V}ar(c)$ and $\mathcal{V}ar(\rho)$, respectively: $\mathcal{V}ar(s_1 \twoheadrightarrow t_1; \cdots ; s_k \twoheadrightarrow t_k) = \mathcal{V}ar(s_1, t_1, \cdots, s_k, t_k)$ and $\mathcal{V}ar(\rho) = \mathcal{V}ar(l, r, c)$. A variable occurring in r or c is called an *extra variables of the rule* ρ if it does not occur in l. The set of extra variables of ρ is denoted by $\mathcal{EV}ar(\rho)$: $\mathcal{EV}ar(\rho) = \mathcal{V}ar(r, c) \setminus \mathcal{V}ar(l)$.

An *oriented conditional term rewriting system* (*CTRS*, for short) over a signature $\mathcal{F}$ is a finite set of oriented conditional rewrite rules over $\mathcal{F}$. In particular, a CTRS is called an *EV-TRS* if all of its rules are unconditional, and called an *extended CTRS* (*eCTRS*, for short) if the condition "$l \notin \mathcal{V}$" of conditional rewrite rules is not imposed. Moreover, a CTRS is called an (*unconditional*) *term rewriting system* (*TRS*, for short) if every rule $l \to r \Leftarrow c$ in it is unconditional and satisfies $\mathcal{V}ar(l) \supseteq \mathcal{V}ar(r)$. Note that an eCTRS is called an *eTRS* if all of its rules are unconditional. For an eCTRS R, the *n-level reduction relation* $\to_{(n),R}$ of R is defined as follows: $\to_{(0),R} = \emptyset$, and $\to_{(i+1),R} = \to_{(i),R} \cup \{(C[l\sigma]_p, C[l\sigma]_p) \mid \rho : l \to r \Leftarrow s_1 \twoheadrightarrow t_1; \cdots ; s_k \twoheadrightarrow t_k \in R,\ s_1\sigma \to^*_{(i),R} t_1\sigma,\ \cdots,\ s_k\sigma \to^*_{(i),R} t_k\sigma\}$ where $i \geq 0$ and $\to^*_{(i),R}$ is the reflexive and transitive closure of $\to_{(i),R}$. The *reduction relation* of R is defined as $\to_R = \bigcup_{n\geq 0} \to_{(n),R}$. To specify the applied rule ρ and the position p, we may write $\to_{\rho,p,R}$ or $\to_{p,R}$ instead of $\to_R$. Moreover, we may write $\to_{>\varepsilon,R}$ instead of $\to_{\rho,p,R}$ or $\to_{p,R}$ if $p > \varepsilon$. The *join* relation $\downarrow_R$ is defined as $\downarrow_R = \{(s,t) \mid \exists u.\ s \to^*_R u \wedge t \to^*_R u\}$. The *parallel reduction* $\rightrightarrows_R$ is defined as $\rightrightarrows_R = \{(C[s_1, \cdots, s_n]_{p_1,\cdots,p_n}, C[t_1, \cdots, t_n]_{p_1,\cdots,p_n}) \mid s_1 \to_R t_1,\ \cdots,\ s_n \to_R t_n\}$. We may write $\rightrightarrows_{>\varepsilon,R}$ instead of $\rightrightarrows_R$ if $p_i > \varepsilon$ for all $1 \leq i \leq p_n$.

An (extended) conditional rewrite rule $\rho : l \to r \Leftarrow c$ is called *left-linear* (*LL*, for short) if l is linear, called *right-linear* (*RL*, for short) if r is linear, called *non-erasing* (*NE*, for short) if $\mathcal{V}ar(l) \subseteq \mathcal{V}ar(r)$, called *non-collapsing* if the right-hand side r is not a variable, and called *non-left-variable* (*non-LV*, for short) if l is not a variable. An eCTRS is called *left-linear* (*right-linear*, *non-erasing*, *non-collapsing*, *non-left-variable*, resp.) if all of its rules are left-linear (right-linear, non-erasing, non-collapsing and non-LV, resp.). Note that a non-LV eCTRS is a CTRS (i.e., it is not an extended one).

An (extended) conditional rewrite rule $\rho : l \to r \Leftarrow s_1 \twoheadrightarrow t_1; \cdots ; s_k \twoheadrightarrow t_k$ is called *deterministic* if $\mathcal{V}ar(s_i) \subseteq \mathcal{V}ar(l, t_1, \cdots, t_{i-1})$ for all $1 \leq i \leq k$. An eCTRS is called *deterministic* (*eDCTRS*, for short) if all of its rules are deterministic. The rule ρ is classified according to the distribution of variables in the rule as follows: *Type 1* if $\mathcal{V}ar(r, s_1, t_1, \cdots, s_k, t_k) \subseteq \mathcal{V}ar(l)$, *Type 2* if $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$, *Type 3* if $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l, s_1, t_1, \cdots, s_k, t_k)$, and *Type 4* otherwise. An e(D)CTRS is called a *1-e(D)CTRS* (*2-e(D)CTRS 3-e(D)CTRS*, and *4-e(D)CTRS*, resp.) if all of its rules are Type 1 (Type 2, Type 3 and Type 4, resp.).

3 Unraveling for DCTRSs

In this section, we first recall an unraveling for DCTRSs proposed by Ohlebusch and its optimized variant. Then, we show some syntactic properties related to the unraveled TRSs. The unravelings and some results are extended to eDCTRSs.

A computable transformation U from eCTRSs into eTRSs is called an *unraveling* if for every eCTRS R, $\downarrow_R \subseteq \downarrow_{U(R)}$ and $U(T \cup R) = T \cup U(R)$ whenever T is an eTRS [11, 12].

Note that a sufficient condition for $\downarrow_R \subseteq \downarrow_{U(R)}$ is $\to_R \subseteq \to^*_{U(R)}$. For an eDCTRS R over a signature $\mathcal{F}$, the unraveling U is called *sound w.r.t. reduction* (*simulation-sound* [17, 19], or simply *sound*) if $\to^*_{U(R)} \subseteq \to^*_R$ on $\mathcal{T}(\mathcal{F},\mathcal{V}) \times \mathcal{T}(\mathcal{F},\mathcal{V})$ (i.e., for any terms s and t in $\mathcal{T}(\mathcal{F},\mathcal{V})$, $s \to^*_{U(R)} t$ implies $s \to^*_R t$).

For a finite set $A = \{a_1, \cdots, a_n\}$, $\overrightarrow{A}$ denotes the unique sequence $a_1, \cdots, a_n$ of elements in A, following some fixed ordering $\prec$ such that $a_1 \prec \cdots \prec a_n$.

▶ **Definition 3.1** (unraveling $\mathbb{U}$ [20])**.** Let R be an eDCTRS over a signature $\mathcal{F}$. For every conditional rule $\rho : l \to r \Leftarrow s_1 \twoheadrightarrow t_1; \cdots; s_k \twoheadrightarrow t_k$ in R, we prepare k fresh function symbols $U^\rho_1, \cdots, U^\rho_k$, called *U symbols*, that do not appear in $\mathcal{F}$. We transform ρ into a set $\mathbb{U}(\rho)$ of $k+1$ unconditional rewrite rules as follows:

$$\mathbb{U}(\rho) = \{\ l \to U^\rho_1(s_1, \overrightarrow{X_1}),\ U^\rho_1(t_1, \overrightarrow{X_1}) \to U^\rho_2(s_2, \overrightarrow{X_2}),\ \cdots,\ U^\rho_k(t_k, \overrightarrow{X_k}) \to r\ \}$$

where $X_i = \mathcal{V}ar(l, t_1, \cdots, t_{i-1})$. Note that $\mathbb{U}(l' \to r') = \{l' \to r'\}$. $\mathbb{U}$ is extended to eDCTRSs (i.e., $\mathbb{U}(R) = \bigcup_{\rho \in R} \mathbb{U}(\rho)$) and $\mathbb{U}(R)$ is an eTRS over the extended signature $\mathcal{F}_{\mathbb{U}(R)}$ where $\mathcal{F}_{\mathbb{U}(R)} = \mathcal{F} \cup \{U^\rho_i \mid \rho : l \to r \Leftarrow s_1 \twoheadrightarrow t_1; \cdots; s_k \twoheadrightarrow t_k \in R,\ 1 \le i \le k\}$.

It is clear that $\to_R \subseteq \to^*_{\mathbb{U}(R)}$ and $\mathbb{U}(T \uplus R) = T \cup \mathbb{U}(R)$ if T is unconditional. Thus, $\mathbb{U}$ is an unraveling for eDCTRSs.

▶ **Definition 3.2** (optimized unraveling $\mathbb{U}_{\mathrm{opt}}$ [14, 4])**.** Let R be an eDCTRS over a signature $\mathcal{F}$. Introducing U symbols $U^\rho_1, \cdots, U^\rho_k$ again, we transform $\rho : l \to r \Leftarrow s_1 \twoheadrightarrow t_1; \cdots; s_k \twoheadrightarrow t_k$ into a set $\mathbb{U}_{\mathrm{opt}}(\rho)$ of $k+1$ unconditional rewrite rules as follows:

$$\mathbb{U}_{\mathrm{opt}}(\rho) = \{\ l \to U^\rho_1(s_1, \overrightarrow{X_1}),\ U^\rho_1(t_1, \overrightarrow{X_1}) \to U^\rho_2(s_2, \overrightarrow{X_2}),\ \cdots,\ U^\rho_k(t_k, \overrightarrow{X_k}) \to r\ \}$$

where $X_i = \mathcal{V}ar(l, t_1, \cdots, t_{i-1}) \cap \mathcal{V}ar(r, t_i, s_{i+1}, t_{i+1}, \cdots, s_k, t_k)$. Note that $\mathbb{U}_{\mathrm{opt}}(l' \to r') = \{l' \to r'\}$. $\mathbb{U}_{\mathrm{opt}}$ is extended to eDCTRSs (i.e., $\mathbb{U}_{\mathrm{opt}}(R) = \bigcup_{\rho \in R} \mathbb{U}_{\mathrm{opt}}(\rho)$) and $\mathbb{U}_{\mathrm{opt}}(R)$ is an eTRS over the extended signature $\mathcal{F}_{\mathbb{U}_{\mathrm{opt}}(R)}$ where $\mathcal{F}_{\mathbb{U}_{\mathrm{opt}}(R)} = \mathcal{F}_{\mathbb{U}(R)}$.

It is clear that $\mathbb{U}_{\mathrm{opt}}$ is also an unraveling for eDCTRSs. Note that X_i in Definition 3.2 is the set of variables which appear in any of $l, t_1, \cdots, t_{i-1}$ and also appear in any of $r, t_i, s_{i+1}, t_{i+1}, \cdots, s_k, t_k$, i.e., every variable in X_i is referred after s_i is considered. On the other hand, X_i in Definition 3.1 is used for carrying all the variables that already appear. This is the only difference between $\mathbb{U}$ and $\mathbb{U}_{\mathrm{opt}}$ and the reason why $\mathbb{U}_{\mathrm{opt}}$ is sometimes called an *optimized* variant of $\mathbb{U}$. Note that all of the following are equivalent: R is in Type 3, $\mathbb{U}(R)$ has no extra variable, and $\mathbb{U}_{\mathrm{opt}}(R)$ has no extra variable.

▶ **Example 3.3.** Consider the following TRS defining addition and multiplication of natural numbers encoded as $\mathsf{0}, \mathsf{s}(\mathsf{0}), \mathsf{s}(\mathsf{s}(\mathsf{0})), \cdots$:

$$R_1 = \left\{ \begin{array}{lll} \mathsf{0} + y \to y & \mathsf{s}(x) + y \to \mathsf{s}(x + y) & \\ \mathsf{0} \times y \to \mathsf{0} & x \times \mathsf{0} \to \mathsf{0} & \mathsf{s}(x) \times \mathsf{s}(y) \to \mathsf{s}((x \times \mathsf{s}(y)) + y) \end{array} \right\}$$

This TRS is inverted to the following 4-DCTRS R_2 [14, 19, 18] where $+^{-1}$ and $\times^{-1}$ are function symbols that define the inverse relation of $+$ and $\times$, respectively (i.e., $+^{-1}(\mathsf{s}^{m+n}(\mathsf{0})) \to^*_{R_2} \mathsf{tp}_2(\mathsf{s}^m(\mathsf{0}), \mathsf{s}^n(\mathsf{0}))$ and $\times^{-1}(\mathsf{s}^{m\times n}(\mathsf{0})) \to^*_{R_2} \mathsf{tp}_2(\mathsf{s}^m(\mathsf{0}), \mathsf{s}^n(\mathsf{0})))$ and tp_2 is a constructor for representing tuples of two terms:

$$R_2 = \left\{ \begin{array}{ll} +^{-1}(y) \to \mathsf{tp}_2(\mathsf{0}, y) & +^{-1}(\mathsf{s}(z)) \to \mathsf{tp}_2(\mathsf{s}(x), y) \Leftarrow +^{-1}(z) \twoheadrightarrow \mathsf{tp}_2(x, y) \\ \times^{-1}(\mathsf{0}) \to \mathsf{tp}_2(\mathsf{0}, y) & \times^{-1}(\mathsf{0}) \to \mathsf{tp}_2(x, \mathsf{0}) \\ \times^{-1}(\mathsf{s}(z)) \to \mathsf{tp}_2(\mathsf{s}(x), \mathsf{s}(y)) \Leftarrow +^{-1}(z) \twoheadrightarrow \mathsf{tp}_2(w, y);\ \times^{-1}(w) \twoheadrightarrow \mathsf{tp}_2(x, \mathsf{s}(y)) & \end{array} \right\}$$

This DCTRS is unraveled by $\mathbb{U}$ and $\mathbb{U}_{\mathrm{opt}}$ as follows:

$$\mathbb{U}(R_2) = \left\{ \begin{array}{rl} +^{-1}(y) & \to \mathsf{tp}_2(\mathsf{0}, y) \\ +^{-1}(\mathsf{s}(z)) & \to \mathsf{U}_1(+^{-1}(z), z) \\ \mathsf{U}_1(\mathsf{tp}_2(x, y), z) & \to \mathsf{tp}_2(\mathsf{s}(x), y) \\ \times^{-1}(\mathsf{0}) & \to \mathsf{tp}_2(\mathsf{0}, y) \\ \times^{-1}(\mathsf{0}) & \to \mathsf{tp}_2(x, \mathsf{0}) \\ \times^{-1}(\mathsf{s}(z)) & \to \mathsf{U}_2(+^{-1}(z), z) \\ \mathsf{U}_2(\mathsf{tp}_2(w, y), z) & \to \mathsf{U}_3(\times^{-1}(w), z, w, y) \\ \mathsf{U}_3(\mathsf{tp}_2(x, \mathsf{s}(y)), z, w, y) & \to \mathsf{tp}_2(\mathsf{s}(x), \mathsf{s}(y)) \end{array} \right\}$$

$$\mathbb{U}_{\mathrm{opt}}(R_2) = \left\{ \begin{array}{rl} +^{-1}(y) & \to \mathsf{tp}_2(\mathsf{0}, y) \\ +^{-1}(\mathsf{s}(z)) & \to \mathsf{U}_1(+^{-1}(z)) \\ \mathsf{U}_1(\mathsf{tp}_2(x, y)) & \to \mathsf{tp}_2(\mathsf{s}(x), y) \\ \times^{-1}(\mathsf{0}) & \to \mathsf{tp}_2(\mathsf{0}, y) \\ \times^{-1}(\mathsf{0}) & \to \mathsf{tp}_2(x, \mathsf{0}) \\ \times^{-1}(\mathsf{s}(z)) & \to \mathsf{U}_2(+^{-1}(z)) \\ \mathsf{U}_2(\mathsf{tp}_2(w, y)) & \to \mathsf{U}_3(\times^{-1}(w), y) \\ \mathsf{U}_3(\mathsf{tp}_2(x, \mathsf{s}(y)), y) & \to \mathsf{tp}_2(\mathsf{s}(x), \mathsf{s}(y)) \end{array} \right\}$$

Unravelings are not sound for every target (e)CTRS. The CTRS shown in the following example is a counterexample against soundness of an unraveling proposed in [11], and also of both $\mathbb{U}$ and $\mathbb{U}_{\mathrm{opt}}$.

▶ **Example 3.4.** Consider the following 3-DCTRS and its unraveled TRS:

$$R_3 = \left\{ \begin{array}{llllll} \mathsf{f}(x) \to x \Leftarrow x \twoheadrightarrow \mathsf{e} & \mathsf{g}(\mathsf{d}, x, x) \to \mathsf{A} & \mathsf{a} \to \mathsf{c} & \mathsf{b} \to \mathsf{c} & \mathsf{c} \to \mathsf{e} & \mathsf{k} \to \mathsf{l} \\ \mathsf{h}(x, x) \to \mathsf{g}(x, x, \mathsf{f}(\mathsf{k})) & \mathsf{d} \to \mathsf{m} & \mathsf{a} \to \mathsf{d} & \mathsf{b} \to \mathsf{d} & \mathsf{c} \to \mathsf{l} & \mathsf{k} \to \mathsf{m} \end{array} \right\}$$

$$\mathbb{U}(R_3) = \mathbb{U}_{\mathrm{opt}}(R_3) = \{\, \mathsf{f}(x) \to \mathsf{U}_4(x, x) \quad \mathsf{U}_4(\mathsf{e}, x) \to x \quad \cdots \,\}$$

We have a reduction sequence of $\mathbb{U}(R_3)$ from $\mathsf{h}(\mathsf{f}(\mathsf{a}), \mathsf{f}(\mathsf{b}))$ to A but not a reduction sequence of R_3. Thus, neither $\mathbb{U}$ nor $\mathbb{U}_{\mathrm{opt}}$ is sound for R_3. We will observe the detail of the reduction sequence in Subsection 4.1.

Soundness of $\mathbb{U}$ can be recovered by restricting the reduction of the unraveled TRSs to the *context-sensitive* one [9] with the *replacement mapping* determined via the application of $\mathbb{U}$: $\mathbb{U}$ is sound for a 3-DCTRS R if the reduction of $\mathbb{U}(R)$ is restricted to context-sensitive rewriting with the replacement mapping μ such that $\mu(U_i^\rho) = \{1\}$ for any U symbol U_i^ρ [23, 22]. This holds for $\mathbb{U}_{\mathrm{opt}}$ by restricting the context-sensitive reduction to one with the *membership constraints* [25] that $x \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ for any variable x appearing in the left-hand sides of rules in $\mathbb{U}_{\mathrm{opt}}(R)$ [19]. Soundness of $\mathbb{U}_{\mathrm{opt}}$ requires a more complicated restriction than $\mathbb{U}$ requires. From this viewpoint, $\mathbb{U}_{\mathrm{opt}}$ does not look an "optimized" variant of $\mathbb{U}$.

To analyze syntactic relationships between eDCTRS and the corresponding unraveled eTRSs, we recall *ultra-properties* of DCTRSs [11, 12], extending them to eDCTRSs.

▶ **Definition 3.5** (ultra-property [11, 12])**.** Let P be a property on (extended) conditional rewrite rules, and U be an unraveling. An (extended) conditional rewrite rule ρ is said to be *ultra-P w.r.t. U* (U-P, for short) if all the rules in $U(\rho)$ satisfy the property P. An eDCTRS R is said to be *ultra-P w.r.t. U* if all the rules in R are ultra-P.

For example, $\mathbb{U}$-*LL*, $\mathbb{U}$-*RL* and $\mathbb{U}$-*NE* denote *ultra-left-linear w.r.t.* $\mathbb{U}$, *ultra-right-linear w.r.t.* $\mathbb{U}$ and *ultra-non-erasing w.r.t.* $\mathbb{U}$, respectively, and $\mathbb{U}_{\mathrm{opt}}$-*LL*, $\mathbb{U}_{\mathrm{opt}}$-*RL* and $\mathbb{U}_{\mathrm{opt}}$-*NE* denote *ultra-left-linear w.r.t.* $\mathbb{U}_{\mathrm{opt}}$, *ultra-right-linear w.r.t.* $\mathbb{U}_{\mathrm{opt}}$ and *ultra-non-erasing w.r.t.* $\mathbb{U}_{\mathrm{opt}}$, respectively. Note that ultra-left-linearity w.r.t. $\mathbb{U}$ is the same as the *semi-linearity* in [12].

▶ **Example 3.6.** The DCTRS R_2 in Example 3.3 is non-LV and non-collapsing w.r.t. both $\mathbb{U}$ and $\mathbb{U}_{\mathrm{opt}}$ but R_2 is not $\mathbb{U}$-LL, $\mathbb{U}$-RL or $\mathbb{U}$-NE, while R_2 is $\mathbb{U}_{\mathrm{opt}}$-RL and $\mathbb{U}_{\mathrm{opt}}$-NE but not $\mathbb{U}_{\mathrm{opt}}$-LL.

The ultra-LL, ultra-RL and ultra-NE properties w.r.t. $\mathbb{U}_{\mathrm{opt}}$ are characterized by syntactic properties of DCTRSs as follows.

▶ **Lemma 3.7** ([14, 18]). *Let $\rho : l \to r \Leftarrow s_1 \twoheadrightarrow t_1; \cdots ; s_k \twoheadrightarrow t_k$ be an extended deterministic conditional rewrite rule. Then, all of the following hold:*

- *ρ is $\mathbb{U}_{\mathrm{opt}}$-LL iff all of $l, t_1, \cdots, t_k$ are linear and $\mathcal{V}ar(t_i) \cap \mathcal{V}ar(l, t_1, \cdots, t_{i-1}) = \emptyset$ for all $1 \leq i \leq k$,*
- *ρ is $\mathbb{U}_{\mathrm{opt}}$-RL iff all of $r, s_1, \cdots, s_k$ are linear and $\mathcal{V}ar(s_i) \cap \mathcal{V}ar(r, t_i, s_{i+1}, t_{i+1}, \cdots, s_k, t_k) = \emptyset$ for all $1 \leq i \leq k$, and*
- *ρ is $\mathbb{U}_{\mathrm{opt}}$-NE iff $\mathcal{V}ar(l) \subseteq \mathcal{V}ar(r, s_1, \cdots, s_k)$ and $\mathcal{V}ar(t_i) \subseteq \mathcal{V}ar(r, s_{i+1}, \cdots, s_k,)$ for all $1 \leq i \leq k$.*

The sufficient and necessary condition for the $\mathbb{U}_{\mathrm{opt}}$-NE property in Lemma 3.7 is equivalent to the one shown in [14, 18] that $\mathcal{V}ar(l) \subseteq \mathcal{V}ar(r, s_1, t_1, \cdots, s_k, t_k)$ and $\mathcal{V}ar(t_i) \subseteq \mathcal{V}ar(r, s_{i+1}, t_{i+1}, \cdots, s_k, t_k)$ for all $1 \leq i \leq k$. Neither of the second nor third claims in Lemma 3.7 holds for $\mathbb{U}$ (cf. Example 3.6), while the first one holds for $\mathbb{U}$. Quite restricted variants of the second and third claims hold for $\mathbb{U}$.

▶ **Lemma 3.8.** *Let $\rho : l \to r \Leftarrow s_1 \twoheadrightarrow t_1; \cdots ; s_k \twoheadrightarrow t_k$ be an extended deterministic conditional rewrite rule. Then, all of the following hold:*

- *ρ is $\mathbb{U}$-LL iff $l, t_1, \cdots, t_k$ are linear and $\mathcal{V}ar(t_i) \cap \mathcal{V}ar(l, t_1, \cdots, t_{i-1}) = \emptyset$ for all $1 \leq i \leq k$, (i.e., ρ is $\mathbb{U}$-LL iff ρ is $\mathbb{U}_{\mathrm{opt}}$-LL),*
- *ρ is $\mathbb{U}$-RL iff r is linear and all of $s_1, \cdots, s_k$ are ground, and*
- *ρ is $\mathbb{U}$-NE iff $\mathcal{V}ar(l, t_1, \cdots, t_k) \subseteq \mathcal{V}ar(r)$.*

By definition of $\mathbb{U}(\rho)$, it is clear that Lemma 3.8 holds. Due to Lemmas 3.7 and 3.8, we have the following relationship between the ultra-RL and ultra-NE properties w.r.t. $\mathbb{U}$ and $\mathbb{U}_{\mathrm{opt}}$.

▶ **Corollary 3.9.** *The $\mathbb{U}$-RL and $\mathbb{U}$-NE properties imply $\mathbb{U}_{\mathrm{opt}}$-RL and $\mathbb{U}_{\mathrm{opt}}$-NE, resp.*

As for the non-collapsing and non-LV properties, we have the following relationships between eDCTRSs and the corresponding unraveled eTRSs.

▶ **Lemma 3.10.** *Let U be either $\mathbb{U}$ or $\mathbb{U}_{\mathrm{opt}}$, and ρ be an (extended) conditional rewrite rule. Then, ρ is non-collapsing (non-LV, resp.) iff $U(\rho)$ is non-collapsing (non-LV, resp.). Thus, an eDCTRS R is non-collapsing (non-LV, resp.) iff $U(R)$ is non-collapsing (non-LV, resp.).*

By definition, it is clear that Lemma 3.10 holds. It follows from Lemma 3.10 that for both $\mathbb{U}$ and $\mathbb{U}_{\mathrm{opt}}$, the non-LV and non-collapsing properties are equivalent to the ultra-non-LV and ultra-non-collapsing properties, respectively.

4 Soundness without Context-Sensitivity

In this section, we first show that the unraveling $\mathbb{U}_{\mathrm{opt}}$ is sound for a $\mathbb{U}_{\mathrm{opt}}$-LL DCTRS if the reduction of the corresponding unraveled EV-TRS is restricted to *EV-basic* ones (see Definition 4.2). Then, we show that $\mathbb{U}_{\mathrm{opt}}$ is sound for DCTRSs that are both $\mathbb{U}_{\mathrm{opt}}$-RL and $\mathbb{U}_{\mathrm{opt}}$-NE. Finally, we show that these claims also hold for the unraveling $\mathbb{U}$. In the rest of this paper, we may write the terminology "*RL-NE*" for "right-linear and non-erasing", and may also write the terminology "*ultra-RL-NE w.r.t. an unraveling U*" (*U-RL-NE*, for short) for "ultra-RL and ultra-NE w.r.t. U".

4.1 Observation of Unsoundness

To begin with, we discuss why $\mathbb{U}_{\mathrm{opt}}$ is not sound for R_3 in Example 3.4. Consider the detail of the derivation $\mathsf{h}(\mathsf{f}(\mathsf{a}), \mathsf{f}(\mathsf{b})) \to^*_{\mathbb{U}_{\mathrm{opt}}(R_3)} \mathsf{A}$:

$$\begin{aligned}\mathsf{h}(\mathsf{f}(\mathsf{a}), \mathsf{f}(\mathsf{b})) &\to^*_{\mathbb{U}_{\mathrm{opt}}(R_3)} \mathsf{h}(\mathsf{U}_4(\mathsf{c}, \mathsf{d}), \mathsf{U}_4(\mathsf{c}, \mathsf{d})) \to^*_{\mathbb{U}_{\mathrm{opt}}(R_3)} \mathsf{g}(\mathsf{U}_4(\mathsf{c}, \mathsf{d}), \mathsf{U}_4(\mathsf{c}, \mathsf{d}), \mathsf{f}(\mathsf{k})) \\ &\to^*_{\mathbb{U}_{\mathrm{opt}}(R_3)} \mathsf{g}(\mathsf{d}, \mathsf{U}_4(\mathsf{l}, \mathsf{m}), \mathsf{U}_4(\mathsf{l}, \mathsf{m})) \to_{\mathbb{U}_{\mathrm{opt}}(R_3)} \mathsf{A}\end{aligned}$$

To succeed in this derivation, the following subderivations are necessary:

- to apply the rule $\mathsf{g}(\mathsf{d}, x, x) \to \mathsf{A}$, the subterm $\mathsf{f}(\mathsf{a})$ in the initial term is reduced to d,
- to apply the rule $\mathsf{h}(x, x) \to \mathsf{g}(x, x, \mathsf{f}(\mathsf{k}))$, both the subterms $\mathsf{f}(\mathsf{a})$ and $\mathsf{f}(\mathsf{b})$ in the initial term are reduced to the same term, and
- to apply the rule $\mathsf{g}(\mathsf{d}, x, x) \to \mathsf{A}$, both the subterm $\mathsf{f}(\mathsf{b})$ in the initial term and the term $\mathsf{f}(\mathsf{k})$ derived from the application of $\mathsf{h}(x, x) \to \mathsf{g}(x, x, \mathsf{f}(\mathsf{k}))$ are reduced to the same term.

In summary, all of the terms $\mathsf{f}(\mathsf{a})$, $\mathsf{f}(\mathsf{b})$ and $\mathsf{f}(\mathsf{k})$ have to be reduced to the same term d. However, this is impossible on the reduction of R_3. Nevertheless, in the above derivation, $\mathsf{h}(x, x) \to \mathsf{g}(x, x, \mathsf{f}(\mathsf{k}))$ is applied after reducing $\mathsf{f}(\mathsf{a})$ and $\mathsf{f}(\mathsf{b})$ to $\mathsf{U}_4(\mathsf{c}, \mathsf{d})$; one of $\mathsf{U}_4(\mathsf{c}, \mathsf{d})$ that comes from $\mathsf{f}(\mathsf{a})$ is reduced to d, and the other $\mathsf{U}_4(\mathsf{c}, \mathsf{d})$ that comes from $\mathsf{f}(\mathsf{b})$ is reduced to $\mathsf{U}_4(\mathsf{l}, \mathsf{m})$ in order to be the same with $\mathsf{f}(\mathsf{k})$; finally, $\mathsf{g}(\mathsf{d}, x, x) \to \mathsf{A}$ is applied. These undesired subderivations must be caused by the non-right-linear rule $\mathsf{h}(x, x) \to \mathsf{g}(x, x, \mathsf{f}(\mathsf{k}))$ and the erasing rule $\mathsf{g}(\mathsf{d}, x, x) \to \mathsf{A}$ in $\mathbb{U}_{\mathrm{opt}}(R_3)$. This is because

- the application of $\mathsf{h}(x, x) \to \mathsf{g}(x, x, \mathsf{f}(\mathsf{k}))$ to $\mathsf{h}(\mathsf{U}_4(\mathsf{c}, \mathsf{d}), \mathsf{U}_4(\mathsf{c}, \mathsf{d}))$ keeps two occurrences of $\mathsf{U}_4(\mathsf{c}, \mathsf{d})$ that are intermediate states of evaluating $\mathsf{f}(\mathsf{a})$ and $\mathsf{f}(\mathsf{b})$, respectively, and each of them has a capability to be reduced to a different term later though they should be the same, and
- $\mathsf{g}(\mathsf{d}, x, x) \to \mathsf{A}$ erases the two occurrences of $\mathsf{U}_4(\mathsf{l}, \mathsf{m})$ as if they come from the same term (in fact, they come from the terms $\mathsf{f}(\mathsf{b})$ and $\mathsf{f}(\mathsf{k})$, respectively, that should be reduced to different terms).

Viewed in this light, it is conjectured that the combination of right-linearity and non-erasingness of the unraveled TRSs is a sufficient condition for soundness of $\mathbb{U}_{\mathrm{opt}}$.

On the other hand, left-linearity of the unraveled TRSs also seems a sufficient condition for soundness of $\mathbb{U}_{\mathrm{opt}}$. A positive witness is that the unravelings for *join* and *normal* CTRSs are sound for left-linear CTRSs [11, 5] and Marchiori's unraveling for 3-DCTRSs is sound for $\mathbb{U}_{\mathrm{opt}}$-LL ones [12]. In addition, left-linearity of the unraveled TRSs seems another solution to avoid the problem mentioned above. Thus, it is conjectured that left-linearity of the unraveled TRSs is a sufficient condition for soundness of $\mathbb{U}_{\mathrm{opt}}$.

In the next two subsections, we will prove these two conjectures above. We first show the case of left-linearity since the other case can be reduced to soundness under the left-linearity case, by transforming a DCTRS into the inverted one. The key features are that the inverted one is $\mathbb{U}_{\mathrm{opt}}$-LL if the DCTRS is $\mathbb{U}_{\mathrm{opt}}$-RL, and that the unraveled TRS of the inverted one is equivalent to the inverted unraveled TRS of the DCTRS if the DCTRS is $\mathbb{U}_{\mathrm{opt}}$-NE. The converse of this approach is impossible since the second key feature needs the $\mathbb{U}_{\mathrm{opt}}$-NE property (i.e., every $\mathbb{U}_{\mathrm{opt}}$-LL DCTRS does not imply the $\mathbb{U}_{\mathrm{opt}}$-NE property of the corresponding inverted DCTRS).

4.2 Soundness on Ultra-Left-Linearity

In this subsection, we show that the optimized unraveling $\mathbb{U}_{\mathrm{opt}}$ is sound for $\mathbb{U}_{\mathrm{opt}}$-LL DCTRSs if the reduction of the unraveled TRSs is restricted to the *EV-basic* one (see Definition 4.2). Roughly speaking, in an EV-basic reduction sequence, any redex introduced via extra variables at the application of rewrite rules is never reduced anywhere. Note that for eTRSs having no extra variables, the EV-basic property is not a restriction at all, since all of their reduction sequences are EV-basic. In practical cases (e.g., *inverse* TRSs [14, 19, 18, 16]), extra variables are instantiated with constructor terms. At the application of rewrite rules, extra variables in the unraveled eTRSs may introduce undesired terms, e.g., terms rooted by U symbols that are not reachable from terms over the original signature. As a consequence,

$\mathbb{U}_{\mathrm{opt}}$ is not always sound w.r.t. non-EV-basic reduction sequences of the unraveled eTRSs (see Example 4.7).

We first prepare a technical lemma to help us to prove the main lemma. Let X be a finite set of variables, σ and θ be substitutions, and $\to$ be a binary relation on terms. Then, we write $X\sigma \to X\theta$ if $x\sigma \to x\theta$ for any $x \in X$.

▶ **Lemma 4.1.** *Let R be an eDCTRS, $\rho : l \to r \Leftarrow s_1 \twoheadrightarrow t_1; \cdots ; s_k \twoheadrightarrow t_k$ be a $\mathbb{U}_{\mathrm{opt}}$-LL conditional rewrite rule in R, $\sigma_1, \cdots, \sigma_{k+1}$ be substitutions, and $X_i = \mathcal{V}ar(l, t_1, \cdots, t_{i-1}) \cap \mathcal{V}ar(r, t_i, s_{i+1}, t_{i+1}, \cdots, s_k, t_k)$ for all $1 \le i \le k$. If $s_i\sigma_i \to_R^* t_i\sigma_{i+1}$ and $X_i\sigma_i \to_R^* X_i\sigma_{i+1}$ for all $1 \le i \le k$, then $l\sigma_1 \to_R^+ r\sigma_{k+1}$.*

Proof. Let σ be the substitution $\sigma_1|_{\mathcal{V}ar(l)} \cup \sigma_2|_{X_1 \setminus \mathcal{V}ar(l)} \cup \cdots \cup \sigma_k|_{X_k \setminus X_{k-1}} \cup \sigma_{k+1}|_{\mathcal{V}ar(t_i, r) \setminus X_k}$. Then, we have that $l\sigma \equiv l\sigma_1$. It follows from $X_i\sigma_i \to_R^* X_i\sigma_{i+1}$ that $X_i\sigma \to_R^* X_i\sigma_{i+1}$ for all $1 \le i \le k$. It follows from the $\mathbb{U}_{\mathrm{opt}}$-LL property and Lemma 3.7 that $\mathcal{V}ar(t_i) \cap \big(\mathcal{D}om(\sigma_1|_{\mathcal{V}ar(l)}) \cup \cdots \cup \mathcal{D}om(\sigma_{i-1}|_{X_{i-1} \setminus X_{i-2}})\big) = \emptyset$ for all $1 \le i \le k$, and hence $t_i\sigma_i \equiv t_i\sigma$ for all $1 \le i \le k$. Thus we have that $s_i\sigma \to_R^* s_i\sigma_i \to_R^* t_i\sigma_{i+1} \equiv t_i\sigma$. Similarly, we have that $r\sigma \to_R^* r\sigma_{k+1}$. Therefore, we have that $l\sigma_1 \equiv l\sigma \to_R r\sigma \to_R^* r\sigma_{k+1}$. ◀

Next we define the notion of *EV-basic* (*EV-safe* [16, 14, 17]) reduction sequences of eTRSs. Roughly speaking, in an EV-basic reduction sequences, any redex introduced via extra variables are not reduced anywhere. This notion can be formalized by relaxing the notion of *basic* reduction sequences [7, 13].

▶ **Definition 4.2** (EV-basic reduction [16])**.** Let R be an eTRS and $\rho_i : l_i \to r_i \in R$ for all $i \ge 1$. Let $t_0 \to_{\rho_1, p_1, R} t_2 \to_{\rho_2, p_2, R} \cdots$ be a reduction sequence of R, and $B_0 \subseteq \mathcal{P}os_{\mathcal{F}}(t_0)$ such that B_0 is prefix closed (i.e., if $p < q$ and $q \in B_0$ then $p \in B_0$). We define the sets $B_1, B_2, \cdots$ of positions from the sequence and B_0 inductively as $B_i = (B_{i-1} \setminus \{q \in B_{i-1} \mid q \ge p_i\}) \cup \{p_iq \mid q \in \mathcal{P}os_{\mathcal{F}}(r_i)\} \cup \{p_ip'q \mid p_ipq \in B_{i-1},\ p \in \mathcal{P}os_{\mathcal{V}}(l_i),\ l_i|_p \equiv r_i|_{p'}\}$ for all $i \ge 1$. Note that $B_1, B_2, \cdots$ are prefix closed. For all $i \ge 0$, positions in B_i are referred as *basic positions of t_i w.r.t. extra variables*. The reduction sequence above is said to be *based on B_0 w.r.t. extra variables* if $p_i \in B_{i-1}$ for all $i \ge 1$. If the sequence is finite with length n, then we denote it by $B_0 : t_0 \xrightarrow[\mathsf{evb}]{}{}_R^* B_n : t_n$ or $B_0 : t_0 \xrightarrow[\mathsf{evb}]{}{}_R^* t_n$. In particular, the reduction sequence is called *basic w.r.t. extra variables* (*EV-basic*, for short) if $B_0 = \mathcal{P}os_{\mathcal{F}}(t_0)$. If the EV-basic sequence is finite with length n, then we denote it by $t_0 \xrightarrow[\mathsf{evb}]{}{}_R^* t_n$.

Note that EV-basicness is different from *basicness* [7, 13] in the sense that all the basic positions are propagated at the application of rewrite rules but none of the positions for extra variables are added to basic positions. A typical instance of EV-basic reduction sequences is a reduction sequence obtained by substituting a normal form for each extra variable when applying rewrite rules.

To specify a set of terms with which extra variables are possibly instantiated at the rule application, we introduce the notion of *EV-instantiation on sets of terms*. Let R be an eTRS and T be a set of terms. A derivation $t_0 \to_{\rho_1, p_1, R} t_1 \to_{\rho_2, p_2, R} \cdots$ is called *EV-instantiated on T* if any extra variable of $\rho_i : l_i \to r_i$ is instantiated by a term in T, i.e., $t_i|_{p_iq} \in T$ for any $q \in \mathcal{P}os_{\mathcal{V}}(r_i)$ such that $r_i|_q \in \mathcal{EV}ar(\rho_i)$. By the same token, the notion of the EV-instantiation property is defined for the parallel reduction of eTRSs. For any of the unraveled eTRSs, its EV-basic reduction sequences have the following property related to EV-instantiation on the set of terms over the original signature.

▶ **Lemma 4.3.** *Let R be a $\mathbb{U}_{\mathrm{opt}}$-LL eDCTRS over a signature $\mathcal{F}$, and s, t be terms in $\mathcal{T}(\mathcal{F}, \mathcal{V})$. If $s \xrightarrow[\mathsf{evb}]{}{}_{\mathbb{U}_{\mathrm{opt}}(R)}^* t$ then there exists a derivation $s \to_{\mathbb{U}_{\mathrm{opt}}(R)}^* t$ that is EV-instantiated on $\mathcal{T}(\mathcal{F}, \mathcal{V})$.*

Proof. It can be proved by induction on the term structure that for a term s, a linear term l with U-symbol-free proper subterms, and substitutions θ, σ, η such that $\theta \in \mathcal{S}ub(\mathcal{F}, \mathcal{V})$ and $\mathsf{root}(x\delta)$ is a U symbol for any $x \in \mathcal{D}om(\eta)$, if $s\theta\eta \equiv l\sigma$, then there exists a substitution σ' such that $s\theta \equiv l\sigma'$ and $l\sigma \equiv l\sigma'\eta$.

To prove this lemma, it suffices to show that for terms $s \in \mathcal{T}(\mathcal{F}_{\mathbb{U}_{\mathrm{opt}}(R)}, \mathcal{V})$ and $t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ and for substitutions θ and η such that $\theta \in \mathcal{S}ub(\mathcal{F}_{\mathbb{U}_{\mathrm{opt}}(R)}, \mathcal{V})$ and $\mathsf{root}(x\eta)$ is a U symbol for any $x \in \mathcal{D}om(\eta)$, if $\mathcal{P}os_{\mathcal{F}}(s\theta) : s\theta\eta \xrightarrow[\mathsf{evb}]{}^{n}_{\mathbb{U}_{\mathrm{opt}}(R)} t$ then there exists a substitution $\sigma \in \mathcal{S}ub(\mathcal{F}, \mathcal{V})$ such that $s\theta\sigma \to^{n}_{\mathbb{U}_{\mathrm{opt}}(R)} t$ and the derivation is EV-instantiated on $\mathcal{T}(\mathcal{F}, \mathcal{V})$. We prove this claim by induction on n.

Suppose that $\mathcal{P}os_{\mathcal{F}}(s\theta) : s\theta\eta \xrightarrow[\mathsf{evb}]{}^{n}_{\mathbb{U}_{\mathrm{opt}}(R)} t$. From the EV-basic property of the derivation and the above claim, we can assume w.l.o.g. that $s\theta$ is of the form $C[s']_p$, $s\theta\eta \equiv C\theta\eta[s'\eta] \equiv C\theta\eta[l\delta\eta]_p \to_{\rho,p,\mathbb{U}_{\mathrm{opt}}(R)} C\theta\eta[r\delta\eta] \to^{n-1}_{\mathbb{U}_{\mathrm{opt}}(R)} t$, $\delta \in \mathcal{S}ub(\mathcal{F}, \mathcal{V})$, $s' \equiv l\delta$, and $p \in \mathcal{P}os_{\mathcal{F}}(s\theta)$, where ρ is $l \to r$, $\mathcal{V}ar(l, r) \cap \mathcal{V}ar(s\theta) = \emptyset$, the set B of EV-basic positions in $C\theta\eta[r\delta\eta]$ is $(\mathcal{P}os_{\mathcal{F}}(s\theta) \setminus \{q \in \mathcal{P}os_{\mathcal{F}}(s\theta) \mid p \le q\}) \cup \{pq \mid q \in \mathcal{P}os_{\mathcal{F}}(r)\} \cup \{pp'q \mid pp''q \in \mathcal{P}os_{\mathcal{F}}(s\theta),\ p'' \in \mathcal{P}os_{\mathcal{V}}(l),\ l|_{p''} \equiv r|_{p'}\}$, and $B : C\theta\eta[r\delta\eta] \xrightarrow[\mathsf{evb}]{}^{n-1}_{\mathbb{U}_{\mathrm{opt}}(R)} t$. Let δ' and δ'' be substitutions such that $\delta' \in \mathcal{S}ub(\mathcal{F}, \mathcal{V})$, $\delta|_{\mathcal{E}\mathcal{V}ar(\rho)} = \delta'\eta$, $\mathcal{D}om(\delta'') \cap (\mathcal{V}ar(l, r) \cup \mathcal{D}om(\eta)) = \emptyset$, and $\mathsf{root}(x\delta'')$ is a U symbol for any $x \in \mathcal{D}om(\delta'')$.

Let $\theta' = \theta|_{\mathcal{V}ar(C[\,])} \cup \delta_{\mathcal{V}ar(l)} \cup \delta'|_{\mathcal{E}\mathcal{V}ar(\rho)}$ and $\eta' = \eta \cup \delta''$. Then, θ' and η' are substitutions such that $C\theta\eta[r\delta] \equiv (C[r])\theta'\eta'$. By the definition of the EV-basic property, we have that $B = \mathcal{P}os_{\mathcal{F}}((C[r])\theta')$. Thus, by the induction hypothesis, we have that there exists a substitution σ in $\mathcal{S}ub(\mathcal{F}, \mathcal{V})$ such that $(C[r])\theta'\sigma \to^{n-1}_{\mathbb{U}_{\mathrm{opt}}(R)} t$ and the derivation is EV-instantiated on $\mathcal{T}(\mathcal{F}, \mathcal{V})$. Now we have that $s\theta\sigma \equiv (C\theta[s'])\sigma \equiv (C\theta[l\delta])\sigma \equiv (C\theta'[l\theta'])\sigma \to_{\mathbb{U}_{\mathrm{opt}}(R)} (C\theta'[r\theta'])\sigma \equiv (C[r])\theta'\sigma' \to^{n-1}_{\mathbb{U}_{\mathrm{opt}}(R)} t$. Since θ and σ are in $\mathcal{S}ub(\mathcal{F}, \mathcal{V})$, any extra variables in r is instantiated by a term in $\mathcal{T}(\mathcal{F}, \mathcal{V})$. Therefore, this derivation is EV-instantiated on $\mathcal{T}(\mathcal{F}, \mathcal{V})$. ◀

The soundness result of this subsection is a consequence of the following key lemma.

▶ **Lemma 4.4.** *Let R be a $\mathbb{U}_{\mathrm{opt}}$-LL eDCTRS over a signature $\mathcal{F}$, s be a term in $\mathcal{T}(\mathcal{F}, \mathcal{V})$, t be a linear term in $\mathcal{T}(\mathcal{F}_{\mathbb{U}_{\mathrm{opt}}(R)}, \mathcal{V})$, and σ be a substitution in $\mathcal{S}ub(\mathcal{F}_{\mathbb{U}_{\mathrm{opt}}(R)}, \mathcal{V})$. Suppose that R is non-LV or non-collapsing. If $s \rightrightarrows^{n}_{\mathbb{U}_{\mathrm{opt}}(R)} t\sigma$ for some $n \ge 0$ and the derivation is EV-instantiated on $\mathcal{T}(\mathcal{F}, \mathcal{V})$, then there exists a substitution θ in $\mathcal{S}ub(\mathcal{F}, \mathcal{V})$ such that $s \to^{*}_{R} t\theta \rightrightarrows^{m}_{\mathbb{U}_{\mathrm{opt}}(R)} t\sigma$ and the derivation $t\theta \rightrightarrows^{m}_{\mathbb{U}_{\mathrm{opt}}(R)} t\sigma$ is EV-instantiated on $\mathcal{T}(\mathcal{F}, \mathcal{V})$ for some $m \le n$ such that if $t\sigma \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ then $m = 0$.*

Proof. We prove this lemma by induction on the lexicographic product (n, s) of n and the structure of s. Suppose that $s \rightrightarrows^{n}_{\mathbb{U}_{\mathrm{opt}}(R)} t\sigma$. Since the case that s is a variable is trivial, we only consider the remaining case that s is rooted by a function symbol.

We first consider the case that $s \rightrightarrows^{n}_{\mathbb{U}_{\mathrm{opt}}(R)} t\sigma$ does not contain any reduction step at the root position. Let s be of the form $f(s_1, \cdots, s_k)$. Then, we have that $s \equiv f(s_1, \cdots, s_k) \rightrightarrows^{n}_{\mathbb{U}_{\mathrm{opt}}(R)} f(t_1, \cdots, t_k)\sigma \equiv t\sigma$ and thus $s_i \rightrightarrows^{n_i}_{\mathbb{U}_{\mathrm{opt}}(R)} t_i\sigma$, where $n_1 + \cdots + n_k = n$. By the induction hypothesis, there exists a substitution $\theta_i \in \mathcal{S}ub(\mathcal{F}, \mathcal{V})$ such that $s_i \to^{*}_{R} t_i\theta_i \rightrightarrows^{m_i}_{\mathbb{U}_{\mathrm{opt}}(R)} t_i\sigma$ and the derivation $t_i\theta_i \rightrightarrows^{m_i}_{\mathbb{U}_{\mathrm{opt}}(R)} t_i\sigma$ is EV-instantiated on $\mathcal{T}(\mathcal{F}, \mathcal{V})$ for some $m_i \le n_i$ such that $m_i = 0$ if $t_i\sigma \in \mathcal{T}(\mathcal{F}, \mathcal{V})$. Let $\theta = \theta_1|_{\mathcal{V}ar(t_1)} \cup \cdots \cup \theta_k|_{\mathcal{V}ar(t_k)}$. Then, it follows from the linearity of t that θ is a substitution in $\mathcal{S}ub(\mathcal{F}, \mathcal{V})$. We have that $s \equiv f(s_1, \cdots, s_k) \to^{*}_{R} f(t_1, \cdots, t_k)\theta \equiv t\theta \rightrightarrows^{m}_{\mathbb{U}_{\mathrm{opt}}(R)} t\sigma$ and the derivation $t\theta \rightrightarrows^{m}_{\mathbb{U}_{\mathrm{opt}}(R)} t\sigma$ is EV-instantiated on $\mathcal{T}(\mathcal{F}, \mathcal{V})$. where $m = m_1 + \cdots + m_k \le n$ such that if $t\sigma \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ then $m = 0$.

Next we consider the remaining case. To simplify the proof, we assume w.l.o.g. that any

rule has two conditions of the form $s_1 \twoheadrightarrow t_1; s_2 \twoheadrightarrow t_2$. Then, we can assume that

$$\begin{aligned} s \Rightarrow^{n_0}_{\mathbb{U}_{\rm opt}(R)} l\sigma_1 \to_{\varepsilon,\mathbb{U}_{\rm opt}(R)} u_1\sigma_1 \Rightarrow^{n_1}_{>\varepsilon,\mathbb{U}_{\rm opt}(R)} u'_1\sigma_2 \\ \to_{\varepsilon,\mathbb{U}_{\rm opt}(R)} u_2\sigma_2 \Rightarrow^{n_2}_{>\varepsilon,\mathbb{U}_{\rm opt}(R)} u'_2\sigma_3 \to_{\varepsilon,\mathbb{U}_{\rm opt}(R)} r\sigma_3 \Rightarrow^{n_3}_{\mathbb{U}_{\rm opt}(R)} t_3\sigma_3, \end{aligned}$$

where $s \Rightarrow^{n_0}_{>\varepsilon,\mathbb{U}_{\rm opt}(R)} l\sigma_1$ if R is non-LV, and $r\sigma_3 \Rightarrow^{n_3}_{>\varepsilon,\mathbb{U}_{\rm opt}(R)} r\sigma$ otherwise (i.e., if R is non-collapsing), $\rho : l \to r \Leftarrow s_1 \twoheadrightarrow t_1; s_2 \twoheadrightarrow t_2 \in R$, $u_i \equiv U^\rho_i(s_i, \overrightarrow{X_i})$, $u'_i \equiv U^\rho_i(t_i, \overrightarrow{X_i})$, $X_1 = \mathcal{V}ar(l) \cap \mathcal{V}ar(r, t_1, s_2, t_2)$, $X_2 = \mathcal{V}ar(l, t_1) \cap \mathcal{V}ar(r)$, and $t\sigma$ is a term between $u_1\sigma$ to $t_3\sigma_3$. We only consider the case that $t\sigma$ is $t_3\sigma_3$ since this case is the most complicated. For this reason, we assume that $t_3\sigma_3 \equiv t\sigma$ and $n_0 + n_1 + n_2 + n_3 + 3 = n$.

By the induction hypothesis, there exists a substitution $\theta_1 \in \mathcal{S}ub(\mathcal{F}, \mathcal{V})$ such that $s \to^*_R l\theta \Rightarrow^{m_0}_{\mathbb{U}_{\rm opt}(R)} l\sigma_1$ and the derivation $l\theta \Rightarrow^{m_0}_{\mathbb{U}_{\rm opt}(R)} l\sigma_1$ is EV-instantiated on $\mathcal{T}(\mathcal{F}, \mathcal{V})$. Let $\theta'_1 = \theta|_{\mathcal{V}ar(l)} \cup \sigma_1|_{\mathcal{E}\mathcal{V}ar(l \to u_1)}$. Then, θ'_1 is a substitution in $\mathcal{T}(\mathcal{F}, \mathcal{V})$. Moreover, it follows from the standard property of the parallel reduction that $u_1\theta'_1 \Rightarrow^{m_1}_{>\varepsilon,\mathbb{U}_{\rm opt}(R)} u_1\sigma_1 \Rightarrow^{n_2}_{>\varepsilon,\mathbb{U}_{\rm opt}(R)} u'_1\sigma_2$. Thus, we have that $s_1\theta'_1 \Rightarrow^{m'_1}_{\mathbb{U}_{\rm opt}(R)} t_1\sigma_2$ and $X_1\theta'_1 \Rightarrow^{m''_1}_{\mathbb{U}_{\rm opt}(R)} X_1\sigma_2$ where m''_1 is the summation of reduction steps and $m'_1 + m''_1 = m_1 + n_2$. Since the $\mathbb{U}_{\rm opt}$-LL property provides $\mathcal{V}ar(t_1) \cap X_1 = \emptyset$, by the induction hypothesis, there exists a substitution $\theta_2 \in \mathcal{S}ub(\mathcal{F}, \mathcal{V})$ such that $s_1\theta'_1 \to^*_R t_1\theta_2 \Rightarrow^{m_2}_{\mathbb{U}_{\rm opt}(R)} t_1\sigma_2$ and $X_1\theta'_1 \to^*_R X_1\theta_2 \Rightarrow^{j_2}_{\mathbb{U}_{\rm opt}(R)} X_1\sigma_2$ where j_2 is the summation of reduction steps and $m_2 + j_2 \leq m_1 + n_2$.

In the same way, we obtain substitutions θ_3 and θ in $\mathcal{S}ub(\mathcal{F}, \mathcal{V})$ such that $s_2\theta_2 \to^*_R t_2\theta_3$, $X_2 \to^*_R X_2\theta_3$, $r\theta_3 \to^*_R t\theta \Rightarrow^m_{\mathbb{U}_{\rm opt}(R)} t\sigma$, where $m \leq n$ such that if $t\sigma \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ then $m = 0$. It follows from Lemma 4.1 that $l\theta_1 \to^*_R r\theta_3$. Therefore, we have that $s \to^*_R r\theta_3 \to^*_R t\theta \Rightarrow^m_{\mathbb{U}_{\rm opt}(R)} t\sigma$ where $m \leq n$ such that if $t\sigma \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ then $m = 0$. ◀

As a consequence of Lemma 4.4, we show the main theorem of this subsection.

▶ **Theorem 4.5.** *$\mathbb{U}_{\rm opt}$ is sound for a $\mathbb{U}_{\rm opt}$-LL eDCTRSs R over a signature $\mathcal{F}$ if R is non-LV or non-collapsing and if the reduction of $\mathbb{U}_{\rm opt}(R)$ is restricted to the EV-basic one (i.e., for any terms s and t in $\mathcal{T}(\mathcal{F}, \mathcal{V})$, if $s \xrightarrow[\mathsf{evb}]{}{}^*_{\mathbb{U}_{\rm opt}(R)} t$ then $s \to^*_R t$).*

Proof. Suppose that $s \xrightarrow[\mathsf{evb}]{}{}^*_{\mathbb{U}_{\rm opt}(R)} t$ and $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$. Then, it follows from Lemma 4.3 that there is a derivation $s \to^*_{\mathbb{U}_{\rm opt}(R)} t$ that is EV-instantiated on $\mathcal{T}(\mathcal{F}, \mathcal{V})$. Since a single step of $\to_{\mathbb{U}_{\rm opt}(R)}$ can be considered as a single step of the parallel reduction, we have the derivation $s \Rightarrow^*_{\mathbb{U}_{\rm opt}(R)} t$ that is EV-instantiated on $\mathcal{T}(\mathcal{F}, \mathcal{V})$. Let x be a variable and σ be a substitution such that $x\sigma \equiv t$. Then, it follows from Lemma 4.4 that $s \to^*_R x\sigma \equiv t$. ◀

It is clear that for a 3-eDCTRS R, any reduction sequence of R is EV-basic. Therefore, $\mathbb{U}_{\rm opt}$ is sound for $\mathbb{U}_{\rm opt}$-LL 3-eDCTRSs.

▶ **Corollary 4.6.** *$\mathbb{U}_{\rm opt}$ is sound for $\mathbb{U}_{\rm opt}$-LL 3-eDCTRSs that are non-LV or non-collapsing.*

Due to the technical proof of Lemma 4.4, we assumed that eDCTRS is non-LV or non-collapsing. It is not known yet that this assumption can be relaxed (or removed). However, this assumption is not so restrictive since every DCTRS (not an extended one) is non-LV. Corollary 4.6 is not a direct consequence of the result in [12] on soundness for $\mathbb{U}_{\rm opt}$-LL 3-DCTRSs since U symbols introduced by $\mathbb{U}_{\rm opt}$ have less arguments than those introduced by Marchiori's unraveling for DCTRSs.

Finally, we show a counterexample against Theorem 4.5 without the EV-basic property.

▶ **Example 4.7.** Consider the following DCTRS and its unraveled EV-TRS:

$$R_4 = \{\ \mathsf{e} \to \mathsf{f}(x) \Leftarrow \mathsf{l} \twoheadrightarrow \mathsf{d},\quad \mathsf{A} \to \mathsf{h}(x,x)\ \}$$
$$\mathbb{U}_{\mathrm{opt}}(R_4) = \mathbb{U}(R_4) = \{\ \mathsf{e} \to \mathsf{U}_5(\mathsf{l}),\quad \mathsf{U}_5(\mathsf{d}) \to \mathsf{f}(x),\quad \mathsf{A} \to \mathsf{h}(x,x)\ \}$$

We have the derivation $\mathsf{A} \to_{\mathbb{U}_{\mathrm{opt}}(R_4)} \mathsf{h}(\mathsf{U}_5(\mathsf{d}),\mathsf{U}_5(\mathsf{d})) \to^*_{\mathbb{U}_{\mathrm{opt}}(R_4)} \mathsf{h}(\mathsf{f}(\mathsf{a}),\mathsf{f}(\mathsf{b}))$ that is not EV-basic: the term $\mathsf{U}_5(\mathsf{d})$ introduced by instantiating the extra variable x in the applied rule $\mathsf{A} \to \mathsf{h}(x,x)$ is reduced. However, A cannot be reduced by R_4 to $\mathsf{h}(\mathsf{f}(\mathsf{a}),\mathsf{f}(\mathsf{b}))$. Therefore, $\mathbb{U}_{\mathrm{opt}}$ is not sound for R_4. Note that $\mathbb{U}$ is not sound either.

4.3 Soundness on Ultra-Right-Linear-Non-Erasing Property

In this subsection, we show that $\mathbb{U}_{\mathrm{opt}}$ is sound for DCTRSs that are ultra-RL-NE w.r.t. $\mathbb{U}_{\mathrm{opt}}$. To prove it, we reduce the soundness to that of $\mathbb{U}_{\mathrm{opt}}$ for ultra-LL DCTRSs.

We first define the operation to transform an eDCTRS into an eDCTRS that defines the inverse relation of the former eDCTRS. Note that the "inverse" has a different meaning from the sense of program inversion.

▶ **Definition 4.8.** Let $\rho : l \to r \Leftarrow s_1 \twoheadrightarrow t_1; \cdots ; s_k \twoheadrightarrow t_k$ be an (extended) conditional rewrite rule. We define the operation $(\ )^{-1}$ as $(l \to r \Leftarrow s_1 \twoheadrightarrow t_1; \cdots ; s_k \twoheadrightarrow t_k)^{-1} = r \to l \Leftarrow t_k \twoheadrightarrow s_k; \cdots ; t_1 \twoheadrightarrow s_1$. This operation is extended to eDCTRSs as $(R)^{-1} = \{(\rho)^{-1} \mid \rho \in R\}$. Moreover, for a binary relation $\to$, we denote the inverse relation of $\to$ by $(\to)^{-1}$: $(\to)^{-1} = \{(t,s) \mid s \to t\}$.

For an eCTRS R, the inverse relation of $\to_R$ is equivalent to the reduction of $(R)^{-1}$.

▶ **Theorem 4.9.** *Let R be an eCTRS. Then, $(\to_R)^{-1} = \to_{(R)^{-1}}$.*

Proof. It suffices to show that $(\to_{(n),R})^{-1} = \to_{(n),(R)^{-1}}$ for all $n \geq 0$. This can be proved by induction on n. ◀

Regarding the operation $(\ )^{-1}$ and the $\mathbb{U}_{\mathrm{opt}}$-NE property, there are dual relationships between $\mathbb{U}_{\mathrm{opt}}$-LL and $\mathbb{U}_{\mathrm{opt}}$-RL and between the non-LV and non-collapsing properties.

▶ **Lemma 4.10.** *Let $\rho : l \to r \Leftarrow s_1 \twoheadrightarrow t_1; \cdots ; s_k \twoheadrightarrow t_k$ be an extended deterministic rewrite rule. Then all of the following hold:*

1. *$\mathcal{V}ar(t_i) \subseteq \mathcal{V}ar(r, s_{i+1}, \cdots, s_k)$ for all $1 \leq i \leq k$ iff $(\rho)^{-1}$ is deterministic,*
2. *$\mathcal{V}ar(l) \subseteq \mathcal{V}ar(r, s_1, \cdots, s_k)$ iff $(\rho)^{-1}$ is in Type 3,*
3. *if $\mathcal{V}ar(t_i) \subseteq \mathcal{V}ar(r, s_{i+1}, \cdots, s_k)$ for all $1 \leq i \leq k$, then*
   a. *$\mathbb{U}_{\mathrm{opt}}((\rho)^{-1}) = (\mathbb{U}_{\mathrm{opt}}(\rho))^{-1}$ up to the renaming of U symbols, and*
   b. *ρ is $\mathbb{U}_{\mathrm{opt}}$-LL ($\mathbb{U}_{\mathrm{opt}}$-RL, resp.) iff $(\rho)^{-1}$ is $\mathbb{U}_{\mathrm{opt}}$-RL ($\mathbb{U}_{\mathrm{opt}}$-LL, resp.),*
4. *ρ is non-LV (non-collapsing, resp.) iff $(\rho)^{-1}$ is non-collapsing (non-LV, resp.).*

Proof. The claims 1, 2 and 4 are trivial. Consider $\mathbb{U}_{\mathrm{opt}}(\rho)$ in Definition 3.2. We can assume w.l.o.g. that $\mathbb{U}_{\mathrm{opt}}((\rho)^{-1}) = \{r \to U^\rho_k(t_k, \overrightarrow{Y_k}), \cdots, U^\rho_2(s_2, \overrightarrow{Y_2}) \to U^\rho_1(t_1, \overrightarrow{Y_1}), U^\rho_1(s_1, \overrightarrow{Y_1}) \to l\}$ where $Y_i = \mathcal{V}ar(r, s_k, \cdots, s_{i+1}) \cap \mathcal{V}ar(l, s_i, t_{i-1}, s_{i-1}, \cdots, t_1, s_1)$. Since ρ is deterministic, we have that $\mathcal{V}ar(s_i) \subseteq \mathcal{V}ar(l, t_1, \cdots, t_{i-1})$ for all $1 \leq i \leq k$.

To prove the claim 3-a, it suffices to show that $X_i = Y_i$ for all $1 \leq i \leq k$. It follows from $\mathcal{V}ar(s_i) \subseteq \mathcal{V}ar(l, t_1, \cdots, t_{i-1})$ that $\mathcal{V}ar(l, s_1, t_1, \cdots, s_{i-1}, t_{i-1}, s_i) = \mathcal{V}ar(l, t_1, \cdots, t_{i-1})$, and hence $Y_i = \mathcal{V}ar(r, s_{i+1}, \cdots, s_k) \cap \mathcal{V}ar(l, t_1, \cdots, t_{i-1})$. Moreover, it follows from $\mathcal{V}ar(t_i) \subseteq \mathcal{V}ar(r, s_{i+1}, \cdots, s_k)$ that $\mathcal{V}ar(r, t_i, s_{i+1}, t_{i+1}, \cdots, s_k, t_k) = \mathcal{V}ar(r, s_{i+1}, \cdots, s_k)$, and hence $X_i = \mathcal{V}ar(l, t_1, \cdots, t_{i-1}) \cap \mathcal{V}ar(r, s_{i+1}, \cdots, s_k)$. Therefore, $X_i = Y_i$ for all $1 \leq i \leq k$.

Finally, we prove the claim 3-b. Suppose that ρ is $\mathbb{U}_{\mathrm{opt}}$-LL. Then, it follows from Lemma 3.7 that l, $U_1^\rho(t_1, \overrightarrow{X_1}), \cdots, U_k^\rho(t_k, \overrightarrow{X_k})$ are linear. Thus, $\mathbb{U}_{\mathrm{opt}}((\rho)^{-1})$ is right-linear and hence $(\rho)^{-1}$ is $\mathbb{U}_{\mathrm{opt}}$-RL. Suppose that ρ is $\mathbb{U}_{\mathrm{opt}}$-RL. Then, it follows from Lemma 3.7 that $r, U_1^\rho(s_1, \overrightarrow{X_1}), \cdots, U_k^\rho(s_k, \overrightarrow{X_k})$ are linear. Thus, $\mathbb{U}_{\mathrm{opt}}((\rho)^{-1})$ is left-linear and hence $(\rho)^{-1}$ is $\mathbb{U}_{\mathrm{opt}}$-LL. The *if* part is similar to the *only-if* part above. ◀

Note that neither the claims 3-a nor 3-b in Lemma 4.10 holds for $\mathbb{U}$.

▶ **Corollary 4.11.** *Let R be an eDCTRS. Then all of the following hold:*

- *R is $\mathbb{U}_{\mathrm{opt}}$-NE iff $(R)^{-1}$ is a 3-eDCTRS,*
- *if R is $\mathbb{U}_{\mathrm{opt}}$-NE, then*
  - *$\mathbb{U}_{\mathrm{opt}}((R)^{-1}) = (\mathbb{U}_{\mathrm{opt}}(R))^{-1}$ up to the renaming of U symbols, and*
  - *R is $\mathbb{U}_{\mathrm{opt}}$-LL ($\mathbb{U}_{\mathrm{opt}}$-RL, resp.) iff $(R)^{-1}$ is $\mathbb{U}_{\mathrm{opt}}$-RL ($\mathbb{U}_{\mathrm{opt}}$-LL, resp.), and*
- *R is non-LV (non-collapsing, resp.) iff $(R)^{-1}$ is non-collapsing (non-LV, resp.).*

Finally, we show soundness of $\mathbb{U}_{\mathrm{opt}}$ for a $\mathbb{U}_{\mathrm{opt}}$-RL-NE eDCTRS R by reducing it to soundness for the $\mathbb{U}_{\mathrm{opt}}$-LL eDCTRS $(R)^{-1}$.

▶ **Theorem 4.12.** *$\mathbb{U}_{\mathrm{opt}}$ is sound for $\mathbb{U}_{\mathrm{opt}}$-RL-NE eDCTRSs that are non-LV or non-collapsing.*

Proof. Let R be a $\mathbb{U}_{\mathrm{opt}}$-RL-NE eDCTRS over a signature $\mathcal{F}$. Then, it follows from Corollary 4.11 that $(R)^{-1}$ is a $\mathbb{U}_{\mathrm{opt}}$-LL 3-eDCTRS that is non-collapsing or non-LV. Thus, it follows from Corollary 4.6 that $\mathbb{U}_{\mathrm{opt}}$ is sound for $(R)^{-1}$, i.e., $\to^*_{\mathbb{U}_{\mathrm{opt}}((R)^{-1})} \subseteq \to^*_{(R)^{-1}}$ holds over $\mathcal{T}(\mathcal{F},\mathcal{V}) \times \mathcal{T}(\mathcal{F},\mathcal{V})$. It follows from Corollary 4.11 that $\mathbb{U}_{\mathrm{opt}}((R)^{-1}) = (\mathbb{U}_{\mathrm{opt}}(R))^{-1}$, and hence $\to^*_{\mathbb{U}_{\mathrm{opt}}((R)^{-1})} = \to^*_{(\mathbb{U}_{\mathrm{opt}}(R))^{-1}}$. By Theorem 4.9, we have that $\to^*_{(\mathbb{U}_{\mathrm{opt}}(R))^{-1}} = (\to^*_{\mathbb{U}_{\mathrm{opt}}(R)})^{-1}$ and $\to^*_{(R)^{-1}} = (\to^*_R)^{-1}$. Thus, we have that $(\to^*_{\mathbb{U}_{\mathrm{opt}}(R)})^{-1} \subseteq (\to^*_R)^{-1}$ (i.e., $\to^*_{\mathbb{U}_{\mathrm{opt}}(R)} \subseteq \to^*_R$) over $\mathcal{T}(\mathcal{F},\mathcal{V}) \times \mathcal{T}(\mathcal{F},\mathcal{V})$. ◀

▶ **Example 4.13.** Consider R_2 in Example 3.3 again. The eDCTRS R_2 is non-LV, $\mathbb{U}_{\mathrm{opt}}$-RL-NE but neither $\mathbb{U}$-RL nor $\mathbb{U}$-LL. Thanks to Theorem 4.12, $\mathbb{U}_{\mathrm{opt}}$ is sound for R_2 and thus $\mathbb{U}_{\mathrm{opt}}(R_2)$ can be used for simulating the reduction of R_2.

Ultra-right-linearity is not a soundness condition for $\mathbb{U}_{\mathrm{opt}}$.

▶ **Example 4.14.** Consider the 3-DCTRS $(R_4)^{-1}$ and the unraveled TRS $(\mathbb{U}_{\mathrm{opt}}(R_4))^{-1}$ obtained from Example 4.7. $(R_4)^{-1}$ is $\mathbb{U}_{\mathrm{opt}}$-RL but not $\mathbb{U}_{\mathrm{opt}}$-NE. We have the derivation $\mathsf{h}(\mathsf{f}(\mathsf{a}),\mathsf{f}(\mathsf{b})) \to^*_{\mathbb{U}_{\mathrm{opt}}((R_4)^{-1})} \mathsf{A}$. However, $\mathsf{h}(\mathsf{f}(\mathsf{a}),\mathsf{f}(\mathsf{b}))$ cannot be reduced by $(R_4)^{-1}$ to A. Therefore, $\mathbb{U}_{\mathrm{opt}}$ is not sound for $(R_4)^{-1}$. Note that $\mathbb{U}$ is sound for $(R_4)^{-1}$.

It is possible to prove Theorem 4.12 directly [17], by using the feature that every reduction sequence of right-linear TRSs can be transformed to a *basic* one [13]. However, Theorem 4.5 cannot be proved by using Theorem 4.12. This is because $\mathbb{U}_{\mathrm{opt}}((R)^{-1}) = (\mathbb{U}_{\mathrm{opt}}(R))^{-1}$ does not hold for every $\mathbb{U}_{\mathrm{opt}}$-LL DCTRSs (see $\mathbb{U}_{\mathrm{opt}}(R_2)$ in Example 3.3).

4.4 Soundness of Ohlebusch's Unraveling

In this subsection, we show that soundness of $\mathbb{U}_{\mathrm{opt}}$ implies that of $\mathbb{U}$.

We first introduce the notion of *argument filterings* [1, 8]. An *argument filtering* over a signature $\mathcal{F}$ is a mapping π from $\mathcal{F}$ to sets of natural numbers such that $\pi(f) \subseteq \{1, \cdots, \mathsf{ar}(f)\}$ for any $f \in \mathcal{F}$. Note that this paper does not use *collapsing definitions* $\pi(f) \in \{1, \cdots, \mathsf{ar}(f)\}$. When $\pi(f)$ is not defined explicitly, we assume that $\pi(f) = \{1, \cdots, \mathsf{ar}(f)\}$. Argument filterings are extended to terms as follows: $\pi(x) = x$ for $x \in \mathcal{V}$, and $\pi(f(t_1, \cdots, t_n)) = f(\pi(t_{i_1}), \cdots, \pi(t_{i_m}))$ for $f \in \mathcal{F}$ where $\pi(f) = \{i_1, \cdots, i_m\}$ and $1 \le i_1 < i_2 < \cdots < i_m \le n$.

They are also extended to eTRSs as follows: $\pi(R) = \{\pi(l) \to \pi(r) \mid l \to r \in R\}$. Note that $\pi(R)$ is an eTRS. Argument filterings have the following properties.

▶ **Lemma 4.15.** *Let π be an argument filtering. Let t be a term and σ, σ_π be substitutions such that $\sigma_\pi = \{x \mapsto \pi(x\sigma) \mid x \in \mathcal{D}om(\sigma)\}$. Then, $\pi(t\sigma) \equiv (\pi(t))\sigma_\pi$. Let R be an eTRS and s, t be terms. If $s \to_R^* t$ then $\pi(s) \to_{\pi(R)}^* \pi(t)$.*

The unraveling $\mathbb{U}_{\mathrm{opt}}$ is an optimized variant of $\mathbb{U}$ in the sense that variables carried by U symbols are optimized. Thus, $\mathbb{U}_{\mathrm{opt}}(R)$ can simulate any reduction sequence of $\mathbb{U}(R)$.

▶ **Lemma 4.16.** *Let R be an eDCTRS over a signature $\mathcal{F}$, and s, t be terms in $\mathcal{T}(\mathcal{F}, \mathcal{V})$. If $s \to_{\mathbb{U}(R)}^* t$, then $s \to_{\mathbb{U}_{\mathrm{opt}}(R)}^* t$.*

Proof. We assume w.l.o.g. that for every rule $\rho : l \to r \Leftarrow s_1 \twoheadrightarrow t_1; \cdots; s_k \twoheadrightarrow t_k$ in R, the same U symbols $U_1^\rho, \cdots, U_k^\rho$ are introduced for $\mathbb{U}(R)$ and $\mathbb{U}_{\mathrm{opt}}(R)$. Let π be an argument filtering such that for every $\rho : l \to r \Leftarrow s_1 \twoheadrightarrow t_1; \cdots; s_k \twoheadrightarrow t_k$ in R, $\pi(U_i^\rho) = \{1, i_1, \cdots, i_m\}$ where $X_i = \mathcal{V}ar(l, t_1, \cdots, t_{i-1})$, $\overrightarrow{X_i}$ is a sequence $x_1, \cdots, x_n$, $Y_i = X_i \cap \mathcal{V}ar(r, t_i, s_{i+1}, t_{i+1}, \cdots, s_k, t_k)$, $\overrightarrow{Y_i}$ is a sequence $y_1, \cdots, y_m$, and $x_{i_j} \equiv y_j$ for all $1 \le j \le m$. Then, it is clear that $\pi(\mathbb{U}(R)) = \mathbb{U}_{\mathrm{opt}}(R)$. Since s, t are in $\mathcal{T}(\mathcal{F}, \mathcal{V})$, we have that $\pi(s) \equiv s$ and $\pi(t) \equiv t$. Thus, it follows from Lemma 4.15 that $s \equiv \pi(s) \to_{\mathbb{U}_{\mathrm{opt}}(R)}^* \pi(t) \equiv t$. ◀

Moreover, $\mathbb{U}_{\mathrm{opt}}(R)$ can simulate every EV-basic reduction sequence of $\mathbb{U}(R)$.

▶ **Lemma 4.17.** *Let R be an eTRS, s, t be terms, and π be an argument filtering such that $\mathcal{EV}ar(\pi(l) \to \pi(r)) \subseteq \mathcal{EV}ar(\rho)$ for every $\rho : l \to r \in R$. If $s \xrightarrow[\mathsf{evb}]{}_R^* t$ then $\pi(s) \xrightarrow[\mathsf{evb}]{}_{\pi(R)}^* \pi(t)$.*

Proof. We first define modifications for a position p of a term u and a set P of positions of u by applying an argument filtering π: $\pi_u(p) = p$ if $p = \varepsilon$; $\pi_u(p) = jp''$ if $u \equiv f(u_1, \cdots, u_n)$, $\pi(f) = \{i_1, \cdots, i_m\}$, $i_1 < \cdots < i_m$, $p = i_j p'$, and $p'' = \pi_{u_i}(p')$; $\pi_u(P) = P$ if $u \in \mathcal{V}$; $\pi_u(P) = \{\varepsilon \mid \varepsilon \in P\} \cup \left\{jp' \mid p' \in \pi_{t_{i_j}}(\{p'' \mid i_j p'' \in P\})\right\}$ if $u \equiv f(u_1, \cdots, u_n)$, $\pi(f) = \{i_1, \cdots, i_m\}$, $i_1 < \cdots < i_m$. We prove that if $B : s \xrightarrow[\mathsf{evb}]{}_R^n B' : t$ and $\pi_s(B) \subseteq B_1 \subseteq \mathcal{P}os(\pi(s))$ then $B_1 : \pi(s) \xrightarrow[\mathsf{evb}]{}_{\pi(R)}^* B_1' : \pi(t)$ and $\pi_t(B') \subseteq B_1'$. To prove this claim by induction on n, it suffices to show that if $B : s \xrightarrow[\mathsf{evb}]{}_{p,R} B' : t$ then $\pi_s(p)$ is defined and $\pi_s(B) : \pi(s) \xrightarrow[\mathsf{evb}]{}_{\pi_s(p),\pi(R)} \pi_t(B') : \pi(t)$. This follows from the assumption and the definitions of $\xrightarrow[\mathsf{evb}]{}$ and $\pi_t(\ )$. ◀

▶ **Lemma 4.18.** *Let R be an eDCTRS over a signature $\mathcal{F}$, and s, t be terms in $\mathcal{T}(\mathcal{F}, \mathcal{V})$. If $s \xrightarrow[\mathsf{evb}]{}_{\mathbb{U}(R)}^* t$ then $s \xrightarrow[\mathsf{evb}]{}_{\mathbb{U}_{\mathrm{opt}}(R)}^* t$.*

Proof. Rules in $\mathbb{U}_{\mathrm{opt}}(R)$ that may contain extra variables are of the form $U_k^\rho(t_k, \overrightarrow{Y_k}) \to r$ where $\rho : l \to r \Leftarrow s_1 \twoheadrightarrow t_1; \cdots; s_k \twoheadrightarrow t_k$ in R. By the definition of $\mathbb{U}$, we have that $\mathbb{U}_k^\rho(t_k, \overrightarrow{X_k}) \to r$ in $\mathbb{U}(R)$, $Y_k \subseteq X_k$, and $\mathcal{V}ar(r) \cap Y_k = \mathcal{V}ar(r) \cap X_k$. Thus, we have that $\mathcal{EV}ar(U_k^\rho(t_k, \overrightarrow{Y_k}) \to r) = \mathcal{V}ar(r) \setminus (\mathcal{V}ar(t_k) \cup Y_k) = \mathcal{V}ar(r) \setminus (\mathcal{V}ar(t_k) \cup X_k) = \mathcal{EV}ar(U_k^\rho(t_k, \overrightarrow{X_k}) \to r)$. Thus, this theorem follows from Lemma 4.17. ◀

Finally, it can be said that soundness of $\mathbb{U}_{\mathrm{opt}}$ implies that of $\mathbb{U}$.

▶ **Theorem 4.19.** *$\mathbb{U}$ is sound for an eDCTRS (w.r.t. EV-basic reduction of $\mathbb{U}(R)$) if $\mathbb{U}_{\mathrm{opt}}$ is sound for the eDCTRS (w.r.t. EV-basic reduction of $\mathbb{U}_{\mathrm{opt}}(R)$).*

Proof. Let R be an eDCTRS over a signature $\mathcal{F}$, and s, t be terms in $\mathcal{T}(\mathcal{F}, \mathcal{V})$ such that $s \to_{\mathbb{U}(R)}^* t$. Then, it follows from Lemma 4.16 that $s \to_{\mathbb{U}_{\mathrm{opt}}(R)}^* t$. Moreover, it follows from soundness of $\mathbb{U}_{\mathrm{opt}}$ that $s \to_R^* t$. Therefore, $\mathbb{U}$ is sound. In the same way, the case of the EV-basic reduction can be proved. ◀

▶ **Corollary 4.20.** *$\mathbb{U}$ is sound for 3-DCTRSs that are $\mathbb{U}$-LL or $\mathbb{U}_{\mathrm{opt}}$-RL-NE.*

The converse of Theorem 4.19 does not hold. For example, $\mathbb{U}$ is sound for the DCTRS $(R_4)^{-1}$ in Example 4.14 but $\mathbb{U}_{\mathrm{opt}}$ is not sound. The reason must be that U symbols introduced via the application of $\mathbb{U}$ have more variables (i.e., information) than the corresponding U symbols introduced by $\mathbb{U}_{\mathrm{opt}}$. Thus, $\mathbb{U}$ is sufficient to produce TRSs that can be used instead of the original DCTRSs. Though, $\mathbb{U}_{\mathrm{opt}}$ will be useful in investigating soundness of $\mathbb{U}$ since the unraveled TRSs obtained by $\mathbb{U}_{\mathrm{opt}}$ are simpler than those obtained by $\mathbb{U}$.

5 Conclusion

In this paper, we have shown that the unravelings for DCTRSs are sound for DCTRSs that are ultra-LL or ultra-RL-NE, and shown that Ohlebusch's unraveling is sound for a DCTRS if the optimized one is sound for the DCTRS. We have also shown necessary and sufficient syntactic conditions for ultra-LL, ultra-RL, and ultra-NE, respectively. Future work is to relax these syntactic conditions for the soundness, e.g., that each rule is ultra-LL or ultra-RL-NE.

Extending the results in [5], it is shown in [6] that $\mathbb{U}$ is sound for confluent 3-DCTRSs w.r.t. the reduction to normal forms, and $\mathbb{U}$ is sound for 3-DCTRSs that are $\mathbb{U}$-RL or weakly left-linear. For the case of $\mathbb{U}$-RL 3-DCTRSs, this result is incompatible with Theorem 4.12 (see Example 4.13). For the case of weakly left-linear 3-DCTRS, this result is strictly stronger than Corollary 4.6 since $\mathbb{U}_{\mathrm{opt}}$-LL 3-DCTRSs are weakly left-linear. Extending the results in [6] w.r.t. confluence and weak left-linearity to $\mathbb{U}_{\mathrm{opt}}$ is a further direction of this research.

Acknowledgements

We would like to thank the anonymous reviewers for their useful comments to improve this paper. This research was partially supported by MEXT KAKENHI #17700009, #21700011 and #20300010, and Kayamori Foundation of Informational Science Advancement.

References

1 T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236(1-2):133–178, 2000.

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

3 F. Durán, S. Lucas, C. Marché, J. Meseguer, and X. Urbain. Proving operational termination of membership equational programs. *Higher-Order and Symbolic Computation*, 21(1-2):59–88, 2008.

4 F. Durán, S. Lucas, J. Meseguer, C. Marché, and X. Urbain. Proving termination of membership equational programs. In *Proc. of the 2004 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*, pp. 147–158, ACM, 2004.

5 K. Gmeiner, B. Gramlich, and F. Schernhammer. On (un)soundness of unravelings. In *Proc. of the 21st International Conference on Rewriting Techniques and Applications*, Vol. 6 of *LIPIcs*, pp. 119–134, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2010.

6 K. Gmeiner, B. Gramlich, and F. Schernhammer. Soundness conditions for unraveling deterministic conditional rewrite systems. Draft version, Jan. 2011.

7 J.-M. Hullot. Canonical forms and unification. In *Proc. of the 5th International Conference on Automated Deduction*, Vol. 87 of *Lecture Notes in Computer Science*, pp. 318–334, 1980.

8 K. Kusakari, M. Nakamura, and Y. Toyama. Argument filtering transformation. In *Proc. of the International Conference on Principles and Practice of Declarative Programming*, Vol. 1702 of *Lecture Notes in Computer Science*, pp. 47–61, Springer, 1999.

9 S. Lucas. Context-sensitive computations in functional and functional logic programs. *Journal of Functional and Logic Programming*, 1998(1), 1998.

10 S. Lucas, C. Marché, and J. Meseguer. Operational termination of conditional term rewriting systems. *Information Processing Letters*, 95(4):446–453, 2005.

11 M. Marchiori. Unravelings and ultra-properties. In *Proc. of the 5th International Conference on Algebraic and Logic Programming*, Vol. 1139 of *Lecture Notes in Computer Science*, pp. 107–121, Springer, 1996.

12 M. Marchiori. On deterministic conditional rewriting. Computation Structures Group, Memo 405, MIT Laboratory for Computer Science, 1997.

13 A. Middeldorp and E. Hamoen. Completeness results for basic narrowing. *Applicable Algebra in Engineering, Communication and Computing*, 5:213–253, 1994.

14 N. Nishida. *Transformational Approach to Inverse Computation in Term Rewriting*. Doctor thesis, Nagoya University, Jan. 2004.

15 N. Nishida and M. Sakai. Completion after program inversion of injective functions. In *Proc. of the 8th International Workshop on Reduction Strategies in Rewriting and Programming*, Vol. 237 of *Electronic Notes in Theoretical Computer Science*, pp. 39–56, Apr. 2009.

16 N. Nishida, M. Sakai, and T. Sakabe. Narrowing-based simulation of term rewriting systems with extra variables and its termination proof. *Functional and Constraint Logic Programming*, *Electronic Notes in Theoretical Computer Science*, 86(3):1–18, Nov. 2003.

17 N. Nishida, M. Sakai, and T. Sakabe. On simulation-completeness of unraveling for conditional term rewriting systems. IEICE Technical Report SS2004-18, IEICE, Vol. 104, No. 243, pp. 25–30, Aug. 2004.

