26th International Conference on Rewriting Techniques and Applications

RTA'15, June 29 to July 1, 2015, Warsaw, Poland

Edited by

Maribel Fernández

Editor

Maribel Fernández
Department of Informatics
King's College London, UK
Maribel.Fernandez@kcl.ac.uk

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 978-3-939897-85-9

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at http://www.dagstuhl.de/dagpub/9978-3-939897-85-9.

Publication date
June, 2015

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.

License
This work is licensed under a Creative Commons Attribution 3.0 Unported license (CC-BY 3.0): http://creativecommons.org/licenses/by/3.0/legalcode.

In brief, this license authorizes each and everybody to share (to copy, distribute and transmit) the work under the following conditions, without impairing or restricting the authors' moral rights:

- Attribution: The work must be attributed to its authors.

The copyright is retained by the corresponding authors.

Digital Object Identifier: 10.4230/LIPIcs.RTA.2015.i

ISBN 978-3-939897-85-9 **ISSN 1868-8969** **http://www.dagstuhl.de/lipics**

LIPIcs – Leibniz International Proceedings in Informatics

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

Editorial Board

- Susanne Albers (TU München)
- Chris Hankin (Imperial College London)
- Deepak Kapur (University of New Mexico)
- Michael Mitzenmacher (Harvard University)
- Madhavan Mukund (Chennai Mathematical Institute)
- Catuscia Palamidessi (INRIA)
- Wolfgang Thomas (*Chair*, RWTH Aachen)
- Pascal Weil (CNRS and University Bordeaux)
- Reinhard Wilhelm (Saarland University)

ISSN 1868-8969

www.dagstuhl.de/lipics

Contents

Preface
Maribel Fernández vii

Invited Talks

Port Graphs, Rules and Strategies for Dynamic Data Analytics – Extended Abstract
Hélène Kirchner 1

Matching Logic – Extended Abstract
Grigore Roşu 5

Executable Formal Models in Rewriting Logic
Carolyn Talcott 22

Regular Papers

Certification of Complexity Proofs Using CeTA
Martin Avanzini, Christian Sternagel, and René Thiemann 23

Dismatching and Local Disunification in $\mathcal{EL}$
Franz Baader, Stefan Borgwardt, and Barbara Morawska 40

Nominal Anti-Unification
Alexander Baumgartner, Temur Kutsia, Jordi Levy, and Mateu Villaret 57

A faithful encoding of programmable strategies into term rewriting systems
Horatiu Cirstea, Sergueï Lenglet, and Pierre-Etienne Moreau 74

Presenting a Category Modulo a Rewriting System
Florence Clerc and Samuel Mimram 89

Confluence of nearly orthogonal infinitary term rewriting systems
Łukasz Czajka 106

No complete linear term rewriting system for propositional logic
Anupam Das and Lutz Straßburger 127

A Coinductive Framework for Infinitary Rewriting and Equational Reasoning
Jörg Endrullis, Helle Hansen, Dimitri Hendriks, Andrew Polonsky, and Alessandra Silva 143

Proving non-termination by finite automata
Jörg Endrullis and Hans Zantema 160

Reachability Analysis of Innermost Rewriting
Thomas Genet and Yann Salmon 177

Network Rewriting II: Bi- and Hopf Algebras
Lars Hellström 194

Leftmost Outermost Revisited
Nao Hirokawa, Aart Middeldorp, and Georg Moser 209

26th International Conference on Rewriting Techniques and Applications (RTA'15).
Editor: M. Fernández
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Conditional Complexity
Cynthia Kop, Aart Middeldorp, and Thomas Sternagel 223

Constructing Orthogonal Designs in Powers of Two: Gröbner Bases Meet Equational Unification
Ilias Kotsireas, Temur Kutsia, and Dimitris E. Simos 241

Improving Automatic Confluence Analysis of Rewrite Systems by Redundant Rules
Julian Nagele, Bertram Felgenhauer, and Aart Middeldorp 257

Certified Rule Labeling
Julian Nagele and Harald Zankl 269

Transforming Cycle Rewriting into String Rewriting
David Sabel and Hans Zantema 285

Confluence of Orthogonal Nominal Rewriting Systems Revisited
Takaki Suzuki, Kentaro Kikuchi, Takahito Aoto, and Yoshihito Toyama 301

Matrix Interpretations on Polyhedral Domains
Johannes Waldmann 318

System Description Papers

Inferring Lower Bounds for Runtime Complexity
Florian Frohn, Jürgen Giesl, Jera Hensel, Cornelius Aschermann, and Thomas Ströder 334

A Simple and Efficient Step Towards Type-Correct XSLT Transformations
Markus Lepper and Baltasar Trancón y Widemann 350

DynSem: A DSL for Dynamic Semantics Specification
Vlad Vergu, Pierre Neron, and Eelco Visser 365

Preface

This volume contains the papers presented at RTA 2015, the 26th International Conference on Rewriting Techniques and Applications, which was held from 29 June to 1 July 2015, in Warsaw, Poland. RTA 2015 was co-located with the 13th International Conference on Typed Lambda Calculi and Applications (TLCA 2015), as part of RDP 2015, the eighth edition of the International Conference on Rewriting, Deduction, and Programming. The following workshops were also part of RDP 2015: Higher-Dimensional Rewriting and Applications (HDRA), Homotopy Type Theory / Univalent Foundations (HoTT/UF), the Annual Meeting of the IFIP Working Group 1.6 on Term Rewriting, the 29th International Workshop on Unification (UNIF) and the Second International Workshop on Rewriting Techniques for Program Transformations and Evaluation (WPTE).

RTA is the major forum for the presentation of research on all aspects of rewriting. Previous RTA conferences were held in Dijon (1985), Bordeaux (1987), Chapel Hill (1989), Como (1991), Montreal (1993), Kaiserslautern (1995), New Brunswick (1996), Sitges (1997), Tsukuba (1998), Trento (1999), Norwich (2000), Utrecht (2001), Copenhagen (2002), Valencia (2003), Aachen (2004), Nara (2005), Seattle (2006), Paris (2007), Hagenberg/Linz (2008), Brasilia (2009), Edinburgh (2010), Novi Sad (2011), Nagoya (2012), Eindhoven (2013) and Vienna (2014).

RTA 2015 received 43 submissions from 16 countries. The programme committee selected 19 regular papers and 3 system description papers for presentation at the conference. Each paper was reviewed by four members of the programme committee, with the help of external reviewers. The submission and reviewing process, programme committee discussion, and author notifications were all handled seamlessly by the Easychair conference management system.

In addition to the contributed papers, there were three invited talks at RTA 2015, by Hélène Kirchner (joint invited speaker with TLCA 2015), Grigore Roşu and Carolyn Talcott. We thank the three invited speakers for contributing to the success of the conference with their interesting talks and papers.

The programme committee gave the award for *best contribution to RTA 2015* to Jörg Endrullis, Helle Hvid Hansen, Dimitri Hendriks, Andrew Polonsky and Alessandra Silva for the paper *A Coinductive Framework for Infinitary Rewriting and Equational Reasoning*.

The proceedings of RTA 2015 are published as a volume in the LIPIcs series. We thank the LIPIcs editorial office for their help in the preparation of these proceedings.

I would like to thank the members of the organisation committee, and in particular the chair, Aleksy Schubert, for taking care of every detail to make the conference enjoyable for all the participants. It was also a pleasure to work with Thorsten Altenkirch, programme chair of TLCA 2015.

I am very grateful to all the members of the RTA 2015 programme committee and external reviewers for their careful and efficient evaluation of the papers submitted.

RTA 2015 gratefully acknowledges the financial support of the University of Warsaw and the Warsaw Center of Mathematics and Computer Science.

Maribel Fernández

London, 12 May 2015

Programme Committee

Mauricio Ayala-Rincón

Horatiu Cirstea

Stephanie Delaune

Alessandra Di Pierro

Gilles Dowek

Maribel Fernández (chair)

Jürgen Giesl

Michael Hanus

Delia Kesner

Temur Kutsia

Jordi Levy

Salvador Lucas

Christopher Lynch

Ian Mackie

Georg Moser

Detlef Plump

Femke van Raamsdonk

Kristoffer Rose

Masahiko Sakai

Andre Scedrov

Manfred Schmidt-Schauss

Carsten Schürmann

Peter Selinger

Paula Severi

Kazunori Ueda

26th International Conference on Rewriting Techniques and Applications (RTA'15).
Editor: M. Fernández
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Organisation Committee

Marcin Benke

Jacek Chrząszcz

Łukasz Czajka

Patryk Czarnik

Krystyna Jaworska

Aleksy Schubert (RDP chair)

Pawel Urzyczyn

Daria Walukiewicz-Chrząszcz

Maciej Zielenkiewicz

List of Subreviewers

María Alpuente
Sandra Alves
Takahito Aoto
Kyungmin Bae
Patrick Bahr
Steffen van Bakel
Demis Ballis
Alexander Baumgartner
Iovka Boneva
Peter Brottveit Bock
Alan Cain
Łukasz Czajka
Ugo Dal Lago
Roel De Vrijes
Frank Drewes
Joshua Dunfield
Rachid Echahed
Joerg Endrullis
Santiago Escobar
Francisco Ferreira
Traian Florin Serbanuta
Florian Frohn
Murdoch J. Gabbay
Thomas Genet
Stéphane Gimenez
Guillem Godoy
Jakob Grue Simonsen
Yves Guiraud
Daniel Gustafsson
Raúl Gutiérrez
Peter Habermehl
Rita Hartel
Willem Heijltjes
Dimitri Hendriks
Claudio Hermida
Peter Hibbs
Vincent Hugot
Munehiro Iwami
Florent Jaquemard
Artur Jez
Stefan Kahrs
Yoshiharu Kojima
Boris Konev
Serguei Lenglet
Flavio L. C. De Moura
Luigi Liquori
Carlos Lombardi
Dorel Lucanu
Philippe Malbos
António Malheiro
Andrew M. Marshall
Sebastian Maneth
Alexander Maringele
Mircea Marin
Wim Martens
Edgar Martinez-Moro
Marino Miculan
Samuel Mimram
Akimasa Morihata
Guillaume Munch-Maccagnoni
César Muñoz
Naoki Nishida

Mizuhito Ogawa

Ana C. R. Oliveira

Vincent Padovani

Andrei Paskevich

Andrei Popescu

Timothy Porter

Nicolas Pouillard

Thomas Powell

Silvio Ranise

Daniel R. Licata

Camilo Rocha

David Sabel

Pawel Sobocinski

Christian Sternagel

Christoph Sticksel

Thomas Ströder

René Thiemann

Yoshihito Toyama

Daniel Ventura

Jamie Vicary

Herbert Wiklicky

Akihisa Yamada

Harald Zankl

Hans Zantema

Port Graphs, Rules and Strategies for Dynamic Data Analytics – Extended Abstract

Hélène Kirchner

Inria
France
helene.kirchner@inria.fr

Abstract

In the context of understanding, planning and anticipating the behaviour of complex systems, such as biological networks or social networks, this paper proposes port graphs, rules and strategies, combined in strategic rewrite programs, as foundational ingredients for interactive and visual programming and shows how they can contribute to dynamic data analytics.

1998 ACM Subject Classification D.3.3 Language Constructs and Features (E.2)

Keywords and phrases Graphs, Rewrite Rules, Strategic Rewriting

Digital Object Identifier 10.4230/LIPIcs.RTA.2015.1

Category Invited Talk

1 Introduction

Understanding, planning and anticipating the behaviour of complex systems, such as biological networks or social networks, raise a number of theoretical and practical challenges. Their complexity comes from heterogeneity of components, from their dynamics, their increasing number, or from the data deluge they generate or manage. Handling this complexity requires languages with a high-level of expressivity and with modular constructs. But this also needs powerful visualization tools to represent data and their dynamic evolution, analysis of different alternatives, parameter tuning. Capability of re-playing or backtracking are important concerns to address.

Three ingredients contribute to address these challenges: port graphs provide a powerful representation of data, rules capture their evolution and provide high-level prototyping mechanism, strategy makes the control and choices explicit. They are combined in the concept of strategic rewrite programs whose logical and semantic background is well-understood. The interactive PORGY environment, yet in-progress, illustrates what can be done and the actual visualization challenges, through an application to the study of propagation in social networks.

2 Data and Graphs

Graph formalisms are useful to easily describe complex structures in an intuitive way, like UML diagrams, proof representation, micro-processors design, or workflows. Port graphs [1] are a general class of labelled graphs that have been used to model a variety of complex data, such as biological networks or social networks. Intuitively, a port graph is a graph where nodes have explicit connection points called ports. Edges are undirected, exclusively attached to ports and two ports may be connected by more than one edge. Nodes, ports

© Hélène Kirchner;
licensed under Creative Commons License CC-BY
26th International Conference on Rewriting Techniques and Applications (RTA'15).
Editor: Maribel Fernández; pp. 1–4
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

and edges are labelled by a set of properties. For instance, an edge may be associated with a state (such as used or marked) and a node may have for instance a name, a colour and a user-defined function as properties. Properties may be used to define the behaviour of the modelled system and for visualization purposes. Thanks to these features, port graphs provide a rich structure able to model many kinds of data and processes.

3 Rewrite rule programming

Rewriting has to be understood here in a broad sense: rewriting transforms syntactic structures that may be words, terms, propositions, dags, graphs, geometric objects like segments, and in general any kind of structured objects. Transformations are expressed with rules, built on the same syntax but with an additional set of variables and with a binder $\Rightarrow$, relating the left and the right-hand side of the rule. Optionally, a condition or constraint restricts the set of values allowed for the variables.

In this rewriting process, there are many possible choices: the rule itself, the position(s) in the structure, the matching homomorphism(s). For instance, one may choose to apply a rule concurrently at all disjoint positions where it matches, or using matching modulo an equational theory like associativity-commutativity.

Rewriting Logic [8] and Rewriting Calculus [5] have contributed to establish rewriting as a model of computation accounting for concurrency, parallelism, communication, and interaction. It also has good properties as a metalogical framework for representing logics.

Graph transformations have many applications in specification, programming, and simulation tools and several languages and tools are based on this formalism. The dynamics of a complex system modeled by a port graph can then be specified using graph rewriting rules.

4 Control via Strategies

While rules describe local transformations, strategies describe the control of rule application. Strategy is an explicit concept in sequential path-building games, in automated deduction and reasoning systems and more generally are used to express complex designs for control in modeling, proof search, program transformation, SAT solving. In these domains, deterministic rule-based computations or deductions are often not sufficient to capture complex computations or proof developments. Strategies provide the formal mechanism needed, for instance, to sequentialize the search for different solutions, to check context conditions, to request user input to instantiate variables, to process subgoals in a particular order, etc.

Several approaches exist to describe strategies: a proof term expressed in rewriting logic; a ρ-term in rewriting calculus; a subset of paths in a derivation tree ; a partial function that associates to a reduction-in-progress, the possible next steps in the reduction sequence ; positional strategies that choose where rules are allowed to apply.

A few strategy languages have been designed in the rewriting community, in particular in ELAN [4], Stratego [11], Maude [6] or Tom [2]. Common constructs with some variants are emerging from these proposals.

5 Strategic Programming

Port graphs, rules and strategies are combined in the concept of strategic programs. A *strategic rewrite program* consists of a finite set of rewrite rules $\mathcal{R}$, a strategy expression S, built from $\mathcal{R}$ using a strategy language, and a given structure G.

There are several ways to describe the operational semantics of a programming language. Due to the fact that rewriting logic is reflexive, it is tempting to describe the operational semantics of a strategy language with a set of rewrite rules. This has been done for instance for ELAN [3], Maude [6] and PORGY [1] at least. Another way by defining a transition relation on configurations using semantic rules in the SOS style is given in [7].

As presented in [9], it is expected from a strategy language and its operational semantics to satisfy the properties of correctness and completeness w.r.t. rewriting derivations.

6 The PORGY environment

PORGY [1, 7] is a visual environment that allows users to define port graphs and port graph rewrite rules, and to apply the rewrite rules in an interactive way, or via the use of strategies. To control the application of rewriting rules, PORGY provides a strategy language. A distinctive feature of PORGY's language is that it allows users to define strategies using not only operators to combine graph rewriting rules but also operators to define the location in the target graph where rules should, or should not, apply. Users can create graph rewriting derivations and specify graph traversals using the language primitives to select rewriting rules and the subgraphs where the rules apply.

In order to support the various tasks involved in the study of a port graph rewriting system, the system provides facilities:

- to offer different views on each component of the rewriting system: the current graph being rewritten, the derivation tree, the rules and the strategy, with drag-and-drop mechanisms to apply rules and strategies on a given state,
- to explore a derivation tree with all possible derivations,
- to perform on-demand reduction using a strategy language which permits to restrict or guide the reductions,
- to track the reduction throughout the whole tree,
- to navigate in the tree, for instance, backtracking and exploring different branches,
- to plot the evolution of a chosen parameter (a specific element in the port graph structure) along a derivation. The system supports synchronisation between the different views: selecting points on the plot view triggers the selection of the corresponding nodes in the trace tree. Such a mechanism obviously helps to track properties of the output graph along the rewriting process.

These features have been successfully applied to propose a visual analytics approach to compare propagation models in social networks in [10].

7 Conclusion

Although first results are promising, this approach raises numerous challenges in different domains; let un mention some of them. Considering the huge amount of data to manage, knowledge representation in a structured way should allow better efficient mining, reasoning and inference. For pattern matching on big graphs with millions of nodes, fast or fuzzy matching can be explored as well as exploiting the graph structure. Graph rewriting to model big graphs evolution has to be adapted to their increasing sizes, both for efficient concurrent application of rules and for adapting the granularity of transformation steps to the property of interest. Strategies should help in this respect. From the point of view of strategies and strategic rewrite programs, properties of confluence, termination, or completeness for rewriting under strategies have been already addressed. But other properties of strategies

such as fairness or loop-freeness could be worth-fully explored by making connections between different communities (functional programming, proof theory, verification, game theory,...). Finally, at all levels of graphs, rules and strategies, as well as for visualisation of processes evolution, structuration is a challenge and modular constructs for composability have to be studied in a way coherent between all these levels.

Acknowledgements. The results presented here are based on joint work on ELAN and Tom languages designed in the Protheo team from 1990 to 2008, and on PORGY since 2008. I am grateful to José Meseguer and to the members of the Protheo and the PORGY teams, for many inspiring discussions on the topics of this paper.

References

1 Oana Andrei, Maribel Fernandez, Hélène Kirchner, Guy Melançon, Olivier Namet, and Bruno Pinaud. PORGY: Strategy-Driven Interactive Transformation of Graphs. In Rachid Echahed, editor, *TERMGRAPH, 6th Int'l Workshop on Computing with Terms and Graphs*, vol. 48 of *Electronic Proceedings in Theoretical Computer Science (EPTCS)*, pp. 54–68, 2011.

2 Emilie Balland, Paul Brauner, Radu Kopetz, Pierre-Etienne Moreau, and Antoine Reilles. Tom: Piggybacking Rewriting on Java. In Franz Baader, editor, *Proceedings of the 18th Conference on Rewriting Techniques and Applications*, vol. 4533 of *Lecture Notes in Computer Science*, pp. 36–47. Springer, 2007.

3 Peter Borovanský, Claude Kirchner, Hélène Kirchner, and Pierre-Etienne Moreau. ELAN from a rewriting logic point of view. *Theoretical Computer Science*, 2(285):155–185, 2002.

4 Peter Borovanský, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, and Christophe Ringeissen. An overview of ELAN. *Electr. Notes Theor. Comput. Sci.*, 15:55–70, 1998.

5 Horatiu Cirstea and Claude Kirchner. The rewriting calculus – Part I *and* II. *Logic Journal of the Interest Group in Pure and Applied Logics*, 9(3):427–498, May 2001.

6 Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *All About Maude – A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, vol. 4350. Springer, 2007.

7 Maribel Fernández, Hélène Kirchner, and Olivier Namet. Strategic port graph rewriting: an interactive modelling and analysis framework. In Alberto Lluch Lafuente Dragan Bosnacki, Stefan Edelkamp and Anton Wij, editors, *Proceedings 3rd Workshop on GRAPH Inspection and Traversal Engineering (GRAPHITE 2014), Grenoble, France, 5th April 2014*, vol. 159 of *Electronic Proceedings in Theoretical Computer Science*, pp. 15–29, 2014.

8 Narciso Martí-Oliet and José Meseguer. Rewriting logic as a logical and semantic framework. In J. Meseguer, editor, *Electronic Notes in Theoretical Computer Science*, vol. 4. Elsevier Science Publishers, 2000.

9 Narciso Martí-Oliet, José Meseguer, and Alberto Verdejo. A rewriting semantics for Maude strategies. *Electronic Notes in Theoretical Computer Science*, 238(3):227–247, 2008.

10 Jason Vallet, Hélène Kirchner, Guy Melançon, Olivier Namet, and Bruno Pinaud. A visual analytics approach to compare propagation models in social networks. In Arend Rensink and Eduardo Zambon, editors, *Graphs as Models*, vol. 181 of *Electronic Proceedings in Theoretical Computer Science (EPTCS)*, 2015.

11 Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*, vol. 2051 of *Lecture Notes in Computer Science*, pp. 357–361. Springer, May 2001.

Matching Logic – Extended Abstract*

Grigore Roşu

University of Illinois at Urbana-Champaign, USA
`grosu@illinois.edu`

Abstract

This paper presents *matching logic*, a first-order logic (FOL) variant for specifying and reasoning about structure by means of patterns and pattern matching. Its sentences, the *patterns*, are constructed using *variables*, *symbols*, *connectives* and *quantifiers*, but no difference is made between function and predicate symbols. In models, a pattern evaluates into a power-set domain (the set of values that *match* it), in contrast to FOL where functions and predicates map into a regular domain. Matching logic uniformly generalizes several logical frameworks important for program analysis, such as: propositional logic, algebraic specification, FOL with equality, and separation logic. Patterns can specify separation requirements at any level in any program configuration, not only in the heaps or stores, without any special logical constructs for that: the very nature of pattern matching is that if two structures are matched as part of a pattern, then they can only be spatially separated. Like FOL, matching logic can also be translated into pure predicate logic, at the same time admitting its own sound and complete proof system. A practical aspect of matching logic is that FOL reasoning remains sound, so off-the-shelf provers and SMT solvers can be used for matching logic reasoning. Matching logic is particularly well-suited for reasoning about programs in programming languages that have a rewrite-based operational semantics.

1998 ACM Subject Classification D.2.4 Software/Program Verification, D.3.1 Formal Definitions and Theory, F.3 Logics and Meanings of Programs, F.4 Mathematical Logic and Formal Languages

Keywords and phrases Program logic, First-order logic, Rewriting, Verification

Digital Object Identifier 10.4230/LIPIcs.RTA.2015.5

Category Invited Talk

1 Introduction, Motivation and Overview

In their simplest form, as term templates with variables, patterns abound in mathematics and computer science. They match a concrete, or ground, term if and only if there is some substitution applied to the pattern's variables that makes it equal to the concrete term, possibly via domain reasoning. This means, intuitively, that the concrete term obeys the structure specified by the pattern. We show that when combined with logical connectives and variable constraints and quantifiers, patterns provide a powerful means to specify and reason about the structure of states, or configurations, of a programming language.

Matching logic was inspired from the domain of programming language semantics, specifically from attempting to use rewrite-based operational semantics for program verification. For example, a series of large and complete semantic definitions of real languages has been

* The work presented in this paper was supported in part by the Boeing grant on "Formal Analysis Tools for Cyber Security" 2014–2015, the NSF grants CCF-1218605, CCF-1318191 and CCF-1421575, and the DARPA grant under agreement number FA8750-12-C-0284.

© Grigore Roşu;
licensed under Creative Commons License CC-BY
26th International Conference on Rewriting Techniques and Applications (RTA'15).
Editor: Maribel Fernández; pp. 5–21
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

```
struct listNode { int val; struct listNode *next; };

void list_read_write(int n) {
rule  ⟨$ ⇒ return; ···⟩k ⟨A ⇒ · ···⟩in ⟨··· · ⇒ rev(A)⟩out  requires  n = len(A)
  int i=0;
  struct listNode *x=0;
inv  ⟨β ···⟩in ⟨··· list(x,α) ···⟩heap ∧ i ≤ n ∧ len(β) = n − i ∧ A = rev(α)@β
  while (i < n) {
    struct listNode *y = x;
    x = (struct listNode*) malloc(sizeof(struct listNode));
    scanf("%d", &(x->val));
    x->next = y;
    i += 1; }
inv  ⟨··· α⟩out ⟨··· list(x,β) ···⟩heap ∧ rev(A) = α@β
  while (x) {
    struct listNode *y;
    y = x->next;
    printf("%d␣",x->val);
    free(x);
    x = y; }
}
```

Figure 1 Reading, storing, and reverse writing a sequence of integers.

recently developed using the $\mathbb{K}$ framework (`http://kframework.org` [18, 19]), such as C11 (POPL'12 [6], PLDI'15 [8]), Java 1.4 (POPL'15 [2]), JavaScript ES5 (PLDI'15 [13]), with many other similar but partial semantics of other languages. Each of these language semantics has more than 1,000 semantic rules and has been thoroughly tested on benchmarks and test suites that implementations of these languages use to test their conformance, where available. Unfortunately, the current state-of-the-art in program verification is to define yet another semantics for these languages, amenable for reasoning about programs, such as an axiomatic or a dynamic logic semantics, because the general belief is that operational semantics are too low level for program verification. Moreover, when the correctness of the verifier itself is a concern, tedious proofs of equivalence between the operational and the alternative semantics are produced. That is because operational semantics are comparatively much easier to define and at the same time are executable (and thus also testable), so they are often considered as reference models of the corresponding languages, while the alternative semantics devised for verification purposes tend to be more mathematically involved and are not executable so they may hide tricky errors. Defining even one semantics for a real language like C or Java is already a huge effort. Defining more semantics, each good for a different purpose, is at best very uneconomical, with or without proofs of equivalence with the reference semantics.

Matching logic was born from our firm belief that programming languages must have formal definitions of their syntax and semantics, and that all the execution and analysis tools for a given language, such as parsers, interpreters, compilers, state-space explorers, model checkers, deductive program verifiers, etc., can be derived from just *one* reference formal definition of the language, which is executable and easy to test. No other semantics for the same language should be needed. This is the ideal scenario and we believe that there is enough evidence that it is within our reach in the short term. The main idea is that semantic rules match and apply on program *configurations*, which are algebraic data types defined as terms constrained by equations capturing the needed mathematical domains, such as lists (e.g., for input/output buffers, function stacks, etc.), sets (e.g., for concurrent threads or processes, for resources held, etc.), maps (e.g., for environments, heaps, etc), and so on.

To reason about programs we need to be able to reason about program configurations. Specifically, we need to define configuration abstractions and reason with them. Consider, for example, the program in Figure 1 which shows a C function that reads `n` elements from the standard input and prints them to the standard output in reversed order (for now, we can ignore the specifications, which are grayed). While doing so, it allocates a singly linked list storing the elements as they are read, and then deallocates the list as the elements are printed. In the end, the heap stays unchanged. To state the specification of this program, we need to match an abstract sequence of `n` elements in the input buffer, and then to match its reverse at the end of the output buffer when the function terminates. Further, to state the invariants of the two loops we need to identify a singly linked pattern in the heap, which is a partial map. Many such sequence or map patterns, as well as operations on them, can be easily defined using conventional algebraic data types (ADTs). But some of them cannot.

A major limitation of ADTs and of FOL is that operation symbols are interpreted as functions in models, which sometimes is insufficient. E.g., a two-element linked list in the heap (we regard heaps as maps from locations to values) starting with location 7 and holding values 9 and 5, written as pattern $\mathit{list}(7, 9 \cdot 5)$, can allow infinitely many heap values, one for each location where the value 5 may be stored. So we cannot define *list* as an operation symbol $\mathit{Int} \times \mathit{Seq} \to \mathit{Map}$. The FOL alternative is to define *list* as a predicate $\mathit{Int} \times \mathit{Seq} \times \mathit{Map}$, but mentioning the map all the time as an argument makes specifications verbose and hard to read, use and reason about. An alternative offered by separation logic [11, 14, 12] is to fix and move the map domain from explicit in models to implicit in the core of the logic, so that $\mathit{list}(7, 9 \cdot 5)$ is interpreted as a predicate but the map and the non-deterministic choices are implicit in the logic. We then may need custom separation logics for different languages that require different variations of map models or different configurations making use of different kinds of resources. This may also require specialized separation logic theorem provers needed for each, or otherwise encodings that need to be proved correct. Matching logic avoids the limitations of both approaches above, by interpreting its terms/formulae as *sets* of values.

Matching logic's formulae, or *patterns*, are defined using variables, symbols from a signature, and FOL connectives and quantifiers. We only treat the many-sorted first-order case here, but the same ideas can be extended to order-sorted or higher-order contexts. Specifically, if (S, Σ) is a many-sorted signature and Var an S-sorted set of variables, then a pattern φ of sort $s \in S$ can inductively be a variable in Var_s, or have the form $\sigma(\varphi_1, \ldots, \varphi_n)$ where $\sigma \in \Sigma$ is a symbol of result s (and arguments of any sorts) and $\varphi_1, \ldots, \varphi_n$ are patterns of appropriate sorts, or $\neg\varphi'$ or $\varphi' \wedge \varphi''$ or $\exists x.\varphi'$ where φ' and φ'' are patterns of sort s and $x \in \mathit{Var}$ (of any sort). Derived constructs $\vee$, $\forall$, $\to$, $\leftrightarrow$, $\top$, $\bot$ can be defined as usual. One way to think of patterns is that they collapse the function and predicate symbols of FOL, allowing patterns to be simultaneously regarded *both* as terms and as predicates. When regarded as terms they build structure, and when regarded as predicates they express constraints.