18 N. Nishida, M. Sakai, and T. Sakabe. Generation of inverse computation programs of constructor term rewriting systems. *The IEICE Transactions on Information and Systems*, J88-D-I(8):1171–1183, Aug. 2005 (in Japanese).

19 N. Nishida, M. Sakai, and T. Sakabe. Partial inversion of constructor term rewriting systems. In *Proc. of the 16th International Conference on Rewriting Techniques and Applications*, Vol. 3467 of *Lecture Notes in Computer Science*, pp. 264–278, Springer, 2005.

20 E. Ohlebusch. Termination of logic programs: Transformational methods revisited. *Applicable Algebra in Engineering, Communication and Computing*, 12(1-2):73–116, 2001.

21 E. Ohlebusch. *Advanced Topics in Term Rewriting*. Springer-Verlag, April 2002.

22 F. Schernhammer and B. Gramlich. Characterizing and proving operational termination of deterministic conditional term rewriting systems. *The Journal of Logic and Algebraic Programming – Revised selected papers of NWPT 2008*, 79(7):659–688, Oct. 2010.

23 F. Schernhammer and B. Gramlich. On proving and characterizing operational termination of deterministic conditional rewrite systems. In *Proc. of the 9th International Workshop on Termination*, pp. 82–85, June 2007.

24 T.-F. Serbanuta and G. Rosu. Computationally equivalent elimination of conditions. In *Proc. of the 17th International Conference on Rewriting Techniques and Applications*, Vol. 4098 of *Lecture Notes in Computer Science*, pp. 19–34 Springer, 2006.

25 Y. Toyama. Confluent term rewriting systems with membership conditions. In *Proc. of the 1st International Workshop on Conditional Term Rewriting Systems*, Vol. 308 of *Lecture Notes in Computer Science*, pp. 228–241, Springer, 1987.

26 P. Viry. Elimination of conditions. *Journal of Symbolic Computation*, 28(3):381–401, 1999.

Program Inversion for Tail Recursive Functions

Naoki Nishida[1] and Germán Vidal[2]

1 **Graduate School of Information Science, Nagoya University**
Furo-cho, Chikusa-ku, 4648603 Nagoya, Japan
nishida@is.nagoya-u.ac.jp

2 **MiST, DSIC, Universitat Politècnica de València**
Camino de Vera, s/n, 46022 Valencia, Spain
gvidal@dsic.upv.es

Abstract

Program inversion is a fundamental problem that has been addressed in many different programming settings and applications. In the context of term rewriting, several methods already exist for computing the inverse of an injective function. These methods, however, usually return non-terminating inverted functions when the considered function is tail recursive. In this paper, we propose a direct and intuitive approach to the inversion of tail recursive functions. Our new technique is able to produce good results even without the use of an additional post-processing of determinization or completion. Moreover, when combined with a traditional approach to program inversion, it constitutes a promising approach to define a general method for program inversion. Our experimental results confirm that the new technique compares well with previous approaches.

1998 ACM Subject Classification F.4.2 Grammars and Other Rewriting Systems

Keywords and phrases term rewriting, program transformation, termination

Digital Object Identifier 10.4230/LIPIcs.RTA.2011.283

Category Regular Research Paper

1 Introduction

Inverse computation for an n-ary function f is, given an output v of f, the calculation of (all) the possible inputs $v_1, \cdots, v_1$ of f such that $f(v_1, \cdots, v_n) = v$ [27, 28]. To be more precise this is usually called *full* inverse computation, in contrast to *partial* one where some inputs are also provided, i.e., given the output v of f and part of its inputs, say $v_{i_1}, \ldots, v_{i_m}$, the partial inverse computation computes the remaining inputs $v_{j_1}, \ldots, v_{j_k}$ such that $f(v_1, \ldots, v_n) = v$ with $\{v_{i_1}, \ldots, v_{i_m}\} \cup \{v_{j_1}, \ldots, v_{j_k}\} = \{v_1, \ldots, v_n\}$ and $\{v_{i_1}, \ldots, v_{i_m}\} \cap \{v_{j_1}, \ldots, v_{j_k}\} = \emptyset$. Two approaches to inverse computation are distinguished [1]: *inverse interpreters* [4, 1, 32, 15] that perform inverse computation taking the output v, the given inputs (if any), and the definition of f as input, and *inversion compilers* [16, 9, 12, 27, 28, 22, 20, 24, 23, 6, 7, 11, 2] that performs *program inversion*. More precisely, inversion compilers take the definition of f as input and compute the definition of a (possibly partial) inverse function f^{-1}. Note that inverse interpreters can be transformed into inversion compilers by producing inverse functions including an inverse interpreter in the target programming language that is specialized for the function being inverted (similarly to, e.g., [15]). *Semi-inversion* [17, 18, 19] is a more general notion than partial inversion that allows the original output to be partially given.

22nd International Conference on Rewriting Techniques and Applications (RTA'11).
Editor: M. Schmidt-Schauß; pp. 283–298

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


Typical applications of (full and partial) program inversion are the automatic development of dual programs: encryption and decryption (e.g., cryptographic encoder $\mathsf{enc}(x,k)$ and decoder $\mathsf{dec}(y,k)$ with a symmetric key k), data compression and decompression (e.g., zip/unzip), data/program translation and re-translation between different programming languages, and so on. Given one-half of dual programs, inversion provides the other program automatically and without bugs, thus guaranteeing high reliability. This is specially important for encryption/decryption and compression/decompression since a bug in these programs may cause serious security problems.

The most popular target of program inversion is the class of injective functions (or functions that are injective w.r.t. the unknown arguments when *partial* inversion is considered). Deterministic definitions are expected as inverses of injective functions since the inverse relation for injective functions is one-to-one. However, this is not ensured by existing methods and in many cases overlapping and/or non-terminating functions are produced instead. Thus, the elimination of non-determinism in inverted functions has recently attracted a lot of interest [6, 7, 11, 2, 21]. Indeed, since all the inversion techniques developed so far are essentially similar, one can say that the difficult part of program inversion for injective functions lies in the elimination of the undesired non-determinism.

In the field of term rewriting, a full-inversion method for constructor term rewriting systems (TRS, for short) has been proposed, and later extended to partial inversion [20, 24, 23]. The compiler (fully or partially) inverts a constructor TRS into a conditional term rewriting system (CTRS, for short) that completely defines inverses of functions defined in the original TRS. The *conditional parts* of rewrite rules in the CTRS can be seen as `let`-structures for declaring variables that are locally used in the rewrite rules.

The method for eliminating non-determinism in [21], a post process for the compilers in [20, 24, 23], consists in applying a restricted variant of *completion* to the systems obtained from the inverse conditional systems by *unraveling* [14, 26]. This method requires the conditional systems to be *operationally terminating* [13] (i.e., any derivation is finite) and outputs computationally-equivalent unconditional systems that are terminating and non-overlapping when the method halts successfully. Although this method is quite restrictive and does not always succeed, it was able to successfully transform all the benchmarks shown in [10, 6, 7, 11] with operationally terminating inverses [21], while it was not applicable to the other benchmarks with non-operationally-terminating inverses (see Example 3.2 below). On the other hand, the method for eliminating non-determinism in [6, 7, 11] is based on applying LR parsing techniques to a grammar-based representation of functional programs. This is quite an interesting non-standard application of LR parsing and performs surprisingly well for the benchmarks of [10] that contain several kinds of schemes of function definitions (such as tail-recursion, non-tail-recursion, and the combination of both), despite the fact that only LR(0) parsing is considered. The authors do not consider partial inversion since the grammar-based representation is not adequate to identify known and unknown arguments separately. Moreover, the grammar-based programs cannot express functions containing erasing rules (though these rules arise quite naturally when considering partially inverted programs).

Given an n-ary function f, traditional approaches to (partial) inversion are based on the property "$f(v_1,\cdots,v_n)=v$ iff $f^{-1}(v,v_{i_1},\cdots,v_{i_m})=(v_{j_1},\cdots,v_{j_k})$ where $\{i_1,\cdots,i_m\}$ are known input arguments and $\{i_1,\cdots,i_m\}\uplus\{j_1,\cdots,j_k\}=\{1,\cdots,n\}$", i.e., the equation $f(v_1,\cdots,v_n)=v$ is replaced by $f^{-1}(v,v_{i_1},\cdots,v_{i_m})=(v_{j_1},\cdots,v_{j_k})$ and instruction sequences are inverted [27, 28, 22, 20, 24, 23, 6, 7, 11, 2]. For example, when considering full inversion, a rewrite rule is normalized to a conditional rule $f(t_1,\cdots,t_n)\to t \Leftarrow$

$f_1(\overrightarrow{u_1}) \twoheadrightarrow x_1; \cdots ; f_k(\overrightarrow{u_k}) \twoheadrightarrow x_k$ (see Definition 3.4), and it is inverted to a conditional rule $f^{-1}(t) \to (t_1, \cdots, t_n) \Leftarrow f_k^{-1}(x_k) \twoheadrightarrow (\overrightarrow{u_k}); \cdots ; f_1^{-1}(x_1) \twoheadrightarrow (\overrightarrow{u_1})$ where $t_1, \cdots, t_n, t$ are constructor terms, $\overrightarrow{u_1}, \cdots, \overrightarrow{u_k}$ are sequences of constructor terms, and $x_1, \cdots, x_n$ are variables (see Example 3.1 below). This approach is in principle applicable to arbitrary functions. Unfortunately, for a tail-recursive function defined by a rule like $f(t_1, \cdots, t_n) \to f(u_1, \cdots, u_n)$, this approach generates a non-operationally-terminating rule of the form $f^{-1}(x) \to (t_1, \cdots, t_n) \Leftarrow f^{-1}(x) \twoheadrightarrow (u'_1, \cdots, u'_n); \cdots$ when full inversion is considered.

When the first argument of the inverse f^{-1} takes as input the output of the original function f (i.e., the input value of the first argument of f^{-1} is in the range of f), a breadth-first search is enough to get the output (i.e., the original input) since there exists a finite path from the original input to the original output. Therefore, for a non-operationally-terminating inverse system of an injective function, a breadth-first search is enough to compute the output. However, the finiteness of the breadth-first search is guaranteed only when the input is one of the outputs of the original function. Thus, in general, the breadth-first search might be non-terminating (i.e., the search space might be infinite). Furthermore, when the given function is not surjective (which is itself difficult to know), it is not easy to determine whether the input is one of the outputs of the original function or not. Moreover, practical rewriting systems (or functional programming environments) do not usually implement breadth-first search strategies. For these reasons, the (operational) termination of inverted systems is desired.

As stated above, the non-determinism elimination method of [6, 7, 11] can solve the non-determinism of some inverted programs so that the resulting system is terminating. This method, however, does not succeed for all inverted systems. On the other hand, the method in [21] cannot be applied to any non-operationally-terminating system since the method requires termination. As mentioned before, when full inversion is considered, tail recursive rules of the form $f(t_1, \cdots, t_n) \to f(u_1, \cdots, u_n)$ are inverted to $f^{-1}(x) \to (t_1, \cdots, t_n) \Leftarrow f^{-1}(x) \twoheadrightarrow (u'_1, \cdots, u'_n); \cdots$ that cause non-termination (since $f^{-1}(x)$ calls $f^{-1}(x)$ again). Therefore, the traditional approaches to inversion are not suitable for tail recursive rules (unless some post-processing is applied to recover more suitable definitions). Actually, the basic methods for program inversion have not been significantly improved since their original definitions (the focus has been in the development of methods for eliminating non-determinism instead). However, we think that there is still room for improving the current inversion techniques.

In this paper, we introduce a novel approach to program inversion which is specially tailored to deal with tail recursive functions defined by means of a rewrite system. We combine this approach with the previous technique in [20, 23] to produce a general inversion method. For the sake of readability, we only consider the full inversion of functions (as in [20, 23]), though it would not be difficult to combine our technique with the approach to partial inversion of [20, 24]. In addition, we do not consider *sorts* in this paper, though the results can be straightforwardly extended to many-sorted systems. Our research is motivated by the fact that tail recursive functions are extensively used due to their good computational properties (i.e., they are usually compiled as iterations, which are much more efficient than standard recursive functions).

This paper is organized as follows. In Section 2 we briefly review notions and notations of term rewriting. Section 3 introduces an inversion method for tail recursive functions. Section 4 shows a summary of our experimental evaluation and compare the new approach with related works. Finally, Section 5 concludes and points out some directions for future research. Proofs of technical results can be found in the extended version [25] of this paper.

2 Preliminaries

In this section, we recall some basic notions and notations of term rewriting [3, 26].

Throughout this paper, we use $\mathcal{V}$ as a countably infinite set of *variables*. Let $\mathcal{F}$ be a *signature*, i.e., a finite set of *function symbols* with a fixed arity denoted by $\mathsf{ar}(f)$ for a function symbol f. The set of *terms* over $\mathcal{F}$ and $\mathcal{V}$ is denoted by $\mathcal{T}(\mathcal{F},\mathcal{V})$, and the set of variables appearing in terms $t_1,\cdots,t_n$ is denoted by $\mathcal{V}ar(t_1,\cdots,t_n)$. The *identity* of terms s and t is written by $s \equiv t$. The notation $C[t_1,\cdots,t_n]_{p_1,\cdots,p_n}$ represents the term obtained by replacing each *hole* $\Box$ at *position* p_i of an n-hole *context* $C[\ ]$ with term t_i for $1 \leq i \leq n$. We may omit the subscripts $p_1,\cdots,p_n$ when they are clear from the context. The *domain* and *range* of a *substitution* σ are denoted by $\mathcal{D}om(\sigma)$ and $\mathcal{R}an(\sigma)$, respectively; a substitution σ will be denoted by $\{x_1 \mapsto t_1,\cdots,x_n \mapsto t_n\}$ if $\mathcal{D}om(\sigma) = \{x_1,\cdots,x_n\}$ and $\sigma(x_i) \equiv t_i$ for $1 \leq i \leq n$. The set of variables appearing in the range of σ is denoted by $\mathcal{VR}an(\sigma)$: $\mathcal{VR}an(\sigma) = \bigcup_{x\in\mathcal{D}om(\sigma)} \mathcal{V}ar(x\sigma)$. The application $\sigma(t)$ of substitution σ to term t is abbreviated to $t\sigma$.

An (*oriented*) *conditional rewrite rule* over a signature $\mathcal{F}$ is a triple (l,r,c), denoted by $l \to r \Leftarrow c$, such that the *left-hand side* l is a non-variable term of $\mathcal{T}(\mathcal{F},\mathcal{V})$, the *right-hand side* r is a term of $\mathcal{T}(\mathcal{F},\mathcal{V})$, and the *conditional part* c is a sequence $s_1 \twoheadrightarrow t_1;\cdots;s_k \twoheadrightarrow t_k$ $(k \geq 0)$ where $s_1,\cdots,s_k,t_1,\cdots,t_k$ are terms of $\mathcal{T}(\mathcal{F},\mathcal{V})$. In particular, the rewrite rule is called *unconditional* if the conditional part is the empty sequence (i.e., $k = 0$), and we may abbreviate it to $l \to r$. We sometimes attach a unique label ρ to rule $l \to r \Leftarrow c$, written $\rho : l \to r \Leftarrow c$, so that we can use the label to refer to this rule. The set of variables in c and ρ are denoted by $\mathcal{V}ar(c)$ and $\mathcal{V}ar(\rho)$, resp.: $\mathcal{V}ar(s_1 \twoheadrightarrow t_1;\cdots;s_k \twoheadrightarrow t_k) = \mathcal{V}ar(s_1,t_1,\cdots,s_k,t_k)$ and $\mathcal{V}ar(\rho) = \mathcal{V}ar(l,r,c)$.

An (*oriented*) *conditional term rewriting system* (*CTRS*, for short) R over a signature $\mathcal{F}$ is a finite set of conditional rewrite rules over $\mathcal{F}$. In particular, R is called an (*unconditional*) *term rewriting system* (*TRS*, for short) if every rule $l \to r \Leftarrow c$ in R is unconditional and satisfies $\mathcal{V}ar(l) \supseteq \mathcal{V}ar(r)$. The *rewrite relation* $\to_R$ of R is defined as follows: $\to_{(0),R} = \emptyset$, $\to_{(i+1),R} = \to_{(i),R} \cup\{(C[l\sigma],C[r\sigma]) \mid l \to r \Leftarrow s_1 \twoheadrightarrow t_1;\cdots;s_k \twoheadrightarrow t_k \in R,\ s_1\sigma \to^*_{(i),R} t_1\sigma,\ \cdots,\ s_k\sigma \to^*_{(i),R} t_k\sigma\}$ for $i \geq 0$, and $\to_R = \bigcup_{i\geq 0} \to_{(i),R}$. A notion of *operational termination* of CTRSs is defined via the absence of infinite well-formed proof trees in some inference system [13]: a CTRS R is *operationally terminating* if for any terms s and t, any proof tree attempting to prove that $s \to^*_R t$ cannot be infinite.

A conditional rewrite rule $l \to r \Leftarrow s_1 \twoheadrightarrow t_1;\cdots;s_k \twoheadrightarrow t_k$ is called *deterministic* if $\mathcal{V}ar(s_i) \subseteq \mathcal{V}ar(l,t_1,\cdots,t_{i-1})$ for $1 \leq i \leq k$. Note that the terminology "deterministic" refers to the sequential evaluation of conditional parts, i.e., when a CTRS is deterministic, the conditional parts can be evaluated from left to right. In contrast, the determinacy of the rewrite rule application is denoted with the terminology "non-overlapping". A CTRS is called *deterministic* (*DCTRS*, for short) if all of its rules are deterministic. A CTRS is called *non-erasing* if every rule $l \to r \Leftarrow c$ satisfies $\mathcal{V}ar(l) \subseteq \mathcal{V}ar(r)$.

Let R be a CTRS over a signature $\mathcal{F}$. The set of *defined symbols of* R is denoted by $\mathcal{D}_R = \{\mathsf{root}(l) \mid l \to r \Leftarrow c \in R\}$ and the set of *constructors of* R is denoted by $\mathcal{C}_R = \mathcal{F} \setminus \mathcal{D}_R$. A term in $\mathcal{T}(\mathcal{C}_R,\mathcal{V})$ is called a *constructor term of* R. The CTRS R is called a *constructor system* if for each rule in R, any proper subterm of the left-hand side is a constructor term of R. A substitution σ is called a *constructor substitution of* R if $\mathcal{R}an(\sigma) \subseteq \mathcal{T}(\mathcal{C}_R,\mathcal{V})$. For a term t, $\mathsf{cap}(t)$ is a term obtained by replacing each proper subterm rooted by a defined symbol with a fresh variable [26].

As a model of the call-by-value evaluation, we define the *constructor-based reduction*

relation $\xrightarrow[c]{}_R$ of a CTRS R, a restricted variant of *constructor rewriting* in [31], as follows:

- $\xrightarrow[c]{}_{(0),R} = \emptyset$,
- $\xrightarrow[c]{}_{(n+1),R} = \xrightarrow[c]{}_{(n),R} \cup \{(C[l\theta], C[r\theta]) \mid l \to r \Leftarrow s_1 \twoheadrightarrow t_1; \cdots; s_k \twoheadrightarrow t_k \in R,\ \mathcal{R}an(\theta) \subseteq \mathcal{T}(\mathcal{C}_R, \mathcal{V}),\ s_1\theta \xrightarrow[c]{}^*_{(n),R} t_1\theta,\ \cdots,\ s_k\theta \xrightarrow[c]{}^*_{(n),R} t_k\theta\}$, where $n \geq 0$, and
- $\xrightarrow[c]{}_R = \bigcup_{i>0} \xrightarrow[c]{}_{(i),R}$.

Roughly speaking, for a CTRS that can be regarded as a functional program, the constructor-based reduction corresponds to the *call-by-value* evaluation where constructor terms are considered as data objects. Note that $\xrightarrow[c]{}_R$ is a strict subrelation of the *operationally innermost reduction* [21], the innermost reduction of DCTRSs, if every rule $l \to r \Leftarrow s_1 \twoheadrightarrow t_1; \cdots; s_k \twoheadrightarrow t_k$ satisfies that $t_1, \cdots, t_k \in \mathcal{T}(\mathcal{C}_R, \mathcal{V})$. Unlike the innermost reduction of CTRSs (cf. [8]), $\xrightarrow[c]{}_R$ is well-defined for every CTRS since constructor terms are well-defined while normal forms of non-operationally-terminating CTRSs are not well-defined.

Let R be a CTRS. Two rewrite rules $l_1 \to r_1 \Leftarrow c_1$ and $l_2 \to r_2 \Leftarrow c_2$ are called *overlapping* if there exists a context $C[\,]$ and a non-variable term t such that $l_2 \equiv C[t]$ and l_1 and t are unifiable, where we assume w.l.o.g. that these rules share no variable. Then, a conditional pair of terms $((C[r_1])\theta, r_2\theta) \Leftarrow c_1\theta; c_2\theta$ is called a *critical pair* of R where θ is a most general unifier of l_1 and t. A critical pair $(s, t) \Leftarrow c$ is called *trivial* if $s \equiv t$, and it is called *infeasible w.r.t.* $\xrightarrow[c]{}_R$ if for any substitution σ, c contains a condition $u \twoheadrightarrow v$ such that $u\sigma \not\xrightarrow[c]{}^*_R v\sigma$ [33].

Let R be a DCTRS and f be a defined symbol of R. A rule $l \to r$ in R is called an *f-rule* if $\mathsf{root}(l)$ is f. We denote the set of f-rules in R by $R|_f$. For a set D of defined symbols, $R|_D$ denotes $\bigcup_{g \in D} R|_g$. The set $\mathsf{Dep}_R(f)$ of defined symbols that f *depends on* is the minimum set such that

- $\mathsf{Dep}_R(f)$ contains every defined symbol appearing in r, c of $l \to r \Leftarrow c \in R|_f$, and
- $\mathsf{Dep}_R(f) \supseteq \mathsf{Dep}_R(g)$ if $g \in \mathsf{Dep}_R(f)$.

The symbol f is called (*mutually*) *recursive* if there exists a symbol g in $\mathsf{Dep}_R(f)$ that depends on f. Moreover, f is called *self-recursive* if the only such symbol is f itself. An f-rule $l \to r$ is called a *non-recursive rule* if any defined symbol in r does not depend on f; otherwise, the f-rule is called a *recursive rule*. Let $t_1, \cdots, t_n$ be constructor terms of R. Then the term $f(t_1, \cdots, t_n)$ is called a *call pattern*.

We introduce special constructors $\mathsf{tp}_0, \mathsf{tp}_1, \mathsf{tp}_2, \cdots$ with $\mathsf{ar}(\mathsf{tp}_i) = i$ in order to denote tuples (records) of terms, e.g., the tuple $(t_1, \cdots, t_n)$ of terms $t_1, \cdots, t_n$ is denoted by $\mathsf{tp}_n(t_1, \cdots, t_n)$. The reason why these constructors are introduced is that the inverses of n-ary functions with $n > 1$ will return tuples of terms. A CTRS R is called tp*-free* if it does not contain any tuple symbol. Although it may seem trivially correct in a functional setting, we will never replace $\mathsf{tp}_1(t)$ by t in our term rewriting context. The reason is that, given a CTRS R that is operationally terminating w.r.t. $\to_R$, the CTRS obtained from R by replacing each occurrence of $\mathsf{tp}_1(t)$ by t is not always operationally terminating w.r.t. $\to_R$ while it is operationally terminating w.r.t. $\xrightarrow[c]{}_R$.

We are now ready to introduce our notion of full inverses for functions:

▶ **Definition 2.1** (full inverse)**.** Let R be a tp-free CTRS over a signature $\mathcal{F}$ and S be a CTRS over a signature $\mathcal{G}$ such that $\mathcal{C}_R \subseteq \mathcal{C}_S$. A defined symbol g of S is called a *full inverse* of f if

- for any constructor terms $t_1, \cdots, t_n, u$ of R, $f(t_1, \cdots, t_n) \xrightarrow[c]{}^*_R u$ if and only if $g(u) \xrightarrow[c]{}^*_S \mathsf{tp}_n(t_1, \cdots, t_n)$.

In the following, the full inverse of a function f is denoted by f^{-1}. We say that a rewrite system S is a *full inverse system of f in R* if S defines f^{-1}.

3 Inversion Transformation for Tail Recursive Functions

In this section, we introduce a method for the inversion of constructor TRSs which extends the previous approach of [20, 23] with an appropriate technique for the inversion of tail recursive functions.

▶ **Example 3.1.** Consider the following rewrite system defining the well-known reverse function with an accumulating parameter:

$$R_{\mathsf{reverse}} = \left\{ \begin{array}{c} \mathsf{reverse}(xs) \to \mathsf{rev}(xs, \mathsf{nil}) \\ \mathsf{rev}(\mathsf{nil}, ws) \to ws \\ \mathsf{rev}(\mathsf{cons}(x, xs), ys) \to \mathsf{rev}(xs, \mathsf{cons}(x, ys)) \end{array} \right\}$$

This system is inverted by the previous inversion technique [20, 23, 6, 7, 11] as follows:

$$\overline{R_{\mathsf{reverse}}} = \left\{ \begin{array}{l} \mathsf{reverse}^{-1}(ys) \to \mathsf{tp}_1(xs) \Leftarrow \mathsf{rev}^{-1}(ys) \twoheadrightarrow \mathsf{tp}_2(xs, \mathsf{nil}) \\ \quad \mathsf{rev}^{-1}(ws) \to \mathsf{tp}_2(\mathsf{nil}, ws) \\ \quad \mathsf{rev}^{-1}(zs) \to \mathsf{tp}_2(\mathsf{cons}(x, xs), ys) \Leftarrow \mathsf{rev}^{-1}(zs) \twoheadrightarrow \mathsf{tp}_2(xs, \mathsf{cons}(x, ys)) \end{array} \right\}$$

The functions $\mathsf{reverse}$ and $\mathsf{reverse}^{-1}$ are full inverses of each other, and the functions rev and rev^{-1} are full inverses of each other too. However, $\mathsf{reverse}$ is injective but rev is not. Unfortunately, this inverse system is neither non-overlapping nor operationally terminating. Moreover, the method in [21] is not applicable because the inverted system is not operationally terminating. Luckily, for this example, the non-determinism elimination method in [6, 7, 11] can transform $\overline{R_{\mathsf{reverse}}}$ into a computationally equivalent CTRS that is operationally terminating and non-overlapping. However, this method is not always successful (see Example 3.14 below) and thus we plan to introduce instead a direct approach that is able to produce the right inverse without the need of a post-process (which nevertheless will be still applicable when the direct approach does not succeed).

For the sake of readability, we only consider functions defined by *unconditional* and non-erasing constructor TRSs as a target of inversion. We also require *tail recursive* functions to be *self-recursive*[1] according to the following scheme:

- there is a single non-recursive rule $f(w_1, \cdots, w_n) \to r$ such that the right-hand side r is a constructor term, and
- there are one or more recursive rules of the form $f(t_1, \cdots, t_n) \to f(u_1, \cdots, u_n)$.

These restrictions are not essential and thus the results shown later can easily be extended to functions that do not fulfill them. Given a tail recursive function f, a call pattern $f(t_1, \cdots, t_m)$ is called *initial* if it is a renamed variant of $\mathsf{cap}(t)$ where t is either

- a subterm of the right-hand side of some rule which is not an f-rule, or
- a strict subterm of the right-hand side of some recursive f-rule.

Intuitively speaking, initial call patterns represent initial calls to tail recursive functions.

▶ **Example 3.2.** Consider again the TRS R_{reverse} of Example 3.1. Then, the only initial call of rev in R_{reverse} is $\mathsf{rev}(xs, \mathsf{nil})$.

The next definition introduces a notion of inverse which is specially tailored to tail recursive functions.

[1] It is possible in general to transform mutually recursive functions to a computationally equivalent self-recursive functions, by adding an extra argument to each function to show the original function and by replacing all the recursive function symbols with the same fresh one. However, this transformation is not necessary since the inversion shown later can easily be extended to tail mutually-recursive functions.

▶ **Definition 3.3** (tail recursive inverse)**.** Let R be a tp-free CTRS over a signature $\mathcal{F}$ and S be a CTRS over a signature $\mathcal{G}$ such that $\mathcal{C}_R \subseteq \mathcal{C}_S$. Given a tail recursive function f of $\mathcal{R}$, a defined symbol g of S is called a *tail recursive inverse of* f if

- for any constructor terms $t_1, \cdots, t_n$ of R, $f(t_1, \cdots, t_n) \xrightarrow[c]{}{}^*_R u$ if and only if $u \equiv r\sigma$ and $g(w_1, \cdots, w_n)\sigma \xrightarrow[c]{}{}^*_S \mathsf{tp}_n(t_1, \cdots, t_n)$ for a constructor substitution σ of R and a non-recursive f-rule $f(w_1, \cdots, w_n) \to r$.

In the following, we denote by $\widetilde{f}$ the tail recursive inverse of f.

Let us first illustrate our technique with an example. In Example 3.1 above, one can observe that the computation of reverse has the following scheme:

$$\begin{aligned}
&\mathsf{reverse}(xs)\theta_0 \to \\
&\mathsf{rev}(xs, \mathsf{nil})\theta_0 \equiv \mathsf{rev}(\mathsf{cons}(x_1, xs_1), ys_1)\theta_1 &&\to \mathsf{rev}(xs_1, \mathsf{cons}(x_1, ys_1))\theta_1 \\
&\qquad\equiv \mathsf{rev}(\mathsf{cons}(x_2, xs_2), ys_2)\theta_2 &&\to \mathsf{rev}(xs_2, \mathsf{cons}(x_2, ys_2))\theta_2 \\
&\qquad\qquad\vdots \\
&\qquad\equiv \mathsf{rev}(\mathsf{cons}(x_n, xs_n), ys_n)\theta_n \to \mathsf{rev}(xs_n, \mathsf{cons}(x_n, ys_n))\theta_n &&\equiv \mathsf{rev}(\mathsf{nil}, ws)\theta \to ws\theta
\end{aligned}$$

Therefore, the inverse function $\mathsf{reverse}^{-1}$ should start with a call to $\mathsf{rev}(\mathsf{nil}, ws)\theta$ and should end with a call to $\mathsf{rev}(xs, \mathsf{nil})\theta_0$. Then, the computation of $\widetilde{\mathsf{rev}}$ should follow the pattern

$$\begin{aligned}
\widetilde{\mathsf{rev}}(\mathsf{nil}, ws)\theta \equiv \widetilde{\mathsf{rev}}(xs_n, \mathsf{cons}(x_n, ys_n))\theta_n &\to \widetilde{\mathsf{rev}}(\mathsf{cons}(x_n, xs_n), ys_n)\theta_n \\
&\vdots \\
\equiv \widetilde{\mathsf{rev}}(xs_1, \mathsf{cons}(x_1, ys_1))\theta_1 &\to \widetilde{\mathsf{rev}}(\mathsf{cons}(x_1, xs_1), ys_1)\theta_1 \equiv \widetilde{\mathsf{rev}}(xs, \mathsf{nil})\theta_0
\end{aligned}$$

so that it outputs $\mathsf{tp}_2(xs, \mathsf{nil})\theta_0$. Thus, this pattern means that we should require $\widetilde{\mathsf{rev}}(\mathsf{nil}, ws)$ to be reduced to $\mathsf{tp}_2(xs, \mathsf{nil})$. To summarize, the inversion of the tail-recursive function rev will proceed as follows:

- First, we *normalize* the right-hand side of the rules in order to avoid defined function symbols (except for the topmost one when the function is tail-recursive; see the formal definition below):

$$\left\{\begin{array}{r} \mathsf{reverse}(xs) \to ys \Leftarrow \mathsf{rev}(xs, \mathsf{nil}) \twoheadrightarrow ys \\ \mathsf{rev}(\mathsf{nil}, ws) \to ws \\ \mathsf{rev}(\mathsf{cons}(x, xs), ys) \to \mathsf{rev}(xs, \mathsf{cons}(x, ys)) \end{array}\right\}$$

- Then, the first rule is transformed to

  $$\mathsf{reverse}(xs) \to ys \Leftarrow \mathsf{rev}(xs, \mathsf{nil}) \twoheadrightarrow ws;\ ws \twoheadrightarrow ys$$

  The condition $ws \twoheadrightarrow ys$ is added to make it explicit that ys should be equal to ws, the output of function rev (i.e., the right-hand side of the base case). Now, we basically proceed to exchange the left- and right-hand sides of every equation except for the call to the tail-recursive function rev which is transformed as explained above:

  $$\mathsf{reverse}^{-1}(ys) \to xs \Leftarrow ys \twoheadrightarrow ws;\ \widetilde{\mathsf{rev}}(\mathsf{nil}, ws) \twoheadrightarrow \mathsf{tp}_2(xs, \mathsf{nil})$$

- The intermediate reduction steps in the computation for $\widetilde{\mathsf{rev}}$ are done using the inverse of the recursive rule for rev (we just exchanged the left- and right-hand sides):

  $$\widetilde{\mathsf{rev}}(xs, \mathsf{cons}(x, ys)) \to \widetilde{\mathsf{rev}}(\mathsf{cons}(x, xs), ys)$$

- Finally, the base case for $\widetilde{\mathsf{rev}}$ has now the form

  $$\widetilde{\mathsf{rev}}(xs, \mathsf{nil}) \to \mathsf{tp}_2(xs, \mathsf{nil})$$

according to the reasoning above. Therefore, we get the following non-overlapping and operationally terminating system:

$$\overline{R_{\mathsf{reverse}}} = \left\{ \begin{array}{c} \mathsf{reverse}^{-1}(ws) \to xs \Leftarrow \widetilde{\mathsf{rev}}(\mathsf{nil}, ws) \twoheadrightarrow \mathsf{tp}_2(xs, \mathsf{nil}) \\ \widetilde{\mathsf{rev}}(xs, \mathsf{nil}) \to \mathsf{tp}_2(xs, \mathsf{nil}) \\ \widetilde{\mathsf{rev}}(xs, \mathsf{cons}(x, ys)) \to \widetilde{\mathsf{rev}}(\mathsf{cons}(x, xs), ys) \end{array} \right\}$$

where the condition $ys \twoheadrightarrow ws$ has been removed by substituting ws with ys.
The idea above is similar to that in [18]. However, there is no formalization nor correctness proofs in [18]. In the rest of this section, we formalize this idea and prove its correctness.

In the previous example, no nested function calls occurred in the right-hand sides of rev. However, we could easily extend the above approach to tail recursive functions whose right-hand sides have nested function calls by dealing with them inductively. Note that we keep applying the old approach of [20, 23] to non-tail-recursive functions.

The following definition introduces a pre-processing of normalization which is similar to that in [2] for functional programs with `let` expressions. The main difference is that the root symbols of the right-hand sides of tail recursive rules are not normalized in our case.

▶ **Definition 3.4** (normalization)**.** Let R be a constructor TRS and f be a defined symbol of R. For an f-rule $l \to r$, the normalized rewrite rule $\mathcal{N}(l \to r)$ is a conditional rewrite rule obtained by repeatedly applying the following transformation until the right-hand side has either no defined symbol or just one at the root position when f is tail recursive:

- Given a rule $l \to C[t] \Leftarrow c$ such that t is rooted by a defined symbol of R and has no defined symbol in its proper subterms, we transform it into $l \to C[x] \Leftarrow c;\ t \twoheadrightarrow x$, where x is a fresh variable.

The normalization $\mathcal{N}$ is extended to TRSs as follows $\mathcal{N}(R) = \{\mathcal{N}(l \to r) \mid l \to r \in R\}$.

The correctness of the normalization process is a consequence of the following results:

▶ **Lemma 3.5.** *Let R and S be constructor CTRSs such that $R = R_0 \uplus \{\rho : l \to C[r] \Leftarrow c\}$ and $S = R_0 \uplus \{\rho' : l \to C[x] \Leftarrow c; r \twoheadrightarrow x\}$ such that x is a fresh variable, r contains a defined symbol of R and, for each $s' \twoheadrightarrow t'$ in c, t' is a constructor term of R. Then, for all terms s and constructor terms t of R, $s \xrightarrow[\mathsf{c}]{}{}^*_R t$ iff $s \xrightarrow[\mathsf{c}]{}{}^*_S t$.*

Proof (Sketch). The *if* and *only-if* parts can be proved by induction on the lexicographic product (k, n) of $\xrightarrow[\mathsf{c}]{}{}^n_{(k),S}$ and $\xrightarrow[\mathsf{c}]{}{}^n_{(k),R}$, respectively (see [25]). ◀

▶ **Lemma 3.6.** *Let R be a constructor TRS, s be a term, and t be a constructor term of R. Then, $s \xrightarrow[\mathsf{c}]{}{}^*_R t$ iff $s \xrightarrow[\mathsf{c}]{}{}^*_{\mathcal{N}(R)} t$.*

Proof (Sketch). This lemma is a direct consequence of Lemma 3.5. ◀

For the sake of readability, we assume w.l.o.g. that a constructor TRS contains only the rewrite rules usable for the computation of the main function it defines. Thus, the analysis used in [20, 24, 23] to collect which functions are inverted is not necessary and we simply invert all the functions of the TRS. Moreover, we assume that the main function is not tail recursive since the new approach for tail recursive functions requires at least one initial call to the tail recursive function (as it happens in practice).

▶ **Definition 3.7** (inversion)**.** Let R be a constructor TRS such that its main function is not tail recursive. Then, the transformation $\mathcal{I}nv$ is defined as follows:

TRS inversion $\quad \mathcal{I}nv(R) = \bigcup_{l \to r \Leftarrow c \in \mathcal{N}(R)} \mathcal{I}nv_{\mathsf{rule}}(l \to r \Leftarrow c)$

Rule inversion $\mathcal{I}nv_{\mathsf{rule}}$:

(i) If f is not tail recursive, then

$$\begin{aligned}&\mathcal{I}nv_{\mathsf{rule}}(\ f(u_1,\cdots,u_n)\to r \Leftarrow s_1 \twoheadrightarrow t_1;\cdots;s_k\twoheadrightarrow t_k\ )\\&\quad=\{\ f^{-1}(r)\to \mathsf{tp}_n(u_1,\cdots,u_n)\Leftarrow \mathcal{I}nv_{\mathsf{c}}(s_k\twoheadrightarrow t_k);\cdots;\mathcal{I}nv_{\mathsf{c}}(s_1\twoheadrightarrow t_1)\ \}\end{aligned}$$

(ii) If f is tail recursive and the corresponding rule is a non-recursive f-rule, then

$$\begin{aligned}&\mathcal{I}nv_{\mathsf{rule}}(\ f(u_1,\cdots,u_n)\to r\ )=\\&\quad\{\ \widetilde{f}(w_1,\cdots,w_n)\to \mathsf{tp}_n(w_1,\cdots,w_n)\mid f(w_1,\cdots,w_n)\text{ is an initial call of }f\text{ in }R\ \}\end{aligned}$$

(iii) Otherwise (i.e., if f is tail recursive and the corresponding rule is a recursive f-rule),

$$\begin{aligned}&\mathcal{I}nv_{\mathsf{rule}}(\ f(u_1,\cdots,u_n)\to f(r_1,\cdots,r_n)\Leftarrow s_1\twoheadrightarrow t_1;\cdots;s_k\twoheadrightarrow t_k\ )\\&\quad=\{\ \widetilde{f}(r_1,\cdots,r_n)\to\widetilde{f}(u_1,\cdots,u_n)\Leftarrow\mathcal{I}nv_{\mathsf{c}}(s_k\twoheadrightarrow t_k);\cdots;\mathcal{I}nv_{\mathsf{c}}(s_1\twoheadrightarrow t_1)\ \}\end{aligned}$$

Condition inversion $\mathcal{I}nv_{\mathsf{c}}$:

(i) If f is not tail recursive, then

$$\mathcal{I}nv_{\mathsf{c}}(f(u_1,\cdots,u_n)\twoheadrightarrow x)=f^{-1}(x)\twoheadrightarrow \mathsf{tp}_n(u_1,\cdots,u_n)$$

(ii) Otherwise,

$$\mathcal{I}nv_{\mathsf{c}}(f(u_1,\cdots,u_n)\twoheadrightarrow x)=x\twoheadrightarrow r;\widetilde{f}(v_1,\cdots,v_n)\twoheadrightarrow \mathsf{tp}_n(u_1,\cdots,u_n)$$

where $f(v_1,\cdots,v_n)\to r$ is a renamed variant of a non-recursive f-rule such that the variables in $v_1,\cdots,v_n$ are fresh.

Each inversion according to Definition 3.7 proceeds as follows:

- Rule inversion (i) and Condition inversion (i) process non-tail-recursive functions similarly to the previous inversion methods of [20, 23].
- Rule inversion (ii) generates a non-recursive rule of $\widetilde{f}$ for a tail recursive function f. In this case, the input rule $f(u_1,\cdots,u_n)\to r$ is not used, except for the root defined symbol f of the left-hand side. This case is only a trigger to generate non-recursive rules for $\widetilde{f}$, while the discarded terms $u_1,\cdots,u_n,r$ are consumed via $f(v_1,\cdots,v_n)\to r$ in Condition inversion (ii).
- Rule inversion (iii), the main part of our new approach, inverts a recursive rule of a tail recursive function f following the idea described above.
- Condition inversion (ii) inverts the condition of a tail recursive function f into the condition that is an initial call of $\widetilde{f}$, adding a condition that connects x and r.

We will show later how to remove the assumption that non-recursive rules of tail recursive functions are unique (see Example 3.13 below). Returned tuples of $\widetilde{f}$ can sometimes be further optimized (e.g., by eliminating trivial elements such as nil in $\mathsf{tp}_2(xs,\mathsf{nil})$ returned by $\widetilde{\mathsf{rev}}$). However, when there are several initial calls, such optimizations might destroy the correctness of the transformation and should be carefully considered.

▶ **Example 3.8.** Consider the TRS R_{reverse} from Example 3.1 again. Normalization only changes the first rule as follows:

$$\mathcal{N}(\mathsf{reverse}(xs)\to\mathsf{rev}(xs,\mathsf{nil}))=\mathsf{reverse}(xs)\to ys\Leftarrow\mathsf{rev}(xs,\mathsf{nil})\twoheadrightarrow ys$$

Then, each rule in $\mathcal{N}(R_{\mathsf{reverse}})$ is inverted as follows:

$$\begin{aligned}&\mathcal{I}nv_{\mathsf{rule}}(\mathsf{reverse}(xs)\to ys\Leftarrow\mathsf{rev}(xs,\mathsf{nil})\twoheadrightarrow ys)\\&\qquad=\{\ \mathsf{reverse}^{-1}(ys)\to\mathsf{tp}_1(xs)\Leftarrow\mathcal{I}nv_{\mathsf{c}}(\mathsf{rev}(xs,\mathsf{nil})\twoheadrightarrow ys)\ \}\\&\qquad=\{\ \mathsf{reverse}^{-1}(ys)\to\mathsf{tp}_1(xs)\Leftarrow ys\twoheadrightarrow zs;\ \widetilde{\mathsf{rev}}(\mathsf{nil},zs)\twoheadrightarrow\mathsf{tp}_2(xs,\mathsf{nil})\ \}\\&\mathcal{I}nv_{\mathsf{rule}}(\mathsf{rev}(\mathsf{nil},ws)\to ws)=\{\ \widetilde{\mathsf{rev}}(xs,\mathsf{nil})\twoheadrightarrow\mathsf{tp}_2(xs,\mathsf{nil})\ \}\\&\mathcal{I}nv_{\mathsf{rule}}(\mathsf{rev}(\mathsf{cons}(x,xs),ys)\to\mathsf{rev}(xs,\mathsf{cons}(x,ys)))\\&\qquad=\{\ \widetilde{\mathsf{rev}}(xs,\mathsf{cons}(x,ys))\to\widetilde{\mathsf{rev}}(\mathsf{cons}(x,xs),ys)\ \}\end{aligned}$$

Thus, R_{reverse} is inverted as follows:

$$\mathcal{I}nv(R_{\mathsf{reverse}}) = \left\{ \begin{array}{r} \mathsf{reverse}^{-1}(ys) \to \mathsf{tp}_1(xs) \Leftarrow ys \twoheadrightarrow zs;\ \widetilde{\mathsf{rev}}(\mathsf{nil}, zs) \twoheadrightarrow \mathsf{tp}_2(xs, \mathsf{nil}) \\ \widetilde{\mathsf{rev}}(xs, \mathsf{nil}) \to \mathsf{tp}_2(xs, \mathsf{nil}) \\ \widetilde{\mathsf{rev}}(xs, \mathsf{cons}(x, ys)) \to \widetilde{\mathsf{rev}}(\mathsf{cons}(x, xs), ys) \end{array} \right\}$$

$\mathcal{I}nv(R_{\mathsf{reverse}})$ is non-overlapping and operationally terminating.

Now, we show the correctness of $\mathcal{I}nv$.

▶ **Lemma 3.9** (completeness). *Let R be a* tp*-free constructor TRS over a signature $\mathcal{F}$, f be a defined symbol of R, and c be $\mathcal{I}nv_{\mathsf{c}}(f(u_1, \cdots, u_n) \twoheadrightarrow x)$ that is needed for generating $\mathcal{I}nv(R)$, where x is a fresh variable. For any constructor substitution σ, if $f(u_1, \cdots, u_n)\sigma \xrightarrow{*}_{\mathsf{c}, \mathcal{N}(R)} x\sigma$, then there exists an extended constructor substitution σ' of σ such that $s\sigma' \xrightarrow{*}_{\mathsf{c}, \mathcal{I}nv(R)} t\sigma'$ for each condition $s \twoheadrightarrow t$ in c.*

Proof (Sketch). This can be proved by induction on the length n of $\xrightarrow{n}_{\mathsf{c}, \mathcal{N}(R)}$ (see [25]). ◀

▶ **Lemma 3.10** (soundness). *Let R be a* tp*-free constructor TRS over a signature $\mathcal{F}$, f be a defined symbol of R, and c be $\mathcal{I}nv_{\mathsf{c}}(f(u_1, \cdots, u_n) \twoheadrightarrow x)$ that is needed for generating $\mathcal{I}nv(R)$, where x is a fresh variable. For any constructor substitution σ, if $s\sigma \xrightarrow{*}_{\mathsf{c}, \mathcal{I}nv(R)} t\sigma$ for each condition $s \twoheadrightarrow t$ in c, then $f(u_1, \cdots, u_n)\sigma \xrightarrow{*}_{\mathsf{c}, \mathcal{N}(R)} x\sigma$.*

Proof (Sketch). This lemma can be proved by induction on the lexicographic product (m, n) where m and n are the maximum integers among (m', n') of $s\sigma \xrightarrow{n'}_{\mathsf{c}(m'), \mathcal{I}nv(R)} t\sigma$ (see [25]). ◀

▶ **Theorem 3.11** (correctness). *Let R be a* tp*-free constructor TRS over a signature $\mathcal{F}$, and f be a defined symbol of R. Then, all of the following hold:*

- *If f is not tail recursive, then f^{-1} in $\mathcal{I}nv(R)$ is a full inverse of f.*
- *If f is tail recursive, then $\widetilde{f}$ in $\mathcal{I}nv(R)$ is a tail recursive inverse of f.*

Proof (Sketch). This theorem follows from Lemmas 3.6, 3.9 and 3.10. ◀

Now, we show two simple optimizations that can be applied after $\mathcal{I}nv$:

1. A rule $\rho : l \to r \Leftarrow s_1 \twoheadrightarrow t_1; \cdots, p_1 \twoheadrightarrow p_2; \cdots; s_k \twoheadrightarrow t_k$ can be replaced by $l\sigma \to r\sigma \Leftarrow s_1\sigma \twoheadrightarrow t_1\sigma; \cdots; s_k\sigma \twoheadrightarrow t_k\sigma$ if p_1 and p_2 are unifiable constructor terms, where σ is a most general unifier of p_1 and p_2 such that $\mathcal{VR}an(\sigma) \cap \mathcal{V}ar(\rho) = \emptyset$.
2. The rule ρ above can be removed from the rewrite system if p_1 and p_2 are not unifiable.

For a constructor TRS R, the CTRS obtained by applying the above two optimizations to $\mathcal{I}nv(R)$ as much as possible is denoted by $\mathcal{I}nv_{\mathsf{opt}}(R)$.

▶ **Example 3.12.** $\mathcal{I}nv(R_{\mathsf{reverse}})$ in Example 3.8 is optimized as follows:

$$\mathcal{I}nv_{\mathsf{opt}}(R_{\mathsf{reverse}}) = \left\{ \begin{array}{r} \mathsf{reverse}^{-1}(ys) \to \mathsf{tp}_1(xs) \Leftarrow \widetilde{\mathsf{rev}}(\mathsf{nil}, ys) \twoheadrightarrow \mathsf{tp}_2(xs, \mathsf{nil}) \\ \widetilde{\mathsf{rev}}(xs, \mathsf{nil}) \to \mathsf{tp}_2(xs, \mathsf{nil}) \\ \widetilde{\mathsf{rev}}(xs, \mathsf{cons}(x, ys)) \to \widetilde{\mathsf{rev}}(\mathsf{cons}(x, xs), ys) \end{array} \right\}$$

As mentioned before, the technique for the elimination of non-determinism of [6, 7, 11] is able to produce a similar result starting from the full inverse system obtained by the old approach (i.e., the system shown in Example 3.1). However, their technique is very sensitive to the structure of the program and does not always produce non-overlapping deterministic programs (e.g., the next example cannot be inverted following the approach of [6, 7, 11]).

Our next example illustrates the application of the inversion technique when there are several initial call patterns.

▶ **Example 3.13.** Consider the following TRS $R_{\mathsf{treepaths}}$:

$$R_{\mathsf{treepaths}} = \left\{ \begin{array}{r} \mathsf{treepaths}(t) \to \mathsf{paths}(t, \mathsf{nil}, \mathsf{nil}) \\ \mathsf{paths}(\mathsf{leaf}, qs, qss) \to \mathsf{cons}(qs, qss) \\ \mathsf{paths}(\mathsf{bin}(l, r), ps, pss) \to \mathsf{paths}(l, \mathsf{cons}(0, ps), \mathsf{paths}(r, \mathsf{cons}(1, ps), pss)) \end{array} \right\}$$

The function treepaths takes a binary tree over bin and leaf as input and returns the list of paths from the root to leaves, e.g., for $\mathsf{bin}(\mathsf{leaf}, \mathsf{bin}(\mathsf{bin}(\mathsf{leaf}, \mathsf{leaf}), \mathsf{bin}(\mathsf{bin}(\mathsf{leaf}, \mathsf{leaf}), \mathsf{leaf})))$, the function treepaths returns the list corresponding to `[[0],[0,0,1],[1,0,1],[0,0,1,1],[1,0,1,1],[1,1,1]]`, where 0 and 1 indicate the left and right children, resp. The right-hand side of the third rule contains a nested function call to paths, so it is an initial call of paths. Thus, $R_{\mathsf{treepaths}}$ is inverted as follows:

$$\mathcal{I}nv_{\mathsf{opt}}(R_{\mathsf{treepaths}}) = \left\{ \begin{array}{r} \mathsf{treepaths}^{-1}(\mathsf{cons}(qs, qss)) \to \mathsf{tp}_1(t) \Leftarrow \widetilde{\mathsf{paths}}(\mathsf{leaf}, qs, qss) \twoheadrightarrow \mathsf{tp}_3(t, \mathsf{nil}, \mathsf{nil}) \\ \widetilde{\mathsf{paths}}(t, \mathsf{nil}, \mathsf{nil}) \to \mathsf{tp}_3(t, \mathsf{nil}, \mathsf{nil}) \\ \widetilde{\mathsf{paths}}(r, \mathsf{cons}(1, ps), pss) \to \mathsf{tp}_3(r, \mathsf{cons}(1, ps), pss) \\ \widetilde{\mathsf{paths}}(l, \mathsf{cons}(0, ps), \mathsf{cons}(qs, qss)) \to \widetilde{\mathsf{paths}}(\mathsf{bin}(l, r), ps, pss) \\ \Leftarrow \widetilde{\mathsf{paths}}(\mathsf{leaf}, qs, qss) \twoheadrightarrow \mathsf{tp}_3(r, \mathsf{cons}(1, ps), pss)) \end{array} \right\}$$

▶ **Example 3.14.** Consider the TRS $R_{\mathsf{unbin2}} = R_{\mathsf{ub}} \cup R_{\mathsf{inc}}$ (a slight variation of that in [6, 7, 11]) with

$$R_{\mathsf{ub}} = \left\{ \begin{array}{r} \mathsf{unbin}(u) \to \mathsf{ub}(u, \mathsf{nil}) \\ \mathsf{ub}(\mathsf{s}(\mathsf{zero}), b) \to b \\ \mathsf{ub}(\mathsf{s}(\mathsf{s}(v)), b) \to \mathsf{ub}(\mathsf{s}(v), \mathsf{inc}(b)) \end{array} \right\}; \; R_{\mathsf{inc}} = \left\{ \begin{array}{r} \mathsf{inc}(\mathsf{nil}) \to \mathsf{cons}(0, \mathsf{nil}) \\ \mathsf{inc}(\mathsf{cons}(0, xs)) \to \mathsf{cons}(1, xs) \\ \mathsf{inc}(\mathsf{cons}(1, xs)) \to \mathsf{cons}(0, \mathsf{inc}(xs)) \end{array} \right\}$$

The function unbin translates natural numbers $\mathsf{s}(\mathsf{zero})$, $\mathsf{s}(\mathsf{s}(\mathsf{zero}))$, $\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{zero})))$, $\cdots$ into the (reversed) binary numbers nil, $\mathsf{cons}(0, \mathsf{nil})$, $\mathsf{cons}(1, \mathsf{nil})$, $\mathsf{cons}(0, \mathsf{cons}(0, \mathsf{nil}))$, $\cdots$, resp., where nil represents 1 and inc is used to increment binary numbers. R_{unbin2} is inverted as follows:

$$\mathcal{I}nv_{\mathsf{opt}}(R_{\mathsf{unbin2}}) = \left\{ \begin{array}{r} \mathsf{unbin}^{-1}(b) \to \mathsf{tp}_1(u) \Leftarrow \widetilde{\mathsf{ub}}(\mathsf{s}(\mathsf{zero}), b) \twoheadrightarrow \mathsf{tp}_2(u, \mathsf{nil}) \\ \widetilde{\mathsf{ub}}(u, \mathsf{nil}) \to \mathsf{tp}_2(u, \mathsf{nil}) \\ \widetilde{\mathsf{ub}}(\mathsf{s}(v), x) \to \widetilde{\mathsf{ub}}(\mathsf{s}(\mathsf{s}(v)), b) \Leftarrow \mathsf{inc}^{-1}(x) \twoheadrightarrow \mathsf{tp}_1(b) \end{array} \right\} \cup \mathcal{I}nv_{\mathsf{opt}}(R_{\mathsf{inc}})$$

where

$$\mathcal{I}nv_{\mathsf{opt}}(R_{\mathsf{inc}}) = \left\{ \begin{array}{r} \mathsf{inc}^{-1}(\mathsf{cons}(0, \mathsf{nil})) \to \mathsf{tp}_1(\mathsf{nil}) \\ \mathsf{inc}^{-1}(\mathsf{cons}(1, xs)) \to \mathsf{tp}_1(\mathsf{cons}(0, xs)) \\ \mathsf{inc}^{-1}(\mathsf{cons}(0, ys)) \to \mathsf{tp}_1(\mathsf{cons}(1, xs)) \Leftarrow \mathsf{inc}^{-1}(ys) \twoheadrightarrow \mathsf{tp}_1(xs) \end{array} \right\}$$

The result is operationally terminating and overlapping while the method in [6, 7, 11] fails to generate an inverse of unbin due to a so-called *shift/shift* conflict. Moreover, the application of the non-determinism elimination in [6, 7, 11] to $\mathcal{I}nv_{\mathsf{opt}}(R_{\mathsf{unbin2}})$ also fails due to a shift/shift conflict.

Unfortunately, although $\mathcal{I}nv_{\mathsf{opt}}(R_{\mathsf{unbin2}})$ is operationally terminating, its rules are overlapping. Luckily, there exists a simple approach to improve the result of the inversion transformation.

If a CTRS is non-overlapping, then the application of rewrite rules to innermost redexes is deterministic. For analyzing confluence of CTRSs, the notion of *infeasible* critical pairs is

useful since overlapping rules producing infeasible critical pairs can be considered to be non-overlapping in practice [33]. However, this notion is not so helpful to determine which rules should be applied to redexes before evaluating the conditional parts. On the other hand, a bit more attentive analysis is sometimes helpful. Consider $\mathcal{I}nv_{\mathsf{opt}}(R_{\mathsf{unbin2}})$ in Example 3.14 again. The critical pairs of $\mathcal{I}nv_{\mathsf{opt}}(R_{\mathsf{unbin2}})$ are the following:

$$\begin{array}{llll}
(& \mathsf{tp}_2(\mathsf{s}(v), \mathsf{nil}), & \widetilde{\mathsf{ub}}(\mathsf{s}(\mathsf{s}(v)), b) &) \Leftarrow \mathsf{inc}^{-1}(\mathsf{nil}) \twoheadrightarrow \mathsf{tp}_1(b) \\
(& \widetilde{\mathsf{ub}}(\mathsf{s}(\mathsf{s}(v)), b), & \mathsf{tp}_2(\mathsf{s}(v), \mathsf{nil}) &) \Leftarrow \mathsf{inc}^{-1}(\mathsf{nil}) \twoheadrightarrow \mathsf{tp}_1(b) \\
(& \mathsf{tp}_1(\mathsf{nil}), & \mathsf{tp}_1(\mathsf{cons}(1, xs)) &) \Leftarrow \mathsf{inc}^{-1}(\mathsf{nil}) \twoheadrightarrow \mathsf{tp}_1(xs) \\
(& \mathsf{tp}_1(\mathsf{cons}(1, xs)), & \mathsf{tp}_1(\mathsf{nil}) &) \Leftarrow \mathsf{inc}^{-1}(\mathsf{nil}) \twoheadrightarrow \mathsf{tp}_1(xs)
\end{array}$$

Since $\mathsf{inc}^{-1}(\mathsf{nil})$ is undefined, it is easy to see that these critical pairs are infeasible and that we do not need to apply the second rule of $\widetilde{\mathsf{ub}}$ and the third rule of inc^{-1} to terms of the forms $\widetilde{\mathsf{ub}}(u, \mathsf{nil})$ and $\mathsf{inc}^{-1}(\mathsf{cons}(0, \mathsf{nil}))$, resp. This observation comes from the performance of the restricted completion in [21] that is used for transforming *unraveled* inverted TRSs into confluent ones. We call a critical pair $(u, v) \Leftarrow c$ *statically infeasible w.r.t.* $\underset{\mathsf{c}}{\rightarrow}$ if c contains a condition $f(s_1, \cdots, s_n) \twoheadrightarrow t$ such that $s_1, \cdots, s_n, t$ are constructor terms and $f(s_1, \cdots, s_n)$ is unifiable with none of the left-hand sides of f-rules. We call a CTRS *practically non-overlapping* if every critical pair of the CTRS is either trivial and at least one of the rules forming the critical pair is unconditional, or it is statically infeasible. For a practically non-overlapping constructor CTRS, we can straightforwardly give a priority between its rewrite rules in advance as *priority rewrite systems* [29] that are more faithful to functional programs.

▶ **Example 3.15.** Consider $\mathcal{I}nv_{\mathsf{opt}}(R_{\mathsf{unbin2}})$ in Example 3.14 again. All the critical pairs are statically infeasible and thus $\mathcal{I}nv_{\mathsf{opt}}(R_{\mathsf{unbin2}})$ is practically non-overlapping.

When removing the assumption that a tail recursive function has a single non-recursive rule, we extend Definition 3.7 as follows: the condition sequence $\mathcal{I}nv_{\mathsf{c}}(s_k \twoheadrightarrow t_k); \cdots; \mathcal{I}nv_{\mathsf{c}}(s_1 \twoheadrightarrow t_1)$ is replaced by $c_1; \cdots; c_k$ where $c_i \in \mathcal{I}nv_{\mathsf{c}}(s_i \twoheadrightarrow t_i)$ for all $1 \leq i \leq k$, and Condition inverse (ii) is replaced by $\mathcal{I}nv_{\mathsf{c}}(f(u_1, \cdots, u_n) \twoheadrightarrow x) = \{x \twoheadrightarrow r; \widetilde{f}(v_1, \cdots, v_n) \twoheadrightarrow \mathsf{tp}_n(u_1, \cdots, u_n) \mid f(v_1, \cdots, v_n) \to r$ is a renamed variant of a non-recursive f-rule$\}$. Our next example illustrates the application of this extended inversion technique to tail recursive functions having two non-recursive rules.

▶ **Example 3.16** (reverse2)**.** Consider the following variant of reverse in Example 3.1:

$$R_{\mathsf{reverse2}} = \left\{ \begin{array}{r}
\mathsf{reverse}(xs) \to \mathsf{rev}(xs, \mathsf{nil}) \\
\mathsf{rev}(\mathsf{nil}, ys) \to ys \\
\mathsf{rev}(\mathsf{cons}(x, \mathsf{nil}), ys) \to \mathsf{snoc}(ys, x) \\
\mathsf{rev}(\mathsf{cons}(x_1, \mathsf{cons}(x_2, xs)), ys) \to \mathsf{rev}(xs, \mathsf{cons}(x_2, \mathsf{cons}(x_1, ys))) \\
\mathsf{snoc}(\mathsf{nil}, y) \to \mathsf{cons}(y, \mathsf{nil}) \\
\mathsf{snoc}(\mathsf{cons}(x, xs), y) \to \mathsf{cons}(x, \mathsf{snoc}(xs, y))
\end{array} \right\}$$

Applying the extended inversion, R_{reverse2} is inverted as follows:

$$\mathcal{I}nv_{\mathsf{opt}}(R_{\mathsf{reverse2}}) = \left\{ \begin{array}{r}
\mathsf{reverse}^{-1}(ys) \to \mathsf{tp}_1(xs) \Leftarrow \widetilde{\mathsf{rev}}(\mathsf{nil}, ys) \twoheadrightarrow \mathsf{tp}_2(xs, \mathsf{nil}) \\
\mathsf{reverse}^{-1}(ys) \to \mathsf{tp}_1(xs) \Leftarrow \mathsf{snoc}^{-1}(ys) \twoheadrightarrow \mathsf{tp}_2(zs, z); \\
\qquad \widetilde{\mathsf{rev}}(\mathsf{cons}(z, \mathsf{nil}), zs) \twoheadrightarrow \mathsf{tp}_2(xs, \mathsf{nil}) \\
\widetilde{\mathsf{rev}}(xs, \mathsf{nil}) \to \mathsf{tp}_2(xs, \mathsf{nil}) \\
\widetilde{\mathsf{rev}}(xs, \mathsf{cons}(x_2, \mathsf{cons}(x_1, ys))) \to \widetilde{\mathsf{rev}}(\mathsf{cons}(x_1, \mathsf{cons}(x_2, xs)), ys) \\
\mathsf{snoc}^{-1}(\mathsf{cons}(y, \mathsf{nil})) \to \mathsf{tp}_2(\mathsf{nil}, y) \\
\mathsf{snoc}^{-1}(\mathsf{cons}(x, ys)) \to \mathsf{tp}_2(\mathsf{cons}(x, xs), y) \Leftarrow \mathsf{snoc}^{-1}(ys) \twoheadrightarrow \mathsf{tp}_2(xs, y)
\end{array} \right\}$$

The result is operationally terminating but not practically non-overlapping while the method in [6, 7, 11] fails to generate an inverse of reverse2 due to a shift/shift conflict. Unfortunately, the application of the non-determinism elimination in [6, 7, 11] to $\mathcal{I}nv_{\mathsf{opt}}(R_{\mathsf{reverse2}})$ also fails due to a shift/shift conflict.

4 Comparison with Previous Approaches

The method in this paper is a conservative extension of [20, 23] to better deal with the inversion of tail recursive functions. While the previous approach always produces non-terminating rules from the inversion of tail recursive functions, this is not always the case for the new approach, which is thus strictly better regarding the generation of terminating systems. Note that the method in this paper can invert every constructor TRS, though the resulting CTRS is not always practically non-overlapping and operationally terminating.

In the following, we show some experimental results from the evaluation of our new approach. First, we applied it to the standard 15 benchmarks from [10].[2] Ten of the benchmarks are non-tail-recursive functions without tail-recursive rules. For these benchmarks, our method generates the same inverse systems as the previous method [20, 24, 23] and all the results are operationally terminating and practically non-overlapping. Note that the method in [6, 7, 11] is also successful for the 10 benchmarks.

As for the remaining 5 benchmarks, three of them are tail recursive ones, and the other two are non-tail recursive functions containing tail-recursive rules. Table 1 summarizes the results on the 3 tail recursive functions, reverse, unbin and treepaths, and also contains the results on other tail recursive functions: unbin2 (Example 3.14) and reverse2 (Example 3.16). Moreover, the lower half of Table 1 summarizes the results of applying the inversion to inverses obtained from the first 3 benchmarks: $\mathcal{I}nv_{\mathsf{opt}}(\mathsf{reverse})$, $\mathcal{I}nv_{\mathsf{opt}}(\mathsf{unbin})$ and $\mathcal{I}nv_{\mathsf{opt}}(\mathsf{treepaths})$ are the systems obtained by applying $\mathcal{I}nv_{\mathsf{opt}}$ to reverse, unbin and treepaths, resp., and lrinv-reverse, lrinv-unbin and lrinv-treepaths are obtained by applying the method [6, 7, 11] to reverse, unbin and treepaths, respectively. Note that the benchmarks in the second half are in principle out of scope of our approach, but we applied our inversion method to them extending it straightforwardly. Operational termination of the resulting systems obtained by our method was proved by the termination tool VMTL [30].

By lack of space, we did not show a transformation of non-tail-recursive functions to equivalent tail-recursive ones, that is an extension of the idea of *continuation passing style* to the first-order setting. However, after transforming the remaining two benchmarks that are not tail-recursive but contain tail-recursive rules into tail-recursive ones, our method succeeds in generating operationally terminating and practically non-overlapping inverse systems. Thus, our method is successful for all the benchmarks shown in [6, 7, 11]. A similar transformation is called *context-moving* [5]. However, this transformation is sound when the contexts to be moved satisfy a property like associativity or commutativity. Thus, the context-moving transformation is not applicable to arbitrary non-tail-recursive functions.

There exist some examples that the method in [6, 7, 11] fails to invert but our approach succeeds, e.g., unbin2. On the other hand, we have never found an example in which the non-determinism problem cannot be solved by our approach but can be solved by the method in [6, 7, 11], though it may exist. To summarize, a promising strategy for program inversion

[2] Unfortunately, the site shown in [10] is not accessible now. Some of the benchmarks can be found in [6, 7, 11]. All the benchmarks are reviewed in [21] and also available from the following URL for the implementation of our method: `http://www.trs.cm.is.nagoya-u.ac.jp/repius/`.

Table 1 Comparison with the previous approaches [20, 24] and [6, 7, 11]: "SN", "NOV" and "PNOV" mean that the inverse is "operationally terminating", "non-overlapping" and "practically non-overlapping", respectively.

	inverse by [20, 24, 23]	inverse by [6, 7, 11]	inverse by this paper
benchmark	SN?/NOV?/PNOV?	SN?/NOV?	SN?/NOV?/PNOV?
reverse [10, 6, 7, 11]	no/no/no	**yes**/**yes**	**yes**/**yes**/**yes**
unbin [10, 6, 7, 11]	no/no/no	**yes**/**yes**	**yes**/no/**yes**
treepaths [10, 6, 7, 11]	no/no/no	**yes**/**yes**	**yes**/**yes**/**yes**
unbin2 (Example 3.14)	no/no/no	fail (no output)	**yes**/no/**yes**
reverse2 (Example 3.16)	no/no/no	fail (no output)	**yes**/no/no
$\mathcal{I}nv_{\mathsf{opt}}$(reverse)	no/no/no	**yes**/**yes**	**yes**/**yes**/**yes**
$\mathcal{I}nv_{\mathsf{opt}}$(unbin)	no/no/no	**yes**/**yes**	**yes**/**yes**/**yes**
$\mathcal{I}nv_{\mathsf{opt}}$(treepaths)	no/no/no	**yes**/**yes**	**yes**/**yes**/**yes**
lrinv-reverse	no/no/no	**yes**/**yes**	**yes**/**yes**/**yes**
lrinv-unbin	no/no/no	**yes**/**yes**	**yes**/**yes**/**yes**
lrinv-treepaths	no/no/no	fail (no output)	**yes**/**yes**/**yes**

would first apply our approach and, then, the non-determinism elimination of [6, 7, 11] (appropriately extended to deal with erasing rules), when the output of our approach contains some non-determinism.

As stated before, the idea of our approach to inversion of tail recursive functions is similar to that in [18], though there the idea is only illustrated by using some examples and there is no formal transformation nor correctness proof. On the other hand, we introduced a formal definition and its correctness proof, together with an analysis on overlaps between generated rewrite rules. Moreover, our formalization can be easily extended to tail recursive functions with several initial calls and non-recursive rules.

Another related work is the method in [15], though it is more related to inverse computation than to program inversion. Nevertheless, we note that this method is only applicable to linear functions and it is not guaranteed the termination of the produced programs for any input (only for the inputs that are original outputs). Moreover, tail recursive functions are out of scope of this paper.

5 Conclusion

In this paper, we have proposed the notion of tail recursive inverses and have introduced an appropriate approach to program inversion which extends the previous technique from [20, 23] in order to deal with systems containing tail recursive functions. As mentioned before, for the sake of readability, we assumed the following restrictions in this paper: the original TRS is unconditional, non-erasing and tp-free; and tail recursive functions are self-recursive. However, it would not be difficult to remove these assumptions (and, indeed, they are not required in the implemented method). Moreover, it would be easy to combine the new approach to the inversion of tail recursive functions with the method of [20, 24] for partial inversion, so there is ample room for improving and extending our approach.

As for future work, we plan to develop a method for the elimination of non-determinism in inverted rules since there are examples, such as reverse2 in Example 3.16. Another promising direction for future work is the search of sufficient conditions for tail recursive functions so that the computed inverses with our technique are operationally terminating.

Acknowledgements

We thank the anonymous reviewers for their useful comments to improve this paper. This work has been partially supported by MEXT KAKENHI #21700011 and the Spanish *Ministerio de Ciencia e Innovación* under grant TIN2008-06622-C03-02. Part of this research was done while the first author was visiting the MiST group at the *Universitat Politècnica de València* as part of an Institutional Program for Young Researcher Overseas Visits. The first author is grateful to the members of the MiST and ELP groups.

References

1 S. M. Abramov and R. Glück. The universal resolving algorithm and its correctness: inverse computation in a functional language. *Science of Computer Programming*, 43(2-3):193–229, 2002.

2 J. M. Almendros-Jiménez and G. Vidal. Automatic partial inversion of inductively sequential functions. In *Proc. of the 18th Int'l Symposium on Implementation and Application of Functional Languages (Revised Selected Papers)*, volume 4449 of *Lecture Notes in Computer Science*, pp. 253–270, Springer, 2007.

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

4 N. Dershowitz and S. Mitra. Jeopardy. In *Proc. of the 10th International Conference on Rewriting Techniques and Applications*, volume 1631 of *Lecture Notes in Computer Science*, pp. 16–29, Trento, Italy, July 1999.

5 J. Giesl. Context-moving transformations for function verification. In *Selected Papers of the 9th International Workshop on Logic Programming Synthesis and Transformation*, volume 1817 of *Lecture Notes in Computer Science*, pp. 293–312, Springer, 2000.

6 R. Glück and M. Kawabe. A program inverter for a functional language with equality and constructors. In *Proc. of the 1st Asian Symposium on Programming Languages and Systems*, volume 2895 of *Lecture Notes in Computer Science*, pp. 246–264, Springer, 2003.

7 R. Glück and M. Kawabe. A method for automatic program inversion based on LR(0) parsing. *Fundamenta Informaticae*, 66(4):367–395, 2005.

8 B. Gramlich. On the (non-)existence of least fixed points in conditional equational logic and conditional rewriting. In *Proc. of the 2nd International Workshop on Fixed Points in Computer Science – Extended Abstracts*, pp. 38–40, Paris, France, July 2000.

9 P. G. Harrison. Function inversion. In *Proc. of IFIP TC2 Workshop on Partial Evaluation and Mixed Computation*, pp. 153–166, North-Holland, 1988.

10 M. Kawabe and Y. Futamura. Case studies with an automatic program inversion system. In *Proc. of the 21st Conference of Japan Society for Software Science and Technology*, number 6C-3, 5 pages, 2004.

11 M. Kawabe and R. Glück. The program inverter LRinv and its structure. In *Proc. of the 7th International Symposium on Practical Aspects of Declarative Languages*, volume 3350 of *Lecture Notes in Computer Science*, pp. 219–234, Springer, Jan. 2005.

12 H. Khoshnevisan and K. M. Sephton. InvX: An automatic function inverter. In *Proc. of the 3rd International Conference on Rewriting Techniques and Applications*, volume 355 of *Lecture Notes in Computer Science*, pp. 564–568, Springer, 1989.

13 S. Lucas, C. Marché, and J. Meseguer. Operational termination of conditional term rewriting systems. *Information Processing Letters*, 95(4):446–453, 2005.

14 M. Marchiori. Unravelings and ultra-properties. In *Proc. of the 5th International Conference on Algebraic and Logic Programming*, volume 1139 of *Lecture Notes in Computer Science*, pp. 107–121, Springer, 1996.

15 K. Matsuda, S.-C. Mu, Z. Hu, and M. Takeichi. A grammar-based approach to invertible programs. In *Proc. of the 19 the European Symposium on Programming*, volume 6012 of *Lecture Notes in Computer Science*, pp. 448–467, Springer, 2010.

16 J. McCarthy. The inversion of functions defined by Turing machines. In *Automata Studies*, pp. 177–181. Princeton University Press, 1956.

17 T. Æ. Mogensen. Semi-inversion of guarded equations. In *Proc. of the 4th International Conference on Generative Programming and Component Engineering*, volume 3676 of *Lecture Notes in Computer Science*, pp. 189–204, Springer, 2005.

18 T. Æ. Mogensen. Report on an implementation of a semi-inverter. In *Proc. of the 6th International Andrei Ershov Memorial Conference on Perspectives of Systems Informatics*, volume 4378 of *Lecture Notes in Computer Science*, pp. 322–334, Springer, 2006.

19 T. Æ. Mogensen. Semi-inversion of functional parameters. In *Proc. of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pp. 21–29, New York, NY, USA, 2008. ACM Press.

20 N. Nishida. *Transformational Approach to Inverse Computation in Term Rewriting*. Doctor thesis, Graduate School of Engineering, Nagoya University, Nagoya, Japan, Jan. 2004.

21 N. Nishida and M. Sakai. Completion after program inversion of injective functions. In *Proc. of the 8th International Workshop on Reduction Strategies in Rewriting and Programming*, Volume 237 of *Electronic Notes in Theoretical Computer Science*, pp. 39–56, Apr. 2009.

22 N. Nishida, M. Sakai, and T. Sakabe. Generation of inverse term rewriting systems for pure treeless functions. In *Proc. of the International Workshop on Rewriting in Proof and Computation*, pp. 188–198, Sendai, Japan, Oct. 2001.

23 N. Nishida, M. Sakai, and T. Sakabe. Generation of inverse computation programs of constructor term rewriting systems. *IEICE Transactions on Information and Systems*, J88-D-I(8):1171–1183, Aug. 2005 (in Japanese).

24 N. Nishida, M. Sakai, and T. Sakabe. Partial inversion of constructor term rewriting systems. In *Proc. of the 16th Int'l Conference on Rewriting Techniques and Applications*, volume 3467 of *Lecture Notes in Computer Science*, pp. 264–278, Springer, Apr. 2005.

25 N. Nishida and G. Vidal. Program inversion for tail recursive functions. The full version of this paper, available from `http://www.trs.cm.is.nagoya-u.ac.jp/~nishida/rta11/`.

26 E. Ohlebusch. *Advanced Topics in Term Rewriting*. Springer-Verlag, April 2002.

27 A. Romanenko. The generation of inverse functions in Refal. In *Proc. of IFIP TC2 Workshop on Partial Evaluation and Mixed Computation*, pp. 427–444, North-Holland, 1988.

28 A. Romanenko. Inversion and metacomputation. In *Proc. of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, volume 26 of *SIGPLAN Notices*, pp. 12–22, ACM Press, Sept. 1991.

29 M. Sakai and Y. Toyama. Semantics and strong sequentiality of priority term rewriting systems. *Theoretical Computer Science*, 208(1-2):87–110, 1998.

30 F. Schernhammer and B. Gramlich. VMTL—a modular termination laboratory. In *Proc. of the 20th International Conference on Rewriting Techniques and Applications*, volume 5595 of *Lecture Notes in Computer Science*, pp. 285–294, Springer, 2009.

31 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):52 pages, Oct. 2009.

32 J. P. Secher and M. H. Sørensen. From checking to inference via driving and dag grammars. In *Proc. of the 2002 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, volume 37 of *SIGPLAN Notices*, pp. 41–51, Portland, Oregon, USA, March 2002.

33 Y. Takahashi, M. Sakai, and Y. Toyama. On the confluence property of conditional term rewriting systems. *IEICE Transactions*, J79-D-I(11):897–902, 1996 (in Japanese).

Refinement Types as Higher-Order Dependency Pairs

Cody Roux[1]

1 **INRIA - Lorraine**
615 r. du Jardin Botanique, 54600 Villers-lès-Nancy, France

Abstract

Refinement types are a well-studied manner of performing in-depth analysis on functional programs. The dependency pair method is a very powerful method used to prove termination of rewrite systems; however its extension to higher-order rewrite systems is still the subject of active research. We observe that a variant of refinement types allows us to express a form of higher-order dependency pair method: from the rewrite system labeled with typing information, we build a *type-level approximated dependency graph*, and describe a type level *embedding preorder*. We describe a syntactic termination criterion involving the graph and the preorder, which generalizes the *simple projection criterion* of Middeldorp and Hirokawa [21], and prove our main result: if the graph passes the criterion, then every well-typed term is strongly normalizing.

Keywords and phrases Dependency Pairs, Higher-Order, Refinement Types

Digital Object Identifier 10.4230/LIPIcs.RTA.2011.299

Category Regular Research Paper

1 Introduction

Types are used to perform static analysis on programs. Various type systems have been developed to infer information about termination, run-time complexity, or the presence of uncaught exceptions.

We are interested in one such development, namely *dependent types* [30, 13]. Dependent types explicitly allow "object level" terms to appear in the types, and may be used to fully specify (extensional) program behavior using the so called *Curry-Howard isomorphism*. We are particularly interested here in *refinement types* [36, 17]. For a given base type B and a property P on programs, we may form a type R which is a *refinement of* B and which is intuitively given the semantics:

$$R = \{t : B \mid P(t)\}.$$

Programing languages based on dependent type systems have the reputation of being unwieldy, due to the perceived weight of proof obligations in heavily specified types. The field of dependently typed programing can be seen as a quest to find the compromise between expressivity of types and ease of use for the programmer. In this paper we propose a type system which we believe achieves such a compromise for a termination analysis based on the shape of constructor terms.

Dependency pairs are a highly successful technique for proving termination of first-order rewrite systems [4]. However, it is difficult to apply the method to higher-order rewrite systems. Indeed, the data-flow of such systems is significantly different than that of first-order ones. Let us examine the rewrite rule:

$$f\ (S\ x) \to (\lambda y.f\ y)\ x$$

22nd International Conference on Rewriting Techniques and Applications.
Editor: M. Schmidt-Schauß; pp. 299–312
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Which can for example be written in the higher-order rewriting framework of Jouannaud & Okada [26]. The termination of well-typed terms under this rewrite system combined with β-reduction cannot be inferred by simply looking at the left-hand side $f\ (S\ x)$ and the recursive call $f\ y$ in the right hand side as it could be in first-order rewriting. Here we need to infer that the variable y can only be instantiated by a subterm of $S\ x$. This can be done using dependent types, using a framework called *size-based termination* or sometimes *type-based termination* [22, 1, 5, 7, 10].

The dependency pair method rests on the examination of the aptly-named *dependency pairs*, which correspond to left-hand sides of rules and function calls with their arguments in the right-hand side of the rules. For instance with a rule

$$f(c(x,y),z) \to g(f(x,y))$$

We would have two dependency pairs, the pair $f(c(x,y),z) \to f(x,y)$ and the pair $f(c(x,y),z) \to g(f(x,y))$ (in the case that g is a defined symbol).

We can then define a *chain* to be a pair (θ,ϕ) of substitutions, and a couple $(t_1 \to u_1, t_2 \to u_2)$ of dependency pairs such that $u_1\theta$ reduces to $t_2\phi$. We may connect chains in an intuitive manner, and the fundamental theorem of dependency pairs may be stated: *a (first-order) rewrite system is terminating if and only if there are no infinite chains.* See also the original article [4] for details.

To prove that no infinite chains exist, one wants to work with the *dependency graph*: the graph built using the dependency pairs as nodes and with a vertex between $N_1 = t_1 \to u_1$ and $N_2 = t_2 \to u_2$ if there exist θ and ϕ such that $(\theta,\phi),(N_1,N_2)$ form a chain. It is then shown that if the system is finite, then it is sufficient to consider only the cycles in this graph and prove that they may not lead to infinite chains [18]. It is known that in general computing the dependency graph is undecidable (this is the *unification modulo rewriting* problem, see *e.g.* Jouannaud *et al* . [25]), so in practice we compute an approximation (or estimation) of the graph that is *conservative*: all edges in the dependency graph are sure to appear in the approximated graph. One common (see for instance Giesl [20]) and reasonable approximation is to perform ordinary unification on non-defined symbols (that is, symbols that are not at the head of a left-hand side), while replacing each subterm headed by a defined symbol by a fresh variable, ensuring that it may unify with any other term.

In this article, we show that the dependency pair technique with the approximated dependency graph can be modeled using a form of refinement types containing *patterns* which denote sets of possible values to which a term reduces. The syntax and type system are described in section 2. These type-patterns must be explicitly abstracted and applied, a choice that allows us to have very simple type inference. We then use these types to build a notion of type-based dependency pair for higher-order rewrite rules, as well as an approximated dependency graph which corresponds to the estimation described above. We describe an order on the type annotations, that essentially captures the subterm ordering, and use this order to express a *decrease condition* along cycles in the approximated dependency graph. In section 3 we describe a suitable generalization of the *simple projection criterion* first described by Middeldorp and Hirokawa [21]: if in every *strongly connected component* of the graph and every cycle in the component, the decrease condition holds, then every well-typed term is strongly normalizing under combination of the rewrite rules and β-reduction. The actual operational semantics are defined not on the terms themselves, but on *erased terms* in which we remove the explicit type information. Section 4 concludes with a comparison with other approaches to higher-order dependency pairs and possible extensions of our criterion.

2 Syntax and Typing Rules

The language we consider is simply a variant of the λ-calculus with constants. For simplicity we only consider the datatype of binary (unlabeled) trees. The development may be generalized without difficulty to other first-order datatypes, *i.e.* types whose constructors do not have higher-order recursive arguments. We define the syntax of *patterns*

$$p, q \in \mathcal{P} \coloneqq \alpha \mid \_ \mid \bot \mid \text{leaf} \mid \text{node}(p, q)$$

With $\alpha \in \mathcal{V}$ a set of *pattern variables*, and _ is called *wildcard*. Patterns appear in types to describe possible reducts of terms. We define the set of types:

$$T, U \in \mathcal{T} \coloneqq \boldsymbol{B}(p) \mid T \to U \mid \forall \alpha.T$$

An *atomic type* is a type of the form $\boldsymbol{B}(p)$. The set of terms of our language is defined by:

$$t, u \in \mathcal{T}rm \coloneqq x \mid f \mid t\ u \mid t\ p \mid \lambda x\!:\!T.t \mid \lambda \alpha.t \mid \text{Leaf} \mid \text{Node}$$

With $x \in \mathcal{X}$ a set of term variables, $f \in \Sigma$ is a set of *defined function symbols* and $\alpha \in \mathcal{V}$.

A *constructor* is either Leaf or Node. A *context* is a list of judgements $x\!:\!T$ with $x \in \mathcal{X}$ and $T \in \mathcal{T}$, with each variable appearing only once. Notice that application and abstraction of patterns is explicit.

Intuitively, $\boldsymbol{B}(p)$ denotes the set of terms for which *every* reduct in normal form *matches* the pattern p. For instance, any binary tree t is in the semantics of $\boldsymbol{B}(\_)$, only binary trees that reduce to Node $t_1\ t_2$ for some binary trees t_1 and t_2 are in $\boldsymbol{B}(\text{node}(\_, \_))$, and only terms that *never* reduce to a constructor are in $\boldsymbol{B}(\bot)$. Our operational semantics is defined by rewriting, which has the following consequences, which may be surprising to a programming language theorist:

- It may be the case that a term t has several distinct normal forms. Indeed we do not require our system to be orthogonal, or even confluent (we do require it to be finitely branching though). Therefore a term is in the semantics of $\boldsymbol{B}(\text{node}(\_, \_))$ if *all* its reducts reduce to a term of the form Node $t\ u$.
- It is possible for a term to be *stuck* in the empty context, that is in normal form and not headed by a constructor or an abstraction. Therefore $\boldsymbol{B}(\bot)$ is not necessarily empty even in the empty context.

We write $\mathcal{FV}(t)$ (resp. $\mathcal{FV}(T)$, $\mathcal{FV}(\Gamma)$) for the set of free (type or term) variables in a term t (resp. a type T, a context Γ). If a term (resp. pattern) does not contain any free variables, we say that it is *closed*. We write $\forall \vec{\alpha}.T$ for $\forall \alpha_1.\forall \alpha_2 \ldots \forall \alpha_n.T$, and arrows and application are associative to the left and right respectively, as usual. A pattern variable α appears in $\boldsymbol{B}(p)$ if it appears in p. It appears *positively* in a type T if:

- $T = \boldsymbol{B}(p)$ and α appears in p
- $T = T_1 \to T_2$ and α appears positively in T_2 or negatively in T_1 (or both),

with α appearing *negatively* in T if $T = T_1 \to T_2$ and α appears negatively in T_2 or positively in T_1 (or both).

We consider a *type assignment* $\tau\!:\!\Sigma \to \mathcal{T}$, such that for each $f \in \Sigma$, there is a number k such that τ_f is of the form

$$\tau_f = \forall \alpha_1, \ldots, \alpha_k.\boldsymbol{B}(\alpha_1) \to \ldots \to \boldsymbol{B}(\alpha_k) \to T_f$$

and each α_i may appear only *positively* in T_f. In this case k is called the number of *recursive arguments*.

$$\frac{}{\Gamma, x:T, \Delta \vdash x:T}\ \textbf{ax}$$

$$\frac{\Gamma, x:T \vdash t:U}{\Gamma \vdash \lambda x:T.t:T \to U}\ \textbf{t-lam} \qquad \alpha \notin \mathcal{FV}(\Gamma)\ \frac{\Gamma \vdash t:T}{\Gamma \vdash \lambda\alpha.t:\forall\alpha.T}\ \textbf{p-lam}$$

$$\frac{}{\Gamma \vdash \mathrm{Leaf}:\boldsymbol{B}(\mathrm{leaf})}\ \textbf{leaf-intro}$$

$$\frac{}{\Gamma \vdash \mathrm{Node}:\forall\alpha\beta.\boldsymbol{B}(\alpha) \to \boldsymbol{B}(\beta) \to \boldsymbol{B}(\mathrm{node}(\alpha,\beta))}\ \textbf{node-intro}$$

$$\frac{\Gamma \vdash t:T \to U \qquad \Gamma \vdash u:T}{\Gamma \vdash t\ u:U}\ \textbf{t-app} \qquad \frac{\Gamma \vdash t:\forall\alpha.T}{\Gamma \vdash t\ p:T\{\alpha \mapsto p\}}\ \textbf{p-app}$$

$$\frac{}{\Gamma \vdash f:\tau_f}\ \textbf{symb}$$

Figure 1 Typing Rules

The positivity condition is quite similar to the one used in the usual formulation of type-based termination, see for instance Abel [2] for an in depth analysis. The typing rules are also similar to the ones for type-based termination. The typing rules of our system are given by the typing rules in Figure 1.

To these rules we add the subtyping rule:

$$\frac{\Gamma \vdash t:T \qquad T \leq U}{\Gamma \vdash t:U}\ \textbf{sub}$$

Where the subtyping relation is defined, first on patterns, then on types by:

$$\frac{}{p \ll \_} \qquad \frac{}{\bot \ll p}$$

$$\frac{}{\alpha \ll \alpha} \qquad \frac{}{\mathrm{leaf} \ll \mathrm{leaf}}$$

$$\frac{p_1 \ll q_1 \qquad p_2 \ll q_2}{\mathrm{node}(p_1,p_2) \ll \mathrm{node}(q_1,q_2)}$$

$$\frac{p \ll q}{\boldsymbol{B}(p) \leq \boldsymbol{B}(q)}$$

$$\frac{T_2 \leq T_1 \qquad U_1 \leq U_2}{T_1 \to U_1 \leq T_2 \leq U_2}$$

$$\frac{T \leq U}{\forall\alpha.T \leq \forall\alpha.U}$$

This type system is similar to the refinement types described by Freeman *et al* . [17], for a subset of the ML language. However they consider more complex refinements in which arbitrary unions are allowed (and for which type checking is undecidable!) and which does not allow one to explicitly *name* the shape of a term in the type, *i.e.* it does not allow (our version of) type-level variables. Furthermore, our system is not very distant from *generalized algebraic datatypes* as are implemented in certain Haskell extensions [33], though subtyping is not present in that framework.

$$\frac{}{\Gamma, x : \boldsymbol{B}(\alpha), \Gamma' \vdash_{\min} x : \boldsymbol{B}(\alpha)}\ \alpha \notin \Gamma, \Gamma'$$

$$\frac{}{\Gamma \vdash_{\min} \text{Leaf} : \boldsymbol{B}(\text{leaf})}$$

$$\frac{\Gamma \vdash_{\min} c_1 : \boldsymbol{B}(p_1) \qquad \Gamma \vdash_{\min} c_2 : \boldsymbol{B}(p_2)}{\Gamma \vdash_{\min} \text{Node}\ \ p_1\ p_2\ c_1\ c_2 : \boldsymbol{B}(\text{node}(p_1, p_2))}$$

$$\frac{\Gamma \vdash_{\min} c_1 : \boldsymbol{B}(p_1) \qquad \ldots \qquad \Gamma \vdash_{\min} c_k : \boldsymbol{B}(p_k)}{\Gamma \vdash_{\min} f\ p_1 \ldots p_k\ c_1 \ldots c_k : T_f \phi}\ \vec{\alpha} \notin \Gamma$$

With $\tau_f = \forall \alpha_1 \ldots \alpha_k . \boldsymbol{B}(\alpha_1) \to \ldots \to \boldsymbol{B}(\alpha_k) \to T_f$ and $\phi(\alpha_i) = p_i$ for $1 \leq i \leq k$.

Figure 2 Minimal Typing Rules

It may seem surprising that we choose to represent pattern abstraction (by $\lambda \alpha . t$), and pattern application (by $t\ p$) explicitly in our system. This choice is justified by the simplicity of type inference with explicit parameters. In the author's opinion, implicit arguments should be handled by the following schema: at the user level a language without implicit parameters; these parameters are inferred by the compiler, which type-checks a language with all parameters present. Then at run-time they are once again erased. This is exactly analogous to a Hindley-Milner type language in which System F is used as an intermediate language [31, 24]. It is also our belief that explicit parameters will allow this criterion to be more easily integrated into languages with pre-existing dependent types, *e.g.* Agda [32], Epigram [29] or Coq [15].

A *constructor term* $c \in \mathcal{C}$ is a term built following the rules:

$$c_1, c_2 \in \mathcal{C} \coloneqq x \mid \text{Leaf} \mid \text{Node}\ \ p_1\ p_2\ c_1\ c_2$$

with $x \in \mathcal{X}$.

A rewrite rule is a pair of terms (l, r) which we write $l \to r$, such that l is of the form $f\ p_1 \ldots p_n\ c_1 \ldots c_k$ with $f \in \Sigma$, $p_i \in \mathcal{P}$ and $c_i \in \mathcal{C}$, and k is the number of recursive arguments of f. We suppose that the free variables of r appear in l. Note that there is no linearity restriction on the left-hand sides of rules, and that left-hand sides may not contain any abstractions.

We suppose in addition that every function symbol $g \in r$ is *fully applied* to its pattern arguments, that is if $\tau_g = \forall \alpha_1 \ldots \alpha_l . T$ then for each occurrence of g in r there are patterns $p_1, \ldots, p_l \in \mathcal{P}$ such that $g\ p_1 \ldots p_l$ appears at that position.

In the following we consider a *finite* set $\mathcal{R}$ of rewrite rules. The set $\mathcal{R}$ is *well-typed* if for each rule $l \to r \in \mathcal{R}$, there is a context Γ and a type T such that

$$\Gamma \vdash_{min} l : T$$

and

$$\Gamma \vdash r : T$$

with $\vdash_{\min}$ the *minimal typing relation* defined in Figure 2.

Notice that if $\Gamma \vdash_{\min} c_i : T$ then T is *unique*. Minimal typing is related to other work on size-based termination [11], in which it is called the *pattern condition*. Its purpose is to constrain the possible types of constructor terms in left-hand sides, so that the type gives an *exact* semantics of the matched terms. In particular, if $\Gamma \vdash_{min} x : \boldsymbol{B}(p)$, then $p = \alpha$, $x : \boldsymbol{B}(\alpha)$ and α may *not* appear in the type of any other variable. Furthermore, subtyping is forbidden, so that a constructor term of the form Node $p\ q\ t\ u$ is given a type of the form $\boldsymbol{B}(\text{node}(p, q))$, and Leaf is given the type $\boldsymbol{B}(\text{leaf})$.

We give the following theorem without proof.

▶ **Theorem 2.1.** *Type checking is* decidable*: there is a procedure which, given* Γ, t *and* T*, decides whether*

$$\Gamma \vdash t : T$$

is derivable.

It may be useful to note here that subtyping is necessary to type all but the most trivial of programs: let $f : \forall\alpha.\boldsymbol{B}(\alpha) \to \boldsymbol{B}(\_)$ be the function that computes the mirror image of a binary tree, which may be defined in our system by the rules

$$\begin{aligned} f\ \text{leaf}\ \text{Leaf} &\to \text{Leaf} \\ f\ \text{node}(\alpha,\beta)\ (\text{Node}\ \alpha\ \beta\ x\ y) &\to \text{Node}\ \_\ \_\ (f\ \beta\ y)\ (f\ \alpha\ x) \end{aligned}$$

This function is well-typed using the minimal typing rules in the context $x : \boldsymbol{B}(\alpha),\ y : \boldsymbol{B}(\beta)$, but subtyping is necessary to type the second rule, as the term $\text{Node}\ \_\ \_\ (f\ \beta\ y)\ (f\ \alpha\ x)$ has type $\text{node}(\_,\_)$ and not $\_$ which is the required return type of f.

We can then define a higher-order analogue of dependency pairs, which uses type information instead of term information.

▶ **Definition 2.2.** *Let* $\rho = f\ \vec{p}\ \vec{c} \to r$ *be a rule in* $\mathcal{R}$*, with* Γ *such that* $\Gamma \vdash_{min} f\ \vec{p}\ \vec{c} : T$*, and* $\Gamma \vdash r : T$*. The set of* type dependency pairs $DP_{\mathcal{T}}(\rho)$ *is the set*

$$\left\{ f^\sharp(p_1,\dots,p_k) \to g^\sharp(q_1,\dots,q_l) \mid \forall i, \Gamma \vdash_{\min} c_i : \boldsymbol{B}(p_i) \wedge g\ q_1 \dots q_l \text{ appears in } r \right\}$$

The set $DP_{\mathcal{T}}(\mathcal{R})$ *is defined as the union of all* $DP_{\mathcal{T}}(\rho)$*, for* $\rho \in \mathcal{R}$*, where we suppose that all variables are disjoint between dependency pairs.*

The set of higher-order dependency pairs defined above should already be seen as an abstraction of the traditional dependency pair notion (for example those defined in [4]). Indeed, due to subtyping, there may be some information lost in the types, if for instance the wildcard pattern is used. As an example, if f, g and h all have type $\forall\alpha.\boldsymbol{B}(\alpha) \to \boldsymbol{B}(\_)$, consider the rule

$$f\ \alpha\ x \to g\ \_\ (h\ \alpha\ x)$$

The dependency pair we obtain is

$$f^\sharp(\alpha) \to g^\sharp(\_)$$

The information that g is called on the argument $h\ \alpha\ x$ is lost.

This approach can therefore be seen as a type-based manner to study an approximation of the dependency graph. Note that in the case where h is given a more precise type, like $\forall\alpha.\boldsymbol{B}(\alpha) \to \boldsymbol{B}(\text{leaf})$, which is the case if every normal form of $h\ p\ t$ is either neutral or Leaf, we have a more precise approximation.

Note that in the following definition, a dependency pair can not be easily seen as a rule of the system itself, though it may be seen as a first-order rewrite rule which operates on "type level" function symbols and constructors.

▶ **Definition 2.3.** *Let p and q be patterns. We define* pattern-unification *as the smallest relation verifying:*

$$\frac{}{p \bowtie \_} \qquad \frac{}{p \bowtie \alpha}$$

$$\frac{}{\text{leaf} \bowtie \text{leaf}} \qquad \frac{}{\bot \bowtie \bot} \qquad \frac{p \bowtie q}{q \bowtie p}$$

$$\frac{p_1 \bowtie q_1 \qquad p_2 \bowtie q_2}{\text{node}(p_1, p_2) \bowtie \text{node}(q_1, q_2)}$$

The standard typed dependency graph $\mathcal{G}_{\mathcal{R}}$ *is defined as the graph with*

- *As set of nodes the set $DP_{\mathcal{T}}(\mathcal{R})$.*
- *An edge from the dependency pair $t \to g^\sharp(p_1, \ldots, p_k)$ to $g^\sharp(q_1, \ldots, q_k) \to u$ if for every $1 \leq i \leq k, p_i \bowtie q_i$.*

This definition gives us an adequate higher-order notion of standard approximated dependency graph. We will now show that it is possible to give an order on the terms in the dependency pairs, which is similar to a simplification order and which will allow us to show termination of well-typed terms under the rules, if the graph satisfies an intuitive decrease criterion.

▶ **Definition 2.4.** *Given patterns p and q which do* not *contain $\_$, we define the* embeddeding preorder *on $\mathcal{P}$ written $p \rhd q$ by the following rules*

- $p_i \trianglerighteq q \Rightarrow \text{node}(p_1, p_2) \rhd q$ *for* $i = 1, 2$
- $p_1 \rhd q_1 \wedge p_2 \trianglerighteq q_2 \Rightarrow \text{node}(p_1, p_2) \rhd \text{node}(q_1, q_2)$
- $p_1 \trianglerighteq q_1 \wedge p_2 \rhd q_2 \Rightarrow \text{node}(p_1, p_2) \rhd \text{node}(q_1, q_2)$

With $\trianglerighteq$ as the reflexive closure of $\rhd$.

The preorder $\rhd$ can be used to verify a structural decrease in values: if $t : \boldsymbol{B}(p)$, $u : \boldsymbol{B}(q)$ in a common context and $p \rhd q$, then the maximum size of normal forms of t is strictly greater that the maximum size of normal forms in u. This explains why we must forbid $\_$ in the definition of $\rhd$, as a term can be simultaneously typed in $\boldsymbol{B}(\text{node}(\_, \_))$ and $\boldsymbol{B}(\_)$ (and so no decrease is possible).

Non termination can intuitively be traced to cycles in the dependency graph. We wish to consider termination on terms with erased pattern arguments and type annotations.

3 Operational Semantics and the Main Theorem

Rewriting needs to be performed over terms with erased pattern annotations. The problem with the naïve definition of rewriting arises when trying to match on patterns. Take the rule

$$f \;\; \text{node}(\alpha, \beta) \;\; (\text{Node} \;\; x \;\; y) \to \text{Leaf}$$

In the presence of this rule, we wish to have, for instance, the reduction

$$f \;\; \_ \;\; (\text{Node} \;\; (g \;\; x) \;\; (h \;\; x)) \to \text{Leaf}$$

where g and h are arbitrary defined symbols. However, there is no substitution θ such that $\text{node}(\alpha, \beta)\theta = \_$. There are two ways to deal with this. Either we take subtyping into account

when performing matching, or we erase the pattern arguments when performing reduction. We adopt the second solution, which has the advantage of requiring fewer reductions, and is closer to practice in languages with dependent type annotations (see for example McKinna [30]). Symmetrically, we erase pattern abstractions as well.

▶ **Definition 3.1.** *We define the set of* erased terms $\mathcal{T}rm^{|\cdot|}$ *as:*

$$t, u \in \mathcal{T}rm^{|\cdot|} \coloneqq x \mid f \mid \lambda x.t \mid t\ u \mid \text{Leaf} \mid \text{Node}$$

Where $x \in \mathcal{X}$ and $f \in \mathcal{F}$.

Given a term $t \in \mathcal{T}rm$, we define the erasure $|t| \in \mathcal{T}rm^{|\cdot|}$ *of t as:*

$$\begin{array}{ll} |x| & = x \\ |f| & = f \\ |\lambda x\!:\!T.t| & = \lambda x.|t| \\ |\lambda\alpha.t| & = |t| \\ |t\ u| & = |t|\ |u| \\ |t\ p| & = |t| \\ |\,\text{Leaf}\,| & = \text{Leaf} \\ |\,\text{Node}\,| & = \text{Node} \end{array}$$

An erased term can intuitively be thought of as the compiled form of a well typed term.

▶ **Definition 3.2.** *An erased term t* head rewrites *to a term u if there is some rule $l \to r \in \mathcal{R}$ and some substitution σ from $\mathcal{X}$ to terms in $\mathcal{T}rm^{|\cdot|}$ such that*

$$|l|\sigma = t \wedge |r|\sigma = u$$

We define β-reduction $\to_\beta$ as

$$\lambda x.t\ u \to_\beta t\{x \mapsto u\}$$

And we define the reduction $\to$ *as the closure of head-rewriting and β-reduction by term contexts. We then define $\to^*$ and $\to^+$ as the symmetric transitive and transitive closure of $\to$, respectively.*

We can now express our termination criterion. We need to consider the *strongly connected components*, or SCCs of the typed dependency graph. A strongly connected component of a graph $\mathcal{G}$ is a full subgraph such that each node is reachable from all the others.

▶ **Definition 3.3.** *Let $\mathcal{G}$ be the typed dependency graph for $\mathcal{R}$ and let $\mathcal{G}_1, \ldots, \mathcal{G}_n$ be the SCCs of $\mathcal{G}$. Suppose that for each $\mathcal{G}_i$, there is a* simple projection $\iota^i : \Sigma \to \mathbb{N}$ *which to $f \in \Sigma$ associates an integer $1 \leq \iota^i_f \leq k$ (with k the number of recursive arguments of f).*

We say that $\mathcal{R}$ passes the simple projection criterion *for ι if*

- *For each $1 \leq i \leq n$ and each rule $f^\sharp(p_1, \ldots, p_n) \to g^\sharp(q_1, \ldots, q_m)$ in $\mathcal{G}_i$, we have $p_{\iota^i_f} \trianglerighteq q_{\iota^i_g}$.*
- *For each cycle in $\mathcal{G}_i$, there is some rule $f^\sharp(p_1, \ldots, p_n) \to g^\sharp(q_1, \ldots, q_m)$ such that*

$$p_{\iota^i_f} \triangleright q_{\iota^i_g}$$

▶ **Theorem 3.4. (Main theorem)**
Suppose that there is ι such that $\mathcal{R}$ passes the simple projection criterion for ι. Then for every Γ, t, T such that $\Gamma \vdash t : T$,

$$|t| \in \mathcal{SN}_\mathcal{R}$$

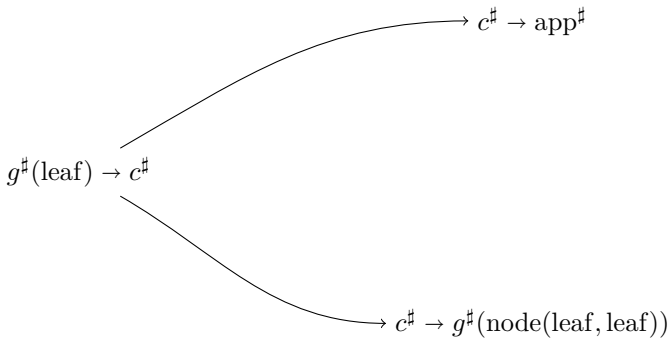


Figure 3 Dependency graph of Example 1

The proof of this theorem uses classic computability methods, and can be found in the online version, from the authors homepage. Let us give two examples of the application of this technique.

▶ **Example 1.** *Take the rewrite system given by the signature:*

$$\text{app} : \forall\alpha\beta.(\boldsymbol{B}(\alpha) \to \boldsymbol{B}(\beta)) \to \boldsymbol{B}(\alpha) \to \boldsymbol{B}(\beta), \quad c : \boldsymbol{B}(\text{leaf}), \quad g : \forall\alpha.\boldsymbol{B}(\alpha) \to \boldsymbol{B}(\text{leaf})$$

We give the rewrite rules:

$$\text{app} \ \to \lambda\alpha\beta.\lambda x : \boldsymbol{B}(\alpha) \to \boldsymbol{B}(\beta).\lambda y : \boldsymbol{B}(\alpha).x \ y$$

$$c \to \text{app} \ \ \text{node}(\text{leaf}, \text{leaf}) \ \ \text{leaf} \ \ (g \ \ \text{node}(\text{leaf}, \text{leaf})) \ \ (\text{Node}\, \text{leaf}\, \text{leaf} \ \ \text{Leaf} \ \ \text{Leaf})$$

$$g \ \ \text{node}(\alpha, \beta) \ \ (\text{Node}\, \alpha \ \ \beta \ \ x \ \ y) \to \text{Leaf}$$

$$g \ \ \text{leaf} \ \ \text{Leaf} \to c$$

or, in more readable form with pattern arguments and type annotations omitted:

$$\begin{aligned} \text{app} &\to \lambda x.\lambda y.x \ y \\ c &\to \text{app} \ \ g \ \ (\text{Node} \ \ \text{Leaf} \ \ \text{Leaf}) \\ g \ \ (\text{Node} \ \ x \ \ y) &\to \text{Leaf} \\ g \ \ \text{Leaf} &\to c \end{aligned}$$

It is possible to verify that the criterion can be applied and that in consequence, according to Theorem 3.4, all well typed terms are strongly normalizing under $\mathcal{R} \cup \beta$.

Indeed, we may easily check that each of these rules is minimally typed in some context. Furthermore, we can check that the dependency graph in Figure 3 has no cycles.

One may object that if we inline the definition of app and perform β-reduction on the right-hand sides of rules we obtain a rewrite system that can be treated with more conventional methods, such as those performed by the `AProVe` tool [19] (on terms without abstraction, and without β-reduction). However this operation can be very costly if performed automatically and is, in its most naïve form, ineffective for even slightly more complex higher-order programs such as *map*, which performs pattern matching and for which we need to instantiate. By resorting to typing, we allow termination to be proven using only "local" considerations, as the information encoding the semantics of app is contained in its type.

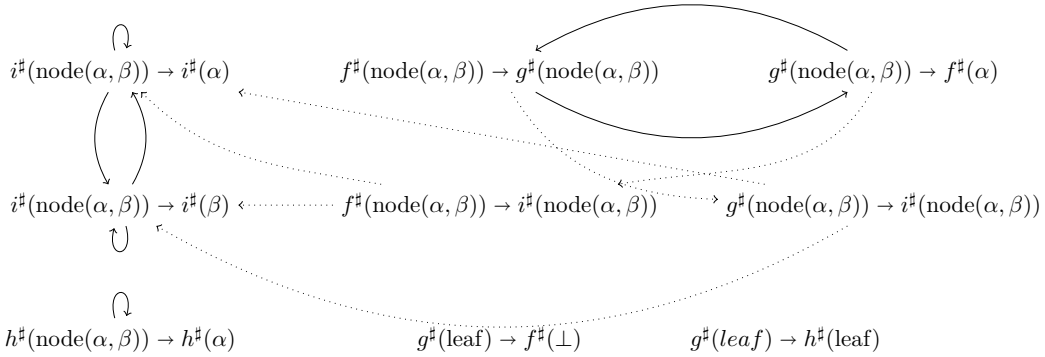


Figure 4 The dependency graph for Example 2

However it becomes necessary, if one desires a fully automated termination check on an unannotated system, to somehow infer the type of defined constants, and possibly perform an analysis quite similar in effect to the one proposed above. We believe that to this end one may apply known type inference technology, such as the one described in [14], to compute these annotated types. In conclusion, what used to be a termination problem becomes a type inference problem, and may benefit from the knowledge and techniques of this new community, as well as facilitate integration of these techniques into type-theoretic based proof assistants like Coq [15].

Let us examine a second, slightly more complex example, in which there is "real" recursion.

▶ **Example 2.** *Let $\mathcal{R}$ be the rewrite system defined by*

$$\begin{aligned}
f\ (\mathrm{Node}\ x\ y) &\to g\ (i\ (\mathrm{Node}\ x\ y)\\
g\ (\mathrm{Node}\ x\ y) &\to f\ (i\ x)\\
g\ \mathrm{Leaf} &\to f\ (h\ \mathrm{Leaf})\\
i\ (\mathrm{Node}\ x\ y) &\to \mathrm{Node}\ (i\ x)\ (i\ y)\\
i\ \mathrm{Leaf} &\to \mathrm{Leaf}\\
h\ (\mathrm{Node}\ x\ y) &\to h\ x
\end{aligned}$$

Again with the type arguments omitted for readability, and with types $f, g : \forall\alpha.\boldsymbol{B}(\alpha) \to \boldsymbol{B}(\_)$, $h : \forall\alpha.\boldsymbol{B}(\alpha) \to \boldsymbol{B}(\bot)$ and $i : \forall\alpha.\boldsymbol{B}(\alpha) \to \boldsymbol{B}(\alpha)$. Every equation can be typed in the context $\Gamma = x : \boldsymbol{B}(\alpha), y : \boldsymbol{B}(\beta)$. The system with full type annotations is given in the appendix (or in the online version). The dependency graph is given in Figure 4, and has as SCCs the full subgraphs of $\mathcal{G}_\mathcal{R}$ with nodes $\{i^\sharp(\mathrm{node}(\alpha,\beta)) \to i^\sharp(\alpha), i^\sharp(\mathrm{node}(\alpha,\beta)) \to i^\sharp(\beta)\}$, $\{f^\sharp(\mathrm{node}(\alpha,\beta)) \to g^\sharp(\mathrm{node}(\alpha,\beta), g^\sharp(\mathrm{node}(\alpha,\beta) \to f^\sharp(\alpha)\}$ and $\{h^\sharp(\mathrm{node}(\alpha,\beta)) \to h^\sharp(\alpha)\}$ respectively.

Taking $\iota_s = 1$ for every SCC and every symbol $s \in \Sigma$, it is easy to show that every SCC respects the decrease criterion on cycles. For example, in the cycle

$$f^\sharp(\mathrm{node}(\alpha,\beta)) \to g^\sharp(\mathrm{node}(\alpha,\beta)) \leftrightarrows g^\sharp(\mathrm{node}(\alpha,\beta)) \to f^\sharp(\alpha)$$

we have $\mathrm{node}(\alpha,\beta) \trianglerighteq \mathrm{node}(\alpha,\beta)$ and $\mathrm{node}(\alpha,\beta) \triangleright \alpha$, so the cycle is weakly decreasing with at least one strict decrease.

We may then again apply the correctness theorem to conclude that the erasure of all well-typed terms are strongly normalizing with respect to $\mathcal{R} \cup \beta$.

Note that the minimality condition is important: otherwise one could take

$$f : \forall\alpha\beta.\boldsymbol{B}(\alpha) \to \boldsymbol{B}(\beta) \to \boldsymbol{B}(\_)$$

with the rule

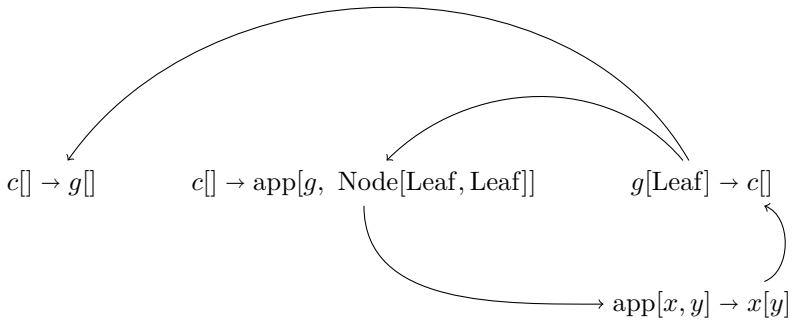
$$f\,\mathrm{node}(\mathrm{leaf},\mathrm{leaf})\,\mathrm{leaf}\;\; x\;y \to f\,\mathrm{leaf}\,\mathrm{leaf}\;\; y\;y$$

This rule can be typed in the context $x : \boldsymbol{B}(\mathrm{node}(\mathrm{leaf},\mathrm{leaf})), y : \boldsymbol{B}(\mathrm{leaf})$, but not minimally typed, as the variables x and y do not have type $\boldsymbol{B}(\alpha)$ for some variable α, and passes the termination criterion: the dependency graph is without cycles, as $\mathrm{node}(\mathrm{leaf},\mathrm{leaf})$ does not unify with leaf. However, this system leads to the non terminating reduction $f\,\mathrm{Leaf}\,\mathrm{Leaf} \to f\,\mathrm{Leaf}\,\mathrm{Leaf}$.

4 Comparison, future work

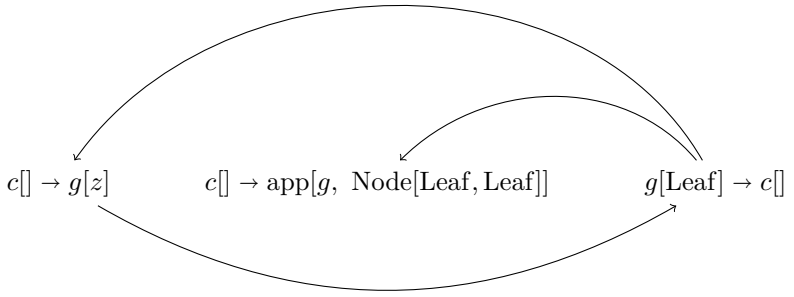
Several extensions of dependency pairs to different forms of higher-order rewriting have been proposed, first for applicative systems (variables may appear in application position, but there are no λ-abstractions) [19, 35, 3] and subsequently for more expressive systems including λ-abstractions [27, 8]. For the frameworks that do not handle the presence of bound variables, the usual approach is to defunctionalize (also called *lambda-lifting*) [16, 23] which is a whole program transformation which yields operationally equivalent terms for a given rewrite system.

All the techniques cited above, when applied to Example 1, where we may replace the rule $\mathrm{app} \to \lambda x.\lambda y.x\;y$ with the rule $\mathrm{app}\;\; x\;y \to x\;y$ (which does not involve bound variables), generate a dependency graph *with* cycles. For example, in Sakai & Kusakari [35], using the so-called "dynamic approach" the dependency graph is:



It is of course possible to prove that there are no infinite chains for this problem (the criterion is complete), but we have not much progressed from the initial formulation!

Using the so-called "static approach" from the same paper, which is based on computability (as is our framework), we obtain the following graph:



However it is not possible to prove that there are no infinite chains for this problem, as there is one! Therefore the criterion presented in the present paper allows a finer analysis of the possible calls.

The termination checking software `AProVE`, which used methods drawn in part from Giesl *et al* [19] succeeds in proving termination of Example 1, by using an analysis involving the computation of variable instances and symbolic reduction. As noted previously, our approach does not need such an expensive analysis as the information required *is already contained in the type information*. However it seems that such an analysis may be used to *infer* the type annotations required in our framework. At the moment it is unclear how the typing approach precisely compares to these techniques. More investigation is clearly needed in this direction.

`AProVE` can also easily prove termination of the second rewrite system (Example 2). However semantic information needs to be inferred (for example a polynomial interpretation needs to be given) when trying to well-order the cycle

$$f\ (\text{Node}\ x\ y) \to g\ (i\ (\text{Node}\ x\ y)) \leftrightarrows g\ (\text{Node}\ x\ y) \to f\ (i\ x)$$

This information is already supplied by our type system (through the fact that i is of type $\forall\alpha.\boldsymbol{B}(\alpha) \to \boldsymbol{B}(\alpha)$), and therefore it suffices to consider only syntactic information on the approximated dependency graph. The *subterm criterion* by Aoto and Yamada [3] is insufficient to treat this example.

Work by Bove and Capretta [12] allows one to use dependent types to encode functions that terminate for complex reasons, using functions which can be shown to be structurally recursive. While the theoretical power of this approach is stronger than that of ours, it not possible to give a straightforward encoding of our type-based framework using this approach, due to the presence of subtyping. Note also that our criterion applies to open terms with erased arguments, whereas the Bove-Capretta method does not.

The framework described here is only the first step towards a satisfactory *type-based dependency pair framework* using refinement types. We intuitively consider a "type level" first-order rewrite system, use standard techniques to show that that system is terminating, and show that this implies termination of the object level system. More work is required to obtain a satisfactory "dependency pairs by typing" framework.

Our work seems quite orthogonal to the *size-change principle* [28], which suggests we could apply this principle to treat cycles in the typed dependency graph, as a more powerful criterion than simple decrease on one indexed argument.

It is clear that the definitions and proofs in the current work extend to other first-order inductive types like lists, Peano natural numbers, etc. We conjecture that this framework can be extended to more general positive inductive types, like the type of Brower ordinals [9]. These kinds of inductive types seem to be difficult to treat with other (non type-based) methods.

For now types have to be explicitly given by the user, and for complete automation of our criterion it is necessary to infer the type annotations. Notice that trivial annotations (return type always $\boldsymbol{B}(\_)$) can very easily be infered automatically. Some work on automatic inference of type-level annotations has been carried out by Chin *et al* . [14] which considers annotations in the language of linear arithmetic, and by Barthe, Gregoire and Pastawski [6] for a more restricted language of size-types. We believe that inference of explicit type information in the terms is quite feasible with current state-of-the-art methods, for example those used for inferring the type of functional programs using GADTs [33].

We only consider matching on non-defined symbols, though an extension to a framework with matching on defined symbols seems feasible if we add some conversion rule to our type system.

We believe that refinement types are simply an alternative way of presenting the dependency pair method for higher-order rewrite systems. It is the occasion to draw a parallel between the types community and the rewriting community, by emphasizing that techniques used for the inference of dependent type annotations (for example work on *liquid types* [34]), may in fact be used to infer information necessary for proving termination and (we believe) vice-versa. It may also be interesting in the case of a programming language for the user to supply the types as documentation, in what some call "type directed programing".

Acknowledgements We thank Frederic Blanqui and Andreas Abel for the discussions that led to the birth of this work and for very insightful comments concerning a draft of this paper, as well as anonymous referees for numerous corrections.

References

1 A. Abel. Termination checking with types. *Theoretical Informatics and Applications*, 38(4):277–319, 2004.

2 A. Abel. Semi-continuous sized types and termination. In Z. Ésik, editor, *CSL*, volume 4207 of *Lecture Notes in Computer Science*, pages 72–88. Springer, 2006.

3 T. Aoto and T. Yamada. Dependency pairs for simply typed term rewriting. In J. Giesl, editor, *RTA*, volume 3467 of *Lecture Notes in Computer Science*, pages 120–134. Springer, 2005.

4 T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236(1-2):133–178, 2000.

5 G. Barthe, M. J. Frade, E. Giménez, L. Pinto, and T. Uustalu. Type-based termination of recursive definitions. *Mathematical Structures in Computer Science*, 14(1):97–141, 2004.

6 G. Barthe, B. Grégoire, and F. Pastawski. Practical inference for typed-based termination in a polymorphic setting. In *Typed Lambda Calculi and Applications*, pages 71–85, 2009.

7 F. Blanqui. A type-based termination criterion for dependently-typed higher-order rewrite systems. In *Proc. of the 15th International Conference on Rewriting Techniques and Applications*, volume 3091 of *Lecture Notes in Computer Science*, 2004.

8 F. Blanqui. Higher-order dependency pairs. In *Proceedings of the 8th International Workshop on Termination*, 2006.

9 F. Blanqui, J.-P. Jouannaud, and M. Okada. Inductive-data-type Systems. *Theoretical Computer Science*, 272:41–68, 2002.

10 F. Blanqui and C. Riba. Combining typing and size constraints for checking the termination of higher-order conditional rewrite systems. In *Proceedings of the 13th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, Lecture Notes in Computer Science 4246, 2006.

11 F. Blanqui and C. Roux. On the relation between sized-types based termination and semantic labelling. In E. Grädel and R. Kahle, editors, *CSL*, volume 5771 of *Lecture Notes in Computer Science*, pages 147–162. Springer, 2009.

12 A. Bove and V. Capretta. Modelling general recursion in type theory. *Mathematical Structures in Computer Science*, 15(4):671–708, 2005.

13 N. G. D. Bruijn. The mathematical language automath, its usage, and some of its extensions. In M. Laudet, editor, *Proceedings of the Symposium on Automatic Demonstration*, volume 125, pages 29–61. Springer-Verlag, 1968.

14 W.-N. Chin and S.-C. Khoo. Calculating sized types. *Journal of Higher-Order and Symbolic Computation*, 14(2-3):261–300, 2001.

15 Coq Development Team. *The Coq Reference Manual, Version 8.2.* INRIA Rocquencourt, France, 2008. `http://coq.inria.fr/`.

16 O. Danvy and L. R. Nielsen. Defunctionalization at work. In *proceedings of PPDP*, pages 162–174. ACM, 2001.

17 T. Freeman and F. Pfenning. Refinement types for ML. *SIGPLAN Not.*, 26(6):268–277, 1991.

18 J. Giesl, T. Arts, and E. Ohlebusch. Modular termination proofs for rewriting using dependency pairs. *J. Symb. Comput.*, 34:21–58, July 2002.

19 J. Giesl, R. Thiemann, and P. Schneider-Kamp. Proving and disproving termination of higher-order functions. In *proceedings of the 5th FROCOS conference*, pages 216–231. Springer, 2005.

20 J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Mechanizing and improving dependency pairs. *Journal of Automated Reasoning*, 37(3):155–203, 2006.

21 N. Hirokawa and A. Middeldorp. Tyrolean termination tool: Techniques and features. *Information and Compututation*, 205:474–511, April 2007.

22 J. Hughes, L. Pareto, and A. Sabry. Proving the correctness of reactive systems using sized types. In *Proceedings of the 23th ACM Symposium on Principles of Programming Language*, 1996.

23 T. Johnsson. Lambda lifting: Transforming programs to recursive equations. In *Functional Programming Languages and Computer Architecture*, pages 190–203. Springer-Verlag, 1985.

24 S. Jones and E. Meijer. Henk: a typed intermediate language, 1997.

25 J.-P. Jouannaud, C. Kirchner, and H. Kirchner. Incremental construction of unification algorithms in equational theories. In *Proceedings of the 10th Colloquium on Automata, Languages and Programming*, pages 361–373, London, UK, 1983. Springer-Verlag.

26 J.-P. Jouannaud and M. Okada. A computational model for executable higher-order algebraic specification languages. In *Proceedings of the sixth annual IEEE Symposium on Logic in Computer Science (LICS '91)*, pages 350–361, 1991.

27 K. Kusakari, Y. Isogai, M. Sakai, and F. Blanqui. Static dependency pair method based on strong computability for higher-order rewrite systems. *IEICE Transactions*, 92-D(10):2007–2015, 2009.

28 C. S. Lee, N. D. Jones, and A. Ben-Amram. The size-change principle for program termination. In *Proceedings of POPL'01*, pages 81–92. ACM Press, 2001.

29 C. McBride and J. McKinna. The view from the left. *Journal of Functionnal Programming*, 14(1):69–111, 2004.

30 J. McKinna. Why dependent types matter. *SIGPLAN Not.*, 41(1):1–1, 2006.

31 R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

32 U. Norell. *Towards a practical programming language based on dependent type theory.* PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

33 S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for gadts. In *Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*, ICFP '06, pages 50–61, New York, NY, USA, 2006. ACM.

34 P. M. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In R. Gupta and S. P. Amarasinghe, editors, *PLDI*, pages 159–169. ACM, 2008.

35 M. Sakai and K. Kusakari. On dependency pair method for proving termination of higher-order rewrite systems. *IEICE Transactions on Information and Systems*, E88-D(3):583–593, 2005.

36 H. Xi and D. Scott. Dependent types in practical programming. In *In Proceedings of ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 214–227. ACM Press, 1998.

Weakening the Axiom of Overlap in Infinitary Lambda Calculus

Paula Severi[1,2] and Fer-Jan de Vries[1,3]

1 **Department of Computer Science, University of Leicester, UK**
2 ps56@mcs.le.ac.uk
3 fdv1@mcs.le.ac.uk

Abstract

In this paper we present a set of necessary and sufficient conditions on a set of lambda terms to serve as the set of meaningless terms in an infinitary bottom extension of lambda calculus. So far only a set of sufficient conditions was known for choosing a suitable set of meaningless terms to make this construction produce confluent extensions. The conditions covered the three main known examples of sets of meaningless terms. However, the much later construction of many more examples of sets of meaningless terms satisfying the sufficient conditions renewed the interest in the necessity question and led us to reconsider the old conditions.

The key idea in this paper is an alternative solution for solving the overlap between beta reduction and bottom reduction. This allows us to reformulate the Axiom of Overlap, which now determines together with the other conditions a larger class of sets of meaningless terms. We show that the reformulated conditions are not only sufficient but also necessary for obtaining a confluent and normalizing infinitary lambda beta bottom calculus. As an interesting consequence of the necessity proof we obtain for infinitary lambda calculus with beta and bot reduction that confluence implies normalization.

1998 ACM Subject Classification F.4.1 Mathematical Logic (F.1.1, I.2.2, I.2.3, I.2.4)

Keywords and phrases Infinitary Lambda Calculus, Confluence, Normalization

Digital Object Identifier 10.4230/LIPIcs.RTA.2011.313

Category Regular Research Paper

1 Introduction

In Lambda Calculus there exists a perhaps surprising number of different formalisations of the idea of undefined or meaningless term [3, 4, 1, 13, 6, 11]. The rough intuition is that such terms cannot contribute information to any context in which they are placed, and may be mapped to the bottom element of the semantic domain of a denotational semantics. In this paper we are interested in the sets of meaningless terms that arise when one tries to extend lambda calculus with infinite terms and infinite strongly converging reductions in such a way that the confluence property is preserved.

The first attempt to characterise *sets of meaningless terms* axiomatically was made for first order term rewriting [2]. These axioms were revised and further extended to lambda calculus in [11, 7], and recently to combinatory reduction systems [12]. The axioms are general assumptions for ensuring confluence and normalization of infinitary lambda calculi $\lambda^{\infty}_{\beta\bot_{\mathcal{U}}}$ with a $\bot_{\mathcal{U}}$-rule that rewrites the terms of the set $\mathcal{U}$ of meaningless terms to $\bot$. This general notion of set of meaningless terms captures two well-known examples from lambda calculus: the set $\overline{\mathcal{HN}}$ of terms without head normal form and the set $\overline{\mathcal{WN}}$ of terms without

22nd International Conference on Rewriting Techniques and Applications.
Editor: M. Schmidt-Schauß; pp. 313–328

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


weak head normal form. The initial papers on infinitary lambda calculus revealed a third main example: the set $\overline{\mathcal{TN}}$ of terms without top normal form [3, 6, 9]. Only later in [15, 16] we realised that there are far more sets of meaningless lambda terms which give rise to an ample collection of models of the finitary and the infinitary lambda calculus.

It is now natural to ask: for which sets $\mathcal{U}$ of meaningless terms is the corresponding infinitary lambda calculus $\lambda^{\infty}_{\beta\bot_{\mathcal{U}}}$ confluent? The confluence proofs in [11, 7] show that the axioms in the notion of set of meaningless term are a sufficient condition, but are they also *necessary*?

In this paper we will show sufficient and necessary conditions for having a confluent and normalizing infinitary lambda calculus $\lambda^{\infty}_{\beta\bot_{\mathcal{U}}}$. The key idea can be found in Section 3 where we present a new solution for solving the overlap between beta reduction and bottom reduction. This allows us to give a reformulation of the Axiom of Overlap from [11, 7] which we call *Axiom of Weak Overlap*. If we replace Overlap by Weak Overlap in the definition of set of meaningless terms, then we obtain the larger class of *sets of weak meaningless terms*. In Section 4 we give many new examples of sets of weak meaningless sets. In Section 5, we prove that the infinitary lambda beta bottom calculus $\lambda^{\infty}_{\beta\bot_{\mathcal{U}}}$ is confluent for any set of weak meaningless terms $\mathcal{U}$. In Section 8, we prove the converse: whenever an infinitary lambda beta bottom calculus $\lambda^{\infty}_{\beta\bot_{\mathcal{U}}}$ is confluent, there exists a set $\mathcal{U}'$ of weak meaningless terms that defines the same reduction as $\mathcal{U}$. As an unexpected result in Section 7 we obtain that confluence implies normalization for infinitary lambda beta bottom calculi $\lambda^{\infty}_{\beta\bot_{\mathcal{U}}}$.

2 Infinitary Lambda Calculus

We will now briefly recall some notions and facts of infinitary lambda calculus from our earlier work [8, 9, 7, 14, 17]. We assume familiarity with basic notions and notations from [3]. Let Λ be the set of λ-terms and $\Lambda_{\bot}$ be the set of finite λ-terms with $\bot$.

▶ **Definition 2.1** (Finite and Infinite Lambda Terms)**.** The set $\Lambda^{\infty}_{\bot}$ of finite and infinite λ-terms is defined by coinduction using the grammar:

$$M ::= \bot \mid x \mid (\lambda x.M) \mid (MM)$$

where x is a variable from some fixed, large enough set of variables $\mathcal{V}$. The set Λ^{∞} consists of the terms in $\Lambda^{\infty}_{\bot}$ which do not contain $\bot$. The set $(\Lambda^{\infty})^0$ consists of the terms in Λ^{∞} that are closed, i.e. without free variables.

Having defined the raw terms, we now follow the usual conventions on syntax of finitary and infinitary lambda calculus [3, 7]. As explained in the latter, many concepts from finitary lambda calculus generalise immediately to the infinitary setting, context, position, (head) redex, free and bound variables, (head) normal form and so on. As customary in finitary lambda calculus, we identify terms that are α-convertible and we use the variable convention (bound variables are implicitly renamed before a substitution is made) to avoid variable capture. We will use the notation M^{σ} to denote the simultaneous substitution of the free variables in M by substitution $\sigma : \mathcal{V} \to \Lambda^{\infty}_{\bot}$.

▶ Notation 2.2. We will use the following abbreviations of λ-terms:

$$\begin{array}{lcllcllcl}
\mathbf{I} & = & \lambda x.x & \mathbf{O} & = & \lambda x_1.\lambda x_2.\lambda x_3.\ldots & \mathbf{\Omega} & = & (\lambda x.xx)\lambda x.xx \\
\mathbf{1} & = & \lambda xy.xy & M^{\omega} & = & M(M(M\ldots)) & \mathbf{\Omega}_{\eta} & = & \lambda x_1.(\lambda x_2.(\lambda x_3.\ldots x_3)x_2)x_1 \\
\mathbf{K} & = & \lambda xy.x & & & & & &
\end{array}$$

The point of the syntax of infinitary lambda calculus is to have one framework that captures both finite and infinite terms. This has the pleasant consequence that Böhm trees don't have to be defined in a separate formalism. Böhm trees are nothing else than normal forms under a particular notion of reduction. So, in the following we will freely identify trees with terms in $\Lambda^\infty_\bot$. In [9, 11, 7], an alternative definition of the set $\Lambda^\infty_\bot$ is given using a metric. The coinductive and metric definitions are equivalent [5]. Note that here we follow [7] and consider only one set of λ-terms, namely $\Lambda^\infty_\bot$, in contrast to the formulations in [9, 11] where several metric completions (all subsets of $\Lambda^\infty_\bot$) of the set of finite terms are considered.

We will consider infinitary lambda calculus with two reductions rules: the familiar beta rule and the $\bot_\mathcal{U}$-rule which is parametrised by some set $\mathcal{U}$ of terms. This $\bot_\mathcal{U}$-rule generalises the $\bot$-rule used to define Böhm trees in which terms without head normal form with are identified with bottom [11, 7].

▶ **Definition 2.3** (β-rule)**.** We consider the β-rule on $\Lambda^\infty_\bot$:

$$(\lambda x.M)N \to M[x := N] \quad (\beta)$$

The one step reduction $\to_\beta$ is the smallest binary relation containing β and closed under contexts.

▶ **Definition 2.4** ($\bot_\mathcal{U}$-rule)**.** Let $\mathcal{U} \subseteq \Lambda^\infty$. We define the $\bot_\mathcal{U}$-rule on $\Lambda^\infty_\bot$:

$$\frac{M[\bot := \mathbf{\Omega}] \in \mathcal{U} \quad M \neq \bot}{M \to \bot}\ (\bot_\mathcal{U})$$

Occasionally, we may denote $\bot_\mathcal{U}$ just by $\bot$. The one step reduction $\to_{\bot_\mathcal{U}}$ is the smallest binary relation containing $\bot_\mathcal{U}$ and closed under contexts. The reduction $\to_{\beta\bot_\mathcal{U}}$ is the smallest binary relation containing β and $\bot_\mathcal{U}$ closed under contexts.

We will consider calculi with various combinations of these rules: $\lambda^\infty_{\beta\bot_\mathcal{U}}$, λ^∞_β and $\lambda^\infty_{\bot_\mathcal{U}}$. We will use the notation λ^∞_ρ where ρ is a variable ranging over $\{\beta\bot_\mathcal{U},\ \beta,\ \bot_\mathcal{U}\}$.

▶ **Definition 2.5** (Subterm at a certain Position)**.** Positions are finite sequences of 0, 1 and 2's and include the empty sequence $\langle\rangle$. Provided it exists, the subterm $M|_p$ of a term $M \in \Lambda_\bot$ at position p is defined by induction as usual:

$$M|_{\langle\rangle} = M \quad (\lambda xM)|_{0p} = M|_p \quad (MN)|_{1p} = M|_p \quad (MN)|_{2p} = N|_p$$

The depth of a subterm N at position p occurs in M is the length of p.

▶ **Definition 2.6** (Truncation)**.** The truncation of M at depth n is obtained by replacing all subterms at depth n by $\bot$ and is denoted by M^n.

▶ **Definition 2.7** (Metric)**.** We define a metric $d : \Lambda_\bot \times \Lambda_\bot \to [0, 1]$ as follows: $d(M, N) = 0$, if $M = N$ and $d(M, N) = 2^{-m}$, where $m = max\{M^n = N^n \mid n \in \mathbb{N}\}$.

The metric will be used in the definition of a transfinite reduction sequence. Note that we will use customary notation like α, β, γ for arbitrary ordinals and λ for limit ordinals. The context will disambiguate the overloading.

▶ **Definition 2.8** (Strongly Converging Reductions [7])**.** Let $\lambda^\infty_\rho = (\Lambda^\infty_\bot, \to_\rho)$.

1. A transfinite reduction sequence of length α in λ^∞_ρ, where α is any ordinal, is a sequence of reduction steps $(M_\beta \to_\rho M_{\beta+1})_{\beta<\alpha}$. In the step $M_\beta \to_\rho M_{\beta+1}$, we denote the position of the contracted redex in M_β by p_β and the depth of this redex by d_β.

2. We define that a sequence $(M_\beta \to_\rho M_{\beta+1})_{\beta<\alpha}$ is a Cauchy (converging) reduction sequence from M_0 to M_α if, for every limit ordinal $\lambda \leq \alpha$, the distance $d(M_\beta, M_\lambda)$ tends to 0 as β approaches λ from below.
3. We define that a sequence $(M_\beta \to_\rho M_{\beta+1})_{\beta<\alpha}$ is *strongly converging* if, it is Cauchy converging and if, for every limit ordinal $\lambda \leq \alpha$, the depth d_β of the contracted redex in $M_\beta \to_\rho M_{\beta+1}$ tends to infinity as β approaches λ from below.

In contrast to strongly converging reductions Cauchy converging reductions don't project well. Hence strongly converging reduction is the natural notion of reduction to study. This preference is reflected in the next notation.

▶ Notation 2.9. Let $\lambda^\infty_\rho = (\Lambda^\infty_\bot, \to_\rho)$.
1. $M \to_\rho N$ denotes a one step reduction from M to N;
2. $M \twoheadrightarrow_\rho N$ denotes a finite reduction from M to N;
3. $M \twoheadrightarrow\!\!\!\!\!\to_\rho N$ denotes a strongly converging reduction from M to N.

▶ **Definition 2.10.** Let $\lambda^\infty_\rho = (\Lambda^\infty_\bot, \to_\rho)$.
1. λ^∞_ρ is *confluent*, if ${}_\rho\twoheadleftarrow \circ \twoheadrightarrow_\rho \subseteq \twoheadrightarrow_\rho \circ {}_\rho\twoheadleftarrow$.
2. A term M in λ^∞_ρ is in *ρ-normal form*, if there is no N in λ^∞_ρ such that $M \to_\rho N$.
3. λ^∞_ρ is *normalizing*, if for all $M \in \Lambda^\infty_\bot$ there is an N in ρ-normal form such that $M \twoheadrightarrow\!\!\!\!\!\to_\rho N$.

If $\lambda^\infty_{\beta\bot_\mathcal{U}}$ is confluent and normalizing, the normal form of a term M is unique and denoted by $\mathsf{nf}_\mathcal{U}(M)$.

▶ **Definition 2.11** (Rootactive)**.** Let $M \in \Lambda^\infty_\bot$. We say that M is *rootactive*, if for any $N \in \Lambda^\infty_\bot$, if $M \twoheadrightarrow\!\!\!\!\!\to_\beta N$ then $N \twoheadrightarrow_\beta (\lambda x.P)Q$ for some $P, Q \in \Lambda^\infty_\bot$. Let $\mathcal{R}$ denote the set $\{M \in \Lambda^\infty \mid M \text{ is rootactive}\}$ of bottom free rootactive terms.

▶ **Definition 2.12.** Let $M, N \in \Lambda^\infty$. We write $M \overset{\mathcal{U}}{\longleftrightarrow} N$, if N can be obtained from M by replacing some (possibly infinitely many) subterms in $\mathcal{U}$ by other terms in $\mathcal{U}$.

In the next definition, we follow the axiomatisation of [11] which is equivalent to the one in [7] which combines Closure under β-reduction and Closure under substitutions in one Descendants axiom.

▶ **Definition 2.13** ([7])**.** We give names to the following properties that a set $\mathcal{U} \subseteq \Lambda^\infty$ may satisfy:
1. **Axiom of Rootactiveness:** $\mathcal{R} \subseteq \mathcal{U}$.
2. **Axiom of Closure under β-reduction:** $M \twoheadrightarrow\!\!\!\!\!\to_\beta N$ implies $N \in \mathcal{U}$ for all $M, N \in \mathcal{U}$.
3. **Axiom of Closure under Substitution:** $M^\sigma \in \mathcal{U}$ for all $M \in \mathcal{U}$ and substitutions σ.
4. **Axiom of Overlap:** for all $M \in \mathcal{U}$, if $M = \lambda x.P$ then $(\lambda x.P)Q \in \mathcal{U}$ for all $Q \in \Lambda^\infty$.
5. **Axiom of Indiscernibility:** for all $M, N \in \Lambda^\infty$ such that $M \overset{\mathcal{U}}{\longleftrightarrow} N$, $M \in \mathcal{U}$ if and only if $N \in \mathcal{U}$.

In order to guarantee confluence of the Infinitary Lambda Calculi, we define the notion of sets of meaningless terms [11, 7].

▶ **Definition 2.14** (Meaningless Set)**.** 1. A set $\mathcal{U} \subseteq \Lambda^\infty$ is called a *set of meaningless terms* (meaningless set for short), if it satisfies the Axioms (1-5). These axioms are called *the axioms of meaningless terms.*
2. $\mathbb{M} = \{\mathcal{U} \subseteq \Lambda^\infty \mid \mathcal{U} \text{ is a set of meaningless terms}\}$.

In Section 4, we will show many examples of meaningless sets other than $\mathcal{R}$.

▶ **Theorem 2.15** (Sufficiency of Rootactiveness for Normalization [11, 7]). *Let $\mathcal{U} \subseteq \Lambda^\infty$. If $\mathcal{U}$ satisfies Rootactiveness, then $\lambda^\infty_{\beta\bot_\mathcal{U}}$ is normalizing.*

▶ **Theorem 2.16** (Sufficiency of Meaninglessness for Confluence and Normalization [11, 7]). *Let $\mathcal{U} \subseteq \Lambda^\infty$. If $\mathcal{U}$ is a set of meaningless terms, then $\lambda^\infty_{\beta\bot_\mathcal{U}}$ is confluent and normalizing.*

The following theorem relates the infinitary lambda calculus with models of the finite lambda calculus (see Definitions 5.2.7 and 5.3.1 in [3]).

▶ **Theorem 2.17** (λ-model $\mathfrak{M}_\mathcal{U}$). *Each set $\mathcal{U}$ such that $\lambda^\infty_{\beta\bot_\mathcal{U}}$ is confluent and normalizing gives rise to a λ-model denoted by $\mathfrak{M}_\mathcal{U}$.*

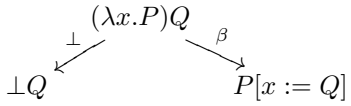
Proof. The domain of $\mathfrak{M}_\mathcal{U}$ is the set $\mathsf{nf}_\mathcal{U}(\Lambda^\infty_\bot)$ of normal forms of $\lambda^\infty_{\beta\bot_\mathcal{U}}$. We interpret a lambda term M by its normal form $\mathsf{nf}_\mathcal{U}(M)$ and we define application simply by $\mathsf{nf}_\mathcal{U}(M) \bullet \mathsf{nf}_\mathcal{U}(N) = \mathsf{nf}_\mathcal{U}(MN)$. ◀

We denote by $\mathbb{MOD}(\lambda) = \{\mathfrak{M}_\mathcal{U} \mid \mathcal{U}$ defines a confluent and normalizing $\lambda^\infty_{\beta\bot_\mathcal{U}}\}$, the class of models induced by the confluent and normalizing infinitary lambda calculi.

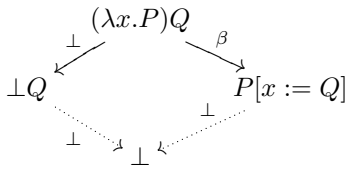
3 Reconsidering the Axiom of Overlap

In [11] it was noted that there is the possibility of overlap between the beta and the bottom rule for meaningles terms of the form $\lambda x.M$. This was resolved by the Axiom of Overlap. That was a satisfactory solution, as it covered the examples of meaningless terms that were known at the time. However, as we will show here, it is not the only way.

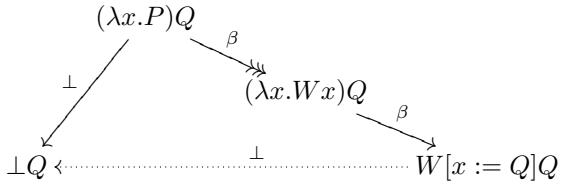
Let us first re-examine the rationale behind the Axiom of Overlap in detail. Let $\mathcal{U}$ be a set of meaningless terms. Overlap between $\bot$-reduction and β-reduction occurs when the $\bot$-redex is of the form $\lambda x.P$. This gives a divergence



The Axiom of Overlap solves this divergence in combination with Rootactiveness, Closure under β-reduction and Indiscernibility. When $\lambda x.P \in \mathcal{U}$ then by Overlap we have that $(\lambda x.P)Q \in \mathcal{U}$. By Rootactiveness and Indiscernibility also $\mathbf{\Omega}Q \in \mathcal{U}$. On one hand we have $\bot Q \to_\bot \bot$ since $(\bot Q)[\bot := \mathbf{\Omega}] = \mathbf{\Omega}Q \in \mathcal{U}$. On the other, by Closure under β-reduction, we have $P[x := Q] \in \mathcal{U}$ and thus also $P[x := Q] \to_\bot \bot$.



There is, however, another way of resolving this divergence, not considered in [11]. Suppose besides $\lambda x.P \in \mathcal{U}$ we also have $P \twoheadrightarrow_\beta Wx$ with $W \in \mathcal{U}$. Then if $\mathcal{U}$ satisfies Closure under substitution, we have that $W[x := Q] \in \mathcal{U}$ and then $(Wx)[x := Q] = W[x := Q]Q \to_\bot \bot Q$:



Thus we find an alternative axiom of overlap and an alternative notion of meaningless set:

▶ **Definition 3.1** (Axiom of Alternative Overlap, Set of Alternative Meaningless Terms)**.**

1. A set $\mathcal{U} \subseteq \Lambda^\infty$ is said to satisfy **Alternative Overlap**, if for each abstraction $\lambda x.P \in \mathcal{U}$, there is some $W \in \mathcal{U}$ such that $P \twoheadrightarrow_\beta Wx$.
2. A set $\mathcal{U} \subseteq \Lambda^\infty$ is called a *set of alternative meaningless terms*, if it satisfies Rootactiveness, Closure under β-reduction, Substitution, Alternative Overlap and Indiscernibility.
3. $\mathbb{AM} = \{\mathcal{U} \subseteq \Lambda^\infty \mid \mathcal{U}$ is a set of alternative meaningless terms$\}$.

We can capture both axioms, Overlap and Alternative Overlap, in a single general axiom:

▶ **Definition 3.2** (Axiom of Weak Overlap, Set of Weak Meaningless Terms)**.**

1. A set $\mathcal{U} \subseteq \Lambda^\infty$ is said to satisfy the axiom of **Weak Overlap**, if for each abstraction $\lambda x.P \in \mathcal{U}$ there is some $W \in \mathcal{U}$ such that $P \twoheadrightarrow_\beta Wx$, or $(\lambda x.P)Q \in \mathcal{U}$ for all $Q \in \Lambda^\infty$.
2. A set $\mathcal{U} \subseteq \Lambda^\infty$ is called a *set of weak meaningless terms*, if it satisfies the Axioms of Closure under β-reduction, Substitution, Weak Overlap, Rootactiveness and Indiscernibility.
3. $\mathbb{WM} = \{\mathcal{U} \subseteq \Lambda^\infty \mid \mathcal{U}$ is a set of weak meaningless terms$\}$.

It is trivial to see that if $\mathcal{U}$ satisfies either Overlap or Alternative Overlap then it satisfies Weak Overlap. The converse is also true as proved in the following theorem.

▶ **Theorem 3.3.** *Let $\mathcal{U} \subseteq \Lambda^\infty$ satisfy the Axioms of Closure under β-reduction and Indiscernibility. Then, $\mathcal{U}$ satisfies the Axiom of Weak Overlap if and only if $\mathcal{U}$ either satisfies the Axiom of Overlap or the Axiom of Alternative Overlap. Moreover, if $\mathcal{U}$ contains an abstraction then $\mathcal{U}$ cannot satisfy both the Axioms of Overlap and Alternative Overlap simultaneously.*

Proof. $\Leftarrow$ is trivial. We prove $\Rightarrow$. Suppose $\lambda x.P_1 \in \mathcal{U}$ for which we have that $(\lambda x.P_1)Q \in \mathcal{U}$ for all $Q \in \Lambda^\infty$. Therefore for any other abstraction $\lambda x.P_2 \in \mathcal{U}$ we get by Indiscernibility $(\lambda x.P_2)Q \in \mathcal{U}$ for all $Q \in \Lambda^\infty$. That is, the axiom of Overlap holds.

If, however, for no abstraction $\lambda x.P_1 \in \Lambda^\infty$ we have that $(\lambda x.P_1)Q \in \mathcal{U}$ for all $Q \in \Lambda^\infty$, then by Weak Overlap it must be that for each abstraction $\lambda x.P \in \mathcal{U}$ we have that there is some $W \in \mathcal{U}$ such that $P \twoheadrightarrow_\beta Wx$. Hence the axiom of Alternative Overlap holds.

Assume $\lambda x.P \in \mathcal{U}$. Suppose $\mathcal{U}$ satisfies the Axiom of Overlap. Then by Overlap we have $(\lambda x.P)x \in \mathcal{U}$ and hence $P \in \mathcal{U}$ by Closure under β-reduction. By Indiscernibility we find $\lambda x.\mathbf{\Omega} \in \mathcal{U}$. But there is no W such that $\mathbf{\Omega}$ reduces to Wx. Therefore $\mathcal{U}$ does not satisfy Alternative Overlap. Hence $\mathcal{U}$ cannot satisfy both axioms simultaneously. ◀

▶ **Corollary 3.4.** *1.* $\mathbb{WM} = \mathbb{M} \cup \mathbb{AM}$

2. If $\mathcal{U} \in \mathbb{M} \cap \mathbb{AM}$, then $\mathcal{U}$ does not contain any abstraction.

4 Examples of Sets of Weak Meaningless Terms

In this section, we recall some examples of sets of meaningless terms from [11, 7, 16] and give new examples of sets of weak meaningless terms.

▶ **Definition 4.1.** Let $M \in \Lambda^\infty$. We say that

1. M is a *head normal form* (hnf) if $M = \lambda x_1 \dots x_n.yP_1 \dots P_k$. We define $\mathcal{HN} = \{M \in \Lambda^\infty \mid M \twoheadrightarrow_\beta N$ and N is a head normal form$\}$.
2. M is a *weak head normal form* (whnf) if M is a hnf or $M = \lambda x.N$. We define $\mathcal{WN} = \{M \in \Lambda^\infty \mid M \twoheadrightarrow_\beta N$ and N and N is a weak head normal form$\}$.

3. M is a *top normal form* (tnf) if it is either a whnf or an application (NP) where there is no Q such that $N \twoheadrightarrow_\beta \lambda x.Q$. We define $\mathcal{TN} = \{M \in \Lambda^\infty \mid M \twoheadrightarrow_\beta N$ and N is a top normal form$\}$.
4. M is a *strong active form* (saf) if $M = RP_1 \dots P_k$ and R is rootactive. We define $\mathcal{SA} = \{M \in \Lambda^\infty \mid M \twoheadrightarrow_\beta N$ and N is a strong active form$\}$.
5. M is a *strong active form relative to* X if $M = RP_1 \dots P_k$, R is rootactive and $P_1, \dots, P_k \in X$. We define $\mathcal{SA}_X = \{M \in \Lambda^\infty \mid M \twoheadrightarrow\!\!\!\!\!\rightarrow_\beta N$ and N is a strong active form relative to $X\}$.
6. M is a *strong infinite left spine form* (silsf) if $M = (\dots P_2)P_1$. We define $\mathcal{SIL} = \{M \in \Lambda^\infty \mid M \twoheadrightarrow\!\!\!\!\!\rightarrow_\beta N$ and N is a strong infinite left spine form$\}$.
7. M is a *head active form* (haf) if $M = \lambda x_1 \dots x_n.RP_1 \dots P_k$ and R is rootactive. We define $\mathcal{HA} = \{M \in \Lambda^\infty \mid M \twoheadrightarrow_\beta N$ and N is a head active form$\}$.
8. M is an *infinite left spine form* (ilsf) if $M = \lambda x_1 \dots x_n.(\dots P_2)P_1$. We define $\mathcal{IL} = \{M \in \Lambda^\infty \mid M \twoheadrightarrow\!\!\!\!\!\rightarrow_\beta N$ and N is an infinite left spine form$\}$.
9. We define $\mathcal{O} = \{M \in \Lambda^\infty \mid M \twoheadrightarrow\!\!\!\!\!\rightarrow_\beta \mathbf{O}\}$ where $\mathbf{O} = \lambda x_1.\lambda x_2.\lambda x_3.\dots$.

By $\overline{\mathcal{HN}}$, $\overline{\mathcal{WN}}$ and $\overline{\mathcal{TN}}$ we denote the complements in Λ^∞ of $\mathcal{HN}$, $\mathcal{WN}$ and $\mathcal{TN}$ respectively. Note that $\mathcal{R} = \overline{\mathcal{TN}}$, $\overline{\mathcal{WN}} = \mathcal{SA} \cup \mathcal{SIL}$ and $\overline{\mathcal{HN}} = \mathcal{HA} \cup \mathcal{IL} \cup \mathcal{O}$.

▶ **Theorem 4.2** ([11, 7]). *$\overline{\mathcal{TN}}$, $\overline{\mathcal{WN}}$ and $\overline{\mathcal{HN}}$ are meaningless sets.*

The Berarducci tree $\mathsf{BerT}(M)$ of a term M is its normal form in $\lambda^\infty_{\beta\bot_\mathcal{U}}$ where $\mathcal{U}$ is $\mathcal{R} = \overline{\mathcal{TN}}$ [6, 8]. The Lévy-Longo tree $\mathsf{LLT}(M)$ is the normal form of M in $\lambda^\infty_{\beta\bot_\mathcal{U}}$ where $\mathcal{U}$ is $\overline{\mathcal{WN}}$ [8]. The Böhm tree $\mathsf{BT}(M)$ is the normal form of M in $\lambda^\infty_{\beta\bot_\mathcal{U}}$ where $\mathcal{U}$ is $\overline{\mathcal{HN}}$ [3].

▶ **Theorem 4.3** ([16]). *The following are meaningless sets:*

1. *$\mathcal{SA}$, $\mathcal{HA}$, $\mathcal{HA} \cup \mathcal{IL}$ and $\mathcal{HA} \cup \mathcal{O}$.*
2. *$\mathcal{SA}_X$, provided $X \subseteq \mathsf{BerT}(\Lambda^\infty_\bot) \cap (\Lambda^\infty)^0$.*

Any set of meaningless terms that does not contain abstractions such as $\mathcal{R}$ or $\mathcal{SA}$ is trivially a set of alternative meaningless terms. We will give examples of sets of alternative meaningless terms which contain abstractions and which are not sets of meaningless terms.

▶ **Definition 4.4.** Define $\mathcal{U}^\eta = \mathcal{U} \cup \{M \mid M \twoheadrightarrow\!\!\!\!\!\rightarrow_\beta \lambda x.Nx$ and $N \in \mathcal{U}\}$ for $\mathcal{U} \subseteq \Lambda^\infty$.

If $\mathcal{U}$ is a set of meaningless terms, $\mathcal{U}^\eta$ does not have to be a set of weak meaningless terms. For example, $\mathcal{SA}^\eta$ is not a set of weak meaningless terms. It does not satisfy Indiscernibility since $\mathbf{\Omega}x \in \mathcal{SA}^\eta$ is a subterm of $\lambda x.\mathbf{\Omega}x \in \mathcal{SA}^\eta$ but $\lambda x.\mathbf{\Omega} \notin \mathcal{SA}^\eta$. Similarly, $(\mathcal{SA} \cup \mathcal{SIL})^\eta$ is not a set of weak meaningless terms. On the other hand, for $\mathcal{U} \in \{\mathcal{HA}, \mathcal{HA} \cup \mathcal{IL}, \mathcal{HA} \cup \mathcal{O}, \mathcal{HA} \cup \mathcal{IL} \cup \mathcal{O}, \Lambda^\infty\}$, the set $\mathcal{U}^\eta$ is a meaningless set because $\mathcal{U}^\eta = \mathcal{U}$; but $\mathcal{U}^\eta$ is not a set of alternative meaningless terms, as it cannot be both by Corollary 3.4.

▶ **Theorem 4.5.** 1. *$\mathcal{R}^\eta$ is a set of alternative meaningless terms.*
2. *$\mathcal{SA}^\eta_X$ is a set of alternative meaningless terms, provided $X \subseteq \mathsf{BerT}(\Lambda^\infty_\bot) \cap (\Lambda^\infty)^0$.*

We skip the proof as it follows the same pattern as the proofs for meaningless sets presented in [16]. By Corollary 3.4, the sets in the above theorem are not sets of meaningless terms because they contain abstractions. Since $\{\mathcal{SA}^\eta_X \mid X \subseteq \mathsf{BerT}(\Lambda^\infty_\bot) \cap (\Lambda^\infty)^0\}$ has the cardinality $2^\mathfrak{c}$ of the continuum, we have that:

▶ **Corollary 4.6.** *Let $\mathfrak{c}$ be the cardinality of the continuum. There are $2^\mathfrak{c}$ sets of alternative meaningless terms which are not meaningless sets.*

▶ Remark. The set $\mathbb{WM}$ of all sets of weak meaningless terms forms a poset with a top, $\mathcal{R}$, and a bottom, Λ^∞. In Figure 1 we depict the relative order of the sets mentioned in this section. The notation $\mathcal{U}_1 \to \mathcal{U}_2$ indicates that $\mathcal{U}_1 \supset \mathcal{U}_2$.

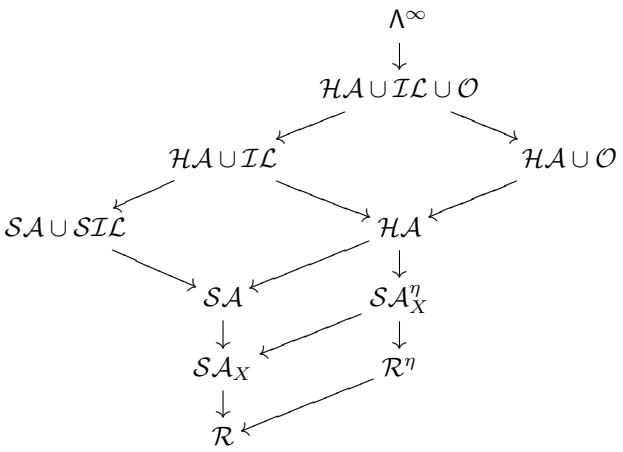


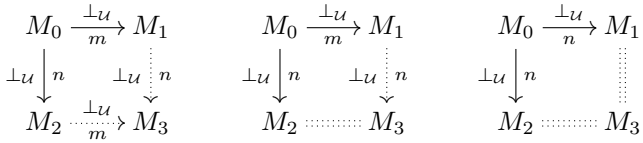
Figure 1 A poset of sets of weak meaningless terms

5 Weak Meaninglessness implies Confluence and Normalization

In this section, we prove confluence of $\lambda^\infty_{\beta\bot_\mathcal{U}}$ when $\mathcal{U}$ is a set of weak meaningless terms. This extends the result in [11, 7] where confluence of $\lambda^\infty_{\beta\bot_\mathcal{U}}$ is shown under the provision that $\mathcal{U}$ is a set of meaningless terms. First we need some auxiliary results.

▶ **Proposition 5.1.** *Let $\mathcal{U} \subseteq \Lambda^\infty$ satisfy the Axioms of Rootactiveness and of Indiscernibility. Then the calculus $\lambda^\infty_{\bot_\mathcal{U}} = (\Lambda^\infty_\bot, \to_{\bot_\mathcal{U}})$ is confluent.*

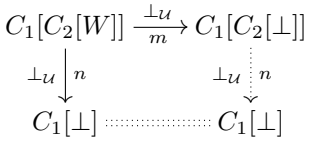
Proof. We sketch a standard transfinite inductive tiling diagram proof. The basic information that this proof uses are the following elementary tiling diagrams for one-set coinitial $\bot_\mathcal{U}$-reductions.



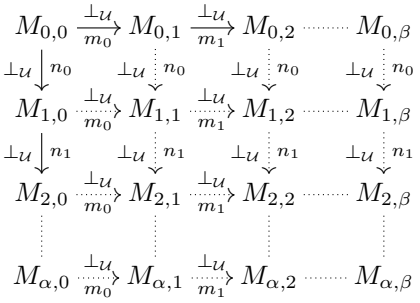
The labels n, m used in the diagrams indicate the depth at which the $\bot_\mathcal{U}$ reduction takes place. The important thing to note is that the depth of a $\bot_\mathcal{U}$-redex in a term does not change when it is not erased in the contraction of another $\bot_\mathcal{U}$-redex elsewhere in the term.

The diagrams reflect three possibilities. The redexes in the two coinitial reductions are either disjoint, properly nested or identical. In each case the respective diagram show how to complete confluence with two cofinal $\bot_\mathcal{U}$-reductions, which are either one-step or empty.

The middle diagram requires Indiscernibility. Suppose M_1 is of the form $C_1[C_2[W]]$ for contexts $C_1[\ \ ], C_2[\ \ ]$. And suppose W and $C_2[W]$ belong to $\mathcal{U}$. Then by Indiscernibility we get $C_2[\mathbf{\Omega}] \in \mathcal{U}$ and so $C_1[C_2[\bot]] \to_\bot C_1[\bot]$. This completes the diagram:



Given two transfinite coinitial $\bot_{\mathcal{U}}$-reductions one now constructs the following tiling diagram [7] inductively in which all vertical and horizontal reductions are strongly converging.



We skip the proof, which is similar to confluence proof of $\lambda^{h\infty}_{\eta}$ in [14], because the elementary tiles that load the induction are similar for $\bot_{\mathcal{U}}$ and η. The simplicity of these rules makes it unnecessary to specify the positions; the information of the depth in each step of the reduction sequence suffices. ◀

▶ **Lemma 5.2.** [11, Lemma 27] *Let $\mathcal{U} \subseteq \Lambda^{\infty}$ satisfy the Axiom of Closure under Substitution. If $M \twoheadrightarrow_{\beta\bot_{\mathcal{U}}} N$, then $M \twoheadrightarrow_{\beta} L \twoheadrightarrow_{\bot_{\mathcal{U}}} N$ for some $L \in \Lambda^{\infty}_{\bot}$.*

We need somre terminology and notation: An outermost $\bot_{\mathcal{U}}$-redex of M is a maximal subterm N of M such that $N[\bot := \mathbf{\Omega}] \in \mathcal{U}$. We denote by $M \xrightarrow{out}_{\bot_{\mathcal{U}}} N$ if the contracted redex in $M \to_{\bot_{\mathcal{U}}} N$ is an outermost $\bot_{\mathcal{U}}$-redex.

The information stored at the root of a term M is denoted by $\mathsf{root}(M)$ and defined by cases: $\mathsf{root}(x) = x$, $\mathsf{root}(\lambda x.M) = \lambda x$ and $\mathsf{root}(MN) = @$. We denote $M \sim_{\mathsf{root}} N$ if $\mathsf{root}(M) = \mathsf{root}(N)$.

▶ **Lemma 5.3.** *Let $\mathcal{U} \subseteq \Lambda^{\infty}$ satisfy the Axioms of Rootactiveness and Indiscernibility. If $M \twoheadrightarrow_{\bot_{\mathcal{U}}} N$ and N is in $\beta\bot_{\mathcal{U}}$-normal form then $M \overset{out}{\twoheadrightarrow}_{\bot_{\mathcal{U}}} N$.*

Proof. Suppose $M = M_0 \twoheadrightarrow_{\bot_{\mathcal{U}}} M_{\alpha} = N$ is a reduction of length α and N is a $\beta\bot_{\mathcal{U}}$-normal form. We define a new reduction $N_0 \overset{out}{\twoheadrightarrow}_{\bot_{\mathcal{U}}} N_{\alpha}$ by induction on α satisfying the property $\Phi(\beta)$ for all $0 \leq \beta \leq \alpha$ where $\Phi(\beta)$ is defined as follows. If $(M_{\beta})|_p = \bot$, then $(N_{\beta})|_p[\bot := \mathbf{\Omega}] \in \mathcal{U}$. Otherwise, if $(M_{\beta})|_p \neq \bot$, then $(M_{\beta})|_p \sim_{\mathsf{root}} (N_{\beta})|_p$.

Base Case. Let N_0 be equal to M.

Successor Case. Suppose we have constructed $N_0 \twoheadrightarrow_{\bot_{\mathcal{U}}} N_{\beta}$ and $\Phi(\beta)$ holds. And suppose $M_{\beta} \to_{\bot_{\mathcal{U}}} M_{\beta+1}$ by contraction of a term of $\mathcal{U}$ at position p in M_{β}. If the subterm at p in M_{β} is an outermost $\bot_{\mathcal{U}}$-redex, we construct $N_{\beta} \xrightarrow{out}_{\bot_{\mathcal{U}}} N_{\beta+1}$ by reducing the corresponding term at p in N_{β} to $\bot$. And if it is not an outermost $\bot_{\mathcal{U}}$-redex, we put $N_{\beta+1} = N_{\beta}$. It is not difficult to prove $\Phi(\beta+1)$ using Indiscernibility.

Note that the constructed reduction sequence is strongly convergent because the original sequence $M = M_0 \twoheadrightarrow_{\bot_{\mathcal{U}}} M_{\alpha} = N$ is strongly convergent.

Limit Case. Since the constructed reduction sequences $N_0 \overset{out}{\twoheadrightarrow}_{\bot_{\mathcal{U}}} N_{\beta}$ are strongly convergent, the limit λ always exists. It is not difficult to prove $\Phi(\lambda)$ using strong convergence of the reduction and induction hypothesis, i.e. $\Phi(\beta)$ holds for all $\beta < \lambda$.

Thus we have constructed $M_0 = N_0 \overset{out}{\twoheadrightarrow}_{\bot_{\mathcal{U}}} N_{\alpha}$ satisfying Φ. Since M_{α} is a $\bot_{\mathcal{U}}$-normal form, the $\bot$'s remaining in M_{α} have been introduced by an outermost $\bot$-reduction. Hence we find $\bot$'s at the same location in N_{α}. By $\Phi(\alpha)$ we get that at the other positions p, $(M_{\alpha})|_p \sim_{\mathsf{root}} (N_{\alpha})|_p$. That is $M_{\alpha} = N_{\alpha}$. Hence $M_0 = N_0 \overset{out}{\twoheadrightarrow}_{\bot_{\mathcal{U}}} N_{\alpha} = M_{\alpha}$. ◀

▶ **Lemma 5.4.** *Let $\mathcal{U} \subseteq \Lambda^\infty$ satisfy the Axiom of Rootactiveness, Closure under Substitution, β-reduction and Indiscernibility. If $W \in \mathcal{U}$ then $\mathsf{nf}_\mathcal{R}(W)[\bot := \mathbf{\Omega}] \in \mathcal{U}$.*

Proof. We have $W \twoheadrightarrow_{\beta\bot\mathcal{R}} \mathsf{nf}_\mathcal{R}(W)$. By Lemma 5.2 and Closure under Substitution, there exists W_0 such that $W \twoheadrightarrow_\beta W_0 \twoheadrightarrow_{\bot\mathcal{R}} \mathsf{nf}_\mathcal{R}(W)$. By Closure under β-reduction, $W_0 \in \mathcal{U}$. We can assume that the $\bot$-steps in $W_0 \twoheadrightarrow_{\bot\mathcal{R}} \mathsf{nf}_\mathcal{R}(W_0)$ all contract outermost redexes by Lemma 5.3. Then W_0 is obtained from $\mathsf{nf}_\mathcal{R}(W_0)$ by replacing rootactive subterms by $\bot$. Since $\mathcal{R} \subseteq \mathcal{U}$, we have that $W_0 \overset{\mathcal{U}}{\longleftrightarrow} \mathsf{nf}_\mathcal{R}(W_0)[\bot := \mathbf{\Omega}]$. By Indiscernibility, $\mathsf{nf}_\mathcal{R}(W_0)[\bot := \mathbf{\Omega}] \in \mathcal{U}$. ◀

▶ **Proposition 5.5.** *Let $\mathcal{U} \subseteq \Lambda^\infty$ be a set of alternative meaningless terms. If $L \twoheadrightarrow_{\bot\mathcal{U}} N$ and N is a $\beta\bot\mathcal{U}$-normal form, then $\mathsf{nf}_\mathcal{R}(L) \twoheadrightarrow_{\bot\mathcal{U}} N$.*

Proof. If $L \twoheadrightarrow_{\bot\mathcal{U}} N$ and N is a $\beta\bot\mathcal{U}$ normal form. By Lemma 5.3 we may assume that $L \overset{out}{\twoheadrightarrow}_{\bot\mathcal{U}} N$. Since N is a $\beta\bot\mathcal{U}$ normal form, if a β-redex $(\lambda x.P)Q$ occurs in L, then either $(\lambda x.P)[\bot := \mathbf{\Omega}] \in \mathcal{U}$ or $(\lambda x.P)Q$ is contained in some subterm W of L such that $W[\bot := \mathbf{\Omega}] \in \mathcal{U}$. We consider the following set:

$$\mathcal{W} = \{W \mid W \text{ is a maximal subterm of } L \text{ such that } W[\bot := \mathbf{\Omega}] \in \mathcal{U}\}$$

We enumerate $\mathcal{W}$ in the order of the increasing depth of its subterms, and subterms with the same depth we order them from left to right. We obtain either $\mathcal{W} = \{W_n \mid n \leq k\}$ if $\mathcal{W}$ is finite or $\mathcal{W} = \{W_n \mid n \in \mathbb{N}\}$ if $\mathcal{W}$ is infinite. We will define inductively a $\beta\bot\mathcal{R}$ reduction $L = L_0 \twoheadrightarrow_{\beta\bot\mathcal{R}} L_1 \twoheadrightarrow_{\beta\bot\mathcal{R}} L_2 \twoheadrightarrow_{\beta\bot\mathcal{R}} \dots$ with a segment $L_{n-1} \twoheadrightarrow_{\beta\bot\mathcal{R}} L_n$ for each $W_n \in \mathcal{W}$. We will show by induction on n that the following properties hold for all n,

(A) Let $L = C[W_1, \dots, W_n]$. Then $L_n = C[W'_1, \dots, W'_n]$ and $W'_m[\bot := \mathbf{\Omega}] \in \mathcal{U}$ for all $m \leq n$.

(B) The new terms $W'_1, \dots W'_n$ of L_n are in $\beta\bot\mathcal{R}$-normal form. If for some i, W'_i is an abstraction, then it does not overlap a β-redex, i.e. W'_i does not occur in an application W'_iQ of L_n.