Semantically, a *model* M consists of a carrier M_s for each sort s, like in FOL, but interprets symbols $\sigma \in \Sigma_{s_1 \ldots s_n, s}$ as maps $\sigma_M : M_{s_1} \times \cdots \times M_{s_n} \to \mathcal{P}(M_s)$ yielding a *set* of elements. In particular, σ_M can be a function, when the set contains only one element, or a partial function, when the set contains at most one element. Any M-*valuation* $\rho : \mathit{Var} \to M$ extends to a map $\overline{\rho}$ taking patterns to sets of values, where $\neg$ is interpreted as the complement, $\wedge$ as intersection, and $\exists$ as union over all compatible valuations. If φ is a pattern and $a \in \overline{\rho}(\varphi)$ then we say that *a matches φ (with ρ)*. The name of matching logic was inspired from the case when M is a term model, quite common in the context of programming language semantics where M typically represents a program configuration or a fragment of it. In that case, if a is a ground term and φ is a term with variables, then "a matches φ" in matching

logic becomes precisely the usual notion of pattern matching. Pattern φ is *valid in M* iff $\overline{\rho}(\varphi) = M$ (i.e., it is matched by all elements of M), and it is *valid* iff it is valid in all models.

It turns out that, unlike in FOL, equality can be defined in matching logic (Section 4.3): i.e., $\varphi_1 = \varphi_2$ is a pattern so that, given any model M and any M-valuation $\rho : Var \to M$, $\varphi_1 = \varphi_2$ is either matched by all elements when $\overline{\rho}(\varphi_1) = \overline{\rho}(\varphi_2)$, or by none otherwise.

Let us discuss some simple examples. If Σ is the signature of Peano natural numbers and M is the model of natural numbers with 0 and *succ* interpreted accordingly, then the pattern $\exists x . succ(x)$ is matched by all positive numbers: indeed, by the semantics of the existential quantifier, it is the *union* of all successors of natural numbers. If we want to only allow models in which 0 is interpreted as one element, *succ* as a total and injective function, and whose elements are either zero or successors of other elements, then we add the axioms:

$$\exists y . 0 = y \qquad\qquad 0 \vee \exists x . succ(x)$$
$$\exists y . succ(x) = y \qquad\qquad succ(x_1) = succ(x_2) \to x_1 = x_2$$

We can go further and axiomatize *plus* the same way we are used to in algebraic specification:

$$plus(0, y) = y \qquad\qquad plus(succ(x), y) = succ(plus(x, y))$$

or equivalently as the following equality matching logic (and not FOL) pattern:

$$plus(x, y) = (x = 0 \wedge y \vee \exists z . x = succ(z) \wedge succ(plus(z, y)))$$

We next define a matching logic specification whose symbols are not all functions anymore. Consider a typical ADT of maps from natural to integer numbers, with *emp* the empty map, $_ \mapsto _$ a one binding map, $_ \mapsto [_]$ a map of consecutive bindings, and $_ * _$ the partial function merging two maps (notations inspired from separation logic [11, 14, 12]). In addition to the usual unit, associativity and commutativity axioms for *emp* and $_ * _$, we also add

$$\neg(0 \mapsto a) \qquad x \mapsto a * x \mapsto b = \bot \qquad x \mapsto [\epsilon] = emp \qquad x \mapsto [a, S] = x \mapsto a * (x + 1) \mapsto [S]$$

The first pattern says 0 cannot serve as the key of any binding. The second pattern says that the keys of different bindings in a map must be distinct. The last two patterns desugar the consecutive binding construct. Consider now a symbol $list \in \Sigma_{Nat \times Seq, Map}$ taking a number x and a sequence of integers S to a set of maps $list(x, S)$, together with the following:

$$list(0, \epsilon) = emp \qquad\qquad list(x, n \cdot S) = \exists z . x \mapsto [n, z] * list(z, S)$$

This looks similar to how the list predicate is defined in separation logic using recursive predicates, although in matching logic there are no predicates and no recursion. The equations above use the same principle to define *list* as the Peano equations did to define *plus*: pattern equations. We can now show (see Section 4.7) that in the model whose *Map* carrier consists of the finite-domain partial maps, and where $\mapsto$ and $*$ are interpreted appropriately, the interpretation of $list(x, S)$ is precisely the set of all singly-linked lists starting with $x \neq 0$ and comprising the sequence of integers S. That is, in the intended model, the $list(x, S)$ pattern is matched by precisely the desired lists. In fact, we can show that the matching logic specification above, in the map model, is equivalent to separation logic (Section 4.9).

Using the generic proof system of matching logic in Section 5, we can now derive properties about lists in this specification, such as, e.g., $(1 \mapsto 5 * 2 \mapsto 0 * 7 \mapsto 9 * 8 \mapsto 1) \to list(7, 9 \cdot 5)$:

$$\begin{array}{lcll}
1 \mapsto 5 * 2 \mapsto 0 * 7 \mapsto 9 * 8 \mapsto 1 & = & 1 \mapsto [5, 0] * 7 \mapsto [9, 1] & = \\
1 \mapsto [5, 0] * list(0, \epsilon) * 7 \mapsto [9, 1] & \to & (\exists z . 1 \mapsto [5, z] * list(z, \epsilon)) * 7 \mapsto [9, 1] & = \\
list(1, 5 \cdot \epsilon) * 7 \mapsto [9, 1] & = & list(1, 5) * 7 \mapsto [9, 1] & \to \\
\exists z . 7 \mapsto [9, z] \wedge list(z, 5) & = & list(7, 9 \cdot 5) &
\end{array}$$

The benefits of matching logic can be perhaps best seen when there are no immediate existing logics to reason about certain structures. Consider, e.g., the operational semantics of a real language like C, whose configuration can be defined with ordinary ADTs but has more than 100 semantic cells [6, 8]. The semantic cells, written using symbols $\langle ... \rangle_{\mathsf{cell}}$, can be nested and their grouping is associative and commutative. There is a top cell $\langle ... \rangle_{\mathsf{cfg}}$ holding a subcell $\langle ... \rangle_{\mathsf{heap}}$ among many others. We can globalize the local reasoning above to the entire C configuration proving the following property using the same proof system in Section 5:

$$\forall c\!:\!\mathit{Cfg}.\forall h\!:\!\mathit{Map}\,.\,(\langle\langle 1 \mapsto 5 * 2 \mapsto 0 * 7 \mapsto 9 * 8 \mapsto 1 * h\rangle_{\mathsf{heap}}\ c\rangle_{\mathsf{cfg}}\ \rightarrow\ \langle\langle \mathit{list}(7, 9 \cdot 5) * h\rangle_{\mathsf{heap}}\ c\rangle_{\mathsf{cfg}})$$

Quantification over the heap or over the configuration are first-order in matching logic. We refer to such variables like h and c matching the remaining contents of a cell "cell" as *(structural) "cell" frames*; e.g., h is the *(structural) heap frame* and c is the *(structural) configuration frame*, and write them as ellipses ("...") when their particular name is irrelevant.

The C semantics consists of more than 2,000 rewrite rules between patterns (ordinary terms with variables are patterns). We are currently developing an extension of the $\mathbb{K}$ framework that allows us to verify programs using a rewrite-based operational semantics of the programming language, like in [4, 16, 20]. Matching logic reasoning is used in-between semantic rewrite rule applications to re-arrange the configuration so that semantic rules match or assertions can be proved. This work-in-progress extension of $\mathbb{K}$ will be reported elsewhere. In the remainder of this section we only want to emphasize, by means of example, that in spite of its generality, matching logic can also be implemented efficiently.

Figure 1 showed a C function whose correctness can be automatically verified by our current prototype prover. The reader can check it, as well as dozens of other programs, using the online MatchC interface at `http://matching-logic.org`; this function is under the `io` folder and it takes about 150ms to verify. The rule specification of the function states its semantics/summary: the body ($) returns in the code cell $\langle\rangle_{\mathsf{k}}$ possibly followed by other code (as mentioned, "..." are structural frames, that is, universally quantified "anonymous" variables), the sequence A of size n is consumed from the prefix of the input buffer (A is rewritten to "$\cdot$", the unit of collections, possibly followed by more input), and the reversed sequence $\mathsf{rev}(\mathsf{A})$ is put at the end of the output buffer.

The first loop invariant says the pattern $\mathsf{list}(\mathsf{x}, \alpha)$ is matched somewhere in the heap, and that the sequence β of size $\mathsf{n} - \mathsf{i}$ is available in the input buffer such that A is the reverse of the sequence that x points to, $\mathsf{rev}(\alpha)$, concatenated with β. By convention, Boolean patterns like $\mathsf{i} \leq \mathsf{n}$ can be used in any context and they are either matched by all elements when they hold, or by no elements when they do not hold (Section 4.5). The variables starting with a ? are assumed existentially quantified. The invariant of the second loop says that a sequence α can be matched as a suffix of the output buffer and sequence β can be matched within a list that x points to in the heap, such that $\alpha @ \beta$ is the reverse of the original sequence A. The verification of this function consists of executing the rewrite semantics of C symbolically on all paths and, each time a pattern is encountered, a pattern implication proof task is deferred to the matching logic prover. For example, the last proof task is:

$$\begin{array}{ll} & \langle\langle I\rangle_{\mathsf{in}}\ \langle O, \alpha\rangle_{\mathsf{out}}\ \langle \mathsf{list}(\mathsf{x}, \beta) * H\rangle_{\mathsf{heap}}\ C\rangle_{\mathsf{cfg}}\ \ \wedge\ \ \mathsf{rev}(\mathsf{A}) = \alpha @ \beta\ \ \wedge\ \ \mathsf{x} = 0 \\ \rightarrow & \langle\langle I\rangle_{\mathsf{in}}\ \langle O, \mathsf{rev}(\mathsf{A})\rangle_{\mathsf{out}}\ \langle H\rangle_{\mathsf{heap}}\ C\rangle_{\mathsf{cfg}} \end{array}$$

which can be proved using the proof system in Section 5 and the given pattern axioms.

Section 2 introduces the syntax and semantics of matching logic. Section 3 shows that, like FOL with equality, matching logic also translates to predicate logic. Section 4 enumerates a series of examples, most notably showing that equality is definable. Section 5 introduces a sound and complete proof system. Section 6 discusses related work and Section 7 concludes.

2 Matching Logic

We assume the reader familiar with many-sorted sets, functions, and FOL. For any given set of sorts S, we assume Var is an S-sorted set of variables, sortwise infinite and disjoint. We may write $x : s$ instead of $x \in Var_s$; when the sort of x is irrelevant, we just write $x \in Var$. We let $\mathcal{P}(M)$ denote the powerset of a many-sorted set M, which is itself many-sorted.

▶ **Definition 1.** Let (S, Σ) be a many-sorted signature of *symbols*. Matching logic (S, Σ)-*formulae*, also called (S, Σ)-*patterns*, or just (matching logic) *formulae* or *patterns* when (S, Σ) is understood from context, are inductively defined as follows for all sorts $s \in S$:

$$\varphi_s \ ::= \ x \in Var_s \ \mid \ \sigma(\varphi_{s_1}, ..., \varphi_{s_n}) \ \text{ with } \sigma \in \Sigma_{s_1 \ldots s_n, s} \ \mid \ \neg\varphi_s \ \mid \ \varphi_s \wedge \varphi_s \ \mid \ \exists x.\varphi_s \text{ with } x \in Var$$

Let $\textsc{Pattern}$ be the S-sorted set of patterns. By abuse of language, we refer to the symbols in Σ also as patterns: think of $\sigma \in \Sigma_{s_1 \ldots s_n, s}$ as the pattern $\sigma(x_1 : s_1, \ldots, x_n : s_n)$.

To compact notation, $\varphi \in \textsc{Pattern}$ means φ is any pattern, while $\varphi_s \in \textsc{Pattern}$ or $\varphi \in \textsc{Pattern}_s$ that it has sort s. We adopt the following derived constructs:

$$\begin{array}{rcl@{\qquad}rcl} \bot_s & \equiv & x : s \wedge \neg x : s & \varphi_1 \rightarrow \varphi_2 & \equiv & \neg\varphi_1 \vee \varphi_2 \\ \top_s & \equiv & \neg\bot_s & \varphi_1 \leftrightarrow \varphi_2 & \equiv & (\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1) \\ \varphi_1 \vee \varphi_2 & \equiv & \neg(\neg\varphi_1 \wedge \neg\varphi_2) & \forall x.\varphi & \equiv & \neg(\exists x.\neg\varphi) \end{array}$$

and let $FV(\varphi)$ denote the *free variables* of φ, defined as usual.

▶ **Definition 2.** A *matching logic* (S, Σ)-*model* M, or simply a *model* when (S, Σ) is understood, consists of: (1) An S-sorted set $\{M_s\}_{s \in S}$, where each set M_s, called the *carrier of sort* s *of* M, is assumed non-empty; and (2) A function $\sigma_M : M_{s_1} \times \cdots \times M_{s_n} \rightarrow \mathcal{P}(M_s)$ for each symbol $\sigma \in \Sigma_{s_1 \ldots s_n, s}$, called the *interpretation* of σ in M.

Note that usual (S, Σ)-algebras are special cases of matching logic models, where $|\sigma_M(m_1, \ldots, m_n)| = 1$ for any $m_1 \in M_{s_1}$, …, $m_n \in M_{s_n}$. Similarly, partial (S, Σ)-algebras also fall as special case, where $|\sigma_M(m_1, \ldots, m_n)| \leq 1$, since we can capture the undefinedness of σ_M on m_1, …, m_n with $\sigma_M(m_1, \ldots, m_n) = \emptyset$. We tacitly use the same notation σ_M for its extension $\mathcal{P}(M_{s_1}) \times \cdots \times \mathcal{P}(M_{s_n}) \rightarrow \mathcal{P}(M_s)$ to argument sets, i.e., $\sigma_M(A_1, \ldots, A_n) = \bigcup\{\sigma_M(a_1, \ldots, a_n) \mid a_1 \in A_1, \ldots, a_n \in A_n\}$, where $A_1 \subseteq M_{s_1}, \ldots, A_n \subseteq M_{s_n}$.

▶ **Definition 3.** Given a model M and a map $\rho : Var \rightarrow M$, called an M-*valuation*, let its extension $\overline{\rho} : \textsc{Pattern} \rightarrow \mathcal{P}(M)$ be inductively defined as follows:

- $\overline{\rho}(x) = \{\rho(x)\}$, for all $x \in Var_s$
- $\overline{\rho}(\sigma(\varphi_{s_1}, \ldots, \varphi_{s_n})) = \sigma_M(\overline{\rho}(\varphi_1), \ldots \overline{\rho}(\varphi_n))$
- $\overline{\rho}(\neg\varphi_s) = M_s \setminus \overline{\rho}(\varphi_s)$ ("$\setminus$" is set difference)
- $\overline{\rho}(\varphi_1 \wedge \varphi_2) = \overline{\rho}(\varphi_1) \cap \overline{\rho}(\varphi_2)$
- $\overline{\rho}(\exists x.\varphi) = \bigcup\{\overline{\rho'}(\varphi) \mid \rho' : Var \rightarrow M,\ \rho'{\restriction}_{Var \setminus \{x\}} = \rho{\restriction}_{Var \setminus \{x\}}\}$ ("$\rho{\restriction}_A$" is ρ restricted to A)

The extension of ρ works as expected with the derived constructs:

- $\overline{\rho}(\bot_s) = \emptyset$ and $\overline{\rho}(\top_s) = M_s$
- $\overline{\rho}(\varphi_1 \vee \varphi_2) = \overline{\rho}(\varphi_1) \cup \overline{\rho}(\varphi_2)$
- $\overline{\rho}(\varphi_1 \rightarrow \varphi_2) = \{m \in M_s \mid m \in \overline{\rho}(\varphi_1) \text{ implies } m \in \overline{\rho}(\varphi_2)\} = M_s \setminus (\overline{\rho}(\varphi_1) \setminus \overline{\rho}(\varphi_2))$
- $\overline{\rho}(\varphi_1 \leftrightarrow \varphi_2) = \{m \in M_s \mid m \in \overline{\rho}(\varphi_1) \text{ iff } m \in \overline{\rho}(\varphi_2)\} = M_s \setminus (\overline{\rho}(\varphi_1)\ \Delta\ \overline{\rho}(\varphi_2))$
  ("Δ" is the set symmetric difference operation)
- $\overline{\rho}(\forall x.\varphi) = \bigcap\{\overline{\rho'}(\varphi) \mid \rho' : Var \rightarrow M,\ \rho'{\restriction}_{Var \setminus \{x\}} = \rho{\restriction}_{Var \setminus \{x\}}\}$

▶ **Definition 4.** Model M *satisfies* φ_s, written $M \models \varphi_s$, iff $\overline{\rho}(\varphi_s) = M_s$ for all $\rho : Var \to M$.

▶ **Proposition 5.** *The following properties hold:*

- *If* $\rho_1, \rho_2 : Var \to M$, $\rho_1{\restriction_{FV(\varphi)}} = \rho_2{\restriction_{FV(\varphi)}}$ *then* $\overline{\rho_1}(\varphi) = \overline{\rho_2}(\varphi)$
- *If* $x \in Var_s$ *then* $M \models x$ *iff* $|M_s| = 1$
- *If* $\sigma \in \Sigma_{s_1 \ldots s_n, s}$ *and* $\varphi_1, \ldots, \varphi_n$ *are patterns of sorts* $s_1, \ldots, s_n$*, respectively, then we have* $M \models \sigma(\varphi_1, \ldots, \varphi_n)$ *iff* $\sigma_M(\overline{\rho}(\varphi_1), \ldots, \overline{\rho}(\varphi_n)) = M_s$ *for any* $\rho : Var \to M$
- $M \models \neg\varphi$ *iff* $\overline{\rho}(\varphi) = \emptyset$ *for any* $\rho : Var \to M$
- $M \models \varphi_1 \wedge \varphi_2$ *iff* $M \models \varphi_1$ *and* $M \models \varphi_2$
- *If* $\exists x.\varphi_s$ *is closed, then* $M \models \exists x.\varphi_s$ *iff* $\bigcup\{\overline{\rho}(\varphi_s) \mid \rho : Var \to M\} = M_s$*; hence,* $M \models \exists x.x$
- $M \models \varphi_1 \to \varphi_2$ *iff* $\overline{\rho}(\varphi_1) \subseteq \overline{\rho}(\varphi_2)$ *for all* $\rho : Var \to M$
- $M \models \varphi_1 \leftrightarrow \varphi_2$ *iff* $\overline{\rho}(\varphi_1) = \overline{\rho}(\varphi_2)$ *for all* $\rho : Var \to M$
- $M \models \forall x.\varphi$ *iff* $M \models \varphi$

Note that property "if φ closed then $M \models \neg\varphi$ iff $M \not\models \varphi$", which holds in FOL, does not hold in matching logic. Indeed, suppose φ is a constant symbol, say 0, of sort s. Then $M \models \neg 0$ is equivalent to $0_M = \emptyset$, while $M \not\models 0$ is equivalent to $0_M \neq M_s$.

▶ **Definition 6.** Pattern φ is *valid*, written $\models \varphi$, iff $M \models \varphi$ for all M. If $F \subseteq$ Pattern then $M \models F$ iff $M \models \varphi$ for all $\varphi \in F$. F *entails* φ, written $F \models \varphi$, iff for all M, we have $M \models F$ implies $M \models \varphi$. A *matching logic specification* is a triple (S, Σ, F) with $F \subseteq$ Pattern.

3 Reduction to Predicate Logic

It is known that FOL formulae can be translated into equivalent predicate logic formulae, by replacing each function symbol with a predicate symbol and then systematically transforming terms into formulae. We can similarly translate patterns into equivalent predicate logic formulae. Consider pure predicate logic with equality and no constants, whose satisfaction relation is $\models^{=}_{PL}$. If (S, Σ) is a matching logic signature, let (S, Π_Σ) be the predicate logic signature with $\Pi_\Sigma = \{\pi_\sigma : s_1 \times \cdots \times s_n \times s \mid \sigma \in \Sigma_{s_1 \ldots s_n, s}\}$. We define the translation PL of matching logic (S, Σ)-patterns into predicate logic (S, Π_Σ)-formulae inductively as follows:

$$PL(\varphi) = \forall r \,.\, PL_2(\varphi, r)$$

$$\begin{aligned}
PL_2(x, r) &= (x = r) \\
PL_2(\sigma(\varphi_1, \ldots, \varphi_n), r) &= \exists r_1 \cdots \exists r_n \,.\, PL_2(\varphi_1, r_1) \wedge \cdots \wedge PL_2(\varphi_n, r_n) \wedge \pi_\sigma(r_1, \ldots, r_n, r) \\
PL_2(\neg\varphi, r) &= \neg PL_2(\varphi, r) \\
PL_2(\varphi_1 \wedge \varphi_2, r) &= PL_2(\varphi_1, r) \wedge PL_2(\varphi_2, r) \\
PL_2(\exists x \,.\, \varphi, r) &= \exists x \,.\, PL_2(\varphi, r)
\end{aligned}$$

$$PL(\{\varphi_1, \ldots, \varphi_n\}) = \{PL(\varphi_1), \ldots, PL(\varphi_n)\}$$

Then the following result holds, like for FOL:

▶ **Proposition 7.** *If* F *is a set of patterns and* φ *is a pattern, then* $F \models \varphi$ *iff* $PL(F) \models^{=}_{PL} PL(\varphi)$

Proposition 7 gives a sound and complete procedure for matching logic reasoning: translate the specification (S, Σ, F) and pattern to prove φ into the predicate logic specification $(S, \Pi_\Sigma, PL(F))$ and formula $PL(\varphi)$, respectively, and then derive it using the sound and complete proof system of predicate logic. However, translating patterns to predicate logic formulae makes reasoning harder not only for humans, but also for machines, since new

quantifiers are introduced. For example, $(1 \mapsto 5*2 \mapsto 0*7 \mapsto 9*8 \mapsto 1) \to \mathit{list}(7, 9{\cdot}5)$ discussed and proved in Section 1, translates into the formula (to keep it small, we do not translate the numbers) $\forall r\,.\,(\exists r_1\,.\,\exists r_2\,.\,\pi_{\mapsto}(1,5,r_1) \wedge (\exists r_3\,.\,\exists r_4\,.\,\pi_{\mapsto}(2,0,r_3) \wedge (\exists r_5\,.\,\exists r_6\,.\,\pi_{\mapsto}(7,9,r_5) \wedge \pi_{\mapsto}(8,1,r_6) \wedge \pi_*(r_5,r_6,r_4)) \wedge \pi_*(r_3,r_4,r_2)) \wedge \pi_*(r_1,r_2,r)) \to \exists r_7\,.\,\pi.(9,5,r_7) \wedge \pi_{\mathit{list}}(7,r_7,r)$. What we would like is to reason directly with matching logic patterns, the same way we reason directly with terms in FOL without translating them to predicate logic.

▶ **Proposition 8.** *The following hold for matching logic:*

1. $\models \varphi$, *where* φ *is a propositional tautology (over patterns)*
2. *Modus ponens:* $\models \varphi_1$ *and* $\models \varphi_1 \to \varphi_2$ *implies* $\models \varphi_2$
3. $\models (\forall x\,.\,\varphi_1 \to \varphi_2) \to (\varphi_1 \to \forall x\,.\,\varphi_2)$ *when* $x \notin FV(\varphi_1)$
4. *Universal generalization:* $\models \varphi$ *implies* $\models \forall x\,.\,\varphi$

Proposition 8 states that the proof system of pure predicate logic is actually sound for matching logic *as is*. Section 5 shows that a few additional proof rules yield a sound and complete proof system for matching logic, similarly to how Substitution ($\forall x\,.\,\varphi \to \varphi[t/x]$) together with the four proof rules of pure predicate logic brings complete deduction to FOL. But before that, we demonstrate the usefulness of matching logic by a series of examples.

4 Examples and Notations

We have already seen some simple patterns in Section 2, such as $\exists x.x$ (satisfied by all models) and $\forall x.x$ (satisfied only by models whose carrier of the sort of x contains only one element). Here we illustrate matching logic by means of a series of more complex examples.

4.1 Propositional logic

If S contains only one sort Prop, Σ is empty, and we drop the existential quantifier, then the syntax of matching logic becomes that of propositional calculus: $\varphi ::= \mathit{Var}_{\mathit{Prop}} \mid \neg\varphi \mid \varphi \wedge \varphi$.

▶ **Proposition 9.** *For any proposition* φ*, the following holds:* $\models_{\mathit{Prop}} \varphi$ *iff* $\models \varphi$*.*

An alternative way to capture propositional logic is to add a constant symbol to Σ for each propositional variable, and then associate a ground pattern to each proposition. Proposition 9 still holds, despite the fact that propositional constants can be interpreted as arbitrary sets. That is since $(\mathcal{P}(M), \neg_M, \cap)$ is a model of propositional logic for any set M.

4.2 Pure predicate logic

If S is a sort set and Π is a set of predicate symbols, the syntax of pure predicate logic formulae (without equality) is $\varphi ::= \pi(x_1, \ldots, x_n)$ with $\pi \in \Pi_{s_1 \ldots s_n} \mid \neg\varphi \mid \varphi \wedge \varphi \mid \exists x.\varphi$. We can pick a new sort name, Pred, and construct a matching logic signature $(S \cup \{\mathit{Pred}\}, \Sigma)$ where $\Sigma_{s_1 \ldots s_n, \mathit{Pred}} = \Pi_{s_1 \ldots s_n}$. Then any predicate logic formula can be trivially regarded as a matching logic pattern. The following result then holds:

▶ **Proposition 10.** *For any predicate logic formula* φ*, the following holds:* $\models_{PL} \varphi$ *iff* $\models \varphi$*.*

4.3 Definedness, Equality, Membership

Pattern definedness, equality and membership can be defined in matching logic, without any special support or logic extensions. Let us first discuss why we cannot use $\leftrightarrow$ as equality. Indeed, since $M \models \varphi_1 \leftrightarrow \varphi_2$ iff $\overline{\rho}(\varphi_1) = \overline{\rho}(\varphi_2)$ for all $\rho : \mathit{Var} \to M$, one may be tempted to

do so. E.g., given a signature with one sort and one unary symbol f, one may think that pattern $\exists y \,.\, f(x) \leftrightarrow y$ defines precisely the models where f is a function. Unfortunately, that is not true. Consider model M with $M = \{1,2\}$ and f_M the non-function $f_M(1) = \{1,2\}$, $f_M(2) = \emptyset$. Let $\rho : \mathit{Var} \to M$; recall (Definition 3) that ρ'extension $\overline{\rho}$ to patterns interprets "$\exists$" as union and "$\leftrightarrow$" as the complement of the symmetric difference. If $\rho(x) = 1$ then $\overline{\rho}(\exists y \,.\, f(x) \leftrightarrow y) = (M \backslash (\{1,2\}\Delta\{1\})) \cup (M \backslash (\{1,2\}\Delta\{2\})) = \{1,2\} = M$. If $\rho(x) = 2$ then $\overline{\rho}(\exists y \,.\, f(x) \leftrightarrow y) = (M \backslash (\emptyset\Delta\{1\})) \cup (M \backslash (\emptyset\Delta\{2\})) = \{1,2\} = M$. Hence, $M \models \exists y \,.\, f(x) \leftrightarrow y$.

The problem above was that the interpretation of $\varphi_1 \leftrightarrow \varphi_2$ is not equivalent to either $\top$ or $\bot$, as we are used to think in FOL. Specifically, $\overline{\rho}(\varphi_1) \neq \overline{\rho}(\varphi_2)$ does not suffice for $\overline{\rho}(\varphi_1 \leftrightarrow \varphi_2) = \emptyset$ to hold. Indeed, $\overline{\rho}(\varphi_1 \leftrightarrow \varphi_2) = M \setminus (\overline{\rho}(\varphi_1) \;\Delta\; \overline{\rho}(\varphi_2))$ and there is nothing to prevent, e.g., $\overline{\rho}(\varphi_1) \cap \overline{\rho}(\varphi_2) \neq \emptyset$, in which case $\overline{\rho}(\varphi_1) \;\Delta\; \overline{\rho}(\varphi_2) \neq M$. What we would like to have is a proper equality, $\varphi_1 = \varphi_2$, which behaves like a predicate: $\overline{\rho}(\varphi_1 = \varphi_2) = \emptyset$ when $\overline{\rho}(\varphi_1) \neq \overline{\rho}(\varphi_2)$, and $\overline{\rho}(\varphi_1 = \varphi_2) = M$ when $\overline{\rho}(\varphi_1) = \overline{\rho}(\varphi_2)$. Moreover, we want equalities to be used with terms of any sort, and in contexts of any sort.