Suppose we have constructed $L = L_0 \twoheadrightarrow_{\beta\bot\mathcal{R}} L_{n-1}$ and W_n is the next maximal meaningless subterm of $\mathcal{W}$ to be considered. Let $L = C[W_1, \dots, W_{n-1}, W_n]$. The context obtained from C instantiating the last hole with W_n has $n-1$ holes. By Induction Hypothesis, $L_{n-1} = C[W'_1, \dots, W'_{n-1}, W_n]$ and $W'_m[\bot := \mathbf{\Omega}] \in \mathcal{U}$ for all $m \leq n-1$. In particular, W_n is a subterm of L_{n-1}. We have two possibilities:

(1) $\mathsf{nf}_\mathcal{R}(W_n) = \lambda x.P$ and W_n occurs in an application W_nQ of L. By Lemma 5.4, $\lambda x.P[\bot := \mathbf{\Omega}] \in \mathcal{U}$ because $W_n \in \mathcal{U}$. By Alternative Overlap $P[\bot := \mathbf{\Omega}] = P_0x$ with $P_0 \in \mathcal{U}$. Since W_n is maximal and it occurs in L_{n-1}, we have $L_{n-1} = C_n[W_nQ_n]$. We extend $L = L_0 \twoheadrightarrow_{\beta\bot\mathcal{R}} L_{n-1}$ with

$$L_{n-1} = C_n[W_nQ_n]] \twoheadrightarrow_{\beta\bot\mathcal{R}} C_n[\mathsf{nf}_\mathcal{R}(P_0)Q_n] = L_n$$

Proof of (A). Now, $L_n = C[W'_1, \dots, W'_{n-1}, W'_n]$ where $W'_n[\bot := \mathbf{\Omega}] = P_0 \in \mathcal{U}$.
Proof of (B). Since P is in $\beta\bot\mathcal{R}$-normal form and $P[\bot := \mathbf{\Omega}] = P_0x$, we have $P = \mathsf{nf}_\mathcal{R}(P) = \mathsf{nf}_\mathcal{R}(P_0)x$. Hence $\mathsf{nf}_\mathcal{R}(P_0)$ cannot be an abstraction and $\mathsf{nf}_\mathcal{R}(P_0)Q_n$ is not a β-redex and there are no β-redexes in $W'_n = \mathsf{nf}_\mathcal{R}(P_0)$ either.

(2) Otherwise, i.e. either $\mathsf{nf}_\mathcal{R}(W_n)$ is not an abstraction or it is an abstraction and it does not occur in an application $\mathsf{nf}_\mathcal{R}(W_n)Q$ in L. Since W_n is a subterm of L_{n-1}, we have $L_{n-1} = C_n[W_n]$. Then, we extend $L = L_0 \twoheadrightarrow_{\beta\bot\mathcal{R}} L_{n-1}$ with

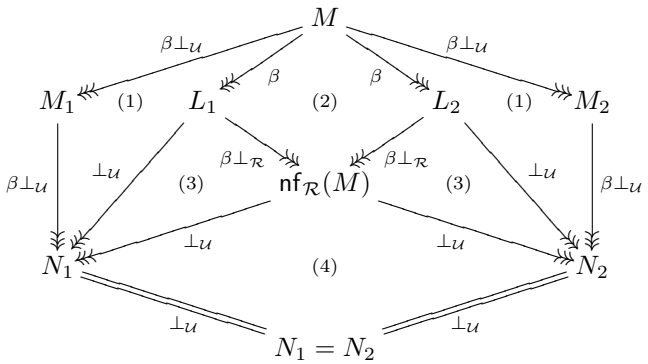
$$L_{n-1} = C_n[W_n] \twoheadrightarrow_{\beta\bot\mathcal{R}} C_n[\mathsf{nf}_\mathcal{R}(W_n)] = L_n$$

Proof of A). Now, $L_n = C[W'_1, \ldots, W'_{n-1}, W'_n]$ where $W'_n[\bot := \mathbf{\Omega}] = \mathsf{nf}_{\mathcal{R}}(W_n)[\bot := \mathbf{\Omega}] \in \mathcal{U}$ by Lemma 5.4.
Proof of (B). By Induction Hypothesis, if a β-redex occurs in L_{n-1} then it should occur inside or overlap some of its subterms in $\{W_m \mid m > n-1\} \subseteq \mathcal{W}$. We replaced W_n by $\mathsf{nf}_{\mathcal{R}}(W_n)$ which does not have any β-redex and it cannot overlap a β-redex in L_n. If a β-redex occurs in L_n then it should occur inside or overlap some of its subterms in $\{W_m \mid m > n\} \subseteq \mathcal{W}$.

The concatenation of all these strongly converging reductions is strongly converging [9]. Let K the last term L_k or (if $\mathcal{W}$ is finite) the limit L_ω of $L = L_0 \twoheadrightarrow_{\beta\bot_{\mathcal{R}}} L_1 \twoheadrightarrow_{\beta\bot_{\mathcal{R}}} L_2 \twoheadrightarrow_{\beta\bot_{\mathcal{R}}} \ldots$. It follows from (A) and strong convergence that K is obtained from L by replacing its subterms in $\mathcal{W} \subseteq \mathcal{U}$ by other subterms in $\mathcal{U}$, and hence, we have $K \twoheadrightarrow_{\bot} N$. It follows from (A,B) and strong convergence that K is obtained from L by replacing all its maximal subterms of $\mathcal{U}$ by $\beta\bot_{\mathcal{R}}$-normal forms and if some subterm $W \in \mathcal{W}$ is an abstraction then it does not overlap with a β-redex. Therefore K is a $\beta\bot_{\mathcal{R}}$ normal form and by Confluence of $\lambda^{\infty}_{\beta\bot_{\mathcal{R}}}$ (Theorems 2.16 and 4.2), we have $K = \mathsf{nf}_{\mathcal{R}}(L)$. ◀

▶ **Theorem 5.6** (Sufficiency of Alternative Meaninglessness for Confluence). *Let $\mathcal{U}$ be a set of alternative meaningless terms. Then, $\lambda^{\infty}_{\beta\bot_{\mathcal{U}}}$ is confluent.*

Proof. The proof is sketched in the following diagram.



Suppose we have a divergence $M_1 \ {}_{\beta\bot}\!\!\twoheadleftarrow M \twoheadrightarrow_{\beta\bot} M_2$. By Rootactiveness for $\mathcal{U}$, we can reduce M_1 and M_2 further to their respective $\beta\bot_{\mathcal{U}}$-normal forms N_1 and N_2 by Theorem 2.15. (1) By Closure under substitution for $\mathcal{U}$ and Lemma 5.2 we find L_1 and L_2 such that $M \twoheadrightarrow_{\beta} L_1 \twoheadrightarrow_{\bot_{\mathcal{U}}} N_1$ and $M \twoheadrightarrow_{\beta} L_2 \twoheadrightarrow_{\bot_{\mathcal{U}}} N_2$. (2) By normalization and confluence of $\lambda^{\infty}_{\beta\bot_{\mathcal{R}}}$ we construct the reductions $L_1 \twoheadrightarrow_{\beta\bot_{\mathcal{U}}} \mathsf{nf}_{\mathcal{R}}(M)$ and $L_2 \twoheadrightarrow_{\beta\bot_{\mathcal{U}}} \mathsf{nf}_{\mathcal{R}}(M)$. (3) By Proposition 5.5 we then find the reductions $\mathsf{nf}_{\mathcal{R}}(L_1) \twoheadrightarrow_{\bot_{\mathcal{U}}} N_1$ and $\mathsf{nf}_{\mathcal{R}}(L_2) \twoheadrightarrow_{\bot_{\mathcal{U}}} N_2$. By normalization and confluence of $\lambda^{\infty}_{\beta\bot_{\mathcal{R}}}$, we have $\mathsf{nf}_{\mathcal{R}}(M) = \mathsf{nf}_{\mathcal{R}}(L_1) = \mathsf{nf}_{\mathcal{R}}(L_2)$. (4) Finally Proposition 5.1 on confluence of $\bot_{\mathcal{U}}$ and the fact that N_1 and N_2 are by construction normal forms for $\bot_{\mathcal{U}}$-reduction implies that N_1 and N_2 are identical. ◀

▶ **Corollary 5.7** (Sufficiency of Weak Meaninglessness for Confluence and Normalization). *Let $\mathcal{U}$ be a set of weak meaningless terms. Then, $\lambda^{\infty}_{\beta\bot_{\mathcal{U}}}$ is confluent and normalizing.*

Proof. Immediate from Theorems 5.6 and 2.16, and Corollary 3.4. ◀

By Theorem 4.5 and Corollary 5.7, the Infinitary lambda calculi $\lambda^{\infty}_{\beta\bot_{\mathcal{U}}}$ where $\mathcal{U} \in \{\mathcal{SA}^{\eta}_X \mid X \subseteq \mathsf{BerT}(\Lambda) \cap (\Lambda^{\infty})^0\}$ are confluent and normalizing. By Theorem 2.17, they all induce different models of the finite lambda calculus. Since $\{\mathcal{SA}^{\eta}_X \mid X \subseteq \mathsf{BerT}(\Lambda) \cap (\Lambda^{\infty})^0\}$ has cardinality 2^{ω}, we have that:

▶ **Corollary 5.8.** *There are 2^ω different models of the finite lambda calculus such that:*
1. *$\lambda x.\mathbf{\Omega}x = \mathbf{\Omega}$ but $\mathbf{I} \neq \mathbf{1}$,*
2. *$M = N$ if $\mathsf{BerT}(M) = \mathsf{BerT}(N)$ and*
3. *$\mathbf{\Omega}M = \mathbf{\Omega}$ for some $M \in \Lambda$ such that $\mathsf{BerT}(M) \in (\Lambda^\infty)^0$.*

6 Axioms of Closure under Expansion

In this section, we define two axioms: Closure under β-expansions [10] and Closure under $\beta\bot$-expansion from $\bot$. In the cited paper we introduced Closure under β-expansion to obtain ω-compression. In this paper we use the axiom to show that an arbitrary weak meaningless set $\mathcal{U} \subseteq \Lambda^\infty$ and its β-expansion $\overline{\mathcal{U}}$ determine the same lambda models.

▶ **Definition 6.1.** We define the following axioms on a set $\mathcal{U} \subseteq \Lambda^\infty$.

1. We say that $\mathcal{U}$ satisfies the Axiom of Closure under β-expansion if for all $N \in \mathcal{U}$, if $M \twoheadrightarrow_\beta N$ then $M \in \mathcal{U}$.
2. We say that $\mathcal{U}$ satisfies the Axiom of Closure under $\beta\bot$-expansion from $\bot$ if for all $M \in \Lambda^\infty$, if $M \twoheadrightarrow_{\beta\bot_\mathcal{U}} \bot$ then $M \in \mathcal{U}$.
3. $\mathbb{B} = \{\mathcal{U} \subseteq \Lambda^\infty \mid \mathcal{U}$ satisfies the Axiom of Closure under $\beta\bot$-expansion from $\bot\}$.

▶ Remark. Note that if $\mathcal{U}$ satisfies Axiom of Closure under $\beta\bot$-expansion from $\bot$ then $\mathcal{U} = \{M \in \Lambda^\infty \mid M \twoheadrightarrow_{\beta\bot_\mathcal{U}} \bot\}$, i.e. $\mathcal{U}$ is the set of $\beta\bot$-expansions of $\bot$. We also have that $\mathcal{U}$ satisfies Closure under β-expansion.

▶ Remark. All sets of weak meaningless terms of Figure 1, have been defined to satisfy Closure under β-expansion to facilitate the proof of Indiscernibility. If a set $\mathcal{U}$ satisfies Rootactiveness and Indiscernibility, then $\mathcal{U}$ is closed under certain β-expansions. Since the set $\mathcal{R}$ is closed under β-expansions, we have that, for example, $\mathbf{I\Omega} \in \mathcal{R}$. By Indiscernibility, if $M \in \mathcal{U}$ then $\mathbf{I}M$ should also belong to $\mathcal{U}$.

▶ Remark. All examples of sets of (weak) meaningless terms given in Section 4 also satisfy Closure under $\beta\bot$-expansion from $\bot$. Suppose $M \twoheadrightarrow_{\beta\bot_\mathcal{U}} \bot$. By Closure under Substitution, by Lemma 5.2, there is an N with $M \twoheadrightarrow_\beta N \twoheadrightarrow_{\bot_\mathcal{U}} \bot$. Hence $N \overset{out}{\twoheadrightarrow}_{\bot_\mathcal{U}} \bot$ by Lemma 5.3. Then N reduces in one step to $\bot$ so that $N \in \mathcal{U}$. Closure under β-expansion implies $M \in \mathcal{U}$.

Given a set $\mathcal{U}$, we can always extend it to a set $\overline{\mathcal{U}}$ that satisfies Closure under $\beta\bot$-expansion from $\bot$ by taking: $\overline{\mathcal{U}} = \{M \in \Lambda^\infty \mid M \twoheadrightarrow_{\beta\bot_\mathcal{U}} \bot\}$. We have that $\mathcal{U}$ and $\overline{\mathcal{U}}$ define the same reduction:

▶ **Theorem 6.2** (Same Reduction)**.** *Let $M, N \in \Lambda^\infty_\bot$ and $\mathcal{U} \subseteq \Lambda^\infty$. Then, $M \twoheadrightarrow_{\beta\bot_\mathcal{U}} N$ if and only if $M \twoheadrightarrow_{\beta\bot_{\overline{\mathcal{U}}}} N$.*

Proof. Let $M = C[P] \to_{\bot_{\overline{\mathcal{U}}}} C[\bot] = N$ where $P \in \overline{\mathcal{U}}$. Then $P \twoheadrightarrow_{\beta\bot_\mathcal{U}} \bot$. Hence, $M \twoheadrightarrow_{\beta\bot_\mathcal{U}} C[\bot] = N$. The converse is trivial. ◀

We define the equivalence relation $\mathcal{U} \sim \mathcal{U}'$ if $\twoheadrightarrow_{\beta\bot_\mathcal{U}}$ and $\twoheadrightarrow_{\beta\bot_{\mathcal{U}'}}$ are equal. Then, every $\sim$-equivalence class $[\mathcal{U}]$ has a unique canonical representative obtained by taking the union of all the members of the class, i.e. $\overline{\mathcal{U}} = \bigcup[U]$.

▶ **Corollary 6.3** (Same Normal Form)**.** *Let $\mathcal{U} \subseteq \Lambda^\infty$.*

1. *$\lambda^\infty_{\beta\bot_\mathcal{U}}$ is confluent (normalizing) if and only if $\lambda^\infty_{\beta\bot_{\overline{\mathcal{U}}}}$ is confluent (normalizing).*
2. *Let $\lambda^\infty_{\beta\bot_\mathcal{U}}$ be confluent and normalizing. Then, $\mathsf{nf}_\mathcal{U} = \mathsf{nf}_{\overline{\mathcal{U}}}$.*

3. Let $\lambda^\infty_{\beta\bot_{\mathcal{U}_1}}$ *and* $\lambda^\infty_{\beta\bot_{\mathcal{U}_2}}$ *be confluent and normalizing. Then,* $\overline{\mathcal{U}_1} = \overline{\mathcal{U}_2}$ *iff* $\mathsf{nf}_{\mathcal{U}_1} = \mathsf{nf}_{\mathcal{U}_2}$.

We say that two models are equal, i.e. $\mathfrak{M}_{\mathcal{U}_1} = \mathfrak{M}_{\mathcal{U}_2}$, if they have the same domain and their interpretation functions are equal. As an immediate consequence of the previous corollary and Theorem 2.17, we have that $\mathcal{U}$ and $\overline{\mathcal{U}}$ define the same model:

▶ **Corollary 6.4** (Same Model). *Let* $\mathfrak{M}_{\mathcal{U}_1}, \mathfrak{M}_{\mathcal{U}_1} \in \mathbb{MOD}(\lambda)$. *Then,* $\overline{\mathcal{U}_1} = \overline{\mathcal{U}_2}$ *iff* $\mathfrak{M}_{\mathcal{U}_1} = \mathfrak{M}_{\mathcal{U}_2}$.

7 Confluence implies Normalization

In this section, we prove that if $\lambda^\infty_{\beta\bot_{\mathcal{U}}}$ is confluent then $\mathcal{U}$ satisfies the Axiom of Rootactiveness provided that $\mathcal{U}$ is the set of expansions of $\bot$. As a corollary, we conclude that confluence of $\lambda^\infty_{\beta\bot_{\mathcal{U}}}$ implies normalization of $\lambda^\infty_{\beta\bot_{\mathcal{U}}}$.

▶ **Definition 7.1.** For any $M \in \Lambda^\infty_\bot$, let $M^{\mathbf{I}}$ be the result of replacing every application PQ in M by $\mathbf{I}(PQ)$.

For example, $\mathbf{\Omega}^{\mathbf{I}} = \mathbf{I}((\lambda x.\mathbf{I}(xx))(\lambda x.\mathbf{I}(xx)))$.

▶ **Lemma 7.2.** *1.* $(P[x := Q])^{\mathbf{I}} = P^{\mathbf{I}}[x := Q^{\mathbf{I}}]$.
2. If $M \to_\beta N$ *then* $M^{\mathbf{I}} \twoheadrightarrow_\beta N^{\mathbf{I}}$.
3. If $M \twoheadrightarrow_\beta (\lambda x.P)Q$ *then* $M^{\mathbf{I}} \twoheadrightarrow_\beta \mathbf{I}(P[x := Q])^{\mathbf{I}}$.

Proof. We prove Part 2. Suppose $M = (\lambda x.P)Q \to_\beta P[x := Q]$. Using Part 1, we have that $M^{\mathbf{I}} = \mathbf{I}((\lambda x.P^{\mathbf{I}})Q^{\mathbf{I}}) \to_\beta \mathbf{I}(P^{\mathbf{I}}[x := Q^{\mathbf{I}}]) = \mathbf{I}(P[x := Q])^{\mathbf{I}} \to_\beta (P[x := Q])^{\mathbf{I}}$.

We prove Part 3. Suppose $M \twoheadrightarrow_\beta (\lambda x.P)Q$. Using Parts 1 and 2, we have $M^{\mathbf{I}} \twoheadrightarrow_\beta ((\lambda x.P)Q)^{\mathbf{I}} = \mathbf{I}((\lambda x.P^{\mathbf{I}})Q^{\mathbf{I}}) \to_\beta \mathbf{I}(P^{\mathbf{I}}[x := Q^{\mathbf{I}}]) = \mathbf{I}(P[x := Q])^{\mathbf{I}}$. ◀

▶ **Lemma 7.3.** *For any* $M \in \mathcal{R}$, $M^{\mathbf{I}}$ *reduces both to* M *and* $\mathbf{I}^\omega$.

Proof. It is easy to show that $M^{\mathbf{I}} \twoheadrightarrow\!\!\!\!\!\to_\beta M$ for all $M \in \Lambda^\infty_\bot$. Since M is rootactive, there is an infinite reduction starting for M containing infinitely many root reduction steps, i.e. $M \twoheadrightarrow_\beta (\lambda x.P_0)Q_0 \to_\beta P_0[x := Q_0] \twoheadrightarrow_\beta (\lambda x.P_1)Q_1 \to_\beta P_1[x := Q_2] \ldots$. Applying Lemma 7.2 Parts 2 and 3, we can construct the following reduction sequence.

$$M^{\mathbf{I}} \twoheadrightarrow_\beta ((\lambda x.P_0)Q_0)^{\mathbf{I}} \to_\beta \mathbf{I}(P_0[x := Q_0])^{\mathbf{I}} \twoheadrightarrow_\beta \mathbf{I}((\lambda x.P_1)Q_1)^{\mathbf{I}} \to_\beta \mathbf{I}(\mathbf{I}(P_1^{\mathbf{I}}[x := Q_1^{\mathbf{I}}])) \ldots$$

The limit of the above sequence is $\mathbf{I}^\omega$. ◀

▶ **Theorem 7.4** (Necessity of Rootactiveness for Confluence). *Let* $\mathcal{U} \subseteq \Lambda^\infty$ *satisfy Closure under* $\beta\bot$*-expansion from* $\bot$. *If* $\lambda^\infty_{\beta\bot_{\mathcal{U}}}$ *is confluent then* $\mathcal{U}$ *satisfies Rootactiveness.*

Proof. We prove that $\mathcal{U}$ satisfies the Axiom of Rootactiveness. By Lemma 7.3, $\mathbf{\Omega}^{\mathbf{I}} \twoheadrightarrow\!\!\!\!\!\to_\beta \mathbf{I}^\omega$ and $\mathbf{\Omega}^{\mathbf{I}} \twoheadrightarrow\!\!\!\!\!\to_\beta \mathbf{\Omega}$. Since $\lambda^\infty_{\beta\bot_{\mathcal{U}}}$ is confluent, there exists P such that $\mathbf{I}^\omega \twoheadrightarrow\!\!\!\!\!\to_{\beta\bot} P$ and $\mathbf{\Omega} \twoheadrightarrow\!\!\!\!\!\to_{\beta\bot} P$. Since $\mathbf{\Omega}$ only β-reduces to itself, we have that $\mathbf{\Omega} \to_\bot Q \twoheadrightarrow\!\!\!\!\!\to_{\beta\bot} P$. Hence, $\mathbf{\Omega} = C[M] \to_\bot C[\bot] = Q$ for $M \in \mathcal{U}$. Suppose M is a proper subterm of Ω. We have the following cases.

1. Case $M = x$. Then $x[x := P] \to_\bot \bot$ and $P \in \mathcal{U}$ for all $P \in \Lambda^\infty$. In particular, $\mathbf{\Omega} \in \mathcal{U}$.
2. Case $M = xx$. Then $xx[x := \lambda x.xx] \to_\bot \bot$. Hence, $\mathbf{\Omega} \in \mathcal{U}$.
3. Case $M = \lambda x.xx$. Hence $\mathbf{\Omega} \twoheadrightarrow_\bot \bot\bot$ and also $\mathbf{I}^\omega \twoheadrightarrow\!\!\!\!\!\to_{\beta\bot} \bot\bot$. Since $\mathbf{I}^\omega$ can only β-reduce to itself, $\mathbf{I}^\omega = C'[N] \to_\bot C[\bot] \twoheadrightarrow\!\!\!\!\!\to_{\beta\bot} \bot\bot$. Suppose N is a proper subterm of $\mathbf{I}^\omega$. There are two possibilities:
   a. Case $N = \mathbf{I}^\omega$. Then $\mathbf{\Omega} \twoheadrightarrow\!\!\!\!\!\to_{\beta\bot} \bot$ and $\mathbf{\Omega} \in \mathcal{U}$.

b. Case $N = \mathbf{I}$. Then, $\mathbf{I}^\omega \twoheadrightarrow_\bot \bot^\omega = \bot(\bot(\bot\ldots))$. On the other hand, $\mathbf{I}^\omega \twoheadrightarrow_{\beta\bot} \bot\bot$. This is possible only if $\bot^\omega \to_\bot \bot$. Hence, $\mathbf{\Omega} \to_\bot \bot$ and $\mathbf{\Omega} \in \mathcal{U}$.

Hence, $Q = \bot = P$ and also $\mathbf{I}^\omega \twoheadrightarrow_{\beta\bot} \bot$. By Lemma 7.3, for any $M \in \mathcal{R}$, $M^{\mathbf{I}} \twoheadrightarrow_\beta M$ and $M \twoheadrightarrow_\beta \mathbf{I}^\omega$. Since $\lambda^\infty_{\beta\bot_\mathcal{U}}$ is confluent and $\mathbf{I}^\omega \to_\bot \bot$ we have $M \twoheadrightarrow_{\beta\bot} \bot$. Since $\mathcal{U}$ is the set of expansions of $\bot$, we have $M \in \mathcal{U}$. ◀

▶ **Corollary 7.5** (Confluence implies Normalization). *If $\lambda^\infty_{\beta\bot_\mathcal{U}}$ is confluent then $\lambda^\infty_{\beta\bot_\mathcal{U}}$ is normalizing.*

Proof. Let $\lambda^\infty_{\beta\bot_\mathcal{U}}$ be confluent. By Corollary 6.3, $\lambda^\infty_{\beta\bot_{\overline{\mathcal{U}}}}$ is confluent. By Theorem 7.4, $\overline{\mathcal{U}}$ satisfies Rootactiveness. By Theorem 2.15 and Corollary 6.3 $\lambda^\infty_{\beta\bot_{\overline{\mathcal{U}}}}$ and $\lambda^\infty_{\beta\bot_\mathcal{U}}$ are normalizing. ◀

As a consequence of the previous corollary, if the infinitary lambda calculus $\lambda^\infty_{\beta\bot_\mathcal{U}}$ is confluent then it induces a λ-model (see Theorem 2.17).

8 Confluence implies Weak Meaninglessness

In Section 5, we proved that if $\mathcal{U}$ is a set of weak meaningless terms then $\lambda^\infty_{\beta\bot}$ is confluent (Theorem 5.6). In this section, we study whether the converse holds. We will prove that confluence of $\lambda^\infty_{\beta\bot_\mathcal{U}}$ implies that $\overline{\mathcal{U}}$ is a set of weak meaningless terms. In other words, if $\lambda^\infty_{\beta\bot_\mathcal{U}}$ is confluent then there exists a set $\mathcal{U}'$ of weak meaningless terms that defines the same reduction as $\mathcal{U}$.

▶ **Theorem 8.1** (Necessity of Weak Meaninglessness for Confluence I). *Let $\mathcal{U} \subseteq \Lambda^\infty$ satisfy Closure under $\beta\bot$-expansion from $\bot$. If $\lambda^\infty_{\beta\bot_\mathcal{U}}$ is confluent then $\mathcal{U}$ is a set of weak meaningless terms.*

Proof. Suppose $\lambda^\infty_{\beta\bot}$ is confluent. Rootactiveness of $\mathcal{U}$ follows from Theorem 7.4. We prove that $\mathcal{U}$ satisfies the remaining axioms:

We prove that $\mathcal{U}$ satisfies Indiscernibility. Suppose $M \overset{\mathcal{U}}{\longleftrightarrow} N$. It is not difficult to show that there exists P such that $M \twoheadrightarrow_\bot P$ and $N \twoheadrightarrow_\bot P$. If $M \in \mathcal{U}$ then $M \to_\bot \bot$. Since $\lambda^\infty_\bot$ is confluent, we have $N \twoheadrightarrow_{\beta\bot} \bot$. By Closure under $\beta\bot$-expansion from $\bot$, we get $N \in \mathcal{U}$.

We prove that $\mathcal{U}$ satisfies Closure under Substitution. Let $P \in \mathcal{U}$ and $Q \in \Lambda^\infty$. We will prove $P[x := Q] \in \mathcal{U}$. Since $P \in \mathcal{U}$, we have $(\lambda x.P)Q \to_\bot (\lambda x.\bot)Q \to_\beta \bot$. We also have $(\lambda x.P)Q \to_\beta P[x := Q]$. Since $\lambda^\infty_{\beta\bot_\mathcal{U}}$ is confluent, $P[x := Q] \twoheadrightarrow_{\beta\bot} \bot$. By Closure under $\beta\bot$-expansion from $\bot$, we have $P[x := Q] \in \mathcal{U}$.

We prove that $\mathcal{U}$ satisfies Closure under β-reduction. If $M \twoheadrightarrow_\beta N$ and $M \in \mathcal{U}$ then $M \to_\bot \bot$. By Confluence, $N \twoheadrightarrow_{\beta\bot} \bot$. By Closure under $\beta\bot$-expansion from $\bot$, we find $N \in \mathcal{U}$.

Finally, we prove that $\mathcal{U}$ satisfies Weak Overlap. If $\lambda x.P \in \mathcal{U}$ then $(\lambda x.P)x \to_\bot \bot x$ and $(\lambda x.P)x \to_\beta P$. Since $\lambda^\infty_{\beta\bot_\mathcal{U}}$ is confluent, there exists N such that $P \twoheadrightarrow_{\beta\bot} N$ and $\bot x \twoheadrightarrow_{\beta\bot} N$. We have two possibilities:

1. $N = \bot x$. Then $P \twoheadrightarrow_{\beta\bot} \bot x$. By Theorem 5.2, we have that $P \twoheadrightarrow_\beta P' \twoheadrightarrow_\bot \bot x$ for some $P' \in \Lambda^\infty_\bot$. Then, $P' = Wx$ and $W \twoheadrightarrow_\bot \bot$. By Closure under $\beta\bot$-expansion from $\bot$, we have $W \in \mathcal{U}$. Trivially, $W \in \Lambda^\infty$ because $P \in \Lambda^\infty$ and $P \twoheadrightarrow_\beta Wx$.
2. $N = \bot$. Then, $(\lambda x.P)x \to_\bot P \twoheadrightarrow_{\beta\bot} \bot$. By Closure under $\beta\bot$-expansion from $\bot$, we have that $(\lambda x.P)x \in \mathcal{U}$. By Closure under Substitutions, $(\lambda x.P)Q \in \mathcal{U}$ for all $Q \in \Lambda^\infty$.

◀

▶ **Corollary 8.2** (Necessity of Weak Meaninglessness for Confluence II). *If $\lambda^{\infty}_{\beta\bot_{\mathcal{U}}}$ is confluent then there exists a set $\mathcal{U}'$ of weak meaningless terms that defines the same reduction as $\mathcal{U}$.*

Proof. By Theorem 6.2, $\mathcal{U}$ and $\overline{\mathcal{U}}$ define the same reduction. By Corollary 6.3, $\lambda^{\infty}_{\beta\bot_{\overline{\mathcal{U}}}}$ is confluent. By Theorem 8.1, $\overline{\mathcal{U}}$ is a set of weak meaningless terms. ◀

The following corollary can also be proved directly.

▶ **Corollary 8.3.** *If $\mathcal{U}$ is a set of weak meaningless terms then so is $\overline{\mathcal{U}}$.*

Proof. Let $\mathcal{U}$ be a set of weak meaningless terms. By Corollary 5.7, $\lambda^{\infty}_{\beta\bot_{\mathcal{U}}}$ is confluent and normalizing. By Corollary 6.3, we have that $\lambda^{\infty}_{\beta\bot_{\overline{\mathcal{U}}}}$ is confluent and normalizing. By Theorem 8.1, $\overline{\mathcal{U}}$ is a set of weak meaningless terms. ◀

▶ **Corollary 8.4.** $\mathbb{MOD}(\lambda) = \{\mathfrak{M}_{\mathcal{U}} \mid \mathcal{U} \in \mathbb{WM} \cap \mathbb{B}\} = \{\mathfrak{M}_{\mathcal{U}} \mid \mathcal{U} \in \mathbb{WM}\}$.

Proof. We first prove $\mathbb{MOD}(\lambda) \subseteq \{\mathfrak{M}_{\mathcal{U}} \mid \mathcal{U} \in \mathbb{WM} \cap \mathbb{B}\}$. Let $\mathfrak{M}_{\mathcal{U}} \in \mathbb{MOD}(\lambda)$. By Corollary 6.3, if $\lambda^{\infty}_{\beta\bot_{\mathcal{U}}}$ is confluent and normalizing, so is $\lambda^{\infty}_{\beta\bot_{\overline{\mathcal{U}}}}$. By Corollary 6.4, $\mathfrak{M}_{\mathcal{U}} = \mathfrak{M}_{\overline{\mathcal{U}}}$. By Theorem 8.1, $\overline{\mathcal{U}} \in \mathbb{WM} \cap \mathbb{B}$. Hence, $\mathfrak{M}_{\mathcal{U}} = \mathfrak{M}_{\overline{\mathcal{U}}} \in \{\mathfrak{M}_{\mathcal{U}} \mid \mathcal{U} \in \mathbb{WM} \cap \mathbb{B}\}$.

It is trivial to see that $\{\mathfrak{M}_{\mathcal{U}} \mid \mathcal{U} \in \mathbb{WM} \cap \mathbb{B}\} \subseteq \{\mathfrak{M}_{\mathcal{U}} \mid \mathcal{U} \in \mathbb{WM}\}$. The inclusion $\{\mathfrak{M}_{\mathcal{U}} \mid \mathcal{U} \in \mathbb{WM}\} \subseteq \mathbb{MOD}(\lambda)$ follows from Corollary 5.7. ◀

▶ **Corollary 8.5.** *There is a bijection from the set $\mathbb{WM} \cap \mathbb{B}$ to $\mathbb{MOD}(\lambda)$.*

Proof. Let $\mathcal{U} \in \mathbb{WM} \cap \mathbb{B}$. By Corollary 5.7, the infinitary lambda calculus $\lambda^{\infty}_{\beta\bot_{\mathcal{U}}}$ is confluent and normalizing. Hence, we can consider the mapping that given $\mathcal{U} \in \mathbb{WM} \cap \mathbb{B}$ yields $\mathfrak{M}_{\mathcal{U}}$. This mapping is surjective by Corollary 8.4 and it is injective by Corollary 6.4. ◀

9 Conclusions and Future Research

In this paper, we have weakend the Axiom of Overlap in order to find an axiomatization that is both necessary and sufficient for having confluent and normalizing infinitary lambda calculi $\lambda^{\infty}_{\beta\bot_{\mathcal{U}}}$.

In a natural sequel to this paper we plan to study the same question for first order term rewriting. Afteral the axioms of meaningless sets (minus substitution) were first formulated for such systems [2, 11]. If successful a generalisation to combinatory reduction systems (extending [12]) may then well be possible.

The sets shown in Figure 1 are not the only sets of weak meaningless terms. We also plan to study the structure of the set $\mathbb{WM}$ of sets of weak meaningless terms closed under β expansion and provide an exhaustive classification if possible.

One reason that this set is of interest is that each such weak meaningless set gives rise to its own model of the infinitary lambda calculus, which in turn defines a finitary lambda theory. We are hopeful that the set of weakly meaningless sets is in fact a lattice. And it is of interest to explore the relation with the well-studied lattice of lambda theories.

Finally it is of interest to see how other denotational semantics can model the infinitary lambda caluli. Or to see whether each of the infinite lambda calculi $\lambda^{\infty}_{\beta\bot_{\mathcal{U}}}$ can be provided with an intersection type discipline sucht that two terms have the same normal form if and only if the have the same type.

Acknowledgements

We would like to thank the reviewers for their detailed and helpful comments and suggestions that they provided.

References

1 S. Abramsky and C.-H. Luke Ong. Full abstraction in the lazy lambda calculus. *Information and Computation*, 105(2):159–267, 1993.

2 Z.M. Ariola, J.R. Kennaway, J.W. Klop, M.R. Sleep, and F.J. de Vries. Syntactic definitions of undefined: On defining the undefined. In *Theoretical Aspects of Computer Software*, volume 789 of *LNCS*, pages 543–554. Springer, 1994.

3 H.P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, Amsterdam, Revised edition, 1984.

4 H.P. Barendregt. Representing 'undefined' in lambda calculus. *Journal of Functional Programming*, 2(3):367–374, July 1992.

5 M. Barr. Terminal coalgebras for endofunctors on sets. *Theoretical Computer Science*, 114(2):299–315, 1999.

6 A. Berarducci. Infinite λ-calculus and non-sensible models. In *Logic and algebra (Pontignano, 1994)*, pages 339–377. Dekker, New York, 1996.

7 J.R. Kennaway and F.J. de Vries. Infinitary rewriting. In Terese, editor, *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*, pages 668–711. Cambridge University Press, 2003.

8 J.R. Kennaway, J.W. Klop, M.R. Sleep, and F.J. de Vries. Infinite lambda calculus and Böhm models. In *Rewriting Techniques and Applications*, volume 914 of *LNCS*, pages 257–270. Springer, 1995.

9 J.R. Kennaway, J.W. Klop, M.R. Sleep, and F.J. de Vries. Infinitary lambda calculus. *Theoretical Computer Science*, 175(1):93–125, 1997.

10 J.R. Kennaway, P.G. Severi, M.R. Sleep, and F.J. de Vries. Infinitary rewriting: From syntax to semantics. In *Processes, Terms and Cycles*, volume 3838 of *LNCS*, pages 148–172. Springer, 2005.

11 J.R. Kennaway, V. van Oostrom, and F.J. de Vries. Meaningless terms in rewriting. *Journal of Functional and Logic Programming*, Article 1:35 pp, 1999.

12 J. Ketema. Comparing Böhm-like trees. In *Rewriting Techniques and Applications*, volume 5595 of *LNCS*, pages 239–254. Springer, 2009.

13 J. Kuper. *Partiality in Logic and Computation, Aspects of Undefinedness*. PhD thesis, Universiteit Twente, February 1994.

14 P.G. Severi and F.J. de Vries. An extensional Böhm model. In *Rewriting Techniques and Applications*, volume 2378 of *LNCS*, pages 159–173. Springer, 2002.

15 P.G. Severi and F.J. de Vries. Continuity and discontinuity in lambda calculus. In *Typed Lambda Calculus and Applications*, volume 3461 of *LNCS*, pages 369–385. Springer, 2005.

16 P.G. Severi and F.J. de Vries. Order Structures for Böhm-like models. In *Computer Science Logic*, volume 3634 of *LNCS*, pages 103–116. Springer, 2005.

17 P.G. Severi and F.J. de Vries. A Lambda Calculus for D_∞. Technical report, University of Leicester, 2002.

Modular and Certified Semantic Labeling and Unlabeling*

Christian Sternagel and René Thiemann

Institute of Computer Science, University of Innsbruck, Austria
{christian.sternagel|rene.thiemann}@uibk.ac.at

Abstract

Semantic labeling is a powerful transformation technique to prove termination of term rewrite systems. The dual technique is unlabeling. For unlabeling it is essential to drop the so called decreasing rules which sometimes have to be added when applying semantic labeling. We indicate two problems concerning unlabeling and present our solutions.

The first problem is that currently unlabeling cannot be applied as a modular step, since the decreasing rules are determined by a semantic labeling step which may have taken place much earlier. To this end, we give an implicit definition of decreasing rules that does not depend on any knowledge about preceding labelings.

The second problem is that unlabeling is in general unsound. To solve this issue, we introduce the notion of extended termination problems. Moreover, we show how existing termination techniques can be lifted to operate on extended termination problems.

All our proofs have been formalized in Isabelle/HOL as part of the IsaFoR/CeTA project.

1998 ACM Subject Classification F.4.2

Keywords and phrases semantic labeling, certification, term rewriting, unlabeling

Digital Object Identifier 10.4230/LIPIcs.RTA.2011.329

Category Regular Research Paper

1 Introduction

In recent years, termination provers for term rewrite systems (TRSs) became more and more powerful. Nowadays, we do no longer have to prove termination by embedding all rules of a TRS into a single reduction order. Instead, most provers construct multi-step proofs by combining different termination techniques[1] resulting in tree-like termination proofs. As a result, termination provers became more complex and thus, more error-prone. It is regularly demonstrated that we cannot blindly trust termination provers. Every now and then, some prover delivers a faulty proof. Most of the time, this is only detected if there is another prover giving a contradictory answer. Furthermore, it just is too much work to check a generated proof by hand. (Besides, checking by hand is not very reliable.)

To solve this issue, recent interest is in the automatic certification of termination proofs [3, 4, 18]. To this end, we formalized many termination techniques in our Isabelle/HOL [15] library IsaFoR [18] (in the remainder we just write *Isabelle*, instead of *Isabelle/HOL*). Using IsaFoR, we obtain CeTA, an automatic certifier for termination proofs.

* This research is supported by FWF (Austrian Science Fund) project P22767-N13.

[1] Several termination techniques are based upon reduction orders, but there are also techniques which do not generate orders. Hence, the multi-step proofs are not just a lexicographic combination of orders.

22nd International Conference on Rewriting Techniques and Applications (RTA'11).
Editor: M. Schmidt-Schauß; pp. 329–344

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

In this paper, we present our formalization of semantic labeling and unlabeling [19], two important termination techniques. Semantic labeling introduces differently labeled variants of the same symbol, allowing a distinction in orders, etc. Semantic labeling typically produces large TRSs. Hence, unlabeling is important to keep the number of symbols and rules small.

▶ **Example 1.1.** Consider the small TRS `Secret_05/teparla3` from the termination problem database (TPDB) which only consists of two rules, has two different symbols, and two variables. We just describe the structure of the proof that has been generated by the termination prover AProVE [10] in 21 seconds during the 2008 termination competition.[2]

After applying the dependency pair transformation [1] and some standard techniques, a termination problem containing three rules and three different symbols is obtained. Then, semantic labeling is applied. The result after simplification, is a system with five rules and seven different symbols. Unlabeling yields a problem with three rules and three symbols. Another labeling produces a new termination problem with 12 rules. This is finally proven to be terminating using a matrix interpretation [6] of dimension two.

Note that without the unlabeling step, the second labeling would have returned a system with 5025 rules instead of 12—for this huge termination problem no suitable matrix interpretation of dimension two is detected.

Whereas the previous example shows that unlabeling is essential to keep systems small, we also found examples where unlabeling was the key to get a successful termination proof at all, cf. Example 4.2 for details.

Unfortunately, unlabeling is not sound in general. In order to allow nested labeling and unlabeling and turn unlabeling into a sound and modular technique (not relying on context information), we have designed a new framework. All existing termination techniques are easily integrated in this framework. In fact, CeTA uses the new framework for certification.

Note that all the proofs that are presented (or omitted) in the following, have been formalized as part of IsaFoR. Hence, we merely give sketches of our "real" proofs. Our goal is to show the general proof outlines and help to understand the full proofs. The 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 Section 2, we recapitulate some required notions of term rewriting as well as the basic definitions of semantic labeling. Afterwards, in Section 3, we give some challenges for modular labeling and unlabeling. Then, in Section 4, we extend the previous results to the dependency pair framework. We discuss challenges for the certification in Section 5. Our experiments are presented in Section 6 before we conclude in Section 7.

2 Preliminaries

2.1 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 $f(\vec{t}_n)$. For brevity we write $\vec{t}_n$ to denote a sequence of n elements $t_1, \ldots, t_n$ and $(h(\vec{t}_n))$

[2] See `http://termcomp.uibk.ac.at/termcomp/competition/resultDetail.seam?resultId=35708`

(note the additional pair of parentheses) for $(h(t_1), \ldots, h(t_n))$, i.e., mapping a function h over the elements of a sequence $\vec{t}_n$. A *context* C is a term containing exactly one hole ($\Box$). Replacing $\Box$ in a context C by a term t is denoted by $C[t]$. A *(rewrite) rule* is a pair of terms $\ell \to r$ and a TRS $\mathcal{R}$ is a set of such rules. The *rewrite relation (induced by $\mathcal{R}$)* $\to_{\mathcal{R}}$ is the closure under substitutions and contexts of $\mathcal{R}$, i.e., $s \to_{\mathcal{R}} t$ iff there is a context C, a rule $\ell \to r \in \mathcal{R}$, and a substitution σ such that $s = C[\ell\sigma]$ and $t = C[r\sigma]$.

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^+ (S^*) for the (reflexive and) transitive closure of S.

▶ **Definition 2.1** (Termination Technique)**.** A *termination technique* is a mapping TT from TRSs to TRSs. It is *sound* if termination of $TT(\mathcal{R})$ implies termination of $\mathcal{R}$.

Using sound termination techniques one tries to modify a given TRS $\mathcal{R}$ until the empty TRS is reached. If this succeeds, one obtains a proof tree showing termination of $\mathcal{R}$.

2.2 Semantic Labeling

An algebra $\mathcal{A}$ over $\mathcal{F}$ is a pair $(A, \{f_{\mathcal{A}}\}_{f \in \mathcal{F}})$ consisting of a non-empty carrier A and an interpretation function $f_{\mathcal{A}} \colon A^n \to A$ for every n-ary function symbol $f \in \mathcal{F}$. Given an assignment $\alpha \colon \mathcal{V} \to A$, we write $[\alpha]_{\mathcal{A}}(t)$ for the interpretation of the term t. An algebra $\mathcal{A}$ is a model of a rewrite system $\mathcal{R}$, if $[\alpha]_{\mathcal{A}}(\ell) = [\alpha]_{\mathcal{A}}(r)$ for all rules $\ell \to r \in \mathcal{R}$ and all assignments α. If the carrier A is equipped with a well-founded order $>_A$ such that $[\alpha]_{\mathcal{A}}(\ell) \geqslant_A [\alpha]_{\mathcal{A}}(r)$ for all $\ell \to r \in \mathcal{R}$ and all assignments α, then $\mathcal{A}$ is a *quasi-model* of $\mathcal{R}$.

For each function symbol f there also is a corresponding non-empty set L_f of labels for f and a labeling function $\ell_f \colon A^n \to L_f$. The labeled signature $\mathcal{F}_{\mathsf{lab}}$ consists of n-ary function symbols f_a for every n-ary function symbol $f \in \mathcal{F}$ and label $a \in L_f$. The labeling function ℓ_f determines the label of the root symbol f of a term $f(\vec{t}_n)$ based on the values of the arguments $\vec{t}_n$. For every assignment $\alpha \colon \mathcal{V} \to A$ the mapping $\mathsf{lab}_\alpha \colon \mathcal{T}(\mathcal{F}, \mathcal{V}) \to \mathcal{T}(\mathcal{F}_{\mathsf{lab}}, \mathcal{V})$ is inductively defined by

$$\mathsf{lab}_\alpha(t) = \begin{cases} f_{\ell_f([\alpha]_{\mathcal{A}}(\vec{t}_n))}(\mathsf{lab}_\alpha(\vec{t}_n)) & \text{if } t = f(\vec{t}_n), \\ t & \text{otherwise.} \end{cases}$$

The labeled TRS $\mathsf{lab}(\mathcal{R})$ over the signature $\mathcal{F}_{\mathsf{lab}}$ consists of the rules $\mathsf{lab}_\alpha(\ell) \to \mathsf{lab}_\alpha(r)$ for all $\ell \to r \in \mathcal{R}$ and $\alpha \colon \mathcal{V} \to A$.

For quasi-models, every set of labels L_f needs to be equipped with a well-founded order $>_{L_f}$, giving rise to the set $\mathcal{D}\mathrm{ec}$ of *decreasing rules*:

$$\mathcal{D}\mathrm{ec} = \{f_a(\vec{x}_n) \to f_b(\vec{x}_n) \mid a, b \in L_f, a >_{L_f} b, n\text{-ary } f \in \mathcal{F}\}$$

Furthermore, every interpretation function $f_{\mathcal{A}}$ and every labeling function ℓ_f has to be weakly monotone, i.e., if $a \geqslant_A a'$ then $f_{\mathcal{A}}(a_1, \ldots, a, \ldots, a_n) \geqslant_A f_{\mathcal{A}}(a_1, \ldots, a', \ldots, a_n)$ and $\ell_f(a_1, \ldots, a, \ldots, a_n) \geqslant_{L_f} \ell_f(a_1, \ldots, a', \ldots, a_n)$.

Unlabeling a symbol is defined via the following function, removing one layer of labels. Then, the function is extended homomorphically to terms, rules, and TRSs.

$$\mathsf{unlab}(f) = \begin{cases} g & \text{if } f = g_a, \\ f & \text{if } f \text{ is not labeled.} \end{cases}$$

In [19], Zantema showed that labeled TRSs can simulate their unlabeled counterparts (corresponding to **1** and **2** in the following lemma; **3** and **4** are obvious).

▶ **Lemma 2.2.** *Let $\mathcal{R}$ be a TRS,$\mathcal{A}$ an algebra, and α an arbitrary assignment.*
1. *If $\mathcal{A}$ is a model of $\mathcal{R}$ then $t \to_{\mathcal{R}} u$ implies $\mathsf{lab}_\alpha(t) \to_{\mathsf{lab}(\mathcal{R})} \mathsf{lab}_\alpha(u)$.*
2. *If $\mathcal{A}$ is a quasi-model of $\mathcal{R}$ then $t \to_{\mathcal{R}} u$ implies $\mathsf{lab}_\alpha(t) \to^+_{\mathsf{lab}(\mathcal{R})\cup\mathcal{D}\mathrm{ec}} \mathsf{lab}_\alpha(u)$.*
3. *$t \to_{\mathsf{lab}(\mathcal{R})} u$ implies $\mathsf{unlab}(t) \to_{\mathcal{R}} \mathsf{unlab}(u)$.*
4. *$t \to_{\mathcal{D}\mathrm{ec}} u$ implies $\mathsf{unlab}(t) = \mathsf{unlab}(u)$.*

From Lemma 2.2 we obtain that $\mathcal{R}$ is terminating if and only if $\mathsf{lab}(\mathcal{R})\,(\cup\,\mathcal{D}\mathrm{ec})$ is terminating when $\mathcal{A}$ is a (quasi-)model of $\mathcal{R}$. Completeness is achieved by unlabeling all terms in a possible infinite rewrite sequence of the labeled TRS. Soundness is proved by transforming a presupposed infinite rewrite sequence in $\mathcal{R}$ into an infinite rewrite sequence in $\mathsf{lab}(\mathcal{R})\,(\cup\,\mathcal{D}\mathrm{ec})$. This is done by applying the labeling function $\mathsf{lab}_\alpha(\cdot)$ (for an arbitrary assignment α) to all terms in the infinite rewrite sequence of $\mathcal{R}$. Hence, semantic labeling is sound and complete for termination (using models and quasi-models, respectively).

3 Modular Semantic Labeling and Unlabeling

One problem with semantic labeling is that the labeled system is usually large. Hence, termination provers such as AProVE [10], Jambox [5], Torpa [20], and TPA [13] perform labeling, then try to simplify the resulting TRS by sound termination techniques, and afterwards *unlabel* the TRS again, to continue on a small system. This poses two challenges:
1. If labeling was performed using a quasi-model, then the decreasing rules are added. However, unlabeling a decreasing rule $f_a(\vec{x}_n) \to f_b(\vec{x}_n)$ leads to the nonterminating rule $f(\vec{x}_n) \to f(\vec{x}_n)$. Hence, one has to remove the decreasing rules before unlabeling.
2. Between labeling and unlabeling, arbitrary (sound) termination techniques may be applied. However, for unlabeling we want to remove the decreasing rules that are determined by the corresponding labeling step. Hence, unlabeling is not a modular technique that only takes a TRS as input. Instead, it relies on context information, namely the decreasing rules that have been used in the corresponding labeling step (which may occur several steps upwards in the termination proof).

Solving the first challenge is technically easy: just remove the decreasing rules before unlabeling. The only question is, whether it is always sound to remove the decreasing rules.

To handle the second challenge, we propose an implicit definition of decreasing rules.

▶ **Definition 3.1** (Decreasing rules of a TRS)**.** We define the *decreasing rules of a TRS $\mathcal{R}$* as $\mathcal{D}(\mathcal{R}) = \{\ell \to r \in \mathcal{R} \mid \mathsf{unlab}(\ell) = \mathsf{unlab}(r) \wedge \ell \neq r\}$. We further define the *unlabeled version of a TRS* as $\mathcal{U}(\mathcal{R}) = \mathsf{unlab}(\mathcal{R} \setminus \mathcal{D}(\mathcal{R}))$.

The condition $\ell \neq r$ ensures that a labeled variant of an original rule is never decreasing. For example, if $f(\vec{x}_n) \to f(\vec{x}_n)$ is a rule (and hence the original TRS is not terminating), then each labeled variant has the form $f_a(\vec{x}_n) \to f_a(\vec{x}_n)$ for some $a \in L_f$. If we would consider such a rule as decreasing, we could transform a nonterminating TRS into a terminating one, using labeling and unlabeling.

▶ **Lemma 3.2.** *Let L_f and $>_{L_f}$ be given for each symbol f to determine $\mathcal{D}\mathrm{ec}$. Then $\mathcal{D}(\mathcal{D}\mathrm{ec}) = \mathcal{D}\mathrm{ec}$, $\mathcal{D}(\mathsf{lab}(\mathcal{R})) = \varnothing$, $\mathcal{D}(\mathsf{lab}(\mathcal{R}) \cup \mathcal{D}\mathrm{ec}) = \mathcal{D}\mathrm{ec}$, and $\mathcal{U}(\mathsf{lab}(\mathcal{R}) \cup \mathcal{D}\mathrm{ec}) = \mathcal{R}$.*

Now it is easy to define a modular version of unlabeling which does not require external knowledge about what the decreasing rules are.

▶ **Definition 3.3** (Unlabeling as modular termination technique)**.** The *unlabeling termination technique* replaces a TRS $\mathcal{R}$ by $\mathcal{U}(\mathcal{R})$.

Hence, we solved the second challenge and made unlabeling into an independent technique which does not need any knowledge on the previous application of semantic labeling that introduced the decreasing rules. Thus, termination proofs can now use the following structure where no global information has to be passed around:

1. Switch from $\mathcal{R}$ to $\mathsf{lab}(\mathcal{R})$ $(\cup\, \mathcal{D}\mathrm{ec})$.
2. Modify $\mathsf{lab}(\mathcal{R})$ $(\cup\, \mathcal{D}\mathrm{ec})$ by sound termination techniques resulting in $\mathcal{R}'$.
3. Unlabel $\mathcal{R}'$ resulting in $\mathcal{U}(\mathcal{R}')$.

Although this approach is used in termination provers, it is unsound in general as not every sound termination technique may be used between labeling and unlabeling. This is illustrated by the following example.

▶ **Example 3.4.** We start with the nonterminating TRS $\mathcal{R} = \{\mathsf{f}(\mathsf{a}) \to \mathsf{f}(\mathsf{b}), \mathsf{b} \to \mathsf{a}\}$. Then, we apply semantic labeling using the algebra $\mathcal{A}$ with $A = \{0, 1\}$, interpretations $\mathsf{f}_{\mathcal{A}}(x) = 0$, $\mathsf{a}_{\mathcal{A}} = 0$, $\mathsf{b}_{\mathcal{A}} = 1$, $L_{\mathsf{f}} = A$, $\ell_{\mathsf{f}}(x) = x$, and the standard order on the naturals. Note that $\mathcal{A}$ is a quasi-model of $\mathcal{R}$. The resulting labeled TRS is $\mathsf{lab}(\mathcal{R}) \cup \mathcal{D}\mathrm{ec} = \{\mathsf{f}_0(\mathsf{a}) \to \mathsf{f}_1(\mathsf{b}), \mathsf{b} \to \mathsf{a}, \mathsf{f}_1(x) \to \mathsf{f}_0(x)\}$. It is sound to replace $\mathsf{lab}(\mathcal{R}) \cup \mathcal{D}\mathrm{ec}$ by the (nonterminating) TRS $\mathcal{R}' = \{\mathsf{f}_1(x) \to \mathsf{f}_0(x), \mathsf{f}_0(x) \to \mathsf{f}_1(x)\}$. However, unlabeling $\mathcal{R}'$ yields $\mathcal{U}(\mathcal{R}') = \varnothing$ as both rules in $\mathcal{R}'$ are decreasing according to Definition 3.1. Hence, some of the performed deductions were not sound. Since semantic labeling and the switch from $\mathsf{lab}(\mathcal{R}) \cup \mathcal{D}\mathrm{ec}$ to $\mathcal{R}'$ are sound, we obtain that unlabeling via $\mathcal{U}$ is unsound.

The problematic step when unlabeling, i.e., when switching from $\mathcal{R}$ to $\mathcal{U}(\mathcal{R}) = \mathsf{unlab}(\mathcal{R} \setminus \mathcal{D}(\mathcal{R}))$, is the removal of the decreasing rules. If the decreasing rules are the only source of nontermination, then this removal is unsound. However, the decreasing rules $\mathcal{D}\mathrm{ec}$ that are obtained from semantic labeling are always terminating. Thus, after labeling we have to prove termination of the labeled system including the decreasing rules, but we may assume that the decreasing rules are terminating. If we know that the decreasing rules are terminating, then unlabeling by $\mathcal{U}$ is sound. We obtain the following structure of termination proofs:

1. Initially we have to prove $\mathsf{SN}(\mathcal{R})$.
2. After labeling, we have to prove $\mathsf{SN}(\mathcal{D}(\mathcal{R}')) \Longrightarrow \mathsf{SN}(\mathcal{R}')$ for $\mathcal{R}' = \mathsf{lab}(\mathcal{R}) \cup \mathcal{D}\mathrm{ec}$.
3. Then, we modify $\mathcal{R}'$ to $\mathcal{R}''$ with $\mathsf{SN}(\mathcal{D}(\mathcal{R}'')) \Longrightarrow \mathsf{SN}(\mathcal{R}'')$ implies $\mathsf{SN}(\mathcal{D}(\mathcal{R}')) \Longrightarrow \mathsf{SN}(\mathcal{R}')$.
4. Finally, we unlabel $\mathcal{R}''$ resulting in $\mathcal{U}(\mathcal{R}'')$ and have to prove $\mathsf{SN}(\mathcal{U}(\mathcal{R}''))$.

This approach works fine for termination proofs where semantic labeling is not nested. However, we are aware of termination proofs where labeling is applied in a nested way.

▶ **Example 3.5.** Consider the TRS `Gebhardt_06/16` from the TPDB. During the 2008 termination competition, Jambox proved termination of this TRS, applying the following steps: labeling - labeling - labeling - polynomial order - unlabeling - four applications of polynomial orders - unlabeling - unlabeling.[3]

To support this kind of proof we define the following variant of strong normalization.

▶ **Definition 3.6.** An *extended termination problem* is a pair $(\mathcal{R}, n)$ consisting of a TRS $\mathcal{R}$ and a number $n \in \mathbb{N}$. An extended problem $(\mathcal{R}, n)$ is *strongly normalizing* $(\mathsf{SN}(\mathcal{R}, n))$ iff

$$(\forall m < n.\, \mathsf{SN}(\mathcal{D}(\mathcal{U}^m(\mathcal{R})))) \quad \Longrightarrow \quad \mathsf{SN}(\mathcal{R}).$$

[3] See `http://termcomp.uibk.ac.at/termcomp/competition/resultDetail.seam?resultId=27220`

An *extended termination technique* is a mapping xTT from extended termination problems to extended termination problems. It is *sound* iff $\mathsf{SN}(xTT(\mathcal{R},n))$ implies $\mathsf{SN}(\mathcal{R},n)$.

The number n in an extended termination problem $(\mathcal{R},n)$ describes how often we can assume that the decreasing rules are terminating, and hence, it tells us how often we can delete the decreasing rules during unlabeling. The following lemma provides the link between both variants of strong normalization.

▶ **Lemma 3.7.** *1.* $\mathsf{SN}(\mathcal{R})$ *iff* $\mathsf{SN}(\mathcal{R},0)$.
2. If $\mathsf{SN}(\mathcal{R})$ *then* $\mathsf{SN}(\mathcal{R},n)$.

▶ **Lemma 3.8** (Extended Unlabeling)**.** *Extended unlabeling is sound where*

$$\mathcal{U}(\mathcal{R},n) = \begin{cases} (\mathcal{U}(\mathcal{R}),n-1) & \text{if } n>0, \\ (\mathsf{unlab}(\mathcal{R}),0) & \text{otherwise.} \end{cases}$$

Proof. We only consider the interesting case where $n>0$. So, we have to show $\mathsf{SN}(\mathcal{R},n)$ under the first assumption $\mathsf{SN}(\mathcal{U}(\mathcal{R}),n-1)$. To prove $\mathsf{SN}(\mathcal{R},n)$, we have to prove $\mathsf{SN}(\mathcal{R})$ under the second assumption $\forall m<n.\,\mathsf{SN}(\mathcal{D}(\mathcal{U}^m(\mathcal{R})))$. Since $n>0$ we can choose $m=0$ and obtain $\mathsf{SN}(\mathcal{D}(\mathcal{R}))$.

To show $\mathsf{SN}(\mathcal{R})$ we assume that there is an infinite $\to_{\mathcal{R}}$-derivation $t_1 \to_{\mathcal{R}} t_2 \to_{\mathcal{R}} \cdots$ and obtain a contradiction. The infinite derivation is also an infinite $\to_{\mathcal{R}\setminus\mathcal{D}(\mathcal{R})} \cup \to_{\mathcal{D}(\mathcal{R})}$-derivation. Since $\mathcal{D}(\mathcal{R})$ is terminating, we know that there are infinitely many i with $t_i \to_{\mathcal{R}\setminus\mathcal{D}(\mathcal{R})} t_{i+1}$. Hence $\mathsf{unlab}(t_i) \to_{\mathcal{U}(\mathcal{R})} \mathsf{unlab}(t_{i+1})$ for all these i as $\mathcal{U}(\mathcal{R}) = \mathsf{unlab}(\mathcal{R}\setminus\mathcal{D}(\mathcal{R}))$. Moreover, for all i where $t_i \to_{\mathcal{D}(\mathcal{R})} t_{i+1}$, we know that $\mathsf{unlab}(t_i) \to_{\mathsf{unlab}(\mathcal{D}(\mathcal{R}))} \mathsf{unlab}(t_{i+1})$ and hence, $\mathsf{unlab}(t_i) = \mathsf{unlab}(t_{i+1})$ since every rule in $\mathsf{unlab}(\mathcal{D}(\mathcal{R}))$ has the same left- and right-hand side. Thus, we have constructed an infinite derivation for $\mathcal{U}(\mathcal{R})$ proving that $\mathsf{SN}(\mathcal{U}(\mathcal{R}))$ does not hold. Together with the assumption $\mathsf{SN}(\mathcal{U}(\mathcal{R}),n-1)$, we obtain that $\forall m<n-1.\,\mathsf{SN}(\mathcal{D}(\mathcal{U}^m(\mathcal{U}(\mathcal{R}))))$ does not hold (by Definition 3.6). Hence, there is some $m<n-1$ such that $\mathsf{SN}(\mathcal{D}(\mathcal{U}^{m+1}(\mathcal{R})))$ does not hold. Now, using the second assumption and $m+1<n$ we obtain $\mathsf{SN}(\mathcal{D}(\mathcal{U}^{m+1}(\mathcal{R})))$, providing the required contradiction. ◀

▶ **Lemma 3.9** (Extended Semantic Labeling)**.** *Semantic labeling is sound as extended termination technique: Whenever we can switch from* $\mathcal{R}$ *to* $\mathsf{lab}(\mathcal{R})$ $(\cup\,\mathcal{D}\mathrm{ec})$ *via semantic labeling, then it is sound to switch from* $(\mathcal{R},n)$ *to* $(\mathsf{lab}(\mathcal{R})\ (\cup\,\mathcal{D}\mathrm{ec}),n+1)$.

Proof. Note that models are just a special case of quasi-models as already observed in [19]. Hence, we only consider quasi-models in the proof. So, assuming $\mathsf{SN}(\mathsf{lab}(\mathcal{R})\cup\mathcal{D}\mathrm{ec},n+1)$ we have to prove $\mathsf{SN}(\mathcal{R},n)$. To show the latter, we may assume $\forall m<n.\,\mathsf{SN}(\mathcal{D}(\mathcal{U}^m(\mathcal{R})))$ and have to prove $\mathsf{SN}(\mathcal{R})$. We do so by assuming that there is an infinite $\mathcal{R}$-derivation $t_1 \to_{\mathcal{R}} t_2 \to_{\mathcal{R}} \cdots$ and deriving a contradiction. As we have a quasi-model we know that $\mathsf{lab}_\alpha(t_1) \to^+_{\mathsf{lab}(\mathcal{R})\cup\mathcal{D}\mathrm{ec}} \mathsf{lab}_\alpha(t_2) \to^+_{\mathsf{lab}(\mathcal{R})\cup\mathcal{D}\mathrm{ec}} \cdots$ is an infinite $\mathsf{lab}(\mathcal{R})\cup\mathcal{D}\mathrm{ec}$-derivation, showing that $\mathsf{SN}(\mathsf{lab}(\mathcal{R})\cup\mathcal{D}\mathrm{ec})$ does not hold. By the conditions of semantic labeling, we further know $\mathsf{SN}(\mathcal{D}\mathrm{ec})$. Using $\mathsf{SN}(\mathsf{lab}(\mathcal{R})\cup\mathcal{D}\mathrm{ec},n+1)$ we conclude that $\forall m<n+1.\,\mathsf{SN}(\mathcal{D}(\mathcal{U}^m(\mathsf{lab}(\mathcal{R})\cup\mathcal{D}\mathrm{ec})))$ does not hold. Hence there is some $m<n+1$ such that $\mathsf{SN}(\mathcal{D}(\mathcal{U}^m(\mathsf{lab}(\mathcal{R})\cup\mathcal{D}\mathrm{ec})))$ does not hold. If $m=0$ then by Lemma 3.2 we know that $\mathcal{D}(\mathcal{U}^m(\mathsf{lab}(\mathcal{R})\cup\mathcal{D}\mathrm{ec})) = \mathcal{D}(\mathsf{lab}(\mathcal{R})\cup\mathcal{D}\mathrm{ec}) = \mathcal{D}\mathrm{ec}$, and thus $\mathsf{SN}(\mathcal{D}\mathrm{ec})$ does not hold, a contradiction. Otherwise, $m=m'+1$ for some m' where $m'<n$. Together with $\forall m<n.\,\mathsf{SN}(\mathcal{D}(\mathcal{U}^m(\mathcal{R})))$, we obtain $\mathsf{SN}(\mathcal{D}(\mathcal{U}^{m'}(\mathcal{R})))$. On the other hand, we know that $\mathsf{SN}(\mathcal{D}(\mathcal{U}^{m'+1}(\mathsf{lab}(\mathcal{R})\cup\mathcal{D}\mathrm{ec})))$ does not hold. This again leads to a contradiction since $\mathcal{D}(\mathcal{U}^{m'+1}(\mathsf{lab}(\mathcal{R})\cup\mathcal{D}\mathrm{ec})) = \mathcal{D}(\mathcal{U}^{m'}(\mathcal{U}(\mathsf{lab}(\mathcal{R})\cup\mathcal{D}\mathrm{ec}))) = \mathcal{D}(\mathcal{U}^{m'}(\mathcal{R}))$ by Lemma 3.2. ◀

The previous two lemmas show that labeling and unlabeling can be performed as independent techniques on extended termination problems.

The question remains how to integrate other existing termination techniques, i.e., which techniques may be applied between labeling and unlabeling. Here, we consider two variants.

▶ **Definition 3.10** (Lift). Let TT be some termination technique. Then $\mathsf{lift}(TT)$ and $\mathsf{lift}_0(TT)$ are extended termination techniques where $\mathsf{lift}(TT)(\mathcal{R}, n) = (TT(\mathcal{R}), n)$ and $\mathsf{lift}_0(TT)(\mathcal{R}, n) = (TT(\mathcal{R}), 0)$.

In principle $\mathsf{lift}(TT)$ is preferable, since it does not change n, allowing to remove the decreasing rules when unlabeling (which is not possible using $\mathsf{lift}_0(TT)$). However, in general the fact that TT is sound does not imply that $\mathsf{lift}(TT)$ is sound. This can easily be seen by reusing Example 3.4 where the extended termination problem $(\mathcal{R}, 0)$ is transformed to $(\mathsf{lab}(\mathcal{R}) \cup \mathcal{D}\mathrm{ec}, 1)$ by semantic labeling, then to $(\mathcal{R}', 1)$ using $\mathsf{lift}(TT)$ for the unnamed sound termination technique TT in Example 3.4, and then to $(\varnothing, 0)$ by unlabeling. Since this establishes a complete termination proof for the nonterminating TRS $\mathcal{R}$, and since labeling and unlabeling are sound, we know that $\mathsf{lift}(TT)$ is unsound.

Since we cannot always use $\mathsf{lift}(TT)$, we give three different approaches to use termination techniques as extended termination techniques (in order of preference):

1. Identify a (hopefully large) class of termination techniques TT for which soundness of TT implies soundness of $\mathsf{lift}(TT)$.
2. Perform a direct proof that $\mathsf{lift}(TT)$ is sound as extended termination technique.
3. Use $\mathsf{lift}_0(TT)$ for any sound termination technique TT.

We first prove soundness of approach **3**.

▶ **Lemma 3.11.** *If TT is sound then $\mathsf{lift}_0(TT)$ is sound.*

Proof. We have to prove that $\mathsf{SN}(TT(\mathcal{R}), 0)$ implies $\mathsf{SN}(\mathcal{R}, n)$. So, assume $\mathsf{SN}(TT(\mathcal{R}), 0)$. Hence, $\mathsf{SN}(TT(\mathcal{R}))$ using Lemma 3.7(**1**). As TT is sound, we conclude $\mathsf{SN}(\mathcal{R})$ and this implies $\mathsf{SN}(\mathcal{R}, n)$ by Lemma 3.7(**2**). ◀

We start to prove soundness of $\mathsf{lift}(TT)$ for some sound termination technique TT in order to detect where the problem is. To prove soundness, we have to show that $\mathsf{SN}(TT(\mathcal{R}), n)$ implies $\mathsf{SN}(\mathcal{R}, n)$. Thus, assume $\mathsf{SN}(TT(\mathcal{R}), n)$. To prove $\mathsf{SN}(\mathcal{R}, n)$ we may assume that $\forall m < n.\, \mathsf{SN}(\mathcal{D}(\mathcal{U}^m(\mathcal{R})))$ and have to prove $\mathsf{SN}(\mathcal{R})$. Since TT is sound, it suffices to prove $\mathsf{SN}(TT(\mathcal{R}))$. To this end, it suffices to show $\forall m < n.\, \mathsf{SN}(\mathcal{D}(\mathcal{U}^m(TT(\mathcal{R}))))$ by using $\mathsf{SN}(TT(\mathcal{R}), n)$. Hence, the only missing step is to conclude

$$\mathsf{SN}(\mathcal{D}(\mathcal{U}^m(\mathcal{R}))) \quad \Longrightarrow \quad \mathsf{SN}(\mathcal{D}(\mathcal{U}^m(TT(\mathcal{R})))). \tag{$\star$}$$

▶ **Lemma 3.12.** *If TT is sound and if ($\star$) is satisfied for all m, then $\mathsf{lift}(TT)$ is sound.*

A sufficient condition to ensure ($\star$) is to demand that $TT(\mathcal{R}) \subseteq \mathcal{R}$ as unlab, $\mathcal{D}$, and $\mathcal{U}$ are monotone w.r.t. set inclusion. Hence, all techniques that remove rules like rule removal via reduction pairs, or (RFC) matchbounds [7, 14] can safely be used between labeling and unlabeling. However, this excludes techniques like the flat context closure which is required for root-labeling.

▶ **Definition 3.13** (Root-Labeling). Let $\mathcal{R}$ be a TRS over the signature $\mathcal{F}$. Let $\mathcal{A}_\mathcal{F}$ be an algebra with carrier $\mathcal{F}$. Moreover, for every n-ary $f \in \mathcal{F}$, we fix the interpretation function $f_{\mathcal{A}_\mathcal{F}}(\vec{x}_n) = f$, the set of labels $L_f = \mathcal{F}^n$, and the labeling function $\ell_f(\vec{x}_n) = (\vec{x}_n)$.

Note that root-labeling is just a specific instantiation of general semantic labeling with models. Hence, it is sound whenever $\mathcal{A}_\mathcal{F}$ is a model of $\mathcal{R}$. However, in general $\mathcal{A}_\mathcal{F}$ does not constitute a model of $\mathcal{R}$. Hence, a transformation technique was introduced that modifies $\mathcal{R}$ in a way that $\mathcal{A}_\mathcal{F}$ always is a model of the result: the *closure under flat contexts*.

▶ **Definition 3.14** (Flat Context Closure)**.** For an n-ary symbol f, the *flat context for the i-th argument* is $\mathcal{FC}^i(f) = f(x_1, \ldots, x_{i-1}, \Box, x_{i+1}, \ldots, x_n)$, where all the x_j are fresh variables. The set of *flat contexts over* $\mathcal{F}$ is defined by $\mathcal{FC}(\mathcal{F}) = \{\mathcal{FC}^i(f) \mid n\text{-ary } f \in \mathcal{F}, 1 \leqslant i \leqslant n\}$. The *closure under flat contexts of a TRS* $\mathcal{R}$ *w.r.t. the signature* $\mathcal{F}$ is given by

$$\mathcal{FC}_\mathcal{F}(\mathcal{R}) = \{C[\ell] \to C[r] \mid C \in \mathcal{FC}(\mathcal{F}), \ell \to r \in \mathcal{R}_\mathsf{a}\} \cup (\mathcal{R} \setminus \mathcal{R}_\mathsf{a})$$

where $\mathcal{R}_\mathsf{a}$ denotes those rules of $\mathcal{R}$, for which the root of the left-hand side and the root of the right-hand side differ.

Since Jambox applies root-labeling recursively (the labeling in Example 3.5 is root-labeling), we definitely would like to aim at a larger class of termination techniques than those which satisfy $TT(\mathcal{R}) \subseteq \mathcal{R}$. A natural extension would be to use the weaker condition $\to_{TT(\mathcal{R})} \subseteq \to_\mathcal{R}$. Then, also root-labeling together with the closure under flat contexts would be supported. Unfortunately, $\to_{TT(\mathcal{R})} \subseteq \to_\mathcal{R}$ does not imply $\to_{\mathcal{D}(\mathcal{U}^m(TT(\mathcal{R})))} \subseteq \to_{\mathcal{D}(\mathcal{U}^m(\mathcal{R}))}$ and thus, does not imply $(\star)$. Moreover, in the following example we show that even if TT is sound and $\to_{TT(\mathcal{R})} \subseteq \to_\mathcal{R}$ then soundness of $\mathsf{lift}(TT)$ cannot be guaranteed.

▶ **Example 3.15.** Consider the TRS $\mathcal{R} = \{\mathsf{f}_1(x) \to \mathsf{f}_0(\mathsf{a}), \mathsf{f}_0(x) \to \mathsf{f}_1(x)\}$. Let TT be the termination technique that replaces $\mathcal{R}$ by $\mathcal{R}' = \{\mathsf{f}_1(\mathsf{a}) \to \mathsf{f}_0(\mathsf{a}), \mathsf{f}_0(x) \to \mathsf{f}_1(x)\}$. Then, TT is sound as $\mathcal{R}'$ is not terminating. Moreover, $\to_{\mathcal{R}'} \subseteq \to_\mathcal{R}$. Nevertheless, $\mathsf{lift}(TT)$ is unsound, since it would replace $(\mathcal{R}, 1)$ by $(\mathcal{R}', 1)$. That this replacement is unsound can be seen as follows: $\mathsf{SN}(\mathcal{R}, 1)$ does not hold since $\mathcal{R}$ is not terminating but the decreasing rules of $\mathcal{R}$ (i.e., $\mathcal{D}(\mathcal{R}) = \{\mathsf{f}_0(x) \to \mathsf{f}_1(x)\}$) are terminating. However, $\mathsf{SN}(\mathcal{R}', 1)$ is satisfied as $\mathcal{D}(\mathcal{R}') = \mathcal{R}'$ and hence termination of $\mathcal{D}(\mathcal{R}')$ implies termination of $\mathcal{R}'$.

We have seen that requiring $TT(\mathcal{R}) \subseteq \mathcal{R}$ is too restrictive to allow root-labeling. But only requiring $\to_{TT(\mathcal{R})} \subseteq \to_\mathcal{R}$ is unsound. However, there is another condition which is weaker than set inclusion, implies soundness, *and* allows the application of flat context closures.

▶ **Definition 3.16.** The *context subset relation* $\subseteq_c$ is defined as

$$\mathcal{R} \subseteq_c \mathcal{S} \text{ iff } \forall \ell \to r \in \mathcal{R}.\, \exists C, \ell' \to r' \in \mathcal{S}.\, \ell = C[\ell'] \wedge r = C[r'].$$

▶ **Lemma 3.17.** *1. $\mathcal{R} \subseteq \mathcal{S}$ implies $\mathcal{R} \subseteq_c \mathcal{S}$*
2. $\mathcal{R} \subseteq_c \mathcal{S}$ implies $\to_\mathcal{R} \subseteq \to_\mathcal{S}$
3. $\mathcal{R} \subseteq_c \mathcal{S}$ implies $\mathcal{D}(\mathcal{R}) \subseteq_c \mathcal{D}(\mathcal{S})$ and $\mathcal{U}(\mathcal{R}) \subseteq_c \mathcal{U}(\mathcal{S})$
4. If TT is sound and $\forall \mathcal{R}.\, TT(\mathcal{R}) \subseteq_c \mathcal{R}$ then $\mathsf{lift}(TT)$ is sound

Proof. 1. To show $\mathcal{R} \subseteq_c \mathcal{S}$, let $\ell \to r \in \mathcal{R}$. Using $\mathcal{R} \subseteq \mathcal{S}$ we know that $\ell \to r \in \mathcal{S}$. Hence, $\exists C, \ell' \to r' \in \mathcal{S}.\, \ell = C[\ell'] \wedge r = C[r']$ by choosing $C = \Box$ and $\ell' \to r' = \ell \to r$.

2. Assume $t = D[\ell\sigma] \to_\mathcal{R} D[r\sigma] = s$ using some rule $\ell \to r \in \mathcal{R}$. As $\mathcal{R} \subseteq_c \mathcal{S}$, we obtain C and $\ell' \to r' \in \mathcal{S}$ such that $\ell = C[\ell']$ and $r = C[r']$. Hence, $t = D[\ell\sigma] = D[C[\ell']\sigma] = D[C\sigma[\ell'\sigma]] \to_\mathcal{S} D[C\sigma[r'\sigma]] = D[C[r']\sigma] = D[r\sigma] = s$.

3. We first show $\mathcal{D}(\mathcal{R}) \subseteq_c \mathcal{D}(\mathcal{S})$. So, let $\ell \to r \in \mathcal{D}(\mathcal{R})$. Hence, $\ell \to r \in \mathcal{R}$, $\mathsf{unlab}(\ell) = \mathsf{unlab}(r)$ and $\ell \neq r$. Using $\mathcal{R} \subseteq_c \mathcal{S}$ we obtain C and $\ell' \to r' \in \mathcal{S}$ such that $\ell = C[\ell']$ and $r = C[r']$. Thus, $\mathsf{unlab}(C)[\mathsf{unlab}(\ell')] = \mathsf{unlab}(C[\ell']) = \mathsf{unlab}(\ell) = \mathsf{unlab}(r) = \mathsf{unlab}(C[r']) =$

$\mathsf{unlab}(C)[\mathsf{unlab}(r')]$ shows that $\mathsf{unlab}(\ell') = \mathsf{unlab}(r')$. Similarly, $C[\ell'] = \ell \neq r = C[r']$ implies $\ell' \neq r'$. So, $\ell' \to r' \in \mathcal{D}(\mathcal{S})$ and thus, $\exists C, \ell' \to r' \in \mathcal{D}(\mathcal{S}).\, \ell = C[\ell'] \wedge r = C[r']$. Now let us show $\mathcal{U}(\mathcal{R}) = \mathsf{unlab}(\mathcal{R} \setminus \mathcal{D}(\mathcal{R})) \subseteq_c \mathsf{unlab}(\mathcal{S} \setminus \mathcal{D}(\mathcal{S})) = \mathcal{U}(\mathcal{S})$. This property is the crucial part, since potentially we remove less rules from $\mathcal{R}$ than from $\mathcal{S}$. Assume $\mathsf{unlab}(\ell) \to \mathsf{unlab}(r) \in \mathcal{U}(\mathcal{R})$, i.e., $\ell \to r \in \mathcal{R}$ and $\mathsf{unlab}(\ell) \neq \mathsf{unlab}(r) \vee \ell = r$. As $\mathcal{R} \subseteq_c \mathcal{S}$ we obtain C and $\ell' \to r' \in \mathcal{S}$ such that $\ell = C[\ell']$ and $r = C[r']$. Hence, $\mathsf{unlab}(\ell) = \mathsf{unlab}(C[\ell']) = \mathsf{unlab}(C)[\mathsf{unlab}(\ell')]$ and $\mathsf{unlab}(r) = \mathsf{unlab}(C[r']) = \mathsf{unlab}(C)[\mathsf{unlab}(r')]$. Thus, we can simplify $\mathsf{unlab}(\ell) \neq \mathsf{unlab}(r) \vee \ell = r$ to $\mathsf{unlab}(C)[\mathsf{unlab}(\ell')] \neq \mathsf{unlab}(C)[\mathsf{unlab}(r')] \vee C[\ell'] = C[r']$ and further to $\mathsf{unlab}(\ell') \neq \mathsf{unlab}(r') \vee \ell' = r'$. Using $\ell' \to r' \in \mathcal{S}$ this shows that $\ell' \to r' \in \mathcal{S} \setminus \mathcal{D}(\mathcal{S})$ and thus, $\mathsf{unlab}(\ell') \to \mathsf{unlab}(r') \in \mathcal{U}(\mathcal{S})$. By choosing the context $\mathsf{unlab}(C)$ and the rule $\mathsf{unlab}(\ell') \to \mathsf{unlab}(r')$ we have finally shown that $\exists C, \ell' \to r' \in \mathcal{U}(\mathcal{S}).\, \mathsf{unlab}(\ell) = C[\ell'] \wedge \mathsf{unlab}(r) = C[r']$.

4. By Lemma 3.12 we only have to prove ($\star$). Using $TT(\mathcal{R}) \subseteq_c \mathcal{R}$ and **3** one can show that $\mathcal{U}^m(TT(\mathcal{R})) \subseteq_c \mathcal{U}^m(\mathcal{R})$ by induction on m. Using **3** again, we conclude $\mathcal{D}(\mathcal{U}^m(TT(\mathcal{R})))$ $\subseteq_c \mathcal{D}(\mathcal{U}^m(\mathcal{R}))$ and thus, $\to_{\mathcal{D}(\mathcal{U}^m(TT(\mathcal{R})))} \subseteq \to_{\mathcal{D}(\mathcal{U}^m(\mathcal{R}))}$ by **2**. Then ($\star$) immediately follows. ◀

▶ **Corollary 3.18.** *Let $\mathcal{R}$ be a TRS over the signature $\mathcal{F}$. Then $\mathsf{lift}(\mathcal{FC}_{\mathcal{F}})$ is sound.*

Proof. It was shown in [16] that $\mathcal{FC}_{\mathcal{F}}$ is sound for TRSs. Furthermore, $\mathcal{FC}_{\mathcal{F}}(\mathcal{R}) \subseteq_c \mathcal{R}$ by definition of $\mathcal{FC}(\mathcal{F})$ and thus, by Lemma 3.17(**4**), $\mathsf{lift}(\mathcal{FC}_{\mathcal{F}})$ is sound, too. ◀

Note that several termination techniques TT satisfy $TT(\mathcal{R}) \subseteq_c \mathcal{R}$ and hence, can be used between labeling and unlabeling. However, there are still some techniques which do not satisfy this requirement. Examples would be string reversal and uncurrying [11].

Of course, it is possible to use $\mathsf{lift}_0(TT)$, however, for string reversal also a direct soundness proof can be performed to show that lifting string reversal is sound.

▶ **Theorem 3.19.** *Let TT be the technique of string reversal where $TT(\mathcal{R}) = \mathsf{rev}(\mathcal{R})$, if $\mathcal{R}$ is a string rewrite system, and $TT(\mathcal{R}) = \mathcal{R}$, otherwise. Then $\mathsf{lift}(TT)$ is sound.*

Proof. By Lemma 3.12 we just have to prove ($\star$), i.e., we have to show for all m that $\mathsf{SN}(\mathcal{D}(\mathcal{U}^m(\mathcal{R})))$ implies $\mathsf{SN}(\mathcal{D}(\mathcal{U}^m(\mathsf{rev}(\mathcal{R}))))$. To this end, we have proven that reversing can be commuted with both $\mathcal{D}$ and $\mathcal{U}$: $\mathsf{rev}(\mathcal{D}(\mathcal{R})) = \mathcal{D}(\mathsf{rev}(\mathcal{R}))$ and $\mathsf{rev}(\mathcal{U}(\mathcal{R})) = \mathcal{U}(\mathsf{rev}(\mathcal{R}))$. Hence, $\mathsf{rev}(\mathcal{D}(\mathcal{U}^m(\mathcal{R}))) = \mathcal{D}(\mathcal{U}^m(\mathsf{rev}(\mathcal{R})))$. This completes the proof: since string reversal is complete, we know that termination of $\mathcal{D}(\mathcal{U}^m(\mathcal{R}))$ implies termination of $\mathsf{rev}(\mathcal{D}(\mathcal{U}^m(\mathcal{R})))$ and therefore, also of $\mathcal{D}(\mathcal{U}^m(\mathsf{rev}(\mathcal{R})))$. ◀

To summarize, we can now certify termination proofs where labeling and unlabeling are modular techniques (and hence, can be applied recursively), and where all supported techniques of CeTA (except uncurrying) can be used between labeling and unlabeling.

An easy alternative to our extended termination techniques would be the use of relative rewriting. The obvious idea is to add the decreasing rules as relative rules when performing semantic labeling. In this way, unlabeling would directly be modular and sound, since one can always remove relative rules where both sides of the rule are identical. This alternative is used in the independent and unpublished formalization of semantic labeling in the CoLoR library. The main problem with this alternative is that some techniques like RFC matchbounds can be used in our framework, but not in combination with relative rewriting in general (during the termination competition in 2010 a tool has been disqualified for giving a wrong answer for a relative termination problem; the reason was the use of RFC matchbounds). For a further discussion on matchbounds and relative rewriting we refer to [12].

4 Dependency Pair Framework

The DP framework [8] is a way to modularize termination proofs. Instead of TRSs one investigates 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 $\mathsf{DP}(\mathcal{R})$ denotes the *dependency pairs* of $\mathcal{R}$ [1]. A $(\mathcal{P}, \mathcal{R})$-*chain* is a possibly infinite derivation of the 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 $i > 0$. 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* [8], if there is no minimal infinite $(\mathcal{P}, \mathcal{R})$-chain. Proving finiteness of a DP problem is done by simplifying $(\mathcal{P}, \mathcal{R})$ using so called *processors* recursively. A processor transforms a DP problem into a new DP problem. The aim is to reach a DP problem where the $\mathcal{P}$-component is empty (such DP problems are trivially finite). To conclude finiteness of the initial DP problem, the applied processors need to be *sound*. A processor $\mathcal{P}\mathrm{roc}$ is sound whenever for all DP problems $(\mathcal{P}, \mathcal{R})$ we have that finiteness of $\mathcal{P}\mathrm{roc}(\mathcal{P}, \mathcal{R})$ implies finiteness of $(\mathcal{P}, \mathcal{R})$.

Semantic labeling can easily be lifted to DP problems. Soundness of the following processor is an immediate consequence of [19].

▶ **Theorem 4.1.** *Let $(\mathcal{P}, \mathcal{R})$ be a DP problem and $\mathcal{A}$ be an algebra. If $\mathcal{A}$ is a quasi-model of $\mathcal{R}$, then it is sound to return the DP problem* $(\mathsf{lab}(\mathcal{P}), \mathsf{lab}(\mathcal{R}) \cup \mathcal{D}\mathrm{ec})$.

The following example shows that unlabeling is not only necessary for efficiency, but that unlabeling is required to apply other techniques.

▶ **Example 4.2.** We consider the TRS `Secret_07/4` from the TPDB.

$$\begin{array}{lll} 1: & \mathsf{g}(\mathsf{c}, \mathsf{g}(\mathsf{c}, x)) \to \mathsf{g}(\mathsf{e}, \mathsf{g}(\mathsf{d}, x)) & 4: \quad \mathsf{g}(x, \mathsf{g}(y, \mathsf{g}(x, y))) \to \mathsf{g}(\mathsf{a}, \mathsf{g}(x, \mathsf{g}(y, \mathsf{b}))) \\ 2: & \mathsf{g}(\mathsf{d}, \mathsf{g}(\mathsf{d}, x)) \to \mathsf{g}(\mathsf{c}, \mathsf{g}(\mathsf{e}, x)) & 5: \quad \mathsf{f}(\mathsf{g}(x, y)) \to \mathsf{g}(y, \mathsf{g}(\mathsf{f}(\mathsf{f}(x)), \mathsf{a})) \\ 3: & \mathsf{g}(\mathsf{e}, \mathsf{g}(\mathsf{e}, x)) \to \mathsf{g}(\mathsf{d}, \mathsf{g}(\mathsf{c}, x)) & \end{array}$$

In the 2008 termination competition AProVE found a termination proof of the following structure (we present a simplified version, missing some unnecessary steps that have been applied in the original proof).[4] First, the initial DP problem is transformed into $(\mathcal{P}, \{1\text{–}4\})$ where $\mathcal{P}$ consists of the pairs $\mathsf{G}(\mathsf{c}, \mathsf{g}(\mathsf{c}, x)) \to \mathsf{G}(\mathsf{e}, \mathsf{g}(\mathsf{d}, x))$, $\mathsf{G}(\mathsf{d}, \mathsf{g}(\mathsf{d}, x)) \to \mathsf{G}(\mathsf{c}, \mathsf{g}(\mathsf{e}, x))$, and $\mathsf{G}(\mathsf{e}, \mathsf{g}(\mathsf{e}, x)) \to \mathsf{G}(\mathsf{d}, \mathsf{g}(\mathsf{c}, x))$. Then, labeling and further processing yields the DP problem $(\mathcal{P}', \mathcal{R}')$ where $\mathcal{P}'$ contains the pairs

$$\begin{array}{ll} \mathsf{G}_{00}(\mathsf{c}, \mathsf{g}_{00}(\mathsf{c}, x)) \to \mathsf{G}_{00}(\mathsf{e}, \mathsf{g}_{00}(\mathsf{d}, x)) & \mathsf{G}_{00}(\mathsf{e}, \mathsf{g}_{00}(\mathsf{e}, x)) \to \mathsf{G}_{00}(\mathsf{d}, \mathsf{g}_{00}(\mathsf{c}, x)) \\ \mathsf{G}_{00}(\mathsf{d}, \mathsf{g}_{00}(\mathsf{d}, x)) \to \mathsf{G}_{00}(\mathsf{c}, \mathsf{g}_{00}(\mathsf{e}, x)) & \end{array}$$

and $\mathcal{R}'$ is the following TRS.

$$\begin{array}{ll} \mathsf{g}_{00}(\mathsf{c}, \mathsf{g}_{00}(\mathsf{c}, x)) \to \mathsf{g}_{00}(\mathsf{e}, \mathsf{g}_{00}(\mathsf{d}, x)) & \mathsf{g}_{00}(\mathsf{c}, \mathsf{g}_{01}(\mathsf{c}, x)) \to \mathsf{g}_{00}(\mathsf{e}, \mathsf{g}_{01}(\mathsf{d}, x)) \\ \mathsf{g}_{00}(\mathsf{d}, \mathsf{g}_{00}(\mathsf{d}, x)) \to \mathsf{g}_{00}(\mathsf{c}, \mathsf{g}_{00}(\mathsf{e}, x)) & \mathsf{g}_{00}(\mathsf{d}, \mathsf{g}_{01}(\mathsf{d}, x)) \to \mathsf{g}_{00}(\mathsf{c}, \mathsf{g}_{01}(\mathsf{e}, x)) \\ \mathsf{g}_{00}(\mathsf{e}, \mathsf{g}_{00}(\mathsf{e}, x)) \to \mathsf{g}_{00}(\mathsf{d}, \mathsf{g}_{00}(\mathsf{c}, x)) & \mathsf{g}_{00}(\mathsf{e}, \mathsf{g}_{01}(\mathsf{e}, x)) \to \mathsf{g}_{00}(\mathsf{d}, \mathsf{g}_{01}(\mathsf{c}, x)) \end{array}$$

[4] See `http://termcomp.uibk.ac.at/termcomp/competition/resultDetail.seam?resultId=35909`

Hence, all labeled versions of Rule 4 have been deleted, and unlabeling yields the DP problem $(\mathcal{P}, \{1\text{–}3\})$. This DP problem is applicative. Hence, we may apply the $\mathcal{A}$-transformation [9] to obtain the DP problem having the pairs

$$\mathsf{C}(\mathsf{c}(x)) \to \mathsf{E}(\mathsf{d}(x)) \qquad \mathsf{D}(\mathsf{d}(x)) \to \mathsf{C}(\mathsf{e}(x)) \qquad \mathsf{E}(\mathsf{e}(x)) \to \mathsf{D}(\mathsf{c}(x))$$

and the rules

$$\mathsf{c}(\mathsf{c}(x)) \to \mathsf{e}(\mathsf{d}(x)) \qquad \mathsf{d}(\mathsf{d}(x)) \to \mathsf{c}(\mathsf{e}(x)) \qquad \mathsf{e}(\mathsf{e}(x)) \to \mathsf{d}(\mathsf{c}(x))$$

This DP problem is solved using standard techniques. Note that for the $\mathcal{A}$-transformation it was essential that unlabeling was performed, as the DP problem $(\mathcal{P}', \mathcal{R}')$ is not applicative.

Unfortunately, unlabeling as processor is in general unsound. In contrast to unlabeling on TRSs, here a problem already arises when using the model-version of semantic labeling without decreasing rules. The main reason is that unlabeling might introduce nontermination. Hence, minimality of an unlabeled infinite chain cannot be guaranteed.[5]

▶ **Example 4.3.** Consider the DP problem $(\mathcal{P}, \varnothing)$ where $\mathcal{P} = \{\mathsf{F}(x) \to \mathsf{F}(\mathsf{g}(\mathsf{a}))\}$. This DP problem is obviously not finite. Applying semantic labeling is trivially possible since there are no rules which have to satisfy the (quasi-)model condition. We choose $A = \{0, 1\}$, and for each f we define $f_{\mathcal{A}}(\ldots) = 0$ and $\ell_f(\vec{x}_n) = (\vec{x}_n)$. We obtain the labeled pairs $\mathsf{lab}(\mathcal{P}) = \{\mathsf{F}_0(x) \to \mathsf{F}_0(\mathsf{g}_0(\mathsf{a})), \mathsf{F}_1(x) \to \mathsf{F}_0(\mathsf{g}_0(\mathsf{a}))\}$ and by Theorem 4.1 we know that the DP problem $(\mathsf{lab}(\mathcal{P}), \varnothing)$ is again not finite. We can further modify the DP problem by replacing it with $(\mathsf{lab}(\mathcal{P}), \mathcal{R})$ where $\mathcal{R} = \{\mathsf{g}_1(x) \to \mathsf{g}_1(x)\}$. Note that this modification is sound since $(\mathsf{lab}(\mathcal{P}), \mathcal{R})$ still allows a minimal infinite chain and is therefore not finite.

However, applying unlabeling we obtain the DP problem $(\mathcal{P}, \mathsf{unlab}(\mathcal{R}))$ which is finite as now the right-hand side $\mathsf{F}(\mathsf{g}(\mathsf{a}))$ of the only pair in $\mathcal{P}$ is not terminating w.r.t. $\mathcal{U}(\mathcal{R}) = \{\mathsf{g}(x) \to \mathsf{g}(x)\}$. Hence, unlabeling is unsound in general. The main problem is again that the notion of soundness is too weak. It allows the application of processors between labeling and unlabeling which may replace $(\mathsf{lab}(\mathcal{P}), \varnothing)$ by $(\mathsf{lab}(\mathcal{P}), \mathcal{R})$.

To solve this problem, we again add a counter n which tells us how often we may unlabel.

▶ **Definition 4.4.** An *extended DP problem* is a triple $(\mathcal{P}, \mathcal{R}, n)$ where $(\mathcal{P}, \mathcal{R})$ is a DP problem and $n \in \mathbb{N}$. An extended DP problem $(\mathcal{P}, \mathcal{R}, n)$ is *finite* iff there is no infinite chain

$$s_1\sigma_1 \to_{\mathcal{P}} t_1\sigma_1 \to^*_{\mathcal{R}} s_2\sigma_2 \to_{\mathcal{P}} t_2\sigma_2 \to^*_{\mathcal{R}} s_3\sigma_3 \to_{\mathcal{P}} t_3\sigma_3 \to^*_{\mathcal{R}} \cdots$$

such that for all i: $\forall m \leqslant n.\, \mathsf{SN}_{\mathcal{U}^m(\mathcal{R})}(\mathsf{unlab}^m(t_i\sigma_i))$.

Hence, the only difference between finiteness of DP problems and extended DP problems is the minimality condition ($\mathsf{SN}_{\mathcal{R}}(t_i\sigma_i)$ versus $\forall m \leqslant n.\, \mathsf{SN}_{\mathcal{U}^m(\mathcal{R})}(\mathsf{unlab}^m(t_i\sigma_i))$). We therefore obtain a similar lemma to Lemma 3.7, but now for DP problems.

▶ **Lemma 4.5.** *1. $(\mathcal{P}, \mathcal{R})$ is finite iff $(\mathcal{P}, \mathcal{R}, 0)$ is finite.*
2. If $(\mathcal{P}, \mathcal{R})$ is finite then $(\mathcal{P}, \mathcal{R}, n)$ is finite.

As for termination techniques we can lift every processor to an extended processor.

[5] There is no problem in the formalization of semantic labeling in CoLoR at this point, as it does not feature *minimal* chains.

▶ **Definition 4.6** (Lift). Let $\mathcal{P}\mathrm{roc}$ be a processor with $\mathcal{P}\mathrm{roc}(\mathcal{P},\mathcal{R}) = (\mathcal{P}',\mathcal{R}')$. Then $\mathsf{lift}(\mathcal{P}\mathrm{roc})$ and $\mathsf{lift}_0(\mathcal{P}\mathrm{roc})$ are extended processors where $\mathsf{lift}(\mathcal{P}\mathrm{roc})(\mathcal{P},\mathcal{R},n) = (\mathcal{P}',\mathcal{R}',n)$ and $\mathsf{lift}_0(\mathcal{P}\mathrm{roc})(\mathcal{P},\mathcal{R},n) = (\mathcal{P}',\mathcal{R}',0)$.

We obtain similar results for lift_0 as for termination techniques: whenever $\mathcal{P}\mathrm{roc}$ is sound then $\mathsf{lift}_0(\mathcal{P}\mathrm{roc})$ is sound. However, additionally demanding that $\mathcal{R}' \subseteq_c \mathcal{R}$ or even $\mathcal{P}' \subseteq \mathcal{P} \wedge \mathcal{R}' = \mathcal{R}$ where $\mathcal{P}\mathrm{roc}(\mathcal{P},\mathcal{R}) = (\mathcal{P}',\mathcal{R}')$ does not suffice to ensure soundness of $\mathsf{lift}(\mathcal{P}\mathrm{roc})$. This is demonstrated in the upcoming example.

▶ **Example 4.7.** Let $\mathcal{P} = \{\mathsf{F}_0(x) \to \mathsf{F}_0(\mathsf{b})\}$, $\mathcal{P}' = \{\mathsf{F}_0(x) \to \mathsf{F}_0(\mathsf{g}_0(\mathsf{b}))\}$, and $\mathcal{R} = \{\mathsf{g}_1(x) \to \mathsf{g}_0(\mathsf{h}_1(x))\}$. Then $(\mathcal{P},\mathcal{R},1)$ is not finite as obviously there is an infinite $(\mathcal{P},\mathcal{R})$-chain where all terms in the chain are $\mathsf{F}_0(\mathsf{b})$ and moreover, $\mathsf{F}_0(\mathsf{b})$ is terminating w.r.t. $\mathcal{R}$ and $\mathsf{unlab}(\mathsf{F}_0(\mathsf{b})) = \mathsf{F}(\mathsf{b})$ is terminating w.r.t. $\mathcal{U}(\mathcal{R}) = \{\mathsf{g}(x) \to \mathsf{g}(\mathsf{h}(x))\}$. Hence, also $(\mathcal{P} \cup \mathcal{P}',\mathcal{R},1)$ is not finite by constructing the same chain.

Note that the processor $\mathcal{P}\mathrm{roc}$ which replaces $(\mathcal{P} \cup \mathcal{P}',\mathcal{R})$ by $(\mathcal{P}',\mathcal{R})$ is sound, since $(\mathcal{P}',\mathcal{R})$ is not finite: again, there is an infinite $(\mathcal{P}',\mathcal{R})$-chain, and every chain is also minimal since $\mathcal{R}$ is terminating. However, $\mathsf{lift}(\mathcal{P}\mathrm{roc})$ is unsound as $(\mathcal{P}',\mathcal{R},1)$ is finite: otherwise, there would be an infinite chain where $\mathsf{F}_0(\mathsf{g}_0(\mathsf{b}))$ is terminating w.r.t. $\mathcal{R}$ and $\mathsf{unlab}(\mathsf{F}_0(\mathsf{g}_0(\mathsf{b}))) = \mathsf{F}(\mathsf{g}(\mathsf{b}))$ is terminating w.r.t. $\mathcal{U}(\mathcal{R})$. But it is easy to see that $\mathsf{F}(\mathsf{g}(\mathsf{b}))$ is not terminating w.r.t. $\mathcal{U}(\mathcal{R})$.

Since requiring just $\mathcal{R}' \subseteq_c \mathcal{R}$ (or even $\mathcal{P}' \subseteq \mathcal{P} \wedge \mathcal{R}' = \mathcal{R}$) does not suffice to ensure soundness of $\mathsf{lift}(\mathcal{P}\mathrm{roc})$ we demand a slightly stronger property than soundness.

▶ **Definition 4.8.** A processor $\mathcal{P}\mathrm{roc}$ is *chain-identifying* iff whenever $\mathcal{P}\mathrm{roc}(\mathcal{P},\mathcal{R}) = (\mathcal{P}',\mathcal{R}')$ and there is some minimal infinite $(\mathcal{P},\mathcal{R})$-chain

$$s_1\sigma_1 \to_{\mathcal{P}} t_1\sigma_1 \to^*_{\mathcal{R}} s_2\sigma_2 \to_{\mathcal{P}} t_2\sigma_2 \to^*_{\mathcal{R}} s_3\sigma_3 \to_{\mathcal{P}} t_3\sigma_3 \to^*_{\mathcal{R}} \cdots$$

then $\mathcal{R}' \subseteq_c \mathcal{R}$ and there is some k such that

$$s_k\sigma_k \to_{\mathcal{P}'} t_k\sigma_k \to^*_{\mathcal{R}'} s_{k+1}\sigma_{k+1} \to_{\mathcal{P}'} t_{k+1}\sigma_{k+1} \to^*_{\mathcal{R}'} s_{k+2}\sigma_{k+2} \to_{\mathcal{P}'} t_{k+2}\sigma_{k+2} \to^*_{\mathcal{R}'} \cdots$$

is an infinite $(\mathcal{P}',\mathcal{R}')$-chain.

Chain-identifying processors ensure that every minimal infinite chain of $(\mathcal{P},\mathcal{R})$ has an infinite tail where $\mathcal{R}^*$-steps can be replaced by $\mathcal{R}'^*$-steps and all pairs are from $\mathcal{P}'$. Note that every chain-identifying processor is sound. Moreover, several processors are indeed chain-identifying. Some examples are the reduction pair processor, the dependency graph processor, and all standard processors which just remove pairs and rules. The following lemma shows that chain-identifying processors can be used as extended processors via lift.

▶ **Lemma 4.9.** ***1.*** *If* $\mathcal{P}\mathrm{roc}$ *is sound, then* $\mathsf{lift}_0(\mathcal{P}\mathrm{roc})$ *is sound.*
2. *If* $\mathcal{P}\mathrm{roc}$ *is chain-identifying then* $\mathsf{lift}(\mathcal{P}\mathrm{roc})$ *is sound.*

Proof. Let $\mathcal{P}$, $\mathcal{R}$, $\mathcal{P}'$, and $\mathcal{R}'$ be given such that $\mathcal{P}\mathrm{roc}(\mathcal{P},\mathcal{R}) = (\mathcal{P}',\mathcal{R}')$.

1. We assume that $(\mathcal{P}',\mathcal{R}',0)$ is finite and have to show that $(\mathcal{P},\mathcal{R},n)$ is finite. By Lemma 4.5(**1**) and the assumption we know that $(\mathcal{P}',\mathcal{R}')$ is finite. Thus, also $(\mathcal{P},\mathcal{R})$ is finite using the soundness of $\mathcal{P}\mathrm{roc}$. By Lemma 4.5(**2**) we conclude finiteness of $(\mathcal{P},\mathcal{R},n)$.
2. Here, we may assume that $(\mathcal{P}',\mathcal{R}',n)$ is finite and have to show that $(\mathcal{P},\mathcal{R},n)$ is finite. We show finiteness of $(\mathcal{P},\mathcal{R},n)$ via contraposition. So, assume $(\mathcal{P},\mathcal{R},n)$ is not finite. This shows that there is an infinite $(\mathcal{P},\mathcal{R})$-chain

$$s_1\sigma_1 \to_{\mathcal{P}} t_1\sigma_1 \to^*_{\mathcal{R}} s_2\sigma_2 \to_{\mathcal{P}} t_2\sigma_2 \to^*_{\mathcal{R}} s_3\sigma_3 \to_{\mathcal{P}} t_3\sigma_3 \to^*_{\mathcal{R}} \cdots$$

such that for all i we have $\forall m \leqslant n.\, \mathsf{SN}_{\mathcal{U}^m(\mathcal{R})}(\mathsf{unlab}^m(t_i\sigma_i))$. By choosing $m = 0$ we also have $\mathsf{SN}_{\mathcal{R}}(t_i\sigma_i)$ for all i. Hence, the chain is also a minimal infinite $(\mathcal{P}, \mathcal{R})$-chain. Since $\mathcal{P}\mathrm{roc}$ is chain-identifying we know that $\mathcal{R}' \subseteq_c \mathcal{R}$ and there is some k such that

$$s_k\sigma_k \to_{\mathcal{P}'} t_k\sigma_k \to^*_{\mathcal{R}'} s_{k+1}\sigma_{k+1} \to_{\mathcal{P}'} t_{k+1}\sigma_{k+1} \to^*_{\mathcal{R}'} s_{k+2}\sigma_{k+2} \to_{\mathcal{P}'} t_{k+2}\sigma_{k+2} \to^*_{\mathcal{R}'} \cdots$$

is an infinite $(\mathcal{P}', \mathcal{R}')$-chain. We continue to prove that for every i and every $m \leqslant n$ we have $\mathsf{SN}_{\mathcal{U}^m(\mathcal{R}')}(\mathsf{unlab}^m(t_i\sigma_i))$. This leads to the desired contradiction, since then we have shown that $(\mathcal{P}', \mathcal{R}', n)$ is not finite.
To prove $\mathsf{SN}_{\mathcal{U}^m(\mathcal{R}')}(\mathsf{unlab}^m(t_i\sigma_i))$ we first use minimality of the $(\mathcal{P}, \mathcal{R})$-chain to conclude $\mathsf{SN}_{\mathcal{U}^m(\mathcal{R})}(\mathsf{unlab}^m(t_i\sigma_i))$. Then the result immediately follows since the rewrite relation of $\mathcal{U}^m(\mathcal{R}')$ is a subset of the rewrite relation of $\mathcal{U}^m(\mathcal{R})$ by Lemma 3.17, **2** and **3**. ◀

Using these results allowed us to develop the first certified proof of the TRS in Example 4.2. We only had to change the given proof such that uncurrying [11] is used instead of the $\mathcal{A}$-transformation, since we have only formalized the former technique. The detailed proof is provided in the IsaFoR/CeTA repository.[6]

However, unlike for TRSs, root-labeling is not directly supported as root-labeling on DP problems [16, 17] is not a chain-identifying processor. Here again, root-labeling itself is not the problem, but making sure that the fixed algebra is a model of $\mathcal{R}$, which is again done by closing under flat contexts. In the DP framework we need the auxiliary function $\mathsf{block}_\triangle$, given by the equations $\mathsf{block}_\triangle(f(\vec{t}_n)) = f(\triangle(\vec{t}_n))$ and $\mathsf{block}_\triangle(x) = x$.

▶ **Definition 4.10** (Flat Context Closure)**.** Let $(\mathcal{P}, \mathcal{R})$ be a DP problem such that $\mathcal{R}$ is left-linear and $\mathcal{F}$ is a superset of the signature of $\mathcal{R}$ combined with the non-root symbols of $\mathcal{P}$. Furthermore, let $\triangle$ be a function symbol not in $\mathcal{F}$. Then the closure under flat contexts of $(\mathcal{P}, \mathcal{R})$ is given by $\mathcal{FC}_{\mathcal{F}}(\mathcal{P}, \mathcal{R}) = (\mathsf{block}_\triangle(\mathcal{P}), \mathcal{FC}_{\{\triangle\}\cup\mathcal{F}}(\mathcal{R}))$.

As the pairs of a DP problem are modified, we do not get soundness of $\mathsf{lift}(\mathcal{FC}_{\mathcal{F}})$ via Lemma 4.9. Nevertheless, by using the definition of finiteness of extended DP problems and providing a manual proof one can show that $\mathsf{lift}(\mathcal{FC}_{\mathcal{F}})$ is indeed sound.

5 Problems in Certification

We present three problems that arose when trying to certify proofs with semantic labeling.

The first problem for the certifier is that internally it only works on extended termination/DP problems, whereas in the provided proofs just TRSs and DP problems are given without the additional numbers. However, this problem is fixed by computing the number during certification. This is easy and seems to be a safe solution: the format for termination proofs remains unchanged, and so far no termination proof was refused with the reason that the internal computation of the number was wrong.

The second and third problem are concerned with how semantic labeling is applied, since usually variations of Lemma 3.9 and Theorem 4.1 are used in termination provers.

The second problem occurs for TRSs as well as DP problems. The theory about semantic labeling demands that $\mathcal{D}\mathrm{ec}$ is added to the new TRS when using quasi-models. However, termination provers typically reduce the set of rules and "optimize" semantic labeling by only adding rules $\mathcal{D}\mathrm{ec}'$ such that $\to_{\mathcal{D}\mathrm{ec}} \subseteq \to^+_{\mathcal{D}\mathrm{ec}'}$.

[6] `http://cl2-informatik.uibk.ac.at/rewriting/mercurial.cgi/IsaFoR/raw-file/v1.16/examples/secret_07_trs_4_top.proof.xml`

For example, if $L_f = \{0, 1, 2\}$ and the order is the standard order on the naturals, then $\mathcal{D}\mathrm{ec} = \{f_2(x) \to f_1(x), f_1(x) \to f_0(x), f_2(x) \to f_0(x)\}$. However, the last rule is often omitted since it can be simulated by the previous two rules. To certify these termination proofs, we fist need to show that we may safely replace $\mathcal{D}\mathrm{ec}$ by any $\mathcal{D}\mathrm{ec}'$ where $\to_{\mathcal{D}\mathrm{ec}} \subseteq \to^+_{\mathcal{D}\mathrm{ec}'}$. Moreover, we have to provide a certified algorithm which for a given TRS $\mathcal{D}\mathrm{ec}'$ and a given order can ensure that the condition $\to_{\mathcal{D}\mathrm{ec}} \subseteq \to^+_{\mathcal{D}\mathrm{ec}'}$ is satisfied. Furthermore, the algorithm should accept *all* $\mathcal{D}\mathrm{ec}'$ where the condition is satisfied.

The third problem only occurs when dealing with quasi-models in the DP framework. Note that in standard DP problems the roots of $\mathcal{P}$ are special symbols (tuple symbols) which do not occur in the remaining DP problem. However, when applying Theorem 4.1 as it is, this invariant is destroyed since the decreasing rules for tuple symbols are added as new rules. We illustrate the problem and two possible solutions in the following example.

▶ **Example 5.1.** Consider a DP problem $(\mathcal{P}, \mathcal{R})$ where $\mathsf{F}(\mathsf{s}(x), \mathsf{a}) \to \mathsf{F}(x, \mathsf{b}) \in \mathcal{P}$ and b is not defined in $\mathcal{R}$. Then, the dependency graph estimation EDG [1] can detect that there is no connection from the mentioned pair to itself. However, when performing labeling with a quasi-model where $\mathsf{s}(x)$ is interpreted as $\min(x + 1, 2)$ and where $\ell_{\mathsf{F}}(x, y) = x$ then for the mentioned pair we get all three rules $\{6, 8, 10\}$ in the labeled pairs $\mathcal{P}'$ and the decreasing rules for F are $\mathcal{D}\mathrm{ec}_{\mathsf{F}} = \{7, 9, 11\}$.

$$\begin{array}{rrrrrr} 6: & \mathsf{F}_2(\mathsf{s}(x), \mathsf{a}) \to \mathsf{F}_2(x, \mathsf{b}) & 8: & \mathsf{F}_2(\mathsf{s}(x), \mathsf{a}) \to \mathsf{F}_1(x, \mathsf{b}) & 10: & \mathsf{F}_1(\mathsf{s}(x), \mathsf{a}) \to \mathsf{F}_0(x, \mathsf{b}) \\ 7: & \mathsf{F}_2(x, y) \to \mathsf{F}_1(x, y) & 9: & \mathsf{F}_2(x, y) \to \mathsf{F}_0(x, y) & 11: & \mathsf{F}_1(x, y) \to \mathsf{F}_0(x, y) \end{array}$$

Note that when adding $\mathcal{D}\mathrm{ec}_{\mathsf{F}}$ as new rules, then the EDG contains an edge from $\mathsf{F}_2(\mathsf{s}(x), \mathsf{a}) \to \mathsf{F}_1(x, \mathsf{b})$ to all other pairs since F_1 is defined in $\mathcal{D}\mathrm{ec}_{\mathsf{F}}$. Hence, this is not the preferred way to add decreasing rules: not even the decrease in the labels is recognized.

One solution is to add $\mathcal{D}\mathrm{ec}_{\mathsf{F}}$ as new pairs. Then one obtains a standard DP problem and the decrease in the labels is reflected in the EDG. But there still is a path from $\mathsf{F}_2(\mathsf{s}(x), \mathsf{a}) \to \mathsf{F}_2(x, \mathsf{b})$ to $\mathsf{F}_1(\mathsf{s}(x), \mathsf{a}) \to \mathsf{F}_0(x, \mathsf{b})$ via the pair $\mathsf{F}_2(x, y) \to \mathsf{F}_1(x, y)$, since the information that the second argument of F_n is b is lost when passing the pair $\mathsf{F}_2(x, y) \to \mathsf{F}_1(x, y)$.

To encounter this problem, there is another solution where $\mathcal{D}\mathrm{ec}_{\mathsf{F}}$ is not produced at all, but where the labels of all tuple-symbols in right-hand sides of $\mathcal{P}'$ are decreased. In this example, one would have to add the additional pair $\mathsf{F}_2(\mathsf{s}(x), \mathsf{a}) \to \mathsf{F}_0(x, \mathsf{b})$ to $\mathcal{P}'$.

Hence, termination proofs might have used one of the two variants instead of Theorem 4.1. Here, the first variant returns the problem $(\mathsf{lab}(\mathcal{P}) \cup \mathcal{D}\mathrm{ec}_{\mathcal{F}^\sharp}, \mathsf{lab}(\mathcal{R}) \cup \mathcal{D}\mathrm{ec}_{\mathcal{F}})$ and the second variant returns $(\mathsf{lab}(\mathcal{P})^{\geq}, \mathsf{lab}(\mathcal{R}) \cup \mathcal{D}\mathrm{ec}_{\mathcal{F}})$ where $\mathcal{D}\mathrm{ec}_{\mathcal{F}^\sharp}$ are the decreasing rules for all tuple symbols, $\mathcal{D}\mathrm{ec}_{\mathcal{F}}$ are the decreasing rules for the remaining symbols, and $\mathsf{lab}(\mathcal{P})^{\geq} = \{s \to f_{\ell'}(\vec{t}) \mid s \to f_\ell(\vec{t}) \in \mathsf{lab}(\mathcal{P}), \ell \geq_{L_f} \ell'\}$.

To certify these termination proofs the problem was mainly in formalizing that these variants of Theorem 4.1 are indeed sound.

▶ **Theorem 5.2.** *Both variants of Theorem 4.1 are sound, provided that they are applied on DP problems $(\mathcal{P}, \mathcal{R})$ where neither left- nor right-hand sides of $\mathcal{P}$ are variables and the roots of $\mathcal{P}$ are distinguished tuple symbols which do not occur in the remaining DP problem.*

We shortly describe the proof idea. The main problem is that we cannot w.l.o.g. restrict the substitutions in a chain such that they do not contain tuple symbols [17]. Thus, we may have to apply rules in $\mathcal{D}\mathrm{ec}_{\mathcal{F}^\sharp}$ also below the root, in order to simulate a reduction $t_i\sigma_i \to^*_{\mathcal{R}} s_{i+1}\sigma_{i+1}$. The trick is to introduce a second set of labels and labeling functions for the tuple symbols. The new labeling functions label all tuple symbols by the same element.

Hence, no decreasing rules are required for them (w.r.t. the second set of labeling functions) and on all other symbols the labeling functions coincide.

Afterwards, we use a combined labeling of terms: The root of the term is labeled according to the original function, and below the root it is labeled w.r.t. the second labeling function. In this way no decreasing rules for the tuple symbols have to be applied below the root and moreover, on all terms in the DP problem, the original and the combined labeling produce the same result. Thus, we can transform a given $(\mathcal{P}, \mathcal{R})$-chain into a $(\mathsf{lab}(\mathcal{P}) \cup \mathcal{D}\mathrm{ec}_{\mathcal{F}^\sharp}, \mathsf{lab}(\mathcal{R}) \cup \mathcal{D}\mathrm{ec}_{\mathcal{F}})$-chain. Theorem 5.2 easily follows.

To summarize, we discussed some problems which occurred when trying to certify existing proofs which are mainly due to optimizations of the basic semantic labeling theorems. Of course, we also need to check the model condition, whether the orders are weakly monotone when using quasi-orders, etc. Whereas the general theorems about soundness of semantic labeling have been formalized for arbitrary carriers, for the certification we currently only support finite carriers. Then checking the required conditions is performed via enumerating all possible assignments.

In total, our formalization of pure semantic labeling consists of 3300 lines of Isabelle, where roughly half of it is about semantic labeling on generic algebras, and the other half contains executable functions for the certifier using algebras over finite carriers and soundness proofs for these functions. Moreover, the theory about the semantic labeling framework with extended termination techniques, extended DP problems, etc., consists of another 1000 lines.

6 Experiments

To test the impact of our formalization we ran AProVE on the TPDB (version 8.0), considering all 2795 TRSs. We used two different strategies which are similar to the strategy CERT that was used during the 2010 termination competition in the certified termination category: −SL is like CERT but with semantic labeling removed, and +SL is like CERT including all three variants of semantic labeling that are supported by AProVE (root-labeling, semantic-labeling on finite carriers with models and quasi-models).

We performed all our experiments on a machine with two 2.8 GHz Quad-Core Intel Xeon processors and 6 GB of main memory. The following results where obtained using a 60 seconds timeout.

	−SL	+SL	total
termination proofs	1137	1207	1227
nontermination proofs	225	218	227
total time (in minutes)	1186	1219	
certification time (in minutes)	1	3	

CeTA (version 1.17) certified all but two proofs. On one TRS, both −SL and +SL delivered a faulty proof, caused by a bug in the LPO output of AProVE (which will be fixed soonish).

The results show that by using semantic labeling we obtain 90 new certified termination proofs. This is an increase of nearly 8 %. Note that +SL has not solved all TRSs where −SL was successful. This is due to timing issues in the strategy.

7 Conclusion

During our formalization of semantic labeling we have detected that unlabeling is unsound when using the current semantics of termination problems. We solved the problem by

extending termination problems and the DP framework such that recursive labeling and unlabeling are supported, as well as all other existing termination techniques. This framework forms the semantic basis of our certifier CeTA which now fully supports semantic labeling.

Acknowledgments We thank Christian Kuknat and Carsten Fuhs for their support in providing certifiable proofs with semantic labeling generated by AProVE.

References

1 T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theor. Comput. Sci.*, 236(1-2):133–178, 2000.
2 F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge Univ. Press, 1999.
3 F. Blanqui, W. Delobel, S. Coupet-Grimal, S. Hinderer, and A. Koprowski. CoLoR, a Coq library on rewriting and termination. In *WST*, pages 69–73, 2006.
4 É. Contejean, A. Paskevich, X. Urbain, P. Courtieu, O. Pons, and J. Forest. A3PAT, an approach for certified automated termination proofs. In *PEPM*, pages 63–72, 2010.
5 J. Endrullis. Jambox. Available at `http://joerg.endrullis.de`.
6 J. Endrullis, J. Waldmann, and H. Zantema. Matrix interpretations for proving termination of term rewriting. *J. Autom. Reasoning*, 40(2-3):195–220, 2008.
7 A. Geser, D. Hofbauer, J. Waldmann, and H. Zantema. On tree automata that certify termination of left-linear term rewriting systems. *Inf. Comput.*, 205(4):512–534, 2007.
8 J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Mechanizing and improving dependency pairs. *J. Autom. Reasoning*, 37(3):155–203, 2006.
9 J. Giesl, R. Thiemann, and P. Schneider-Kamp. Proving and disproving termination of higher-order functions. In *FroCoS*, LNAI 3717, pages 216–231, 2005.
10 J. Giesl, P. Schneider-Kamp, and R. Thiemann. AProVE 1.2: Automatic termination proofs in the dependency pair framework. In *IJCAR*, LNAI 4130, pages 281–286, 2006.
11 N. Hirokawa, A. Middeldorp, and H. Zankl. Uncurrying for termination. In *LPAR*, LNAI 5330, pages 667–681, 2008.
12 D. Hofbauer and J. Waldmann. Match-bounds for relative termination. In *WST*, 2010.
13 A. Koprowski. TPA: Termination proved automatically. In *RTA*, LNCS 4098, 2006.
14 M. Korp and A. Middeldorp. Match-bounds revisited. *Inf. Comput.*, 207(11):1259–1283, 2009.
15 T. Nipkow, L.C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic.* LNCS 2283. Springer, 2002.
16 C. Sternagel and A. Middeldorp. Root-labeling. In *RTA*, LNCS 5117, pages 336–350, 2008.
17 C. Sternagel and R. Thiemann. Signature extensions preserve termination. In *CSL*, LNCS 6247, pages 514–528, 2010.
18 R. Thiemann and C. Sternagel. Certification of termination proofs using CeTA. In *TPHOLs*, LNCS 5674, pages 452–468, 2009.
19 H. Zantema. Termination of term rewriting by semantic labelling. *Fundam. Inform.*, 24(1-2):89–105, 1995.
20 H. Zantema. Termination of string rewriting proved automatically. *J. Autom. Reasoning*, 34(2):105–139, 2005.

Type Preservation as a Confluence Problem*

Aaron Stump[1], Garrin Kimmell[1], and Roba El Haj Omar[1]

1 **Computer Science**
The University of Iowa
astump@acm.org, gkimmell@cs.uiowa.edu, roba-elhajomar@uiowa.edu

Abstract

This paper begins with recent work by Kuan, MacQueen, and Findler, which shows how standard type systems, such as the simply typed lambda calculus, can be viewed as abstract reduction systems operating on terms. The central idea is to think of the process of typing a term as the computation of an abstract value for that term. The standard metatheoretic property of type preservation can then be seen as a confluence problem involving the concrete and abstract operational semantics, viewed as abstract reduction systems (ARSs).

In this paper, we build on the work of Kuan et al. by showing show how modern ARS theory, in particular the theory of decreasing diagrams, can be used to establish type preservation via confluence. We illustrate this idea through several examples of solving such problems using decreasing diagrams. We also consider how automated tools for analysis of term-rewriting systems can be applied in testing type preservation.[1]

1998 ACM Subject Classification D.3.1 Formal Definitions and Theory

Keywords and phrases Term rewriting, Type Safety, Confluence

Digital Object Identifier 10.4230/LIPIcs.RTA.2011.345

Category Regular Research Paper

1 Introduction

The central idea of this paper is to view typing as an abstract operational semantics, and then study its interaction with concrete operational semantics using the theory of abstract reduction systems (ARSs). This idea was already proposed by Kuan, MacQueen, and Findler in 2007 [10]. They did not, however, use modern tools of term-rewriting theory in their study. Ellison et al. also explored a similar rewriting approach to type inference, but again without applying term-rewriting theory for the metatheory [7] (also [9]). In contrast, we seek to apply powerful tools from term-rewriting theory to prove standard metatheoretic properties of type systems.

Like Kuan et al., we view typing as abstract reduction, and consider types to be abstract forms of the values produced by reduction. For example, the concrete term, $\lambda x : \texttt{int}.x$ reduces to the abstract value $\texttt{int} \Rightarrow \texttt{int}$. Reduction is defined on *mixed terms*, which contain both (standard) concrete terms, and partially abstract ones. We show that the combined relation consisting of both abstract and concrete reduction is confluent, for typed terms. Kuan et al. also claim this result, but their proof sketch appeals to the standard

* This work was partially supported by the U.S. National Science Foundation, contract CCF-0910510, as part of the Trellys project.

[1] An extended version of this paper, including proofs of Theorems 2.1 and 5.1, is available at http://www.cs.uiowa.edu/~astump/papers/rta11.pdf

22nd International Conference on Rewriting Techniques and Applications.
Editor: M. Schmidt-Schauß; pp. 345–360

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

metatheoretic property of Type Preservation. In contrast, we prove confluence directly using decreasing diagrams, and show how this then implies the standard metatheoretic property of Type Preservation.

The contribution of this paper is to show how type preservation, cast as a confluence problem, can be solved using the tools of abstract reduction systems and term-rewriting. This provides an alternative proof method for establishing type preservation for programming languages. Having alternative methods is, of course, valuable in its own right, but we will see that the rewriting approach is qualitatively simpler than the traditional one, and arguably more intuitive. This may enable shorter and simpler proofs of type preservation for other systems, as we consider further in the Conclusion.

We begin (Section 2) by developing the approach in detail for a straightforward example, namely call-by-value Simply Typed Lambda Calculus (STLC). We consider next several variations on this, including extending STLC with a fixed-point combinator (Section 3), adding polymorphism (Section 4), and implementing type inference (Section 5). Kuan et al. also considered type inference, but did not make the connection which we identify, that type inference from the rewriting perspective corresponds to narrowing, rather than rewriting. These first systems we consider can all can be analyzed using a very simple labeling for the decreasing diagrams argument. We conclude with a trickier example, namely simply typed combinators with uniform syntax (i.e., no syntactic distinction between terms and types) in Section 6. We show how automated term-rewriting tools can be used to partially automate the proof of type preservation.

2 A Rewriting View of Simple Typing

This section demonstrates the proposed new approach, for the example of STLC. While our focus in this paper is Type Preservation, we also consider the standard Progress and Type Safety properties. For Progress, it is instructive to include reduction rules for some selected constants (a and f below), so that there are stuck terms that should be ruled out by the type system. Otherwise, in pure STLC, every closed normal form is a value, namely a λ-abstraction. We see how to view a type-computation (also called type-synthesis) system for STLC as an abstract operational semantics, and type preservation as a form of confluence for the combination of the abstract and the standard concrete operational semantics. We first recapitulate the standard approach to the definitions.

We use several standard notations in this paper. For an abstract reduction relation $\to$, we define $\to^=$ as its reflexive closure, $\to^+$ as its transitive closure, and $\to^*$ as its reflexive-transitive closure.

2.1 Syntax and Semantics

The syntax for terms, types, and contexts is the following, where f and a are specific constants, and x ranges over a countably infinite set of variables:

$$\begin{array}{lcl} \textit{types } T & ::= & A \mid T_1 \Rightarrow T_2 \\ \textit{standard terms } t & ::= & f \mid a \mid x \mid t_1\ t_2 \mid \lambda x : T.t \\ \textit{contexts } \Gamma & ::= & \cdot \mid \Gamma, x : T \end{array}$$

We assume standard additional conventions and notations, such as $[t/x]t'$ for the capture-avoiding substitution of t for x in t', and $E[t]$ for grafting a term into a context. Figure 1 defines a standard type system for STLC, which can be viewed as deterministically computing

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T} \qquad \frac{}{\Gamma \vdash f : A \Rightarrow A} \qquad \frac{}{\Gamma \vdash a : A}$$

$$\frac{\Gamma \vdash t_1 : T_2 \Rightarrow T_1 \quad \Gamma \vdash t_2 : T_2}{\Gamma \vdash t_1\ t_2 : T_1} \qquad \frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x : T_1 . t : T_1 \Rightarrow T_2}$$

Figure 1 Type-computation rules for simply typed lambda calculus with selected constants

$$\frac{}{E[(\lambda x : T . t)\ v] \ \rightarrow\ E[[v/x]t]} \qquad \begin{array}{rcl} \textit{values } v & ::= & \lambda x : T . t \mid a \mid f \\ \textit{evaluation contexts } E & ::= & * \mid (E\ t) \mid (v\ E) \end{array}$$

$$\frac{}{E[f\ a] \ \rightarrow\ E[a]}$$

Figure 2 A call-by-value small-step operational semantics

a type T as output, given a term t and a typing context Γ as inputs. A standard small-step call-by-value (CBV) operational semantics is defined using the rules of Figure 2.

As mentioned above, we are including constants so that the Progress theorem is not trivial. These constants are a and f, with the reduction rule $E_c[fa] \rightarrow_c E_c[a]$. Using these constants we can also construct stuck terms, such as ff, which we demonstrate are ill-typed in the proof of Progress. An example concrete reduction is (with redexes underlined):

$$\underline{(\lambda x : (A \rightarrow A) . x\ (x\ a))\ f}\ \rightarrow_c\ f\ \underline{(f\ a)}\ \rightarrow_c\ \underline{f\ a}\ \rightarrow_c\ a$$

2.2 Basic Metatheory

The main theorem relating the reduction relation $\rightarrow$ and typing is **Type Preservation**, which states:

$$(\Gamma \vdash t : T \ \wedge\ t \rightarrow t') \ \Rightarrow\ \Gamma \vdash t' : T$$

The standard proof method is to proceed by induction on the structure of the typing derivation, with case analysis on the reduction derivation (cf. Chapters 8 and 9 of [12]). A separate induction is required to prove a substitution lemma, needed critically for type preservation for β-reduction steps:

$$\Gamma \vdash t : T \ \wedge\ \Gamma, x : T \vdash t' : T' \ \Rightarrow\ \Gamma \vdash [t/x]t' : T'$$

One also typically proves **Progress**:

$$(\cdot \vdash t : T \ \wedge\ t \not\rightarrow) \ \Rightarrow\ \exists v.\ t = v$$

Here, the notation $t \not\rightarrow$ means $\forall t' .\ \neg(t \rightarrow t')$; i.e., t is a normal form. Normal forms which are not values are called *stuck* terms. An example is $f\ f$. Combining Type Preservation and Progress allows us to prove **Type Safety** [19]. This property states that the normal forms of closed well-typed terms are values, not stuck terms, and in our setting can be stated:

$$(\cdot \vdash t : T \ \wedge\ t \rightarrow^*\ t' \not\rightarrow) \ \Rightarrow\ \exists v.\ t' = v$$

This is proved by induction on the length of the reduction sequence from t to t'.

$$
\begin{array}{lll}
\textit{types } T & ::= & A \mid T_1 \Rightarrow T_2 \\
\textit{standard terms } t & ::= & x \mid \lambda x : T.\, t \mid t\ t' \mid a \mid f \\
\textit{mixed terms } m & ::= & x \mid \lambda x : T.\, m \mid m\ m' \mid a \mid f \mid \\
 & & A \mid T \Rightarrow m \\
\textit{standard values } v & ::= & \lambda x : T.t \mid a \mid f \\
\textit{mixed values } u & ::= & \lambda x : T.m \mid T \Rightarrow m \mid A \mid a \mid f
\end{array}
$$

Figure 3 Syntax for STLC using mixed terms

$$
\frac{}{E_c[f\ a] \ \to_c \ E_c[a]}\ c(f\text{-}\beta) \qquad \frac{}{E_c[(\lambda x : T.\, m)\ u] \ \to_c \ E_c[[u/x]m]}\ c(\beta)
$$

$$
\frac{}{E_a[(T \Rightarrow m)\ T] \ \to_a \ E_a[m]}\ a(\beta) \qquad \frac{}{E_a[\lambda x : T.\, m] \ \to_a \ E_a[T \Rightarrow [T/x]m]}\ a(\lambda)
$$

$$
\frac{}{E_a[f] \ \to_a \ E_a[A \Rightarrow A]}\ a(f) \qquad \frac{}{E_a[a] \ \to_a \ E_a[A]}\ a(a)
$$

$$
\begin{array}{lll}
\textit{mixed evaluation contexts } E_c & ::= & * \mid (E_c\ t) \mid (u\ E_c) \\
\textit{abstract evaluation contexts } E_a & ::= & * \mid (E_a\ m) \mid (m\ E_a) \mid \lambda x : T.\, E_a \mid T \Rightarrow E_a
\end{array}
$$

Figure 4 Abstract and concrete operational semantics for STLC

2.3 Typing as Abstract Operational Semantics

To view typing as an abstract form of reduction, we use mixed terms, defined in Figure 3. Types like $T_1 \Rightarrow T_2$ will serve as abstractions of λ-abstractions. Figure 4 gives rules for concrete ($\to_c$) and abstract ($\to_a$) reduction. We denote the union of these reduction relations as $\to_{ca}$. The definition of abstract evaluation contexts makes abstract reduction nondeterministic, as reduction is allowed anywhere inside a term. This is different from the approach followed by Kuan et al., where abstract and concrete reduction are both deterministic. Here is an example reduction using the abstract operational semantics:

$$
\begin{array}{l}
\lambda x : (A \Rightarrow A).\, \lambda y : A.\, (x\ (x\ y)) \ \to_a \\
\lambda x : (A \Rightarrow A).\, A \ \Rightarrow \ (x\ (x\ A)) \ \to_a \\
(A \Rightarrow A) \ \Rightarrow \ A \ \Rightarrow ((A \Rightarrow A)\ ((A \Rightarrow A)\ A)) \ \to_a \\
(A \Rightarrow A) \ \Rightarrow \ A \ \Rightarrow ((A \Rightarrow A)\ A) \ \to_a \\
(A \Rightarrow A) \ \Rightarrow \ A \ \Rightarrow A
\end{array}
$$

The final result is a type T. Indeed, using the standard typing rules of Section 2.1, we can prove that the starting term of this reduction has that type T, in the empty context. Abstract reduction to a type plays the role of typing above.

If we look back at our standard typing rules (Figure 1), we can now see them as essentially big-step abstract operational rules. Recall that big-step CBV operational semantics for STLC is defined by:

$$
\frac{t_1 \ \Downarrow \ \lambda x : T.t_1' \quad t_2 \ \Downarrow \ t_2' \quad [t_2'/x]t_1' \ \Downarrow \ t'}{t_1\ t_2 \ \Downarrow \ t'}
$$

In our setting, this would be concrete big-step reduction, which we might denote $\Downarrow_c$. The abstract version of this rule, where we abstract λ-abstractions by arrow-types, is

$$\frac{t_1 \ \Downarrow_a \ T \Rightarrow T' \quad t_2 \ \Downarrow_a \ T}{t_1 \ t_2 \ \Downarrow_a \ T'}$$

If we drop the typing context from the typing rule for applications (from Figure 1), we obtain essentially the same rule.

The standard approach to proving type preservation relates a small-step concrete operational semantics with a big-step abstract operational semantics (i.e., the standard typing relation). We find it both more elegant, and arguably more informative to relate abstract and concrete small-step relations, as we will do in the next section.

▶ **Theorem 2.1** (Relation with Standard Typing). *For standard terms t, we have $x_1 : T_1, \cdots, x_n : T_n \vdash t : T$ iff $[T_1/x_1, \cdots, T_n/x_n]t \rightarrow_a^* T$.*

See the companion technical report for the proof [14].

▶ **Theorem 2.2** (Termination of Abstract Reduction). *The relation $\rightarrow_a$ is terminating.*

Proof. Whenever $m \rightarrow_a m'$, the following measure is strictly decreased from m to m': the number of occurrences of term constructs (listed in the definition of *terms*) which are not also type constructs (listed in the definition of *types*) and which occur in the term. Term constructs of STLC which are not also type constructs are constants, variables, λ-abstractions, and applications. **End proof.**

2.4 Type Preservation as Confluence

The relation $\rightarrow_{ca}$ is not confluent in general, as Kuan et al. note also in their setting. It is, however, confluent if restricted to *typable* terms, which are mixed terms m such that $m \rightarrow_a^* T$ for some type T. We will make use here of the standard notion of confluence of an element with respect to a binary relation $\rightarrow$: m is confluent (with respect to $\rightarrow$) iff for all m_1 and m_2 such that $m_2 \leftarrow^* m \rightarrow^* m_1$, there exists $\hat{m}$ such that $m_1 \rightarrow^* \hat{m} \leftarrow^* m_2$. In this section, we prove the following result:

▶ **Theorem 2.3** (Confluence of combined reduction). *Every typable mixed term is confluent with respect to the reduction relation $\rightarrow_{ca}$.*

We obtain the following as an obvious corollary, noting that types T are in normal form (so joinability of m' and T becomes just reducibility of m' to T):

▶ **Theorem 2.4** (Type Preservation). *If $m \rightarrow_a^* T$ and $m \rightarrow_c^* m'$, then $m' \rightarrow_a^* T$.*

We can phrase this result already in terms of multistep concrete reduction, while as described above, the standard approach to type preservation is stated first for single-step reduction, and then extended inductively to multistep reduction. Theorem 2.4 can also be phrased in terms of standard typing, using Theorem 2.1.

We prove Theorem 2.3 by a simple application of *decreasing diagrams* [17]. Recall that the main result of the theory of decreasing diagrams is that if every local peak can be completed to a *locally decreasing diagram* with respect to a fixed well-founded partial ordering on labeled steps in the diagram, then the ARS is confluent. A locally decreasing diagram has peak $s_1 \leftarrow_\alpha t \rightarrow_\beta s_2$ and valley of the form

$$s_1 \rightarrow_{\Upsilon\alpha}^* \rightarrow_\beta^= \rightarrow_{(\Upsilon\alpha)\cup(\Upsilon\beta)}^* \hat{t} \leftarrow_{(\Upsilon\alpha)\cup(\Upsilon\beta)}^* \leftarrow_\alpha^= \leftarrow_{\Upsilon\beta}^* s_2$$

where $\Upsilon\alpha$ denotes the set of labels strictly smaller in the fixed well-founded ordering than α, and if A is a set of labels, then $\rightarrow_A$ denotes the relation $\bigcup_{\alpha\in A} \rightarrow_\alpha$.

For Theorem 2.3, we label every step $m \rightarrow_c m'$ with c, and every step $m \rightarrow_a m'$ with a, using the label ordering $a < c$. We must prove that every local peak starting with a typable term can be completed to a locally decreasing diagram. Since $\rightarrow_c$ is deterministic (due to the restrictions inherent in the definition of reduction contexts E_c), we must only consider local peaks of the form $m_1 \leftarrow_a m \rightarrow_a m_2$ (we will call these aa-peaks) and $m_1 \leftarrow_a m \rightarrow_c m_2$ (ac-peaks). For the first, we have the following theorem:

▶ **Theorem 2.5.** *The relation $\rightarrow_a$ is confluent.*

Proof. In fact, we can prove that $\rightarrow_1$ has the diamond property (i.e., $(\leftarrow_a \cdot \rightarrow_a) \subseteq (\rightarrow_a \cdot \leftarrow_a)$, which is well-known to imply confluence). Suppose $m \rightarrow_a m_1$ and $m \rightarrow_a m_2$. No critical overlap is possible between these steps, because none of the redexes in the a-rules of Figure 4 (such as $(T \Rightarrow m)\ T$ in the $a(\beta)$ rule) can critically overlap another such redex. If the positions of the redexes in the terms are parallel, then (as usual) we can join m_1 and m_2 by applying to each the reduction required to obtain the other. Finally, we must consider the case of non-critical overlap (where the position of one redex in m is a prefix of the other position). We can also join m_1 and m_2 in this case by applying the reduction to m_i which was used in $m \rightarrow_a m_{3-i}$, because abstract reduction cannot duplicate or delete an a-redex. The only duplication of any subterm in the abstract reduction rules of Figure 4 is of the type T in $a(\lambda)$. The only deletion possible is of the type T in $a(\beta)$. Since types cannot contain redexes, there is no duplication or deletion of redexes. This means that if the position of the first redex is a prefix of the second (say), then there is exactly one descendant (see Section 4.2 of [15]) of the second redex in m_1, and this can be reduced in one step to join m_1 with the reduct of m_2 obtained by reducing the first redex. So every aa-peak can be completed with one joining step on each side of the diagram (and local decreasingness obviously holds). This gives the diamond property and thus confluence for $\rightarrow_a$. ◀

We return now to the rest of the proof of Theorem 2.3, and consider the ac-peaks. There are only two possibilities. First, we could have the a-redex at a position parallel to the position of the c-redex. In this case, the diagram can be appropriately completed by commuting the steps. Second we could have the a-redex inside a subterm of the c-redex. There are four simple situations, and one somewhat more complex situation. The first two simple situations are that the a-redex is inside m or inside mixed value u, respectively, where the redex is $(\lambda x : T.\, m)\ u$. In the first case, the peak is

$$E_c[(\lambda x : T.\, m')\ u] \ \leftarrow_a \ E_c[(\lambda x : T.\, m)\ u] \ \rightarrow_c \ E_c[[u/x]m]$$

The required valley is just:

$$E_c[(\lambda x : T.\, m')\ u] \ \rightarrow_c \ E_c[[u/x]m'] \ \leftarrow_a \ E_c[[u/x]m]$$

The right joining step is justified because abstract reduction is closed under substitution. The labels on the single joining sides are as required for local decreasingness. In the second simple situation, the peak is:

$$E_c[(\lambda x : T.\, m)\ u'] \ \leftarrow_a \ E_c[(\lambda x : T.\, m)\ u] \ \rightarrow_c \ E_c[[u/x]m]$$

The required valley is:

$$E_c[(\lambda x : T.\, m)\ u'] \ \rightarrow_c \ E_c[[u'/x]m] \ \leftarrow_a^* \ E_c[[u/x]m]$$

Here, the labels on the right joining path are all less than the label on the right edge of the peak, satisfying the requirement for local decreasingness. Note that the left joining step would not be possible if we had phrased concrete reduction using just standard terms: we need to apply the $c(\beta)$ rule with mixed value u, which might not be a standard value. Also, we require the following easily proved lemma (proof omitted), to conclude that contracting the a-redex in u indeed results in a new value u':

▶ **Lemma 2.6.** *If $u \to_a^* m$ then m is also a mixed value.*

The second two simple situations involve $f\ a$. First:

$$E_a[(A \Rightarrow A)\ a] \leftarrow_a E_a[f\ a] \to_c E_a[a]$$

The joining valley, which is again locally decreasing because its labels are less than c, is:

$$E_a[(A \Rightarrow A)\ a] \to_a E_a[(A \Rightarrow A)\ A] \to_a E_a[A] \leftarrow_a E_a[a]$$

Second:

$$E_a[f\ A] \leftarrow_a E_a[f\ a] \to_c E_a[a]$$

The joining valley is:

$$E_a[f\ A] \to_a E_a[(A \Rightarrow A)\ A] \to_a E_a[A] \leftarrow_a E_a[a]$$

Finally, the more complicated case is shown in Figure 5. Since the term at the peak is typable, we know that $u \to_a^* T_1$. This is because abstract reduction cannot eliminate an application from a term, except via $a(\beta)$: no abstract reduction rule can erase a term. So we know that $u \to_a^* T_1$, or else the application displayed at the peak could not be eliminated by abstract reduction.

Considering the diagram in Figure 5, we see that again, the labels in the joining steps are all less than the label on the right edge of the peak. So the condition for local decreasingness is satisfied. Note that the bottom-right joining step, from $E_c[[u/x]m]$ to $E_c[[T_1/x]m]$ requires nondeterminism of $\to_a$ (and if $x \notin \mathit{Vars}(m)$, is just an identity step). It would not hold if abstract reduction were restricted to the abstract analog of call-by-value concrete reduction, as done by Kuan et al. We have proved the following, which suffices by the theory of decreasing diagrams to prove **Confluence** (Theorem 2.3):

▶ **Lemma 2.7** (Local Decrease for Type Diagrams)**.** *Every local peak of $\to_{ac}$ can be completed to a locally decreasing diagram.*

We can complete the basic metatheory for STLC by proving:

▶ **Theorem 2.8** (Progress)**.** *If standard term t is closed, $t \to_a^* T$, and $t \not\to_c$, then t is a concrete standard value.*

Proof. The only possibility for a closed standard term that is a normal form of $\to_c$ other than a standard value is a stuck term $E_c[d\ t]$, where t is a c-normal form and either $d \equiv a$, or else $d \equiv f$ and $t \not\equiv a$. Let $d\ t$ be the smallest such stuck term in $E_c[d\ t]$. The term $E_c[d\ t]$ has an a-normal form (by Theorem 2.2), which must contain a descendant of $d\ t$. This is because, as already noted, our abstract reduction rules drop an expression only if it is a type (rule $a(\beta)$). Also, abstract reduction cannot contract a descendant of $f\ t$ itself, which we argue by cases on the form of $f\ t$ (call this term $\hat{t}$). If $\hat{t} \equiv a\ t$, then for some m, $\hat{t}$ a-normalizes to

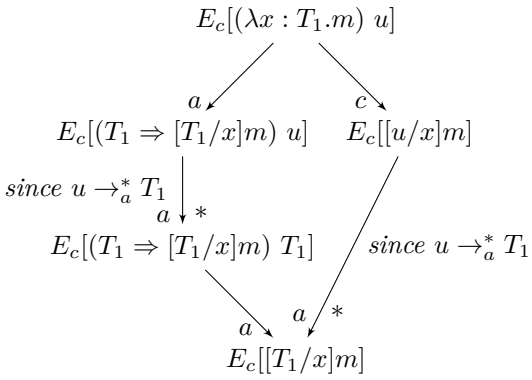


Figure 5 Crucial locally decreasing diagram for STLC

$A\ m$, which is still a descendant of $\hat{t}$. If $\hat{t} \equiv f\ f$, then $\hat{t}$ a-normalizes to $(A \Rightarrow A)\ (A \Rightarrow A)$ (also still a descendant). If $\hat{t} \equiv f\ \lambda x : T.t$, then $\hat{t}$ a-normalizes to $(A \Rightarrow A)\ (A \Rightarrow m)$ for some m. This shows that a descendant of $d\ t$ must still be contained in the a-normal form of $t \equiv E_c[d\ t]$, contradicting $t \to_a^* T$. ◀

We then obtain the final result as a direct corollary of Theorems 2.4 and 2.8.

▶ **Theorem 2.9** (Type Safety). *If standard term t is closed, $t \to_a^* T$, and $t \to_c^* t' \not\to_c$, then t' is a concrete value.*

The development in this section compares favorably with the standard one, which requires an induction proof for type preservation, another for the additionally required substitution lemma, and a final one for Type Safety. In contrast, here all that was required was to analyze ac-peaks for local decreasingness (the aa-peaks being easily joinable in one step) in order to establish Type Preservation. That analysis is both more local and more informative, as it gives a more direct insight into how the (small-step) process of abstracting a term to its type relates to the small-step operational semantics.

3 STLC with a Fixed-Point Combinator

We may easily extend the results of the previous section to include a fixed-point combinator *fix*. This is a standard example, which enables now possibly diverging concrete reductions. The approach using decreasing diagrams easily adapts to this new situation. Figure 6 shows the additions to the syntax of STLC. The proof of **Termination of Abstract Reduction** (Theorem 2.2) goes through exactly as stated above, since *fix* is a term construct but not a type construct, and our new abstract reduction for *fix*-terms again reduces the number of occurrences of term constructs which are not type constructs. For **Local Decrease for Type Diagrams** (Lemma 2.7), there is still no critical overlap between a-steps, and no possibility of erasing or duplicating a redex, so $\to_a$ still has the diamond property. The simple ac-peaks are easily completed, similarly to the simple ones for STLC. We must then just consider this new ac-peak:

$$E_c[(T \Rightarrow T)\ [T/f]m) \leftarrow_a \underline{E_c[\mathit{fix}\ f : T.\, m]} \to_a E_c[[\mathit{fix}\ f : T.\, m/f]m]$$

$$
\begin{array}{lll}
\textit{(standard) terms } t & ::= & \ldots \mid \mathit{fix}\ f : T.\, t \\
\textit{(mixed) terms } m & ::= & \ldots \mid \mathit{fix}\ f : T.\, m \\
\textit{abstract evaluation contexts } E_a & ::= & \ldots \mid \mathit{fix}\ f : T.\, E_a
\end{array}
$$

$$
\frac{}{E_c[\mathit{fix}\ f : T.\, m] \ \to_c \ E_c[[\mathit{fix}\ f : T.\, m/f]m]} \qquad \frac{}{E_a[\mathit{fix}\ f : T.\, m] \ \to_a \ E_a[(T \Rightarrow T)\ [T/f]m]}
$$

Figure 6 Extending STLC with a fixed-point combinator

$$
\begin{array}{lll}
\textit{types } T & ::= & \ldots \mid \alpha \mid \forall \alpha.T \\
\textit{standard terms } t & ::= & \ldots \mid \Lambda\alpha.t \mid t@T \\
\textit{mixed terms } m\ t & ::= & \ldots \mid \Lambda\alpha.m \mid m@T \mid \forall \alpha.m \\
\textit{standard values } v & ::= & \ldots \mid \Lambda\alpha.t \\
\textit{mixed values } u & ::= & \ldots \mid \Lambda\alpha.m \mid \forall \alpha.m
\end{array}
$$

Figure 7 Polymorphism syntactic extensions

This can be completed as follows:

$$
\begin{array}{ll}
L. & E_c[(T \Rightarrow T)\ [T/f]m] \to_a^* E_c[(T \Rightarrow T)\ T] \to_a E_c[T] \\
R. & E_c[[\mathit{fix}\ f : T.\, m/f]m] \to_a^* E_c[[((T \Rightarrow T)\ [T/f]m)/f]m] \to_a^* E_c[[T/f]m] \to_a^* E_c[T]
\end{array}
$$

Here, several steps use $[T/f]m \to_a^* T$, which holds because the term at the peak is typable. This diagram is again locally decreasing. We conclude Confluence (Theorem 2.3) as above. There are no possible additional stuck terms, so **Progress** (Theorem 2.8) is trivially extended, allowing us to conclude **Type Safety** (Theorem 2.9).

4 Adding Polymorphism

In this section, we extend STLC with System-F style polymorphism. Figures 7 and 8 show the additional syntax, evaluation contexts, and reduction rules. At the term level, we add syntax for type abstraction (Λ), type application ($t@t$), and type variables α. At the type level, we add universally quantified types $\forall\alpha.\ T$. As is standard practice, concrete reduction now includes a β-like rule for eliminating type applications of type abstractions. The abstract reduction relation has a similar rule for type applications of universal types, and also a rule for computing a universal type from a type abstraction.

The argument for **Termination of Abstraction Reduction** (Theorem 2.2) is extended easily, since the two new $\to_a$ rules, $a(\Lambda)$ and $a(\beta_T)$, again strictly decrease the number of occurrences of terms which are not types. We can extend the proof of **Local Decrease for Type Diagrams** (Lemma 2.7) by considering new $\to_{ac}$ peaks. As before, these can be separated into cc, aa, and ac cases. As with the STLC case, the $\to_c$ relation is deterministic, so there are no cc peaks. There are no critical overlaps between $\to_a$ steps, and again no possibility to erase or duplicate a redex. So we maintain the diamond property for aa-peaks. A c-redex of the form $(\Lambda x.\, t)@T$ gives rise to a possible ac-peak, where a-reduction occurs under the Λ:

$$
E_c[(\Lambda\alpha.\, m')@T] \ \leftarrow_a \ E_c[(\Lambda\alpha.\, m)@T] \to_c \ E_c[[T/\alpha]m]
$$

$$\frac{}{E_c[(\Lambda\alpha.m)@T] \;\to_c\; E_c[[T/\alpha]m]}\; c(\beta_T) \qquad \frac{}{E_a[(\forall\alpha.m)@T] \;\to_a\; E_a[[T/\alpha]m]}\; a(\beta_T)$$

$$\frac{}{E_a[\Lambda\alpha.t] \;\to_a\; E_a[\forall\alpha.t]}\; a(\Lambda)$$

$$\begin{array}{lll} \textit{mixed evaluation contexts } E_c & ::= & \ldots \mid (E_c \;@\; T) \mid (u \;@\; E_c) \\ \textit{abstract evaluation contexts } E_a & ::= & \ldots \mid (E_a\ m) \mid ((\forall\alpha.m)@\ E_a) \end{array}$$

Figure 8 Polymorphism operational semantics extensions

As with the STLC, a-reduction is closed under substitution, so this peak can be completed as shown:

$$E_c[(\Lambda\alpha.\, m')@T] \;\to_c\; E_c[[T/\alpha]m'] \leftarrow_a\; E_c[[T/\alpha]m]$$

The grammar for the polymorphic language requires that all type-level applications contain a proper type T in the argument position, so there does not exist a ac-peak analagous to the STLC ac-peak where an a-step occurs in the argument subterm of a c-redex. There is only one critical overlap, due to the $a(\Lambda)$ and $c(\beta_T)$ rules,

$$E_c[(\forall\alpha.m)@T] \leftarrow_a\; E_c[(\Lambda\alpha.m)@T] \to_c E_c[[T/\alpha]m]$$

This can be completed with a single c-reduction:

$$E_c[(\forall\alpha.m)@T] \to_c E_c[[T/\alpha]m] \leftarrow_c^{=} E_c[[T/\alpha]m]$$

This completes the proof for **Local Decrease of Type Diagrams**.

Finally, the proof of **Progress** for STLC is extended by considering a few new stuck terms. These are terms of the form $a@T$ and $f@T$, which reduce to non-type a-normal forms $A@T$ and $(A \Rightarrow A)@T$, respectively, contradicting Progress's assumption of typability. Also, we could have $a\ \Lambda\alpha.m$ or $f\ \Lambda\alpha.m$, but these reduce respectively to non-type a-normal forms $A\ \forall\alpha.m'$ and $(A\ \Rightarrow A)\ \forall\alpha.m'$, for some m'. We again conclude **Type Safety**.

5 Type Inference for STLC

We can modify STLC reduction rules to allow us to infer, rather than check, the type of STLC terms where type annotations are omitted for λ-bound variables. Inferring simple types is a central operation in ML-style type inference. The central idea is to base abstract reduction on narrowing, rather than just rewriting. Our focus in this paper is the use of confluence to prove type safety, and not necessarily the efficient implementations of the resulting reduction system. Nevertheless, we believe that abstract reduction systems can serve as the basis for efficient implementation, as demonstrated by Kuan [11] for similar reduction systems.

Figure 9 shows the grammar of STLC-inf. In this language, λ-bound variables do not have type annotations, as they will be inferred. The syntax of types now includes type variables α_i, which may be instantiated by narrowing.

Using narrowing, we can define the abstract reduction system shown in Figure 9 to infer the types of terms. In the $a(\beta)$ rule, we calculate the most general unifier of the function

$$
\begin{array}{lcl}
\textit{types } T & ::= & T \Rightarrow T \mid \alpha_i \mid A \\
\textit{standard terms } t & ::= & x \mid t\ t' \mid \lambda x.t \mid a \mid f \\
\textit{mixed terms } m & ::= & x \mid m\ m' \mid \lambda x.m \mid a \mid f \mid T \Rightarrow m \mid A \mid \alpha_i \\
\textit{standard values } v & ::= & a \mid f \mid \lambda x.t \\
\textit{mixed values } u & ::= & a \mid f \mid \lambda x.m \mid T \Rightarrow u \mid A \\
\textit{mixed evaluation contexts } E_c & ::= & * \mid (E_c\ m) \mid (u\ E_c) \\
\textit{abstract evaluation contexts } E_a & ::= & * \mid (E_a\ m) \mid (m\ E_a) \mid T \Rightarrow E_a \mid \alpha_i\ E_a
\end{array}
$$

$$\frac{}{E_c[(\lambda x.m)u] \ \rightarrow_c \ E_c[[u/x]m]}\ c(\beta) \qquad \frac{\alpha_i \notin FV(E_a[\lambda x.m])}{E_a[\lambda x.m] \ \rightarrow_a \ E_a[\alpha_i \Rightarrow [\alpha_i/x]m]}\ a(\lambda)$$

$$\frac{\sigma \textit{ is } mgu(T_1, T_2)}{E_a[(T_1 \Rightarrow m)T_2] \ \rightarrow_a \ \sigma(E_a[m])}\ a(\beta) \qquad \frac{}{E_a[f] \rightarrow_a E_a[A \Rightarrow A]}\ a(f)$$

$$\frac{\alpha_j \notin FV(E_a[\alpha_i\ T])}{E_a[\alpha_i\ T] \rightarrow_a [(T \Rightarrow \alpha_j)/\alpha_i](E_a[\alpha_j])}\ a(gen) \qquad \frac{}{E_a[a] \rightarrow_a E_a[A]}\ a(a)$$

$$\frac{\rho \textit{ is a permutation of type variables}}{t \rightarrow_a \rho\ t}\ a(rename)$$

Figure 9 Type Inference Syntax and Semantics

type's domain and the argument, and apply it to the entire term, including the evaluation context. This contrasts with the rules we have seen in previous systems presented in this paper, where substitutions are applied only to the focus of the evaluation context. We assume for all rules that introduce new type variables that they do so following a fixed order. We include the rule $a(rename)$ to avoid non-confluence due to different choices of new type variables in the rules $a(gen)$ and $a(\lambda)$.

The following theorem relates STLC-inf typing to STLC typing. The *erasure* of an STLC term, $|t|$, drops type annotations from λ bindings, producing an STLC-inf term.

▶ **Theorem 5.1** (Relation with STLC Typing). *If $t \rightarrow_a^* T$, where T contains free type variables $\alpha_1, ..., \alpha_n$, then for all types $T_1, ...T_n$, there exists a term t' in STLC such that $t' \rightarrow_a^* [T_1/\alpha_1, ...T_n/\alpha_n]T$ in STLC and $|t'| = t$.*

The complete proof can be found in the Appendix of the expanded technical report.

We extend the metatheoretic results to STLC-inf as follows. We no longer have **Termination of Abstract Reduction**, due to $a(rename)$. This does not impact subsequent results, as aa-peaks are still joinable in one step. The ac-peak depicted in Figure 5 is adapted to account for the inference of types for λ-bound variables, as shown in Figure 10. The completion of the a reduction relies on the assumption that the generated type variable α is not free in the original term.

Because the $a(\beta)$ rule uses narrowing to compute λ-bound variable types, it is possible to generate aa-peaks when a polymorphic function is used at multiple monotypes. Consider the following reduction resulting in a mixed term:

$$(\lambda g.\lambda x.\lambda y.\lambda h.h\ (g\ x)\ (g\ y))\ (\lambda w.w)\ f\ a \quad \rightarrow_a^+ \quad \alpha_h(\alpha_g(A \Rightarrow A))(\alpha_g A)$$

We can reduce this latter term to the following distinct stuck terms, depending on which

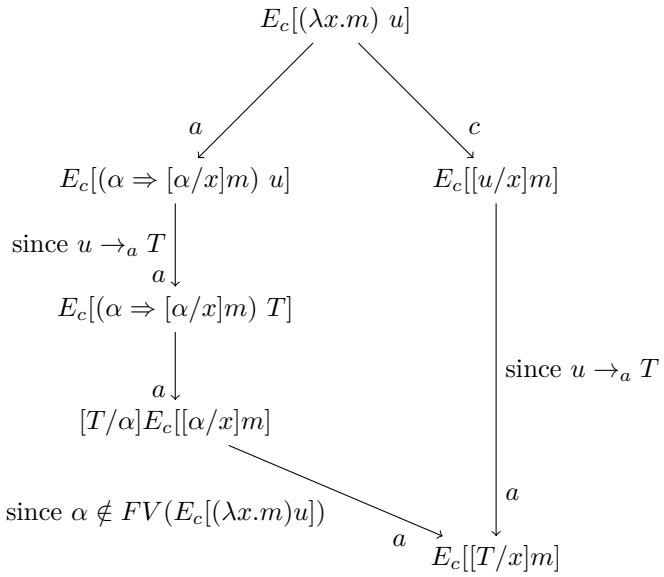


Figure 10 Crucial locally decreasing diagram for STLC-inf

$$
\begin{array}{lll}
\textit{mixed terms } t & ::= & S\langle t_1, t_2, t_3\rangle \mid K\langle t_1, t_2\rangle \mid t_1\ t_2 \mid t_1 \Rightarrow t_2 \mid A \mid \textit{kind}(t_1,\ t_2) \\
\textit{mixed values } u & ::= & S\langle t_1, t_2, t_3\rangle \mid K\langle t_1, t_2\rangle \mid A \mid t_1 \Rightarrow t_2 \\
\textit{concrete evaluation contexts } E_c & ::= & * \mid E_c\ t \mid u\ E_c
\end{array}
$$

Figure 11 Uniform-STLC language syntax and evaluation contexts

application of α_g we contract with the *a(gen)* rule first:

$$\alpha_h\ ((A \Rightarrow \alpha_j)\ (A \Rightarrow A))\ \alpha_j \qquad \alpha_h\ \alpha_j\ (((A \Rightarrow A)\ \Rightarrow \alpha_j)\ A)$$

But we are proving **Confluence** only for typeable terms $t \to_a^* T$. So such peaks can be disregarded. We conclude **Confluence** and then **Type Preservation** as above. The proof of **Progress** is adapted directly from STLC, as the changes in STLC-inf do not essentially affect the form of stuck terms. So we can conclude **Type Safety**.

6 Simply Typed Combinators with Uniform Syntax

In this section, we consider a language, which we call Uniform-STC, that does not distinguish terms and types syntactically. Advanced type systems like Pure Type Systems must often rely solely on the typing rules to distinguish terms and types (and kinds, superkinds, etc.) [4]. In Uniform-STC, we explore issues that arise in applying the rewriting approach to more advanced type systems. We must now implement kinding (i.e., type checking of types) as part the abstract reduction relation. We adopt a combinatory formulation so that the abstract reduction relation can be described by a first-order term-rewriting system.

Figure 11 shows the syntax for the Uniform-STC language. There is a single syntactic category t for mixed terms and types, which include a base type A and simple function types. $S\langle t_1, t_2, t_3\rangle$ and $K\langle t_1, t_2\rangle$ are the usual combinators, indexed by terms which determine their

$$c(\beta\text{-}S).\quad \overline{E_c[S\langle t_1,t_2,t_3\rangle\ u\ u'\ u'']\to_c E_c[u\ u''\ (u'\ u'')]}$$

$$c(\beta\text{-}K).\quad \overline{E_c[K\langle t_1,t_2\rangle\ u\ u']\to_c E_c[u]}$$

$$\begin{array}{ll} a(S). & S\langle t_1,t_2,t_3\rangle \to_a \mathit{kind}(t_1,\mathit{kind}(t_2,\mathit{kind}(t_3,(t_1{\Rightarrow}t_2{\Rightarrow}t_3){\Rightarrow}(t_1{\Rightarrow}t_2){\Rightarrow}(t_1{\Rightarrow}t_3))))\\ a(K). & K\langle t_1,t_2\rangle \to_a \mathit{kind}(t_1,\mathit{kind}(t_2,(t_1\Rightarrow t_2\Rightarrow t_1)))\\ a(\beta). & (t_1\Rightarrow t_2)\ t_1\to_a \mathit{kind}(t_1,t_2)\\ a(k\text{-}\Rightarrow). & \mathit{kind}((t_1\Rightarrow t_2),t)\to_a \mathit{kind}(t_1,\mathit{kind}(t_2,t))\\ a(k\text{-}A). & \mathit{kind}(A,t)\to_a t \end{array}$$

Figure 12 Concrete and abstract reduction rules

simple types. The *kind* construct for terms is used to implement kinding. The rules for concrete and abstract reduction are given in Figure 12. The concrete rules are just the standard ones for call-by-value reduction of combinator terms. For abstraction reduction, we are using first-order term-rewriting rules (unlike for previous systems).

For STLC (Section 2), abstract β-redexes have the form $(T \Rightarrow t)\ T$. For Uniform-STC, since there is no syntactic distinction between terms and types, abstract β-redexes take the form $(t_1 \Rightarrow t_2)\ t_1$, and we must use kinding to ensure that t_1 is a type. This is why the $a(\beta)$ rule introduces a *kind*-term. We also enforce kinding when abstracting simply typed combinators $S\langle t_1,t_2,t_3\rangle$ and $K\langle t_1,t_2\rangle$ to their types. The rules for *kind*-terms ($a(k\text{-}\Rightarrow)$ and $a(k\text{-}A)$) make sure that the first term is a type, and then reduce to the second term.

Following our general procedure, we wish to show that every local peak at a typable term can be completed to a locally decreasing diagram. Here, we define typability by value u to mean abstract reduction to u where u is *kindable*, which we define as $\mathit{kind}(u,A)\to_a^* A$. This definition avoids the need to define types syntactically.

Abstract reduction for Uniform-STC does not have the diamond property, non-left-linear rule $a(\beta)$, where there could indeed be redexes in the expressions matching the repeated variable t_1. Fortunately, abstract reduction can be automatically analyzed for termination: AProve reports that it can be shown terminating using a recursive path ordering [8]. This means that we can use van Oostrom's source-labeling heuristic for a-steps [18]. We label steps as follows:

$$\mathit{label}(m\to_c m') = c \qquad \mathit{label}(m\to_a m') = (a,m)$$

Steps are then ordered by the relation $\succ$ defined by:

$$\forall m.c\succ(m,a) \qquad \forall m,m'.m\to_a^+ m' \Leftrightarrow (m,a)\succ(m',a)$$

The abstract reduction rules are non-overlapping. The aa-peaks which occur can all be joined using either one a-step on either side as for STLC, or else using additional balancing steps if one of the rules applied is $a(\beta)$. In the latter case, the diagram is still decreasing due to the termination of abstract reduction.

But we can actually use even a simpler argument for aa-peaks. The automated confluence prover ACP reports that the abstract reduction relation is confluent [2]. So by completeness of decreasing diagrams for countable relations (recalled in [18] from van Oostrom's PhD thesis), there must exist a labeling of abstract reduction steps that allows all aa-peaks to

be completed to locally decreasing diagrams. We can then extend this labeling to include c (labeling c-steps), with c bigger in the extended ordering than all the labels of a-steps.

With this ordering (or the source-labeling), we then have the following locally decreasing diagram (the one for $c(\beta\text{-}S)$ is similar and omitted), where $\hat{t}$ is $(t_1 \Rightarrow t_2 \Rightarrow t_1)$ and u is $kind(t_1, kind(t_2, *))$:

$P.$ $E_c[u[(\hat{t}\ t\ t')]] \leftarrow_a E_c[(K\langle t_1, t_2\rangle\ t\ t')] \rightarrow_a E_c[t]$
$L.$ $E_c[u[(\hat{t}\ t\ t')]] \rightarrow_a^* E_c[u[(\hat{t}\ t_1\ t'')]] \rightarrow_a E_c[u[((t_2 \Rightarrow t_1)\ t'')]] \rightarrow_a^*$
$E_c[u[((t_2 \Rightarrow t_1)\ t_2)]] \rightarrow_a E_c[kind(t_1, kind(t_2, t_1))] \rightarrow_a^* E_c[t_1]$
$R.$ $E_c[t] \rightarrow_a^* E_c[t_1]$

The $\rightarrow_a^*$-steps are justified because the peak term (shown on line (P)) is assumed to be typable. By confluence of abstract reduction, this implies that the sources of all the left steps are also typable. For each $\rightarrow_a^*$-step, since abstract reduction cannot drop redexes (as all rules are non-erasing), we argue as for STLC that a descendant of the appropriate displayed *kind*-term or application must eventually be contracted, as otherwise, a stuck descendant of such would remain in the final term. Kindable terms cannot contain stuck applications or stuck *kind*-terms, because our abstract reduction rules are non-erasing. And contraction of those displayed *kind*-terms or applications requires the reductions used for the $\rightarrow_a^*$-steps, which are sufficient to complete the diagram. The diagram is again locally decreasing because the c-step from the peak is greater than all the other steps in the diagram. We thus have **Confluence** of ac-reduction for typable terms, and the following statement of type preservation (relying on our definition above of typability):

▶ **Theorem 6.1** (Type Preservation). *If t has type t_1 and $t \rightarrow_c t'$, then t' also has type t_1.*

As an aside, note that a natural modification of this problem is out of the range of ACP, version 0.20. Suppose we are trying to group kind-checking terms so that we can avoid duplicate kind checks for the same term. For this, we may wish to permute *kind*-terms, and pull them out of other term constructs. The following rules implement this idea, and can be neither proved confluent nor disproved by ACP, version 0.20. Just the first seven rules are also unsolvable by ACP.

```
(VAR a b c A B C D)
(RULES
  S(A,B,C) -> kind(A,kind(B,kind(C,
              arrow(arrow(arrow(A,arrow(B,C)),arrow(A,B)),arrow(A,C)))))
  K(A,B) -> kind(A,kind(B,arrow(A,arrow(B,A))))
  app(arrow(A,b),A) -> kind(A,b)
  kind(base,a) -> a
  kind(arrow(A,B),a) -> kind(A, kind(B, a))
  kind(A,kind(A,a)) -> kind(A,a)
  kind(A,kind(B,a)) -> kind(B,kind(A,a))
  app(kind(A,b),c) -> kind(A,app(b,c))
  app(c,kind(A,b)) -> kind(A,app(c,b))
  arrow(kind(A,b),c) -> kind(A,arrow(b,c))
  arrow(c,kind(A,b)) -> kind(A,arrow(c,b))
  kind(kind(a,b),c) -> kind(a,kind(b,c))
)
```

7 Conclusion

We have seen how to use decreasing diagrams to establish confluence of a reduction relation combining concrete reduction (the standard operational semantics) with abstract reduction,

which can be thought of as a small-step typing relation. Type Preservation is then an immediate corollary of confluence. We have applied this to several example type systems. We highlight that we are able to cast type inference as a form of narrowing, and that for first-order systems, we can apply termination and confluence checkers to automate part of the proof of type preservation.

For future work, the approach should be applied to more advanced type systems. Dependent type systems pose a particular challenge, because from the point of view of abstract reduction, Π-bound variables must play a dual role. When computing a dependent function type $\Pi x : T.\ T'$ from an abstraction $\lambda x : T.t$, we may need to abstract x to T, as for STLC; but we may also need to leave it unabstracted, since with dependent types, x is allowed to appear in the range type T'. We conjecture that this can be accommodated by substituting a pair (x, T) for x in the body of the λ-abstraction, and then choosing either the term or type part of the pair depending on how the pair is used.

It would be interesting to try to use the rewriting method to automate type preservation proofs completely. While the Programming Languages community has invested substantial effort in recent years on computer-checked proofs of properties like Type Safety for programming languages (initiated particularly by the POPLmark Challenge [3]), there is relatively little work on fully automatic proofs of type preservation (an example is [13]). The rewriting approach could contribute to filling that gap.

Our longer term goal is to use this approach to design and analyze type systems for symbolic simulation. In program verification tools like Pex and KeY, symbolic simulation is a central component [5, 16]. But these systems do not seek to prove that their symbolic-simulation algorithms are correct. Indeed, the authors of the KeY system argue against expending the effort to do this [6]. The rewriting approach promises to make it easier to relate symbolic simulation, viewed as an abstract reduction relation, with the small-step operational semantics.

Acknowledgments. We thank the anonymous RTA 2011 reviewers for their helpful comments, which have improved this paper.

References

1 S. Abramsky, D. Gabbay, and T. Maibaum, editors. *Handbook of Logic in Computer Science*. Oxford University Press, 1992.

2 T. Aoto, J. Yoshida, and Y. Toyama. Proving Confluence of Term Rewriting Systems Automatically. In R. Treinen, editor, *Rewriting Techniques and Applications (RTA)*, pages 93–102, 2009.

3 B. Aydemir, A. Bohannon, M. Fairbairn, J. Foster, B. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: The POPLmark Challenge. In *Proceedings of the Eighteenth International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2005)*, 2005.