The above can be achieved methodologically in matching logic, by adding to the signature a *definedness* symbol $\lceil _ \rceil_{s_1}^{s_2} \in \Sigma_{s_1,s_2}$ for any sorts s_1 and s_2, together with the pattern axiom $\lceil x{:}s_1 \rceil_{s_1}^{s_2}$ enforcing $(\lceil _ \rceil_{s_1}^{s_2})_M(m_1) = M_{s_2}$ in all models M for all $m_1 \in M_{s_1}$, that is, for any $\rho : \mathit{Var} \to M$, $\overline{\rho}(\lceil \varphi \rceil_{s_1}^{s_2})$ is either $\emptyset$ when $\overline{\rho}(\varphi) = \emptyset$ (i.e., φ undefined in ρ), or is M_{s_2} when $\overline{\rho}(\varphi) \neq \emptyset$ (i.e., φ defined). We can now use $_ =_{s_1}^{s_2} _$ and $_ \in_{s_1}^{s_2} _$, respectively, as aliases:

$$\begin{array}{rcll} \varphi =_{s_1}^{s_2} \varphi' & \equiv & \neg\lceil \neg(\varphi \leftrightarrow \varphi') \rceil_{s_1}^{s_2} & \text{where } \varphi, \varphi' \in \textsc{Pattern}_{s_1} \\ x \in_{s_1}^{s_2} \varphi & \equiv & \lceil x \wedge \varphi \rceil_{s_1}^{s_2} & \text{where } x \in \mathit{Var}_{s_1}, \varphi \in \textsc{Pattern}_{s_1} \end{array}$$

▶ **Proposition 11.** *With the above, the following hold:*
1. $\overline{\rho}(\varphi =_{s_1}^{s_2} \varphi') = \emptyset$ *iff* $\overline{\rho}(\varphi) \neq \overline{\rho}(\varphi')$*, and* $\overline{\rho}(\varphi =_{s_1}^{s_2} \varphi') = M_{s_2}$ *iff* $\overline{\rho}(\varphi) = \overline{\rho}(\varphi')$
2. $\models \varphi =_{s_1}^{s_2} \varphi'$ *iff* $\models \varphi \leftrightarrow \varphi'$
3. $\overline{\rho}(x \in_{s_1}^{s_2} \varphi) = \emptyset$ *iff* $\rho(x) \notin \overline{\rho}(\varphi)$*; and* $\overline{\rho}(x \in_{s_1}^{s_2} \varphi) = M_{s_2}$ *iff* $\rho(x) \in \overline{\rho}(\varphi)$
4. $\models (x \in_{s_1}^{s_2} \varphi) =_{s_2}^{s_3} (x \wedge \varphi =_{s_1}^{s_2} x)$

From now on we assume equality and membership in all specifications, without mentioning the constructions above. Moreover, since s_1 and s_2 can usually be inferred from context, we write $\lceil _ \rceil$, $=$ and $\in$ instead of $\lceil _ \rceil_{s_1}^{s_2}$, $=_{s_1}^{s_2}$, and $\in_{s_1}^{s_2}$, respectively. If the sort decorations cannot be inferred from context, then we assume the stated property/axiom/rule holds for all such sorts. For example, the generic pattern axiom "$\lceil x \rceil$ where $x \in \mathit{Var}$" replaces all the axioms $\lceil x{:}s_1 \rceil_{s_1}^{s_2}$ above for the definedness symbol, for all the sorts s_1 and s_2. Similarly, the axiom in Section 4.7 defining list patterns within maps, $\mathit{list}(x) = (x = 0 \wedge \mathit{emp} \vee \exists z \,.\, x \mapsto z * \mathit{list}(z))$, is equivalent to the explicit axioms (for all sorts s), $\mathit{list}(x) =_{\mathit{Map}}^{s} (x =_{\mathit{Nat}}^{\mathit{Map}} 0 \wedge \mathit{emp} \vee \exists z \,.\, x \mapsto z * \mathit{list}(z))$.

Proposition 8 showed that four of the proof rule/axiom schemas of FOL are already sound for matching logic. The soundness of several others are shown below, essentially stating the soundness of the matching logic proof system, except one rule, Substitution (Section 5):

▶ **Proposition 12.** *The following hold:*
1. *Equality introduction:* $\models \varphi = \varphi$
2. *Equality elimination:* $\models \varphi_1 = \varphi_2 \wedge \varphi[\varphi_1/x] \to \varphi[\varphi_2/x]$
3. $\models \forall x \,.\, x \in \varphi$ *iff* $\models \varphi$
4. $\models (x \in y) = (x = y)$ *when* $x, y \in \mathit{Var}$
5. $\models (x \in \neg\varphi) = \neg(x \in \varphi)$
6. $\models (x \in \varphi_1 \wedge \varphi_2) = (x \in \varphi_1) \wedge (x \in \varphi_2)$
7. $\models (x \in \exists y.\varphi) = \exists y.(x \in \varphi)$*, with* x *and* y *distinct*
8. $\models x \in \sigma(\varphi_1, ..., \varphi_{i-1}, \varphi_i, \varphi_{i+1}, ..., \varphi_n) = \exists y.(y \in \varphi_i \wedge x \in \sigma(\varphi_1, ..., \varphi_{i-1}, y, \varphi_{i+1}, ..., \varphi_n))$

4.4 Defining special relations

Here we show how to define special relations using patterns. For example, $\exists y \,.\, \sigma(x_1, \ldots, x_n) = y$ states that $\sigma \in \Sigma_{s_1 \ldots s_n, s}$ is a function in all models. Indeed, if M is any model satisfying the pattern above and $a_1 \in M_{s_1}, \ldots, a_n \in M_{s_n}$ then let $\rho : Var \to M$ be an M-valuation such that $\rho(x_1) = a_1, \ldots, \rho(x_n) = a_n$. Since M satisfies the pattern, it follows that $M_s = \bigcup\{\overline{\rho'}(\sigma(x_1, \ldots, x_n) = y) \mid \rho' : Var \to M,\ \rho'\restriction_{Var \setminus \{y\}} = \rho\restriction_{Var \setminus \{y\}}\}$. Since $\overline{\rho'}(\sigma(x_1, \ldots, x_n) = y)$ is either M_s or $\emptyset$, depending upon whether $\sigma_M(x_1, \ldots, x_n) = \{\rho'(y)\}$ holds or not, we conclude that there exists some $\rho' : Var \to M$ such that $\sigma_M(a_1, \ldots, a_n) = \{\rho'(y)\}$, that is, $\sigma_M(a_1, \ldots, a_n)$ is a one-element set. Therefore, σ_M represents a total function. To avoid writing such boring function patterns, from now on we automatically assume such an axiom whenever we write a symbol $\sigma \in \Sigma_{s_1 \ldots s_n, s}$ using the function notation $\sigma : s_1 \times \cdots \times s_n \to s$.

Pattern $(f(x) = f(y)) \to (x = y)$ states that f is injective. If $(M, f_M : M \to M)$ is any model satisfying this specification, then f_M must be injective. Indeed, let $a, b \in M$ such that $a \neq b$ and $f_M(a) = f_M(b)$. Pick $\rho : Var \to M$ such that $\rho(x) = a$ and $\rho(y) = b$. Since M satisfies the axiom above, we get $\overline{\rho}(f(x) = f(y)) \subseteq \overline{\rho}(x = y)$. But Proposition 11 implies that $\overline{\rho}(x = y) = \emptyset$ and $\overline{\rho}(f(x) = f(y)) = M$, which is a contradiction. We can also show that any model whose f is injective satisfies the axiom. Let $(M, f_M : M \to M)$ be any model such that f_M is injective. It suffices to show $\overline{\rho}(f(x) = f(y)) \subseteq \overline{\rho}(x = y)$ for any $\rho : Var \to M$, which follows by Proposition 11: if $\rho(x) = \rho(y)$ then $\overline{\rho}(f(x) = f(y)) = \overline{\rho}(x = y) = M$, and if $\rho(x) \neq \rho(y)$ then $\overline{\rho}(f(x) = f(y)) = \overline{\rho}(x = y) = \emptyset$ because f_M is injective.

From here on in the rest of the paper we take the freedom to write $\varphi \neq \varphi'$ instead of $\neg(\varphi = \varphi')$. With this, another way to capture the injectivity of f is $(x \neq y) \to (f(x) \neq f(y))$.

Pattern $(\sigma(x_1, \ldots, x_n) = \bot_s) \vee \exists y \,.\, \sigma(x_1, \ldots, x_n) = y$ states that $\sigma \in \Sigma_{s_1 \ldots s_n, s}$ is a partial function, and from now on we use the notation (note the "$\rightharpoonup$" symbol) $\sigma : s_1 \times \cdots \times s_n \rightharpoonup s$ to automatically assume a pattern like the above for σ. For example, a division partial function which is undefined in all models when the denominator is 0 can be specified as:

$$_/_ : Nat \times Nat \rightharpoonup Nat \qquad\qquad \neg(x/0)$$

i.e., as a symbol $_/_ \in \Sigma_{Nat \times Nat, Nat}$ with patterns $(x/y = \bot_{Nat}) \vee \exists z \,.\, x/y = z$ and $\neg(x/0)$.

Total relations can be defined with $\lceil \sigma(x_1, \ldots, x_n) \rceil_s^s$, equivalent to $\sigma(x_1, \ldots, x_n) \neq \bot_s$. We write $\sigma : s_1 \times \cdots \times s_n \Rightarrow s$ to automatically state that σ is a total relation.

4.5 Algebraic specifications and matching logic modulo theories

An algebraic specification is a many-sorted signature (S, Σ) together with a set of equations E over Σ-terms with variables. To translate an algebraic specification into a matching logic specification we only need to ensure that symbols get a function interpretation as described in Section 4.4, and to regard each equation $t = t'$ as an equality pattern $t = t'$.

▶ **Proposition 13.** *Let (S, Σ, F) be the matching logic specification associated to the algebraic specification (S, Σ, E) as above. Then for any Σ-equation e, we have $E \models_{alg} e$ iff $F \models e$.*

Using the notations introduced so far, Peano natural numbers can be defined as follows:

$$\begin{array}{lll} 0 : \to Nat & succ : Nat \to Nat & plus : Nat \times Nat \to Nat \\ plus(0, y) = y & & plus(succ(x), y) = succ(plus(x, y)) \end{array}$$

This looks identical to the conventional algebraic specification definition.

Note, however, that matching logic allows us to add more than just equational patterns. For example, we can add to F the pattern $0 \vee \exists x \,.\, succ(x)$ stating that any number is either 0

or the successor of another number. Nevertheless, since matching logic ultimately has the same expressive power as predicate logic (Proposition 7), we cannot finitely axiomatize in matching logic any mathematical domains that do not already admit finite FOL axiomatizations. In practice, we follow the same standard approach as the first-order SMT solvers, namely desired domains are theoretically presented with potentially infinitely many axioms but are implemented using specialized decision procedures. Indeed, our current matching logic implementation prototype in $\mathbb{K}$ defers to Z3 [5] the solving of all the domain constraints.

Algebraic specifications and decision procedures of mathematical domains abound in the literature. All of these can be used in the context of matching logic. We do not discuss these further, but only mention that from now on we tacitly assume definitions of integer and of natural numbers, as well as of Boolean values, with common operations on them. We assume that these come with three sorts, *Int*, *Nat* and *Bool*, and the operations on them use the conventional syntax and writing; e.g., $_ \leq _ : \mathit{Nat} \times \mathit{Nat} \to \mathit{Bool}$, $x \leq y$, etc. To compact writing, we take the freedom to write b instead of $b = \mathit{true}$ for Boolean expressions b, in any sort context. For example, we write $\varphi_s \wedge x \leq y$ instead of $\varphi_s \wedge (x \leq y =^s_{\mathit{Bool}} \mathit{true})$.

4.6 Sequences, Multisets and Sets

Sequences, multisets and sets are typical ADTs. Matching logic enables, however, some useful developments and shortcuts. For simplicity, we only discuss collections over *Nat*, and name the corresponding sorts *Seq*, *MultiSet*, and *Set*. Ideally, we would build upon an order-sorted algebraic signature setting, so that we can regard $x : \mathit{Nat}$ not only as an element of sort *Nat*, but also as one of sort *Seq* (a one-element sequence), as one of sort *MultiSet*, as well as one of sort *Set*. Extending matching logic to an order-sorted setting is not difficult, but would deviate from our main objective in this paper, so we refrain from doing it. Instead, we rely on the reader to assume either that order-sortedness does not bring complications (besides those of order-sortedness itself in the context of algebraic specification) or that elements of sort *Nat* used in a *Seq*, *MultiSet*, or *Set* context are wrapped with injection symbols.

Sequences can be defined with two symbols and corresponding equations:

$$\epsilon : \to \mathit{Seq} \qquad _ \cdot _ : \mathit{Seq} \times \mathit{Seq} \to \mathit{Seq} \qquad \epsilon \cdot x = x \qquad x \cdot \epsilon = x \qquad (x \cdot y) \cdot z = x \cdot (y \cdot z)$$

We assume that lowercase variables have sort *Nat*, and uppercase ones have the appropriate collection sort. To avoid adding initiality constraints on models yet be able to do proofs by case analysis and elementwise equality, we may add $\epsilon \vee \exists x \,.\, \exists S.\, x \cdot S$ and $(x \cdot S = x' \cdot S') = (x = x') \wedge (S = S')$ as pattern axioms. We next define some operations on sequences:

$$\begin{array}{llll} \mathit{rev} : \mathit{Seq} \to \mathit{Seq} & \mathit{rev}(\epsilon) = \epsilon & \neg(x \in \epsilon) & x \in x \cdot S \\ _ \in _ : \mathit{Nat} \times \mathit{Seq} \to \mathit{Bool} & \mathit{rev}(x \cdot S) = \mathit{rev}(S) \cdot x & x \in y \cdot S \wedge (x \neq y) = x \in S & \end{array}$$

We can transform sequences into multisets adding the equality axiom $x \cdot y = y \cdot x$, and into sets by also including $x \cdot x = \bot$ or $x \cdot x = x$. Here is one way to axiomatize intersection:

$$_ \cap _ : \mathit{Set} \times \mathit{Set} \to \mathit{Set} \qquad \epsilon \cap S_2 = \epsilon \qquad (x \cdot S_1) \cap S_2 = ((x \in S_2 \to x) \wedge (\neg(x \in S_2) \to \epsilon)) \cdot (S_1 \cap S_2)$$

4.7 Maps and Map Patterns

Finite-domain maps are also a typical example of an ADT. We only discuss maps from natural numbers to natural numbers, but they can be similarly defined over arbitrary domains as

keys and as values. We use a syntax for maps that resembles that of separation logic [11]:

$$\begin{array}{ll}
_ \mapsto _ : \mathit{Nat} \times \mathit{Nat} \rightharpoonup \mathit{Map} & \mathit{emp} * H = H \\
\mathit{emp} : \to \mathit{Map} & H_1 * H_2 = H_2 * H_1 \\
_ * _ : \mathit{Map} \times \mathit{Map} \rightharpoonup \mathit{Map} & (H_1 * H_2) * H_3 = H_1 * (H_2 * H_3) \\
0 \mapsto a = \bot & x \mapsto a * x \mapsto b = \bot
\end{array}$$

When regarding the above ADT as a matching logic specification, we can prove that the bottom two pattern equations above are equivalent to $\neg(0 \mapsto a)$ and, respectively, $(x \mapsto a * y \mapsto b) \to x \neq y$, giving the $_ \mapsto _$ and $_ * _$ the feel of "predicates".

Consider the canonical model of partial maps M, where: $M_{\mathit{Nat}} = \{0, 1, 2, \ldots\}$; $M_{\mathit{Map}} =$ partial maps from natural numbers to natural numbers with finite domains and undefined in 0, with emp interpreted as the map undefined everywhere, with $_ \mapsto _$ interpreted as the corresponding one-element partial map except when the first argument is 0 in which case it is undefined (note that $_ \mapsto _$ was declared using $\rightharpoonup$), and with $_ * _$ interpreted as map merge when the two maps have disjoint domains, or undefined otherwise (note that $_ * _$ was also declared using $\rightharpoonup$). M satisfies all axioms above.

We next define two common patterns, for complete linked lists and for list fragments:

$$\begin{array}{ll}
\mathit{list} : \mathit{Nat} \Rightarrow \mathit{Map} & \mathit{lseg} : \mathit{Nat} \times \mathit{Nat} \Rightarrow \mathit{Map} \\
\mathit{list}(0) = \mathit{emp} & \mathit{lseg}(x, x) = \mathit{emp} \\
\mathit{list}(x) \wedge x \neq 0 = \exists z \,.\, x \mapsto z * \mathit{list}(z) & \mathit{lseg}(x, y) \wedge x \neq y = \exists z \,.\, x \mapsto z * \mathit{lseg}(z, y)
\end{array}$$

It can be shown that in the model M of partial maps described above, there is a unique way to interpret *list* and *lseg*, namely as the expected linked lists and, respectively, linked list fragments. Specifically, we can show that $\mathit{lseg}_M : M_{\mathit{Nat}} \times M_{\mathit{Nat}} \to \mathcal{P}(M_{\mathit{Map}})$ (we only discuss *lseg*, because *list* is similar and simpler) can only be the following function:

$$\begin{array}{l}
\mathit{lseg}_M(n, n) = \{\mathit{emp}_M\} \text{ for all } n \geq 0 \\
\mathit{lseg}_M(n, m) = \{\; n \mapsto_M n_1 *_M n_1 \mapsto_M n_2 *_M \cdots *_M n_k \mapsto_M m \\
\qquad \qquad \mid k \geq 0, \text{ and } n_0 = n, n_1, n_2, \ldots, n_k > 0 \text{ all different}\}
\end{array}$$

Complete details can be found in [15].

It should be clear that patterns can be specified in many different ways. E.g., the first list pattern can also be specified as $\mathit{list}(x) = (x = 0 \wedge \mathit{emp} \vee \exists z \,.\, x \mapsto z * \mathit{list}(z))$. In a similar style, we can define more complex patterns, such as lists with data. But first, we specify a convenient operation for defining maps over contiguous keys, making use of a sequence data-type. The latter can be defined like in Section 4.6; for notational convenience, we take the freedom to use comma "," instead of "$\cdot$" for sequence concatenation in some places:

$$_ \mapsto [_] : \mathit{Nat} \times \mathit{Seq} \to \mathit{Map} \qquad x \mapsto [\epsilon] = \mathit{emp} \qquad x \mapsto [a, S] = x \mapsto a * (x+1) \mapsto [S]$$

In our model M, we can take M_{Seq} to be the finite sequences of natural numbers, with ϵ_M and $_ \cdot_M _$ interpreted as the empty sequence and, respectively, sequence concatenation.

We can now define lists with data as follows:

$$\begin{array}{ll}
\mathit{list} : \mathit{Nat} \times \mathit{Seq} \Rightarrow \mathit{Map} & \mathit{lseg} : \mathit{Nat} \times \mathit{Seq} \times \mathit{Nat} \Rightarrow \mathit{Map} \\
\mathit{list}(0, \epsilon) = \mathit{emp} & \mathit{lseg}(x, \epsilon, x) = \mathit{emp} \\
\mathit{list}(x, n \cdot S) = \exists z \,.\, x \mapsto [n, z] * \mathit{list}(z, S) & \mathit{lseg}(x, n \cdot S, y) = \exists z \,.\, x \mapsto [n, z] * \mathit{lseg}(z, S, y)
\end{array}$$

Note that, unlike in the case of lists without data, this time we have not required the side conditions $x \neq 0$ and $x \neq y$, respectively. The side conditions were needed in the former

case because without them we can infer, e.g., $\mathit{list}(0) = \bot$ (from the second equation of *list*), which using the first equation would imply $\mathit{emp} = \bot$. However, they are not needed in the latter case because it is safe (and even desired) to infer $\mathit{list}(0, n \cdot S) = \bot$ for any n and S. We can show, using a similar approach like for lists without data, that the pattern $\mathit{lseg}(x, S, y)$ matches in M precisely the lists starting with x, exiting to y, and holding data sequence S.

We can similarly define other data-type specifications, such as trees with data:

$$\mathit{none} : \to \mathit{Tree} \qquad \mathit{node} : \mathit{Nat} \times \mathit{Tree} \times \mathit{Tree} \to \mathit{Tree} \qquad \mathit{tree} : \mathit{Nat} \times \mathit{Tree} \Rightarrow \mathit{Map}$$
$$\mathit{tree}(0, \mathit{none}) = \mathit{emp} \qquad \mathit{tree}(x, \mathit{node}(n, t_1, t_2)) = \exists y\, z\,.\, x \mapsto [n, y, z] * \mathit{tree}(y, t_1) * \mathit{tree}(z, t_2))$$

Therefore, fixing the interpretations of the basic mathematical domains, such as those of natural numbers, sequences, maps, etc., suffices in order to define interesting map patterns that appear in verification of heap properties of programs, in the sense that the axioms themselves uniquely define the desired data-types. No inductive predicates or principles were needed to define them, although induction or initiality may be needed in order to define the desired models. Choosing the right basic mathematical domains is, however, crucial. For example, if we allow the maps in M_{Map} to have infinite domains then the list patterns without data above (the first ones) also include infinite lists. The lists with data cannot include infinite lists, because we only allow finite sequences. This would, of course, change if we allow infinite sequences, or streams, in the model. In that case, *list* and *lseg* would not admit unique interpretations anymore, because we can interpret them to be either all the finite domain lists, or both the finite and the infinite-domain lists. Writing patterns which admit the desired solution in the desired model suffices in practice; our reasoning techniques developed in the sequel allow us to derive properties that hold in all models satisfying the axioms, so any derived property is sound also for the intended model and interpretations.

4.8 First-Order Logic

First-order logic (FOL) allows both function and predicate symbols:

$$\begin{array}{lcl} t_s & ::= & x \in \mathit{Var}_s \;\mid\; \sigma(t_1, \ldots, t_n) \text{ with } \sigma \in \Sigma_{s_1 \ldots s_n, s} \\ \varphi & ::= & \pi(x_1, \ldots, x_n) \text{ with } \pi \in \Pi_{s_1 \ldots s_n} \;\mid\; \neg\varphi \;\mid\; \varphi \wedge \varphi \;\mid\; \exists x . \varphi \end{array}$$

Let (S, Σ, Π) be a FOL signature. Like in pure predicate logic, we add a *Pred* sort and regard the predicate symbols as symbols of result *Pred*. Let (S^{ML}, Σ^{ML}) be the matching logic signature with $S^{ML} = S \cup \{\mathit{Pred}\}$ and $\Sigma^{ML} = \Sigma \cup \{\pi : s_1 \ldots s_n \to \mathit{Pred} \mid \pi \in \Pi_{s_1 \ldots s_n}\}$, and let F be $\{\exists z{:}s\,.\,\sigma(x_1{:}s_1, \ldots, x_n{:}s_n) = z \mid \sigma \in \Sigma_{s_1 \ldots s_n, s}\}$ saying each symbol is a function.

▶ **Proposition 14.** *For any FOL formula φ, we have $\models_{FOL} \varphi$ iff $F \models \varphi$.*

4.9 Separation Logic

Matching logic has inherent support for structural separation, without a need for any special logic constructs or extensions. That is because pattern matching has a spatial meaning by its very nature: matching a subterm already separates that subterm from the rest of the context, so matching two or more terms can only happen when there is no overlapping between them.

Separation logic [11, 14, 12] is a logic for reasoning about heap structures. There are many variants, but here we only consider the one in [11]. Its syntax extends FOL as follows:

$$\varphi \;::=\; \textit{(FOL syntax)} \;\mid\; \mathit{emp} \;\mid\; \mathit{Int} \mapsto \mathit{Int} \;\mid\; \varphi * \varphi \;\mid\; \varphi \mathrel{-\!\!*} \varphi$$

Its semantics is based on a fixed model of stores and heaps, which are finite-domain maps from variables and locations (particular integers), respectively, to integers. The semantics of each

construct is given in terms of a pair (s,h) of a store and a heap, called a *state*. For example, $(s,h) \models_{SL} E_1 \mapsto E_2$ iff $Dom(h) = \overline{s}(E_1)$ and $h(\overline{s}(E_1)) = \overline{s}(E_2)$, and $(s,h) \models_{SL} P_1 * P_2$ iff there exist h_1 and h_2 of disjoint domains such that $h = h_1 * h_2$ and $(s,h_1) \models_{SL} P_1$ and $(s,h_2) \models_{SL} P_2$. The semantics of "magic wand", $P_1 {-\!\!*}\, P_2$ is defined as the states whose heaps extended with a fragment satisfying P_1 result in ones satisfying P_2: $(s,h) \models_{SL} P_1 {-\!\!*}\, P_2$ iff for any h_1 of domain disjoint of h's, if $(s,h_1) \models_{SL} P_1$ then $(s, h * h_1) \models_{SL} P_2$.

We can define a matching logic specification and a model of it, which precisely capture separation logic. The FOL constructs are already captured by the generic syntax of patterns as explained in previous sections. The spatial constructs, except for the ${-\!\!*}$, are given by the matching logic specification of maps discussed in Section 4.7, in which we substitute *Int* for *Nat*. For the magic wand, we add the partial function $_{-\!\!*}_ : Map \times Map \rightharpoonup Map$ and the pattern $P_1 {-\!\!*}\, P_2 = \exists H \,.\, H \wedge [H * P_1 \to P_2]$. Recall from Section 4.3 that $[_]$ leverages the non-emptyness of its argument to the total set. In words, $P_1 {-\!\!*}\, P_2$ is the set of all maps h which merged with maps satisfying P_1 yield only maps satisfying P_2. With the above, any separation logic formula can be regarded, *as is*, as a matching logic pattern of sort *Map*.

We next construct our model. Let M be identical to the model for maps in Section 4.7, except that we replace natural numbers with integer numbers. The only thing left is to define the partial function $_{-\!\!*}_M_ : Map \times Map \to \mathcal{P}(Map)$, which we do as follows: $h_1 {-\!\!*}_M h_2 = \{h \mid Dom(h) \cap Dom(h_1) = \emptyset \text{ and } h *_M h_1 = h_2\}$. Note that $h_1 {-\!\!*}_M h_2$ is either the empty set or it is a set of precisely one map. Then the following result holds:

▶ **Proposition 15.** *If φ is a separation logic formula, then $\models_{SL} \varphi$ iff $M \models \varphi$. More specifically, for any store s and any heap h, we have $(s,h) \models_{SL} \varphi$ iff $h \in \overline{s}(\varphi)$.*

5 Sound and Complete Deduction

As shown in Section 3, the proof system of predicate logic is sound for matching logic as is. Ideally, we would like the same to hold true for FOL with equality, that is, we would like its proof system to be sound as is for matching logic reasoning, where we replace terms and predicates with arbitrary patterns. Unfortunately, FOL's Substitution axiom, $(\forall x \,.\, \varphi) \to \varphi[t/x]$, is not sound if we replace t with any pattern. For example, consider the tautology $\forall x \,.\, \exists y \,.\, x = y$ and let φ be $\exists y \,.\, x = y$. If FOL's Substitution were sound for arbitrary patterns φ' instead of t, then the formula $\exists y \,.\, \varphi' = y$, stating that φ' evaluates to a unique element for any valuation, would be valid for any pattern φ'. However, this is not true in matching logic, because patterns can evaluate to any set of elements, including the empty set or the total set; several examples of such patterns were discussed in Section 4. We need to modify Substitution to indicate that φ' admits unique evaluations:

Substitution: $\vdash (\forall x \,.\, \varphi) \wedge (\exists y \,.\, \varphi' = y) \to \varphi[\varphi'/x]$

Condition $\exists y \,.\, \varphi' = y$ holds when φ' is a term built with symbols σ obeying the functional axioms $\exists y \,.\, \sigma(x_1, \ldots, x_n) = y$ discussed in Section 4.4. So the constrained substitution axiom is still more general than the original substitution axiom in FOL, since it can also apply when φ' is not built only from functional symbols but can be proved to have unique evaluation. It is interesting to note that a similar modification of Substitution was needed in the context of partial FOL [7], where the interpretations of functional symbols are partial functions, so terms may be undefined; axiom PFOL5 in [7] requires φ' to be *defined* in the Substitution rule, and several rules for proving definedness are provided. Note that our condition $\exists y \,.\, \varphi' = y$ is equivalent to definedness in the special case of PFOL, and that, thanks to the definability of equality in matching logic, we do not need special machinery for proving definedness.

<u>FOL axioms and rules:</u>

1. $\vdash$ propositional tautologies
2. Modus ponens: $\vdash \varphi_1$ and $\vdash \varphi_1 \to \varphi_2$ imply $\vdash \varphi_2$
3. $\vdash (\forall x \,.\, \varphi_1 \to \varphi_2) \to (\varphi_1 \to \forall x \,.\, \varphi_2)$ when $x \notin FV(\varphi_1)$
4. Universal generalization: $\vdash \varphi$ implies $\vdash \forall x \,.\, \varphi$
5. Substitution: $\vdash (\forall x \,.\, \varphi) \wedge (\exists y \,.\, \varphi' = y) \to \varphi[\varphi'/x]$
6. Equality introduction: $\vdash \varphi = \varphi$
7. Equality elimination: $\vdash \varphi_1 = \varphi_2 \wedge \varphi[\varphi_1/x] \to \varphi[\varphi_2/x]$

<u>Membership axioms and rules:</u>

8. $\vdash \forall x \,.\, x \in \varphi$ iff $\vdash \varphi$
9. $\vdash x \in y = (x = y)$ when $x, y \in \mathit{Var}$
10. $\vdash x \in \neg\varphi = \neg(x \in \varphi)$
11. $\vdash x \in \varphi_1 \wedge \varphi_2 = (x \in \varphi_1) \wedge (x \in \varphi_2)$
12. $\vdash (x \in \exists y.\varphi) = \exists y.(x \in \varphi)$, with x and y distinct
13. $\vdash x \in \sigma(\varphi_1,..,\varphi_{i-1},\varphi_i,\varphi_{i+1},..,\varphi_n) = \exists y.(y \in \varphi_i \wedge x \in \sigma(\varphi_1,..,\varphi_{i-1},y,\varphi_{i+1},..,\varphi_n))$

Figure 2 Sound and complete proof system of matching logic.