4 H. Barendregt. *Lambda Calculi with Types*, pages 117–309. Volume 2 of Abramsky et al. [1], 1992.

5 B. Beckert, R. Hähnle, and P. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer-Verlag, 2007.

6 B. Beckert and V. Klebanov. Must Program Verification Systems and Calculi be Verified? In *Proceedings, 3rd International Verification Workshop (VERIFY), Workshop at Federated Logic Conferences (FLoC), Seattle, USA*, pages 34–41, 2006.

7 C. Ellison, T. Şerbănuţă, and G. Roşu. A Rewriting Logic Approach to Type Inference. In A. Corradini and U. Montanari, editors, *Recent Trends in Algebraic Development Techniques (WADT)*, pages 135–151, 2008.

8 J. Giesl, P. Schneider-Kamp, and R. Thiemann. Automatic Termination Proofs in the Dependency Pair Framework. In U. Furbach and N. Shankar, editors, *Automated Reasoning, Third International Joint Conference (IJCAR)*, pages 281–286, 2006.

9 M. Hills and G. Rosu. A Rewriting Logic Semantics Approach to Modular Program Analysis. In C. Lynch, editor, *Proceedings of the 21st International Conference on Rewriting Techniques and Applications, RTA 2010, July 11-13, 2010, Edinburgh, Scotland, UK*, pages 151–160, 2010.

10 G. Kuan, D. MacQueen, and R. Findler. A rewriting semantics for type inference. In *Proceedings of the 16th European conference on Programming (ESOP)*, pages 426–440. Springer-Verlag, 2007.

11 George Kuan. Type checking and inference via reductions. In Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt, editors, *Semantics Engineering with PLT Redex*. MIT Press, 2009.

12 B. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

13 C. Schürmann and F. Pfenning. Automated Theorem Proving in a Simple Meta-Logic for LF. In C. Kirchner and H. Kirchner, editors, *15th International Conference on Automated Deduction (CADE)*, pages 286–300, 1998.

14 A. Stump, G. Kimmell, and R. El Haj Omar. Type Preservation as a Confluence Problem. Companion report, available from first author's home page.

15 TeReSe, editor. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.

16 N. Tillmann and W. Schulte. Parameterized Unit Tests. *SIGSOFT Softw. Eng. Notes*, 30:253–262, 2005.

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

18 V. van Oostrom. Confluence by Decreasing Diagrams, Converted. In A. Voronkov, editor, *Rewriting Techniques and Applications*, pages 306–320, 2008.

19 A. Wright and M. Felleisen. A Syntactic Approach to Type Soundness. *Information and Computation*, 115(1):38–94, 1994.

Left-linear Bounded TRSs are Inverse Recognizability Preserving

Irène Durand and Marc Sylvestre

LaBRI, Université Bordeaux 1
351 Cours de la libération, F-33405 Talence cedex, France

Abstract

Bounded rewriting for linear term rewriting systems has been defined in (I. Durand, G. Sénizergues, M. Sylvestre. Termination of linear bounded term rewriting systems. Proceedings of the 21st International Conference on Rewriting Techniques and Applications) as a restriction of the usual notion of rewriting. We extend here this notion to the whole class of left-linear term rewriting systems, and we show that bounded rewriting is effectively inverse-recognizability preserving. The *bounded class* (BO) is, by definition, the set of left-linear systems for which every derivation can be replaced by a bottom-up derivation. The class BO contains (strictly) several classes of systems which were already known to be inverse-recognizability preserving: the left-linear growing systems, and the inverse right-linear finite-path overlapping systems.

1998 ACM Subject Classification Primary: F.4.2, Secondary: F.3.2, F.4.1

Keywords and phrases Term Rewriting, Preservation of Recognizability, Rewriting Strategies

Digital Object Identifier 10.4230/LIPIcs.RTA.2011.361

Category Regular Research Paper

1 Introduction

General framework. A term rewriting system (TRS) $\mathcal{R}$ is effectively recognizability preserving (respectively inverse recognizability preserving) if for every recognizable set of terms T the set $[T](\rightarrow_{\mathcal{R}}^*) = \{s \mid \exists t \in T, t \rightarrow_{\mathcal{R}}^* s\}$ (resp. $(\rightarrow_{\mathcal{R}}^*)[T] = \{s \mid \exists t \in T, s \rightarrow_{\mathcal{R}}^* t\}$) is recognizable and can be built. Many efforts have been made for finding subclasses of TRSs which are (inverse) recognizability preserving. The identification of such subclasses is important and has applications in equational reasoning, formal computation, automated deduction, and verification. For example, the reachability problem which is central in these areas, particularly in verification, is decidable for these classes of TRSs. The techniques used to prove reachability are often based on the computation of $[E](\rightarrow_{\mathcal{R}}^*)$ for some set E, and are coming from the Knuth and Bendix completion algorithm (see [14] for the seminal paper). An entire workshop is devoted to the reachability problem: the Workshop on Reachability Problem (RP). Each result of recognizability preservation yields also almost directly a new decidable call-by-need class [4] and decidability results on confluence (see [1] or [7] for a survey) and joinability. This notion has also been used to prove termination of systems for which none of the already known termination techniques work [10]. Different techniques for proving termination have been implemented in several softwares (Matchbox [24], AProVE [11], TORPA [25], CiME [3]). Consequently, the seek of a class which preserves the recognizability is well motivated. Many such classes have been defined by imposing syntactical restrictions on the rewrite rules (e.g. growing TRSs [16, 12] and finite-path overlapping TRSs [20, 21]). Another way is to use a *strategy*, i.e. restrictions on the derivations rather than on the rules, to ensure preservation of recognizability. Various such strategies where studied in [8, 17, 19, 5, 6]. In this paper,

22nd International Conference on Rewriting Techniques and Applications (RTA'11).
Editor: M. Schmidt-Schauß; pp. 361–376

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


we extend the bounded rewriting for linear TRSs to left-linear TRSs that may have non right-linear rules and we prove that this strategy is inverse recognizability preserving.

From linear TRS*s to left-linear* TRS*s.* Bounded rewriting for linear TRSs is essentially a new version of bottom-up rewriting [5] that is easier to define and has better properties. The reader may refer to [6] for more details on bounded rewriting for linear TRSs. Intuitively, for a linear TRS $\mathcal{R}$, a derivation is k-bounded ($\mathsf{lbo}(k)$) if when a rule is applied, the parts of the substitution located at a depth greater than k are not rewritten further in the derivation, i.e. do not match the left-handside of a rule applied further. A linear TRS $\mathcal{R}$ is $\mathsf{lbo}(k)$ if for any derivation $s \to_{\mathcal{R}}^* t$ there exists a $\mathsf{lbo}(k)$ derivation $s \ {}_k\!\!\to_{\mathcal{R}}^* t$. The class of linear $\mathsf{lbo}(k)$ TRSs is denoted by $\mathsf{LBO}(k)$. One of the goals of this paper is to drop the right-linear restriction and propose an extension of bounded rewriting to left-linear TRSs. This extension cannot be the trivial one: even if nothing in the definition of $\mathsf{LBO}(k)$ TRSs requires the linear condition, keeping this definition unchanged would define a class containing only linear TRSs (see example 4.14).

To solve this problem, we introduce a binary symbol E and a set $\mathcal{E}$ of three rewrite rules to handle this symbol: the introduction rule $x \to E(x, x)$, and two selection rules $E(x, y) \to x$ and $E(x, y) \to y$. Intuitively, E allows to store several descendants of the same initial subterm. Let $\mathcal{R}$ be a left-linear TRS over a signature $\mathcal{F}$. Roughly speaking, a derivation in $\mathcal{R} \cup \mathcal{E}$ is k-bounded if when a rule is applied, the parts of the substitution located at a depth greater than k (without taking the E into consideration) are not rewritten further in the derivation, i.e. do not match the left-handside of a rule of $\mathcal{R}$ applied further. A derivation in $s \to_{\mathcal{R}}^* t$ is k-bounded convertible ($\mathsf{boc}(k)$) if there exists a $\mathsf{bo}(k)$-derivation from s to t in $\mathcal{R} \cup \mathcal{E}$. Note that this definition does not constrain the application of the rules of $\mathcal{E}$. A TRS is $\mathsf{bo}(k)$ if every derivation is $\mathsf{boc}(k)$.

The class of $\mathsf{bo}(k)$ TRSs is denoted by $\mathsf{BO}(k)$. Let us see how we use the symbol E. Suppose that $\mathsf{f}(\mathsf{a}) \to_{\mathsf{f}(x)\to\mathsf{g}(x,x)} \mathsf{g}(\mathsf{a},\mathsf{a}) \to_{\mathsf{a}\to\mathsf{b}} \mathsf{g}(\mathsf{a},\mathsf{b})$. The symbol E is used to apply the rule $\mathsf{a} \to \mathsf{b}$ before the rule $\mathsf{f}(x) \to \mathsf{g}(x, x)$. First, we use E to create an envelop which contains a and b: $\mathsf{f}(\mathsf{a}) \to_{x\to E(x,x)} \mathsf{f}(E(\mathsf{a},\mathsf{a})) \to_{\mathsf{a}\to\mathsf{b}} \mathsf{f}(E(\mathsf{a},\mathsf{b}))$. Then we can apply the rule $\mathsf{f}(x) \to \mathsf{g}(x, x)$, and use the selections rules to obtain $\mathsf{g}(\mathsf{a},\mathsf{b})$: $\mathsf{f}(E(\mathsf{a},\mathsf{b})) \to \mathsf{g}(E(\mathsf{a},\mathsf{b}), E(\mathsf{a},\mathsf{b})) \to_{E(x,y)\to x} \mathsf{g}(\mathsf{a}, E(\mathsf{a},\mathsf{b})) \to_{E(x,y)\to y} \mathsf{g}(\mathsf{a},\mathsf{b})$. The introduction of the symbol E can be viewed as a counterpart of the construction of the powerset automaton in the extension of Jacquemard's saturation method [12] by Nagaya and Toyama [16] (this saturation method is used to prove that left-linear growing TRSs are inverse recognizability preserving).

Inverse recognizability preservation. In section 5, we prove that bounded rewriting for left-linear TRSs is effectively inverse recognizability preserving. This result is obtained by simulating $\mathsf{bo}(k)$-derivations by a ground tree transducer. The idea of simulating $\mathsf{bo}(k)$-derivations is similar to the idea developed in [5] where bottom-up(k) derivations are simulated using a ground TRS. This simulation yields directly to the inverse preservation result since GTTs are effectively inverse recognizability preserving.

Strongly bounded systems. In section 6, we introduce a subclass of $\mathsf{BO}(k)$ called the strongly bounded class ($\mathsf{SBO}(k)$). The membership problem for $\mathsf{SBO}(k)$ is decidable whereas the membership problem for $\mathsf{BO}(0)$ is undecidable. The class of strongly bounded TRSs contains inverse right-linear finite-path overlapping TRSs [22] and left-linear growing TRSs [16]. Note that a long version of this paper is available at: `http://hal.archives-ouvertes.fr/hal-00580528/fr/`.

2 Preliminaries

Given a set E, we denote by $\mathcal{P}(E)$ its powerset i.e. the set of all its subsets. Its cardinality is denoted by $\mathsf{Card}(E)$. 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.

Assume that the set A is ordered. We denote by $\preceq_{\mathsf{Lex}_A}$ the lexicographic order on the set of words A^*. We may omit Lex_A when it is clear from the context. We assume the reader familiar with terms and automata (see e.g. [2] or [23] for an introduction). We call *signature* a set $\mathcal{F}$ of symbols with arity $ar : \mathcal{F} \to \mathbb{N}$. The subset of symbols with arity $m \in \mathbb{N}$ is denoted by $\mathcal{F}_m$.

As usual, a finite 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 arity assignment. We denote by $\mathcal{T}(\mathcal{F}, \mathcal{V})$ the set of first-order terms over the signature $\mathcal{F} \cup \mathcal{V}$, where $\mathcal{F}$ is a 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 $\mathcal{V}\mathsf{ar}(t)$. A variable x is said to occur at depth n in t if there exists a position $u \in \mathcal{P}\mathsf{os}(t)$ such that $t(u) = x$ and $|u| = n$. The root symbol of t is denoted by $\mathsf{root}(t)$. Given a set of symbols and variables $A \subseteq \mathcal{F} \cup \mathcal{V}$ and a term t, the set of positions $u \in \mathcal{P}\mathsf{os}(t)$ such that $t(u) \in A$ is denoted by $\mathcal{P}\mathsf{os}_A(t)$ and the set of position $u \in \mathcal{P}\mathsf{os}(t)$ such that $t(u) \notin A$ is denoted by $\mathcal{P}\mathsf{os}_{\setminus A}(t)$. Let X be either A or $\setminus A$ and $u \in \mathcal{P}\mathsf{os}_X(t)$. We denote by $\mathcal{P}\mathsf{os}_X{}^{\preceq u}(t)$ (respectively $\mathcal{P}\mathsf{os}_X{}^{\prec u}(t)$) the set of positions $v \in \mathcal{P}\mathsf{os}_X(t)$ such that $v \preceq u$ (resp. $v \prec u$) and by $\mathcal{P}\mathsf{os}_X{}^{\succeq u}(t)$ (respectively $\mathcal{P}\mathsf{os}_X{}^{\succ u}(t)$) the set of positions $v \in \mathcal{P}\mathsf{os}_X(t)$ such that $v \succeq u$ (resp. $v \succ u$). When $A = \{f\}$ for some $f \in \mathcal{F} \cup \mathcal{V}$ we may denote $\mathcal{P}\mathsf{os}_f(t)$ (respectively $\mathcal{P}\mathsf{os}_{\setminus f}(t)$) instead of $\mathcal{P}\mathsf{os}_{\{f\}}(t)$ (resp. $\mathcal{P}\mathsf{os}_{\setminus\{f\}}(t)$). A *substitution* σ is a mapping from $\mathcal{V}$ into $\mathcal{T}(\mathcal{F}, \mathcal{V})$. The substitution σ is naturally extended to a morphism $\sigma : \mathcal{T}(\mathcal{F}, \mathcal{V}) \to \mathcal{T}(\mathcal{F}, \mathcal{V})$, where $\sigma(f(t_1, .., t_n)) = f(\sigma(t_1), \ldots, \sigma(t_n))$, for each $f \in \mathcal{F}_n$, $t_i \in \mathcal{T}(\mathcal{F}, \mathcal{V})$. Substitutions will often be used in postfix notation: $t\sigma$ is the result of applying σ to the term t. The *depth* of a term t is defined by $dpt(t) := \sup\{\mathsf{Card}(\mathcal{P}\mathsf{os}^{\preceq u}(t)) \mid u \in \mathcal{P}\mathsf{os}_{\setminus\mathcal{V}}(t)\}$. This definition is extended to substitutions $dpt(\sigma) := \max\{dpt(x\sigma) \mid x \in \mathcal{V}\}$. For a term t and a symbol $f \in \mathcal{F}$, we define $dpt_{\setminus f}(t)$ by: $dpt_{\setminus f}(t) := \sup\{\mathsf{Card}(\mathcal{P}\mathsf{os}_{\setminus f}{}^{\preceq u}(t)) \mid u \in \mathcal{P}\mathsf{os}_{\setminus\{f\}\cup\mathcal{V}}(t)\}$. This definition is extended to substitutions $dpt_{\setminus f}(\sigma) := \max\{dpt_{\setminus f}(x\sigma) \mid x \in \mathcal{V}\}$. The set of *leaves* of t is the set $\mathcal{P}\mathsf{os}_{\mathcal{V}\cup\mathcal{F}_0}(t)$ and is also denoted by $\mathcal{L}\mathsf{v}(t)$. For a variable $x \in \mathcal{V}\mathsf{ar}(t)$, the set of positions $\mathcal{P}\mathsf{os}_x(t)$ is also denoted by $\mathcal{P}\mathsf{os}(t, x)$. Let $w \in \mathcal{L}\mathsf{v}(t)$. The *branch* containing w is the set of positions u such that $u \preceq w$.

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 $\mathcal{P}\mathsf{os}(t/u) = \{w \mid uw \in \mathcal{P}\mathsf{os}(t)\}$ and $\forall w \in \mathcal{P}\mathsf{os}(t/u)$, $t/u(w) = t(uw)$.

A term that 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 denote by $\mathsf{pos}(t, x)$ the position of x in t.

A term containing no variable is called *ground*. The set of ground terms is abbreviated to $\mathcal{T}(\mathcal{F})$ or $\mathcal{T}$ whenever $\mathcal{F}$ is understood.

We denote by $C[t_1, \ldots, t_n]_{u_1,\ldots,u_n}$ the term obtained from $C[]_{u_1,\ldots,u_n}$ by replacing, for every $i \in \{1, \ldots, n\}$, the symbol $\Box$ at position u_i by the term t_i. Let t be a term, and $\{u_1, \ldots, u_n\} \subset \mathcal{P}\mathsf{os}(t)$ be a set of incomparable positions given in lexicographic order. We

denote by $t[]_{u_1,\ldots,u_m}$ the context obtained from t by replacing each subterm t/u_i at a position u_i by a leaf labeled by $\Box$.

A *rewrite rule* over 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* if its left-handside is linear.

A *term rewriting system* (TRS for short) is a pair $(\mathcal{F}, \mathcal{R})$ where $\mathcal{F}$ is a signature and $\mathcal{R}$ a finite set of rewrite rules over 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 $\mathcal{R}$.

We denote by $\mathsf{LHS}(\mathcal{R})$ the set of lhs of $\mathcal{R}$, and by $\mathsf{RHS}(\mathcal{R})$ the set of rhs of $\mathcal{R}$.

Rewriting is defined as usual: for every $s, t \in \mathcal{T}(\mathcal{F})$, $s \to_{\mathcal{R}} t$ means that there exist a position $v \in \mathcal{P}\mathsf{os}(s)$, a rule $l \to r \in \mathcal{R}$, and a substitution σ such that $s = s[l\sigma]_v$ and $t = s[r\sigma]_v$.

We denote by $\to_{\mathcal{R}}^+$ the transitive closure of $\to$, by $\to_{\mathcal{R}}^{0,1}$ its reflexive closure, and by $\to_{\mathcal{R}}^*$ its reflexive and transitive closure. We may omit $\mathcal{R}$ when it is clear from the context. We say that there exists a derivation from s to t in $\mathcal{R}$ when $s \to_{\mathcal{R}}^* t$. The *length* of a derivation is the number of steps in this derivation. An n-step derivation from s to t is denoted by $s \to^n t$. More generally, the notation defined in [13] will be used in proofs.

A TRS is *linear* (resp. *left-linear*) if each of its rules is linear (resp. left-linear). A TRS $\mathcal{R}$ is *growing* [12] if every variable of a right-handside occurs at depth at most 1 in the corresponding left-handside.

We shall consider finite bottom-up term (tree) automata [2] (which we abbreviate to *f.t.a.*). An automaton $\mathcal{A}$ is given by a 4-tuple $(\mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Gamma)$ where $\mathcal{F}$ is a signature, $\mathcal{Q}$ is a finite set of symbols of arity 0, called the set of states and such that $\mathcal{Q} \cap \mathcal{F}_0 = \emptyset$, $\mathcal{Q}_f \subseteq \mathcal{Q}$ is the set of final states, Γ is the set of transitions. A transition has either the form $q \to r$ for some $q, r \in \mathcal{Q}$, or $f(q_1, \ldots, q_m) \to q$ for some $m \geq 0, f \in \mathcal{F}_m, q_1, \ldots, q_m \in \mathcal{Q}$. Note that we can have rules of the form $c \to q$ with $c \in \mathcal{F}_0$, and $q \in \mathcal{Q}$. We shall also consider automaton on a denumerable signature. Such an automaton is given by a 4-tuple $(\mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Gamma)$ where $\mathcal{F}$ is a denumerable signature, $\mathcal{Q}$ is a finite set of symbols of arity 0, $\mathcal{Q}_f \subseteq \mathcal{Q}$ is the set of final states, and Γ is an denumerable set of transitions.

The set of rules Γ can be viewed as a TRS over the signature $\mathcal{F} \cup \mathcal{Q}$. We then denote by $\to_{\mathcal{A}}$ the one-step rewriting relation generated by Γ. Given an automaton $\mathcal{A}$, the set of terms accepted by $\mathcal{A}$ is defined by: $\mathcal{L}(\mathcal{A}) := \{t \in \mathcal{T}(\mathcal{F}) \mid \exists q \in \mathcal{Q}_f, t \to_{\mathcal{A}}^* q\}$. A set of terms T is *recognizable* if there exists a term automaton $\mathcal{A}$ such that $T = \mathcal{L}(\mathcal{A})$. The automaton $\mathcal{A}$ is called *deterministic* if there is no rule of the form $q \to r$ for some $q, r \in \mathcal{Q}$ and if for every $t \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q}), q, q' \in \mathcal{Q}$, $(t \to q \in \Gamma \;\&\; t \to q' \in \Gamma) \Rightarrow (q = q')$. The automaton $\mathcal{A}$ is called *complete* if for every $m \geq 0$, $f \in \mathcal{F}_m$ and m-tuple of states $(q_1, \ldots, q_m) \in \mathcal{Q}^m$, there exists $q \in \mathcal{Q}$ such that $f(q_1, \ldots, q_m) \to q \in \Gamma$.

Ground tree transducers have been introduced in [15]. A *ground tree transducer* (GTT) is a pair $V := (\mathcal{A}_1, \mathcal{A}_2)$ of f.t.a. automata over a signature $\mathcal{F}$. Let $\mathcal{A}_1 = (\mathcal{F}, \mathcal{Q}_1, \emptyset, \Gamma_1)$, $\mathcal{A}_2 = (\mathcal{F}, \mathcal{Q}_2, \emptyset, \Gamma_2)$. The relation recognized by V is the set $\mathcal{L}(V) := \{(t, t') \mid t, t' \in \mathcal{T}(\mathcal{F}), \exists s \in \mathcal{T}(\mathcal{F} \cup (\mathcal{Q}_1 \cap \mathcal{Q}_2)),\ t \to_{\mathcal{A}_1}^* s, t' \to_{\mathcal{A}_2}^* s\}$. A set $T \subseteq \mathcal{T}(\mathcal{F}) \times \mathcal{T}(\mathcal{F})$ is said to be *recognizable* by a GTT if there exists a GTT V such that $T = \mathcal{L}(V)$. The reflexive and transitive closure of the relation $\mathcal{L}(V)$ is recognizable by a GTT (see e.g. chapter 3.2 of [2]).

A *ground recognizable* TRS (GRS) $(\mathcal{F}, \mathcal{G})$ is a (possibly infinite) TRS of the form $\mathcal{G} = \{l \to r \mid i \in I, l \in R_i, r \in K_i\}$, where I is a finite set, R_i and K_i for all $i \in I$ are recognizable sets of terms over $\mathcal{F}$. One can easily check that the relation $\to_{\mathcal{G}}^*$ is recognizable by a GTT.

Given a TRS $\mathcal{R}$ and a set of terms T, we define $(\to_{\mathcal{R}}^*)[T] := \{s \in \mathcal{T}(\mathcal{F}) \mid \exists t \in T, s \to_{\mathcal{R}}^* t\}$

and $[T](\rightarrow_{\mathcal{R}}^*) := \{s \in \mathcal{T}(\mathcal{F}) \mid \exists t \in T, t \rightarrow_{\mathcal{R}}^* s\}$. A TRS $\mathcal{R}$ is *effectively recognizability preserving* if for every recognizable T, $[T](\rightarrow_{\mathcal{R}}^*)$ is recognizable and can be built. A TRS $\mathcal{R}$ is *effectively inverse recognizability preserving* if for every recognizable T, $(\rightarrow_{\mathcal{R}}^*)[T]$ is recognizable and can be built.

We shall illustrate many of our definitions with the following left-linear TRS $(\mathcal{F}_1, \mathcal{R}_1)$ and the following complete deterministic automaton $\mathcal{A}_1$.

▶ **Example 2.1.** $\mathcal{F}_1 = \{\mathsf{a}, \mathsf{b}, \mathsf{f}(), \mathsf{h}(), \mathsf{g}(,), \mathsf{i}(,)\}$ is a signature, $\{x, y\}$ is a set of variables, $\mathcal{R}_1 = \{\mathsf{a} \rightarrow \mathsf{b}, \mathsf{f}(x) \rightarrow \mathsf{g}(x, x), \mathsf{h}(\mathsf{b}) \rightarrow \mathsf{b}, \mathsf{g}(\mathsf{h}(x), y) \rightarrow \mathsf{i}(x, y)\}$ is a set of rules, $\mathcal{A}_1 = (\mathcal{F}, \mathcal{Q}_{\mathcal{A}_1}, \{\mathsf{q}_f\}, \Gamma_{\mathcal{A}_1})$ with $\mathcal{Q}_{\mathcal{A}_1} = \{\mathsf{q}_f, \mathsf{q}_\mathsf{a}, \mathsf{q}_\mathsf{b}\, \mathsf{q}_\perp\}$, $\Gamma_{\mathcal{A}_1} = \{\mathsf{a} \rightarrow \mathsf{q}_\mathsf{a}, \mathsf{b} \rightarrow \mathsf{q}_\mathsf{b}, \mathsf{h}(\mathsf{q}_\mathsf{a}) \rightarrow \mathsf{q}_\mathsf{a}, \mathsf{h}(\mathsf{q}_\mathsf{b}) \rightarrow \mathsf{q}_\mathsf{b}, \mathsf{h}(\mathsf{q}_\perp) \rightarrow \mathsf{q}_\perp, \mathsf{i}(\mathsf{q}_\mathsf{a}, \mathsf{q}_\mathsf{b}) \rightarrow \mathsf{q}_f\} \cup \{\mathsf{f}(\mathsf{q}) \rightarrow \mathsf{q}_\perp \mid \mathsf{q} \in \mathcal{Q}_{\mathcal{A}_1}\} \cup \{\mathsf{g}(\mathsf{q}, \mathsf{q}') \rightarrow \mathsf{q}_\perp \mid \mathsf{q}, \mathsf{q}' \in \mathcal{Q}_{\mathcal{A}_1}\} \cup \{\mathsf{i}(\mathsf{q}, \mathsf{q}') \rightarrow \mathsf{q}_\perp \mid \mathsf{q}, \mathsf{q}' \in \mathcal{Q}_{\mathcal{A}_1}, (\mathsf{q}, \mathsf{q}') \neq (\mathsf{q}_\mathsf{a}, \mathsf{q}_\mathsf{b})\}$ is a complete deterministic automaton. We have:
$\mathcal{L}(\mathcal{A}_1) = \{\mathsf{i}(t_1, t_2) \mid t_1 \in \{\mathsf{a}, \mathsf{h}(\mathsf{a}), \ldots, \mathsf{h}(\mathsf{h}(\ldots(\mathsf{a}))), \ldots\}, t_2 \in \{\mathsf{b}, \mathsf{h}(\mathsf{b}), \ldots, \mathsf{h}(\mathsf{h}(\ldots(\mathsf{b}))), \ldots\}\}$
and
$(\rightarrow_{\mathcal{R}_1}^*)(\mathcal{L}(\mathcal{A}_1)) = \{\mathsf{i}(t_1, t_2), \mathsf{g}(\mathsf{h}(t_1), t_2), \mathsf{f}(\mathsf{h}(t_1)) \mid t_1 \in \{\mathsf{a}, \mathsf{h}(\mathsf{a}), \ldots, \mathsf{h}(\mathsf{h}(\ldots(\mathsf{a}))), \ldots\},$
$t_2 \in \{\mathsf{a}, \mathsf{h}(\mathsf{a}), \ldots, \mathsf{h}(\mathsf{h}(\ldots(\mathsf{a}))), \ldots\} \cup \{\mathsf{b}, \mathsf{h}(\mathsf{b}), \ldots, \mathsf{h}(\mathsf{h}(\ldots(\mathsf{b}))), \ldots\}\}$.

3 The TRS $\mathcal{E}$

From now on, until the end of this paper, we denote by $\mathcal{F}$ a finite signature, and by $\mathcal{R}$ a left-linear TRS over $\mathcal{F}$. Let $E \notin \mathcal{F}$ be a fresh symbol of arity 2.

▶ **Definition 3.1.** Let $x, y \in \mathcal{V}$. The left-linear TRS $\mathcal{E}$ is the TRS over $\mathcal{F} \cup \{E\}$ with the rules

$$x \rightarrow E(x, x) \qquad (1) \qquad\qquad E(x, y) \rightarrow x \text{ and } E(x, y) \rightarrow y \qquad (2)$$

Rule (1) is the *introduction rule*. Rules (2) are the *selection rules*. Note that the TRS $\mathcal{R} \cup \mathcal{E}$ is left-linear. This TRS will be used to define bounded rewriting (section 4) and has the following property.

▶ **Proposition 3.2.** *Let $s, t \in \mathcal{T}(\mathcal{F})$. We have $s \rightarrow_{\mathcal{R}}^* t$ iff $s \rightarrow_{\mathcal{R} \cup \mathcal{E}}^* t$.*

4 Bounded Rewriting

Roughly speaking, a derivation in $\mathcal{R} \cup \mathcal{E}$ is k-bounded ($\mathsf{bo}(k)$) if when a rule is applied, the parts of the substitution located at a depth greater than k (without taking the E into consideration) do not match a left-handside of a rule of $\mathcal{R}$ applied further. To indicate which positions are allowed to be rewritten further, we are going to apply a marking process. A mark is an integer. A marked term $\overline{t}$ is just a term t where all the symbols are marked. To every derivation $s_0 \rightarrow_{\mathcal{R} \cup \mathcal{E}} \ldots \rightarrow_{\mathcal{R} \cup \mathcal{E}} s_n$, we associate a marked derivation $\overline{s_0} \circ\!\!\rightarrow_{\mathcal{R} \cup \mathcal{E}} \ldots \circ\!\!\rightarrow_{\mathcal{R} \cup \mathcal{E}} \overline{s_n}$ (i.e. a derivation where all terms are marked terms). This derivation starts on the term $\overline{s_0}$ which is obtained from s_0 by setting all the marks to 0. Now, if we consider a marked term $\overline{s_j}$ in this derivation, a mark i on a symbol f in s_j indicates that the maximal depth (again, without taking the E into consideration) at which the symbol appears in a substitution during the derivation $s_0 \rightarrow_{\mathcal{R} \cup \mathcal{E}}^* s_j$ is i. The derivation will be said $\mathsf{bo}(k)$ if the maximal mark that appears on a lhs in the marked derivation $\overline{s_0} \circ\!\!\rightarrow_{\mathcal{R} \cup \mathcal{E}} \ldots \circ\!\!\rightarrow_{\mathcal{R} \cup \mathcal{E}} \overline{s_n}$ is $\leq k$. Formal definitions are given in the next sections.

4.1 Marked Terms

We define the signature of marked symbols: $\mathcal{F}^{\mathbb{N}} := \{f^i \mid i \in \mathbb{N}, f \in \mathcal{F}\}$. The operation $\mathsf{m}()$ returns the mark of a marked symbol: for $f \in \mathcal{F}, i \in \mathbb{N}, \mathsf{m}(f^i) = i$. We extend this operation to the symbol E: $\mathsf{m}(E) = 0$, and to variables: $\forall x \in \mathcal{V}, \mathsf{m}(x) = 0$. We define $\mathcal{F}^{\leq k}$ by $\mathcal{F}^{\leq k} := \{f^i \mid i \in \{0, \ldots, k\}, f \in \mathcal{F}\}$ and by $\mathcal{F}^{\geq k}$ the signature $\mathcal{F}^{\geq k} := \{f^i \mid i \geq k, f \in \mathcal{F}\}$. Marked terms are elements of $\mathcal{T}_M(\mathcal{V}) := \mathcal{T}(\mathcal{F}^{\mathbb{N}} \cup \{E\}, \mathcal{V})$. The set of ground marked terms is denoted by $\mathcal{T}_M$. The operation m extends to marked terms: if $t \in \mathcal{V}$, $\mathsf{m}(t) = 0$, otherwise, $\mathsf{m}(t) = \mathsf{m}(\mathsf{root}(t))$. We use $\mathsf{mmax}(t)$ to denote the maximal mark on t. We denote by t^i the term obtained by setting all the marks in t at a position $u \in \mathcal{P}\mathsf{os}_{\mathcal{F}}(t)$ to i. We extend this notation to sets of terms ($S^i := \{s^i | s \in S\}$), and to substitutions ($\sigma^i : x \to (x\sigma)^i$). For every $f \in \mathcal{F}$, we identify f^0 and f; it follows that $\mathcal{T}(\mathcal{F}) \subseteq \mathcal{T}(\mathcal{F}^{\mathbb{N}})$, and $\mathcal{T}(\mathcal{F} \cup \{E\}) \subseteq \mathcal{T}_M$. We usually denote by $\bar{t}$ (or $\widehat{t}$) a marked term such that $\bar{t}^0 = t$ (where $t \in \mathcal{T}(\mathcal{F}^{\mathbb{N}} \cup \{E\}, \mathcal{V})$). The same rule will apply to substitutions and contexts. For a set of terms $T \subseteq \mathcal{T}(\mathcal{F}, \mathcal{V})$, we denote by $T^{\mathbb{N}}$ the set of terms $\{\bar{t} \in \mathcal{T}_M \mid t \in T\}$.

▶ **Example 4.1.** $\mathsf{m}(\mathsf{f}^3(E(\mathsf{a}^4, \mathsf{b}^1))) = 3, \mathsf{m}(x) = 0, \mathsf{m}(E(\mathsf{a}^1, \mathsf{b}^2)) = 0, \mathsf{mmax}(\mathsf{f}^3(E(\mathsf{a}^4, \mathsf{b}^1))) = 4, \mathsf{mmax}(E(\mathsf{a}^1, \mathsf{b}^2)) = 2$, and if $\bar{t} = \mathsf{g}^3(\mathsf{a}^0, E(x, \mathsf{b}^2))$, then $\bar{t}^1 = \mathsf{g}^1(\mathsf{a}^1, E(x, \mathsf{b}^1))$.

From now on and until the end of section 5, let us fix, a language $T \subseteq \mathcal{T}(\mathcal{F})$ recognized by a complete deterministic automaton, $\mathcal{A} = (\mathcal{F}, \mathcal{Q}_{\mathcal{A}}, \mathcal{Q}_{f,\mathcal{A}}, \Gamma_{\mathcal{A}})$.

We start giving some technical definitions and lemmas.

The Automaton $\mathcal{A}_{\mathcal{P}}$

▶ **Definition 4.2.** We denote by $\overline{\mathcal{A}}$ the (infinite) automaton $\overline{\mathcal{A}} := (\mathcal{F}^{\mathbb{N}}, \mathcal{Q}_{\mathcal{A}}, \mathcal{Q}_{f,A}, \Gamma_{\overline{A}})$, with: $\Gamma_{\overline{A}} = \{f^i(q_1, \ldots, q_n) \to q \mid i \in \mathbb{N}, (f(q_1, \ldots, q_n) \to q) \in \Gamma_{\mathcal{A}}\}$.

Note that $\overline{\mathcal{A}}$ is deterministic and complete over $\mathcal{F}^{\mathbb{N}}$, and contains all the rules $c^i \to q$ for $i \in \mathbb{N}$, $c \in \mathcal{F}_0$, $(c \to q) \in \Gamma_{\mathcal{A}}$.

▶ **Lemma 4.3.** *Let $\bar{t}, \widehat{t} \in \mathcal{T}(\mathcal{F}^{\mathbb{N}}), q \in \mathcal{Q}_{\mathcal{A}}, m > 0$. If $\bar{t} \to^*_{\overline{\mathcal{A}}} q$ then $\widehat{t} \to^*_{\overline{\mathcal{A}}} q$.*

▶ **Definition 4.4.** We define the (infinite) automaton $\mathcal{A}_{\mathcal{P}} := \{\mathcal{F}^{\mathbb{N}} \cup \{E\}, \mathcal{Q}_{\mathcal{P}}, \mathcal{Q}_{f,\mathcal{P}}, \Gamma_{\mathcal{P}}\}$ built from $\overline{\mathcal{A}}$, where $\mathcal{Q}_{\mathcal{P}} = \mathcal{P}(\mathcal{Q}_{\mathcal{A}})$, $\mathcal{Q}_{f,\mathcal{P}} = \{\{q\} \mid q \in \mathcal{Q}_{f,\mathcal{A}}\}$, $\Gamma_{\mathcal{P}} = \{E(S_1, S_2) \to S_1 \cup S_2 \mid S_1, S_2 \in \mathcal{Q}_{\mathcal{P}}\} \cup \{f^i(S_1, \ldots, S_n) \to S_{f^i(S_1,\ldots,S_n)} \mid i \in \mathbb{N}, f \in \mathcal{F}_n, S_1, \ldots, S_n \in \mathcal{Q}_{\mathcal{P}}\}$ with $S_{f^i(S_1,\ldots,S_n)} = \{q \in \mathcal{Q}_{\mathcal{A}} \mid \forall j \in \{1, \ldots, n\}, \exists s_j \in S_j \text{ s.t. } f^i(s_1, \ldots, s_n) \to q \in \Gamma_{\overline{\mathcal{A}}}\}$

Note that subsets rules are obtained like in a classical determinization procedure $\mathcal{A}_{\mathcal{P}}$ contains all the rules $c^i \to \{q\}$ for $c \in \mathcal{F}_0$, $i \in \mathbb{N}$, $c \to q \in \Gamma_{\mathcal{A}}$, and that $\mathcal{A}_{\mathcal{P}}$ is deterministic and complete over $\mathcal{F}^{\mathbb{N}} \cup \{E\}$. The language recognized by $\mathcal{A}_{\mathcal{P}}$ is $\mathcal{L}(\mathcal{A}_{\mathcal{P}}) = T^{\mathbb{N}}$. For every term $\bar{t} \in \mathcal{T}(\mathcal{F}^{\mathbb{N}} \cup \{E\} \cup \mathcal{Q}_{\mathcal{P}})$, there is a unique state $Q \in \mathcal{Q}_{\mathcal{P}}$ such that $\bar{t} \to_{\mathcal{A}_{\mathcal{P}}} Q$. The state Q is the normal form associated to $\bar{t}$ and is denoted by $\mathsf{nf}_{\mathcal{A}_{\mathcal{P}}}(\bar{t})$. Since $\mathcal{A}_{\mathcal{P}}$ erases the marks $\mathsf{nf}_{\mathcal{A}_{\mathcal{P}}}(\bar{t}) = \mathsf{nf}_{\mathcal{A}_{\mathcal{P}}}(t)$. We extend the operation m to $\mathcal{T}(\mathcal{F}^{\mathbb{N}} \cup \mathcal{Q}_{\mathcal{P}} \cup \{E\}, \mathcal{V})$ by setting $\mathsf{m}(S) = 0$. For a term $t \in \mathcal{T}(\mathcal{F}^{\mathbb{N}} \cup \mathcal{Q}_{\mathcal{P}} \cup \{E\}, \mathcal{V})$, and an integer i we denote by t^i the term obtained by setting all the marks in t at a position $u \in \mathcal{P}\mathsf{os}_{\mathcal{F}}(t)$ to i, and we usually denote by $\bar{t}$ or $\widehat{t}$ a term such that $\bar{t}^0 = t$ (where $t \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q}_{\mathcal{P}} \cup \{E\}, \mathcal{V})$).

▶ **Example 4.5.** Let us consider the automaton $\mathcal{A}_1$ from example 2.1. The following rules belong to the set of rules of $\mathcal{A}_{1\mathcal{P}}$: $E(\{\mathsf{q}_\mathsf{a}\}, \{\mathsf{q}_\mathsf{b}\}) \to \{\mathsf{q}_\mathsf{a}, \mathsf{q}_\mathsf{b}\}, \mathsf{a}^3 \to \{\mathsf{q}_\mathsf{a}\}, \mathsf{h}^2(\{\mathsf{q}_\mathsf{a}, \mathsf{q}_\perp\}) \to \{\mathsf{q}_\mathsf{a}, \mathsf{q}_\perp\}, \mathsf{i}^1(\{\mathsf{q}_\mathsf{a}, \mathsf{q}_\mathsf{b}\}, \{\mathsf{q}_\mathsf{b}\}) \to \{\mathsf{q}_\perp, \mathsf{q}_f\}$,

$E(\{\mathsf{q}_\mathsf{a}, \mathsf{q}_f\}, \{\mathsf{q}_\mathsf{b}, \mathsf{q}_\mathsf{a}\}) \to \{\mathsf{q}_\mathsf{a}, \mathsf{q}_\mathsf{b}, \mathsf{q}_f\}, \mathsf{g}^4(\{\mathsf{q}_\mathsf{a}\}, \{\mathsf{q}_\mathsf{b}\}) \to \{\mathsf{q}_\perp\}$.

▶ **Definition 4.6.** We denote by $\mathcal{A}^+_{\mathcal{P}}$ the automaton $(\mathcal{F}^{\mathbb{N}} \cup \{E\}, \mathcal{Q}_{\mathcal{P}}, \mathcal{Q}_{f,\mathcal{P}}, \Gamma^+_{\mathcal{P}})$, where $\Gamma^+_{\mathcal{P}} = \Gamma_{\mathcal{P}} \cup \{S \to S' \mid S \in \mathcal{Q}_{\mathcal{P}}, S' \subset S\}$.

The language recognized by the automaton $\mathcal{A}^+_{\mathcal{P}}$ is $\mathcal{L}(\mathcal{A}^+_{\mathcal{P}}) = (\to^*_{\{E(x,y)\to x, E(x,y)\to y\}})[T^{\mathbb{N}}]$.

▶ **Definition 4.7.** For an automaton $\mathcal{C} = (\mathcal{F}^{\mathbb{N}} \cup \{E\}, \mathcal{Q}_{\mathcal{C}}, \mathcal{Q}_{\mathcal{C},f}, \Gamma_{\mathcal{C}})$ and for every $n \in \mathbb{N}$, we denote by $\mathcal{C}^{\leq n}$ (respectively $\mathcal{C}^{\geq n}$) the (finite) automaton $\mathcal{C}^{\leq n} := (\mathcal{F}^{\leq n} \cup \{E\}, \mathcal{Q}_{\mathcal{C}}, \mathcal{Q}_{\mathcal{C},f}, {\Gamma_{\mathcal{C}}}^{\leq n})$ (resp. $\mathcal{C}^{\geq n} := (\mathcal{F}^{\geq n} \cup \{E\}, \mathcal{Q}_{\mathcal{C}}, \mathcal{Q}_{\mathcal{C},f}, {\Gamma_{\mathcal{C}}}^{\geq n})$) where ${\Gamma_{\mathcal{C}}}^{\leq n} := \{l \to r \in \Gamma_{\mathcal{C}} \mid l, r \in \mathcal{T}(\mathcal{F}^{\leq n} \cup \{E\} \cup \mathcal{Q}_{\mathcal{C}})\}$ (resp. ${\Gamma_{\mathcal{C}}}^{\geq n} := \{l \to r \in \Gamma_{\mathcal{C}} \mid l, r \in \mathcal{T}(\mathcal{F}^{\geq n} \cup \{E\} \cup \mathcal{Q}_{\mathcal{C}})\}$).

The automaton ${\mathcal{A}^+_{\mathcal{P}}}^{\geq k+1}$ will be used to define the top of a marked term, i.e. the top part of the term that could be used in a k-bounded derivation (see definition 5.3 given further). The automaton ${\mathcal{A}^+_{\mathcal{P}}}^{\leq k}$ will be a part of the GRS $\mathcal{G}$ used to simulate k-bounded derivations (see definition 5.14).

▶ **Definition 4.8.** For all linear terms $\bar{t} \in \mathcal{T}(\mathcal{F}^{\mathbb{N}} \cup \{E\} \cup \mathcal{Q}_{\mathcal{P}}, \mathcal{V})$, for all $n \in \mathbb{N}$, we define $\bar{t} \odot n$ as the unique marked term such that $(\bar{t} \odot n)^0 = t$, and, $\forall u \in \mathcal{P}\mathsf{os}_{\mathcal{F}}(t)$, $\mathsf{m}(\bar{t} \odot n/u) = \max(\mathsf{m}(\bar{t}/u), \mathsf{Card}({\mathcal{P}\mathsf{os}_{\setminus E}}^{\prec u}(t)) + n)$

4.2 Marked Rewriting

From now on and until the end of this paper, let us fix an integer $k > 0$. We introduce here the rewrite relation $\circ\!\!\to$ between marked terms.

▶ **Definition 4.9** (Marked rewriting step)**.** A ground marked term $\overline{s} \in \mathcal{T}_M$ rewrites to a ground marked term $\overline{t} \in \mathcal{T}_M$ in $\mathcal{R} \cup \mathcal{E}$ if there exist a rule $l \to r \in \mathcal{R} \cup \mathcal{E}$, a position $v \in \mathcal{P}\mathsf{os}(s)$, a marked term $\overline{l}$, and a marked substitution $\overline{\sigma}$ such that : $\overline{s} = \overline{s}[\overline{l}\overline{\sigma}]_v, \overline{t} = \overline{s}[r(\overline{\sigma} \odot j)]_v$, where: $j = 0$, if $l \to r \in \mathcal{E}$, and $j = 1$, if $l \to r \in \mathcal{R}$. We then just write $\overline{s} \circ\!\!\to_{\mathcal{R}\cup\mathcal{E}} \overline{t}$.

We may omit $\mathcal{R} \cup \mathcal{E}$ when it is clear from the context. We use two different marking ($j = 0$ or $j = 1$) depending on the rule applied only to properly extend the notion of weakly bottom-up for linear TRSs (defined in [5]) to left-linear TRSs (see section 6). This notion is helpful to prove that several already known classes of TRSs belong to the class of bounded TRSs. Let us give some properties of marked derivations.

Associated Marked Derivation

Every derivation
$d : \ s_0 = s_0[l_0\sigma_0]_{v_0} \to_{\mathcal{R}\cup\mathcal{E}} s_0[r_0\sigma_0]_{v_0} = s_1 \to_{\mathcal{R}\cup\mathcal{E}} \ldots \to_{\mathcal{R}\cup\mathcal{E}} s_{n-1}[r_{n-1}\sigma_{n-1}]_{v_{n-1}} = s_n$,
is mapped to a marked derivation $\overline{d}$ called the *marked derivation associated* to d
$\overline{d} : \ \overline{s_0} = \overline{s_0}[\overline{l_0}\overline{\sigma_0}]_{v_0} \circ\!\!\to_{\mathcal{R}\cup\mathcal{E}} \overline{s_0}[r_0(\overline{\sigma_0} \odot i_0)]_{v_0} = \overline{s_1} \circ\!\!\to_{\mathcal{R}\cup\mathcal{E}}$
$\ldots \circ\!\!\to_{\mathcal{R}\cup\mathcal{E}} \overline{s_{n-1}}[r_{n-1}(\overline{\sigma_{n-1}} \odot i_{n-1})]_{v_{n-1}} = \overline{s_n}$
where $\overline{s_0} = s_0$. Note that this map is unique since the position v_j, the rule (l_j, r_j), and $\overline{s_j}$ completely determine $\overline{s}_{j+1}$.

4.3 Bounded Derivations and Bounded TRSs

▶ **Definition 4.10** (Bounded derivations)**.** A marked rewriting step $\overline{s} = \overline{s}[\overline{l}\overline{\sigma}]_v \circ\!\!\to_{\mathcal{R}\cup\mathcal{E}} \overline{t} = \overline{s}[r(\overline{\sigma} \odot j)]_v$ is *k-bounded* ($\mathsf{bo}(k)$) if $l \to r \in \mathcal{E}$ or if $l \to r \in \mathcal{R}$ and the following assertion holds: $(l \notin \mathcal{V} \Rightarrow \mathsf{mmax}(\overline{l}) \leq k)$ and $(l \in \mathcal{V} \Rightarrow \sup(\{\mathsf{m}(\overline{s}/u) \mid u \prec v\}) \leq k)$.
A marked derivation in $\mathcal{R} \cup \mathcal{E}$ is $\mathsf{bo}(k)$ if all its rewriting steps are $\mathsf{bo}(k)$. A derivation in $\mathcal{R} \cup \mathcal{E}$ is $\mathsf{bo}(k)$ if the associated marked derivation is $\mathsf{bo}(k)$. A derivation $s \to^*_{\mathcal{R}} t, s, t \in \mathcal{T}(\mathcal{F})$

is k-*bounded convertible* ($\mathsf{boc}(k)$) if there exists a $\mathsf{bo}(k)$-derivation $s \to^*_{\mathcal{R}\cup\mathcal{E}} t$ in $\mathcal{R}\cup\mathcal{E}$. The left-linear TRS $\mathcal{R}$ is k-bounded if every derivation in $\mathcal{R}$ is $\mathsf{boc}(k)$. We denote by $\mathsf{BO}(k)$ the class of k-bounded TRS and by BO the class $\bigcup_{k\in\mathbb{N}} \mathsf{BO}(k)$.

▶ **Example 4.11.** Let $\mathcal{R}_1$ be the TRS of example 2.1, and let
$d : s_0 = \mathsf{f}(\mathsf{h}(\mathsf{a})) \to_{\mathsf{f}(x)\to\mathsf{g}(x,x)} s_1 = \mathsf{g}(\mathsf{h}(\mathsf{a}),\mathsf{h}(\mathsf{a})) \to_{\mathsf{a}\to\mathsf{b}} s_2 = \mathsf{g}(\mathsf{h}(\mathsf{a}),\mathsf{h}(\mathsf{b})) \to_{\mathsf{g}(\mathsf{h}(x),y)\to\mathsf{i}(x,y)}$
$s_3 = \mathsf{i}(\mathsf{a},\mathsf{h}(\mathsf{b})) \to_{\mathsf{h}(\mathsf{b})\to\mathsf{b}} s_4 = \mathsf{i}(\mathsf{a},\mathsf{b})$ be a derivation in $\mathcal{R}_1$. Let us prove that this derivation is $\mathsf{boc}(1)$, i.e. that there is a derivation d' in $\mathcal{R}_1\cup\mathcal{E}$ which is $\mathsf{bo}(1)$.

- Let us take $d' = d$. We are going to prove that this derivation is $\mathsf{bo}(2)$ but not $\mathsf{bo}(1)$, i.e. that in the associated marked derivation, the maximal mark that appears on a lhs is 2 By definition, $\overline{d}$ starts on the term $\mathsf{f}^0(\mathsf{h}^0(\mathsf{a}^0))$. To build the associated marked derivation, we just apply the marking process exposed in section 4.2. We obtain the following derivation $\overline{d} : \overline{s_0} = \mathsf{f}^0(\mathsf{h}^0(\mathsf{a}^0)) \circ\!\to \overline{s_1} = \mathsf{g}^0(\mathsf{h}^1(\mathsf{a}^2),\mathsf{h}^1(\mathsf{a}^2)) \circ\!\to \overline{s_2} = \mathsf{g}^0(\mathsf{h}^1(\mathsf{a}^2),\mathsf{h}^1(\mathsf{b}^0))$ $\circ\!\to \overline{s_3} = \mathsf{i}^0(\mathsf{a}^2,\mathsf{h}^1(\mathsf{b}^2)) \circ\!\to \overline{s_4} = \mathsf{i}^0(\mathsf{a}^2,\mathsf{b}^0)$. We now look at the marks that appear on a lhs during this derivation. The lhs are f^0, a^2, $\mathsf{g}^0(\mathsf{h}^1(x),y)$, and $\mathsf{h}^1(\mathsf{b}^2)$, and the maximal mark that appears on the lhs is 2. Thus, we have proved that d is $\mathsf{boc}(2)$, but we want to prove that d is $\mathsf{boc}(1)$.
- To obtain a derivation d' which is $\mathsf{bo}(1)$, we apply the rules going from the bottom to the top. We apply the rules in this order: $\mathsf{a}\to\mathsf{b}$, $\mathsf{h}(\mathsf{b})\to\mathsf{b}$, then $\mathsf{f}(x)\to\mathsf{g}(x,x)$ and, to finish $\mathsf{g}(\mathsf{h}(x),y)\to\mathsf{i}(x,y)$. Since the rule $\mathsf{f}(x)\to\mathsf{g}(x,x)$ is not linear and duplicates the variable x we need to use the symbol E to apply the rules in the correct order. We introduce an E just above a and then apply the rule $\mathsf{a}\to\mathsf{b}$:
  $\mathsf{f}^0(\mathsf{h}^0(\mathsf{a}^0)) \to_{x\to E(x,x)} \mathsf{f}^0(\mathsf{h}^0(E(\mathsf{a}^0,\mathsf{a}^0))) \to_{\mathsf{a}\to\mathsf{b}} \mathsf{f}^0(\mathsf{h}^0(E(\mathsf{a}^0,\mathsf{b}^0)))$.
  Now, we introduce a second symbol E above the symbol h and get ride of the first one with selection rules, and then apply the rule $\mathsf{h}(\mathsf{b})\to\mathsf{b}$:
  $\mathsf{f}^0(\mathsf{h}^0(E(\mathsf{a}^0,\mathsf{b}^0))) \to_{x\to E(x,x)} \mathsf{f}^0(E(\mathsf{h}^0(E(\mathsf{a}^1,\mathsf{b}^1)),\mathsf{h}^0(E(\mathsf{a}^1,\mathsf{b}^1)))) \to_{E(x,y)\to x}$
  $\mathsf{f}^0(E(\mathsf{h}^0(\mathsf{a}^1),\mathsf{h}(E(\mathsf{a}^1,\mathsf{b}^1)))) \to_{E(x,y)\to y} \mathsf{f}^0(E(\mathsf{h}^0(\mathsf{a}^1),\mathsf{h}^0(\mathsf{b}^1))) \to_{\mathsf{h}(\mathsf{b})\to\mathsf{b}} \mathsf{f}^0(E(\mathsf{h}^0(\mathsf{a}^1),\mathsf{b}^0))$.
  Hence, we apply the rule $\mathsf{f}(x)\to\mathsf{g}(x,x)$:
  $\mathsf{f}^0(E(\mathsf{h}^0(\mathsf{a}^1),\mathsf{b}^0)) \to_{\mathsf{f}(x)\to\mathsf{g}(x,x)} \mathsf{g}^0(E(\mathsf{h}^1(\mathsf{a}^2),\mathsf{b}^1),E(\mathsf{h}^1(\mathsf{a}^2),\mathsf{b}^1))$, then select the needed copies:
  $\mathsf{g}^0(E(\mathsf{h}^1(\mathsf{a}^2),\mathsf{b}^1),E(\mathsf{h}^1(\mathsf{a}^2),\mathsf{b}^1)) \to_{E(x,y)\to x} \mathsf{g}^0(\mathsf{h}^1(\mathsf{a}^2),E(\mathsf{h}^1(\mathsf{a}^2),\mathsf{b}^1))$
  $\to_{E(x,y)\to y} \mathsf{g}^0(\mathsf{h}^1(\mathsf{a}^2),\mathsf{b}^1)$,
  and apply the last rule:
  $\mathsf{g}^0(\mathsf{h}^1(\mathsf{a}^2),\mathsf{b}^1) \to_{\mathsf{g}(\mathsf{h}(x),y)\to\mathsf{i}(x,y)} \mathsf{i}^0(\mathsf{a}^2,\mathsf{b}^1)$.
  The maximal mark that appears on a lhs is 1. So, d' is $\mathsf{bo}(1)$ and d is $\mathsf{boc}(1)$.

Let us introduce a convenient notation.

▶ **Definition 4.12.** The binary relation ${}_k\circ\!\to_{\mathcal{R}\cup\mathcal{E}}$ over $\mathcal{T}(\mathcal{F}^{\mathbb{N}}\cup\{E\})$ is defined by $\overline{s}\ {}_k\circ\!\to_{\mathcal{R}\cup\mathcal{E}}\ \overline{t}$ if there is a $\mathsf{bo}(k)$ marked rewriting step in $\mathcal{R}\cup\mathcal{E}$ between $\overline{s}$ and $\overline{t}$. The binary relation ${}_k\circ\!\to^*_{\mathcal{R}\cup\mathcal{E}}$ over $\mathcal{T}(\mathcal{F}^{\mathbb{N}}\cup\{E\})$ is defined by $\overline{s}\ {}_k\circ\!\to^*_{\mathcal{R}\cup\mathcal{E}}\ \overline{t}$ if there is a $\mathsf{bo}(k)$ marked derivation from $\overline{s}$ to $\overline{t}$. The binary relation ${}_k\to^*_{\mathcal{R}}$ over $\mathcal{T}(\mathcal{F})$ is defined by $s\ {}_k\to^*_{\mathcal{R}}\ t$ if there is a $\mathsf{boc}(k)$ derivation in $\mathcal{R}$ from s to t.

Since the composition of two $\mathsf{bo}(k)$ marked derivations is a $\mathsf{bo}(k)$ marked derivation, ${}_k\circ\!\to^*_{\mathcal{R}\cup\mathcal{E}}$ is the transitive and reflexive closure of ${}_k\circ\!\to_{\mathcal{R}\cup\mathcal{E}}$. Note that the composition of two $\mathsf{boc}(k)$-derivation is not always a $\mathsf{boc}(k)$-derivation.

Let us recall the notion of linear k-bounded rewriting defined in [6] which will be denoted here lbo to avoid confusion.

▶ **Definition 4.13.** Let $\mathcal{R}$ be a linear TRS. A marked rewriting step $\overline{s} = \overline{s}[\overline{l}\overline{\sigma}]_v \circ\!\!\rightarrow_{\mathcal{R}} \overline{t} = \overline{s}[r(\overline{\sigma} \odot 1)]_v$ is *linear k-bounded* ($\mathsf{lbo}(k)$) if the following assertion holds

$$(l \notin \mathcal{V} \Rightarrow \mathsf{mmax}(\overline{l}) \leq k), \text{ and } (l \in \mathcal{V} \Rightarrow \sup(\{\mathsf{m}(\overline{s}/u) \mid u \prec v\}) \leq k) \quad (3)$$

A marked derivation is $\mathsf{lbo}(k)$ if all its rewriting steps are $\mathsf{bo}(k)$. A derivation in $\mathcal{R}$ is $\mathsf{lbo}(k)$ if the associated marked derivation is $\mathsf{lbo}(k)$. The TRS $\mathcal{R}$ is linear k-bounded if every derivation $s \rightarrow_{\mathcal{R}}^* t$ can be replaced by a $\mathsf{lbo}(k)$ derivation from s to t. We denote by $\mathsf{LBO}(k)$ the class of linear k-bounded TRSs and by LBO the class $\bigcup_{k\in\mathbb{N}} \mathsf{LBO}(k)$.

By definition, $\mathsf{LBO}(k) \subseteq \mathsf{BO}(k)$. Moreover, one can easily check that for every linear TRS $\mathcal{R}$, $\mathcal{R} \in \mathsf{LBO}(k)$ iff $\mathcal{R} \in \mathsf{BO}(k)$. Since the $\mathsf{LBO}(0)$ membership problem is undecidable, the $\mathsf{BO}(0)$ membership problem is undecidable too. Note that in the definition of an $\mathsf{lbo}(k)$-derivation, nothing requires the linear condition. But if we consider $\mathsf{lbo}(k)$-derivations for left-linear TRSs, then the class $\mathsf{LBO}(k)$ does not contain left-linear TRSs with non right-linear rules. This is illustrated in the following example.

▶ **Example 4.14.** Let $\mathcal{R}_2 = \{\mathsf{f}(x) \to \mathsf{g}(x,x), \mathsf{a} \to \mathsf{b}\}$ and let $k \in \mathbb{N}$. There is a $\mathsf{bo}(0)$-derivation $\mathsf{f}(\mathsf{f}(\ldots(\mathsf{f}(\mathsf{a}))\ldots)) \to_{\mathcal{E}} \mathsf{f}(\mathsf{f}(\ldots(\mathsf{f}(E(\mathsf{a},\mathsf{a}))\ldots))) \to_{\mathsf{a}\to\mathsf{b}} \mathsf{f}(\mathsf{f}(\ldots(\mathsf{f}(E(\mathsf{a},\mathsf{b}))\ldots)))$ $\to_{\mathsf{f}(x)\to\mathsf{g}(x,x)} \mathsf{g}(\mathsf{f}(\ldots(\mathsf{f}(E(\mathsf{a},\mathsf{b}))\ldots)),\mathsf{f}(\ldots(\mathsf{f}(E(\mathsf{a},\mathsf{b}))\ldots))) \to_{E(x,y)\to x}$ $\mathsf{g}(\mathsf{f}(\ldots(\mathsf{f}(\mathsf{a})\ldots)),\mathsf{f}(\ldots(\mathsf{f}(E(\mathsf{a},\mathsf{b}))\ldots))) \to_{E(x,y)\to y} \mathsf{g}(\mathsf{f}(\ldots(\mathsf{f}(\mathsf{a}))\ldots),\mathsf{f}(\ldots(\mathsf{f}(\mathsf{b}))\ldots))$ but there is no $\mathsf{bo}(k)$-derivation from $\mathsf{f}(\mathsf{f}(\ldots(\mathsf{f}(\mathsf{a}))\ldots))$ to $\mathsf{g}(\mathsf{f}(\ldots(\mathsf{f}(\mathsf{a}))\ldots),\mathsf{f}(\ldots(\mathsf{f}(\mathsf{b}))\ldots))$ which does not use the rules of $\mathcal{E}$. Note that the TRS $\mathcal{R}_2$ is $\mathsf{bo}(0)$ since every derivation in $\mathcal{R}_2$ is $\mathsf{boc}(0)$.

Well-marked Derivation

Terms that appear on a marked derivation starting on a term $s \in \mathcal{T}(\mathcal{F})$ have a special form and are said to be *well-marked*.

▶ **Definition 4.15** (well-marked)**.** A term $\overline{s} \in \mathcal{T}(\mathcal{F}^{\mathbb{N}} \cup \{E\} \cup \mathcal{Q}_{\mathcal{P}}, \mathcal{V})$ is *well-marked* for k if these two assertions holds

1. for all $w \in \mathcal{P}\mathsf{os}_{\mathcal{V}}(s)$, for all $v \preceq w$, $\mathsf{m}(\overline{s}/v) \leq k$,
2. for all $w \in \mathcal{L}\mathsf{v}(s) \backslash \mathcal{P}\mathsf{os}_{\mathcal{V}}(s)$, one of these two assertions holds
   - **a.** for all $v \preceq w$, $\mathsf{m}(\overline{s}/v) \leq k$,
   - **b.** there exists $u \in \mathcal{P}\mathsf{os}_{\mathcal{F}}{}^{\preceq w}(\overline{s})$ such that:
     - for all $v \prec u$, $\mathsf{m}(\overline{s}/v) \leq k$,
     - for all $v \in \mathcal{P}\mathsf{os}_{\mathcal{F}}{}^{\preceq w}(s)$ such that $v \succeq u$, $\mathsf{m}(\overline{s}/v) = k + 1 + \mathsf{Card}(\mathcal{P}\mathsf{os}_{\backslash E}{}^{\prec v}(s)) - \mathsf{Card}(\mathcal{P}\mathsf{os}_{\backslash E}{}^{\prec u}(s))$.

A marked derivation is *well-marked* if every term in the derivation is well-marked.

So, a term is well-marked if for every $w \in \mathcal{L}\mathsf{v}(t)$, the sequence of marks on the symbols of $\mathcal{F}$ that appear on the branch containing w has the form: $m_0, m_1, \ldots, m_n, k+1, k+2, \ldots, k+l$ with $m_i \leq k$ in case 2*b.* is satisfied and $m_0, m_1, \ldots, m_n$ with $m_i \leq k$ in case 1. or 2*a.* is satisfied. Note that an unmarked term is well-marked, and that condition 2*a.* is equivalent to for all $v \prec u$,$\mathsf{m}(\overline{s}/v) \leq k$, $\mathsf{m}(\overline{s}/u) = k+1$, and for all $v \in \mathcal{P}\mathsf{os}_{\mathcal{F}}{}^{\preceq w}(s)$ such that $v \succeq u$, $\mathsf{m}(\overline{s}/v) - \mathsf{m}(\overline{s}/u) = \mathsf{Card}(\mathcal{P}\mathsf{os}_{\backslash E}{}^{\prec v}(s)) - \mathsf{Card}(\mathcal{P}\mathsf{os}_{\backslash E}{}^{\prec u}(s))$.

▶ **Example 4.16.** Let $k = 3$ and let $\mathcal{R}_1$ and $\mathcal{A}_1$ be the TRS and the automaton from example 2.1. The terms $\mathsf{f}^1(E(\mathsf{f}^2(\mathsf{a}^2),x))$, $\mathsf{f}^0(E(\mathsf{f}^3(\mathsf{a}^4),x))$, $\mathsf{f}^0(\mathsf{f}^3(E(\mathsf{f}^4(\mathsf{a}^5),\mathsf{f}^3(\mathsf{a}^3))))$, $\mathsf{f}^2(E(\mathsf{f}^0(\mathsf{a}^4),x))$, $\mathsf{f}^4(\mathsf{f}^5(E(\mathsf{f}^6(\{\mathsf{q}_\mathsf{a},\mathsf{q}_\perp\}),\mathsf{f}^6(\mathsf{b}^7))))$ are well-marked. The terms $\mathsf{f}^4(E(\mathsf{f}^5(\mathsf{a}^6),x))$, $\mathsf{f}^2(\mathsf{f}^3(E(\mathsf{f}^4(\mathsf{a}^4),\mathsf{f}^3(\mathsf{a}^4))))$, and $\mathsf{f}^2(\mathsf{f}^3(E(\mathsf{f}^4(\mathsf{a}^6),\mathsf{f}^3(\{\mathsf{q}_\mathsf{b},\mathsf{q}_f\}))))$ are not well-marked since neither 1 or 2 hold.

▶ **Lemma 4.17.** *A* $\mathsf{bo}(k)$*-derivation is well-marked iff it is starting on a well-marked term.*

5 Main Result

The main theorem of this section (and of the paper) is the following.

▶ **Theorem 5.1.** *Let* $\mathcal{R}$ *be a left-linear rewriting* TRS *over a signature* $\mathcal{F}$*. Let* T *be some recognizable subset of* $\mathcal{T}(\mathcal{F})$ *and let* $k > 0$*. Then, the set* $({}_k{\to}^*_{\mathcal{R}})[T]$ *is recognizable too and can be built.*

To obtain this result, we simulate $\mathsf{bo}(k)$-derivations using a GTT. The construction of the proof can be divided into three steps:

- First, we define the top part $\mathsf{Top}(\overline{t})$ of a well-marked term $\overline{t}$ which is the part of $\overline{t}$ that could be rewritten using a rule of $\mathcal{R}$ in a $\mathsf{bo}(k)$-derivation.
- Then we define a GRS $\mathcal{G}$ which has the following properties:
  - If $\overline{s} \to^*_{\mathcal{G}} \overline{t}$, then there exists $\overline{t'}$ such that $\overline{s}\ {}_k{\circ}{\to}^*_{\mathcal{R}}\ \overline{t'} \to^*_{\mathcal{A}_{\mathcal{P}^+}} \overline{t}$ (lifting rewriting with $\mathcal{G}$ to $\mathcal{R}$).
  - If $\overline{s}\ {}_k{\circ}{\to}^*_{\mathcal{R}}\ \overline{t}$ then $\mathsf{Top}(\overline{s}) \to^*_{\mathcal{G}} \mathsf{Top}(\overline{t})$ (projecting rewriting with $\mathcal{R}$ to $\mathcal{G}$).
- From these two properties of $\mathcal{G}$ and using some technical lemmas, we obtain the simulation lemma 5.22. The relation $\to^*_{\mathcal{G}}$ is recognizable by a GTT, and since GTTs are effectively inverse recognizability preserving, we obtain theorem 5.1.

Top of a Marked Term

By definition of a $\mathsf{bo}(k)$-derivation, a symbol in a term $\overline{t}$ can match a *lhs* of a rule of $\mathcal{R}$ only if the mark of this symbol is smaller or equal to k. This leads us to define the top part of a well-marked term $\overline{t}$ which is (intuitively) obtained by replacing all the useless subterms $\overline{t}/u$ by their normal form $\mathsf{nf}_{\mathcal{A}_{\mathcal{P}}}(\overline{t}/u)$.

▶ **Definition 5.2.** Let $\overline{t} \in \mathcal{T}(\mathcal{F}^{\mathbb{N}} \cup \{E\} \cup \mathcal{Q}_{\mathcal{P}}, \mathcal{V})$ be well-marked. We define $\mathsf{Topd}(\overline{t})$ the top domain of $\overline{t}$ as: $u \in \mathsf{Topd}(\overline{t})$ iff $u \in \mathcal{P}\mathsf{os}(t)$ and $\forall v \prec u, \mathsf{m}(\overline{t}/v) \leq k$.

▶ **Definition 5.3.** Let $\overline{t} \in \mathcal{T}(\mathcal{F}^{\mathbb{N}} \cup \{E\} \cup \mathcal{Q}_{\mathcal{P}}, \mathcal{V})$ be well-marked. We denote by $\mathsf{Top}(\overline{t})$ the unique term such that

- $\mathcal{P}\mathsf{os}(\mathsf{Top}(\overline{t})) = \mathsf{Topd}(\overline{t})$,
- $\overline{t} \to^*_{\mathcal{A}_{\mathcal{P}^{\geq k+1}}} \mathsf{Topd}(\overline{t})$,
- for all $\overline{t'}$ such that $\mathcal{P}\mathsf{os}(\overline{t'}) = \mathsf{Topd}(\overline{t})$ and $\overline{t} \to^*_{\mathcal{A}_{\mathcal{P}^{\geq k+1}}} \overline{t'}$, we have $\overline{t'} \to^*_{\mathcal{A}_{\mathcal{P}^{\geq k+1}}} \mathsf{Top}(\overline{t})$.

▶ **Example 5.4.** Let $k = 3$ and let $\mathcal{R}_1$ and $\mathcal{A}_1$ be the TRS and the automaton from example 2.1. Let $\overline{t_0} = \mathsf{f}^0(E(\{\mathsf{q_a}\}, \mathsf{g}^0(\mathsf{a}^0, \mathsf{b}^0)))$, $\overline{t_1} = \mathsf{f}^2(E(\{\mathsf{q_a}\}, \mathsf{g}^0(\mathsf{a}^3, \mathsf{b}^4)))$, $\overline{t_2} = \mathsf{f}^2(E(\{\mathsf{q_a}\}, \mathsf{g}^3(\mathsf{a}^4, \mathsf{b}^4)))$, $\overline{t_3} = \mathsf{f}^2(E(\{\mathsf{q_a}\}, \mathsf{g}^4(\mathsf{a}^5, \mathsf{b}^5)))$, $\overline{t_4} = \mathsf{f}^4(E(\{\mathsf{q_a}\}, \mathsf{g}^5(\mathsf{a}^6, \mathsf{b}^6)))$. Note that these terms are well-marked. We have $\mathsf{Topd}(\overline{t_0}) = \mathsf{Topd}(\overline{t_1}) = \mathsf{Topd}(\overline{t_2}) = \mathcal{P}\mathsf{os}(t_0)$, $\mathsf{Topd}(\overline{t_3}) = \{\epsilon, 0, 00, 01\}$, $\mathsf{Topd}(\overline{t_4}) = \{\epsilon\}$ and $\overline{t_0} \to^0_{\mathcal{A}_{1\mathcal{P}^{\geq 4}}} \mathsf{Top}(\overline{t_0}) = \overline{t_0}$, $\overline{t_1} \to_{\mathcal{A}_{1\mathcal{P}^{\geq 4}}} \mathsf{f}^2(E(\{\mathsf{q_a}\}, \mathsf{g}^0(\mathsf{a}^3, \{\mathsf{q_b}\}))) = \mathsf{Top}(\overline{t_1})$, $\overline{t_2} \to_{\mathcal{A}_{1\mathcal{P}^{\geq 4}}} \mathsf{f}^2(E(\{\mathsf{q_a}\}, \mathsf{g}^3(\mathsf{a}^4, \{\mathsf{q_b}\}))) \to_{\mathcal{A}_{1\mathcal{P}^{\geq 4}}} \mathsf{f}^2(E(\{\mathsf{q_a}\}, \mathsf{g}^3(\{\mathsf{q_a}\}, \{\mathsf{q_b}\}))) = \mathsf{Top}(\overline{t_2})$, $\overline{t_3} \to_{\mathcal{A}_{1\mathcal{P}^{\geq 4}}} \mathsf{f}^2(E(\{\mathsf{q_a}\}, \mathsf{g}^4(\mathsf{a}^5, \{\mathsf{q_b}\}))) \to_{\mathcal{A}_{1\mathcal{P}^{\geq 4}}} \mathsf{f}^2(E(\{\mathsf{q_a}\}, \mathsf{g}^4(\{\mathsf{q_a}\}, \{\mathsf{q_b}\}))) \to_{\mathcal{A}_{1\mathcal{P}^{\geq 4}}}$ $\mathsf{f}^2(E(\{\mathsf{q_a}\}, \{\mathsf{q}_\perp\})) = \mathsf{Top}(\overline{t_3})$, $\overline{t_4} \to_{\mathcal{A}_{1\mathcal{P}^{\geq 4}}} \mathsf{f}^4(E(\{\mathsf{q_a}\}, \mathsf{g}^5(\mathsf{a}^6, \{\mathsf{q_b}\}))) \to_{\mathcal{A}_{1\mathcal{P}^{\geq 4}}}$ $\mathsf{f}^4(E(\{\mathsf{q_a}\}, \mathsf{g}^5(\{\mathsf{q_a}\}, \{\mathsf{q_b}\}))) \to_{\mathcal{A}_{1\mathcal{P}^{\geq 4}}} \mathsf{f}^4(E(\{\mathsf{q_a}\}, \{\mathsf{q}_\perp\})) \to_{\mathcal{A}_{1\mathcal{P}^{\geq 4}}} \mathsf{f}^4(\{\mathsf{q_a}, \mathsf{q}_\perp\})$ $\to_{\mathcal{A}_{1\mathcal{P}^{\geq 4}}} \{\mathsf{q}_\perp\} = \mathsf{Top}(\overline{t_4})$

5.1 Definition of the GRS $\mathcal{G}$ Used for the Simulation

Comb Associated to a Term and the Set $\mathcal{C}_{\leq n}$

Before giving the definition of $\mathcal{G}$ we need to introduce some notations.

▶ **Definition 5.5.** We define the binary relation $\sqsubset$ over $\mathcal{T}(\mathcal{F}^{\mathbb{N}} \cup \mathcal{Q}_{\mathcal{P}} \cup \{E\})$ by $s \sqsubset t$ if $\mathsf{root}(s) \in \mathcal{F}^{\mathbb{N}} \cup \mathcal{Q}_{\mathcal{P}}$, $\mathsf{root}(t) = E$, and there exists $u \in \{w \in \mathcal{P}\mathsf{os}_{\setminus E}(t) \mid \mathcal{P}\mathsf{os}^{\prec w}_{\setminus E}(t) = \emptyset\}$ such that $t/u = s$. We define $s \not\sqsubset t$ by $s \not\sqsubset t$ if $s \sqsubset t$ does not hold.

Note that for all t, $t \not\sqsubset t$.

▶ **Definition 5.6.** Let $t \in \mathcal{T}(\mathcal{F}^{\mathbb{N}} \cup \mathcal{Q}_{\mathcal{P}} \cup \{E\})$. Let B be the TRS $B := \{E(x, E(y, z)) -> E(E(x, y), z)\}$. We denote by $\ll t \gg$ the normal form associated to t: $\ll t \gg = \mathsf{nf}_B(t)$.

Note that B can be easily shown to be terminating and confluent.

▶ **Definition 5.7.** Let $t \in \mathcal{T}(\mathcal{F}^{\mathbb{N}} \cup \mathcal{Q}_{\mathcal{P}} \cup \{E\})$. Let D be the (infinite) ground TRS
$D := \{E(E(x\sigma, y\sigma), z\sigma) \to E(x\sigma, y\sigma) \mid \sigma : \mathcal{V} \to \mathcal{T}(\mathcal{F}^{\mathbb{N}} \cup \mathcal{Q}_{\mathcal{P}} \cup \{E\}) \ \&\ z\sigma \sqsubset E(x\sigma, y\sigma)\}$
$\cup \{E(E(x\sigma, y\sigma), z\sigma) \to E(x\sigma, z\sigma) \mid \sigma : \mathcal{V} \to \mathcal{T}(\mathcal{F}^{\mathbb{N}} \cup \mathcal{Q}_{\mathcal{P}} \cup \{E\}) \ \&\ x\sigma = y\sigma\}$.
The *comb* associated to t is denoted by $\lfloor t \rfloor$ and is defined by $\lfloor t \rfloor := \mathsf{nf}_D(\ll t \gg)$. We extend this definition to marked substitutions ($\lfloor \sigma \rfloor : x \mapsto \lfloor x\sigma \rfloor$).

Note that D can be shown to be terminating, and that there is a unique normal form associated to $\ll t \gg$.

▶ **Example 5.8.** Let $t_0 = E(a^0, a^0)$, $t_1 = E(a^1, E(a^2, b^1))$, $t_2 = E(a^1, E(a^1, b^1))$ and $t_3 = f(E(E(b^1, a^2), E(b^2, a^2)))$. We have $\ll t_0 \gg = t_0$, $\ll t_1 \gg = E(E(a^1, a^2), b^1)$, $\ll t_2 \gg = E(E(a^1, a^1), b^1)$, $\ll t_3 \gg = f(E(E(E(b^1, a^2), b^2), a^2))$, $\lfloor t_0 \rfloor = t_0$, $\lfloor t_1 \rfloor = \ll t_1 \gg$, $\ll t_2 \gg = E(a^1, b^1)$, and $\lfloor t_3 \rfloor = f(E(E(b^1, a^2), b^2))$.

▶ **Definition 5.9.** Let $n \in \mathbb{N}$. Let $A = \{\}$. We denote by $\mathcal{C}_{\leq n}$ the (finite) set of combs: $\mathcal{C}_{\leq n} := \{\lfloor t \rfloor \mid t \in \mathcal{T}(\mathcal{F}^{\leq k} \cup \mathcal{Q}_{\mathcal{P}} \cup \{E\}) \ \&\ dpt_{\setminus E}(t) \leq n\}$.

Note that for a comb t, the set of term $\{s \mid \lfloor s \rfloor = t\}$ is recognizable. The next lemma is used to prove the projecting lemma 5.18.

▶ **Proposition 5.10** (Comb form proposition). *Let $\overline{s}, \overline{t} \in \mathcal{T}(\mathcal{F}^{\leq k} \cup \mathcal{Q}_{\mathcal{P}} \cup \{E\})$, $\overline{u} \in \mathcal{C}_{\leq k+2}$, $n \in \mathbb{N}$. If $\lfloor \overline{s} \odot n \rfloor = \overline{u}$ and $\lfloor \overline{t} \odot n \rfloor = \overline{u}$, then $\overline{s} \odot n \circ\!\!\rightarrow^*_{\mathcal{E}} \overline{t} \odot n$.*

▶ **Definition 5.11.** Let $\overline{t} \in \mathcal{T}(\mathcal{F}^{\mathbb{N}} \cup \mathcal{Q}_{\mathcal{P}} \cup \{E\}, \mathcal{V})$, $\mathcal{V}\mathsf{ar}(t) = \{x_1, \ldots, x_n\}$. For all $1 \leq i \leq n$, let $j_i = \mathsf{Card}(\mathcal{P}\mathsf{os}(t, x_i))$, and let $\mathcal{P}\mathsf{os}(t, x_i) = \{v_{1,1}, \ldots, v_{1,j_i}\}$ where the $v_{p,q}$ are given in lexicographic order. We define $\mathsf{lin}(\overline{t})$ as the term
$\mathsf{lin}(\overline{t}) := \overline{t}[x_{1,1}, \ldots, x_{1,j_1}, \ldots, x_{n,1}, \ldots, x_{n,j_n}]_{v_{1,1},\ldots,v_{1,j_1},\ldots v_{n,1},\ldots v_{n,j_n}}$,
where the $x_{i,j}$ are distinct variables.

Each time we use the notation $\mathsf{lin}(\overline{t})$, we implicitly suppose that if $\mathcal{V}\mathsf{ar}(t) = \{x_1, \ldots, x_n\}$, then the variables in $\mathcal{V}\mathsf{ar}(\mathsf{lin}(t))$ are denoted $x_{i,j}$ as in definition 5.11.

Overview of the Simulation

Let us give an overview of the proof of the projecting and lifting lemmas used to simulate $\mathsf{bo}(k)$-derivations by a GTT (lemmas 5.18 and 5.15).
For every rule $l \to r \in \mathcal{R} \cup \mathcal{E}$, every term $\overline{l} \in \mathcal{T}(\mathcal{F}^{\leq k} \cup \mathcal{Q}_{\mathcal{P}} \cup \{E\})$ and every substitution $\overline{\tau} : \mathcal{V} \to \mathcal{C}_{\leq k+2}$ such that $\overline{\tau} \odot a : \mathcal{V} \to \mathcal{C}_{\leq k+2}$ (where $a = 1$ if $l \to r \in \mathcal{R}$, and $a = 0$ otherwise),

we build a GRS $\mathcal{G}_{\bar{l},\overline{r},\overline{\tau}} = (L, R)$, where L is the recognizable set containing all the terms $\bar{l}\overline{\sigma}$ such that $\lfloor\overline{\sigma}\rfloor = \overline{\tau}$ and R is the recognizable set containing all the terms $\mathsf{lin}(r)(\overline{\sigma} \odot a)$ such that for all $x_{i,j} \in \mathcal{V}\mathsf{ar}(\mathsf{lin}(r))$, the associated comb of $(x_{i,j}\overline{\sigma} \odot a)$ is $\lfloor x_i\overline{\tau} \odot a\rfloor$. The GRS $\mathcal{G}$ over $\mathcal{F}^{\leq k} \cup \mathcal{Q}_{\mathcal{P}} \cup \{E\}$ is hence defined as the union of all the GRS $\mathcal{G}_{\bar{l},r,\overline{\tau}}$ and $\mathcal{A}_{\mathcal{P}}^{+\leq k}$.
Now, let us see how the simulation works. The simulation is based on the projecting lemma 5.18 and the lifting lemma 5.15. Let us start with the projecting lemma. Let $l \to r \in \mathcal{R}$, and let us suppose that we want to simulate a rewriting step $\overline{s} = \overline{s}[\bar{l}\overline{\sigma}]_v \ {}_k\circ\!\!\to_{\mathcal{R}\cup\mathcal{E}} \bar{t} = \overline{s}[\overline{r\sigma} \odot a]_v$ using the GRS $\mathcal{G}$.

The projecting lemma 5.18 claims that we can rewrite the useful top part of $\overline{s}$ to the useful top part of $\bar{t}$, i.e. that $\mathsf{Top}(\overline{s}) = \mathsf{Top}(\overline{s})[\bar{l}\mathsf{Top}(\overline{\sigma})]_v \to_{\mathcal{G}}^* \mathsf{Top}(\bar{t}) = \mathsf{Top}(\overline{s})[r\mathsf{Top}(\overline{\sigma} \odot a)]$. We obtain this derivation in two steps:

- First, we cut the useless part of $\mathsf{Top}(\overline{s})$ using $\mathcal{A}_{\mathcal{P}}{}^{\leq k}$, i.e. the parts of $x\overline{\sigma}$ that are marked by an integer greater than k in $x\overline{\sigma} \odot a$. Let us denote by $\overline{\sigma'}$ the substitution obtained after this step (i.e. the unique substitution such that $\overline{\sigma'} \odot a = \mathsf{Top}(\overline{\sigma} \odot a)$).
- Then, since $l\lfloor\overline{\sigma'}\rfloor \to r\lfloor\overline{\sigma'} \odot a\rfloor \in \mathcal{G}(\bar{l}, r, \lfloor\overline{\sigma'}\rfloor)$, we obtain the required derivation $\mathsf{Top}(\overline{s}) = \overline{s}[\bar{l}\overline{\sigma}]_v \to_{\mathcal{A}_{\mathcal{P}}^{\leq k}} \mathsf{Top}(\overline{s})[\overline{l\sigma'}]_v \to_{\mathcal{G}(\bar{l},r,\lfloor\overline{\sigma'}\rfloor)} \mathsf{Top}(\overline{s})[r(\overline{\sigma'} \odot a)]_v = \mathsf{Top}(\overline{s})[r\mathsf{Top}(\overline{\sigma} \odot a)]_v = \mathsf{Top}(\bar{t})$.

Now, let us see how the lifting lemma works. Let $\overline{s} = \overline{s}[l\overline{\sigma}] \to_{\mathcal{G}(\bar{l},r,\overline{\tau})} \bar{t} = \overline{s}[\mathsf{lin}(r)\overline{\sigma'}]$. We want to prove that there exists a term $\overline{s'}$ and a derivation $\overline{s} \ {}_k\circ\!\!\to_{\mathcal{R}\cup\mathcal{E}} \overline{s'} \to_{\mathcal{A}_{\mathcal{P}}}^* \bar{t}$. First, we apply the rule $l \to r$. We obtain a derivation $\overline{s} = \overline{s}[\bar{l}\overline{\sigma}]_v \ {}_k\circ\!\!\to_{\mathcal{R}} \overline{s}[r(\overline{\sigma} \odot a)]$. We then use the comb form proposition 5.10, and some other technical lemmas to obtain a term $\overline{s'}$ and a derivation $\overline{s} \ {}_k\circ\!\!\to_{\mathcal{R}} \overline{s}[r(\overline{\sigma} \odot a)] \ {}_k\circ\!\!\to_{\mathcal{E}}^* \overline{s'} \to_{\mathcal{A}_{\mathcal{P}}} \bar{t}$.