Our approach to obtain a sound and complete proof system for matching logic is to build upon its reduction to predicate logic in Section 3. Specifically, to use Proposition 7 and the complete proof system of predicate logic. Given a matching logic signature (S, Σ), let (S, Π_Σ) be the predicate logic signature obtained like in Section 3. In addition to the PL translation there, we also define a backwards translation ML of (S, Π_Σ)-formulae into (S, Σ)-patterns:

$$\begin{array}{rcl}
ML(x = r) & = & x = r \\
ML(\pi_\sigma(r_1, \ldots, r_n, r)) & = & r \in \sigma(r_1, \ldots, r_n) \\
ML(\neg\psi) & = & \neg ML(\psi) \\
ML(\psi_1 \wedge \psi_2) & = & ML(\psi_1) \wedge ML(\psi_2) \\
ML(\exists x \,.\, \psi) & = & \exists x \,.\, ML(\psi) \\
ML(\{\psi_1, \ldots, \psi_n\}) & = & \{ML(\psi_1), \ldots, ML(\psi_n)\}
\end{array}$$

Recall from Section 4.3 that we assume equality and membership in all specifications.

Figure 2 shows our sound and complete proof system for matching logic reasoning, which was specifically crafted to include the proof system of first-order logic. Indeed, the first group of axiom and rule schemas include all the axioms and proof rules of FOL with equality as instances (the rules Substitution, Equation introduction and Equation elimination allow more general patterns instead of terms). The second group of proof rules, for reasoning about membership, is introduced for technical reasons, namely for the proof of Theorem 16:

▶ **Theorem 16.** *The proof system in Figure 2 is sound and complete:* $F \models \varphi$ *iff* $F \vdash \varphi$*.*

6 Additional Related Work

Matching logic builds upon intuitions from and relates to at least four important logical frameworks: (1) *Relation algebra (RA)* (see, e.g., [21]), noticing that our interpretations of symbols as functions to powersets are equivalent to relations; although our interpretation of symbols captures better the intended meaning of pattern and matching, and our proof system is quite different from that of RA, like with FOL we expect a tight relationship between matching logic and RA, which is left as future work; (2) *Partial FOL* (see, e.g., [7] for a recent work and a survey), noticing that our interpretations of symbols into powersets

are more general than partial functions (Section 4.3 shows how we defined definedness); and (3) *Separation logics (SL)* (see, e.g.,[11]), which we briefly discussed in Section 4.9 but refer the reader to [15] for more details; and (4) Precursors of matching logic in [17, 20, 16], which proposed the pattern idea by extending FOL with particular "configuration" terms:

$$
\begin{array}{rcl}
t_s & ::= & x \in Var_s \quad | \quad \sigma(t_1, \ldots, t_n) \ \text{ with } \sigma \in \Sigma_{s_1 \ldots s_n, s} \\
\varphi & ::= & \pi(x_1, \ldots, x_n) \ \text{ with } \pi \in \Pi_{s_1 \ldots s_n} \quad | \quad \neg\varphi \quad | \quad \varphi \wedge \varphi \quad | \quad \exists x . \varphi \\
& | & t \in T_{\Sigma, Cfg}(X)
\end{array}
$$

where $T_{\Sigma, Cfg}(X)$ is the set of terms of a special sort *Cfg* (from "configurations") over variables in set X. To avoid naming conflicts, we propose to call the variant above *topmost matching logic* from here on. Topmost matching logic can trivially be desugared into FOL with equality by regarding a particular pattern predicate $t \in T_{\Sigma, Cfg}(X)$ as syntactic sugar for "(current state/configuration is) equal to t". One major limitation of topmost matching logic, which motivated the generalization in this paper, is that its restriction to patterns of sort *Cfg* prevented us to define local patterns (e.g., the heap list pattern) and perform local reasoning.

The idea of regarding arbitrary terms as patterns is reminiscent to *pattern calculus* [10], although note that matching logic's patterns are intended to express and reason about static properties of data-structures or program configurations, while pattern calculi are aimed at generally and compactly expressing computations and dynamic behaviors of systems. So far we used rewriting to define dynamic language semantics; it would be interesting to explore the combination of pattern calculus and matching logic for language semantics and reasoning.

7 Conclusion and Future Work

Matching logic is a sound and complete FOL variant that makes no distinction between function and predicate symbols. Its formulae, called patterns, mix symbols, logical connectives and quantifiers, and evaluate in models to sets of values, those that "match" them, instead of just one value as terms do or a truth value as predicates do in FOL. Equality can be defined and several important variants of FOL fall as special fragments. Separation logic can be framed as a matching logic theory within the particular model of partial finite-domain maps, and heap patterns can be elegantly specified using equations. Matching logic allows spatial specification and reasoning anywhere in a program configuration, and for any language, not only in the heap or other particular and fixed semantic components.

We made no efforts to minimize the number of rules in our proof system, because our main objective here was to include the proof system for FOL with equality. It is likely that a minimal proof system working directly with the core symbols $\lceil _ \rceil_{s_1}^{s_2} \in \Sigma_{s_1, s_2}$ for all sorts $s_1, s_2 \in S$ can be obtained such that the equality and membership axioms and rules in Figure 2 can be proved as lemmas. Likewise, we refrained from discussing any computationally effective fragments of matching logic, although we are implementing them in $\mathbb{K}$. Finally, complexity results in the style of [1, 3, 9] for separation logic can likely also be obtained for fragments of matching logic.

Acknowledgments. I wish to express my deepest thanks to the $\mathbb{K}$ team[1], who share the belief that programming languages should have only one semantics, which should be executable, and formal analysis tools, including fully fledged deductive program verifiers, should be obtained from such semantics at little or no extra cost. I would like to also warmly thank

[1] `http://kframework.org`

the following colleagues and friends for their comments and criticisms on previous drafts of this paper: Nikolaj Bjorner, Claudia-Elena Chiriţă, Maribel Fernández, Dorel Lucanu, Brandon Moore, Cosmin Rădoi, Andrei Ştefănescu.

References

1 Timos Antonopoulos, Nikos Gorogiannis, Christoph Haase, Max I. Kanovich, and Joël Ouaknine. Foundations for decision problems in separation logic with general inductive predicates. In *FOSSACS'14*, LNCS 8412, pages 411–425, 2014.
2 Denis Bogdănaş and Grigore Roşu. K-Java: A Complete Semantics of Java. In *POPL'15*, pages 445–456. ACM, 2015.
3 James Brotherston, Carsten Fuhs, Nikos Gorogiannis, and Juan Navarro Pérez. A decision procedure for satisfiability in separation logic with inductive predicates. Technical Report RN/13/15, University College London, 2013.
4 Andrei Ştefănescu, Ştefan Ciobâcă, Radu Mereuţă, Brandon M. Moore, Traian Florin Şerbănuţă, and Grigore Roşu. All-path reachability logic. In *RTA-TLCA'14*, volume 8560 of *LNCS*, pages 425–440. Springer, 2014.
5 Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *TACAS*, pages 337–340, 2008. LNCS 4963.
6 Chucky Ellison and Grigore Rosu. An executable formal semantics of C with applications. In *POPL*, pages 533–544, 2012.
7 William M. Farmer and Joshua D. Guttman. A set theory with support for partial functions. *Studia Logica*, 66(1):59–78, 2000.
8 Chris Hathhorn, Chucky Ellison, and Grigore Roşu. Defining the undefinedness of C. In *PLDI'15*. ACM, 2015.
9 Radu Iosif, Adam Rogalewicz, and Jirí Simácek. The tree width of separation logic with recursive definitions. In *CADE'13*, LNCS 7898, 2013.
10 C. Barry Jay. The pattern calculus. *ACM Trans. Program. Lang. Syst.*, 26(6):911–937, November 2004.
11 Peter O'Hearn, John Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *CSL*, pages 1–19. LNCS 2142, 2001.
12 Peter W. O'Hearn and David J. Pym. The logic of bunched implications. *Bulletin of Symb. Logic*, 5(2):215–244, 1999.
13 Daejun Park, Andrei Ştefănescu, and Grigore Roşu. KJS: A complete formal semantics of JavaScript. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*. ACM, 2015.
14 John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74, 2002.
15 Grigore Roşu. Matching logic: A logic for structural reasoning. Technical Report http://hdl.handle.net/2142/47004, University of Illinois, Jan 2014.
16 Grigore Roşu, Andrei Ştefănescu, Ştefan Ciobâcă, and Brandon M. Moore. One-path reachability logic. In *LICS'13*. IEEE, 2013.
17 Grigore Rosu, Chucky Ellison, and Wolfram Schulte. Matching logic: An alternative to Hoare/Floyd logic. In *AMAST*, volume 6486 of *LNCS*, pages 142–162, 2010.
18 Grigore Roşu and Traian Florin Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.
19 Grigore Rosu and Traian Florin Serbanuta. K overview and simple case study. In *Proc. of International K Workshop (K'11)*, volume 304 of *ENTCS*, pages 3–56. Elsevier, 2014.
20 Grigore Rosu and Andrei Stefanescu. Checking reachability using matching logic. In *OOPSLA'12*, pages 555–574. ACM, 2012.
21 A. Tarski and S.R. Givant. *A Formalization of Set Theory Without Variables.* Number 41 in Colloquium Publications. AMS, 1987.

Executable Formal Models in Rewriting Logic

Carolyn Talcott

SRI International
Menlo Park, California, USA
clt@csl.sri.com

Abstract

Formal executable models provide a means to gain insights into the behavior of complex distributed systems. Ideas can be prototyped and assurance gained by carrying out analyses at different levels of fidelity: searching for desirable or undesirable behaviors, determining effects of perturbing the system, and eventually investing effort to carry out formal proofs of key properties. This modeling approach applies to a wide range of systems, including a variety of protocols and networked cyber-physical systems. It is also emerging as an important tool in understanding many different aspects of biological systems.

Rewriting logic (RWL) is a formalism that is well-suited to developing and working with formal executable models. In RWL term rewriting is used to represent both structure (equational properties and functions) and transformation / behavior. Logics and inference systems can be naturally represented in RWL, as can the structure and behavior of distributed systems both engineered and natural.

Maude is a high performance realization of Rewriting Logic. Maude specifications are naturally executable and the Maude environment provides a variety analysis tools to reason about properties of models. These include reachability analysis, symbolic execution (narrowing), and model-checking. In addition, Maude is reflective. This provides a powerful mechanism for extension.

The talk will present a sampling of executable specifications using Maude and its extensions. The examples will illustrate a range of modeling problems and analysis approaches, including

- Analysis of engineered systems
  - finding problems and fixing the system,
  - optimizing performance.
- Analysis of natural systems
  - finding problems and fixing the model,
  - using the model to predict consequences of perturbations.

To be self-contained, the talk will begin with a little introduction to RWL and Maude.

1998 ACM Subject Classification D.2.4 Formal methods, F.3.1 Specification techniques, mechanical verification

Keywords and phrases Executable model, formal analysis, rewriting logic

Digital Object Identifier 10.4230/LIPIcs.RTA.2015.22

Category Invited Talk

© Carolyn Talcott;
licensed under Creative Commons License CC-BY
26th International Conference on Rewriting Techniques and Applications (RTA'15).
Editor: Maribel Fernández; pp. 22–22
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Certification of Complexity Proofs using CeTA

Martin Avanzini[1], Christian Sternagel[2], and René Thiemann[2]

1 **Università di Bologna, Italy, and INRIA, Sophia Antipolis, France**
martin.avanzini@uibk.ac.at

2 **University of Innsbruck, Austria**
{christian.sternagel|rene.thiemann}@uibk.ac.at

Abstract

Nowadays certification is widely employed by automated termination tools for term rewriting, where certifiers support most available techniques. In complexity analysis, the situation is quite different. Although tools support certification in principle, current certifiers implement only the most basic technique, namely, suitably tamed versions of reduction orders. As a consequence, only a small fraction of the proofs generated by state-of-the-art complexity tools can be certified. To improve upon this situation, we formalized a framework for the certification of modular complexity proofs and incorporated it into CeTA. We report on this extension and present the newly supported techniques (match-bounds, weak dependency pairs, dependency tuples, usable rules, and usable replacement maps), resulting in a significant increase in the number of certifiable complexity proofs. During our work we detected conflicts in theoretical results as well as bugs in existing complexity tools.

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

Keywords and phrases complexity analysis, certification, match-bounds, weak dependency pairs, dependency tuples, usable rules, usable replacement maps

Digital Object Identifier 10.4230/LIPIcs.RTA.2015.23

1 Introduction

The last decade saw a wealth of techniques for automated termination tools, closely followed by techniques and tools for automated complexity analysis in recent years. In individual proofs, such tools often apply several techniques in combination, making human inspection ever more unrealistic, due to their sheer size. Moreover, the increasing power of automated tools comes at the cost of amplified complexity, reducing reliability; hence the interest in automatic certification of termination and complexity proofs.

Whereas our certifier CeTA [18] is already able to certify most proofs generated by current termination tools for term rewrite systems (TRSs), initial support for complexity proofs was added only recently [17]. In this paper we present a significant extension of CeTA towards the certification of complexity proofs. To this end, we formalized several techniques for complexity analysis within the proof assistant Isabelle/HOL [14] as part of our formal library IsaFoR.[1] On top of these general results, we augmented CeTA by corresponding functions, that check whether specific applications of techniques, encountered inside automatically generated complexity proofs, are correct.

As a result, the power of CeTA for certifying complexity proofs has almost tripled in comparison to last year [17], and more than 75 % of all tool-generated proofs can be certified.

[1] http://cl-informatik.uibk.ac.at/software/ceta

© Martin Avanzini, Christian Sternagel, and René Thiemann;
licensed under Creative Commons License CC-BY
26th International Conference on Rewriting Techniques and Applications (RTA'15).
Editor: Maribel Fernández; pp. 23–39
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Moreover, via certification we detected and fixed several bugs in current complexity tools, some of which had remained undetected for more than five years.

Contribution and Overview. After giving some preliminaries in Section 2, we present our main contributions. In Section 3, we explain our formalization of a framework which admits us to certify composite complexity proofs. At this point, we also report on conflicting notions of basic complexity definitions in the literature. In Section 4 we describe our formalization of the match-bounds technique. Here, the transition from termination to complexity results is surprisingly easy. Concerning the integration of match-bounds for relative rewriting, we provide a new example showing that two existing variants are incomparable. In Section 5, we discuss our formalization of two dependency pair related techniques: weak dependency pairs and dependency tuples. We choose to conduct the respective proofs using two slightly different approaches – one focusing on contexts, the other on sets of positions – and comment on our findings. In Section 6, we slightly generalize one variant of usable rules, and also support another variant for innermost rewriting, for which we reuse existing proofs from termination analysis. Furthermore, we present a new theorem combining usable rules, usable replacement maps, and argument filters. Finally, in Section 7, we discuss conducted experiments and conclude.

All of the proofs that are presented (or omitted) in the following have been formalized and made available as part of IsaFoR. Direct links to the formalization are available from the following website, that also contains all details on our experiments.

`http://cl-informatik.uibk.ac.at/software/ceta/experiments/complexity/`

2 Preliminaries

We assume basic familiarity with term rewriting (as for example obtained by reading the textbook by Baader and Nipkow [3]) but shortly recall some basic notions and notations that are used later on.

By $\mathcal{T}(\mathcal{F},\mathcal{V})$ we denote the set of *(first-order) terms* w.r.t. a *signature* $\mathcal{F}$ and a set of *variables* $\mathcal{V}$, and by $\mathcal{T}(\mathcal{F})$ the set of ground terms. We write $\mathsf{root}(t)$ for the root symbol of a non-variable term t. The *size* $|t|$ of a term t is defined by $|x| = 1$ for $t = x \in \mathcal{V}$, and $|f(t_1,\dots,t_n)| = 1 + \sum_{i=1}^{n} |t_i|$, otherwise. A *(multihole) context* is a term that may contain an arbitrary number of *holes*, represented by the special symbol $\square$. Replacing the holes in a given multihole context C by terms $t_1,\dots,t_n$ is written $C[t_1,\dots,t_n]$. (At this point it might be worth mentioning that in our formalization we have to make sure that the number of holes in C corresponds to the number of terms n. For simplicity's sake we do not make this explicit in the remainder). Whenever $s = C[t]$ for some context C ($\neq \square$), then t is called a *(proper) subterm* of s. We write $t\sigma$ for the application of a substitution σ to a term t.

A *TRS* $\mathcal{R}$ is a set of *(rewrite) rules*, where a rule $\ell \to r$ is a pair of terms such that $\ell \notin \mathcal{V}$ and only variables already occurring in ℓ are allowed in r. The *defined symbols* of $\mathcal{R}$, written $\mathcal{D}(\mathcal{R})$, are those that are roots of left-hand sides of its rules. We use $\mathcal{F}\mathsf{un}(\cdot)$ to denote the set of function symbols occurring in a given term, context, or TRS. A TRS is *left-linear* (*non-duplicating*) if and only if for all rules $\ell \to r \in \mathcal{R}$, no variable occurs more than once in ℓ (more often in r than in ℓ).

The standard way of uniquely referring to subterms is via *positions*, denoted by lists of natural numbers. The subterm of a term t at position p is written $t|_p$. We use $\leq$ for the usual partial order on positions, and denote by $p \mathrel{||} q$ that positions p and q are *parallel*, i.e., incomparable by $\leq$. The strict part of $\leq$ is denoted by $<$.

There is a *rewrite step* from term s to term t w.r.t. the rewrite relation induced by TRS $\mathcal{R}$, denoted $s \to_{\mathcal{R}} t$, whenever there are C, σ, and $\ell \to r \in \mathcal{R}$ such that $s = C[\ell\sigma]$ and $t = C[r\sigma]$. Equivalently, we say that s *rewrites* to t at position p, where p is the unique position of $\Box$ in C. The subterm $\ell\sigma$ above is called an *($\mathcal{R}$-)redex*. Terms not containing any $\mathcal{R}$-redexes are called $\mathcal{R}$-normal or *normal forms*, and we write $\mathsf{NF}(\mathcal{R})$ for the set of all $\mathcal{R}$-normal forms. We sometimes use the same notion not only for TRSs but also for sets of terms, since right-hand sides of rules are irrelevant for the existence of redexes anyway.

For termination analysis *Q-restricted rewriting* (named after the additional parameter, a set of terms, which is usually denoted $\mathcal{Q}$) was introduced in order to cover full rewriting and innermost rewriting (as well as variations that lie somewhere in between) under a single framework [9]. Here, a rewrite step $C[\ell\sigma] \xrightarrow{\mathcal{Q}}_{\mathcal{R}} C[r\sigma]$ is a standard rewrite step $C[\ell\sigma] \to_{\mathcal{R}} C[r\sigma]$ whose redex $\ell\sigma$ additionally satisfies the condition that all its proper subterms are $\mathcal{Q}$-normal, i.e., do not match any term in $\mathcal{Q}$ (in that way standard rewriting is Q-restricted rewriting with empty $\mathcal{Q}$ and for innermost rewriting we take the left-hand sides of $\mathcal{R}$ as $\mathcal{Q}$). This proves convenient also for complexity analysis and its notions of runtime complexity and innermost runtime complexity. Additionally, *relative rewriting* is important for complexity analysis, since it can be employed to obtain modular proofs. A *relative rewrite system* consists of two TRSs $\mathcal{S}$ and $\mathcal{W}$, and is denoted by $\mathcal{S}/\mathcal{W}$. The corresponding *relative rewrite relation*, written $\to_{\mathcal{S}/\mathcal{W}}$, is given by $\to_{\mathcal{W}}^{*} \cdot \to_{\mathcal{S}} \cdot \to_{\mathcal{W}}^{*}$. Combined this leads to *Q-restricted relative rewriting*, where $\xrightarrow{\mathcal{Q}}_{\mathcal{S}/\mathcal{W}}$ denotes the relation $\xrightarrow{\mathcal{Q}}{}_{\mathcal{W}}^{*} \cdot \xrightarrow{\mathcal{Q}}_{\mathcal{S}} \cdot \xrightarrow{\mathcal{Q}}{}_{\mathcal{W}}^{*}$. Note that we fix the same $\mathcal{Q}$ for "strict" and "weak" steps, which is sufficient for our purposes.[2]

Given a binary relation $\to$ and a set A, we define $\to(A) = \{b \mid \exists a \in A.\, a \to b\}$.

3 A Framework for Modular Complexity Proofs

In complexity analysis of TRSs we are usually interested in the maximal number of steps that are possible when starting from a given set of terms. To this end, the basic ingredient of our formalization is the *derivation bound* (defined in theory `Complexity`; see also [17]), where a function g constitutes a derivation bound of relation R w.r.t. *starting elements* from a family of sets S, written $\mathsf{db}_R^S(g)$, if and only if for every $n \in \mathbb{N}$ and $x \in S_n$, every sequence of R-steps starting at x is of length at most $g(n)$. The intuition is that S_n contains "objects" of size n. This, more or less, corresponds to the usual notion of complexity. To be more precise, Avanzini and Moser [2] define $\mathsf{cp}(n, T, R) = \mathsf{max}\{\mathsf{dh}(t, R) \mid \exists t \in T.\, |t| \leq n\}$, where dh denotes the *derivation height* of a term, and derivational complexity as well as runtime complexity are obtained by suitably instantiating T and R. However, as argued earlier [17], using the derivation bound g as argument avoids undefined situations that arise with the usual definition, e.g., taking the maximum of a potentially infinite set. Whenever $\mathsf{cp}(n, T, R)$ is defined, we have $\mathsf{db}_R^S(g)$ with $S_n = \{t \in T \mid |t| \leq n\}$ and $g(n) = \mathsf{cp}(n, T, R)$, as well as $h(n) \geq \mathsf{cp}(n, T, R)$ for all other derivation bounds h. That is, our bounds are not tight, but arbitrary upper bounds.

Depending on the set of starting elements, we obtain the usual notions of *derivational complexity* and *runtime complexity*, respectively. For the former we consider all terms of size n w.r.t. a given signature $\mathcal{F}$, whereas the latter is based on *basic terms* of size n. Given two sets of function symbols $\mathcal{D}$ (defined symbols) and $\mathcal{C}$ (constructors), and a set of variables $\mathcal{V}$, the set of basic terms $\mathsf{BT}(\mathcal{D}, \mathcal{C}, \mathcal{V})$ consists of those terms which are rooted by a symbol from

[2] A more general relation with separate $\mathcal{Q}$s for $\mathcal{S}$ and $\mathcal{W}$ would be imaginable. However, since tools do not support this variation, we stick to the simpler case.

$\mathcal{D}$ and where all arguments are terms of $\mathcal{T}(\mathcal{C},\mathcal{V})$. At this point we would like to mention that there are conflicting notions of basic terms: Hirokawa and Moser [10] and Noschinski et al. [15] use the above definition of basic terms. In contrast, Avanzini [1] additionally restricts basic terms to be ground, intending that constructor ground terms correspond to values, and thus, basic terms correspond to function application on input values. Since IsaFoR does not enforce basic terms to be ground, every (upper) derivation bound that is certified by CeTA also is valid w.r.t. Avanzini's notion of basic terms. However, there might be valid derivation bounds w.r.t. the ground semantics which cannot be certified in the non-ground setting:

▶ **Example 1.** Let $\mathcal{R} = \{\mathsf{f}(\mathsf{f}(x)) \to \mathsf{g}(x), \mathsf{g}(x) \to \mathsf{f}(\mathsf{f}(x)), \mathsf{f}(\mathsf{a}) \to \mathsf{a}\}$. Then there are only two basic ground terms, $\mathsf{f}(\mathsf{a})$ and $\mathsf{g}(\mathsf{a})$. Since the longest innermost derivation starting from these terms is of length 3, $\mathcal{R}$ has constant innermost runtime complexity w.r.t. ground basic terms. But there is an infinite innermost derivation starting from the non-ground basic term $\mathsf{g}(x)$.

We adopt the following notions from Avanzini and Moser [2]. A *(complexity) problem* $\mathcal{P} = \langle \mathcal{S}/\mathcal{W}, \mathcal{Q}, \mathcal{T}\rangle$ consists of two TRSs $\mathcal{S}$, $\mathcal{W}$, and two sets of terms $\mathcal{Q}$, $\mathcal{T}$. We asses the complexity of a problem $\mathcal{P}$ by a *(complexity) judgment* of the form $\vdash \mathcal{P}\colon g$, which is *valid* whenever g is a bound for $\xrightarrow{\mathcal{Q}}_{\mathcal{S}/\mathcal{W}}$-derivations starting from $\mathcal{T}$. For sets of functions G we define that $\vdash \mathcal{P}\colon G$ is valid whenever $\vdash \mathcal{P}\colon G$ is valid for some $g \in G$. Often, G is an asymptotic complexity class like $\mathcal{O}(n^3)$. A *(complexity) processor* turns a given judgment $\vdash \mathcal{P}\colon G$ into a list of judgments $\vdash \mathcal{P}_1\colon G_1, \ldots, \vdash \mathcal{P}_n\colon G_n$. It is *sound* whenever the validity of each of $\vdash \mathcal{P}_i\colon G_i$ also implies validity of $\vdash \mathcal{P}\colon G$. A processor is *terminal* if the returned list of judgments is empty.

The problem $\mathcal{P}$ is called a *runtime complexity problem* if $\mathcal{T} = \mathsf{BT}(\mathcal{D},\mathcal{C},\mathcal{V})$, with $\mathcal{S}$ and $\mathcal{W}$ not defining any constructor $\mathcal{C}$, i.e., $\mathcal{D}(\mathcal{S}\cup\mathcal{W})\cap\mathcal{C} = \varnothing$. The problem $\mathcal{P}$ is called an *innermost problem* if $\mathsf{NF}(\mathcal{Q}) \subseteq \mathsf{NF}(\mathcal{S}\cup\mathcal{W})$. In this case, $\xrightarrow{\mathcal{Q}}_{\mathcal{S}/\mathcal{W}}$ is a composition of innermost rewrite steps with respect to $\mathcal{S}\cup\mathcal{W}$.

As a first example processor, we formulate a theorem by Zankl and Korp [20, Thm. 3.6] within our framework which admits modular complexity proofs.

▶ **Theorem 2** (Split Processor). *Let $\mathcal{P} = \langle \mathcal{S}_1 \cup \mathcal{S}_2/\mathcal{W}, \mathcal{Q}, \mathcal{T}\rangle$ be a complexity problem and define $\mathcal{P}_1 = \langle \mathcal{S}_1/\mathcal{S}_2 \cup \mathcal{W}, \mathcal{Q}, \mathcal{T}\rangle$ and $\mathcal{P}_2 = \langle \mathcal{S}_2/\mathcal{S}_1 \cup \mathcal{W}, \mathcal{Q}, \mathcal{T}\rangle$. The* split processor *translates the judgment $\vdash \mathcal{P}\colon \mathcal{O}(g)$ into the judgments $\vdash \mathcal{P}_1\colon \mathcal{O}(g)$ and $\vdash \mathcal{P}_2\colon \mathcal{O}(g)$.*

The split processor is sound.

▶ **Example 3.** The split processor is used whenever rules should be shifted from the strict into the weak component, e.g., when applying match-bounds for relative rewriting or when using orderings. As an example, consider a TRS with rules numbered from 1 to 5 where cubic complexity has been proven. In the proof, first rules 2, 3, and 4 have been oriented strictly and rules 1 and 5 are oriented weakly by some ordering o_1 with quadratic complexity. Afterwards rule 1 could be moved into the weak component by match-bounds, and finally rule 5 is oriented strictly by some ordering o_2 with cubic complexity, where the remaining rules 1 to 4 are oriented weakly. This proof is restructured via split as follows. First, the initial complexity judgment $\vdash \langle\{1,2,3,4,5\}/\varnothing, \mathcal{Q}, \mathcal{T}\rangle\colon \mathcal{O}(n^3)$ is splitted into $\vdash \langle\{2,3,4\}/\{1,5\}, \mathcal{Q}, \mathcal{T}\rangle\colon \mathcal{O}(n^3)$ and $\vdash \langle\{1,5\}/\{2,3,4\}, \mathcal{Q}, \mathcal{T}\rangle\colon \mathcal{O}(n^3)$ by the split processor where the former judgment is validated via o_1. The latter complexity problem is split again into $\vdash \langle\{1\}/\{2,3,4,5\}, \mathcal{Q}, \mathcal{T}\rangle\colon \mathcal{O}(n^3)$ and $\vdash \langle\{5\}/\{1,2,3,4\}, \mathcal{Q}, \mathcal{T}\rangle\colon \mathcal{O}(n^3)$ where the former judgment is validated via match-bounds, and the latter one via o_2.

The example demonstrates that via splitting it suffices to restrict match-bounds and orderings to terminal complexity processors. This is the reason why we present both techniques as terminal processors in Sections 4 and 6.

4 Match-Bounds

The match-bounds technique was introduced as termination method by Geser et al. [7]. We shortly recapitulate the main underlying ideas, before explaining our formalization (theory `Matchbounds`) and the necessary adaptations to use it for complexity analysis [19, 20].