The GRS $\mathcal{G}$

For every linear term $\bar{t} \in \mathcal{T}(\mathcal{F}^{\mathbb{N}} \cup \mathcal{Q}_{\mathcal{P}} \cup \{E\}, \mathcal{V})$, and every substitution $\overline{\sigma} : \mathcal{V} \to \mathcal{C}_{\leq k+2}$, the set $\{\bar{t}\,\overline{\sigma'} \mid \overline{\sigma'} : \mathcal{V} \to \mathcal{T}(\mathcal{F}^k \cup \mathcal{Q}_{\mathcal{P}} \cup \{E\}), \forall x \in \mathcal{V}, \lfloor x\overline{\sigma'}\rfloor = x\overline{\sigma}\}$ is recognizable.

▶ **Definition 5.12.** We denote by Λ_a the set of substitutions $\overline{\tau} : \mathcal{V} \to \mathcal{C}_{\leq k+2}$ such that $\lfloor\overline{\tau} \odot a\rfloor : \mathcal{V} \to \mathcal{C}_{\leq k+2}$.

▶ **Definition 5.13.** Let $l \to r \in \mathcal{R} \cup \mathcal{E}$, $\bar{l} \in \mathcal{T}(\mathcal{F}^{\leq k} \cup \mathcal{Q}_{\mathcal{P}} \cup \{E\}, \mathcal{V})$. Let $a = 0$ if $l \to r \in \mathcal{E}$, and let $a = 1$ if $l \to r \in \mathcal{R}$. Let $\mathcal{V}\mathsf{ar}(l) = \{x_1, \ldots, x_n\}$ and $\mathcal{V}\mathsf{ar}(r) = \{x_1, \ldots, x_m\}$. Let $\tau \in \Lambda_a$, and let

$L = \{\bar{l}\overline{\sigma} | \overline{\sigma} : \mathcal{V} \to \mathcal{T}(\mathcal{F}^{\leq k} \cup \mathcal{Q}_{\mathcal{P}} \cup \{E\}), \forall i \in \{1, \ldots, n\}, \lfloor x_i\overline{\sigma}\rfloor = x_i\overline{\tau}\}$, $R = \{\mathsf{lin}(r)(\overline{\sigma} \odot a) | \overline{\sigma} : \mathcal{V} \to \mathcal{T}(\mathcal{F}^{\leq k} \cup \mathcal{Q}_{\mathcal{P}} \cup \{E\}),\ \forall i \in \{1, \ldots, m\}, \forall j \in \{1, \ldots, \mathsf{Card}(\mathcal{P}\mathsf{os}(r, x_i))\}, \lfloor x_{i,j}\overline{\sigma} \odot a\rfloor = \lfloor x_i\overline{\tau} \odot a\rfloor\}$. Note that L and R are two recognizable sets of ground terms. We define the $\mathcal{G}(\bar{l}, r, \overline{\tau})$ over $\mathcal{F}^{\leq k} \cup \mathcal{Q}_{\mathcal{P}} \cup \{E\}$ by: $\mathcal{G}(\bar{l}, r, \overline{\tau}) := \{l \to r \mid l \in L, r \in R\}$.

▶ **Definition 5.14.** Let $\mathcal{G}_{\mathcal{R}} = \{\mathcal{G}(\bar{l}, r, \overline{\tau}) \mid l \to r \in \mathcal{R}, \bar{l} \in \mathcal{T}(\mathcal{F}^{\leq k} \cup \{E\}, \mathcal{V}), \overline{\tau} \in \Lambda_1\}$, $\mathcal{G}_{\mathcal{E}} := \{\mathcal{G}(l, r, \overline{\tau}) \mid l \to r \in \mathcal{E}, \overline{\tau} \in \Lambda_0\}$. We define $\mathcal{G}$ as the GRS over $\mathcal{F}^{\leq k} \cup \{E\} \cup \mathcal{Q}_{\mathcal{P}}$ $\mathcal{G} := \mathcal{G}_{\mathcal{R}} \cup \mathcal{G}_{\mathcal{E}} \cup \Gamma_{\mathcal{P}(\mathcal{A})^{+\leq k}}$.

The transitive and reflexive closure of a GRS is recognizable by a GTT. The GTT recognizing $\to_{\mathcal{G}}^*$ will be used to simulate $\mathsf{bo}(k)$-derivations in $\mathcal{R} \cup \mathcal{E}$ (see lemma 5.22).

Lifting Lemma

The lifting lemma simulates a derivation $\overline{s} \to_{\mathcal{A}_{\mathcal{P}}^+}^* \overline{s'} \to_{\mathcal{G}} \bar{t}$ by a $\mathsf{bo}(k)$-derivation in $\mathcal{R} \cup \mathcal{E}$ followed by a derivation in ${\mathcal{A}_{\mathcal{P}}}^+$. The proof can be found in the long version of this article.

▶ **Lemma 5.15** (lifting lemma). *Let $l \to r \in \mathcal{R} \cup \mathcal{E}$, $\overline{l} \in \mathcal{T}(\mathcal{F}^{\leq k} \cup \mathcal{Q}_{\mathcal{P}} \cup \{E\})$, let $a = 0$ if $l \to r \in \mathcal{E}$ and $a = 1$ if $l \to r \in \mathcal{R}$, and let $\overline{\sigma} \in \Lambda_a$ be a substitution. Let $\overline{s} \in \mathcal{T}(\mathcal{F}^{\mathbb{N}} \cup \{E\})$, $\overline{s'}, \overline{t} \in \mathcal{T}(\mathcal{F}^{\leq k} \cup \mathcal{Q}_{\mathcal{P}} \cup \{E\})$ be such that $\overline{s} \to^*_{\mathcal{A}^+_{\mathcal{P}}} \overline{s'} \to_{\mathcal{G}(\overline{l},r,\overline{\sigma})} \overline{t}$. There exists $\overline{t'} \in \mathcal{T}(\mathcal{F}^{\mathbb{N}}, \cup\{E\})$ such that $\overline{s}\ {}_k\circ\!\!\to^+_{\mathcal{R}\cup\mathcal{E}} \overline{t'} \to^*_{\mathcal{A}^+_{\mathcal{P}}} \overline{t}$.*

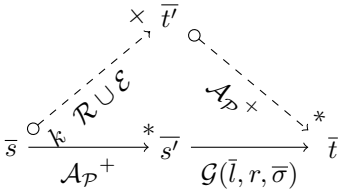


Figure 1 Lifting One Step.

▶ **Example 5.16.** Let $\mathcal{R}_1$ and $\mathcal{A}_1$ be the TRS and the automaton of example 2.1 and let $k = 1$. Let $\overline{s} = \mathsf{f}^0(E(\mathsf{h}^0(\mathsf{a}^1), E(\mathsf{a}^0, \mathsf{a}^1))) \to_{\mathcal{A}_{\mathcal{P}}{}^+} \overline{s'} = \mathsf{f}^0(E(\mathsf{h}^0(\{\mathsf{q}_\mathsf{a}\}), E(\mathsf{a}^0, \mathsf{a}^1)))$, let $x\overline{\sigma} = E(\mathsf{h}^0(\{\mathsf{q}_\mathsf{a}\}), E(\mathsf{a}^0, \mathsf{a}^1))$, and let $\overline{\tau} = \lfloor \overline{\sigma} \rfloor$. Let $\overline{s'} \to_{\mathcal{G}(\mathsf{f}^0(x),\mathsf{g}^0(x,x),\overline{\tau})} \overline{t} = \mathsf{g}^0(E(E(\mathsf{h}^1(\{\mathsf{q}_\mathsf{a}\}), \mathsf{h}^1(\{\mathsf{q}_\mathsf{a}\})), \mathsf{a}^1), E(\mathsf{h}^1(\{\mathsf{q}_\mathsf{a}\}), E(\mathsf{a}^1, \mathsf{a}^1)))$ (this step holds since $\lfloor E(E(\mathsf{h}^1(\{\mathsf{q}_\mathsf{a}\}), \mathsf{h}^1(\{\mathsf{q}_\mathsf{a}\})), \mathsf{a}^1) \rfloor = \lfloor E(\mathsf{h}^1(\{\mathsf{q}_\mathsf{a}\}), E(\mathsf{a}^1, \mathsf{a}^1)) \rfloor = \lfloor x\overline{\tau} \odot a \rfloor$ $= E(\mathsf{h}^1(\{\mathsf{q}_\mathsf{a}\}), \mathsf{a}^1)$). We want to find a term $\overline{t'}$ such that $\overline{s}\ {}_1\circ\!\!\to^*_{\mathcal{R}\cup\mathcal{E}} \overline{t'} \to^*_{\mathcal{A}^+_{\mathcal{P}}} \overline{t}$. First, we apply the rule $\mathsf{f}(x) \to \mathsf{g}(x, x)$ which gives the $\mathsf{bo}(1)$-step $\overline{s'}\ {}_1\circ\!\!\to_{\mathcal{R}} \overline{t'} = \mathsf{g}^0(E(\mathsf{h}^1(\mathsf{a}^2), E(\mathsf{a}^1, \mathsf{a}^1)), E(\mathsf{h}^1(\mathsf{a}^2), E(\mathsf{a}^1, \mathsf{a}^1)))$. Then, since $\lfloor E(\mathsf{h}^1(\mathsf{a}^2), E(\mathsf{a}^1, \mathsf{a}^1)) \rfloor = x\lfloor \overline{\tau} \odot 1 \rfloor$, using the comb form proposition 5.10 we obtain a derivation $\overline{t'}\ {}_1\circ\!\!\to^*_{\mathcal{E}} \mathsf{g}^0(E(E(\mathsf{h}^1(\mathsf{a}^2), \mathsf{h}^1(\mathsf{a}^2)), \mathsf{a}^1), E(\mathsf{h}^1(\mathsf{a}^2), E(\mathsf{a}^1, \mathsf{a}^1))) \to^*_{\mathcal{A}_{\mathcal{P}}{}^+} \overline{t}$.

▶ **Corollary 5.17** (lifting n-steps). *Let $\overline{s} \in \mathcal{T}(\mathcal{F}^{\mathbb{N}} \cup \mathcal{Q}_{\mathcal{P}} \cup \{E\})$, $\overline{s'}, \overline{t} \in \mathcal{T}(\mathcal{F}^{\leq k} \cup \mathcal{Q}_{\mathcal{P}} \cup \{E\})$ be such that $\overline{s} \to^*_{\mathcal{G}} \overline{t}$. There exists $\overline{t'} \in \mathcal{T}(\mathcal{F}^{\mathbb{N}} \cup \mathcal{Q}_{\mathcal{P}} \cup \{E\})$ such that $\overline{s}\ {}_k\circ\!\!\to^*_{\mathcal{R}\cup\mathcal{E}} \overline{t'} \to^*_{\mathcal{A}^+_{\mathcal{P}}} \overline{t}$.*

Projecting Lemma

The projecting lemma simulates one $\mathsf{bo}(k)$-step $\overline{s}\ {}_k\circ\!\!\to_{\mathcal{R}\cup\mathcal{E}} \overline{t}$ by a derivation in $\mathcal{G}$ from $\mathsf{Top}(\overline{s})$ to $\mathsf{Top}(\overline{t})$. The full proof is given in the long version of this paper.

▶ **Lemma 5.18** (Projecting one step). *Let $\overline{s}$, $\overline{t} \in \mathcal{T}(\mathcal{F}^{\mathbb{N}} \cup \mathcal{Q}_{\mathcal{P}} \cup \{E\})$, be such that $\overline{s}$ is well-marked and $\overline{s} = \overline{s}[\overline{l}\overline{\sigma}]_v\ {}_k\circ\!\!\to_{\mathcal{R}\cup\mathcal{E}} \overline{t} = \overline{s}[r(\overline{\sigma} \odot j)]_v$.*

1. *If $\forall u \prec v$, $\mathsf{m}(\overline{s}/u) \leq k$ then there exist a term $\overline{s'} \in \mathcal{T}(\mathcal{F}^{\leq k} \cup \mathcal{Q}_{\mathcal{P}} \cup \{E\})$, a substitution $\overline{\sigma'} : \mathcal{V} \to \mathcal{C}_{\leq k+2}$ such that $\mathsf{Top}(\overline{s}) \to^*_{\mathcal{A}_{\mathcal{P}}{}^{\leq k}} \overline{s'} \to_{\mathcal{G}(\overline{l},r,\overline{\sigma'})} \mathsf{Top}(\overline{t})$,*
2. *otherwise, $\mathsf{Top}(\overline{s}) \to^*_{\mathcal{A}^+_{\mathcal{P}}{}^{\leq k}} \mathsf{Top}(\overline{t})$.*

▶ **Example 5.19.** Let us consider the TRS $\mathcal{R}_1$, and the automaton $\mathcal{A}_1$ of example 2.1, and $k = 1$. We have the following derivation between these two well-marked terms $\overline{s} =$ $\mathsf{f}^1(E(\mathsf{a}^0, E(\mathsf{a}^1, E(\mathsf{b}^1, \mathsf{h}^0(\mathsf{h}^0(\mathsf{a}^1))))))) \to_{\mathsf{f}(x)\to\mathsf{g}(x,x)} \overline{t} =$ $\mathsf{g}(E(\mathsf{a}^1, E(\mathsf{a}^1, E(\mathsf{b}^1, \mathsf{h}^1(\mathsf{h}^2(\mathsf{a}^3))))))), E(\mathsf{a}^1, E(\mathsf{a}^1, E(\mathsf{b}^1, \mathsf{h}^1(\mathsf{h}^2(\mathsf{a}^3))))))$. Let $x\overline{\sigma} = E(\mathsf{a}^0, E(\mathsf{a}^1, E(\mathsf{b}^1, \mathsf{h}^0(\mathsf{h}^0((\mathsf{a}^1))))))$. We have $\overline{s} = \mathsf{f}^1(x\overline{\sigma})$, $\overline{t} = \mathsf{g}(x\overline{\sigma} \odot 1, x\overline{\sigma} \odot 1)$, $\mathsf{Top}(\overline{s}) = \overline{s}$ and $\mathsf{Top}(\overline{t}) = \mathsf{g}(E(\mathsf{a}^1, E(\mathsf{a}^1, E(\mathsf{b}^1, \mathsf{h}^1(\{\mathsf{q}_\perp\}))))), E(\mathsf{a}^1, E(\mathsf{a}^1, E(\mathsf{b}^1, \mathsf{h}^1(\{\mathsf{q}_\perp\})))))$. First, we cut the "useless" part of $\overline{s}$ using $\mathcal{A}_{1\mathcal{P}}{}^{\leq 1}$ i.e. the part of $x\overline{\sigma}$ that is marked by an integer greater that 1 in $x\overline{\sigma} \odot 1 = E(\mathsf{a}^1, E(\mathsf{a}^1, E(\mathsf{b}^1, \mathsf{h}^1(\mathsf{h}^2(\mathsf{a}^3))))))$. We obtain the following

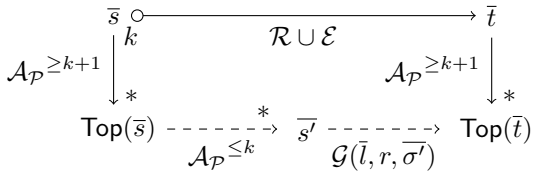


Figure 2 Projecting One Step, case 1.

derivation $\mathsf{Top}(\overline{s}) \to^*_{\mathcal{A}_{1\mathcal{P}}^{\leq k}} \mathsf{f}^1(E(\mathsf{a}^0, E(\mathsf{a}^1, E(\mathsf{b}^1, \mathsf{h}^0(\{\mathsf{q}_\perp\})))))$. We are now ready to apply the step of the GRS that simulates the rule $l \to r$. Let $x\overline{\sigma'} = E(\mathsf{a}^0, E(\mathsf{a}^1, E(\mathsf{b}^1, \mathsf{h}^1(\{\mathsf{q}_\perp\}))))$. Let $\overline{\tau} = \lfloor \overline{\sigma'} \rfloor$ and $\overline{s'} = \mathsf{f}(x\overline{\sigma'})$. The comb associated to $x\overline{\sigma'}$ is $x\overline{\tau} = E(E(E(\mathsf{a}^0, \mathsf{a}^1), \mathsf{b}^1), \mathsf{h}^1(\{\mathsf{q}_\perp\}))$. Moreover, the comb associated to $x\overline{\sigma'} \odot 1 = E(\mathsf{a}^1, E(\mathsf{a}^1, E(\mathsf{b}^1, \mathsf{h}^1(\{\mathsf{q}_\perp\}))))$ is $\lfloor x\overline{\tau} \odot 1 \rfloor = E(E(\mathsf{a}^1, \mathsf{b}^1), \mathsf{h}^1(\{\mathsf{q}_\perp\}))$. By definition, it means that $\overline{l\sigma'} \to r(\overline{\sigma'} \odot 1) \in \mathcal{G}(\bar{l}, r, \overline{\tau})$. Hence, we obtain the derivation $\mathsf{Top}(\overline{s}) \to_{\mathcal{A}_{1\mathcal{P}}^{\leq 1}} \overline{s'} = \mathsf{f}^1(x\overline{\sigma'}) \to_{\mathcal{G}(\bar{l}, r, \lfloor x\overline{\sigma'} \rfloor)} \mathsf{g}(x\overline{\sigma'} \odot 1, x\overline{\sigma'} \odot 1) = \mathsf{Top}(\overline{t})$.

▶ **Corollary 5.20** (Projecting n -steps). *Let $\overline{s}$, $\overline{t} \in \mathcal{T}(\mathcal{F}^\mathbb{N} \cup \mathcal{Q}_\mathcal{P} \cup \{E\})$, be such that $\overline{s}$ is well-marked and $\overline{s}\ {}_k\circ\!\to^*_{\mathcal{R}\cup\mathcal{E}} \overline{t}$. We have $\mathsf{Top}(\overline{s}) \to^*_\mathcal{G} \mathsf{Top}(\overline{t})$.*

▶ **Lemma 5.21.** *Let $s \in \mathcal{T}(\mathcal{F})$, $q \in \mathcal{Q}_\mathcal{A}$. We have $\exists \overline{t} \in \mathcal{T}(\mathcal{F}^\mathbb{N})$, $s\ {}_k\circ\!\to^*_{\mathcal{R}\cup\mathcal{E}} \overline{t} \to^*_{\overline{\mathcal{A}}} q$ iff $s \to^*_\mathcal{G} \{q\}$.*

Inverse Recognizability Preservation

▶ **Lemma 5.22** (simulation lemma). *We have $(\to^*_\mathcal{G})[\mathcal{Q}_{f,\mathcal{P}}] \cap \mathcal{T}(\mathcal{F}) = (\ {}_k\!\to^*_\mathcal{R})[T]$.*

Proof. Let $s \in (\ {}_k\!\to^*_\mathcal{R})[T]$. By definition, there exist $t \in \mathcal{T}(\mathcal{F})$ and $q \in \mathcal{Q}_{f,\mathcal{A}}$ such that $s\ {}_k\!\to^*_\mathcal{R} t \to^*_\mathcal{A} q$. By definition of a $\mathsf{boc}(k)$ derivation, there exists a marked term $\overline{t}$ such that $s\ {}_k\circ\!\to^*_{\mathcal{R}\cup\mathcal{E}} \overline{t}$. By lemma 4.3, since $t \to^*_\mathcal{A} q$, we have $\overline{t} \to^*_{\overline{\mathcal{A}}} q$. By lemma 5.21, $s \to^*_\mathcal{G} \{q\}$, and since $\{q\} \in \mathcal{Q}_{f,\mathcal{P}}$, we have $s \in (\to^*_\mathcal{G})[\mathcal{Q}_{f,\mathcal{P}}]$. Hence, $(\ {}_k\!\to^*_\mathcal{R})[T] \subseteq (\to^*_\mathcal{G})[\mathcal{Q}_{f,\mathcal{P}}] \cap \mathcal{T}(\mathcal{F})$.
Let $s \in (\to^*_\mathcal{G})[\mathcal{Q}_{f,\mathcal{P}}] \cap \mathcal{T}(\mathcal{F})$. There exists $q \in \mathcal{Q}_{f,\mathcal{A}}$ such that $s \to^*_\mathcal{G} \{q\}$. By lemma 5.21, there exists $\overline{t} \in \mathcal{T}(\mathcal{F}^\mathbb{N})$ such that $s\ {}_k\circ\!\to^*_{\mathcal{R}\cup\mathcal{E}} \overline{t} \to^*_{\overline{\mathcal{A}}} q$. By proposition 3.2, $s \to^*_\mathcal{R} t$, and since there exists a $\mathsf{bo}(k)$ marked derivation from s to $\overline{t}$, $s\ {}_k\!\to^*_\mathcal{R} t$. By lemma 4.3, since $\overline{t} \to^*_{\overline{\mathcal{A}}} q$, we have $t \to^*_\mathcal{A} q$. So, $t \in T$, and $s \in (\ {}_k\!\to^*_\mathcal{R})[T]$. ◀

We are now ready to prove theorem 5.1.

▶ **Theorem 5.1.** *Let $\mathcal{R}$ be some (finite) left-linear* TRS *over a signature $\mathcal{F}$. Let T be some recognizable subset of $\mathcal{T}(\mathcal{F})$ and let $k > 0$. Then, the set $(\ {}_k\!\to^*_\mathcal{R})[T]$ is recognizable too.*

By lemma 5.22, $(\to^*_\mathcal{G})[\mathcal{Q}_{f,\mathcal{P}}] \cap \mathcal{T}(\mathcal{F}) = (\ {}_k\!\to^*_\mathcal{R})[T]$. The relation $\to^*_\mathcal{G}$ is recognizable by a GTT, and since GTTs are inverse recognizability preserving (see e.g. chapter 3.2 of [2]), $(\to^*_\mathcal{G})[\mathcal{Q}_{f,\mathcal{P}}] \cap \mathcal{T}(\mathcal{F})$ is recognizable, and thus $(\ {}_k\!\to^*_\mathcal{R})[T]$ is recognizable. To effectively build the set $(\ {}_k\!\to^*_\mathcal{R})[T]$, we need to construct the automaton $\mathcal{A}_\mathcal{P}^{\leq k}$ and the GTT $(\to^*_\mathcal{G})$. Since GTTs are effectively inverse recognizability preserving, the result holds.

▶ **Corollary 5.23.** *Every* BO(k) TRS *are effectively inverse recognizability preserving.*

6 Strongly Bounded TRSs

We introduce here strongly bounded TRSs. The reader may refer to the long version of the article for more details.

▶ **Definition 6.1.** A marked step $\overline{s} = \overline{s}[\overline{l}\overline{\sigma}]_v \circ\!\!\rightarrow_{\mathcal{R}} \overline{t} = \overline{s}[r(\overline{\sigma} \odot j)]_v$ is *weakly bottom-up* (wbu for short) if $l \to r \in \mathcal{E}$ or if $l \to r \in \mathcal{R}$ and the following assertion holds: $(l \notin \mathcal{V} \Rightarrow \mathsf{m}(\overline{l}) = 0)$ and $(l \in \mathcal{V} \Rightarrow \sup(\{\mathsf{m}(\overline{s}/u) \mid u \prec v) = 0)$. A marked derivation is wbu if all its rewriting steps are wbu. A derivation $s \to^*_{\mathcal{R}\cup\mathcal{E}} t$ is wbu if the associated marked derivation is wbu. A derivation $s \to^*_{\mathcal{R}} t$ is *weakly bottom-up convertible* (wbuc for short) if there exists a wbu derivation $s \to^*_{\mathcal{R}\cup\mathcal{E}} t$. Let $k \in \mathbb{N}$. A TRS is strongly k-bounded ($\mathsf{SBO}(k)$ for short) if every wbu derivation starting on a term $s \in \mathcal{T}(\mathcal{F})$ is $\mathsf{bo}(k)$. We denote by $\mathsf{SBO}(k)$ the class of $\mathsf{SBO}(k)$ TRSs. Finally, the class of strongly bounded TRS SBO is defined by: $\mathsf{SBO} = \bigcup_{k\in\mathbb{N}} \mathsf{SBO}(k)$.

Note that every marked derivation in $\mathcal{E}$ is wbu. Roughly speaking, a wbu derivation is a derivation in which the rules of $\mathcal{R}$ are applied going from the bottom to the top. Moreover, every derivation is wbuc and $\mathsf{SBO}(k) \subset \mathsf{BO}(k)$. The class SBO contains inverse right-linear finite-path overlapping TRSs [20], and left-linear growing TRSs [16]. Moreover, the membership problem for the class of $\mathsf{SBO}(k)$ TRSs such that $\mathsf{LHS}(\mathcal{R}) \cap \mathcal{V} = \emptyset$ is decidable, whereas the membership problem for $\mathsf{BO}(0)$ is undecidable, as shown in [5].

7 Perspectives

Here are some natural perspectives of development for this work.

- The method developed here also might be used for testing some termination properties and might lead to a proof of the decidability of the termination of left-linear growing TRSs as conjectured in [16].
- A dual notion of *top-down* rewriting should be defined (at least for linear TRSs). The class would presumably extend the class of layered transducing TRSs defined in [18].
- The TRSs considered in [9] and the TRSs considered here might be treated in a unifed manner for the linear case and if so, might be extended to the left-linear case.

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 B. Buchberger. Basic features and development of the critical-pair/completion procedure. In *Proceedings of the 1st International Conference on Rewriting Techniques and Applications*, pages 1–45, 1985.

2 H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available at: `http://www.grappa.univ-lille3.fr/tata`, 2007. release October, 12th 2007.

3 Evelyne Contejean, Claude Marché, and Xavier Urbain. CiME, 2004. Available at `http://cime.lri.fr/`.

4 I. Durand and A. Middeldorp. Decidable call-by-need computations in term rewriting. *Inf. Comput.*, 196(2):95–126, 2005.

5 I. Durand and G. Sénizergues. Bottom-up rewriting for words and terms, March 2009. Available at: `http://arXiv.org/abs/0903.2554`.

6 I. Durand, G. Sénizergues, and M. Sylvestre. Termination of linear bounded term rewriting systems. In Christopher Lynch, editor, *Proceedings of the 21st International Conference on Rewriting Techniques and Applications*, pages 341–356. LIPIcs, July 2010.

7 Guillaume Feuillade, Thomas Genet, and Valérie Viet Triem Tong. Reachability Analysis over Term Rewriting Systems. Research Report RR-4970, INRIA, 2003.

8 Z. Fülöp, E. Jurvanen, M. Steinby, and S. Vágvölgyi. On one-pass term rewriting. In *MFCS*, pages 248–256, 1998.

9 A. Geser, D. Hofbauer, and J. Waldmann. Match-bounded string rewriting systems. *Journal Applicable Algebra in Engineering, Communication and Computing*, 15(3-4):149–171, November, 2004.

10 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.

11 J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Automated Termination Proofs with AProVE (system description). In *Proceedings of the 15th International Conference on Rewriting Techniques and Applications*, pages 210–220, 2004.

12 F. Jacquemard. Decidable approximations of term rewriting systems. In *Proceedings of the 7th International Conference on Rewriting Techniques and Applications*, volume 1103 of *LNCS*, pages 362–376, 1996.

13 J.W. Klop. Term rewriting systems. In *Handbook of Logic in Computer Science, Vol. 2*, pages 1–116. Oxford University Press, 1992.

14 D. Knuth and P. Bendix. Simple word problems in universal algebras. In Leech, editor, *Computational problems in abstract algebra*, pages 263–297. Pergamon Press, 1970.

15 P. Lescanne, T. Heuillard, M. Dauchet, and S. Tison. Decidability of the confluence of ground term rewriting systems. Research Report RR-0675, INRIA, 1987.

16 T. Nagaya and Y. Toyama. Decidability for left-linear growing term rewriting systems. In *Proceedings of the 10th International Conference on Rewriting Techniques and Applications*, pages 256–270, London, UK, 1999. Springer-Verlag.

17 P. Réty and J. Vuotto. Tree automata for rewrite strategies. *J. Symb. Comput.*, 40(1):749–794, 2005.

18 H. Seki, T. Takai, Y. Fujinaka, and Y. Kaji. Layered transducing term rewriting system and its recognizability preserving property. In *Proceedings of the 13th International Conference on Rewriting Techniques and Applications*, volume 2378 of *LNCS*. Springer Verlag, 2002.

19 F. Seynhaeve, S. Tison, and M. Tommasi. Homomorphisms and concurrent term rewriting. In *FCT*, pages 475–487, 1999.

20 T. Takai, Y. Kaji, and H. Seki. Right-linear finite path overlapping term rewriting systems effectively preserve recognizability. In *Proceedings of the 11th International Conference on Rewriting Techniques and Applications*, pages 246–260, 2000.

21 T. Takai, Y. Kaji, and H. Seki. Termination property of inverse finite path overlapping term rewriting system is decidable. *IEICE transactions on information and systems*, 85(3):487–496, 2002-03-01.

22 T. Takai, Y. Kaji, and H. Seki. Right-linear finite-path overlapping term rewriting systems effectively preserve recognizability. *Scienticae Mathematicae Japonicae*, 2006. (to appear, preliminary version: IEICE Technical Report COMP98-45).

23 Terese. *Term Rewriting Systems by Terese*, volume 55. Cambridge University Press, 2003.

24 J. Waldmann. Matchbox: A Tool for Match-Bounded String Rewriting (system description). In *Proceedings of the 15th International Conference on Rewriting Techniques and Applications*, pages 85–94, 2004.

25 H. Zantema. TORPA: Termination of Rewriting Proved Automatically (system description). In *Proceedings of the 15th International Conference on Rewriting Techniques and Applications*, pages 95–104, 2004.

Labelings for Decreasing Diagrams*

Harald Zankl, Bertram Felgenhauer, and Aart Middeldorp

Institute of Computer Science, University of Innsbruck, 6020 Innsbruck, Austria
{harald.zankl,bertram.felgenhauer,aart.middeldorp}@uibk.ac.at

Abstract

This paper is concerned with automating the decreasing diagrams technique of van Oostrom for establishing confluence of term rewrite systems. We study abstract criteria that allow to lexicographically combine labelings to show local diagrams decreasing. This approach has two immediate benefits. First, it allows to use labelings for linear rewrite systems also for left-linear ones, provided some mild conditions are satisfied. Second, it admits an incremental method for proving confluence which subsumes recent developments in automating decreasing diagrams. The techniques proposed in the paper have been implemented and experimental results demonstrate how, e.g., the rule labeling benefits from our contributions.

1998 ACM Subject Classification F.3.1 Specifying and Verifying and Reasoning about Programs, F.4.2 Grammars and Other Rewriting Systems

Keywords and phrases term rewriting, confluence, decreasing diagrams, labeling

Digital Object Identifier 10.4230/LIPIcs.RTA.2011.377

Category Regular Research Paper

1 Introduction

The decreasing diagrams technique of van Oostrom [10] is a powerful method for showing confluence of abstract rewrite systems, i.e., it is complete for countable systems. The main idea of the approach is to show confluence by establishing local confluence under the side condition that rewrite steps of the joining sequences must *decrease* with respect to some well-founded order. For term rewrite systems however, the main problem for automation of decreasing diagrams is that in general infinitely many local peaks must be considered. To reduce this problem to a finite set of local peaks one can label rewrite steps with functions that satisfy special properties. In [12] van Oostrom presented the rule labeling that allows to conclude confluence of *linear* rewrite systems by checking decreasingness of the critical peaks (those emerging from critical overlaps). The rule labeling has recently been implemented by Aoto [1] and Hirokawa and Middeldorp [8]. Already in [12] van Oostrom presented constraints that allow to apply the rule labeling to *left-linear* systems. This approach has recently been implemented and extended by Aoto [1]. Our framework subsumes the above ideas.

The contributions of this paper comprise the extraction of abstract constraints on a labeling such that for a (left-)linear rewrite system decreasingness of the critical peaks ensures confluence. We show that the rule labeling adheres to our constraints and present additional labeling functions. Furthermore such labeling functions can be combined lexicographically to obtain new labeling functions satisfying our constraints. This approach allows the formulation of an abstract criterion that makes virtually every labeling function for linear rewrite systems also applicable to left-linear systems. Consequently, confluence of the TRS in Example 1.1

* This research is supported by FWF (Austrian Science Fund) project P22467.

22nd International Conference on Rewriting Techniques and Applications (RTA'11).
Editor: M. Schmidt-Schauß; pp. 377–392

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

can be established automatically, e.g., by the rule labeling, while current approaches based on the decreasing diagrams technique [1, 8] as well as standard confluence criteria fail.

▶ **Example 1.1.** Consider the TRS $\mathcal{R}$ (from [14]) consisting of the rules

$$\begin{array}{lll}
1\colon x + (y + z) \to (x + y) + z & 5\colon x + y \to y + x & 7\colon \mathsf{s}(x) + y \to x + \mathsf{s}(y) \\
2\colon (x + y) + z \to x + (y + z) & 6\colon x \times y \to y \times x & 8\colon x + \mathsf{s}(y) \to \mathsf{s}(x) + y \\
3\colon \quad \mathsf{sq}(x) \to x \times x & & 9\colon x \times \mathsf{s}(y) \to x + (x \times y) \\
4\colon \quad \mathsf{sq}(\mathsf{s}(x)) \to (x \times x) + \mathsf{s}(x + x) & & 10\colon \mathsf{s}(x) \times y \to (x \times y) + y
\end{array}$$

This system is locally confluent since all its 34 critical pairs are joinable.

The remainder of this paper is organized as follows. After recalling preliminaries in Section 2 we present constraints (on a labeling) such that decreasingness of the critical peaks ensures confluence for (left-)linear rewrite systems in Section 3. The merits of this approach are assessed in Section 4. Implementation issues are addressed in Section 5 before Section 6 gives an empirical evaluation of our results. Section 7 concludes.

2 Preliminaries

We assume familiarity with term rewriting [4, 15].

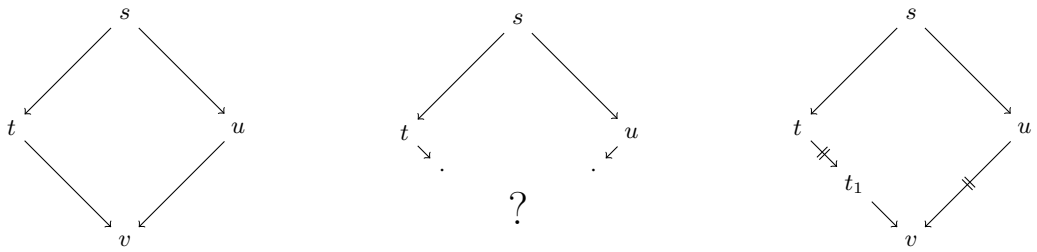
Let $\mathcal{F}$ be a signature and let $\mathcal{V}$ be a set of variables disjoint from $\mathcal{F}$. By $\mathcal{T}(\mathcal{F}, \mathcal{V})$ we denote the set of terms over $\mathcal{F}$ and $\mathcal{V}$. The expression $|t|_x$ indicates how often variable x occurs in term t. The *set of positions* of a term t is defined as $\mathcal{P}\mathsf{os}(t) = \{\epsilon\}$ if t is a variable and as $\mathcal{P}\mathsf{os}(t) = \{\epsilon\} \cup \{iq \mid q \in \mathcal{P}\mathsf{os}(t_i)\}$ if $t = f(t_1, \ldots, t_n)$. We write $p \leqslant q$ if $q = pp'$ for some position p', in which case $q \backslash p$ is defined to be p'. Furthermore $p < q$ if $p \leqslant q$ and $p \neq q$. Finally, $p \parallel q$ if neither $p \leqslant q$ nor $q < p$. Positions are used to address occurrences of subterms. The subterm of t at position $p \in \mathcal{P}\mathsf{os}(t)$ is defined as $t|_p = t$ if $p = \epsilon$ and as $t|_p = t_i|_q$ if $p = iq$. We write $s[t]_p$ for the result of replacing $s|_p$ with t in s. The set of function symbol positions $\mathcal{P}\mathsf{os}_{\mathcal{F}}(t)$ is $\{p \in \mathcal{P}\mathsf{os}(t) \mid t|_p \notin \mathcal{V}\}$ and $\mathcal{P}\mathsf{os}_{\mathcal{V}}(t) = \mathcal{P}\mathsf{os}(t) \setminus \mathcal{P}\mathsf{os}_{\mathcal{F}}(t)$.

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 rewrite rule $l \to r$ is duplicating if $|l|_x < |r|_x$ for some $x \in \mathcal{V}$. A term rewrite system (TRS) is a signature together with a finite set of rewrite rules over this signature. In the sequel signatures are implicit. By $\mathcal{R}_{\mathsf{d}}$ and $\mathcal{R}_{\mathsf{nd}}$ we denote the duplicating and non-duplicating rules of a TRS $\mathcal{R}$, 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 $\to^{=}$ ($\to^*$) denotes the reflexive (reflexive and transitive) closure of $\to$ and $\twoheadrightarrow$ denotes rewriting at parallel positions.

A relative TRS $\mathcal{R}/\mathcal{S}$ is a pair of TRSs $\mathcal{R}$ and $\mathcal{S}$ with the induced rewrite relation $\to_{\mathcal{R}/\mathcal{S}} = \to_{\mathcal{S}}^* \cdot \to_{\mathcal{R}} \cdot \to_{\mathcal{S}}^*$. Sometimes we identify a TRS $\mathcal{R}$ with the relative TRS $\mathcal{R}/\varnothing$ and vice versa. A TRS $\mathcal{R}$ (relative TRS $\mathcal{R}/\mathcal{S}$) is terminating if $\to_{\mathcal{R}}$ ($\to_{\mathcal{R}/\mathcal{S}}$) is well-founded. Two relations $\geqslant$ and $>$ are called compatible if $\geqslant \cdot > \cdot \geqslant \subseteq >$. A monotone reduction pair $(\geqslant, >)$ consists of a quasi-order $\geqslant$ and a well-founded order $>$ such that $\geqslant$ and $>$ are compatible and closed under contexts and substitutions. We recall how to prove relative termination incrementally according to Geser [6]:

▶ **Theorem 2.1.** *A relative TRS $\mathcal{R}/\mathcal{S}$ is terminating if $\mathcal{R} = \varnothing$ or there exists a monotone reduction pair $(\geqslant, >)$ such that $\mathcal{R} \cup \mathcal{S} \subseteq \geqslant$ and $(\mathcal{R} \setminus >)/(\mathcal{S} \setminus >)$ is terminating.* ◀

An overlap $(l_1 \to r_1, p, l_2 \to r_2)_\mu$ of a TRS $\mathcal{R}$ consists of variants $l_1 \to r_1$ and $l_2 \to r_2$ of rewrite rules of $\mathcal{R}$ without common variables, a position $p \in \mathcal{P}\mathsf{os}_{\mathcal{F}}(l_2)$, and a most general



(a) (parallel) **(b)** (critical overlap) **(c)** (variable overlap)

Figure 1 Three kinds of local peaks.

unifier μ of l_1 and $l_2|_p$. If $p = \epsilon$ then we require that $l_1 \to r_1$ and $l_2 \to r_2$ are not variants. From an overlap $(l_1 \to r_1, p, l_2 \to r_2)_\mu$ we obtain a critical peak $l_2\mu[r_1\mu]_p \leftarrow l_2\mu \to r_2\mu$ and a critical pair $l_2\mu[r_1\mu]_p \leftarrow\!\!\to r_2\mu$.

We write $\langle A, \{\to_\alpha\}_{\alpha\in I}\rangle$ to denote the ARS $\langle A, \to\rangle$ where $\to$ is the union of $\to_\alpha$ for all $\alpha \in I$. Let $\langle A, \{\to_\alpha\}_{\alpha\in I}\rangle$ be an ARS and let $>$ be a relation on I. We write $\overset{\vee}{\to}_{\alpha_1\ldots\alpha_n}$ for the union of $\to_\beta$ where $\beta < \alpha_i$ for some $1 \leqslant i \leqslant n$. We say $\to_\alpha$ and $\to_\beta$ are *extended locally decreasing* (with respect to $\geqslant$ and $>$) if ${}_\alpha\!\leftarrow \cdot \to_\beta \subseteq \overset{\vee}{\to}{}^*_\alpha \cdot \overset{\mathbb{V}}{\to}{}^=_\beta \cdot \overset{\vee}{\to}{}^*_{\alpha\beta} \cdot {}^*_{\alpha\beta}\overset{\vee}{\leftarrow} \cdot {}^=_\alpha\overset{\mathbb{V}}{\leftarrow} \cdot {}^*_\beta\overset{\vee}{\leftarrow}$. An ARS $\langle A, \{\to_\alpha\}_{\alpha\in I}\rangle$ is *extended locally decreasing* if there exists a quasi-order $\geqslant$ and a well-founded order $>$ such that $\geqslant$ and $>$ are compatible and $\to_\alpha$ and $\to_\beta$ are extended locally decreasing for all $\alpha, \beta \in I$ with respect to $\geqslant$ and $>$.

The following theorem is from [8], reformulating a result obtained by van Oostrom [10].

▶ **Theorem 2.2.** *Every extended locally decreasing ARS is confluent.* ◀

3 Confluence by Labeling

In this section we present constraints (on a labeling) such that extended local decreasingness of the critical peaks ensures confluence of linear (Section 3.1) and left-linear (Section 3.2) TRSs. Furthermore, we show that if two labelings satisfy these conditions then also their lexicographic combination satisfies them.

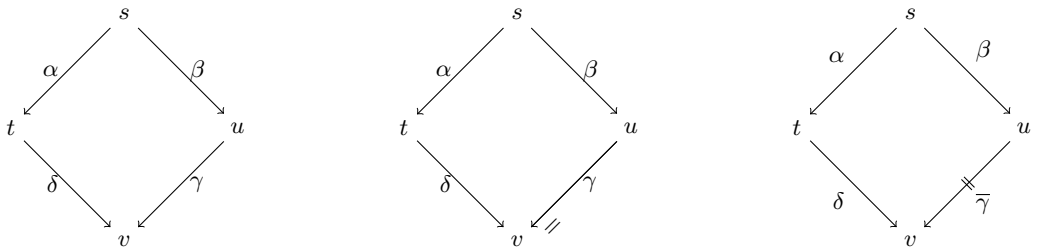
There are three possibilities for a local peak (modulo symmetry):

$$t = s[r_1\sigma]_p \leftarrow s[l_1\sigma]_p = s = s[l_2\sigma]_q \to s[r_2\sigma]_q = u \tag{1}$$

- $p \parallel q$ (parallel)
- $q \leqslant p$ and $p \in \mathcal{P}\mathsf{os}_\mathcal{F}(s[l_2]_q)$ (critical overlap)
- $q < p$ and $p \notin \mathcal{P}\mathsf{os}_\mathcal{F}(s[l_2]_q)$ (variable overlap)

These cases are visualized in Figure 1. Figure 1(a) shows the shape of a local peak where the reductions take place at parallel positions. Here we have $s \to_{p,l_1\to r_1} t$ and $u \to_{p,l_1\to r_1} v$, i.e., the reductions drawn at opposing sides in the diagram are due to the same rules. The question mark in Figure 1(b) conveys that joinability of critical overlaps may depend on auxiliary rules. Variable overlaps (Figure 1(c)) can again be joined by the rules involved in the diverging step. More precisely, if q' is the unique position in $\mathcal{P}\mathsf{os}_\mathcal{V}(l_2)$ such that $qq' \leqslant p$, $x = l_2|_{q'}$, $|l_2|_x = m$, and $|r_2|_x = n$ then we have $t \to^{m-1}_{l_1\to r_1} t_1$, $t_1 \to_{l_2\to r_2} v$, and $u \to^n_{l_1\to r_1} v$.

Labelings are used to compare rewrite steps. In the sequel we denote the set of all rewrite steps for a TRS $\mathcal{R}$ by $\Lambda_\mathcal{R}$ and elements from this set by capital Greek letters Γ and Δ. Furthermore if $\Gamma = s \to_{p,l\to r} t$ then $C(\Gamma\sigma)$ denotes the rewrite step $C[s\sigma] \to_{p'p,l\to r} C[t\sigma]$ for any substitution σ and context C with $C|_{p'} = \Box$.



(a) (parallel) **(b)** (variable-linear) **(c)** (variable-left-linear)

Figure 2 Labeled peaks.

▶ **Definition 3.1.** Let $\mathcal{R}$ be a TRS. A *labeling function* $\ell\colon \Lambda_{\mathcal{R}} \to W$ is a mapping from rewrite steps into some set W. A *labeling* $(\ell, \geqslant, >)$ for $\mathcal{R}$ consists of a labeling function ℓ, a quasi-order $\geqslant$, and a well-founded order $>$ such that $\geqslant$ and $>$ are compatible and for all rewrite steps $\Gamma, \Delta \in \Lambda_{\mathcal{R}}$, contexts C and substitutions σ:

1. $\ell(\Gamma) \geqslant \ell(\Delta)$ implies $\ell(C[\Gamma\sigma]) \geqslant \ell(C[\Delta\sigma])$ and
2. $\ell(\Gamma) > \ell(\Delta)$ implies $\ell(C[\Gamma\sigma]) > \ell(C[\Delta\sigma])$

All labelings we discuss satisfy $> \subseteq \geqslant$ which allows to avoid tedious case distinctions. In the sequel W, $\geqslant$, and $>$ are left implicit when clear from the context and a labeling is identified with the labeling function ℓ. We use the terminology that a labeling ℓ is *monotone* and *stable* if properties 1 and 2 of Definition 3.1 hold. Abstract labels, i.e., labels that are unknown, are represented by lowercase Greek letters α, β, γ, etc. We write $s \xrightarrow{\alpha}_{p,l\to r} t$ (or simply $s \xrightarrow{\alpha} t$ or $s \to_\alpha t$) if $\ell(s \to_{p,l\to r} t) = \alpha$. Often we leave the labeling ℓ implicit and just attach labels to arrows. A local peak $t \leftarrow s \to u$ is called *decreasing for* ℓ if $t \,{}_\alpha\!\leftarrow s \to_\beta u$, and $\to_\alpha$ and $\to_\beta$ are extended locally decreasing with respect to $\geqslant$ and $>$. To employ Theorem 2.2 for TRSs, extended local decreasingness of the ARS $\langle \mathcal{T}(\mathcal{F}, \mathcal{V}), \{\to_w\}_{w\in W}\rangle$ must be shown.

In the sequel we investigate conditions on a labeling such that local peaks according to (parallel) and (variable overlap) are decreasing automatically. This is desirable since in general there are infinitely many local peaks corresponding to these cases (even if the underlying TRS has finitely many rules). There are also infinitely many local peaks according to (critical overlap) in general, but for a finite TRS they are captured by the finitely many overlaps. Still, it is undecidable if they are decreasingly joinable [8].

For later reference, Figure 2 shows labeled peaks for the case (parallel) (Figure 2(a)) and (variable overlap) if the rule $l_2 \to r_2$ in (1) is linear (Figure 2(b)) and left-linear (Figure 2(c)), respectively. In Figure 2(c) the expression $\overline{\gamma}$ means a sequence of labels $\gamma_1, \ldots, \gamma_n$. Since the step from u to v is parallel we can choose any permutation of $\overline{\gamma}$.

3.1 Linear TRSs

The next definition presents sufficient abstract conditions on a labeling such that local peaks according to the cases (parallel) and (variable-linear) in Figure 2 are decreasing. We use the observation that the former can be seen as an instance of the latter to shorten proofs.

▶ **Definition 3.2.** Let ℓ be a labeling for a TRS $\mathcal{R}$. We call ℓ an *L-labeling (for $\mathcal{R}$)* if for local peaks according to (parallel) and (variable-linear) we have $\alpha \geqslant \gamma$ and $\beta \geqslant \delta$ in Figures 2(a) and 2(b), respectively.

The local diagram in Figure 3(a) visualizes the conditions on an L-labeling more succinctly. We call the critical peaks of a TRS $\mathcal{R}$ Φ-*decreasing* if there exists a Φ-labeling ℓ for $\mathcal{R}$ such that the critical peaks of $\mathcal{R}$ are decreasing for ℓ.

The next theorem states that L-labelings may be used to show confluence of linear TRSs.

▶ **Theorem 3.3.** *Let $\mathcal{R}$ be a linear TRS. If the critical peaks of $\mathcal{R}$ are L-decreasing then $\mathcal{R}$ is confluent.*

Proof. By assumption the critical peaks of $\mathcal{R}$ are decreasing for some L-labeling ℓ. We establish confluence of $\mathcal{R}$ by Theorem 2.2, i.e., show extended local decreasingness of the ARS $\langle \mathcal{T}(\mathcal{F},\mathcal{V}), \to_{\mathcal{R}} \rangle$ where rewrite steps are labeled according to ℓ. Since $\mathcal{R}$ is linear, local peaks have the shape (parallel), (variable-linear), or (critical overlap). By definition of an L-labeling the former two are extended locally decreasing. Now consider a local peak according to (critical overlap), i.e., for the peak in (1) we have $q \leqslant p$ and $p \in \mathcal{P}\mathsf{os}_{\mathcal{F}}(s[l_2]_q)$. Let $p' = p \backslash q$. Then $(s|_q)[r_1\sigma]_{p'} = (s[r_1\sigma]_p)|_q \; {}_{p'}\!\leftarrow s|_q \to_\epsilon r_2\sigma$ must be an instance of a critical peak which is decreasing by assumption. By monotonicity and stability of ℓ we obtain extended local decreasingness of the local peak (1). ◀

We recall the rule labeling of van Oostrom [12], parametrized by a mapping $i\colon \mathcal{R} \to \mathbb{N}$.

▶ **Definition 3.4.** Let $\mathcal{R}$ be a TRS. Then $\ell^i_{\mathrm{rl}}(s \to_{p,l\to r} t) = i(l \to r)$.

Often i is left implicit. The rule labeling satisfies the constraints of an L-labeling.

▶ **Lemma 3.5.** *Let $\mathcal{R}$ be a TRS. Then $(\ell^i_{\mathrm{rl}}, \geqslant_{\mathbb{N}}, >_{\mathbb{N}})$ is an L-labeling for $\mathcal{R}$.*

Proof. First we show that $(\ell^i_{\mathrm{rl}}, \geqslant_{\mathbb{N}}, >_{\mathbb{N}})$ is a labeling. The quasi-order $\geqslant_{\mathbb{N}}$ and the well-founded order $>_{\mathbb{N}}$ are compatible. Furthermore $\ell^i_{\mathrm{rl}}(s \to_{p,l\to r} t) = i(l \to r)$ which ensures monotonicity and stability of ℓ^i_{rl}. Hence $(\ell^i_{\mathrm{rl}}, \geqslant_{\mathbb{N}}, >_{\mathbb{N}})$ is a labeling. Next we show the properties demanded in Definition 3.2. For local peaks according to cases (parallel) and (variable-linear) we recall that the steps drawn at opposite sides in the diagram, e.g., the steps labeled with α and γ (β and δ) in Figures 2(a) and 2(b), are due to applications of the same rule. Hence $\alpha = \gamma$ ($\beta = \delta$) in Figures 2(a) and 2(b), which shows the result. ◀

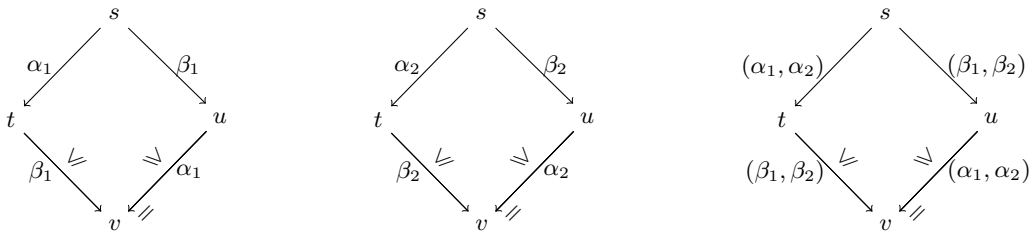
Inspired by [8] we propose a labeling based on relative termination.

▶ **Definition 3.6.** Let $\mathcal{R}$ be a TRS. Then $\ell_{\mathrm{sn}}(s \to t) = s$.

▶ **Lemma 3.7.** *Let $\mathcal{R}$ be a TRS. Then $\ell^{\mathcal{S}}_{\mathrm{sn}} := (\ell_{\mathrm{sn}}, \to^*_{\mathcal{R}}, \to^+_{\mathcal{S}/\mathcal{R}})$ is an L-labeling for $\mathcal{R}$, provided $\to_{\mathcal{S}} \subseteq \to_{\mathcal{R}}$ and $\mathcal{S}/\mathcal{R}$ is terminating.*

Proof. Let $\geqslant \;:= \to^*_{\mathcal{R}}$ and $> \;:= \to^+_{\mathcal{S}/\mathcal{R}}$. First we show that $(\ell_{\mathrm{sn}}, \geqslant, >)$ is a labeling. By definition of relative rewriting, $\to^*_{\mathcal{R}}$ and $\to^+_{\mathcal{S}/\mathcal{R}}$ are compatible and $\to^+_{\mathcal{S}/\mathcal{R}}$ is well-founded by the termination assumption of $\mathcal{S}/\mathcal{R}$. Since rewriting is closed under contexts and substitutions, $\ell^{\mathcal{S}}_{\mathrm{sn}}$ is monotone and stable and hence a labeling. Next we show the properties demanded in Definition 3.2. The assumption $\to_{\mathcal{S}} \subseteq \to_{\mathcal{R}}$ yields $\to^+_{\mathcal{S}/\mathcal{R}} \subseteq \to^*_{\mathcal{R}}$ which ensures $> \subseteq \geqslant$. Combining $\alpha = s = \beta$, $\gamma = u$, and $\delta = t$ with $s \to_{\mathcal{R}} t$ and $s \to_{\mathcal{R}} u$ yields $\alpha = \beta \geqslant \gamma, \delta$ for local peaks according to (parallel) and (variable-linear) in Figures 2(a) and 2(b). ◀

The L-labeling from the previous lemma allows to establish a decrease with respect to some steps of $\mathcal{R}$. The next lemma allows to combine L-labelings. Let $\ell_1\colon \Lambda_{\mathcal{R}} \to W_1$ and $\ell_2\colon \Lambda_{\mathcal{R}} \to W_2$. Then $(\ell_1, \geqslant_1, >_1) \times (\ell_2, \geqslant_2, >_2)$ is defined as $(\ell_1 \times \ell_2, \geqslant_{12}, >_{12})$ where $\ell_1 \times \ell_2\colon \Lambda_{\mathcal{R}} \to W_1 \times W_2$ with $(\ell_1 \times \ell_2)(\Gamma) = (\ell_1(\Gamma), \ell_2(\Gamma))$. Furthermore $(x_1, x_2) \geqslant_{12} (y_1, y_2)$ if and only if $x_1 > y_1$ or $x_1 \geqslant_1 y_1$ and $x_2 \geqslant_2 y_2$ and $(x_1, x_2) >_{12} (y_1, y_2)$ if and only if $x_1 >_1 y_1$ or $x_1 \geqslant_1 y_1$ and $x_2 >_2 y_2$.



(a) Labeling ℓ_1 **(b)** Labeling ℓ_2 **(c)** Labeling $\ell_1 \times \ell_2$

Figure 3 Lexicographic combination of L-labelings.

▶ **Lemma 3.8.** *Let ℓ_1 and ℓ_2 be L-labelings. Then $\ell_1 \times \ell_2$ is an L-labeling.*

Proof. First we remark that $\ell_1 \times \ell_2$ is a labeling whenever ℓ_1 and ℓ_2 are labelings. Next we show that $\ell_1 \times \ell_2$ satisfies the properties from Definition 3.2. If ℓ_1 and ℓ_2 are L-labelings then the diagram of Figure 2(b) has the shape as in Figure 3(a) and 3(b), respectively. It is easy to see that the lexicographic combination is again an L-labeling (cf. Figure 3(c)). ◀

3.2 Left-linear TRSs

For left-linear TRSs the notion of an LL-labeling is introduced.

▶ **Definition 3.9.** Let ℓ be a labeling for a TRS $\mathcal{R}$. We call ℓ an LL-labeling (for $\mathcal{R}$) if $\alpha \geqslant \gamma$ and $\beta \geqslant \delta$ in Figure 2(a) and $\alpha > \overline{\gamma}$ and $\beta \geqslant \delta$ in Figure 2(c). Here $\alpha > \overline{\gamma}$ means $\alpha \geqslant \gamma_1$ and $\alpha > \gamma_i$ for $2 \leqslant i \leqslant n$.

The next theorem states that LL-labelings allow to show confluence of left-linear TRSs.

▶ **Theorem 3.10.** *Let $\mathcal{R}$ be a left-linear TRS. If the critical peaks of $\mathcal{R}$ are LL-decreasing then $\mathcal{R}$ is confluent.*

Proof. By assumption the critical peaks of $\mathcal{R}$ are decreasing for some LL-labeling ℓ. We establish confluence of $\mathcal{R}$ by Theorem 2.2, i.e., show extended local decreasingness of the ARS $\langle \mathcal{T}(\mathcal{F}, \mathcal{V}), \to_{\mathcal{R}} \rangle$ by labeling rewrite steps according to ℓ. By definition of an LL-labeling local peaks according to (parallel) and (variable-left-linear) are extended locally decreasing. The reasoning for local peaks according to (critical overlap) is the same as in the proof of Theorem 3.3. ◀

The rule labeling from Definition 3.4 is not an LL-labeling since in Figure 2(c) we have $\alpha = \gamma_i$ for $1 \leqslant i \leqslant n$ which does not satisfy $\alpha > \overline{\gamma}$ if $n > 1$. (See also [8, Example 5].) In contrast, the L-labeling from Lemma 3.7 can be adapted to an LL-labeling.

▶ **Lemma 3.11.** *Let $\mathcal{R}$ be a left-linear TRS. Then $\ell_{\mathrm{sn}}^{\mathcal{R}_{\mathsf{d}}}$ is an LL-labeling, provided $\mathcal{R}_{\mathsf{d}}/\mathcal{R}_{\mathsf{nd}}$ is terminating.*

Proof. By Theorem 2.1 the relative TRS $\mathcal{R}_{\mathsf{d}}/\mathcal{R}_{\mathsf{nd}}$ is terminating if and only if $\mathcal{R}_{\mathsf{d}}/\mathcal{R}$ is. Hence $(\ell_{\mathrm{sn}}^{\mathcal{R}_{\mathsf{d}}}, \geqslant, >)$ is a labeling by Lemma 3.7. Here $\geqslant := \to_{\mathcal{R}}^*$ and $> := \to_{\mathcal{R}_{\mathsf{d}}/\mathcal{R}}^+$. Since $\ell_{\mathrm{sn}}(s \to t) = s$, we have $\alpha = \beta$ in Figures 2(a) and 2(c). We have $> \subseteq \geqslant$. Hence $\alpha \geqslant \gamma$ and $\alpha \geqslant \delta$ in Figure 2(a) and if $l_2 \to r_2$ in (1) is linear also in Figure 2(c). If $l_2 \to r_2$ is not linear then it must be duplicating and hence $\alpha > \gamma_i$ for $1 \leqslant i \leqslant n$. Combining this with $\alpha \geqslant \delta$ from above we obtain that $\ell_{\mathrm{sn}}^{\mathcal{R}_{\mathsf{d}}}$ is an LL-labeling for $\mathcal{R}$. ◀

To combine the previous lemma with the rule labeling we study how different labelings can be combined and introduce the following notion.

▶ **Definition 3.12.** Let ℓ be an L-labeling. We call ℓ a *weak* LL-labeling if $\alpha \geqslant \overline{\gamma}$ and $\beta \geqslant \delta$. for peaks according to Figure 2(c). Here $\alpha \geqslant \overline{\gamma}$ means $\alpha \geqslant \gamma_i$ for $1 \leqslant i \leqslant n$.

▶ **Remark 3.13.** The L-labelings presented so far (cf. Lemmata 3.5 and 3.7) are weak LL-labelings. Furthermore if ℓ_1 and ℓ_2 are weak LL-labelings then so are $\ell_1 \times \ell_2$ and $\ell_2 \times \ell_1$.

▶ **Lemma 3.14.** *Let ℓ_1 be an LL-labeling and let ℓ_2 be a weak LL-labeling. Then $\ell_1 \times \ell_2$*[1] *and $\ell_2 \times \ell_1$ are LL-labelings.*

Proof. By the proof of Lemma 3.8 $\ell_1 \times \ell_2$ and $\ell_2 \times \ell_1$ are labelings. The only interesting case of (variable-left-linear) is when $l_2 \to r_2$ in (1) is non-linear, i.e., $\overline{\gamma}$ contains more than one element. First we show that $\ell_1 \times \ell_2$ is an LL-labeling. Here labels according to ℓ_1 are suffixed with $_1$ and similarly for ℓ_2. Recall Figure 2(c). By assumption we have $\alpha_1 > \overline{\gamma_1}$, $\beta_1 \geqslant \delta_1$ and $\alpha_2 \geqslant \overline{\gamma_2}$, $\beta_2 \geqslant \delta_2$, which yields the desired $(\alpha_1, \alpha_2) \geqslant (\gamma_{11}, \gamma_{21})$, $(\alpha_1, \alpha_2) > (\gamma_{1i}, \gamma_{2i})$ for $2 \leqslant i \leqslant n$, and $(\beta_1, \beta_2) \geqslant (\delta_1, \delta_2)$. In the proof for $\ell_2 \times \ell_1$ the assumptions yield $(\alpha_2, \alpha_1) \geqslant (\gamma_{21}, \gamma_{11})$, $(\alpha_2, \alpha_1) > (\gamma_{2i}, \gamma_{1i})$ for $2 \leqslant i \leqslant n$, and $(\beta_2, \beta_1) \geqslant (\delta_2, \delta_1)$. ◀

In particular LL-labelings can be composed lexicographically.

▶ **Lemma 3.15.** *Every LL-labeling is a weak LL-labeling.*

Proof. By the global assumption that $> \subseteq \geqslant$. ◀

From Theorem 3.10 and Lemmata 3.11 and 3.14 we obtain the following result.

▶ **Corollary 3.16.** *Let $\mathcal{R}$ be a left-linear TRS. If $\mathcal{R}_{\mathsf{d}}/\mathcal{R}_{\mathsf{nd}}$ is terminating and all critical peaks of $\mathcal{R}$ are weakly LL-decreasing then $\mathcal{R}$ is confluent.*

Proof. By Lemma 3.11 $\ell_{\mathrm{sn}}^{\mathcal{R}_{\mathsf{d}}}$ is an LL-labeling. By assumption the critical peaks of $\mathcal{R}$ are decreasing for some weak LL-labeling ℓ. By Lemma 3.14 also $\ell_{\mathrm{sn}}^{\mathcal{R}_{\mathsf{d}}} \times \ell$ is an LL-labeling. It remains to show decreasingness of the critical peaks for $\ell_{\mathrm{sn}}^{\mathcal{R}_{\mathsf{d}}} \times \ell$. This is obvious since for terms s, t, u with $s \to_{\mathcal{R}} t \to_{\mathcal{R}} u$ we have $\ell_{\mathrm{sn}}^{\mathcal{R}_{\mathsf{d}}}(s \to t) \geqslant \ell_{\mathrm{sn}}^{\mathcal{R}_{\mathsf{d}}}(t \to u)$. Hence decreasingness for ℓ implies decreasingness for $\ell_{\mathrm{sn}}^{\mathcal{R}_{\mathsf{d}}} \times \ell$. Confluence of $\mathcal{R}$ follows from Theorem 3.10. ◀

We revisit the example from the introduction.

▶ **Example 3.17.** Recall the TRS $\mathcal{R}$ from Example 1.1. The polynomial interpretation

$$+_{\mathbb{N}}(x,y) = x + y \qquad \mathsf{s}_{\mathbb{N}}(x) = x + 1 \qquad \times_{\mathbb{N}}(x,y) = x^2 + xy + y^2 \qquad \mathsf{sq}_{\mathbb{N}}(x) = 3x^2 + 1$$

shows termination of $\mathcal{R}_{\mathsf{d}}/\mathcal{R}_{\mathsf{nd}}$. It is easy to check that ℓ_{rl}^{i} with $i(3) = i(6) = 2$, $i(4) = i(10) = 1$, and all other rules labeled 0, shows the 34 critical peaks decreasing.

The next example is more suitable to familiarize the reader with Corollary 3.16. Note that also here no standard criterion for confluence applies.

[1] Here the condition that ℓ_2 is a weak LL-labeling can be weakened to $\alpha \geqslant \gamma_1$ and $\beta \geqslant \delta$ in Figure 2(c).

▶ **Example 3.18.** Consider the TRS $\mathcal{R}$ consisting of the three rules

$$1\colon \mathsf{b} \to \mathsf{a} \qquad 2\colon \mathsf{a} \to \mathsf{b} \qquad 3\colon \mathsf{f}(\mathsf{g}(x,\mathsf{a})) \to \mathsf{g}(\mathsf{f}(x),\mathsf{f}(x))$$

We have $\mathcal{R}_\mathsf{d} = \{3\}$ and $\mathcal{R}_\mathsf{nd} = \{1,2\}$. Termination of $\mathcal{R}_\mathsf{d}/\mathcal{R}_\mathsf{nd}$ can be established by LPO with precedence $\mathsf{a} \sim \mathsf{b}$ and $\mathsf{f} > \mathsf{g}$. The rule labeling that takes the rule numbers as labels shows the only critical peak decreasing, i.e., $\mathsf{f}(\mathsf{g}(x,\mathsf{b})) \;{}_2\!\leftarrow \mathsf{f}(\mathsf{g}(x,\mathsf{a})) \to_3 \mathsf{g}(\mathsf{f}(x),\mathsf{f}(x))$ and $\mathsf{f}(\mathsf{g}(x,\mathsf{b})) \to_1 \mathsf{f}(\mathsf{g}(x,\mathsf{a})) \to_3 \mathsf{g}(\mathsf{f}(x),\mathsf{f}(x))$ which allows to establish confluence of $\mathcal{R}$ by Corollary 3.16.

▶ **Remark 3.19.** Using $\ell^i_{\mathrm{rl}}(\cdot) = 0$ as weak LL-labeling, Corollary 3.16 gives a condition (termination of $\mathcal{R}_\mathsf{d}/\mathcal{R}_\mathsf{nd}$) such that $s \to^= t$ or $t \to^= s$ for all critical pairs $s \leftarrow\!\!\rtimes\!\!\rightarrow t$ implies confluence of a left-linear TRS $\mathcal{R}$. This partially answers one question in RTA LooP #13.[2]

Next we prepare for a different LL-labeling. In [12, Example 20] van Oostrom suggests to count function symbols above the contracted redex, demands that this measurement decreases for variables that are duplicated, and combines this with the rule labeling. Consequently local peaks according to Figure 2(c) are decreasing. Below we exploit this idea but incorporate the following beneficial generalizations. First, we do not restrict to counting function symbols (which has been adopted and extended by Aoto in [1]) but represent the constraints as a relative termination problem. This abstract formulation allows to strictly subsume the criteria from [12, 1] (see Section 4) because more advanced techniques than counting symbols can be applied for proving termination. Additionally, our setting also allows to weaken these constraints significantly (see Lemma 3.27).

The next example motivates an LL-labeling that does not require termination of $\mathcal{R}_\mathsf{d}/\mathcal{R}_\mathsf{nd}$.

▶ **Example 3.20.** Consider the TRS $\mathcal{R}$ consisting of the six rules

$$\mathsf{f}(\mathsf{h}(x)) \to \mathsf{h}(\mathsf{g}(\mathsf{f}(x),x,\mathsf{f}(\mathsf{h}(\mathsf{a})))) \qquad \mathsf{f}(x) \to \mathsf{a} \qquad \mathsf{a} \to \mathsf{b}$$
$$\mathsf{h}(x) \to \mathsf{c} \qquad \mathsf{b} \to \bot \qquad \mathsf{c} \to \bot$$

Since the duplicating rule admits an infinite sequence Corollary 3.16 cannot succeed.

In the sequel we let $\mathcal{G}$ be the signature consisting of unary function symbols $f_1, \dots, f_n$ for every n-ary function symbol $f \in \mathcal{F}$.

▶ **Definition 3.21.** Let $x \in \mathcal{V}$. We define a partial mapping $\star$ from $\mathcal{T}(\mathcal{F},\mathcal{V}) \times \mathcal{P}\mathsf{os}(\mathcal{T}(\mathcal{F},\mathcal{V}))$ to terms in $\mathcal{T}(\mathcal{G},\mathcal{V})$ as follows:

$$\star(f(t_1,\dots,t_n),p) = \begin{cases} f_i(\star(t_i,q)) & \text{if } p = iq \\ x & \text{if } p = \epsilon \end{cases}$$

For a left-linear TRS $\mathcal{R}$ we abbreviate $\mathcal{R}^\star_> / \mathcal{R}^\star_=$ by $\star(\mathcal{R})$. Here, for $\rtimes \in \{>,=\}$,

$$\mathcal{R}^\star_\rtimes = \{\star(l,p) \to \star(r,q) \mid l \to r \in \mathcal{R},\ l|_p = r|_q = y,\ y \in \mathcal{V}, \text{ and } |r|_y \rtimes 1\}$$

The next example illustrates the transformation $\star(\cdot)$.

▶ **Example 3.22.** Consider the TRS $\mathcal{R}$ from Example 3.20. The relative TRS $\star(\mathcal{R}) = \mathcal{R}^\star_> / \mathcal{R}^\star_=$ consists of the TRS $\mathcal{R}^\star_>$ with rules

$$\mathsf{f}_1(\mathsf{h}_1(x)) \to \mathsf{h}_1(\mathsf{g}_1(\mathsf{f}_1(x))) \qquad \mathsf{f}_1(\mathsf{h}_1(x)) \to \mathsf{h}_1(\mathsf{g}_2(x))$$

and the TRS $\mathcal{R}^\star_=$ which is empty.

[2] http://rtaloop.mancoosi.univ-paris-diderot.fr/problems/13.html

▶ **Definition 3.23.** Let $\mathcal{R}$ be a TRS. Then $\ell_\star(s \to_{p,l\to r} t) = \star(s,p)$.

Due to the next lemma a termination proof of $\star(\mathcal{R})$ yields an LL-labeling.

▶ **Lemma 3.24.** *Let $\mathcal{R}$ be a TRS. Then $(\ell_\star, \geqslant, >)$ is an LL-labeling, provided $(\geqslant, >)$ is a monotone reduction pair, $\mathcal{R}^\star_> \subseteq >$, and $\mathcal{R}^\star_> \cup \mathcal{R}^\star_= \subseteq \geqslant$.*

Proof. That $(\ell_\star, \geqslant, >)$ is a labeling for $\mathcal{R}$ follows from the assumption that $(\geqslant, >)$ is a monotone reduction pair. To see that the constraints of Definition 3.9 are satisfied we argue as follows. For Figure 2(a) we have $\alpha = \gamma$ and $\beta = \delta$ because the steps drawn at opposing sides in the diagram take place at the same positions and the function symbols above these positions stay the same. For Figure 2(c) we have $\beta = \delta = x$ since the corresponding reductions take place at the root position and hence $\beta \geqslant \delta$. To see $\alpha > \overline{\gamma}$ recall the peak (1). Let q' be the unique position in $\mathcal{P}\mathsf{os}_\mathcal{V}(l_2)$ such that $qq' \leqslant p$ with $x = l_2|_{q'}$ and $Q = \{q'_1, \ldots, q'_n\}$ with $r_2|_{q'_i} = x$. By construction $\mathcal{R}^\star_>$ contains all rules $\star(s,q') \to \star(u,q'_i)$ for $1 \leqslant i \leqslant n$. Since $u \twoheadrightarrow_Q v$ we obtain $\alpha > \gamma_i$ for $1 \leqslant i \leqslant n$ (from $\mathcal{R}^\star_> \subseteq >$) and hence the desired $\alpha > \overline{\gamma}$. ◀

From Lemma 3.24 we obtain the following corollary.

▶ **Corollary 3.25.** *Let $\mathcal{R}$ be a left-linear TRS and let ℓ be a weak LL-labeling. Let $\ell_\star\ell$ denote $\ell \times \ell_\star$ or $\ell_\star \times \ell$. If $\star(\mathcal{R})$ is terminating and the critical peaks of $\mathcal{R}$ are decreasing for $\ell_\star\ell$ then $\mathcal{R}$ is confluent.*

Proof. If $\star(\mathcal{R})$ is terminating then $\ell_\star$ is an LL-labeling by Lemma 3.24. Lemma 3.14 yields that $\ell_\star\ell$ is an LL-labeling. By assumption the critical peaks are decreasing for $\ell_\star\ell$ and hence Theorem 3.10 yields the confluence of $\mathcal{R}$. ◀

The next example illustrates the use of Corollary 3.25.

▶ **Example 3.26.** We show confluence of the TRS $\mathcal{R}$ from Example 3.20. Termination of $\star(\mathcal{R})$ (cf. Example 3.22) is easily shown, e.g., the polynomial interpretation

$$\mathsf{f}_{1\mathbb{N}}(x) = 2x \qquad \mathsf{g}_{1\mathbb{N}}(x) = \mathsf{g}_{2\mathbb{N}}(x) = x \qquad \mathsf{h}_{1\mathbb{N}}(x) = x + 1$$

orients both rules in $\mathcal{R}^\star_>$ strictly. To show decreasingness of the three critical peaks (two of which are symmetric) we use $\ell_\star \times \ell^i_{\mathrm{rl}}$ with $i(\mathsf{f}(\mathsf{h}(x)) \to \mathsf{h}(\mathsf{g}(\mathsf{f}(x), x, \mathsf{f}(\mathsf{h}(\mathsf{a}))))) = 1$ and all other rules receive label 0. Since it is impractical to label with $\ell_\star$ and compare labels with respect to the monotone reduction pair obtained from the above termination proof we label a step $s \to_{p,l\to r} t$ with the constant part of the interpretation of $\star(s,p)$ (cf. Lemma 5.2 below) and compare labels with $\geqslant_\mathbb{N}$ and $>_\mathbb{N}$. E.g., a step $\mathsf{f}(\mathsf{h}(\mathsf{b})) \to \mathsf{f}(\mathsf{h}(\bot))$ is labeled 2 since $\star(\mathsf{f}(\mathsf{h}(\mathsf{b})), 11) = \mathsf{f}_1(\mathsf{h}_1(x))$ and $[\mathsf{f}_1(\mathsf{h}_1(x))]_\mathbb{N} = 2x + 2$. Hence the critical peak $\mathsf{h}(\mathsf{g}(\mathsf{f}(x), x, \mathsf{f}(\mathsf{h}(\mathsf{a})))) \leftarrow_{0,1} \mathsf{f}(\mathsf{h}(x)) \to_{0,0} \mathsf{a}$ is closed decreasingly by

$$\mathsf{h}(\mathsf{g}(\mathsf{f}(x), x, \mathsf{f}(\mathsf{h}(\mathsf{a})))) \to_{0,0} \mathsf{c} \to_{0,0} \bot \;{}_{0,0}\!\leftarrow \mathsf{b} \;{}_{0,0}\!\leftarrow \mathsf{a}$$

and the critical peak $\mathsf{h}(\mathsf{g}(\mathsf{f}(x), x, \mathsf{f}(\mathsf{h}(\mathsf{a})))) \;{}_{0,1}\!\leftarrow \mathsf{f}(\mathsf{h}(x)) \to_{0,0} \mathsf{f}(\mathsf{c})$ is closed decreasingly by

$$\mathsf{h}(\mathsf{g}(\mathsf{f}(x), x, \mathsf{f}(\mathsf{h}(\mathsf{a})))) \to_{0,0} \mathsf{c} \to_{0,0} \bot \;{}_{0,0}\!\leftarrow \mathsf{b} \;{}_{0,0}\!\leftarrow \mathsf{a} \;{}_{0,0}\!\leftarrow \mathsf{f}(\mathsf{c})$$

which allows to prove confluence of $\mathcal{R}$ by Corollary 3.25.

By definition of $\alpha > \overline{\gamma}$ (cf. Definition 3.9) we observe that the definition of $\star(\mathcal{R})$ can be relaxed. If $l_2 \to r_2$ with $l_2|_{q'} = x \in \mathcal{V}$ and $\{q'_1, \ldots, q'_n\}$ are the positions of the variable x in r_2 then it suffices if $n-1$ instances of $\star(l_2,q') \to \star(r_2,q'_i)$ are put in $\mathcal{R}^\star_>$ while one $\star(l_2,q') \to \star(r_2,q'_j)$ can be put in $\mathcal{R}^\star_=$ (since the steps labeled $\overline{\gamma}$ in Figure 2(c) are at parallel positions we can choose the first closing step such that $\alpha \geqslant \gamma_1$). This improved version of $\star(\mathcal{R})$ is denoted by $\bigstar(\mathcal{R}) = \mathcal{R}^{\star\star}_> / \mathcal{R}^{\star\star}_=$. We obtain the following variant of Lemma 3.24.

▶ **Lemma 3.27.** *Let $\mathcal{R}$ be a TRS. Then $(\ell_\star, \geqslant, >)$ is an LL-labeling, provided $(\geqslant, >)$ is a monotone reduction pair, $\mathcal{R}^{\star\star}_{>} \subseteq >$, and $\mathcal{R}^{\star\star}_{>} \cup \mathcal{R}^{\star\star}_{=} \subseteq \geqslant$.* ◀

Obviously any $\ddagger_\star(\mathcal{R})$ is terminating whenever $\star(\mathcal{R})$ is. The next example shows that the reverse statement does not hold. In Section 5 we show how the intrinsic indeterminism of $\ddagger_\star(\mathcal{R})$ is eliminated in the implementation.

▶ **Example 3.28.** Consider the TRS $\mathcal{R}$ from Example 1.1. Then $\star(\mathcal{R})$ consists of the rules

$\mathcal{R}^\star_>$		$\mathcal{R}^\star_=$
$\mathsf{sq}_1(\mathsf{s}_1(x)) \to +_1(\times_1(x))$	$\times_1(x) \to \times_2(x)$	$+_1(x) \to +_2(x)$
$\mathsf{sq}_1(\mathsf{s}_1(x)) \to +_1(\times_2(x))$	$\times_2(y) \to \times_1(y)$	$+_1(\mathsf{s}_1(x)) \to +_1(x)$
$\mathsf{sq}_1(\mathsf{s}_1(x)) \to +_2(\mathsf{s}_1(+_1(x)))$	$\times_1(\mathsf{s}_1(x)) \to +_1(\times_1(x))$	$+_2(y) \to +_1(y)$
$\mathsf{sq}_1(\mathsf{s}_1(x)) \to +_2(\mathsf{s}_1(+_2(x)))$	$\times_2(\mathsf{s}_1(y)) \to +_2(\times_2(y))$	$+_1(+_1(x)) \to +_1(x)$
$\mathsf{sq}_1(x) \to \times_1(x)$	$+_1(x) \to +_1(\mathsf{s}_1(x))$	$+_1(+_2(y)) \to +_2(+_1(y))$
$\mathsf{sq}_1(x) \to \times_2(x)$	$+_2(\mathsf{s}_1(y)) \to +_2(y)$	$+_2(z) \to +_2(+_2(z))$
$\dagger\colon \times_2(y) \to +_1(\times_2(y))$	$+_2(+_2(z)) \to +_2(z)$	$+_1(x) \to +_1(+_1(x))$
$\times_2(y) \to +_2(y)$	$+_2(y) \to +_2(\mathsf{s}_1(y))$	$+_2(+_1(y)) \to +_1(+_2(y))$
$\times_1(x) \to +_1(x)$		
$\dagger\colon \times_1(x) \to +_2(\times_1(x))$		

Let $\mathcal{R}^\star_\dagger$ denote the rules in $\mathcal{R}^\star_>$ marked with $\dagger$. Termination of $\star(\mathcal{R})$ cannot be established (because $\mathcal{R}^\star_\dagger$ is non-terminating) but we stress that moving these rules into $\mathcal{R}^\star_=$ yields a valid $\ddagger_\star(\mathcal{R})$ which can be proved terminating by the polynomial interpretation with

$$\mathsf{sq}_{1\mathbb{N}}(x) = x + 2 \qquad \times_{1\mathbb{N}}(x) = \times_{2\mathbb{N}}(x) = x + 1$$

that interprets the remaining function symbols by the identity function. We remark that Corollary 3.25 with the labeling from Lemma 3.27 establishes confluence of $\mathcal{R}$. Since all reductions in the 34 joining sequences have only $+$ above the redex and $+_{1\mathbb{N}}(x) = +_{2\mathbb{N}}(x) = x$, the $\ell_\star$ labeling attaches zero to any of these steps. The rule labeling that assigns $i(3) = i(6) = 2$, $i(4) = i(10) = 1$, and zero to all other rules shows the 34 critical peaks decreasing.

4 Assessment

In this section we relate the results from this paper with each other and contributions from [1, 8]. First we observe that Corollaries 3.16 and 3.25 subsume Theorem 3.3 since the preconditions of the corollaries evaporate for linear systems. Note that both results extend Knuth and Bendix' criterion [9] (joinability of critical pairs for terminating systems) for left-linear systems. Next we compare the power of Corollaries 3.16 and 3.25. Example 3.20 and the TRS from the following example show that Corollaries 3.16 and 3.25 are incomparable.

▶ **Example 4.1.** It is easy to adapt the TRS from Example 3.18 such that $\star(\mathcal{R})$ becomes non-terminating. Consider the TRS $\mathcal{R}$

$$1\colon \mathsf{b} \to \mathsf{a} \qquad 2\colon \mathsf{a} \to \mathsf{b} \qquad 3\colon \mathsf{f}(\mathsf{g}(x, \mathsf{a})) \to \mathsf{g}(\mathsf{f}(x), \mathsf{f}(\mathsf{g}(x, \mathsf{c})))$$

for which termination of $\mathcal{R}_{\mathsf{d}}/\mathcal{R}_{\mathsf{nd}}$ and decreasingness of the critical peaks is proved similar to Example 3.18. Note that $\mathsf{f}_1(\mathsf{g}_1(x)) \to \mathsf{g}_2(\mathsf{f}_1(\mathsf{g}_1(x))) \in \mathcal{R}^\star_>$ is non-terminating.[3]

Neither of Corollaries 3.16 and 3.25 gives a necessary confluence criterion for left-linear systems. The TRSs from Example 3.20 and 4.1 are confluent. Hence (by renaming function symbols) so is their direct sum according to Toyama's result [17]. But the combined TRS does not satisfy either precondition of our corollaries.

Next we show that our setting subsumes one of the results from [8]. To this end we define the *critical pair steps* $\mathsf{CPS}(\mathcal{R}) = \{s \to t, s \to u \mid t \leftarrow s \rightarrow u \text{ is a critical peak of } \mathcal{R}\}$.

▶ **Theorem 4.2** ([8, Theorem 6])**.** *Let $\mathcal{R}$ be a left-linear TRS. The TRS $\mathcal{R}$ is confluent if $\leftarrow \rtimes \rightarrow \subseteq \rightarrow^* \cdot {}^*\leftarrow$ and $(\mathsf{CPS}(\mathcal{R}) \cup \mathcal{R}_{\mathsf{d}})/\mathcal{R}_{\mathsf{nd}}$ is terminating.*

By Theorem 2.2 termination of $(\mathsf{CPS}(\mathcal{R}) \cup \mathcal{R}_{\mathsf{d}})/\mathcal{R}_{\mathsf{nd}}$ implies termination of $\mathsf{CPS}(\mathcal{R})/\mathcal{R}$ and $\mathcal{R}_{\mathsf{d}}/\mathcal{R}_{\mathsf{nd}}$. Hence the above result corresponds to Corollary 3.16 using $\ell_{\mathrm{sn}}^{\mathsf{CPS}(\mathcal{R})}$ as weak LL-labeling. Note that our setting is strictly more liberal because of two reasons. First we do not demand a decrease already in the peak, i.e., we can cope with non-terminating $\mathsf{CPS}(\mathcal{R})$. Second, our approach allows to combine ℓ_{sn} lexicographically with further labelings. Examples 3.18 and 3.20 show that the inclusion is strict (for the first reason).

Next we show that Corollary 3.25 generalizes the results from [1, Sections 5 and 6]. It is not difficult to see that the encoding presented in [1, Theorem 5.4] can be mimicked by Corollary 3.25 where linear polynomial interpretations over $\mathbb{N}$ of the shape as in (1)

$$(1)\quad f_{i\mathbb{N}}(x) = x + c_f \qquad\qquad (2)\quad f_{i\mathbb{N}}(x) = x + c_{f_i}$$

are used to prove termination of $\star(\mathcal{R})$ and $\ell_\star \times \ell_{\mathrm{rl}}$ is employed to show LL-decreasingness of the critical peaks. In contrast to [1, Theorem 5.4], which explicitly encodes these constraints in a single formula of linear arithmetic, our abstract formulation admits the following gains. First, we do not restrict to weight functions but allow powerful machinery for proving relative termination and second our approach allows to combine arbitrarily many labelings lexicographically (cf. Lemma 3.14). Furthermore we stress that our abstract treatment of $\star(\mathcal{R})$ allows to implement Corollary 3.25 based on $\ddagger(\mathcal{R})$ (cf. Section 5) which admits further gains in power (cf. Example 1.1 as well as Section 6).

The idea of the extension presented in [1, Example 6.1] amounts to using $\ell_{\mathrm{rl}} \times \ell_\star$ instead of $\ell_\star \times \ell_{\mathrm{rl}}$, which is an application of Lemma 3.14 in our setting. Finally, the extension discussed in [1, Example 6.3] suggests to use linear polynomial interpretations over $\mathbb{N}$ of the shape as in (2) to prove termination of $\star(\mathcal{R})$. Note that these interpretations are still weight functions. This explains why the approach from [1] fails to establish confluence of the TRSs in Examples 3.18 and 3.20 since a weight function cannot show termination of the rules $\mathsf{f}_1(\mathsf{g}_1(x)) \to \mathsf{g}_1(\mathsf{f}_1(x))$ and $\mathsf{f}_1(\mathsf{h}_1(x)) \to \mathsf{h}_1(\mathsf{g}_1(\mathsf{f}_1(x)))$, respectively.