Let $\mathcal{F}$ be a signature containing at least one constant. For match-bounds, $\mathcal{F}$ is expanded such that symbols are labeled by natural numbers, i.e., $\mathcal{F}' = \mathcal{F} \times \mathbb{N}$. Moreover, we define auxiliary functions $\mathsf{base} : \mathcal{T}(\mathcal{F}', \mathcal{V}) \to \mathcal{T}(\mathcal{F}, \mathcal{V})$, $\mathsf{lift}_d : \mathcal{T}(\mathcal{F}, \mathcal{V}) \to \mathcal{T}(\mathcal{F}', \mathcal{V})$, and $\mathsf{lab} : \mathcal{T}(\mathcal{F}', \mathcal{V}) \to 2^{\mathbb{N}}$, where base removes all labels of a term, lift_d labels all symbols of a term by d, and lab returns the set of labels of a term. For a non-duplicating TRS $\mathcal{R}$ over signature $\mathcal{F}$ we construct the TRS $\mathcal{R}' = \mathsf{match}(\mathcal{R})$ over $\mathcal{F}'$.

$$\mathsf{match}(\mathcal{R}) = \{\ell' \to \mathsf{lift}_d(r) \mid \ell \to r \in \mathcal{R}, \mathsf{base}(\ell') = \ell, d = 1 + \min(\mathsf{lab}(\ell'))\} \tag{1}$$

Then for left-linear $\mathcal{R}$, every rewrite step $s \to_{\mathcal{R}} t$ can be simulated by a step $s' \to_{\mathcal{R}'} t'$ with $\mathsf{base}(t') = t$, provided $\mathsf{base}(s') = s$. Hence, every (possibly) infinite derivation (2) gives rise to a step-wise simulation (3) provided $\mathsf{base}(t'_0) = t_0$, which is ensured by choosing $t'_0 = \mathsf{lift}_0(t_0)$.

$$\begin{array}{ccccccc} t_0 & \to_{\mathcal{R}} & t_1 & \to_{\mathcal{R}} & t_2 & \to_{\mathcal{R}} & \cdots \end{array} \tag{2}$$
$$\begin{array}{ccccccc} t'_0 & \to_{\mathcal{R}'} & t'_1 & \to_{\mathcal{R}'} & t'_2 & \to_{\mathcal{R}'} & \cdots \end{array} \tag{3}$$
$$\mathsf{mul}(t'_0) >_{\mathsf{ms}} \mathsf{mul}(t'_1) >_{\mathsf{ms}} \mathsf{mul}(t'_2) >_{\mathsf{ms}} \cdots \tag{4}$$

As the next step, a function mul maps every term t'_i to the multiset of negated labels, where by construction of $\mathsf{match}(\mathcal{R})$ every step with $\mathcal{R}'$ results in a strict decrease w.r.t. the standard multiset-order $>_{\mathsf{ms}}$ on the integers, and thus we can construct (4) from (3).

Let us assume that the initial term t_0 is ground. Then $t'_0 = \mathsf{lift}_0(t_0)$ implies that the initial term in (3) is always a member of $\mathcal{T}(\mathcal{F} \times \{0\})$. We now try to find some bound $b \in \mathbb{N}$, such that $\to^*_{\mathcal{R}'}(\mathcal{T}(\mathcal{F} \times \{0\})) \subseteq \mathcal{T}(\mathcal{F} \times \{0, \ldots, b\})$. If this succeeds, then the labels in derivation (3) are bounded by b, and hence, all numbers in (4) are in the range $-b, \ldots, 0$. Since $>$ is well-founded on this domain, so is $>_{\mathsf{ms}}$. Hence, (4) cannot be infinite, and therefore, also (3), and (2) cannot be infinite, proving termination of $\mathcal{R}$ on ground terms. Moreover, since $\mathcal{F}$ contains at least one constant, termination on ground terms implies termination on all terms.

In total, we formalized the following theorem for termination analysis.

▶ **Theorem 4.** *If $\mathcal{R}$ is a non-duplicating, left-linear TRS over signature $\mathcal{F}$, and there is some language L satisfying $\to^*_{\mathcal{R}'}(\mathsf{lift}_0(\mathcal{T}(\mathcal{F}))) \subseteq L \subseteq \mathcal{T}(\mathcal{F} \times \{0, \ldots, b\})$, then $\mathcal{R}$ is terminating.*

Here, non-duplication is essential in the step from (3) to (4), and left-linearity is required to ensure the one-step simulation property (Lemma 5). The language L usually comes in the form of a finite automaton which has been constructed via tree automata completion [6].

▶ **Lemma 5.** *If $\mathcal{R}$ is left-linear, $s \to_{\mathcal{R}} t$, and $\mathsf{base}(s') = s$, then there exists a term t' such that $s' \to_{\mathcal{R}'} t'$ and $\mathsf{base}(t') = t$.*

The lemma is straightforward to prove on paper, and also its formalization posed no difficulties. Actually, it is no longer present, since IsaFoR now includes a full proof of a more general result by Korp and Middeldorp [12, Lemma 12], applying also to non-left-linear TRSs. It is the essential ingredient to obtain (3) from (2).

Concerning the step from (3) to (4), in the formalization we already require the bound b at this point. This allows us to include an index shift in mul, so that each label i is mapped onto $b - i \in \mathbb{N}$. Then the parameter $>$ of $>_{\mathsf{ms}}$ in (4) is the standard order on natural numbers.

In order to certify match-bounds proofs (which are required to contain L in the form of an automaton), CeTA must be able to check left-linearity and non-duplication, as well as that

the given automaton indeed accepts all terms in $\to^*_{\mathcal{R}'}(\mathcal{T}(\mathcal{F} \times \{0\}))$. For the latter, we make use of earlier work by Felgenhauer and Thiemann [5], and for the former, we rounded off Isabelle/HOL's existing theory on multisets by algorithms for comparing multisets (since a rule is non-duplicating if and only if the multiset of variables of its right-hand side is a subset of the multiset of variables of its left-hand side).

In case $\mathcal{F}$ does not contain a constant, e.g., in case of string rewrite systems, CeTA does an automatic preprocessing step, which invents a fresh constant, includes it into the signature, and adjusts the automaton accordingly.

In the remainder of this section, we adapt Theorem 4 and the corresponding formalization towards complexity analysis, following Zankl and Korp [19, 20].

The first step is to integrate complexity bounds into (2), (3), and (4), starting from (4). Given a term of size n, the initial value $\mathsf{mul}(t'_0)$ is the multiset containing n times the value b. However, this does not immediately give a nice bound on the length of (4), since $>_{\mathsf{ms}}$ does not impose any bound on the length of derivations w.r.t. the initial multiset: $\{\{1\}\} >_{\mathsf{ms}} \{\{0,\dots,0\}\} >_{\mathsf{ms}} \cdots >_{\mathsf{ms}} \{\{0\}\} >_{\mathsf{ms}} \varnothing$. Thus, we replace $>_{\mathsf{ms}}$ by $>_{\mathsf{ms},k}$ in (4), where $>_{\mathsf{ms},k}$ is a bounded version of $>_{\mathsf{ms}}$ such that at most k elements may be added in each comparison: $X >_{\mathsf{ms},k} Y$ if and only if $X = U \cup V$, $Y = U \cup W$, $V >_{\mathsf{ms}} W$, and $|W| \leq k$.

Of course, we have to substitute $>_{\mathsf{ms},k}$ (with suitable k) for $>_{\mathsf{ms}}$ in all previous proofs. Doing so within the formalization was an easy task: take $k \geq 1$ as the maximum size of right-hand sides of $\mathcal{R}$. After this adaptation, it is shown that the length of $>_{\mathsf{ms},k}$-sequences is linearly bounded, using a result by Dershowitz and Manna [4, page 191]. To be more precise, we formalized that $X >^n_{\mathsf{ms},k} Y$ implies $n \leq \sum_{x \in X}(k+1)^x$, leading to the linear bound: Recall, that $\mathsf{mul}(t'_0) = \{b,\dots,b\}$ where the number of b's is $|t_0|$. Hence, sequence (4) can be of length at most $\sum_{x \in \mathsf{mul}(t'_0)}(k+1)^x = \sum_{1,\dots,|t_0|}(k+1)^b = (k+1)^b \cdot |t_0|$. As immediate consequence we conclude that also (3) and (2) are linearly bounded.

In total, we get the following result which is used in CeTA to check complexity proofs via match-bounds, where $\mathcal{T}_{gnd}$ is the set of all ground terms in $\mathcal{T}$. The restriction to ground terms is possible at this point (in contrast to Example 1) as $\mathcal{Q}$ is ignored in the analysis.

► **Theorem 6.** *Let $\mathcal{P} = \langle \mathcal{R}/\varnothing, \mathcal{Q}, \mathcal{T}\rangle$ be a complexity problem. If $\mathcal{R}$ is a non-duplicating and left-linear TRS over signature $\mathcal{F}$, and there is some language L satisfying $\to^*_{\mathcal{R}'}(\mathsf{lift}_0(\mathcal{T}_{gnd})) \subseteq L \subseteq \mathcal{T}(\mathcal{F} \times \{0,\dots,b\})$, then $\vdash \mathcal{P}\colon \mathcal{O}(n)$.*

The next step is to integrate relative rewriting. The main idea to handle weak rules is to use a modified version of match, which only has to ensure a decrease w.r.t. the weak multiset order $\geq_{\mathsf{ms},k}$. To this end, Zankl and Korp [19] define match-rt as in (1) except that the value of d in (1) is sometimes reduced. If $|\ell| \geq |r|$ and all labels in ℓ' are identical, then d is $\min(\mathsf{lab}(\ell'))$ instead of $1 + \min(\mathsf{lab}(\ell'))$. Hence, for some cases it is not required to increase the labels at all, and thus, it is more likely that a bound on the labels can be obtained. In order to integrate match-rt into IsaFoR we could mostly reuse or slightly generalize the existing proofs.

Zankl and Korp give another optimization of match-rt, integrating the bound b: $\mathsf{match\text{-}rt}_b$ is defined in the same way as match-rt, except that $\mathsf{lift}_d(r)$ is replaced by $\mathsf{lift}_{\min(b,d)}(r)$, which results in even smaller labels than match-rt, but which is restricted to non-collapsing strict rules. In total, we have formalized the following theorem.

► **Theorem 7.** *Let $\mathcal{P} = \langle \mathcal{S}/\mathcal{W}, \mathcal{Q}, \mathcal{T}\rangle$ be a complexity problem. Let $\mathcal{S} \cup \mathcal{W}$ be a non-duplicating and left-linear TRS over signature $\mathcal{F}$. Assume that $\mathcal{R}' = \mathsf{match}(\mathcal{S}) \cup \mathsf{match\text{-}rt}(\mathcal{W})$, or both $\mathcal{R}' = \mathsf{match}(\mathcal{S}) \cup \mathsf{match\text{-}rt}_b(\mathcal{W})$ and $\mathcal{S}$ is non-collapsing. If there is some language L satisfying $\to^*_{\mathcal{R}'}(\mathsf{lift}_0(\mathcal{T}_{gnd})) \subseteq L \subseteq \mathcal{T}(\mathcal{F} \times \{0,\dots,b\})$, then $\vdash \mathcal{P}\colon \mathcal{O}(n)$.*

Note that $\mathcal{Q}$ is completely ignored in Theorem 7, since the whole analysis does not take the strategy into account. In fact, the theorem was first proven for $\mathcal{Q} = \varnothing$, while the above statement including $\mathcal{Q}$ follows from $\xrightarrow{\mathcal{Q}}_{\mathcal{S}/\mathcal{W}} \subseteq \rightarrow_{\mathcal{S}/\mathcal{W}}$. The sole reason for this naive integration of $\mathcal{Q}$ was to support match-bounds on innermost problems in the first place. An alternative might be a dedicated processor that transforms $\langle \mathcal{S}/\mathcal{W}, \mathcal{Q}, \mathcal{T} \rangle$ into $\langle \mathcal{S}/\mathcal{W}, \varnothing, \mathcal{T} \rangle$.

When integrating $\mathsf{match\text{-}rt}_b$ in the formalization, we encountered two problems. First, we wanted to get rid of the choice in Theorem 7 and always use the better $\mathsf{match\text{-}rt}_b$ variant. The reason for this aim was that – while the non-collapsing condition on $\mathcal{S}$ appears inside their proofs – Zankl and Korp [20] did not state that its absence violates the main theorem. This is now shown by a counterexample.

▶ **Example 8** (Non-collapsing condition required)**.** Let $\mathcal{S} = \{\mathsf{f}(x) \to x\}$ and $\mathcal{W} = \{\mathsf{a} \to \mathsf{f}(\mathsf{a})\}$. For $\mathcal{R}' = \mathsf{match}(\mathcal{S}) \cup \mathsf{match\text{-}rt}_0(\mathcal{W}) = \{\mathsf{f}_i(x) \to x, \mathsf{a}_i \to \mathsf{f}_0(\mathsf{a}_0)\}$, the language $\rightarrow^*_{\mathcal{R}'}(\mathsf{lift}_0(\mathcal{T}(\mathcal{F})))$ is exactly $\mathcal{T}(\mathcal{F} \times \{0\})$. Without the non-collapsing condition within Theorem 7 one would be able to conclude linear derivational complexity of $\mathcal{S}/\mathcal{W}$, a contradiction.

Hence, the choices in Theorem 7 are really incomparable, and for certification it would be best to include both. Which brings us to the second problem: we did not want to copy and paste the existing proof for match-rt, and then incorporate all the tiny modifications that are required for $\mathsf{match\text{-}rt}_b$. Thus, in IsaFoR we defined an auxiliary relation covering all of match, match-rt, and $\mathsf{match\text{-}rt}_b$, and formalized the main proof step only once.

Currently, CeTA always chooses $\mathsf{match\text{-}rt}_b$ for non-collapsing $\mathcal{S}$, and match-rt, otherwise – the same as in current complexity tools.

5 Certifying Weak Dependency Pairs and Dependency Tuples

The dependency pair framework [9] is a popular setting for termination analysis. Since dependency pairs (DPs for short) in their original definition are not suitable for ensuring small (i.e., polynomial) derivation bounds [13], two variants have been developed. Hirokawa and Moser [10] introduced *weak dependency pairs* (WDPs for short). In general however, one cannot concentrate on counting WDP steps alone. Rather, one also has to take the number of interleaved steps w.r.t. the original TRS into account. Overcoming this complication, Noschinski et al. [15] introduced a variation, called *dependency tuples* (DTs for short). The DT transformation is however only applicable to innermost problems and it is not complete, so that (non-confluent) TRSs with polynomial complexity can be turned into complexity problems of exponential complexity.

Both WDPs and DTs enjoy nice properties that enable us to restrict to usable rules and limit the monotonicity requirements for reduction pairs, which we discuss later. Since the two techniques are incomparable but both used in modern complexity tools, we provide a formalization of either in IsaFoR. To be more precise, we have formalized the corresponding complexity processors of Avanzini and Moser [2], which – unlike DPs – allow us to apply WDPs and DTs also to relative problems.

As a case study, we decided to perform two different styles of proof: For DTs, we stuck more to the original paper proof, where parallel *positions* are used to point to subterms that are potential redexes; while for WDPs, we instead focused on *contexts* around potential redexes. The former requires us to reason about valid positions, whereas the latter makes it necessary to explicitly manage properties of contexts. Although both paper proofs are of comparable length, in our formalization the theories on WDPs are around 30 % shorter than those on DTs (see also `DT_Transformation(_Impl)` and `WDP_Transformation(_Impl)`). We

suspect that this is not mere coincidence, but caused by the fact that contexts can be mostly treated via explicit recursive functions, while positions require a different style of proof that is not as amenable to automation.

For the remainder of this section, we fix a runtime complexity problem $\langle \mathcal{S}/\mathcal{W}, \mathcal{Q}, \mathcal{T} \rangle$ over signature $\mathcal{F}$. For each $f \in \mathcal{F}$, let $f^\sharp$ be a function symbol fresh with respect to $\mathcal{F}$. For a term t we denote *sharping* its root symbol by $\sharp(t)$, where $\sharp(x) = x$ and $\sharp(f(t_1, \ldots, t_n)) = f^\sharp(t_1, \ldots, t_n)$. Sharping is homomorphically extended to sets and lists of symbols and terms.

Weak Dependency Pairs

We start with our formalization of WDPs as defined by Hirokawa and Moser [10].

▶ **Definition 9.** Let $\mathcal{R}$ be a TRS with defined symbols $\mathcal{D}(\mathcal{R})$. For every rule $\ell \to r \in \mathcal{R}$, let $\mathsf{WDP}(\ell \to r)$ denote the new rule $\sharp(\ell) \to \mathsf{COM}(\sharp(u_1), \ldots, \sharp(u_n))$, where $u_1, \ldots, u_n$ are the maximal subterms of r that are either variables or have a root symbol in $\mathcal{D}(\mathcal{R})$. Then the *weak dependency pairs* of $\mathcal{R}$ are defined by $\mathsf{WDP}(\mathcal{R}) = \{\mathsf{WDP}(\ell \to r) \mid \ell \to r \in \mathcal{R}\}$.

In the above definition COM denotes a "function" that assigns fresh function symbols of appropriate arity (a common optimization is to omit such symbols in case the argument list is singleton, i.e., $\mathsf{COM}(t) = t$) to a given list of terms. The thusly generated symbols are called *compound symbols*. Note that Definition 9 implies that for each rule $\ell \to r$ there is a unique ground context C such that $r = C[u_1, \ldots, u_n]$. This is captured by the following two functions:

$$\mathsf{cap}_{\mathcal{D}}(t) = \begin{cases} \Box & \text{if } t \in \mathcal{V} \text{ or } \mathsf{root}(t) \notin \mathcal{C} \\ f(\mathsf{cap}_{\mathcal{D}}(t_1), \ldots, \mathsf{cap}_{\mathcal{D}}(t_n)) & \text{if } t = f(t_1, \ldots, t_n) \text{ and } f \in \mathcal{C} \end{cases}$$

$$\mathsf{max}_{\mathcal{D}}(t) = \begin{cases} t & \text{if } t \in \mathcal{V} \text{ or } \mathsf{root}(t) \notin \mathcal{C} \\ \mathsf{max}_{\mathcal{D}}(t_1), \ldots, \mathsf{max}_{\mathcal{D}}(t_n) & \text{if } t = f(t_1, \ldots, t_n) \text{ and } f \in \mathcal{C} \end{cases}$$

where $\mathcal{C}$ is a set of symbols – which is supposed to contain the compound symbols and the constructors of $\mathcal{S} \cup \mathcal{W}$ – that is disjoint from sharped $\mathcal{F}$-symbols and the defined symbols of $\mathcal{S} \cup \mathcal{W}$, i.e., $(\mathcal{D}(\mathcal{S} \cup \mathcal{W}) \cup \sharp(\mathcal{F})) \cap \mathcal{C} = \varnothing$. Intuitively, $\mathsf{max}_{\mathcal{D}}(t)$ results in the list of maximal subterms of t that are either variables or have a root not in $\mathcal{C}$ (the latter usually implies that the root is a defined symbol; hence the notation), whereas $\mathsf{cap}_{\mathcal{D}}(t)$ computes the surrounding context. Together these two functions constitute a unique decomposition of a given term t, satisfying the property $t = (\mathsf{cap}_{\mathcal{D}}(t))[\mathsf{max}_{\mathcal{D}}(t)]$.

For certification we never actually have to construct the set of WDPs.[3] Instead it suffices to check whether a given pair of terms (p, q) constitutes a WDP for a given rule $\ell \to r$. This is done via the predicate:

$$\mathsf{is\text{-}WDP}(p, q)(\ell \to r) \;\longleftrightarrow\; p = \sharp(\ell) \wedge (\exists C.\, \mathsf{ground}(C) \wedge \mathcal{F}\mathsf{un}(C) \subseteq \mathcal{C} \wedge q = C[\sharp(\mathsf{max}_{\mathcal{D}}(r))])$$

In preparation for later results, we somewhat ambiguously use $\mathsf{WDP}(\mathcal{R})$ for $\mathcal{R} \subseteq \mathcal{S} \cup \mathcal{W}$ to denote an arbitrary set of rules (to be provided by the certificate) that is obligated to contain a WDP for each rule in $\mathcal{R}$, i.e.,

$$\forall \ell \to r \in \mathcal{R}.\, \exists (p, q) \in \mathsf{WDP}(\mathcal{R}).\, \mathsf{is\text{-}WDP}(p, q)(\ell \to r) \tag{5}$$

[3] Which allows us to avoid a tedious formalization of COM that would have to manage the generation of *fresh* symbols using a state monad or similar concept.

The main ingredient for soundness of WDPs is a simulation lemma that states that when two terms are in a certain relation, then every $\mathcal{R}$-rewrite sequence starting from the first term can be simulated by a $\mathsf{WDP}(\mathcal{R}) \cup \mathcal{R}$-rewrite sequence starting from the second one. The mentioned relation is crafted to fit the definition of WDPs. Intuitively, it relates terms whose respective maximal defined subterms (computed by $\mathsf{max}_{\mathcal{D}}$) only differ by sharp symbols. We write $s_1, \ldots, s_n \leq_\sharp t_1, \ldots, t_n$ when for each $i \leq n$ we have that either $s_i = t_i$ or $\sharp(s_i) = t_i$. Then the informal statement from above can be formalized as follows.

▶ **Definition 10.** A term t is *good for* a term s, written $t \gg s$, if and only if $\mathcal{F}\mathsf{un}(s) \subseteq \mathcal{F}$ and there are terms $t_1, \ldots, t_n$ and a ground context C with $\mathcal{F}\mathsf{un}(C) \subseteq \mathcal{C}$ such that $\mathsf{max}_{\mathcal{D}}(s) \leq_\sharp t_1, \ldots, t_n$ and $t = C[t_1, \ldots, t_n]$.

We borrow the terminology *good for* from Avanzini [1], although the above definition slightly differs from the original one. As indicated above, its intuition is that two related terms have the same redexes (or rather an over-approximation, namely, subterms with defined root) where in addition those in the left term may be sharped.

Before we state the main lemma, we give some useful properties of $\mathsf{max}_{\mathcal{D}}$.

▶ **Lemma 11.** *Let t be a term with $\mathsf{max}_{\mathcal{D}}(t) = t_1, \ldots, t_n$. Then:*

1. *If $\mathcal{F}\mathsf{un}(t) \subseteq \mathcal{F}$ and $\mathsf{max}_{\mathcal{D}}(t) \leq_\sharp u_1, \ldots, u_n$, then $\mathsf{max}_{\mathcal{D}}(u_i) = u_i$ for all $i \leq n$.*
2. *If $\mathcal{F}\mathsf{un}(t\sigma) \subseteq \mathcal{F}$ then $\mathsf{max}_{\mathcal{D}}(t\sigma) \leq_\sharp \mathsf{max}_{\mathcal{D}}(\sharp(t_1)\sigma), \ldots, \mathsf{max}_{\mathcal{D}}(\sharp(t_n)\sigma)$.*

In the main simulation lemma below, $\mathcal{Q}$ is extended to a set of terms $\mathcal{Q}'$ taking extensions of the signature $\mathcal{F}$ (by sharped and compound symbols) into account. In particular, the assumption on $\mathcal{Q}'$ ensures that innermost problems are translated to innermost problems, thereby allowing a proof-in-progress to continue with techniques that are specific to the innermost case. The following lemma shows that this does not pose any problems for rewriting, where $\mathcal{Q}_{\neg\mathcal{F}} = \{f(t_1, \ldots, t_n) \mid f \notin \mathcal{F}\}$.

▶ **Lemma 12.** *Every term t with $\mathcal{F}\mathsf{un}(t) \subseteq \mathcal{F}$ that is $\mathcal{Q}$-normal is also $\mathcal{Q}'$-normal for any $\mathcal{Q}' \subseteq \mathcal{Q} \cup \mathcal{Q}_{\neg\mathcal{F}}$.*

Proof. Assume that t is not $\mathcal{Q}'$-normal. Then $t = C[\ell\sigma]$ for some C, σ, and $\ell \in \mathcal{Q}'$; thus either $\ell \in \mathcal{Q}$, contradicting $\mathcal{Q}$-normality, or $\ell \in \mathcal{Q}_{\neg\mathcal{F}}$, contradicting $\mathcal{F}\mathsf{un}(t) \subseteq \mathcal{F}$. ◀

▶ **Lemma 13.** *Let $\mathcal{R} \subseteq \mathcal{S} \cup \mathcal{W}$ and $\mathcal{Q}' \subseteq \mathcal{Q} \cup \mathcal{Q}_{\neg\mathcal{F}}$. If $s \xrightarrow{\mathcal{Q}}_{\mathcal{R}} t$ and $u \gg s$ then there is a term v such that $u \xrightarrow{\mathcal{Q}'}_{\mathsf{WDP}(\mathcal{R}) \cup \mathcal{R}} v$ and $v \gg t$.*

Proof. From $s \xrightarrow{\mathcal{Q}}_{\mathcal{R}} t$ we obtain $\ell \to r \in \mathcal{R}$, σ, and C with $s = C[\ell\sigma]$ and $t = C[r\sigma]$. Moreover, all proper subterms of $\ell\sigma$ are $\mathcal{Q}$-normal. Let $s_1, \ldots, s_n$ denote the result of $\mathsf{max}_{\mathcal{D}}(s)$. Since every redex has a defined root, and all subterms of s with defined root are either contained in, or subterms of one of $s_1, \ldots, s_n$, we further obtain a context D such that $s_i = D[\ell\sigma]$ for some $i \leq n$. By Definition 10 and $u \gg s$, we get $u_1, \ldots, u_n$ with $s_1, \ldots, s_n \leq_\sharp u_1, \ldots, u_n$ and a ground context D' such that $u = D'[u_1, \ldots, u_n]$ and $\mathcal{F}\mathsf{un}(D') \subseteq \mathcal{C}$. Intuitively, it is easy to see that the above, together with Lemma 11(1), implies $\mathsf{max}_{\mathcal{D}}(u) = u_1, \ldots, u_n$ (although the corresponding formalization is somewhat tedious). Recall that $s = (\mathsf{cap}_{\mathcal{D}}(s))[s_1, \ldots, s_n]$ and the considered redex is a subterm of s_i, thus $t = (\mathsf{cap}_{\mathcal{D}}(s))[\mathsf{max}_{\mathcal{D}}(t)]$ with $\mathsf{max}_{\mathcal{D}}(t) = s_1, \ldots, \mathsf{max}_{\mathcal{D}}(D[r\sigma]), \ldots, s_n$. Moreover, from $s_1, \ldots, s_n \leq_\sharp u_1, \ldots, u_n$ we have $u_i = \sharp(D[\ell\sigma]) \vee u_i = D[\ell\sigma]$ and thus proceed by case analysis:

- Assume $u_i = \sharp(D[\ell\sigma])$.

- If $D \neq \Box$, then $\mathsf{max}_{\mathcal{D}}(D[r\sigma]) = D[r\sigma]$ and $\mathsf{cap}_{\mathcal{D}}(D[r\sigma]) = \Box$. We define $v = (\mathsf{cap}_{\mathcal{D}}(u))[u_1, \ldots, \sharp(D[r\sigma]), \ldots, u_n]$. Then, $u \xrightarrow{\mathcal{Q}'}_{\mathsf{WDP}(\mathcal{R}) \cup \mathcal{R}} v$ with the same rule $\ell \to r$, justified by choosing the context $(\mathsf{cap}_{\mathcal{D}}(u))[u_1, \ldots, \sharp(D), \ldots, u_n]$ and employing Lemma 12. Then, $v \gg t$, by definition of v and $\mathsf{max}_{\mathcal{D}}(t) \leq_\sharp u_1, \ldots, \sharp(D[r\sigma]), \ldots, u_n$.
- If $D = \Box$, then the WDP corresponding to $\ell \to r$ is used. From (5), we obtain a term q and a ground context E with $(\sharp(\ell), q) \in \mathsf{WDP}(\mathcal{R})$ and $q = E[\sharp(\mathsf{max}_{\mathcal{D}}(r))]$. Define $v = (\mathsf{cap}_{\mathcal{D}}(u))[u_1, \ldots, q\sigma, \ldots, u_n]$. Then $u \xrightarrow{\mathcal{Q}'}_{\mathsf{WDP}(\mathcal{R}) \cup \mathcal{R}} v$ as witnessed by $u = (\mathsf{cap}_{\mathcal{D}}(u))[u_1, \ldots, \sharp(\ell)\sigma, \ldots, u_n] \to (\mathsf{cap}_{\mathcal{D}}(u))[u_1, \ldots, q\sigma, \ldots, u_n] = v$ together with Lemma 12 (and noting that u is a proper subterm of $\sharp(\ell)\sigma$ if and only if u is a proper subterm of $\ell\sigma$). Moreover, let $\mathsf{max}_{\mathcal{D}}(r) = r_1, \ldots, r_k$, $E_j = \mathsf{cap}_{\mathcal{D}}(\sharp(r_j)\sigma)$, and $v_j = \mathsf{max}_{\mathcal{D}}(\sharp(r_j)\sigma)$ for all $j \leq k$. Hence, $v \gg t$, with $E' = (\mathsf{cap}_{\mathcal{D}}(u))[\ldots, E[E_1, \ldots, E_k], \ldots]$ and observing that $v = E'[u_1, \ldots, u_{i-1}, v_1, \ldots, v_k, u_{i+1}, \ldots, u_n]$ as well as $\mathsf{max}_{\mathcal{D}}(t) \leq_\sharp u_1, \ldots, u_{i-1}, v_1, \ldots, v_k, u_{i+1}, \ldots, u_n$. The latter follows from $\mathsf{max}_{\mathcal{D}}(r\sigma) \leq_\sharp v_1, \ldots, v_k$, which in turn is a consequence of Lemma 11(2) above.

- Assume $u_i = D[\ell\sigma]$. Then we can again employ the original rule $\ell \to r$. Let $E = (\mathsf{cap}_{\mathcal{D}}(u))[\ldots, \mathsf{cap}_{\mathcal{D}}(D[r\sigma]), \ldots]$ and $v = E[u_1, \ldots, u_{i-1}, \mathsf{max}_{\mathcal{D}}(D[r\sigma]), u_{i+1}, \ldots, u_n]$. We conclude $u \xrightarrow{\mathcal{Q}'}_{\mathsf{WDP}(\mathcal{R}) \cup \mathcal{R}} v$ and $v \gg t$ in a similar fashion as in the previous case. ◀

At this point, we obtain a simulation property for relative rewriting as an easy corollary.

▶ **Corollary 14.** *$u \gg s \xrightarrow{\mathcal{Q}}{}^{n}_{\mathcal{S}/\mathcal{W}} t$ implies $u \xrightarrow{\mathcal{Q}'}{}^{n}_{\mathsf{WDP}(\mathcal{S}) \cup \mathcal{S}/\mathsf{WDP}(\mathcal{W}) \cup \mathcal{W}} v \gg t$ for some v.*

▶ **Theorem 15** (WDP Processor)**.** *Let $\mathcal{P} = \langle \mathcal{S}/\mathcal{W}, \mathcal{Q}, \mathcal{T} \rangle$ be a runtime complexity problem. Then the WDP processor transforms $\mathcal{P}$ into $\mathcal{P}' = \langle \mathsf{WDP}(\mathcal{S}) \cup \mathcal{S}/\mathsf{WDP}(\mathcal{W}) \cup \mathcal{W}, \mathcal{Q}', \sharp(\mathcal{T}) \rangle$ for an arbitrary $\mathcal{Q}' \subseteq \mathcal{Q} \cup \mathcal{Q}_{\neg\mathcal{F}}$, and $\vdash \mathcal{P}' \colon G$ implies $\vdash \mathcal{P} \colon G$.*

Proof. Assume $\vdash \mathcal{P}' \colon g$ for some $g \in G$. Moreover, for the sake of a contradiction, assume that there is a term $s \in \mathcal{T}$ of size n and a rewrite sequence $s \xrightarrow{\mathcal{Q}}{}^{m}_{\mathcal{S}/\mathcal{W}} t$ of length $m > g(n)$. Since $s \in \mathcal{T}$, we have $\sharp(s) \in \mathcal{T}^\sharp$ and trivially $\sharp(s) \gg s$. Moreover, by Corollary 14, we obtain a term v with $\sharp(s) \xrightarrow{\mathcal{Q}'}{}^{m}_{\mathsf{WDP}(\mathcal{S}) \cup \mathcal{S}/\mathsf{WDP}(\mathcal{W}) \cup \mathcal{W}} v$, thereby contradicting the initial complexity judgment. ◀

▶ Remark. Note that when $\mathcal{P}$ is an innermost problem, by setting $\mathcal{Q}' = \mathcal{Q} \cup \sharp(\mathcal{Q})$ the WDP processor generates again an innermost problem. In contrast, Avanzini and Moser [1, 2] set $\mathcal{Q}'$ to $\mathcal{Q}$, thereby not retaining the innermost status as claimed.

Dependency Tuples

According to Theorem 15, we cannot focus on applications of weak dependency pairs in $\mathsf{WDP}(\mathcal{S})$ alone, but also have to account for applications of rules from $\mathcal{S}$. This may have severe consequences for a proof-in-progress. In the case of reduction pairs for instance, rather strict monotonicity requirements have to be imposed even after the WDP transformation. DTs overcome this weaknesses, but the corresponding transformation is sound only on innermost problems. In contrast to WDPs, which capture outermost calls, a DT captures *all* calls in a rule. The following definition is due to Noschinski et al. [15].

▶ **Definition 16.** Let $\mathcal{R}$ be a TRS with defined symbols $\mathcal{D}(\mathcal{R})$. For every rule $\ell \to r \in \mathcal{R}$, let $\mathsf{DT}(\ell \to r)$ denote the new rule $\sharp(\ell) \to \mathsf{COM}(\sharp(u_1), \ldots, \sharp(u_n))$, where $u_1, \ldots u_n$ are all subterms of r that have a root symbol in $\mathcal{D}(\mathcal{R})$. Then the *dependency tuples* of $\mathcal{R}$ are defined by $\mathsf{DT}(\mathcal{R}) = \{\mathsf{DT}(\ell \to r) \mid \ell \to r \in \mathcal{R}\}$.

As for weak dependency pairs, our formalization uses a predicate to decide whether a pair of terms (p, q) constitutes a dependency tuple of a rule $\ell \to r$. For a term t, let $\mathsf{Pos}_{\mathcal{D}}(t)$ denote the set of positions of subterms rooted by defined symbols of $\mathcal{S} \cup \mathcal{W}$.

$$\text{is-DT}(p,q)(\ell \to r) \;\longleftrightarrow\; p = \sharp(\ell) \wedge \big(\exists C\ p_1\ \dots\ p_n.\, \mathsf{Pos}_{\mathcal{D}}(r) = \{p_1, \dots, p_k\} \wedge q = C[\sharp(r|_{p_1}), \dots, \sharp(r|_{p_n})]\big)\ .$$

In the following, we use the notation $\mathsf{DT}(\mathcal{R})$ where $\mathcal{R} \subseteq \mathcal{S} \cup \mathcal{W}$, for a set satisfying

$$\forall \ell \to r \in \mathcal{R}.\, \exists (p,q) \in \mathsf{DT}(\mathcal{R}).\, \text{is-DT}(p,q)(\ell \to r)\ . \tag{6}$$

In the remainder, we provide a simulation lemma akin to Lemma 13 for DTs. For a term s, let $\mathsf{RPos}(s)$ denote the restriction of $\mathsf{Pos}_{\mathcal{D}}(s)$ to redex-positions. More precisely, $\mathsf{RPos}(s) = \{q \in \mathsf{Pos}_{\mathcal{D}}(s) \mid \exists t.\ s|_q \xrightarrow{\mathcal{Q}}_{\mathcal{S} \cup \mathcal{W}} t\}$. Closely following the proof by Avanzini [1], we use the following notion of *good for*.

▶ **Definition 17.** A term t is *good for* a term s, written $t \ggg s$, if and only if $\mathcal{F}\mathsf{un}(s) \subseteq \mathcal{F}$ and there is a context C such that $t = C[\sharp(s|_{q_1}), \dots, \sharp(s|_{q_k})]$ for positions $\{q_1, \dots, q_k\} = \mathsf{RPos}(s)$.

We now show that each $\mathcal{R}$-derivation of length n can be simulated by a corresponding derivation of $\mathsf{DT}(\mathcal{R})$ relative to $\mathcal{R}$, of length n. In the proof of the central simulation lemma, we use the following key observations.

▶ **Lemma 18.** *Let $\mathcal{R} \subseteq \mathcal{S} \cup \mathcal{W}$. Suppose $s \xrightarrow{\mathcal{Q}}_{\mathcal{R}} t$ is a step at redex position p with rule $\ell \to r$. Abbreviate $P = \{pq \mid q \in \mathsf{Pos}_{\mathcal{D}}(r)\}$ and $Q = \{q \in \mathsf{RPos}(s) \mid q < p \vee q \parallel p\}$. Then:*
1. *If $\mathsf{NF}(\mathcal{Q}) \subseteq \mathsf{NF}(\mathcal{S} \cup \mathcal{W})$ then $\mathsf{RPos}(t) \subseteq P \cup Q$;*
2. *$\sharp(s|_p) \xrightarrow{\mathcal{Q}}_{\mathsf{DT}(\mathcal{R})} C[\sharp(t|_{p_1}), \dots, \sharp(t|_{p_n})]$ for some context C and $\{p_1, \dots, p_n\} = P$;*
3. *$\sharp(s|_q) \xrightarrow{\mathcal{Q}}{}^{*}_{\mathcal{R}} \sharp(t|_q)$ for all positions $q \in Q$.*

▶ **Lemma 19.** *Let $\mathcal{R} \subseteq \mathcal{S} \cup \mathcal{W}$, suppose $\mathsf{NF}(\mathcal{Q}) \subseteq \mathsf{NF}(\mathcal{S} \cup \mathcal{W})$, and let $\mathcal{Q}' \subseteq \mathcal{Q} \cup \mathcal{Q}_{\neg \mathcal{F}}$. If $s \xrightarrow{\mathcal{Q}}_{\mathcal{R}} t$ and $u \ggg s$, then there is a term v such that $u \xrightarrow{\mathcal{Q}'}{}^{*}_{\mathcal{R}} \cdot \xrightarrow{\mathcal{Q}'}_{\mathsf{DT}(\mathcal{R})} v$ and $v \ggg t$.*

Proof. Consider terms s, t and u with $u \ggg s \xrightarrow{\mathcal{Q}}_{\mathcal{R}} t$. Let p denote the corresponding redex position. Define a function f from positions in s to marked terms as follows: $f(q) = \sharp(t|_q)$ if $q < p$ or $q \parallel p$ and $f(q) = \sharp(s|_q)$ otherwise. Since u is good for s, by Definition 17 we obtain a context C such that $u = C[\sharp(s|_{q_1}), \dots, \sharp(s|_{q_k})]$ for positions $\{q_1, \dots, q_k\} = \mathsf{RPos}(s)$. From Lemma 18(3) and the definition of f we see that $\sharp(s|_{q_i}) \xrightarrow{\mathcal{Q}}{}^{*}_{\mathcal{R}} f(q_i)$ $(i = 1, \dots, k)$ holds. Since $\mathcal{F}\mathsf{un}(s_j) \subseteq \mathcal{F}$ holds by assumption for all arguments of s_j of $\sharp(s|_{q_i})$, with Lemma 12 we can refine these sequences to $\sharp(s|_{q_i}) \xrightarrow{\mathcal{Q}'}{}^{*}_{\mathcal{R}} f(q_i)$ $(i = 1, \dots, k)$.

Observe that $p \in \mathsf{RPos}(s)$, i.e. $p = q_i$ for some $i \in \{1, \dots, k\}$. In particular $\sharp(s|_{q_i}) = \sharp(s|_p) = f(q_i)$ by definition of f. Lemma 18(2) yields a context D such that $f(q_i) \xrightarrow{\mathcal{Q}}_{\mathsf{DT}(\mathcal{R})} D[\sharp(t|_{p_1}), \dots, \sharp(t|_{p_n})]$, and consequently $f(q_i) \xrightarrow{\mathcal{Q}'}_{\mathsf{DT}(\mathcal{R})} D[\sharp(t|_{p_1}), \dots, \sharp(t|_{p_n})]$, for positions $\{p_1, \dots, p_n\} = \{pq \mid q \in \mathsf{Pos}_{\mathcal{D}}(r)\}$. Putting things together, we can thus construct a rewrite sequence

$$\begin{aligned} C[\sharp(s|_{q_1}), \dots, \sharp(s|_{q_i}), \dots, \sharp(s|_{q_k})] &\xrightarrow{\mathcal{Q}'}{}^{*}_{\mathcal{R}} C[f(q_1), \dots, f(q_i), \dots, f(q_k)] \\ &\xrightarrow{\mathcal{Q}'}_{\mathsf{DT}(\mathcal{R})} C[f(q_1), \dots, D[\sharp(t|_{p_1}), \dots, \sharp(t|_{p_n})], \dots, f(q_k)]\ . \end{aligned}$$

Let v be the last term of this sequence. We claim that v is good for t. Abbreviate $Q = \{q \in \mathsf{RPos}(s) \mid q < p \vee q \parallel p\}$. Observe that by Lemma 18(1), $\mathsf{RPos}(t) \subseteq Q \cup \{p_1, \dots, p_n\}$ holds. Since in particular $Q \subseteq \{q_1, \dots, q_{i-1}, q_{i+1}, \dots q_k\}$ and $f(q) = \sharp(t|_q)$ holds by definition of f for all positions $q \in Q$, it is not difficult to see that $v \ggg t$ holds. ◀

The lemma is straightforward to generalize to Q-restricted relative rewrite sequences.

▶ **Corollary 20.** *Suppose* $\mathsf{NF}(\mathcal{Q}) \subseteq \mathsf{NF}(\mathcal{R} \cup \mathcal{S})$*, and let* $\mathcal{Q}' \subseteq \mathcal{Q} \cup \mathcal{Q}_{\neg\mathcal{F}}$*. Then* $u \ggg s \xrightarrow{\mathcal{Q}}{}^{n}_{\mathcal{S}/\mathcal{W}} t$ *implies* $u \xrightarrow{\mathcal{Q}'}{}^{n}_{\mathsf{DT}(\mathcal{S})/\mathsf{DT}(\mathcal{W}) \cup \mathcal{S} \cup \mathcal{W}} v \ggg t$ *for some* v*.*

▶ **Theorem 21** (DT Processor)**.** *Let* $\mathcal{P} = \langle \mathcal{S}/\mathcal{W}, \mathcal{Q}, \mathcal{T} \rangle$ *be an innermost runtime complexity problem. Then the DT processor transforms* $\mathcal{P}$ *into* $\mathcal{P}' = \langle \mathsf{DT}(\mathcal{S})/\mathsf{DT}(\mathcal{W}) \cup \mathcal{S} \cup \mathcal{W}, \mathcal{Q}', \sharp(\mathcal{T}) \rangle$ *for an arbitrary* $\mathcal{Q}' \subseteq \mathcal{Q} \cup \mathcal{Q}_{\neg\mathcal{F}}$*, and* $\vdash \mathcal{P}' \colon G$ *implies* $\vdash \mathcal{P} \colon G$*.*

Proof. Soundness follows from Corollary 20, by reasoning similar to Theorem 15. ◀

Whenever WDPs or DTs are employed in a complexity proof, CeTA requires a clear indication which of the two methods has been applied. Moreover, the set of all WDPs (or DTs) has to be provided, as well as the extension of the strategy component: $\mathcal{Q}' \setminus \mathcal{Q}$. All other information is computed by CeTA, e.g., the set of compound symbols, renaming of variables, etc.

6 Usable Rules and Usable Replacement Maps

Computing usable rules is a simple syntactic technique for innermost termination; detecting that certain rules can never be applied in derivations starting from a given set of terms, and may thus be discarded. While for termination analysis, we start from right-hand sides of dependency pairs (instantiated by normal form substitutions); for complexity analysis, we employ the corresponding set of starting terms. Existing results on innermost usable rules for termination analysis made it quite easy to integrate usable rules for complexity analysis into IsaFoR, cf. `Usable_Rules_Complexity(_Impl)` and `Usable_Replacement_Map(_Impl)`.

Avanzini [1, Def. 14.44] as well as Hirokawa and Moser [10, Def. 14] define usable rules via usable symbols. Our formalization simplifies and generalizes both definitions.

▶ **Definition 22.** Let the set of starting terms $\mathcal{T}$ be included in $\mathcal{T}(\mathcal{F}', \mathcal{V})$. We define $\mathcal{US}$ to be a set of usable symbols and $\mathcal{U}$ a set of usable rules for $\mathcal{S}/\mathcal{W}$ w.r.t. $\mathcal{T}$, if the following three conditions are satisfied..

- $\mathcal{F}' \subseteq \mathcal{US}$
- whenever $\ell \to r \in \mathcal{S} \cup \mathcal{W}$ and $\mathcal{F}\mathsf{un}(\ell) \subseteq \mathcal{US}$, then $\ell \to r \in \mathcal{U}$, and
- whenever $\ell \to r \in \mathcal{U}$ then $\mathcal{F}\mathsf{un}(r) \subseteq \mathcal{US}$.

We believe the above definition to be simpler than previous ones, since we avoid reflexive transitive closures and do not distinguish between dependency pairs and other rules. Still, it is easy to check that Definition 22 simulates previous definitions, by choosing $\mathcal{US} = \mathcal{F}' \cup \mathcal{F}\mathsf{un}(\mathcal{U})$, where $\mathcal{U}$ is the set of usable rules as defined by Avanzini [1] and Hirokawa and Moser [10]. Moreover, Definition 22 is a generalization of the former, since all symbols of left-hand sides are considered, as opposed to just root symbols.

▶ **Example 23.** Let $\mathcal{S} \cup \mathcal{W} = \{\mathsf{f}^\sharp(\mathsf{g}) \to \mathsf{f}^\sharp(\mathsf{f}(\mathsf{g})), \mathsf{g}^\sharp \to \mathsf{com}, \mathsf{f}(\mathsf{g}) \to \mathsf{f}(\mathsf{f}(\mathsf{g})), \mathsf{g} \to \mathsf{a}\}$ and $\mathcal{T} = \mathsf{BT}(\{\mathsf{f}^\sharp, \mathsf{g}^\sharp\}, \{\mathsf{a}\}, \mathcal{V})$. Then according to [1, 10] all rules are usable, whereas Definition 22 allows us to use $\mathcal{F}' = \{\mathsf{f}^\sharp, \mathsf{g}^\sharp, \mathsf{a}\}$, $\mathcal{US} = \{\mathsf{f}^\sharp, \mathsf{g}^\sharp, \mathsf{a}, \mathsf{com}\}$, and $\mathcal{U} = \{\mathsf{g}^\sharp \to \mathsf{com}\}$.

For soundness of usable rules it is easy to prove that every derivation starting from $\mathcal{T}$ does only contain terms in $\mathcal{T}(\mathcal{US}, \mathcal{V})$. Hence we can remove all non-usable rules.

▶ **Theorem 24.** *If* $\mathcal{U}$ *is a set of usable rules for* $\mathcal{S}/\mathcal{W}$ *w.r.t.* $\mathcal{T}$*, then* $\vdash \langle \mathcal{S} \cap \mathcal{U} / \mathcal{W} \cap \mathcal{U}, \mathcal{Q}, \mathcal{T} \rangle \colon G$ *implies* $\vdash \langle \mathcal{S}/\mathcal{W}, \mathcal{Q}, \mathcal{T} \rangle \colon G$*.*

The whole formalization of this theorem via usable symbols, including definitions, occupies only 100 lines, without having to reuse existing results on usable rules in IsaFoR. This is in contrast to IsaFoR's integration of the variant of usable rules used in AProVE, cf. the end of Section 5.1 in [15]. Here, usable rules are based on unification and normal form checks, but only work for innermost rewriting. In this part of the formalization, we heavily reused the existing results for termination, and only little had to be added w.r.t. complexity analysis. As an example, for complexity with its relative rewrite relation, it was required to switch between a sequence of $\mathcal{S}/\mathcal{W}$-steps and a sequence that explicitly lists every single step in each relative $\to_{\mathcal{W}}^{*} \cdot \to_{\mathcal{S}} \cdot \to_{\mathcal{W}}^{*}$-step.

Since both variants of usable rules are incomparable, CeTA supports both. The certificate just requires the set of usable rules. It is then automatically inferred which of the two variants of usable rules is applicable.

Even less usable rules are obtained when employing argument filters from reduction pairs, a well-known technique from termination analysis. This technique has already been adapted for complexity, but we did not find any details in the literature. Thus, in the remainder of this section, we clarify how usable rules, reduction pairs, argument filters, and usable replacement maps can be combined. The upcoming theorem generalizes and improves existing complexity results on reduction pairs ([1, Thm. 14.10], [11, Cor. 20], and [15, Thm. 26]), since usable replacement maps can simulate safe reduction pairs of [11], cf. [1, Lemma 14.34].

Before presenting the main theorem, we first recapitulate the notion of usable replacement maps ([1, Def. 14.5] and [11, Def. 8]). These mainly indicate a superset of all positions where redexes may occur within terms of a derivation. To be more precise, for a replacement map μ, two TRSs $\mathcal{R}$ and $\mathcal{R}'$, and two sets of terms $\mathcal{Q}$ and $\mathcal{T}$; μ is a *usable replacement map* (written $\mathsf{URM}(\mu, \mathcal{Q}, \mathcal{R}, \mathcal{T}, \mathcal{R}')$), if for all $t \in \mathcal{T}$ and $t \xrightarrow{\mathcal{Q}}_{\mathcal{R}}^{*} s$, all redexes of s w.r.t. $\xrightarrow{\mathcal{Q}}_{\mathcal{R}'}$ are at μ-replacing positions of s.

Sufficient criteria to estimate usable replacement maps have been described in [11] for full and innermost rewriting, and in [1, Lemma 14.34] for WDPs and DTs, where currently CeTA only supports innermost rewriting, WDPs and DTs.

We will first present the main theorem, and then explain its ingredients and how to apply it. Here, a complexity pair $(\succ, \succsim)$ consists of two partial orders which are both closed under substitutions, which are compatible ($\succsim \cdot \succ \cdot \succsim \subseteq \succ$) and where $\succsim$ is reflexive. A reduction pair is a complexity pair where $\succsim$ is closed under contexts and $\succ$ is strongly normalizing.

▶ **Theorem 25.** *Let $\langle \mathcal{S}/\mathcal{W}, \mathcal{Q}, \mathcal{T} \rangle$ be an innermost runtime complexity problem with $\mathcal{T} = \mathsf{BT}(\mathcal{D}, \mathcal{C}, \mathcal{V})$. Define $\mathcal{R} = \mathcal{S} \cup \mathcal{W}$. Let $\mu_{\mathcal{S}}, \mu_{\mathcal{W}}$ be replacement maps, let π be an argument filter, let $\mathcal{U}$ be a set of usable rules, and let $(\succ, \succsim)$ be a complexity pair. If all of the following conditions are satisfied, then $\vdash \langle \mathcal{S}/\mathcal{W}, \mathcal{Q}, \mathcal{T} \rangle \colon G$.*

1. *Whenever $i \notin \pi(f)$, then $\succsim$ ignores the i-th argument of f.*
2. *Both $\mathsf{URM}(\mu_{\mathcal{S}}, \mathcal{Q}, \mathcal{R}, \mathcal{T}, \mathcal{S})$ and $\mathsf{URM}(\mu_{\mathcal{W}}, \mathcal{Q}, \mathcal{R}, \mathcal{T}, \mathcal{W})$.*
3. *Whenever $i \in \mu_{\mathcal{S}}(f)$ $(\mu_{\mathcal{W}}(f))$, then $\succ$ $(\succsim)$ is monotone in the i-th argument of f.*
4. *$\mathcal{S} \cap \mathcal{U} \subseteq \succ$ and $\mathcal{W} \cap \mathcal{U} \subseteq \succsim$.*
5. *If $\ell \to r \in \mathcal{R}$ and $\ell \in \mathcal{T}$ then $\ell \to r \in \mathcal{U}$.*
6. *$\mathcal{U}$ is closed under right-hand sides of usable rules w.r.t. $\mathcal{R}$ for both $\mu_{\mathcal{S}}$ and $\pi \cap \mu_{\mathcal{W}}$.*
7. *$\vdash \langle \succ/\varnothing, \varnothing, \mathcal{T} \rangle \colon G$*

In the theorem, we have two replacement maps $\mu_{\mathcal{S}}$ and $\mu_{\mathcal{W}}$ for the strict and weak rules as in [1, Thm. 14.10], but additionally there is the usual argument filter π indicating ignored argument positions of $\succsim$ which is used to reduce the set of usable rules. Let us shortly walk through all conditions of the theorem.

1. π is the standard argument filter as known from termination proofs via reduction pairs, e.g., if $[\mathsf{f}(x_1, x_2, x_3)] = 2x_2 + \frac{1}{2}x_3$, then $\pi(\mathsf{f}) = \{2, 3\}$.
2. Both $\mu_\mathcal{S}$ and $\mu_\mathcal{W}$ are estimated usable replacement maps, which can be computed by one of the methods above, where especially [1, Lemma 14.34] is often only applicable to generate $\mu_\mathcal{S}$.
3. The maps $\mu_\mathcal{S}$ and $\mu_\mathcal{W}$ indicate at which positions redexes may occur, and hence the corresponding orders $\succ$ and $\succsim$ must be monotone w.r.t. these positions.
4. Only usable rules have to be oriented by the complexity pair.
5. In the generation of usable rules, one starts to include all rules which have basic terms on their left-hand sides
6. and then performs the closure of usable rules w.r.t. an argument filter as in [16].
7. Finally, one extracts the derivation bound from the strict order $\succ$, and eventually derives the same bound for the input complexity problem.

We included this theorem into CeTA, where in the certificate just the complexity pair and the usable rules have to be provided, in combination with the strict rules for the split processor of Theorem 2. Since currently IsaFoR only has an interface for reduction pairs the latter condition in 3 does not have to be checked at runtime. All other information will be automatically inferred. To this end, we had to modify our interface of reduction pairs which now has to provide means of querying monotonicity of $\succ$ w.r.t. specific positions.

Using this theorem, CeTA could now certify most combinations of applying a complexity pair with usable rules and/or usable replacement maps in our experiments. Possible improvements at this point are the inclusion of better estimations of usable replacement maps, and better support for the complexity pairs itself, e.g., by removing the restriction to upper triangular matrix interpretations.

7 Experiments

We have tested our new formalization in combination with the only two complexity tools that apply several of the methods described in this paper: AProVE [8] (version 2015.01) and TCT [2] (version 2.2). Both were run on the *termination problem data base*, version 9.0.2,[4] which was also used for the complexity category of the FLoC Olympic Games of 2014. All tests were conducted on a machine with 8 dual core AMD Opteron™ 885 processors running at 2.60 GHz on 64 Gb of RAM and within a timeout of 60 seconds per test.

Table 1 collects our experimental findings. Here we show totals on estimated upper bounds (from constant to polynomial of unknown degree) on runtime complexities w.r.t. full and innermost rewriting, the former being only supported by TCT. To delineate the extend of our new formalization, we have compared the tools when run in various modes:

- In *certification mode* (columns *certification new*) we restrict tools to those methods that can also be certified by CeTA version 2.19. We contrast this data with results obtained from the version of TCT that ran in certification mode at the recent termination competition (columns *certification old*). Note that until now, AProVE did not feature certification support, consequently respective results are not present in the table.
- In *full mode* (columns *full*) we show totals when tools are run in their default setting, i.e., possibly employing methods that cannot be certified by CeTA.

[4] The TPDB is available at `http://termcomp.uibk.ac.at/`.

Table 1 Experimental Results.

	Full Rewriting			Innermost Rewriting				
	TCT			TCT			AProVE	
	certification		full	certification		full	certification	full
	new	old		new	old		new	
constant	0	0	18	0	0	38	1	53
linear	134	67	182	234	117	278	159	249
quadratic	165	107	201	291	157	341	250	350
cubic	165	110	202	299	160	354	283	387
polynomial	165	110	203	301	160	361	283	387

Overall, the experiments confirm significant improvements of CeTA's support for complexity analysis. For instance with TCT we certified polynomially bounded innermost runtime complexity of 301 systems. This corresponds to 83 % of the systems that can be handled by TCT when run in full mode. In contrast, relying on our old formalization TCT could handle only 44 % of the systems. The statement remains essentially correct for AProVE and TCT w.r.t. full rewriting.

Even more important might have been our preliminary experiments, where several proofs have been rejected by CeTA. Although the reason have often just been bugs in the proof-output of the tools, we also revealed and fixed (or at least reported to the developers) some more severe problems: one tool modified the sets $\mathcal{D}$ and $\mathcal{C}$ in the set of starting terms $\mathcal{T} = \mathsf{BT}(\mathcal{D}, \mathcal{C}, \mathcal{V})$ when deleting rules by the usable rules processor in a way that made the tool unnecessarily weak (and unsound for lower complexity bounds); one tool had a bug when computing usable rules which could be exploited to generate linear derivation bounds for non-terminating TRSs; and also some match-bounds certificates have been rejected where the corresponding code had to be disabled. Finally, also the required adaptation of $\mathcal{Q}$ to $\mathcal{Q}' \subseteq \mathcal{Q} \cup \mathcal{Q}_{\neg\mathcal{F}}$, as discussed in Section 5, was only detected by earlier versions of CeTA which did not support this possibility.

8 Conclusion

We presented our formalization of several techniques for complexity analysis that are now part of the formal library IsaFoR: match-bounds, weak dependency pairs, dependency tuples, usable rules, and usable replacement maps. Moreover, we reported on the resulting increase in power of our certifier CeTA, which is now able to certify more than three quarters of all complexity proofs that are generated by state-of-the-art tools.

Acknowledgments. This work was partially supported by Austrian Science Fund (FWF) projects Y757, J3563, and P27502. The authors are listed in alphabetical order regardless of individual contributions or seniority. We thank the anonymous reviewers for their helpful remarks.

References

1 Martin Avanzini. *Verifying Polytime Computability Automatically*. PhD thesis, University of Innsbruck, 2013. http://cl-informatik.uibk.ac.at/~zini/publications/diss.pdf.

2 Martin Avanzini and Georg Moser. A combination framework for complexity. In *Proc. 24th International Conference on Rewriting Techniques and Applications*, volume 21 of *Leibniz International Proceedings in Informatics*, pages 55–70, 2013. 10.4230/LIPIcs.RTA.2013.55.

3 Franz Baader and Tobias Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

4 Nachum Dershowitz and Zohar Manna. Proving termination with multiset orderings. *Communications of the ACM*, 22(8):465–476, 1979. 10.1145/359138.359142.

5 Bertram Felgenhauer and René Thiemann. Reachability analysis with state-compatible automata. In *Proc. 8th International Conference on Language and Automata Theory and Applications*, volume 8370 of *Lecture Notes in Computer Science*, pages 347–359, 2014. 10.1007/978-3-319-04921-2_28.

6 Guillaume Feuillade, Thomas Genet, and Valérie Viet Triem Tong. Reachability analysis over term rewriting systems. *Journal of Automated Reasoning*, 33:341–383, 2004. 10.1007/s10817-004-6246-0.

7 Alfons Geser, Dieter Hofbauer, Johannes Waldmann, and Hans Zantema. On tree automata that certify termination of left-linear term rewriting systems. *Information and Computation*, 205(4):512–534, 2007. 10.1016/j.ic.2006.08.007.

8 Jürgen Giesl, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, Thomas Ströder, Stephanie Swiderski, and René Thiemann. Proving termination of programs automatically with AProVE. In *Proc. 7th International Joint Conference on Automated Reasoning*, volume 8562 of *Lecture Notes in Computer Science*, pages 184–191, 2014. 10.1007/978-3-319-08587-6_13.