Note that both recent approaches [1, 8] based on decreasing diagrams fail to prove the TRS $\mathcal{R}$ from Example 1.1 confluent. The former can, e.g., not cope with the non-terminating rule $\times_1(x) \to +_0(\times_1(x))$ in $\mathcal{R}^\star_>$ (cf. Example 3.28) while overlaps with the non-terminating rule $x + y \to y + x \in \mathcal{R}$ prevent the latter approach from succeeding. On the contrary Examples 3.17 and 3.28 give two confluence proofs based on our setting.

Finally we present a situation when the decreasing diagrams technique typically fails. (In a slightly different setting similar ideas are proposed in [13]. We remark that the recent

[3] We remark that it is easy to extend this example such that also $\ddagger(\mathcal{R})$ is non-terminating. Just consider the rule $\mathsf{f}(\mathsf{g}(x, \mathsf{a})) \to \mathsf{g}(\mathsf{f}(x), \mathsf{g}(\mathsf{f}(\mathsf{g}(x, \mathsf{c}), \mathsf{f}(\mathsf{g}(x, \mathsf{c})))))$.

paper [2] follows a different approach for associativity and commutativity.) To handle such cases we use the following well-known result.

▶ **Lemma 4.3.** *Let $\to \subseteq \rightarrowtail \subseteq \to^*$. Then confluence of $\rightarrowtail$ implies confluence of $\to$.* ◀

The following example contains rules for associativity and commutativity.

▶ **Example 4.4.** Consider the TRS $\mathcal{R}$ consisting of the following two rules

$$x \circ (y \circ z) \to (x \circ y) \circ z \qquad\qquad x \circ y \to y \circ x$$

All four critical peaks are joinable but the critical peak $(y \circ z) \circ x \leftarrow x \circ (y \circ z) \to (x \circ y) \circ z$ cannot be shown decreasing with our labeling functions. Let $\mathcal{S}$ be the TRS $\mathcal{R}$ augmented by the rule $(x \circ y) \circ z \to x \circ (y \circ z)$. All twelve critical peaks of $\mathcal{S}$ can be shown decreasing by the rule labeling and hence $\mathcal{S}$ is confluent. Confluence of $\mathcal{R}$ follows by Lemma 4.3.

5 Implementation

In this section we sketch how the results from this paper can be implemented.

Before decreasingness of critical peaks can be investigated, the critical pairs must be shown convergent. For a critical pair $t \leftarrow\!\!\rtimes\!\!\rightarrow u$ in our implementation we consider all joining sequences such that $t \to^{\leqslant n} \cdot {}^{\leqslant n}\!\leftarrow u$ and there is no smaller n that admits a common reduct.

To exploit the possibility for incremental confluence proofs by lexicographically combining labels (cf. Lemmata 3.8 and 3.14) our implementation labels rewrite steps with tuples of natural numbers. Since our labeling functions are implemented by encoding the constraints in non-linear (integer) arithmetic it is straightforward to combine existing labels (some partial progress) with the search for a new labeling that shows the critical peaks decreasing.

It is straightforward to implement Corollary 3.16. After establishing termination of $\mathcal{R}_{\mathsf{d}}/\mathcal{R}_{\mathsf{nd}}$ (e.g., by an external termination prover) any weak LL-labeling can be tried to show the critical peaks decreasing. In [1, 8] it is shown how the rule labeling can be implemented by encoding the constraints in linear arithmetic.

We sketch how to implement the labeling $\ell_{\mathrm{sn}}^{\mathcal{S}}$ from Lemma 3.7 as a relative termination problem. First we fix a suitable set $\mathcal{S}$, i.e., we extend the definition of critical pair steps to *critical diagram steps*: $\mathsf{CDS}(\mathcal{R}) = \{s \to t, s \to u, t_i \to t_{i+1}, u_j \to u_{j+1} \mid t \leftarrow s \to u$ is a critical peak in $\mathcal{R}$, $t = t_0 \to \cdots \to t_n = u_m \leftarrow \cdots \leftarrow u_0 = u$, $0 \leqslant i < n-1$, $0 \leqslant j < m-1\}$. Facing the relative termination problem $\mathsf{CDS}(\mathcal{R})/\mathcal{R}$ we try to simplify it according to Theorem 2.1 into some $\mathcal{S}_1/\mathcal{S}_2$. Note that it is not necessary to finish the proof. By Theorem 2.1 the relative TRS $(\mathsf{CDS}(\mathcal{R}) \setminus \mathcal{S}_1)/\mathcal{R}$ is terminating and hence by Lemma 3.7 $\ell_{\mathrm{sn}}^{\mathsf{CDS}(\mathcal{R})\setminus\mathcal{S}_1}$ is an L-labeling. Let $\geqslant := \to_{\mathcal{R}}^*$ and $> := \to^+_{(\mathsf{CDS}(\mathcal{R})\setminus\mathcal{S}_1)/\mathcal{R}}$. Since $\geqslant$ and $>$ can never increase by rewriting, it suffices to exploit the first decrease with respect to $>$. Next we show how critical diagrams are labeled with natural numbers. Consider a rewrite sequence $v_1 \to_{\mathcal{R}} v_2 \to_{\mathcal{R}} \cdots \to_{\mathcal{R}} v_l$. If $v_1 \to_{\mathcal{S}_1}^* v_l$ then all steps are labeled with 1. Otherwise take the largest $k < l$ such that $v_1 \to_{\mathcal{S}_1}^* v_k \to_{\mathcal{R}} v_{k+1} \to_{\mathcal{R}}^* v_l$. Then we set $\ell_{\mathrm{sn}}(v_i \to v_{i+1}) = 1$ for $1 \leqslant i \leqslant k$ and $\ell_{\mathrm{sn}}(v_i \to v_{i+1}) = 0$ for $k < i < l$. Note that $v_k \to v_{k+1}$ is the first step that causes a decrease with respect to $>$, i.e., $v_1 \to_{(\mathsf{CDS}(\mathcal{R})\setminus\mathcal{S}_1)/\mathcal{R}} v_{k+1}$. We demonstrate the above idea on an example.

▶ **Example 5.1.** Consider the following TRS $\mathcal{R}$ from [3]:

$$\mathsf{I}(x) \to \mathsf{I}(\mathsf{J}(x)) \qquad \mathsf{J}(x) \to \mathsf{J}(\mathsf{K}(\mathsf{J}(x))) \qquad \mathsf{H}(\mathsf{I}(x)) \to \mathsf{K}(\mathsf{J}(x)) \qquad \mathsf{J}(x) \to \mathsf{K}(\mathsf{J}(x))$$

We show how the labels for the critical peak $\mathsf{H}(\mathsf{I}(\mathsf{J}(x))) \;{}_1\!\leftarrow \mathsf{H}(\mathsf{I}(x)) \to_1 \mathsf{K}(\mathsf{J}(x))$ and the joining sequences $\mathsf{H}(\mathsf{I}(\mathsf{J}(x))) \to_1 \mathsf{K}(\mathsf{J}(\mathsf{J}(x))) \to_0 \mathsf{K}(\mathsf{J}(\mathsf{K}(\mathsf{J}(x)))) \;{}_1\!\leftarrow \mathsf{K}(\mathsf{J}(x))$ can be established by $\ell^{\mathcal{S}}_{\mathrm{sn}}$. Let $\mathcal{S}$ be the TRS generated by the steps in the critical peak and the joining sequences above. The interpretation $\mathsf{K}_{\mathbb{N}}(x) = \mathsf{H}_{\mathbb{N}}(x) = \mathsf{J}_{\mathbb{N}}(x) = x$ and $\mathsf{I}_{\mathbb{N}}(x) = x + 1$ allows to "simplify" termination of the problem $\mathcal{S}/\mathcal{R}$ according to Theorem 2.1. Since the rules that reduce the number of $\mathsf{I}'s$ are dropped from $\mathcal{S}$ (and $\mathcal{R}$), those rules admit a decrease in the labeling.

The above trick does not work to implement Corollary 3.25, since $s \to_{\mathcal{R}} t \to_{\mathcal{R}} v$ does not ensure $\ell_\star(s \to t) \geqslant \ell_\star(t \to v)$. Here the solution is to employ only techniques (for proving the relative TRS $\star(\mathcal{R})$ terminating) that can label a rewrite step with a concrete number. To this end we will recall matrix interpretations [5] which are a very powerful method for proving termination of relative rewrite systems that allow to compute a variant of $\ell_\star$.

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 $[\alpha]_{\mathcal{A}}(\cdot)$ we denote the usual evaluation function of $\mathcal{A}$ according to an assignment α which maps variables to values in A. An $\mathcal{F}$-algebra $\mathcal{A}$ together with a well-founded order $\succ$ and a quasi-order $\succsim$ on A is called a *monotone algebra* if every $f_{\mathcal{A}}$ is monotone with respect to $\succsim$ and $\succ$ and the inclusion $\succsim \cdot \succ \cdot \succsim \subseteq \succ$ holds. Any monotone algebra $(\mathcal{A}, \succsim, \succ)$ induces a well-founded order on terms: $s \succ_{\mathcal{A}} t$ if for any assignment α the condition $[\alpha]_{\mathcal{A}}(s) \succ [\alpha]_{\mathcal{A}}(t)$ holds. The quasi-order $s \succsim_{\mathcal{A}} t$ is similarly defined.

Matrix interpretations $(\mathcal{M}, \succ)$ (often just denoted $\mathcal{M}$) are a special kind of monotone algebras. Here the carrier is $\mathbb{N}^d$ for some fixed dimension $d \in \mathbb{N} \setminus \{0\}$. The orders $\succsim$ and $\succ$ are defined on $\mathbb{N}^d$ as $\vec{u} \succsim \vec{v}$ if $(\vec{u})_i \geqslant_{\mathbb{N}} (\vec{v})_i$ for all $1 \leqslant i \leqslant d$ and $\vec{u} \succ \vec{v}$ if $\vec{u} \succsim \vec{v}$ and $(\vec{u})_1 >_{\mathbb{N}} (\vec{v})_1$. Here $(\vec{v})_1$ denotes the first element of the vector $\vec{v}$. 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 $F_i \in \mathbb{N}^{d \times d}$ for all $1 \leqslant i \leqslant n$ and $\vec{f} \in \mathbb{N}^d$ then monotonicity of $\succ$ is achieved by demanding that the top left entry of every matrix F_i is non-zero. Let α_0 denote the assignment with $\alpha(x) = \vec{0}$ for all variables x.

▶ **Lemma 5.2.** *Let $\mathcal{R}$ be a TRS and $\ell^{\mathcal{M}}_\star(s \to_{p,l\to r} t) = ([\alpha_0]_{\mathcal{M}}(\star(s,p)))_1$ for some matrix interpretation $\mathcal{M}$. Then $(\ell^{\mathcal{M}}_\star, \geqslant_{\mathbb{N}}, >_{\mathbb{N}})$ is a weak LL-labeling, provided $\mathcal{R}^\star_> \cup \mathcal{R}^\star_= \subseteq \succsim_{\mathcal{M}}$.*

Proof. That $\ell^{\mathcal{M}}_\star$ is a labeling follows from the fact that $(\succsim_{\mathcal{M}}, \succ_{\mathcal{M}})$ is a monotone reduction pair and $\mathcal{R}^\star_> \cup \mathcal{R}^\star_= \subseteq \succsim_{\mathcal{M}}$. The latter also ensures that $\ell^{\mathcal{M}}_\star$ is a weak LL-labeling. ◀

To establish progress with Lemma 5.2 the implementation demands $\succ_{\mathcal{M}} \cap \mathcal{R}^\star_> \neq \varnothing$. By repeated applications of Lemmata 5.2 and 3.8 weak LL-labelings are combined lexicographically until they form an LL-labeling. This is exactly the case if termination of $\overset{\star}{\star}(\mathcal{R})$ can be established using matrix interpretations.

Finally, we explain why $\overset{\star}{\star}(\mathcal{R})$ need not be computed explicitly to implement Corollary 3.25 with the labeling from Lemma 3.27. The idea is to start with $\star(\mathcal{R})$ and incrementally prove termination of $\mathcal{R}^\star_>/\mathcal{R}^\star_=$ until some $\mathcal{S}_1/\mathcal{S}_2$ is reached. If all left-hand sides in $\mathcal{S}_1$ are distinct then they must have been derived from different combinations (l, x) with $l \to r \in \mathcal{R}$ and $x \in \mathcal{V}\mathsf{ar}(l)$. Hence they are exactly those rules which should be placed in $\mathcal{R}^\star_=$. We show the idea by means of an example.

▶ **Example 5.3.** We revisit Example 1.1 and try to prove termination of $\star(\mathcal{R})$. By an application of Theorem 2.1 with the interpretation given in Example 1.1 the problem is termination equivalent to $\mathcal{R}_\dagger/\mathcal{R}^\star_=$ and by another application of Theorem 2.1 the same proof can be used to show termination of $(\mathcal{R}^\star_> \setminus \mathcal{R}^\star_\dagger)/(\mathcal{R}^\star_= \cup \mathcal{R}^\star_\dagger)$ which is a suitable candidate for $\overset{\star}{\star}(\mathcal{R})$ since the rules in $\mathcal{R}^\star_\dagger$ have different left-hand sides.

	without Lemma 4.3				with Lemma 4.3			
method	pre	CR(ℓ_{rl})	CR(ℓ_{sn})	CR	pre	CR(ℓ_{rl})	CR(ℓ_{sn})	CR
rule labeling	40(0.2)	35(0.2)	–	–	40(0.2)	37(0.3)	–	–
Corollary 3.16	45(0.3)	40(0.6)	37(1.4)	42(1.5)	45(0.3)	42(0.7)	40(1.4)	44(1.5)
Corollary 3.25$\star$	45(0.3)	42(0.3)	34(1.1)	42(1.2)	45(0.3)	44(0.3)	38(1.1)	44(1.2)
Corollary 3.25$\overset{\star}{\star}$	48(0.3)	45(0.3)	36(1.1)	45(1.2)	48(0.3)	47(0.4)	40(1.1)	47(1.3)
ACP	–	42(0.1)	–	48(0.1)	–	–	–	–

Table 1 Experimental results for 53 left-linear TRSs.

6 Experiments

The results from the paper have been implemented and form the core of the confluence prover CSI [18]. For experiments[4] we used the collection from [1][5] which consists of 106 TRSs from the rewriting literature dealing with confluence. From these systems 67 are left-linear, but 14 of these are known to be non-confluent which gives a theoretical upper bound of 53 systems for which the proposed methods can succeed. Our experiments have been performed on a notebook equipped with an Intel® Core™2 Duo processor U9400 running at a clock rate of 1.4 GHz and 3 GB of main memory.

Table 1 shows an evaluation of the results from this paper. The first column indicates which criterion has been used to investigate confluence. A $\star$ means that the corresponding corollary is implemented using $\star(\mathcal{R})$ whereas $\overset{\star}{\star}$ refers to $\overset{\star}{\star}(\mathcal{R})$. The column labeled pre shows for how many systems the precondition of the respective criterion is satisfied, e.g., for rule labeling the precondition is linearity while for Corollary 3.16 the precondition is termination of $\mathcal{R}_{\mathsf{d}}/\mathcal{R}_{\mathsf{nd}}$. The columns labeled CR(ℓ) give the number of systems for which confluence could be established using labeling ℓ. (For Corollary 3.25 implicitly $\ell_\star$ is also employed.) The column labeled CR corresponds to the full power of our approach, i.e., when the lexicographic combination of all labelings is used. In Table 1 the numbers in parentheses indicate the average time for establishing the precondition (column pre) and finding a confluence proof (remaining columns) in seconds, respectively. These timings show that establishing the precondition is fast and the same holds for ℓ_{rl}. The most costly criterion is ℓ_{sn} which is also used in column CR. All tests finished within 60 seconds.

From the table we draw the following conclusions. Depending on the labeling function (ℓ_{rl} versus ℓ_{sn}) either Corollary 3.16 or Corollary 3.25 can handle more systems. When both labelings are used, Corollary 3.25$\overset{\star}{\star}$ subsumes Corollary 3.16 on this testbed. Corollary 3.25$\star$ does not subsume Corollary 3.16 since $\star(\mathcal{R})$ is non-terminating for the TRS in Example 1.1 which is contained in this testbed. For the systems where Corollary 3.25$\overset{\star}{\star}$ succeeded but Corollary 3.25$\star$ failed the corresponding relative TRS $\star(\mathcal{R})$ is non-terminating. The three systems where the precondition of Corollary 3.25 is satisfied but confluence could not be shown (without Lemma 4.3) contain rules for associativity and commutativity. To cope with (two of) these systems we exploit the ideas from Example 4.4. The corresponding numbers are given in the right part of Table 1.

For reference we also give the data for ACP [3], a powerful confluence prover which implements various confluence criteria from the literature. According to [1] their tool can

[4] Details available from `http://cl-informatik.uibk.ac.at/software/csi/labeling`.
[5] `http://www.nue.riec.tohoku.ac.jp/tools/acp/examples/crexamples-100410.tgz`

method	1.1	3.18	3.20	4.1	5.1
rule labeling	×(0.2)	×(0.2)	×(0.3)	×(0.2)	×(0.2)
Corollary 3.16	✓(4.6)	✓(0.5)	×(0.5)	✓(0.6)	✓(1.9)
Corollary 3.25$\star$	×(1.4)	✓(0.5)	✓(1.5)	×(0.3)	✓(1.8)
Corollary 3.25$\ddagger_\star$	✓(4.6)	✓(0.5)	✓(1.3)	✓(0.5)	✓(1.9)
ACP	×(11.5)	×(0.1)	×(0.7)	×(0.1)	✓(0.1)

Table 2 Experimental results for the examples from the paper.

show 42 systems confluent by (their extensions of) the rule labeling and using its full power ACP can prove 48 systems confluent. In the collection considered for Table 1 there is one system (Example 1.1) which ACP cannot handle but where our approach (because we consider $\ddagger_\star(\mathcal{R})$) succeeds and two systems that we can show confluent by adding rules as proposed in Example 4.4. Consequently we miss four systems compared to the full power of ACP which handles them by considering development closedness [11] and parallel critical pairs [16, 7]. Note that these criteria investigate confluence of $\multimap$ and $\parallel$, while our approach considers $\to$. Since for these systems neither $\mathcal{R}_\mathsf{d}/\mathcal{R}_\mathsf{nd}$ nor $\ddagger_\star(\mathcal{R})$ is terminating there also is not much hope that our current approach can be extended to handle these systems. Hence as future work we will study properties on labeling functions that allow to investigate confluence of $\parallel$ and $\multimap$.

Table 2 does a similar evaluation as Table 1 on the examples from the paper. Here a ✓ indicates that the tool could establish confluence while a × means that the tool failed. The numbers in parentheses give the time (in seconds) the tool spent on the respective example.

From Tables 1 and 2 we conclude that our framework admits a state-of-the-art confluence prover for left-linear systems. For the sake of completeness we remark that ACP also supports confluence analysis for non-left-linear systems.

7 Conclusion

In this paper we studied how the decreasing diagrams technique can be automated. We presented conditions (subsuming recent related results) that ensure confluence of a left-linear TRS whenever its critical peaks are decreasing. The labelings we proposed can be combined lexicographically which allows incremental proofs of confluence and has a modular flavor in the following sense: Whenever a new labeling function is invented, the whole framework gains power. We discussed several situations (Examples 1.1, 3.18, 3.20, 4.1) where standard confluence techniques fail but our approach easily establishes confluence.

Currently all our investigations are aimed to show confluence of $\to$. As motivated in Section 6 one obvious issue for future work is to study conditions on the labelings such that $\parallel$ (or $\multimap$) can be shown confluent. This would allow to handle the systems which we currently lose against ACP in Table 1. Furthermore, if the recent developments in the termination community will also reach confluence, then automatic certification of confluence proofs by means of a theorem prover is inevitable. Since our setting is based on a single method (decreasing diagrams) while still powerful it seems to be a good starting point for certification efforts.

References

1 T. Aoto. Automated confluence proof by decreasing diagrams based on rule-labelling. In *Proc. 21st International Conference on Rewriting Techniques and Applications*, volume 6 of *Leibniz International Proceedings in Informatics*, pages 7–16, 2010.

2 T. Aoto and Y. Toyama. A reduction-preserving completion for proving confluence of non-terminating term rewriting systems. In *Proc. 22nd International Conference on Rewriting Techniques and Applications*, Leibniz International Proceedings in Informatics, 2011. To appear.

3 T. Aoto, J. Yoshida, and Y. Toyama. Proving confluence of term rewriting systems automatically. In *Proc. 20th International Conference on Rewriting Techniques and Applications*, volume 5595 of *Lecture Notes in Computer Science*, pages 93–102, 2009.

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

5 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.

6 A. Geser. *Relative Termination*. PhD thesis, Universität Passau, 1990. Available as technical report 91-03.

7 B. Gramlich. Confluence without termination via parallel critical pairs. In *Proc. 21st International Colloquium on Trees in Algebra and Programming*, volume 1059 of *Lecture Notes in Computer Science*, pages 211–225, 1996.

8 N. Hirokawa and A. Middeldorp. Decreasing diagrams and relative termination. In *Proc. 5th International Joint Conference on Automated Reasoning*, volume 6173 of *Lecture Notes in Artificial Intelligence*, pages 487–501, 2010.

9 D.E. Knuth and P.B. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, 1970.

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

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

12 V. van Oostrom. Confluence by decreasing diagrams – converted. In *Proc. 19th International Conference on Rewriting Techniques and Applications*, volume 5117 of *Lecture Notes in Computer Science*, pages 306–320, 2008.

13 M. Oyamaguchi and Y. Ohta. A new parallel closed condition for Church-Rosser of left-linear term rewriting systems. In *Proc. 8th International Conference on Rewriting Techniques and Applications*, volume 1232 of *Lecture Notes in Computer Science*, pages 187–201, 1997.

14 M. Oyamaguchi and Y. Ohta. On the Church-Rosser property of left-linear term rewriting systems. *IEICE Transactions on Information and Systems*, E86-D(1):131–135, 2003.

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

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

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

18 H. Zankl, B. Felgenhauer, and A. Middeldorp. CSI – A confluence tool. In *Proc. 23rd International Conference on Automated Deduction*, Lecture Notes in Artificial Intelligence, 2011. To appear.

Proving Equality of Streams Automatically

Hans Zantema[1] and Jörg Endrullis[2]

1 **Department of Computer Science, TU Eindhoven, The Netherlands**
Institute for Computing and Information Sciences, Radboud University Nijmegen, The Netherlands
h.zantema@tue.nl

2 **Free University Amsterdam, The Netherlands**
joerg@few.vu.nl

Abstract

Streams are infinite sequences over a given data type. A stream specification is a set of equations intended to define a stream. In this paper we focus on equality of streams, more precisely, for a given set of equations two stream terms are said to be equal if they are equal in every model satisfying the given equations. We investigate techniques for proving equality of streams suitable for automation. Apart from techniques that were already available in the tool CIRC from Lucanu and Roşu, we also exploit well-definedness of streams, typically proved by proving productivity. Moreover, our approach does not restrict to behavioral input format and does not require termination.

We present a tool Streambox that can prove equality of a wide range of examples fully automatically.

Digital Object Identifier 10.4230/LIPIcs.RTA.2011.393

Category Regular Research Paper

1 Introduction

The stream ones specified by ones = 1 : ones is the infinite stream having the data element 1 on every position. The same holds for the stream c specified by $c = 1 : 1 : c$. So we expect that $c =$ ones holds. This paper is about how to prove such stream equalities fully automatically. More precisely, given a set of stream equalities (in this case ones = 1 : ones and $c = 1 : 1 : c$) that are assumed to hold, can we conclude validity of another given stream equality (in this case $c =$ ones)? The semantics of this question is quite natural: we have a basic data type D, typically the booleans or natural numbers, and streams are maps from natural numbers to D. The question now states whether if all constants and functions occurring in the equalities are interpreted by streams and stream functions in such a way that all given equalities hold, then also the goal equality holds.

An excellent basis for treating this problem is the circular co-induction principle as presented in [12, 5, 9, 10, 13, 7], and implemented in the tool CIRC. Indeed, several interesting stream equalities can be proved fully automatically by CIRC.

Both the basic circular co-induction principle and its extension using special contexts are the basis of the current paper. However, we offer several features that are not covered by CIRC and/or its underlying theory:

- We do not require the very specific format of behavioral equations in which the root of every left hand side should be hd or tl and the given equations should be terminating when applied only from left to right. Instead we allow any set of equations on streams. Our default format is the pure stream format, the format as used in [3, 14, 15]. In this format

22nd International Conference on Rewriting Techniques and Applications (RTA'11).
Editor: M. Schmidt-Schauß; pp. 393–408
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

ones is specified by the single equation ones = 1 : ones, where the similar specification in behavioral format consists of the two equations hd(ones) = 1 and tl(ones) = ones.

- We have a simple syntactic criterion for checking for special contexts: we introduce *guardedness* (Definition 4.4) and show that all contexts composed from symbols satisfying this guardedness, are special.
- In case an equality proof requires auxiliary lemmas, we do not need to enter these lemmas ourselves, but our tool Streambox generates and uses suitable lemmas fully automatically.
- If a particular stream is uniquely defined by a set of equations, and some term satisfies these equations, we may conclude that that term is equal to that stream. In this way we may exploit earlier techniques to prove that a stream is uniquely defined, for instance by proving productivity. In case this is appropriate, our tool Streambox calls a current tool for proving context-sensitive termination for proving productivity, typically succeeding for equations that are non-terminating themselves. In this way state-of-the-art termination tools are exploited for proving stream equality.

Combining these features, Streambox can prove several non-trivial equalities fully automatically, like the property that the Toeplitz stream is the boolean derivative of the Thue-Morse stream, as we shall see in Example 6.4.

The tool CIRC is based on the rewriting engine Maude, in particular using rewriting to normal form as the mechanism to check convertibility. However, for this mechanism the equations are only used in one direction, and the approach typically fails if this rewriting is non-terminating.

Only rewriting in one direction gives undesirable restrictions. For instance, if a stream function f is defined by $f(\sigma) = 0 : f(\sigma)$, and a stream a is defined by $a = f(a)$, then a will coincide with $0 : a$ as is very easily shown by the following conversion:

$$a = f(a) = 0 : f(a) = 0 : a.$$

However, in this conversion the equality $a = f(a)$ is used in both directions, so plain rewriting fails to derive this very simple equality. Polishing the rules, for instance by completion, could be helpful but also has limitations. Applying the rule $a \to f(a)$ will cause non-termination, and $f(a) \to a$ will cause critical pairs with the defining rule for f. In this very simple example Knuth-Bendix completion of the equations will succeed, but for many other examples, for instance involving commutativity, this does not hold. Moreover, in a typical scenario the set of equations for which convertibility of two terms has to be tested is not one static set, but is extended all time by co-induction hypotheses and proved goals and lemmas, for which ongoing Knuth-Bendix completion would be problematic.

So instead of forcing to do rewriting in only one direction and requiring termination, a crucial building block of our approach is a check for general convertibility without these restrictions. Accordingly, we developed our own tooling Streambox from scratch.

A typical example for which our tool Streambox succeeds is in proving $M = 0 : M$, where M is specified by both $M = f(M)$ and $M = g(M)$, for f and g defined mutually recursively by

$$f(x : \sigma) = 0 : g(\sigma), \quad g(x : \sigma) = x : f(\sigma).$$

Here x is a boolean variable and σ is a stream variable. Indeed, f replaces every symbol on an even position by 0, and g does the same for odd positions, so if $M = f(M)$ and $M = g(M)$ then indeed M has to consist only of zeros, by which $M = 0 : M$ should hold. Our goal is that such a property can be proved fully automatically after entering only these equations. Extensive experiments show failure of CIRC on all variants we could think of:

keeping the M-rules as they are will yield non-termination, while reversing them yields non-unique normal forms caused by combinations of rules like $f(M) = M$ and $\mathsf{hd}(f(\sigma)) = 0$.

Conceptually, our notions of circular co-induction and the use of special contexts is the same as in [9, 10, 13, 7]. However, we do not restrict to the behavioral format as required there, by which the underlying definitions look quite different. As streams have a natural semantics as mappings from natural numbers to data, it is natural to relate the notions of validity to this semantics. We succeeded in presenting the theory for arbitrary sets of stream equations based on this semantics. In this paper both the validity of the basic circular co-induction principle (Theorem 3.1) and its extension to special contexts (Theorem 4.2) is proved directly by induction on natural numbers, making the theory self-contained and independent of [9, 10, 13, 7] or any theory of co-induction. More abstract variants of Theorem 3.1 and Theorem 4.2 are given in [10, 13]; there they are proved in a more abstract setting in which it is left implicit that streams are a valid instance.

Most of our theory easily generalizes to other infinite data types, like infinite binary trees. However, in order to keep notations simple we chose to focus on streams. It turns out that involved underlying proof principles already appear in streams, and can be motivated by small boolean stream examples. Our theory is given for streams over arbitrary data types (specified by finite constructor ground terms). In the implementation and most of the examples we restrict to the boolean case in which 0 and 1 are the only data elements. Example 3.4 shows how our theory can be applied for stream over natural numbers.

This paper is structured as follows. In Section 2 we present the basic setting and its semantics. In Section 3 we present the basic version of the principal of circular co-induction. In Section 4 we extend this to special contexts and investigate how to recognize special contexts when given in pure stream format, the standard format as used in [3, 14, 15]. In Section 5 we present how to prove equality by using that a stream or stream function is uniquely defined, typically to be proved by proving productivity. Next, in Section 6 we discuss our implementation. We conclude in Section 7.

2 Streams: Specifications and Models

In stream specifications we have two sorts: s (stream) and d (data). We assume the set D of data elements to consist of ground terms $\mathbf{T}(\Sigma_d)$ over some signature Σ_d of which all symbols are of type $d^n \to d$ for some $n \geq 0$. We will focus on the boolean case where $D = \Sigma_d = \{0, 1\}$. We assume three particular symbols:

- : having type $d \times s \to s$;
- hd having type $s \to d$;
- tl having type $s \to s$.

Apart from the symbols in $\Sigma_d \cup \{:, \mathsf{hd}, \mathsf{tl}\}$ there is a set Σ of user defined symbols, each being of type $d^n \times s^m \to s$ or $d^n \times s^m \to d$ for $n, m \geq 0$.

We assume a set $\mathbf{X}_s$ of variables of sort s and a set $\mathbf{X}_d$ of variables of sort d. Using all these ingredients we can build (well-sorted) terms:

- x is a term of sort d if $x \in \mathbf{X}_d$,
- σ is a term of sort s if $\sigma \in \mathbf{X}_s$,
- if $u_1 \ldots, u_n$ are terms of sort d, and $t_1 \ldots, t_m$ of sort s, then $f(u_1 \ldots, u_n, t_1 \ldots, t_m)$ is a term of sort d if $f \in \Sigma \cup \Sigma_d \cup \{\mathsf{hd}\}$ has type $d^n \times s^m \to d$, and a term of sort s if $f \in \Sigma \cup \{:, \mathsf{tl}\}$ has type $d^n \times s^m \to s$.

The set of all such terms is denoted by $\mathbf{T}(\Sigma \cup \Sigma_d \cup \{:, \mathsf{hd}, \mathsf{tl}\}, \mathbf{X}_d, \mathbf{X}_s)$.

As a notational convention variables of sort d will be denoted by x, y, terms of sort d by u, u_i, variables of sort s by σ, τ, and terms of sort s by t, t_i.

▶ **Definition 2.1.** An *equation* is a pair $(\ell, r) \in \mathbf{T}(\Sigma \cup \Sigma_d \cup \{:, \mathsf{hd}, \mathsf{tl}\}, \mathbf{X}_d, \mathbf{X}_s)^2$ such that ℓ and r are of the same sort.

An equation (ℓ, r) is usually written as $\ell = r$.

A *stream specification* is defined to be a set of equations.

▶ **Example 2.2.** For specifying the Thue-Morse stream morse as extensively studied e.g. in [1], we have $D = \Sigma_d = \{0, 1\}$. There are two equations of sort d:

$$\mathsf{not}(0) = 1, \quad \mathsf{not}(1) = 0,$$

and the following equations of sort s:

$$\begin{array}{rclcrcl} \mathsf{morse} & = & 0 : \mathsf{zip}(\mathsf{inv}(\mathsf{morse}), \mathsf{tl}(\mathsf{morse})) & \qquad & \mathsf{tl}(x : \sigma) & = & \sigma \\ \mathsf{inv}(x : \sigma) & = & \mathsf{not}(x) : \mathsf{inv}(\sigma) & & \mathsf{zip}(x : \sigma, \tau) & = & x : \mathsf{zip}(\tau, \sigma) \end{array}$$

Stream specifications are intended to specify streams for the constants in Σ of sort s, and stream functions for the other elements of Σ of sort s. Similarly, elements of Σ of sort d should specify data elements and data functions. The combination of these streams and functions is what we will call a *stream model*.

More precisely, a *stream* over D is a map from the natural numbers to D. Write $S = D^\omega$ for the set of all streams over D. In case of $D = \emptyset$ we have $S = \emptyset$; in case of $\#D = 1$ we have $\#S = 1$. So in non-degenerate cases we have $\#D \geq 2$.

▶ **Definition 2.3.** A *stream model* over $\Sigma_d \cup \Sigma$ is defined to consist of the set $S = D^\omega$ and a set of functions and constants $[f]$ for every $f \in \Sigma$, where $[f] : D^n \times S^m \to S$ if the type of $f \in \Sigma$ is $d^n \times s^m \to s$, and $[f] : D^n \times S^m \to D$ if the type of $f \in \Sigma$ is $d^n \times s^m \to d$.

Apart from the functions and constants $[f]$ for every $f \in \Sigma$, we also have $[f]$ for every $f \in \Sigma_d \cup \{:, \mathsf{hd}, \mathsf{tl}\}$, defined as follows:

- $[f](u_1, \ldots, u_n) = f(u_1, \ldots, u_n)$ for $f \in \Sigma_d$ of type $d^n \to d$, $u_1, \ldots, u_n \in \mathbf{T}(\Sigma_d)$, for $n \geq 0$.
- For $u \in \mathbf{T}(\Sigma_d)$ and $s \in S$ the stream $[:](u, s)$ is defined by

  $$[:](u, s)(0) = u, \quad [:](u, s)(n + 1) = s(n)$$

  for every $n \geq 0$.
- $[\mathsf{hd}](s) = s(0)$ for $s \in S$.
- For $s \in S$ the stream $[\mathsf{tl}](s)$ is defined by

  $$[\mathsf{tl}](s)(n) = s(n + 1)$$

  for every $n \geq 0$.

A *variable assignment* α is defined to be a map $\alpha : \mathbf{X}_d \cup \mathbf{X}_s \to D \cup S$ such that $\alpha(x) \in D$ for $x \in \mathbf{X}_d$ and $\alpha(\sigma) \in S$ for $\sigma \in \mathbf{X}_s$.

For $t \in \mathbf{T}(\Sigma \cup \Sigma_d \cup \{:, \mathsf{hd}, \mathsf{tl}\}, \mathbf{X}_d, \mathbf{X}_s)$ and a variable assignments $\alpha : \mathbf{X}_d \cup \mathbf{X}_s \to D \cup S$ the *stream interpretation* $[t, \alpha]$ in the stream model $(S, ([f])_{f \in \Sigma})$ is defined inductively by:

$$\begin{array}{rcll} [x, \alpha] & = & \alpha(x) & \text{for } x \in \mathbf{X}_d \\ [\sigma, \alpha] & = & \alpha(\sigma) & \text{for } \sigma \in \mathbf{X}_s \\ [f(u_1, \ldots, u_n, t_1, \ldots, t_m), \alpha] & = & [f]([u_1, \alpha], \ldots, [u_n, \alpha], [t_1, \alpha], \ldots, [t_m, \alpha]) & \end{array}$$

for all terms $u_1, \ldots, u_n$ of sort d and all terms $t_1, \ldots, t_m$ of sort s, and every $f \in \Sigma \cup \Sigma_d \cup \{:, \mathsf{hd}, \mathsf{tl}\}$ of type $d^n \times s^m \to d$ or $d^n \times s^m \to s$.

In case t is a ground term then $[t, \alpha]$ does not depend on α, and we simply write $[t]$ rather than $[t, \alpha]$.

▶ **Definition 2.4.** An equation $\ell = r$ is defined to *hold* in the stream model, or the stream model *satisfies* the equation $\ell = r$, if $[\ell, \alpha] = [r, \alpha]$ for every variable assignment α. A stream model is said to *satisfy a set of equations* if it satisfies each of its equations.

So summarizing: a stream model consists of streams, and interpretations of function symbols as functions on streams. For the symbols $:, \mathsf{hd}, \mathsf{tl}$ the interpretations are predefined; for the other symbols the choice is free. As the interpretations of $:, \mathsf{hd}, \mathsf{tl}$ are fixed it is expected that some corresponding equations hold in every model. Indeed this is the case. Let $\mathbf{E}_{ht}$ be the set of equations:

$$\mathbf{E}_{ht} = \left\{ \begin{array}{rcl} \mathsf{hd}(x : \sigma) & = & x \\ \mathsf{tl}(x : \sigma) & = & \sigma \\ \mathsf{hd}(\sigma) : \mathsf{tl}(\sigma) & = & \sigma \end{array} \right.$$

▶ **Lemma 2.5.** *Every stream model satisfies* $\mathbf{E}_{ht}$.

Proof. We have to check that all three equations of $\mathbf{E}_{ht}$ hold in every stream model. Using the definitions of $[:], [\mathsf{hd}], [\mathsf{tl}]$ we obtain

$$[\mathsf{hd}(x : \sigma), \alpha] = [\mathsf{hd}]([x : \sigma, \alpha]) = [x : \sigma, \alpha](0)[:]([x, \alpha], [\sigma, \alpha])(0) = [x, \alpha],$$

proving that the first equation holds, and

$$[\mathsf{tl}(x : \sigma), \alpha](i) = [\mathsf{tl}]([x : \sigma, \alpha])(i) = [x : \sigma, \alpha](i + 1) = [:]([x, \alpha], [\sigma, \alpha])(i + 1) = [\sigma, \alpha](i)$$

for every $i \geq 0$, proving that the second equation holds. For the last equation we have to prove that $[\mathsf{hd}(\sigma) : \mathsf{tl}(\sigma), \alpha](i) = [\sigma, \alpha](i)$ for every $i \geq 0$. For $i = 0$ this holds since

$$[\mathsf{hd}(\sigma) : \mathsf{tl}(\sigma), \alpha](0) = [:]([\mathsf{hd}(\sigma), \alpha], [\mathsf{tl}(\sigma), \alpha])(0) = [\mathsf{hd}(\sigma), \alpha] = [\sigma, \alpha](0);$$

for $i > 0$ this holds since

$$[\mathsf{hd}(\sigma) : \mathsf{tl}(\sigma), \alpha](i) = [:]([\mathsf{hd}(\sigma), \alpha], [\mathsf{tl}(\sigma), \alpha])(i) = [\mathsf{tl}(\sigma), \alpha](i - 1) = [\sigma, \alpha](i).$$

◀

Now we arrive at the central notion of this paper.

▶ **Definition 2.6.** An equation $\ell = r$ is defined to *hold* with respect to a stream specification E, notation $E \models \ell = r$, if $\ell = r$ holds in every stream model satisfying E.

A set E' of equations is defined to *hold* with respect to a stream specification E, notation $E \models E'$, if $E \models \ell = r$ for every $\ell = r \in E'$.

For a stream specification E we write $=_E$ for the congruence generated by E, that is, the closure of E with respect to reflexivity, symmetry, transitivity, contexts and substitution. Only using $=_E$ is called *equational reasoning*.

We obviously have the following theorem.

▶ **Theorem 2.7.** *For every equation* $\ell = r$ *satisfying* $\ell =_{E \cup \mathbf{E}_{ht}} r$ *we have* $E \models \ell = r$.

The converse of Theorem 2.7 is not true: typically $E \models \ell = r$ may hold while $\ell =_{E \cup \mathbf{E}_{ht}} r$ does not hold. For instance, if E consists of the two equations $c = 0 : c$ and $d = 0 : d$, then in every model both $[c]$ and $[d]$ are equal to stream only consisting of zeros, so $E \models c = d$. However, to prove this equational reasoning is not sufficient: $c =_{E \cup \mathbf{E}_{ht}} d$ does not hold.

3 Basic circular co-induction

A main goal of this paper is to extend Theorem 2.7 to more powerful syntactic techniques suitable for implementation for concluding $E \models \ell = r$, which are still based on equational reasoning. As long as the elements of the intended models can be interpreted as finite terms, *inductive theorem proving* is the standard approach for this kind of questions. However, here we deal with streams, that can be seen as infinite terms, and for which we need *co-induction* rather than induction.

For expressing the versions of the co-induction principle as we consider them, we need a fresh *freeze* symbol fr of type $s \to d$. This freeze symbol is used to force that the co-induction hypothesis is only used on top level, and not on subterms. This idea originates from [6], in which square brackets are used as the notation for the freeze operator.

For a set E' of equations of sort s we write $\mathsf{fr}(E')$ for the set of equations $\mathsf{fr}(\ell) = \mathsf{fr}(r)$ for $\ell = r \in E'$. The following theorem can be seen as an instance of Theorem 2 in [13]; for completeness we give a full proof based on our stream semantics and independent of the more abstract and general setting of [13].

▶ **Theorem 3.1** (Basic circular co-induction). *Let E be a stream specification and E' a set of equations of sort s, both not containing the symbol fr, such that*

- $E \models \mathsf{hd}(\ell) = \mathsf{hd}(r)$ *for all* $\ell = r \in E'$, *and*
- $E \cup \mathsf{fr}(E') \models \mathsf{fr}(\mathsf{tl}(\ell)) = \mathsf{fr}(\mathsf{tl}(r))$ *for all* $\ell = r \in E'$.

Then $E \models E'$.

Proof. Take an arbitrary stream model satisfying E; we have to prove that for all $\ell = r \in E'$ we have $[\ell, \alpha] = [r, \alpha]$ for every α, that is, $[\ell, \alpha](n) = [r, \alpha](n)$ for every $n \geq 0$. We do this by induction on n.

For $n = 0$ take $\ell = r \in E'$. We use the property $E \models \mathsf{hd}(\ell) = \mathsf{hd}(r)$ from which we conclude $[\mathsf{hd}(\ell), \alpha)] = [\mathsf{hd}(r), \alpha]$. We obtain $[\ell, \alpha](0) = [\mathsf{hd}(\ell), \alpha)] = [\mathsf{hd}(r), \alpha] = [r, \alpha](0)$.

For $n > 0$ we assume $[\ell, \alpha](n-1) = [r, \alpha](n-1)$ for every α and every $\ell = r \in E'$ as the induction hypothesis. For every $\ell = r \in E'$ we have to prove $[\ell, \alpha](n) = [r, \alpha](n)$, again for every α. As fr does neither occur in E nor in $\ell = r$, both the assumption that the model satisfies E and the induction hypothesis are independent of the symbol fr, and we are still free to define $[\mathsf{fr}]$. Let us define $[\mathsf{fr}](s) = s(n-1)$ for every stream s. Using the induction hypothesis, then for every α we have

$$[\mathsf{fr}(\ell), \alpha] = [\ell, \alpha](n-1) = [r, \alpha](n-1) = [\mathsf{fr}(r), \alpha],$$

so our model satisfies $\mathsf{fr}(\ell) = \mathsf{fr}(r)$ for all $\ell = r \in E'$. So from $E \cup \mathsf{fr}(E') \models \mathsf{fr}(\mathsf{tl}(\ell)) = \mathsf{fr}(\mathsf{tl}(r))$ we conclude that $[\mathsf{fr}(\mathsf{tl}(\ell)), \alpha] = [\mathsf{fr}(\mathsf{tl}(r)), \alpha]$, for all $\ell = r \in E'$. We obtain

$$[\ell, \alpha](n) = [\mathsf{fr}(\mathsf{tl}(\ell)), \alpha] = [\mathsf{fr}(\mathsf{tl}(r)), \alpha] = [r, \alpha](n),$$

concluding the proof. ◀

The set $\mathsf{fr}(E')$ in the second requirement of Theorem 3.1 is called the *co-induction hypothesis*.

▶ **Example 3.2.** Let E consists of the two equations $c = 0 : c$ and $d = 0 : d$. Then $E \models c = d$ is proved using Theorem 3.1 as follows. Choose $E' = \{c = d\}$. Here and in the following in equational reasoning we will abbreviate $=_{E \cup \mathbf{E}_{ht}}$ and $=_{E \cup \mathbf{E}_{ht} \cup \mathsf{fr}(E')}$ to $=$.

The first requirement $E \models \mathsf{hd}(c) = \mathsf{hd}(d)$ follows from

$$\mathsf{hd}(c) = \mathsf{hd}(0 : c) = 0 = \mathsf{hd}(0 : d) = \mathsf{hd}(d).$$

The second requirement follows from

$$\mathsf{fr}(\mathsf{tl}(c)) = \mathsf{fr}(\mathsf{tl}(0:c)) = \mathsf{fr}(c) = \mathsf{fr}(d) = \mathsf{fr}(\mathsf{tl}(0:d)) = \mathsf{fr}(\mathsf{tl}(d)),$$

using the co-induction hypothesis $\mathsf{fr}(c) = \mathsf{fr}(d)$. So Theorem 3.1 implies $E \models c = d$.

The basic machinery of our approach consists of combining Theorem 2.7 and Theorem 3.1: for proving $E \models \ell = r$ first it is tried to prove $\ell =_{E \cup \mathbf{E}_{ht}} r$. If this succeeds we are done, otherwise Theorem 3.1 is tried for $E' = \{\ell = r\}$. For the proof obligations of Theorem 3.1 again first Theorem 2.7 is tried, as was successful in Example 3.2. Where this fails, the failing proof obligation is added to E' after removing the fr symbols, and using this extended E' Theorem 3.1 is tried. This may be repeated any number of times. In a typical scenario in this way a finite set E' is found for which Theorem 3.1 applies. In this way not only validity of $\ell = r$ is proved, but also of any rule in E'.

▶ **Example 3.3.** Let E consist of the equations

$$\begin{array}{rclcrcl} \mathsf{zeros} & = & 0 : \mathsf{zeros} & & \mathsf{alt} & = & 0 : 1 : \mathsf{alt} \\ \mathsf{ones} & = & 1 : \mathsf{ones} & \mathsf{zip}(x:\sigma,\tau) & = & & x : \mathsf{zip}(\tau,\sigma). \end{array}$$

Then $E \models \mathsf{alt} = \mathsf{zip}(\mathsf{zeros}, \mathsf{ones})$ is proved using Theorem 3.1 as follows. First choose $E' = \{\mathsf{alt} = \mathsf{zip}(\mathsf{zeros}, \mathsf{ones})\}$. The first requirement follows from

$$\mathsf{hd}(\mathsf{alt}) = \mathsf{hd}(0:1:\mathsf{alt}) = 0 = \mathsf{hd}(0:\mathsf{zip}(\mathsf{ones},\mathsf{zeros}) =$$

$$\mathsf{hd}(\mathsf{zip}(0:\mathsf{zeros},\mathsf{ones}) = \mathsf{hd}(\mathsf{zip}(\mathsf{zeros},\mathsf{ones}).$$

For the second requirement we need $\mathsf{fr}(\mathsf{tl}(\mathsf{alt})) = \mathsf{fr}(\mathsf{tl}(\mathsf{zip}(\mathsf{zeros},\mathsf{ones})))$, for which equational reasoning using the co-induction hypothesis fails. So we add the equation $\mathsf{tl}(\mathsf{alt}) = \mathsf{tl}(\mathsf{zip}(\mathsf{zeros},\mathsf{ones}))$ to E' and try again to apply Theorem 3.1. By this addition now we obtain the requirements for the original equation in E' for free, and only need to consider the new equation. Now equational reasoning yields $\mathsf{hd}(\mathsf{tl}(\mathsf{alt})) = 1 = \mathsf{hd}(\mathsf{tl}(\mathsf{zip}(\mathsf{zeros},\mathsf{ones})))$, and for the second requirement we use the co-induction hypothesis $\mathsf{fr}(\mathsf{alt}) = \mathsf{fr}(\mathsf{zip}(\mathsf{zeros},\mathsf{ones}))$ to obtain

$$\mathsf{fr}(\mathsf{tl}(\mathsf{tl}(\mathsf{alt}))) = \mathsf{fr}(\mathsf{alt}) = \mathsf{fr}(\mathsf{zip}(\mathsf{zeros},\mathsf{ones})) = \mathsf{fr}(\mathsf{tl}(\mathsf{tl}(\mathsf{zip}(\mathsf{zeros},\mathsf{ones})))),$$

concluding the proof.

▶ **Example 3.4.** Consider streams over the naturals, that is, Σ_d consists of a constant 0 and a unary symbol s representing successor. Let E consist of the equations

$$\begin{array}{rclcrcl} \mathsf{from}(x) & = & x : \mathsf{from}(s(x)) & & & & \\ \mathsf{from2}(x) & = & x : \mathsf{from2}(s(s(x))) & \quad \mathsf{zip}(x:\sigma,\tau) & = & & x : \mathsf{zip}(\tau,\sigma). \end{array}$$

We will prove $E \models \mathsf{from}(x) = \mathsf{zip}(\mathsf{from2}(x), \mathsf{from2}(s(x)))$ using Theorem 3.1. Choose $E' = \{\mathsf{from}(x) = \mathsf{zip}(\mathsf{from2}(x), \mathsf{from2}(s(x)))\}$. The first requirement follows from

$$\mathsf{hd}(\mathsf{from}(x)) = \mathsf{hd}(x:\mathsf{from}(s(x))) = x = \mathsf{hd}(x:\mathsf{zip}(\mathsf{from2}(s(x)),\mathsf{from2}(s(s(x))))) =$$

$$\mathsf{hd}(\mathsf{zip}(x:\mathsf{from2}(s(s(x))),\mathsf{from2}(s(x)))) = \mathsf{hd}(\mathsf{zip}(\mathsf{from2}(x),\mathsf{from2}(s(x)))).$$

The second requirement follows from

$$\mathsf{fr}(\mathsf{tl}(\mathsf{from}(x))) = \mathsf{fr}(\mathsf{tl}(x:\mathsf{from}(s(x)))) = \mathsf{fr}(\mathsf{from}(s(x))) =_{\mathsf{fr}(E')}$$

$$\mathsf{fr}(\mathsf{zip}(\mathsf{from2}(s(x)),\mathsf{from2}(s(s(x))))) = \mathsf{fr}(\mathsf{tl}(x:\mathsf{zip}(\mathsf{from2}(s(x)),\mathsf{from2}(s(s(x)))))) =$$

$$\mathsf{fr}(\mathsf{tl}(\mathsf{zip}(x:\mathsf{from2}(s(s(x))),\mathsf{from2}(s(x))))) = \mathsf{fr}(\mathsf{tl}(\mathsf{zip}(\mathsf{from2}(x),\mathsf{from2}(s(x))))),$$

by which the claim has been proved by Theorem 3.1. In particular, for $x = 0$ this yields that the stream $\mathsf{from}(0)$ of natural numbers is the zip of the stream $\mathsf{from2}(0)$ of even numbers and the stream $\mathsf{from2}(s(0))$ of odd numbers.

More complicated examples over the natural numbers typically will require a combination of circular co-induction and induction over the natural numbers.

4 Special contexts

Next we present a powerful generalization of Theorem 3.1: instead of only assuming $\mathsf{fr}(\ell) = \mathsf{fr}(r)$ we may also assume $\mathsf{fr}(C[\ell]) = \mathsf{fr}(C[r])$ for *special contexts* C, where roughly speaking for every n the n-th element of the stream represented by $C[s]$ only depends on the first n elements of the stream represented by s. This notion originates from [6, 10]. As conceptually our notion is the same, we keep the terminology *special context* as introduced there.

▶ **Definition 4.1.** A context C of sort s and the hole of sort s is called *special* with respect to a stream specification, if for every stream model, for every α, every $n \geq 0$, and every pair of terms t, t' of sort s the following holds:

If $[t,\alpha](i) = [t',\alpha](i)$ for all $i \leq n$, then $[C[t],\alpha](n) = [C[t'],\alpha](n)$.

For example, the empty context $\Box$ is special; as a consequence of Theorem 4.5 we will conclude that $\mathsf{zip}(\Box,\sigma)$ and $\mathsf{zip}(\sigma,\Box)$ and $\mathsf{inv}(\Box)$ are special contexts with respect to the stream specification from Example 2.2.

As the first element of $\mathsf{tl}(t)$ is the second element of t, we observe that the context $\mathsf{tl}(\Box)$ is not special. In Example 4.3 we will show that $\mathsf{even}(\Box)$ is not a special context.

The requirement of a context C to be special is that $[C[\cdot],\alpha]$ is a *causal function* on streams, that is, for every n the the first n elements of the output stream only depend on the first n elements of the input stream. Such causal functions have been studied co-algebraically, e.g., in [8].

Since by definition the empty context is a special context, Theorem 3.1 is a direct consequence of the following theorem, which can be seen as an instance of Theorem 3 in [10]. For completeness we give a full proof based on our stream semantics and our corresponding definition of special context.

▶ **Theorem 4.2** (Circular co-induction with special contexts)**.** *Let E be a stream specification and E' a set of equations of sort s, both not containing the symbol fr, such that*

- $E \models \mathsf{hd}(\ell) = \mathsf{hd}(r)$ *for all* $\ell = r \in E'$*, and*
- $E \cup \{\mathsf{fr}(C[\ell]) = \mathsf{fr}(C[r]) \mid C$ *is a special context,* $\ell = r \in E'\} \models \mathsf{fr}(\mathsf{tl}(\ell)) = \mathsf{fr}(\mathsf{tl}(r))$ *for all* $\ell = r \in E'$*.*

Then $E \models E'$*.*

Proof. Take an arbitrary stream model satisfying E; we have to prove that $[\ell,\alpha] = [r,\alpha]$ for every α and every $\ell = r \in E'$, that is, $[\ell,\alpha](n) = [r,\alpha](n)$ for every $n \geq 0$. We do this by induction on n.

For $n = 0$ we use the property $E \models \mathsf{hd}(\ell) = \mathsf{hd}(r)$ from which we conclude $[\mathsf{hd}(\ell),\alpha)] = [\mathsf{hd}(r),\alpha]$. We obtain $[\ell,\alpha](0) = [\mathsf{hd}(\ell),\alpha)] = [\mathsf{hd}(r),\alpha] = [r,\alpha](0)$.

For $n > 0$ we assume as the induction hypothesis $[\ell,\alpha](i) = [r,\alpha](i)$ for every α and every $i < n$, for all $\ell = r \in E'$, and we have to prove $[\ell,\alpha](n) = [r,\alpha](n)$, again for every α and

$\ell = r \in E'$. As fr does neither occur in E nor in E', both the assumption that the model satisfies E and the induction hypothesis are independent of the symbol fr, and we are still free to define $[\mathsf{fr}]$. Just like in the proof of Theorem 3.1 define $[\mathsf{fr}](s) = s(n-1)$ for every stream s. Let C be any special context. Using the induction hypothesis and the definition of special context, for every α and $\ell = r \in E'$ we obtain

$$[\mathsf{fr}(C[\ell], \alpha] = [C[\ell], \alpha](n-1) = [C[r], \alpha](n-1) = [\mathsf{fr}(C[l], \alpha],$$

so our model satisfies $\mathsf{fr}(C[\ell]) = \mathsf{fr}(C[r])$. As this holds for every special context and $\ell = r \in E'$, and the model still satisfies E, from the condition of the theorem we conclude that the model also satisfies $\mathsf{fr}(\mathsf{tl}(\ell)) = \mathsf{fr}(\mathsf{tl}(r))$ for any $\ell = r \in E'$. Hence we obtain

$$[\ell, \alpha](n) = [\mathsf{fr}(\mathsf{tl}(\ell)), \alpha] = [\mathsf{fr}(\mathsf{tl}(r)), \alpha] = [r, \alpha](n),$$

concluding the proof. ◀

▶ **Example 4.3.** We start by a negative example. Let E consists of the four equations

$$\mathsf{even}(x : \sigma) = x : \mathsf{odd}(\sigma), \quad \mathsf{odd}(x : \sigma) = \mathsf{even}(\sigma), \quad c = 0 : \mathsf{even}(c), \quad d = 0 : \mathsf{even}(d).$$

We now will show that $\mathsf{even}(\Box)$ is not a special context. Consider two streams: one only consisting of zeros, and the other starting by two zeros, and followed by only ones. Then both satisfy the equations for c and d, so we conclude $E \not\models c = d$.

Next assume that $\mathsf{even}(\Box)$ is a special context. The first condition of Theorem 4.2 for proving $E \models c = \mathsf{zeros}$ is easily checked:

$$\mathsf{hd}(c) = \mathsf{hd}(0 : \mathsf{even}(c)) = 0 = \mathsf{hd}(0 : \mathsf{even}(d)) = \mathsf{hd}(d).$$

Using the co-induction hypothesis $\mathsf{fr}(\mathsf{even}(c)) = \mathsf{fr}(\mathsf{even}(d))$ also the second condition holds:

$$\mathsf{fr}(\mathsf{tl}(c)) = \mathsf{fr}(\mathsf{tl}(0 : \mathsf{even}(c))) = \mathsf{fr}(\mathsf{even}(c)) = \mathsf{fr}(\mathsf{even}(d)) = \mathsf{fr}(\mathsf{tl}(0 : \mathsf{even}(d))) = \mathsf{fr}(\mathsf{tl}(d)).$$

So by Theorem 4.2 we would conclude $E \models c = d$, contradicting the assumption that $\mathsf{even}(\Box)$ is a special context.

Theorem 4.2 becomes useful if we have a syntactic criterion to check whether a particular context is special. A suitable criterion is *friendly nestingness* as was introduced in [3] for establishing productivity. The underlying idea is that when using equations from left to right, at each step at most one stream element is consumed and at least one element is produced. Although the underlying idea is the same, for our purpose we need less conditions, for instance, we do not require orthogonality. In other settings this idea is also called *guardedness*, e.g., in 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.

To avoid confusion with the more restricted notion of friendly nestingness, we prefer to call it guardedness. Where the rest of this paper allows arbitrary equations, this guardedness criterion for detecting special contexts is the only spot where we focus on the pure stream format as given in [3, 14, 15].

▶ **Definition 4.4.** A stream specification E over $\Sigma \cup \Sigma_d \cup \{:, \mathsf{hd}, \mathsf{tl}\}$ is called *guarded* if no symbol of sort d in Σ has an argument of sort s, and all equations $\ell = r$ of E of sort s satisfy

- the symbols hd and tl do not occur in ℓ and r,
- r is not a variable, and the root of r is $:$,

- ℓ is of the shape $f(u_1 \ldots, u_n, t_1 \ldots, t_m)$ such that either $t_i \in \mathbf{X}_s$ or $t_i = x : \sigma$ for some $x \in \mathbf{X}_d, \sigma \in \mathbf{X}_s$, for every $i = 1, \ldots, m$,

and E is *exhaustive*, that is, for every term $t = f(u_1 \ldots, u_n, t_1 \ldots, t_m)$ for which $u_i \in D$ for $i = 1, \ldots, n$ and for every $i = 1, \ldots, m$ the term t_i is of the shape $d : t'$ for $d \in D$, there is such an equation $\ell = r$ and a substitution ρ such that $t = \ell\rho$.

▶ **Theorem 4.5.** *Let a stream specification E over $\Sigma \cup \Sigma_d \cup \{:, \mathsf{hd}, \mathsf{tl}\}$ be given, and subsets $E' \subseteq E$ and $\Sigma' \subseteq \Sigma$ such that E' over $\Sigma' \cup \Sigma_d \cup \{:, \mathsf{hd}, \mathsf{tl}\}$ is a guarded stream specification. Then every context over $\Sigma' \cup \Sigma_d \cup \{:\} \cup \mathbf{X}_s \cup \mathbf{X}_d$ of sort s is special with respect to E.*

Proof. Fix a model for E, and an assignment α. We prove the theorem by proving the following claim by induction on n:

> **Claim:** If terms t, t' of sort s satisfy $[t, \alpha](i) = [t', \alpha](i)$ for all $i \leq n$, then $[C[t], \alpha](i) = [C[t'], \alpha](i)$ for all $i \leq n$, for every context C of sort s in which all symbols on the path from the root to the hole (of sort s) are in $\Sigma' \cup \{:\}$.

For $C = \Box$ being the empty context the claim is trivial. For a context in which the hole is deeper then the first level, the claim can be proved by repeatedly applying instances of the claim for contexts in which the hole is immediately below the root. So it remains to prove the claim for $C = f(u_1, \ldots, u_n, t_1, \ldots, t_m)$ in which the hole $\Box$ is one of the arguments $t_1, \ldots, t_m$, say $t_I = \Box$; it can not be in a data argument since we assumed that data symbols have no arguments of sort s. For proving $[C[t], \alpha](i) = [C[t'], \alpha](i)$ for $i \leq n$ it is obtained form the induction hypothesis for $i < n$, we only need to prove $[C[t], \alpha](n) = [C[t'], \alpha](n)$.

In case $f = :$ we have $C = u_1 : \Box$. For $n = 0$ we conclude $[C[t], \alpha](0) = [u_1, \alpha] = [C[t'], \alpha](0)$, and for $n > 0$ we conclude

$$[C[t], \alpha](n) = [t_1[t], \alpha](n-1) = [t_1[t'], \alpha](n-1) = [C[t'], \alpha](n)$$

by the induction hypothesis.

It remains to prove $[C[t], \alpha](n) = [C[t'], \alpha](n)$ for $C = f(u_1, \ldots, u_n, t_1, \ldots, t_m)$ for $f \in \Sigma'$ and $t_I = \Box$. For any term v we have

$$[C[v], \alpha] = [f(u_1, \ldots, u_n, t_1, \ldots, t_{I-1}, v, t_{I+1}, \ldots, t_m)]), \alpha].$$

Here every u_i may be replaced by the data element $[u_1, \alpha)]$, and every t_i may be replaced by $[t_i, \alpha](0) : \mathsf{tl}(t_i)), \alpha]$, since $[t_i, \alpha] = [[t_i, \alpha](0) : \mathsf{tl}(t_i), \alpha]$ by Lemma 2.5, and similar for v. Now the resulting term matches with ℓ for some equation $\ell = r$ in E', due to the definition of guardedness, where the root of r is $:$. Then $[C[v], \alpha] = [u : C'[\mathsf{tl}(v)]^*, \alpha]$ for some $u \in D$ and some context C' having zero or more holes, and every path from the root to a hole only contains symbols from $\Sigma' \cup \{:\}$. The '$*$' in $C'[\mathsf{tl}(v)]^*$ means that every hole is filled by $\mathsf{tl}(v)$. For $n = 0$ we obtain $[C[t], \alpha](0) = [u : \cdots, \alpha](0) = [C[t'], \alpha](0)$, for $n > 0$ we apply the induction hypothesis on C' and the two terms $\mathsf{tl}(t)$ and $\mathsf{tl}(t')$. The condition of the induction hypothesis for $n - 1$ is $[\mathsf{tl}(t), \alpha](i) = [\mathsf{tl}(t'), \alpha](i)$ for all $i \leq n - 1$, which follows from the assumption $[t, \alpha](i) = [t', \alpha](i)$ for all $i \leq n$. So by the induction hypothesis we conclude $C'[\mathsf{tl}(t)]^*, \alpha](n-1) = [C'[\mathsf{tl}(t')]^*, \alpha](n-1)$.

In case C' has 1 hole it is immediate, in case C' has k holes this is concluded by applying the induction hypothesis k times. We conclude

$$\begin{aligned}
[C[t], \alpha](n) &= [u : C'[\mathsf{tl}(t)]^*, \alpha](n) \\
&= [C'[\mathsf{tl}(t)]^*, \alpha](n-1) \\
&= [C'[\mathsf{tl}(t')]^*, \alpha](n-1) \\
&= [u : C'[\mathsf{tl}(t')]^*, \alpha](n) \\
&= [C[t'], \alpha](n),
\end{aligned}$$

concluding the proof. ◀

▶ **Example 4.6.** Let E consist of the two equations

$$\mathsf{zip}(x:\sigma,\tau) = x:\mathsf{zip}(\tau,\sigma),\quad \mathsf{ones} = 1:\mathsf{zip}(\mathsf{ones},\mathsf{ones}).$$

We want to prove that $E \models \mathsf{ones} = 1:\mathsf{ones}$. Applying the standard approach based on Theorem 3.1 turns out to fail: E' will be extended forever by terms containing an increasing number of zip symbols. Instead, using Theorem 4.2 easily applies: according to Theorem 4.5 $\mathsf{zip}(\Box,\sigma)$ is a special context, by which we may use the co-induction hypothesis $\mathsf{fr}(\mathsf{zip}(\mathsf{ones},\sigma)) = \mathsf{fr}(\mathsf{zip}(1:\mathsf{ones},\sigma))$:

$$\begin{array}{lllll} \mathsf{fr}(\mathsf{tl}(\mathsf{ones})) & = & \mathsf{fr}(\mathsf{tl}(1:\mathsf{zip}(\mathsf{ones},\mathsf{ones}))) & = & \mathsf{fr}(\mathsf{zip}(\mathsf{ones},\mathsf{ones})) \\ & = & \mathsf{fr}(\mathsf{zip}(1:\mathsf{ones},\mathsf{ones})) & = & \mathsf{fr}(1:\mathsf{zip}(\mathsf{ones},\mathsf{ones})) \\ & = & \mathsf{fr}(\mathsf{ones}) & = & \mathsf{fr}(\mathsf{tl}(1:\mathsf{ones})). \end{array}$$

For proving $E \models \mathsf{ones} = 1:\mathsf{ones}$ by Theorem 4.2 it remains to prove $\mathsf{hd}(\mathsf{ones}) = \mathsf{hd}(1:\mathsf{ones})$, which is straightforward.

5 Exploiting unicity

Several techniques have been developed to prove that a stream specification has a unique solution, many of which are based on the notion of *productivity*. These techniques can be used for proving equality: if we want to prove $E \models c = t$ for some stream constant c having a unique solution, then we can try to prove that t satisfies the equations for c, that is, $E \models \ell\downarrow = r\downarrow$ for every equation $\ell = r$ in E containing the symbol c, where $\downarrow$ means that every symbol c is replaced by t. If this is the case, then both $[c]$ and $[t]$ satisfy the equations for c in every model, and since this is unique we conclude $[c] = [t]$ in every model, so proving $E \models c = t$. In this section we describe this approach in detail, also for functions rather than only for constants c, and give an example for which all earlier techniques fail and we make use of recent techniques to prove productivity.

▶ **Definition 5.1.** A function symbol $f \in \Sigma$ is called *uniquely defined* with respect to a set E of equations if $[f]_1 = [f]_2$ for every two models $[.]_1$, $[.]_2$ satisfying E.

▶ **Theorem 5.2.** *Let E be a set of equations such that a symbol f is uniquely defined with respect to E. Let $t = t'$ be an equation for which f is the root of t, and all arguments of f in t are variables, all distinct. Moreover, all variables in t' occur in t, and the symbol f does not occur in t'. Write $\downarrow$ for the normal formal with respect to the single rule $t \to t'$. Assume $E \models \ell\downarrow = r\downarrow$ for all equations $\ell = r$ in E. Then $E \models t = t'$.*

Proof. Let $[.]_1$ be any model satisfying E; we have to prove that $[t,\alpha]_1 = [t',\alpha]_1$ for every variable assignment α. Define $[.]_2$ by $[g]_2 = [g]_1$ for every symbol $g \neq f$, and let $[f]_2$ interpret t', that is, if $t = f(x_1,\ldots,x_n,\sigma_1,\ldots,\sigma_m$, then $[f]_2(d_1,\ldots,d_n,s_1,\ldots,s_m) = [t',\alpha]_1$ for α defined by $\alpha(x_i) = d_i$, $\alpha(\sigma_i) = s_i$. So by definition we have $[t,\alpha]_2 = [t',\alpha]_1$ for every α. Using this, one proves by induction on the structure of v that $[v,\alpha]_2 = [v\downarrow,\alpha]_1$ for every term v and every α. Hence for every equation $\ell = r$ in E and every α we obtain

$$[\ell,\alpha]_2 = [\ell\downarrow,\alpha]_1 = [r\downarrow,\alpha]_1 = [r,\alpha]_2,$$

so the model $[.]_2$ satisfies E. Since f is uniquely defined we now conclude $[f]_1 = [f]_2$, so

$$[t,\alpha]_1 = [t,\alpha]_2 = [t',\alpha]_1$$

for every α, concluding the proof. ◀

▶ **Example 5.3.** Let E consist of the equations

$$\begin{array}{rclcrcl} a & = & 0 : f(a) & & f(0:\sigma) & = & 1 : 0 : f(\sigma) \\ \mathsf{alt} & = & 0 : 1 : \mathsf{alt} & & f(1:\sigma) & = & f(\sigma). \end{array}$$

We want to prove that $E \models a = \mathsf{alt}$. Earlier techniques fail for doing this automatically, but we can apply Theorem 5.2 for the term $t = a$. First we have to prove that a is uniquely defined. This follows from productivity of all ground terms, which is proved using Theorem 4.1 from [15]. To this end context-sensitive termination of E has to be proved for the instance of context-sensitive rewriting in which rewriting is allowed on all arguments of all symbols except for the second argument of ":". This is easily proved fully automatically by tools like AProVE [4] and μ-Term [11].

Then by Theorem 5.2 it remains to prove that $E \models \ell\downarrow = r\downarrow$ for all equations $\ell = r$ in E, where $\downarrow$ means replacement of a by alt. As there is only one equation containing the symbol a, we only have to prove $E \models \mathsf{alt} = 0 : f(\mathsf{alt})$. This is easily proved by our basic machinery based on Theorem 3.1; it can also be proved by CIRC.

It is possible to prove $E \models a = \mathsf{alt}$ directly by Theorem 3.1 by choosing

$$E' = \{a = \mathsf{alt}, f^n(a) = 1 : \mathsf{alt}, 0 : f^n(a) = \mathsf{alt} \mid n > 0\}.$$

However, since E' is infinite, and the argument requires to prove $f^n(a) = 1 : 0 : f^{n+1}(a)$ for every $n > 0$, this approach is not suitable for automation.

We want to stress here that productivity of all ground terms does not imply that all functions are uniquely defined on all streams, only on stream interpretations of ground terms. In this example $[f]$ is not uniquely defined for all streams: one can make two distinct models both satisfying E in which $[f](s)$ are distinct, for s being the stream purely consisting of ones.

6 Implementation

We have implemented the techniques as presented in this paper in a tool called 'Streambox'. More precisely, any stream specification with a corresponding goal can be entered, or chosen from a list, and then the tool tries to prove the goal. The basic machinery is equational reasoning in combination with circular co-induction. For equational reasoning the tool simply searches for a conversion, so does not require termination. The tool Streambox is available on-line, as well as for download:

`http://infinity.few.vu.nl/streambox/`

The tool Streambox automatically proves equality of all examples included in this paper.

In this section we describe some aspects of our implementation not yet covered by the theory presented so far, in particular the implementation of *special contexts*, the use of *case analysis*, and *automatic lemma search*. Streambox is not stand alone: for exploiting unicity by using Theorem 5.2 it is tried to prove unicity by proving productivity by proving context-sensitive termination as described in [15] This is done by calling μ-Term [11], and in case this fails by calling AProVE [4]. In this way the power of state-of-the art termination provers is exploited.

6.1 Special contexts

Theorem 4.2 extends circular co-induction with special contexts. Roughly speaking, the idea is that to derive the equality $\mathsf{fr}(\mathsf{tl}(\ell)) = \mathsf{fr}(\mathsf{tl}(r))$, we are allowed to use equations of the form $\mathsf{fr}(C[\ell]) = \mathsf{fr}(C[r])$ for every special context C, in particular for contexts using symbols for which the specification is guarded, as discussed in Theorem 4.5. For example, the special contexts for Example 2.2 include:

$$\mathsf{inv}(\Box),\quad \mathsf{inv}(\mathsf{inv}(\Box)),\quad \mathsf{inv}(\mathsf{inv}(\mathsf{inv}(\Box))),\dots$$
$$\mathsf{zip}(\Box, s),\quad \mathsf{zip}(\mathsf{inv}(\Box), s),\quad \mathsf{zip}(\mathsf{inv}(\mathsf{inv}\Box), s),\quad \mathsf{zip}(\mathsf{zip}(\Box, s), t),\dots$$

Mimicking the use of these infinitely many special contexts can be done by moving the symbol fr up and down, as is described in the following lemma of which the proof is straightforward:

▶ **Lemma 6.1.** *Let E be a stream specification and E' a set of equations of sort s, both not containing the symbol* fr.

Let $\Sigma' \subseteq \Sigma$ such that every context over $\Sigma' \cup \{:\} \cup \mathbf{X}_s \cup \mathbf{X}_d$ of sort s is special. For every symbol $f \in \Sigma'$, let $f_\#$ be a fresh symbol of the same type. We define the set $\mathcal{S}(\Sigma')$ to consist of the following equations:

$$\mathsf{fr}(f(x_1,\dots,x_m,\sigma_1,\dots,\sigma_m)) = f_\#(x_1,\dots,x_m,\mathsf{fr}(\sigma_1),\dots,\mathsf{fr}(\sigma_m))$$

for every $f \in \Sigma'$.

Assume that the following conditions hold:

- *$E \models \mathsf{hd}(\ell) = \mathsf{hd}(r)$ for all $\ell = r \in E'$, and*
- *$E \cup \{\mathsf{fr}(\ell) = \mathsf{fr}(r)\} \cup \mathcal{S}(\Sigma') \models \mathsf{fr}(\mathsf{tl}(\ell)) = \mathsf{fr}(\mathsf{tl}(r))$ for all $\ell = r \in E'$.*

Then $E \models E'$. ◀

Streambox derives the set Σ' according to Theorem 4.5, and then employs the rules given in Lemma 6.1 to treat special contexts.

6.2 Case analysis

Let us consider the following specification:

$$\mathsf{inv}(0:\sigma) \;=\; 1:\mathsf{inv}(\sigma),\quad \mathsf{inv}(1:\sigma) \;=\; 0:\mathsf{inv}(\sigma),$$

and prove the equation $\mathsf{inv}(\mathsf{inv}(\sigma)) = \sigma$. A direct application of the basic circular co-induction principle fails: it needs the observation that every boolean stream σ is either of the shape $\sigma = 0:\sigma'$ or $\sigma = 1:\sigma'$.

The following straightforward lemma exploits this case analysis:

▶ **Lemma 6.2** (Case analysis)**.** *Let E be a stream specification such that* 0 *and* 1 *are the only data constructors. Let $E' \cup \{\ell = r\}$ be a set of equations, and σ a stream variable occurring in $\ell = r$. Then $E \models E' \cup \{\ell = r\}$ if and only if*

$$E \;\models\; E' \cup \{\ell[\sigma \mapsto 0:\sigma] = r[\sigma \mapsto 0:\sigma]\} \cup \{\ell[\sigma \mapsto 1:\sigma] = r[\sigma \mapsto 1:\sigma]\}$$

Applied to the above specification, the proof obligation is

$$\begin{aligned} E \cup \{\mathsf{fr}(\mathsf{inv}(\mathsf{inv}(\sigma))) = \mathsf{fr}(\sigma)\} \;\models\; & \{\mathsf{fr}(\mathsf{tl}(\mathsf{inv}(\mathsf{inv}(0:\sigma)))) = \mathsf{fr}(\mathsf{tl}(0:\sigma))\} \cup \\ & \{\mathsf{fr}(\mathsf{tl}(\mathsf{inv}(\mathsf{inv}(1:\sigma)))) = \mathsf{fr}(\mathsf{tl}(1:\sigma))\} \end{aligned}$$

which is easily proved by equational reasoning.

In Streambox the focus is on boolean streams, and only this boolean version of case analysis has been implemented. For other data structures case analysis can be employed as well. For example, for streams over natural numbers one can distinguish the cases $0 : \sigma$ and $\mathsf{s}(x) : \sigma$.

6.3 Automatic lemma search

For many examples, circular co-induction does not suffice for deriving the goal equations directly from the input system. Then it frequently helps to first find some auxiliary lemmas, which can themselves be proved using circular co-induction. These lemmas then may be employed to prove further lemmas, or the goal equations:

▶ **Lemma 6.3** (Lemma usage). *Let E be a stream specification and E', E'' sets of equations. If $E \models E'$, then $E \models E''$ if and only if $E \cup E' \models E''$.*

Proof. Since $E \models E'$ it follows that E' is valid in every stream model where E is valid. Hence, the set of stream models of $E \cup E'$ coincides with that of E. ◀

Lemma 6.3 is used as follows: if earlier techniques fail to prove $E \models E''$, then a set E' of small equations (the lemmas) is tried to be created for which $E \models E'$ can be proved, again using our approach, see below for more details. After every extension of E' by a new equation, it is tried to prove $E \cup E' \models E''$, and as soon this succeeds we are done. In proving new lemmas in E', it is allowed also to use earlier lemmas in E'.

The advantage with respect to circular co-induction is that the lemmas E' enrich the conversion relation, and thereby can be fruitful for deriving the goal equations.

Our tool Streambox supports an automated search for lemmas in the following way. It enumerates small terms $t_1, t_2, \ldots$ of sort s by increasing 'weight'. For the *weight* of a term t we have chosen the number of function symbols in t minus ξ-times the number of distinct variables in t (where $0 < \xi \leq 0.1$). The intention of this weight function is that the most general lemmas are encountered first. For example, it guarantees that $\mathsf{even}(\mathsf{zip}(s,t))$ is generated before before $\mathsf{even}(\mathsf{zip}(s,s))$, which in turn is found before $\mathsf{even}(\mathsf{zip}(\mathsf{ones},\mathsf{ones}))$.

When t_i is generated, for $j < i$ it is checked whether prefixes upon some depth are equal for t_i and t_j for replacing variables by some random streams. If this is the case, then it is tried to prove $E \models t_i = t_j$. If this succeeds, then $t_i = t_j$ is added to the set E' of lemmas.

▶ **Example 6.4.** Consider the following stream specification:

$$\begin{aligned}
\mathsf{morse} &= 0 : \mathsf{zip}(\mathsf{inv}(\mathsf{morse}), \mathsf{tl}(\mathsf{morse})) & \mathsf{tl}(x : \sigma) &= \sigma \\
\mathsf{inv}(x : \sigma) &= \mathsf{not}(x) : \mathsf{inv}(\sigma) & \mathsf{not}(0) &= 1 \\
\mathsf{zip}(x : \sigma, \tau) &= x : \mathsf{zip}(\tau, \sigma) & \mathsf{not}(1) &= 0 \\
\mathsf{toeplitz} &= 1 : \mathsf{zip}(\mathsf{inv}(\mathsf{toeplitz}), \mathsf{ones}) & \mathsf{xor}(0 : x : xs) &= x : \mathsf{xor}(x : xs) \\
\mathsf{ones} &= 1 : \mathsf{ones} & \mathsf{xor}(1 : x : xs) &= \mathsf{not}(x) : \mathsf{xor}(x : xs)
\end{aligned}$$

with the goal of proving $\mathsf{toeplitz} = \mathsf{xor}(\mathsf{morse})$. Here morse is the Thue-Morse stream from Example 2.2, and $\mathsf{toeplitz}$ is a simple instance of a Toeplitz word as presented in [1], 10.11, exercise 42: $\mathsf{toeplitz}$ is the stream obtained by replacing the ?-symbols in the stream $101?^\omega$ consecutively by the elements of $\mathsf{toeplitz}$. The equality between $\mathsf{toeplitz}$ and this alternative characterization can also be proved by Streambox.

Our tool Streambox succeeds in proving $\mathsf{toeplitz} = \mathsf{xor}(\mathsf{morse})$ within approximately one minute. Among others, it discovers the 8 lemmas displayed below.

	Lemma	Uses Lemma
1	$\mathsf{xor}(\mathsf{tl}(s)) = \mathsf{tl}(\mathsf{xor}(s))$	
2	$\mathsf{xor}(\mathsf{inv}(s)) = \mathsf{xor}(s)$	1
3	$\mathsf{inv}(\mathsf{inv}(s)) = s$	
4	$\mathsf{inv}(\mathsf{ones}) = \mathsf{xor}(\mathsf{ones})$	1
5	$\mathsf{zip}(\mathsf{inv}(t), \mathsf{inv}(s)) = \mathsf{inv}(\mathsf{zip}(t, s))$	
6	$\mathsf{xor}(\mathsf{xor}(\mathsf{zip}(t, s))) = \mathsf{zip}(\mathsf{xor}(t), \mathsf{xor}(s))$	1
7	$\mathsf{xor}(\mathsf{zip}(\mathsf{inv}(t), s)) = \mathsf{inv}(\mathsf{xor}(\mathsf{zip}(t, s)))$	1, 2, 3, 5
8	$\mathsf{tl}(\mathsf{inv}(\mathsf{zip}(s, s))) = \mathsf{xor}(\mathsf{zip}(s, \mathsf{ones}))$	1
9	$\mathsf{toeplitz} = \mathsf{xor}(\mathsf{morse})$	1,2,3,4,5,6,7,8

Using these 8 lemmas, finally the proof is given.

In the default setting only the lemmas are shown that are really used, and all proofs are given in full detail, showing the use of lemmas and co-induction hypotheses in separate colors for clarity. By choosing 'options' in the tool several parameters can be changed.

7 Conclusions

Streams or infinite sequences have been study extensively, see e.g., [1] and its bibliography of over 70 pages. A compact way to specify streams is by giving a set of equations. In this paper we investigated techniques for proving equality of streams automatically, given by such sets of equations. As main achievements we summarize:

- We present self-contained theory for the circular co-induction principle for streams, including special contexts, as presented in [12, 5, 6, 9, 10, 13, 7]. Our proofs use the standard semantics of streams being maps from naturals to data.
- We allow any set of equations on streams; in particular we do not restrict to the specific format of behavioral equations as in [9, 10, 13] in which only equations are allowed with left hand sides having hd or tl on top. In contrast to the approach of the tool CIRC [9] we never require termination. Our default format is the pure stream format as used in [3, 14, 15], in which specifications are often given much shorter than by behavioral equations.
- Where in [10] it is stated that the algorithm for computing special contexts is quite complex, and hence not described in detail, we have a very simple criterion for checking for a powerful class of special contexts: *guarded contexts*, inspired by and closely related to *friendly nestingness* from [3].
- We present a technique to prove stream equality by exploiting unicity: two streams are equal if one of them is uniquely defined and the other satisfies the equations of the first. For using this we need to prove unicity automatically, for which we use a technique to prove productivity by means of context-sensitive termination.
- We offer a tool Streambox combining and exploiting these techniques fully automatically. In particular, for hard examples the tool searches for lemmas autonomously, and uses derived lemmas when appropriate. In this way for several stream equalities Streambox outperforms the earlier tool CIRC [9].

References

1 J.-P. Allouche and J. Shallit. *Automatic Sequences: Theory, Applications, Generalizations.* Cambridge University Press, 2003.

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 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/`.

4 J. Giesl et al. AProVE. Web interface and download: `http://aprove.informatik.rwth-aachen.de`.

5 J. Goguen, K. Lin, and G. Roşu. Circular coinductive rewriting. In *Proceedings, 15th International Conference on Automated Software Engineering (ASE'00).* Institute of Electrical and Electronics Engineers Computer Society, 2000. Grenoble, France, 11-15 September 2000.

6 J. Goguen, K. Lin, and G. Roşu. Conditional circular coinductive rewriting with case analysis. In *Recent Trends in Algebraic Development Techniques (WADT02)*, volume 2755 of *Lecture Notes in Computer Science*, pages 216–232. Springer, 2003.

7 E.-I. Goriac, D. Lucanu, and G. Roşu. Automating coinduction with case analysis. In *Twelfth International Conference on Formal Engineering Methods (ICFEM'10)*, volume 6447 of *Lecture Notes in Computer Science*, pages 220–236. Springer, 2010.

8 J. Kim. Coinductive properties of causal maps. In *Proceedings of the 12th International Conference on Algebraic Methodology and Software Technology (AMAST 2008)*, volume 5140 of *Lecture Notes in Computer Science*, pages 253–267. Springer, 2008.

9 D. Lucanu and G. Roşu. CIRC: A circular coinductive prover. In *CALCO'07*, volume 4624 of *Lecture Notes in Computer Science*, pages 372 – 378. Springer, 2007.

10 D. Lucanu and G. Roşu. Circular coinduction with special contexts. In *Proceedings of the 11th International Conference on Formal Engineering Methods (ICFEM'09)*, volume 5885 of *Lecture Notes in Computer Science*, pages 639–659. Springer, 2009.

11 S. Lucas et al. μ-Term. Web interface and download: `http://zenon.dsic.upv.es/muterm/`.

12 G. Roşu. *Hidden Logic.* PhD thesis, University of California at San Diego, 2000.

13 G. Roşu and D. Lucanu. Circular coinduction: A proof theoretical foundation. In *Proceedings of the 3rd International Conference on Algebra and Coalgebra in Computer Science (CALCO'09)*, volume 5728 of *Lecture Notes in Computer Science*, pages 127–144. Springer, 2009.

14 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.

15 H. Zantema and M. Raffelsieper. Proving productivity in infinite data structures. In Christopher Lynch, editor, *Proceedings of the 21st International Conference on Rewriting Techniques and Applications*, volume 6 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 401–416, Dagstuhl, Germany, 2010. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.