9 Jürgen Giesl, René Thiemann, and Peter 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*, volume 3452 of *Lecture Notes in Computer Science*, pages 301–331, 2005. 10.1007/978-3-540-32275-7_21.

10 Nao Hirokawa and Georg Moser. Automated complexity analysis based on the dependency pair method. In *Proc. 4th International Joint Conference on Automated Reasoning*, volume 5195 of *Lecture Notes in Computer Science*, pages 364–379, 2008. 10.1007/978-3-540-71070-7_32.

11 Nao Hirokawa and Georg Moser. Automated complexity analysis based on context-sensitive rewriting. In *Proc. Joint 25th International Conference on Rewriting Techniques and Applications and 12th International Conference on Typed Lambda Calculi and Applications*, volume 8560 of *Lecture Notes in Computer Science*, pages 257–271, 2014. 10.1007/978-3-319-08918-8_18.

12 Martin Korp and Aart Middeldorp. Match-bounds revisited. *Information and Computation*, 207(11):1259–1283, 2009. 10.1016/j.ic.2009.02.010.

13 Georg Moser and Andreas Schnabl. The derivational complexity induced by the dependency pair method. *Logical Methods in Computer Science*, 7(3:1):1–38, 2011. 10.2168/LMCS-7(3:1)2011.

14 Tobias Nipkow, Lawrence C. Paulson, and Makarius Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science.* Springer, 2002. 10.1007/3-540-45949-9.

15 Lars Noschinski, Fabian Emmes, and Jürgen Giesl. Analyzing innermost runtime complexity of term rewriting by dependency pairs. *Journal of Automated Reasoning*, 51(1):27–56, 2013. 10.1007/s10817-013-9277-6.

16 Christian Sternagel and René Thiemann. Certified subterm criterion and certified usable rules. In *Proc. 21st International Conference on Rewriting Techniques and Applications*, volume 6 of *Leibniz International Proceedings in Informatics*, pages 325–340, 2010. 10.4230/LIPIcs.RTA.2010.325.

17 Christian Sternagel and René Thiemann. Formalizing monotone algebras for certification of termination and complexity proofs. In *Proc. Joint 25th International Conference on Rewriting Techniques and Applications and 12th International Conference on Typed Lambda Calculi and Applications*, volume 8560 of *Lecture Notes in Computer Science*, pages 441–455, 2014. 10.1007/978-3-319-08918-8_30.

18 René Thiemann and Christian Sternagel. Certification of termination proofs using CeTA. In *Proc. 22nd International Conference on Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 452–468, 2009. 10.1007/978-3-642-03359-9_31.

19 Harald Zankl and Martin Korp. Modular complexity analysis via relative complexity. In *Proc. 21st International Conference on Rewriting Techniques and Applications*, volume 6 of *Leibniz International Proceedings in Informatics*, pages 385–400, 2010. 10.4230/LIPIcs.RTA.2010.385.

20 Harald Zankl and Martin Korp. Modular complexity analysis for term rewriting. *Logical Methods in Computer Science*, 10(1:19):1–34, 2014. 10.2168/LMCS-10(1:19)2014.

Dismatching and Local Disunification in $\mathcal{EL}$

Franz Baader, Stefan Borgwardt, and Barbara Morawska*

Theoretical Computer Science, TU Dresden, Germany
{baader,stefborg,morawska}@tcs.inf.tu-dresden.de

Abstract

Unification in Description Logics has been introduced as a means to detect redundancies in ontologies. We try to extend the known decidability results for unification in the Description Logic $\mathcal{EL}$ to disunification since negative constraints on unifiers can be used to avoid unwanted unifiers. While decidability of the solvability of general $\mathcal{EL}$-disunification problems remains an open problem, we obtain NP-completeness results for two interesting special cases: *dismatching problems*, where one side of each negative constraint must be ground, and *local* solvability of disunification problems, where we restrict the attention to solutions that are built from so-called atoms occurring in the input problem. More precisely, we first show that dismatching can be reduced to local disunification, and then provide two complementary NP-algorithms for finding local solutions of (general) disunification problems.

1998 ACM Subject Classification I.2.3 Deduction and Theorem Proving, I.2.4 Knowledge Representation Formalisms and Methods

Keywords and phrases Unification, Description Logics, SAT

Digital Object Identifier 10.4230/LIPIcs.RTA.2015.40

1 Introduction

Description logics (DLs) [6] are a family of logic-based knowledge representation formalisms, which can be used to represent the conceptual knowledge of an application domain in a structured and formally well-understood way. They are employed in various application areas, but their most notable success so far is the adoption of the DL-based language OWL [21] as standard ontology language for the semantic web. DLs allow their users to define the important notions (classes, relations) of the domain using concepts and roles; to state constraints on the way these notions can be interpreted using terminological axioms; and to deduce consequences such as subsumption (subclass) relationships from the definitions and constraints. The expressivity of a particular DL is determined by the constructors available for building concepts.

The DL $\mathcal{EL}$, which offers the concept constructors conjunction ($\sqcap$), existential restriction ($\exists r.C$), and the top concept ($\top$), has drawn considerable attention in the last decade since, on the one hand, important inference problems such as the subsumption problem are polynomial in $\mathcal{EL}$, even with respect to expressive terminological axioms [16]. On the other hand, though quite inexpressive, $\mathcal{EL}$ is used to define biomedical ontologies, such as the large medical ontology SNOMED CT.[1] For these reasons, the most recent OWL version, OWL 2, contains the profile OWL 2 EL,[2] which is based on a maximally tractable extension of $\mathcal{EL}$ [5].

* Supported by DFG under grant BA 1122/14-2.
[1] http://www.ihtsdo.org/snomed-ct/
[2] http://www.w3.org/TR/owl2-profiles/

© Franz Baader, Stefan Borgwardt, and Barbara Morawska;
licensed under Creative Commons License CC-BY

Unification in Description Logics was introduced in [12] as a novel inference service that can be used to detect redundancies in ontologies. It is shown there that unification in the DL $\mathcal{FL}_0$, which differs from $\mathcal{EL}$ in that existential restriction is replaced by value restriction ($\forall r.C$), is ExpTime-complete. The applicability of this result was not only hampered by this high complexity, but also by the fact that $\mathcal{FL}_0$ is not used in practice to formulate ontologies.

In contrast, as mentioned above, $\mathcal{EL}$ is employed to build large biomedical ontologies for which detecting redundancies is a useful inference service. For example, assume that one developer of a medical ontology defines the concept of a *patient with severe head injury* as

$$\mathsf{Patient} \sqcap \exists \mathsf{finding}.(\mathsf{Head_injury} \sqcap \exists \mathsf{severity}.\mathsf{Severe}), \tag{1}$$

whereas another one represents it as

$$\mathsf{Patient} \sqcap \exists \mathsf{finding}.(\mathsf{Severe_finding} \sqcap \mathsf{Injury} \sqcap \exists \mathsf{finding_site}.\mathsf{Head}). \tag{2}$$

Formally, these two concepts are not equivalent, but they are nevertheless meant to represent the same concept. They can obviously be made equivalent by treating the concept names Head_injury and Severe_finding as variables, and substituting the first one by $\mathsf{Injury} \sqcap \exists \mathsf{finding_site}.\mathsf{Head}$ and the second one by $\exists \mathsf{severity}.\mathsf{Severe}$. In this case, we say that the concepts are unifiable, and call the substitution that makes them equivalent a *unifier*. In [10], we were able to show that unification in $\mathcal{EL}$ is of considerably lower complexity than unification in $\mathcal{FL}_0$: the decision problem for $\mathcal{EL}$ is NP-complete. The main idea underlying the proof of this result is to show that any solvable $\mathcal{EL}$-unification problem has a local unifier, i.e., a unifier built from a polynomial number of so-called atoms determined by the unification problem. However, the brute-force "guess and then test" NP-algorithm obtained from this result, which guesses a local substitution and then checks (in polynomial time) whether it is a unifier, is not useful in practice. We thus developed a goal-oriented unification algorithm for $\mathcal{EL}$, which is more efficient since nondeterministic decisions are only made if they are triggered by "unsolved parts" of the unification problem. Another option for obtaining a more efficient unification algorithm is a translation to satisfiability in propositional logic (SAT): in [9] it is shown how a given $\mathcal{EL}$-unification problem Γ can be translated in polynomial time into a propositional formula whose satisfying valuations correspond to the local unifiers of Γ.

Intuitively, a unifier of two $\mathcal{EL}$ concepts proposes definitions for the concept names that are used as variables: in our example, we know that, if we define Head_injury as $\mathsf{Injury} \sqcap \exists \mathsf{finding_site}.\mathsf{Head}$ and Severe_finding as $\exists \mathsf{severity}.\mathsf{Severe}$, then the two concepts (1) and (2) are equivalent w.r.t. these definitions. Of course, this example was constructed such that the unifier (which is actually local) provides sensible definitions for the concept names used as variables. In general, the existence of a unifier only says that there is a structural similarity between the two concepts. The developer who uses unification as a tool for finding redundancies in an ontology or between two different ontologies needs to inspect the unifier(s) to see whether the definitions it suggests really make sense. For example, the substitution that replaces Head_injury by $\mathsf{Patient} \sqcap \mathsf{Injury} \sqcap \exists \mathsf{finding_site}.\mathsf{Head}$ and Severe_finding by $\mathsf{Patient} \sqcap \exists \mathsf{severity}.\mathsf{Severe}$ is also a local unifier, which however does not make sense. Unfortunately, even small unification problems like the one in our example can have too many local unifiers for manual inspection. In [2] we propose to restrict the attention to so-called minimal unifiers, which form a subset of all local unifiers. In our example, the nonsensical unifier is indeed not minimal. In general, however, the restriction to minimal unifiers may preclude interesting local unifiers. In addition, as shown in [2], computing minimal unifiers is actually harder than computing local unifiers (unless the polynomial hierarchy collapses). In the present paper, we propose disunification as a more direct approach

for avoiding local unifiers that do not make sense. In addition to positive constraints (requiring equivalence or subsumption between concepts), a disunification problem may also contain negative constraints (preventing equivalence or subsumption between concepts). In our example, the nonsensical unifier can be avoided by adding the dissubsumption constraint

$$\mathsf{Head_injury} \not\sqsubseteq^? \mathsf{Patient} \tag{3}$$

to the equivalence constraint (1) $\equiv^?$ (2).

Unification and disunification in DLs is actually a special case of unification and disunification modulo equational theories (see [12] and [10] for the equational theories respectively corresponding to $\mathcal{FL}_0$ and $\mathcal{EL}$). Disunification modulo equational theories has, e.g., been investigated in [17, 18]. It is well-known in unification theory that for effectively finitary equational theories, i.e., theories for which finite complete sets of unifiers can effectively be computed, disunification can be reduced to unification: to decide whether a disunification problem has a solution, one computes a finite complete set of unifiers of the equations and then checks whether any of the unifiers in this set also solves the disequations. Unfortunately, for $\mathcal{FL}_0$ and $\mathcal{EL}$, this approach is not feasible since the corresponding equational theories have unification type zero [10, 12], and thus finite complete sets of unifiers need not even exist. Nevertheless, it was shown in [14] that the approach used in [12] to decide unification (reduction to language equations, which are then solved using tree automata) can be adapted such that it can also deal with disunification. This yields the result that disunification in $\mathcal{FL}_0$ has the same complexity (ExpTime-complete) as unification.

For $\mathcal{EL}$, going from unification to disunification appears to be more problematic. In fact, the main reason for unification to be decidable and in NP is locality: if the problem has a unifier then it has a local unifier. We will show that disunification in $\mathcal{EL}$ is not local in this sense by providing an example of a disunification problem that has a solution, but no local solution. Decidability and complexity of disunification in $\mathcal{EL}$ remains an open problem, but we provide partial solutions that are of interest in practice. On the one hand, we investigate *dismatching problems*, i.e., disunification problems where the negative constraints are dissubsumptions $C \not\sqsubseteq^? D$ for which C or D is ground (i.e., does not contain a variable). Note that the dissubsumption (3) from above actually satisfies this restriction since Patient is not a variable. We prove that (general) solvability of dismatching problems can be reduced to *local disunification*, i.e., the question whether a given $\mathcal{EL}$-disunification problem has a *local* solution, which shows that dismatching in $\mathcal{EL}$ is NP-complete. On the other hand, we develop two specialized algorithms to solve local disunification problems that extend the ones for unification [9, 10]: a goal-oriented algorithm that reduces the amount of nondeterministic guesses necessary to find a local solution, as well as a translation to SAT. The reason we present two kinds of algorithms is that, in the case of unification, they have proved to complement each other well in first evaluations [1]: the goal-oriented algorithm needs less memory and finds minimal solutions faster, while the SAT reduction generates larger data structures (of cubic size), but outperforms the goal-oriented algorithm on unsolvable problems.

Full proofs of the results presented below can be found in [4].

2 Subsumption and dissubsumption in $\mathcal{EL}$

The syntax of $\mathcal{EL}$ is defined based on two sets $\mathsf{N_C}$ and $\mathsf{N_R}$ of *concept names* and *role names*, respectively. *Concept terms* are built from concept names using the constructors *conjunction* ($C \sqcap D$), *existential restriction* ($\exists r.C$ for $r \in \mathsf{N_R}$), and *top* ($\top$). An *interpretation* $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$

Table 1 Syntax and semantics of $\mathcal{EL}$.

Name	Syntax	Semantics
top	$\top$	$\top^{\mathcal{I}} := \Delta^{\mathcal{I}}$
conjunction	$C \sqcap D$	$(C \sqcap D)^{\mathcal{I}} := C^{\mathcal{I}} \cap D^{\mathcal{I}}$
existential restriction	$\exists r.C$	$(\exists r.C)^{\mathcal{I}} := \{x \mid \exists y.(x,y) \in r^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\}$

consists of a non-empty domain $\Delta^{\mathcal{I}}$ and an interpretation function that maps concept names to subsets of $\Delta^{\mathcal{I}}$ and role names to binary relations over $\Delta^{\mathcal{I}}$. This function is extended to concept terms as shown in the semantics column of Table 1.

A concept term C is *subsumed* by a concept term D (written $C \sqsubseteq D$) if for every interpretation $\mathcal{I}$ it holds that $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$. We write a *dissubsumption* $C \not\sqsubseteq D$ to abbreviate the fact that $C \sqsubseteq D$ does not hold. The two concept terms C and D are *equivalent* (written $C \equiv D$) if $C \sqsubseteq D$ and $D \sqsubseteq C$. Note that we use "$=$" to denote *syntactic* equality between concept terms, whereas "$\equiv$" denotes semantic equivalence.

Since conjunction is interpreted as intersection, we can treat $\sqcap$ as a commutative and associative operator, and thus dispense with parentheses in nested conjunctions. An *atom* is a concept name or an existential restriction. Hence, every concept term C is a conjunction of atoms or $\top$. We call the atoms in this conjunction the *top-level atoms* of C. Obviously, C is equivalent to the conjunction of its top-level atoms, where the empty conjunction corresponds to $\top$. An atom is *flat* if it is a concept name or an existential restriction of the form $\exists r.A$ with $A \in \mathsf{N_C}$.

Subsumption in $\mathcal{EL}$ is decidable in polynomial time [8] and can be checked by recursively comparing the top-level atoms of the two concept terms.

▶ **Lemma 1** ([10])**.** *For two atoms C, D, we have $C \sqsubseteq D$ iff $C = D$ is a concept name or $C = \exists r.C'$, $D = \exists r.D'$, and $C' \sqsubseteq D'$. If C, D are concept terms, then $C \sqsubseteq D$ iff for every top-level atom D' of D there is a top-level atom C' of C such that $C' \sqsubseteq D'$.*

We obtain the following contrapositive formulation characterizing dissubsumption.

▶ **Lemma 2.** *For two concept terms C, D, we have $C \not\sqsubseteq D$ iff there is a top-level atom D' of D such that for all top-level atoms C' of C it holds that $C' \not\sqsubseteq D'$.*

In particular, $C \not\sqsubseteq D$ is characterized by the existence of a top-level atom D' of D for which $C \not\sqsubseteq D'$ holds. By further analyzing the structure of atoms, we obtain the following.

▶ **Lemma 3.** *Let C, D be two atoms. Then we have $C \not\sqsubseteq D$ iff either*
1. *C or D is a concept name and $C \neq D$; or*
2. *$D = \exists r.D'$, $C = \exists s.C'$, and $r \neq s$; or*
3. *$D = \exists r.D'$, $C = \exists r.C'$, and $C' \not\sqsubseteq D'$.*

3 Disunification

As described in the introduction, we now partition the set $\mathsf{N_C}$ into a set of *(concept) variables* ($\mathsf{N_v}$) and a set of *(concept) constants* ($\mathsf{N_c}$). A concept term is *ground* if it does not contain any variables. We define a quite general notion of disunification problems that is similar to the equational formulae used in [18].

▶ **Definition 4.** A *disunification problem* Γ is a formula built from subsumptions of the form $C \sqsubseteq^? D$, where C and D are concept terms, using the logical connectives $\wedge$, $\vee$, and $\neg$. We use equations $C \equiv^? D$ to abbreviate $(C \sqsubseteq^? D) \wedge (D \sqsubseteq^? C)$, disequations $C \not\equiv^? D$ for $\neg(C \sqsubseteq^? D) \vee \neg(D \sqsubseteq^? C)$, and dissubsumptions $C \not\sqsubseteq^? D$ instead of $\neg(C \sqsubseteq^? D)$. A *basic disunification problem* is a conjunction of subsumptions and dissubsumptions. A *dismatching problem* is a basic disunification problem in which all dissubsumptions $C \not\sqsubseteq^? D$ are such that C or D is ground. Finally, a *unification problem* is a conjunction of subsumptions.

The definition of dismatching problems is partially motivated by the definition of *matching* in description logics, where similar restrictions are imposed on unification problems [7, 11, 23]. Another motivation comes from our experience that dismatching problems already suffice to formulate most of the negative constraints one may want to put on unification problems, as described in the introduction.

To define the semantics of disunification problems, we now fix a *finite signature* $\Sigma \subseteq \mathsf{N_C} \cup \mathsf{N_R}$ and assume that all disunification problems contain only concept terms constructed over the symbols in Σ. A *substitution* σ maps every variable in Σ to a ground concept term constructed over the symbols of Σ. This mapping can be extended to all concept terms (over Σ) in the usual way. A substitution σ *solves* a subsumption $C \sqsubseteq^? D$ if $\sigma(C) \sqsubseteq \sigma(D)$; it *solves* $\Gamma_1 \wedge \Gamma_2$ if it solves both Γ_1 and Γ_2; it solves $\Gamma_1 \vee \Gamma_2$ if it solves Γ_1 or Γ_2; and it solves $\neg\Gamma$ if it does not solve Γ. A substitution that solves a given disunification problem is called a *solution* of this problem. A disunification problem is *solvable* if it has a solution.

In contrast to unification, in disunification it does make a difference whether or not solutions may contain variables from $\mathsf{N_v} \cap \Sigma$ or additional symbols from $(\mathsf{N_C} \cup \mathsf{N_R}) \setminus \Sigma$ [17]. In the context of the application sketched in the introduction, restricting solutions to ground terms over Σ is appropriate: the finite signature Σ contains exactly the symbols that occur in the ontology to be checked for redundancy, and since a solution σ is supposed to provide definitions for the variables in Σ, it should not use the variables themselves to define them; moreover, definitions that contain symbols that are not in Σ would be meaningless to the user.

Reduction to basic disunification problems

We will consider only basic disunification problems in the following. The reason is that there is a straightforward NP-reduction from solvability of arbitrary disunification problems to solvability of basic disunification problems. In this reduction, we view all subsumptions occurring in the disunification problem as propositional variables and guess a satisfying valuation of the resulting propositional formula. It then suffices to check solvability of the basic disunification problem obtained as the conjunction of all subsumptions evaluated to true and the negations of all subsumptions evaluated to false. Since the problems considered in the following sections are all NP-complete, the restriction to basic disunification problems does not affect our complexity results. In the following, we thus restrict the attention to basic disunification problems, which we simply call *disunification problems* and consider them to be sets of subsumptions and dissubsumptions.

Reduction to flat disunification problems

We further simplify our analysis by considering *flat* disunification problems, which means that they may only contain *flat* dissubsumptions of the form $C_1 \sqcap \cdots \sqcap C_n \not\sqsubseteq^? D_1 \sqcap \cdots \sqcap D_m$

for flat atoms $C_1, \dots, C_n, D_1, \dots, D_m$ with $m, n \geq 0$,[3] and *flat* subsumptions of the form $C_1 \sqcap \cdots \sqcap C_n \sqsubseteq^? D_1$ for flat atoms $C_1, \dots, C_n, D_1$ with $n \geq 0$.

The restriction to flat disunification problems is without loss of generality: to flatten concept terms, one can simply introduce new variables and equations to abbreviate subterms [10]. Moreover, a subsumption of the form $C \sqsubseteq^? D_1 \sqcap \cdots \sqcap D_m$ is equivalent to $C \sqsubseteq^? D_1, \dots, C \sqsubseteq^? D_m$. Any solution of a disunification problem Γ can be extended to a solution of the resulting flat disunification problem Γ', and conversely every solution of Γ' also solves Γ.

This flattening procedure also works for unification problems. However, dismatching problems cannot without loss of generality be restricted to being flat since the introduction of new variables to abbreviate subterms may destroy the property that one side of each dissubsumption is ground (see also Section 4).

For solving flat unification problems, it has been shown that it suffices to consider so-called local solutions [10], which are restricted to use only the atoms occurring in the input problem. We extend this notion to disunification as follows. Let Γ be a flat disunification problem. We denote by At the set of all (flat) atoms occurring as subterms in Γ, by Var the set of variables occurring in Γ, and by $\mathsf{At}_{\mathsf{nv}} := \mathsf{At} \setminus \mathsf{Var}$ the set of *non-variable atoms* of Γ. Let $S \colon \mathsf{Var} \to 2^{\mathsf{At}_{\mathsf{nv}}}$ be an *assignment (for Γ)*, i.e. a function that assigns to each variable $X \in \mathsf{Var}$ a set $S_X \subseteq \mathsf{At}_{\mathsf{nv}}$ of non-variable atoms. The relation $>_S$ on Var is defined as the transitive closure of $\{(X, Y) \in \mathsf{Var}^2 \mid Y \text{ occurs in an atom of } S_X\}$. If this defines a strict partial order, i.e. $>_S$ is irreflexive, then S is called *acyclic*. In this case, we can define the substitution σ_S inductively along $>_S$ as follows: if X is minimal, then $\sigma_S(X) := \bigsqcap_{D \in S_X} D$; otherwise, assume that $\sigma_S(Y)$ is defined for all $Y \in \mathsf{Var}$ with $X > Y$, and define

$$\sigma_S(X) := \bigsqcap_{D \in S_X} \sigma_S(D).$$

It is easy to see that the concept terms $\sigma_S(D)$ are ground and constructed from the symbols of Σ, and hence σ_S is a valid candidate for a solution of Γ according to Definition 4.

▶ **Definition 5.** Let Γ be a flat disunification problem. A substitution σ is called *local* if there exists an acyclic assignment S for Γ such that $\sigma = \sigma_S$. The disunification problem Γ is *locally solvable* if it has a local solution, i.e. a solution that is a local substitution. *Local disunification* is the problem of checking flat disunification problems for local solvability.

Note that assignments and local solutions are defined only for *flat* disunification problems.

Obviously, local disunification is decidable in NP: We can guess an assignment S, and check it for acyclicity and whether the induced substitution solves the disunification problem in polynomial time. It has been shown [10] that unification in $\mathcal{EL}$ is *local* in the sense that the equivalent flattened problem has a local solution iff the original problem is solvable. Hence not only local, but also general solvability of unification problems in $\mathcal{EL}$ can be decided in NP. In addition, this shows that NP-hardness already holds for local unification, and thus also for local disunification.

▶ **Fact 6.** *Deciding local solvability of flat disunification problems in $\mathcal{EL}$ is* NP*-complete.*

The next example shows that disunification in $\mathcal{EL}$ is *not local* in this sense.

▶ **Example 7.** Consider the flat disunification problem

$$\Gamma := \{X \sqsubseteq^? B,\ A \sqcap B \sqcap C \sqsubseteq^? X,\ \exists r.X \sqsubseteq^? Y,\ \top \not\sqsubseteq^? Y,\ Y \not\sqsubseteq^? \exists r.B\}$$

[3] Recall that the empty conjunction is $\top$.

with variables X, Y and constants A, B, C. The substitution σ with $\sigma(X) := A \sqcap B \sqcap C$ and $\sigma(Y) := \exists r.(A \sqcap C)$ is a solution of Γ. For σ to be local, the atom $\exists r.(A \sqcap C)$ would have to be of the form $\sigma(D)$ for a non-variable atom D occurring in Γ. But the only candidates for D are $\exists r.X$ and $\exists r.B$, none of which satisfy $\exists r.(A \sqcap C) = \sigma(D)$.

We show that Γ cannot have another solution that is local. Assume to the contrary that Γ has a local solution γ. We know that $\gamma(Y)$ cannot be $\top$ since γ must solve the first dissubsumption. Furthermore, none of the constants A, B, C can be a top-level atom of $\gamma(Y)$ since this would contradict the third subsumption. That leaves only the non-variable atoms $\exists r.\gamma(X)$ and $\exists r.B$, which are ruled out by the last dissubsumption since both $\gamma(X)$ and B are subsumed by B.

The decidability and complexity of general solvability of disunification problems is still open. In the following, we first consider the special case of solving dismatching problems, for which we show a similar result as for unification: every dismatching problem can be polynomially reduced to a flat problem that has a local solution iff the original problem is solvable. The main difference is that this reduction is nondeterministic. In this way, we reduce dismatching to local disunification. We then provide two different NP-algorithms for the latter problem by extending the rule-based unification algorithm from [10] and adapting the SAT encoding of unification problems from [9]. These algorithms are more efficient than the brute-force "guess and then test" procedure on which our argument for Fact 6 was based.

4 Reducing dismatching to local disunification

As mentioned in Section 3, we cannot restrict our attention to flat dismatching problems without loss of generality. Instead, the nondeterministic algorithm we present in the following reduces any dismatching problem Γ to a flat *disunification* problem Γ' with the property that local solvability of Γ' is equivalent to the solvability of Γ. Since the algorithm takes at most polynomial time in the size of Γ, this shows that dismatching in $\mathcal{EL}$ is NP-complete. For simplicity, we assume that the subsumptions and the non-ground sides of the dissubsumptions have already been flattened using the approach mentioned in the previous section. This retains the property that all dissubsumptions have one ground side and does not affect the solvability of the problem.

Our procedure exhaustively applies a set of rules to the (dis)subsumptions in a dismatching problem (see Figures 1 and 2). In these rules, $C_1, \ldots, C_n$ and $D_1, \ldots, D_m$ are atoms. The rule Left Decomposition includes the special case where the left-hand side of $\mathfrak{s}$ is $\top$, in which case $\mathfrak{s}$ is simply removed from the problem. Note that at most one rule is applicable to any given (dis)subsumption. The choice which (dis)subsumption to consider next is don't care nondeterministic, but the choices in the rules Right Decomposition and Solving Left-Ground Dissubsumptions are don't know nondeterministic.

▶ **Algorithm 8.** Let Γ_0 be a dismatching problem. We initialize $\Gamma := \Gamma_0$. While any of the rules of Figures 1 and 2 is applicable to any element of Γ, choose one such element and apply the corresponding rule. If any rule application fails, then return "failure".

To see that every run of the nondeterministic algorithm terminates in polynomial time, note that each rule application takes only polynomial time in the size of the chosen (dis)subsumption. In particular, subsumptions between ground atoms can be checked in polynomial time [8]. Additionally, we can show that the algorithm needs at most polynomially many rule applications since each rule application decreases the following measure on Γ: we sum up all sizes of (dis)subsumptions in Γ to which a rule is still applicable, where the size

Right Decomposition:

Condition: This rule applies to $\mathfrak{s} = C_1 \sqcap \cdots \sqcap C_n \not\sqsubseteq^? D_1 \sqcap \cdots \sqcap D_m$ if $m = 0$ or $m > 1$, and $C_1, \ldots, C_n$, $D_1, \ldots, D_m$ are atoms.
Action: If $m = 0$, then *fail*. Otherwise, choose an index $i \in \{1, \ldots, m\}$ and replace $\mathfrak{s}$ by $C_1 \sqcap \cdots \sqcap C_n \not\sqsubseteq^? D_i$.

Left Decomposition:

Condition: This rule applies to $\mathfrak{s} = C_1 \sqcap \cdots \sqcap C_n \not\sqsubseteq^? D$ if $n = 0$ or $n > 1$, $C_1, \ldots, C_n$ are atoms, and D is a non-variable atom.
Action: Replace $\mathfrak{s}$ by $C_1 \not\sqsubseteq^? D, \ldots, C_n \not\sqsubseteq^? D$.

Atomic Decomposition:

Condition: This rule applies to $\mathfrak{s} = C \not\sqsubseteq^? D$ if C and D are non-variable atoms.
Action: Apply the first case that matches $\mathfrak{s}$:
a) if C and D are ground and $C \sqsubseteq D$, then *fail*;
b) if C and D are ground and $C \not\sqsubseteq D$, then remove $\mathfrak{s}$ from Γ;
c) if C or D is a constant, then remove $\mathfrak{s}$ from Γ;
d) if $C = \exists r.C'$ and $D = \exists s.D'$ with $r \neq s$, then remove $\mathfrak{s}$ from Γ;
e) if $C = \exists r.C'$ and $D = \exists r.D'$, then replace $\mathfrak{s}$ by $C' \not\sqsubseteq^? D'$.

Figure 1 Decomposition rules.

of $C \sqsubseteq^? D$ or $C \not\sqsubseteq^? D$ is defined as $|C| \cdot |D|$, and $|C|$ is the number of symbols needed to write down C (for details, see [4]).

Note that the Solving rule for left-ground dissubsumptions is not limited to non-flat dissubsumptions, and thus the algorithm completely eliminates all left-ground dissubsumptions from Γ. It is also easy to see that, if the algorithm is successful, then the resulting disunification problem Γ is flat. We now prove that this nondeterministic procedure is correct in the following sense.

▶ **Lemma 9.** *The dismatching problem Γ_0 is solvable iff there is a successful run of Algorithm 8 such that the resulting flat disunification problem Γ has a local solution.*

Proof Sketch. Soundness (i.e., the if direction) is easy to show, using Lemmas 1–3. Showing completeness (i.e., the only-if direction) is more involved. Basically, given a solution γ of Γ_0, we can use γ to guide the rule applications and extend γ to the newly introduced variables such that each rule application is successful and the invariant "*γ solves all (dis)subsumptions of Γ*" is maintained. Once no more rules can be applied, we have a flat disunification problem Γ of which the extended substitution γ is a (possibly non-local) solution. To obtain a local solution, we denote by At, Var, and $\mathsf{At}_{\mathsf{nv}}$ the sets as defined in Section 3 and define the assignment S induced by γ as:

$$S_X := \{D \in \mathsf{At}_{\mathsf{nv}} \mid \gamma(X) \sqsubseteq \gamma(D)\},$$

for all (old and new) variables $X \in \mathsf{Var}$. It can be shown that this assignment is acyclic and that the induced local substitution σ_S solves Γ, and thus also Γ_0 (see [4] for details). ◀

The disunification problem of Example 7 is in fact a dismatching problem. The rule Solving Left-Ground Dissubsumptions can be used to replace $\top \not\sqsubseteq^? Y$ with $Y \sqsubseteq^? \exists r.Z$. The presence of the new atom $\exists r.Z$ makes the solution σ introduced in Example 7 local.

Flattening Right-Ground Dissubsumptions:

Condition: This rule applies to $\mathfrak{s} = X \not\sqsubseteq^? \exists r.D$ if X is a variable and D is ground and is not a concept name.
Action: Introduce a new variable X_D and replace $\mathfrak{s}$ by $X \not\sqsubseteq^? \exists r.X_D$ and $D \sqsubseteq^? X_D$.

Flattening Left-Ground Subsumptions:

Condition: This rule applies to $\mathfrak{s} = C_1 \sqcap \cdots \sqcap C_n \sqcap \exists r_1.D_1 \sqcap \cdots \sqcap \exists r_m.D_m \sqsubseteq^? X$ if $m > 0$, X is a variable, $C_1, \dots, C_n$ are flat ground atoms, and $\exists r_1.D_1, \dots, \exists r_m.D_m$ are non-flat ground atoms.
Action: Introduce new variables $X_{D_1}, \dots, X_{D_m}$ and replace $\mathfrak{s}$ by $D_1 \sqsubseteq^? X_{D_1}, \dots, D_m \sqsubseteq^? X_{D_m}$ and $C_1 \sqcap \cdots \sqcap C_n \sqcap \exists r_1.X_{D_1} \sqcap \cdots \sqcap \exists r_m.X_{D_m} \sqsubseteq^? X$.

Solving Left-Ground Dissubsumptions:

Condition: This rule applies to $\mathfrak{s} = C_1 \sqcap \cdots \sqcap C_n \not\sqsubseteq^? X$ if X is a variable and $C_1, \dots, C_n$ are ground atoms.
Action: Choose one of the following options:

- Choose a constant $A \in \Sigma$ and replace $\mathfrak{s}$ by $X \sqsubseteq^? A$. If $C_1 \sqcap \cdots \sqcap C_n \sqsubseteq A$, then *fail*.
- Choose a role $r \in \Sigma$, introduce a new variable Z, replace $\mathfrak{s}$ by $X \sqsubseteq^? \exists r.Z$, $C_1 \not\sqsubseteq^? \exists r.Z$, ..., $C_n \not\sqsubseteq^? \exists r.Z$, and immediately apply Atomic Decomposition to each of these dissubsumptions.

Figure 2 Flattening and solving rules.

Together with Fact 6 and the NP-hardness of unification in $\mathcal{EL}$ [10], Lemma 9 yields the following complexity result.

▶ **Theorem 10.** *Deciding solvability of dismatching problems in $\mathcal{EL}$ is* NP*-complete.*

5 A goal-oriented algorithm for local disunification

In this section, we present an algorithm for local disunification that is based on transformation rules. Basically, to solve the subsumptions, this algorithm uses the rules of the goal-oriented algorithm for unification in $\mathcal{EL}$ [10, 3], which produces only local unifiers. Since any local solution of the disunification problem is a local unifier of the subsumptions in the problem, one might think that it is then sufficient to check whether any of the produced unifiers also solves the dissubsumptions. This would not be complete, however, since the goal-oriented algorithm for unification does *not* produce *all* local unifiers. For this reason, we have additional rules for solving the dissubsumptions. Both rule sets contain (deterministic) *eager* rules that are applied with the highest priority, and *nondeterministic* rules that are only applied if no eager rule is applicable. The goal of the eager rules is to enable the algorithm to detect obvious contradictions as early as possible in order to reduce the number of nondeterministic choices it has to make.

Let now Γ_0 be the flat disunification problem for which we want to decide local solvability, and let the sets At, Var, and $\mathsf{At}_{\mathsf{nv}}$ be defined as in Section 3. We assume without loss of generality that the dissubsumptions in Γ_0 have only a single atom on the right-hand side. If this is not the case, it can easily be achieved by exhaustive application of the nondeterministic rule Right Decomposition (see Figure 1) without affecting the complexity of the overall procedure.

Starting with Γ_0, the algorithm maintains a current disunification problem Γ and a current acyclic assignment S, which initially assigns the empty set to all variables. In addition, for each subsumption or dissubsumption in Γ, it maintains the information on whether it is *solved* or not. Initially, all subsumptions of Γ_0 are unsolved, except those with a variable on the

right-hand side, and all dissubsumptions in Γ_0 are unsolved, except those with a variable on the left-hand side and a non-variable atom on the right-hand side. Subsumptions of the form $C_1 \sqcap \cdots \sqcap C_n \sqsubseteq^? X$ and dissubsumptions of the form $X \not\sqsubseteq^? D$, for a non-variable atom D, are called *initially solved*. Intuitively, they only specify constraints on the assignment S_X. More formally, this intuition is captured by the process of *expanding* Γ w.r.t. the variable X, which performs the following actions:

- every initially solved subsumption $\mathfrak{s} \in \Gamma$ of the form $C_1 \sqcap \cdots \sqcap C_n \sqsubseteq^? X$ is expanded by adding the subsumption $C_1 \sqcap \cdots \sqcap C_n \sqsubseteq^? E$ to Γ for every $E \in S_X$, and
- every initially solved dissubsumption $X \not\sqsubseteq^? D \in \Gamma$ is expanded by adding $E \not\sqsubseteq^? D$ to Γ for every $E \in S_X$.

A (non-failing) application of a rule of our algorithm does the following:

- it solves exactly one unsolved subsumption or dissubsumption,
- it may extend the current assignment S by adding elements of $\mathsf{At}_{\mathsf{nv}}$ to some set S_X,
- it may introduce new flat subsumptions or dissubsumptions built from elements of At,
- it keeps Γ expanded w.r.t. all variables X.

Subsumptions and dissubsumptions are only added by a rule application or by expansion if they are not already present in Γ. If a new subsumption or dissubsumption is added to Γ, it is marked as unsolved, unless it is initially solved (because of its form). Solving subsumptions and dissubsumptions is mostly independent, except for expanding Γ, which can add new unsolved subsumptions and dissubsumptions at the same time, and may be triggered by solving a subsumption or a dissubsumption.

The rules dealing with subsumptions are depicted in Figure 3; these three eager and two nondeterministic rules are essentially the same as the ones in [3], with the only difference that the background ontology $\mathcal{T}$ used there is empty for our purposes. Note that several rules may be applicable to the same subsumption, and there is no preference between them. Using Eager Ground Solving, the algorithm can immediately evaluate ground subsumptions via the polynomial-time algorithm of [8]. If the required subsumption holds, it is marked as solved, and otherwise Γ cannot be solvable and hence the algorithm fails. Eager Solving detects when a subsumption trivially holds because the atom D from the right-hand side is already present on the left-hand side, either directly or via the assignment of a variable. Eager Extension is applicable in case the left-hand side of a subsumption is essentially equivalent to a single variable X due to all its atoms being "subsumed by" S_X. In this case, there is no other option but to add the right-hand side atom to S_X to solve the subsumption, and to expand Γ w.r.t. this new assignment. In case none of the eager rules apply to a subsumption, it can be solved nondeterministically by either extending the assignment of a variable that occurs on the left-hand side (Extension), or decomposing the subsumption by looking for matching existential restrictions on both sides (cf. Lemma 1).

The new rules for solving dissubsumptions are listed in Figure 4. These include variants of the Left Decomposition and Atomic Decomposition rules from the previous section (see Figure 1). In these two rules, which are eager, instead of removing dissubsumptions we mark them as solved. Additionally, Γ may have to be expanded if such a rule adds a new dissubsumption that is initially solved. The new nondeterministic rule Local Extension follows the same idea as the Solving rule for left-ground dissubsumptions (see Figure 2), but does not have to introduce new variables and atoms since we are looking only for local solutions. Note that the left-hand side of $\mathfrak{s}$ may be a variable, and then $\mathfrak{s}$ is of the form $Y \not\sqsubseteq^? X$. This dissubsumption is not initially solved, because X is not a non-variable atom.

▶ **Algorithm 11.** Let Γ_0 be a flat disunification problem. We initialize $\Gamma := \Gamma_0$ and $S_X := \emptyset$ for all variables $X \in \mathsf{Var}$. While Γ contains an unsolved subsumption or dissubsumption, do

Eager Ground Solving:

Condition: This rule applies to $\mathfrak{s} = C_1 \sqcap \cdots \sqcap C_n \sqsubseteq^? D \in \Gamma$, if $\mathfrak{s}$ is ground.
Action: The rule application fails if $\mathfrak{s}$ does not hold. Otherwise, $\mathfrak{s}$ is marked as *solved*.

Eager Solving:

Condition: This rule applies to $\mathfrak{s} = C_1 \sqcap \cdots \sqcap C_n \sqsubseteq^? D \in \Gamma$, if there is an index $i \in \{1, \ldots, n\}$, such that $C_i = D$ or $C_i = X \in \mathsf{Var}$ and $D \in S_X$.
Action: The application of the rule marks $\mathfrak{s}$ as *solved*.

Eager Extension:

Condition: This rule applies to $\mathfrak{s} = C_1 \sqcap \cdots \sqcap C_n \sqsubseteq^? D \in \Gamma$, if there is an index $i \in \{1, \ldots, n\}$, such that $C_i = X \in \mathsf{Var}$ and $\{C_1, \ldots, C_n\} \setminus \{X\} \subseteq S_X$.
Action: The application of the rule adds D to S_X. If this makes S cyclic, the rule application fails. Otherwise, Γ is expanded w.r.t. X and $\mathfrak{s}$ is marked as *solved*.

Decomposition:

Condition: This rule applies to $\mathfrak{s} = C_1 \sqcap \cdots \sqcap C_n \sqsubseteq^? \exists s.D \in \Gamma$, if there is at least one index $i \in \{1, \ldots, n\}$ with $C_i = \exists s.C$.
Action: The application of the rule chooses such an index i, adds $C \sqsubseteq^? D$ to Γ, expands Γ w.r.t. D if D is a variable, and marks $\mathfrak{s}$ as *solved*.

Extension:

Condition: This rule applies to $\mathfrak{s} = C_1 \sqcap \cdots \sqcap C_n \sqsubseteq^? D \in \Gamma$, if there is at least one index $i \in \{1, \ldots, n\}$ with $C_i \in \mathsf{Var}$.
Action: The application of the rule chooses such an index i and adds D to S_{C_i}. If this makes S cyclic, the rule application fails. Otherwise, Γ is expanded w.r.t. C_i and $\mathfrak{s}$ is marked as *solved*.

Figure 3 Rules for subsumptions.

the following:
1. **Eager rule application:** If eager rules are applicable to some unsolved subsumption or dissubsumption $\mathfrak{s}$ in Γ, apply an arbitrarily chosen one to $\mathfrak{s}$. If the rule application fails, return "failure".
2. **Nondeterministic rule application:** If no eager rule is applicable, let $\mathfrak{s}$ be an unsolved subsumption or dissubsumption in Γ. If one of the nondeterministic rules applies to $\mathfrak{s}$, choose one and apply it. If none of these rules apply to $\mathfrak{s}$ or the rule application fails, then return "failure".

Once all (dis)subsumptions in Γ are solved, return the substitution σ_S that is induced by the current assignment.

As with Algorithm 8, the choice which (dis)subsumption to consider next and which eager rule to apply is don't care nondeterministic, while the choice of which nondeterministic rule to apply and the choices inside the rules are don't know nondeterministic. Each of these latter choices may result in a different solution σ_S. All proof details for the following results can be found in [4].

▶ **Lemma 12.** *Every run of Algorithm 11 terminates in time polynomial in the size of* Γ_0.

Proof Sketch. We can show that each (dis)subsumption that is added by a rule or by expansion is either of the form $C \sqsubseteq^? D$ or $C \not\sqsubseteq^? D$, where $C, D \in \mathsf{At}$, or of the form $C_1 \sqcap \cdots \sqcap C_n \sqsubseteq^? E$, where $C_1 \sqcap \cdots \sqcap C_n$ is the left-hand side of a subsumption from the original problem Γ_0 and $E \in \mathsf{At}$. Obviously, there are only polynomially many such

Eager Top Solving:

Condition: This rule applies to $\mathfrak{s} = C \not\sqsubseteq^? \top \in \Gamma$.
Action: The rule application fails.

Eager Left Decomposition:

Condition: This rule applies to $\mathfrak{s} = C_1 \sqcap \cdots \sqcap C_n \not\sqsubseteq^? D \in \Gamma$ if $n = 0$ or $n > 1$, and $D \in \mathsf{At}_{\mathsf{nv}}$.
Action: The application of the rule marks $\mathfrak{s}$ as *solved* and, for each $i \in \{1, \dots, n\}$, adds $C_i \not\sqsubseteq^? D$ to Γ and expands Γ w.r.t. C_i if C_i is a variable.

Eager Atomic Decomposition:

Condition: This rule applies to $\mathfrak{s} = C \not\sqsubseteq^? D \in \Gamma$ if $C, D \in \mathsf{At}_{\mathsf{nv}}$.
Action: The application of the rule applies the first case that matches $\mathfrak{s}$:
a) if C and D are ground and $C \sqsubseteq D$, then the rule application fails;
b) if C and D are ground and $C \not\sqsubseteq D$, then $\mathfrak{s}$ is marked as *solved*;
c) if C or D is a concept name, then $\mathfrak{s}$ is marked as *solved*;
d) if $C = \exists r.C'$ and $D = \exists s.D'$ with $r \neq s$, then $\mathfrak{s}$ is marked as *solved*;
e) if $C = \exists r.C'$ and $D = \exists r.D'$, then $C' \not\sqsubseteq^? D'$ is added to Γ, Γ is expanded w.r.t. C' if C' is a variable and D' is not a variable, and $\mathfrak{s}$ is marked as *solved*.

Local Extension:

Condition: This rule applies to $\mathfrak{s} = C \not\sqsubseteq^? X \in \Gamma$ if $X \in \mathsf{Var}$.
Action: The application of the rule chooses $D \in \mathsf{At}_{\mathsf{nv}}$ and adds D to S_X. If this makes S cyclic, the rule application fails. Otherwise, the new dissubsumption $C \not\sqsubseteq^? D$ is added to Γ, Γ is expanded w.r.t. X, Γ is expanded w.r.t. C if C is a variable, and $\mathfrak{s}$ is marked as *solved*.

Figure 4 New rules for dissubsumptions.

(dis)subsumptions. Additionally, each rule application solves at least one (dis)subsumption and takes at most polynomial time. ◀

To show *soundness* of the procedure, assume that a run of the algorithm terminates with success, i.e. all subsumptions and dissubsumptions are solved. Let $\hat{\Gamma}$ be the set of all subsumptions and dissubsumptions produced by this run, S be the final assignment, and σ_S the induced substitution (see Section 3). To show that σ_S solves $\hat{\Gamma}$, and hence also Γ_0, we use induction on the following order on (dis)subsumptions.

▶ **Definition 13.** Consider any (dis)subsumption $\mathfrak{s}$ of the form $C_1 \sqcap \cdots \sqcap C_n \sqsubseteq^? C_{n+1}$ or $C_1 \sqcap \cdots \sqcap C_n \not\sqsubseteq^? C_{n+1}$ in $\hat{\Gamma}$.

- We define $m(\mathfrak{s}) := (m_1(\mathfrak{s}), m_2(\mathfrak{s}))$, where
  - $m_1(\mathfrak{s}) := \emptyset$ if $\mathfrak{s}$ is ground; otherwise, $m_1(\mathfrak{s}) := \{X_1, \dots, X_m\}$, where $\{X_1, \dots, X_m\}$ is the multiset of all variables occurring in $C_1, \dots, C_n, C_{n+1}$.
  - $m_2(\mathfrak{s}) := |\mathfrak{s}|$, where $|\mathfrak{s}|$ is the size of $\mathfrak{s}$, i.e. the number of symbols in $\mathfrak{s}$.
- The strict partial order $\succ$ on such pairs is the lexicographic order, where the second components are compared w.r.t. the usual order on natural numbers, and the first components are compared w.r.t. the multiset extension of $>_S$ [13].
- We extend $\succ$ to $\hat{\Gamma}$ by setting $\mathfrak{s}_1 \succ \mathfrak{s}_2$ iff $m(\mathfrak{s}_1) \succ m(\mathfrak{s}_2)$.

Since multiset extensions and lexicographic products of well-founded strict partial orders are again well-founded [13], $\succ$ is a well-founded strict partial order on $\hat{\Gamma}$. We can then use the fact that the (dis)subsumptions produced by Algorithm 11 are always smaller w.r.t. this

order than the (dis)subsumptions they were created from to prove the following lemma by well-founded induction over $\succ$.

▶ **Lemma 14.** *σ_S is a local solution of $\hat{\Gamma}$, and thus also of its subset Γ_0.*

To prove *completeness*, assume that σ is a local solution of Γ_0. We can show that σ can guide the choices of Algorithm 11 to obtain a local solution σ' of Γ_0 such that, for every variable X, we have $\sigma(X) \sqsubseteq \sigma'(X)$. The following invariants will be maintained throughout the run of the algorithm for the current set of (dis)subsumptions Γ and the current assignment S:

I. σ is a solution of Γ. II. For each $D \in S_X$, we have that $\sigma(X) \sqsubseteq \sigma(D)$.

By Lemma 1, chains of the form $\sigma(X_1) \sqsubseteq \sigma(\exists r_1.X_2), \dots \sigma(X_{n-1}) \sqsubseteq \sigma(\exists r_{n-1}.X_n)$ with $X_1 = X_n$ are impossible, and thus invariant II implies that S is acyclic. Hence, if extending S during a rule application preserves this invariant, this extension will not cause the algorithm to fail. In [4] it is shown that

- the invariants are maintained by the operation of expanding Γ;
- the application of an eager rule never fails and maintains the invariants; and
- if $\mathfrak{s}$ is an unsolved (dis)subsumption of Γ to which no eager rule applies, then there is a nondeterministic rule that can be successfully applied to $\mathfrak{s}$ while maintaining the invariants.

This concludes the proof of correctness of Algorithm 11, which provides a more goal-directed way to solve local disunification problems than blindly guessing an assignment as described in Section 4.

▶ **Theorem 15.** *The flat disunification problem Γ_0 has a local solution iff there is a successful run of Algorithm 11 on Γ_0.*

6 Encoding local disunification into SAT

The following reduction to SAT is a generalization of the one for unification problems in [9]. We again consider a flat disunification problem Γ and the sets At, Var, and $\mathsf{At}_{\mathsf{nv}}$ as in Section 3. Since we are restricting our considerations to *local* solutions, we can without loss of generality assume that the sets $\mathsf{N_v}$, $\mathsf{N_c}$, and $\mathsf{N_R}$ contain exactly the variables, constants, and role names occurring in Γ. To further simplify the reduction, we assume in the following that all flat dissubsumptions in Γ are of the form $X \not\sqsubseteq^? Y$ for variables X, Y. This is without loss of generality, which can be shown using a transformation similar to the flattening rules from Section 4.

The translation into SAT uses the propositional variables $[C \sqsubseteq D]$ for all $C, D \in \mathsf{At}$. The SAT problem consists of a set of clauses $\mathsf{Cl}(\Gamma)$ over these variables that express properties of (dis)subsumption in $\mathcal{EL}$ and encode the elements of Γ. The intuition is that a satisfying valuation of $\mathsf{Cl}(\Gamma)$ induces a local solution σ of Γ such that $\sigma(C) \sqsubseteq \sigma(D)$ holds whenever $[C \sqsubseteq D]$ is true under the valuation. The solution σ is constructed by first extracting an acyclic assignment S out of the satisfying valuation and then computing $\sigma := \sigma_S$. We additionally introduce the variables $[X > Y]$ for all $X, Y \in \mathsf{N_v}$ to ensure that the generated assignment S is indeed acyclic. This is achieved by adding clauses to $\mathsf{Cl}(\Gamma)$ that express that $>_S$ is a strict partial order, i.e. irreflexive and transitive.

Finally, we use the auxiliary variables $p_{C,X,D}$ for all $X \in \mathsf{N_v}$, $C \in \mathsf{At}$, and $D \in \mathsf{At}_{\mathsf{nv}}$ to express the restrictions imposed by dissubsumptions of the form $C \not\sqsubseteq^? X$ in clausal form. More precisely, whenever $[C \sqsubseteq X]$ is false for some $X \in \mathsf{N_v}$ and $C \in \mathsf{At}$, then the

dissubsumption $\sigma(C) \not\sqsubseteq \sigma(X)$ should hold. By Lemma 2, this means that we need to find an atom $D \in \mathsf{At}_{\mathsf{nv}}$ that is a top-level atom of $\sigma(X)$ and satisfies $\sigma(C) \not\sqsubseteq \sigma(D)$. This is enforced by making the auxiliary variable $p_{C,X,D}$ true, which makes $[X \sqsubseteq D]$ true and $[C \sqsubseteq D]$ false (see Definition 167).

▶ **Definition 16.** The set $\mathsf{Cl}(\Gamma)$ contains the following propositional clauses:

(I) *Translation of* Γ.

- **a.** For every subsumption $C_1 \sqcap \dots \sqcap C_n \sqsubseteq^? D$ in Γ with $D \in \mathsf{At}_{\mathsf{nv}}$:
  $\rightarrow [C_1 \sqsubseteq D] \vee \dots \vee [C_n \sqsubseteq D]$
- **b.** For every subsumption $C_1 \sqcap \dots \sqcap C_n \sqsubseteq^? X$ in Γ with $X \in \mathsf{N}_{\mathsf{v}}$, and every $E \in \mathsf{At}_{\mathsf{nv}}$:
  $[X \sqsubseteq E] \rightarrow [C_1 \sqsubseteq E] \vee \dots \vee [C_n \sqsubseteq E]$
- **c.** For every dissubsumption $X \not\sqsubseteq^? Y$ in Γ: $[X \sqsubseteq Y] \rightarrow$

(IV) *Properties of subsumptions between non-variable atoms.*

- **a.** For every $A \in \mathsf{N}_{\mathsf{c}}$: $\rightarrow [A \sqsubseteq A]$
- **b.** For every $A, B \in \mathsf{N}_{\mathsf{c}}$ with $A \neq B$: $[A \sqsubseteq B] \rightarrow$
- **c.** For every $\exists r.A, \exists s.B \in \mathsf{At}_{\mathsf{nv}}$ with $r \neq s$: $[\exists r.A \sqsubseteq \exists s.B] \rightarrow$
- **d.** For every $A \in \mathsf{N}_{\mathsf{c}}$ and $\exists r.B \in \mathsf{At}_{\mathsf{nv}}$:
  $[A \sqsubseteq \exists r.B] \rightarrow$ and $[\exists r.B \sqsubseteq A] \rightarrow$
- **e.** For every $\exists r.A, \exists r.B \in \mathsf{At}_{\mathsf{nv}}$:
  $[\exists r.A \sqsubseteq \exists r.B] \rightarrow [A \sqsubseteq B]$ and $[A \sqsubseteq B] \rightarrow [\exists r.A \sqsubseteq \exists r.B]$

(VI) *Transitivity of subsumption.*
For every $C_1, C_2, C_3 \in \mathsf{At}$: $[C_1 \sqsubseteq C_2] \wedge [C_2 \sqsubseteq C_3] \rightarrow [C_1 \sqsubseteq C_3]$

(VII) *Dissubsumptions of the form* $C \not\sqsubseteq^? X$ *with a variable* X.
For every $C \in \mathsf{At}$, $X \in \mathsf{N}_{\mathsf{v}}$:
$\rightarrow [C \sqsubseteq X] \vee \bigvee_{D \in \mathsf{At}_{\mathsf{nv}}} p_{C,X,D}$,
and additionally for every $D \in \mathsf{At}_{\mathsf{nv}}$:
$p_{C,X,D} \rightarrow [X \sqsubseteq D]$ and $p_{C,X,D} \wedge [C \sqsubseteq D] \rightarrow$

(VIII) *Properties of* $>$.

- **a.** For every $X \in \mathsf{N}_{\mathsf{v}}$: $[X > X] \rightarrow$
- **b.** For every $X, Y, Z \in \mathsf{N}_{\mathsf{v}}$: $[X > Y] \wedge [Y > Z] \rightarrow [X > Z]$
- **c.** For every $X, Y \in \mathsf{N}_{\mathsf{v}}$ and $\exists r.Y \in \mathsf{At}$: $[X \sqsubseteq \exists r.Y] \rightarrow [X > Y]$

The main difference to the encoding in [9] (apart from the fact that we consider (dis)subsumptions here instead of equivalences) lies in the clauses 7 that ensure the presence of a non-variable atom D that solves the dissubsumption $C \not\sqsubseteq^? X$ (cf. Lemma 2). We also need some additional clauses in 4 to deal with dissubsumptions. It is easy to see that $\mathsf{Cl}(\Gamma)$ can be constructed in time cubic in the size of Γ (due to the clauses in 6 and 2).

To show *soundness* of the reduction, let τ be a valuation of the propositional variables that satisfies $\mathsf{Cl}(\Gamma)$. We define the assignment S^τ as follows:

$$S^\tau_X := \{D \in \mathsf{At}_{\mathsf{nv}} \mid \tau([X \sqsubseteq D]) = 1\}.$$

In [4] it is shown that $X >_{S^\tau} Y$ implies $\tau([X > Y]) = 1$ and that this implies irreflexivity of $>_{S^\tau}$. This in particular shows that S^τ is acyclic. In the following, let σ_τ denote the substitution σ_{S^τ} induced by S^τ. In [4] it is shown that σ_τ is a solution of Γ by proving that for all atoms $C, D \in \mathsf{At}$ it holds that $\tau([C \sqsubseteq D]) = 1$ iff $\sigma_\tau(C) \sqsubseteq \sigma_\tau(D)$.

Since σ_τ is obviously local, this suffices to show soundness of the reduction.

▶ **Lemma 17.** *If* $\mathsf{Cl}(\Gamma)$ *is solvable, then* Γ *has a local solution.*

To show *completeness*, let σ be a local solution of Γ and $>_\sigma$ the resulting partial order on $\mathsf{N_v}$, defined as follows for all $X, Y \in \mathsf{N_v}$:

$$X >_\sigma Y \text{ iff } \sigma(X) \sqsubseteq \exists r_1. \dots \exists r_n.\sigma(Y) \text{ for some } r_1, \dots, r_n \in \mathsf{N_R} \text{ with } n \geq 1.$$

Note that $>_\sigma$ is irreflexive since $X >_\sigma X$ is impossible by Lemma 1, and it is transitive since $\sqsubseteq$ is transitive and closed under applying existential restrictions on both sides. Thus, $>_\sigma$ is a strict partial order. We define a valuation τ_σ as follows for all $C, D \in \mathsf{At}$, $E \in \mathsf{At_{nv}}$, and $X, Y \in \mathsf{N_v}$:

$$\tau_\sigma([C \sqsubseteq D]) := \begin{cases} 1 & \text{if } \sigma(C) \sqsubseteq \sigma(D) \\ 0 & \text{otherwise} \end{cases} \qquad \tau_\sigma([X > Y]) := \begin{cases} 1 & \text{if } X >_\sigma Y \\ 0 & \text{otherwise} \end{cases}$$

$$\tau_\sigma(p_{C,X,E}) := \begin{cases} 1 & \text{if } \sigma(X) \sqsubseteq \sigma(E) \text{ and } \sigma(C) \not\sqsubseteq \sigma(E) \\ 0 & \text{otherwise} \end{cases}$$

In [4] it is proved that τ_σ satisfies $\mathsf{Cl}(\Gamma)$, which shows completeness of the reduction.

▶ **Lemma 18.** *If* Γ *has a local solution, then* $\mathsf{Cl}(\Gamma)$ *is solvable.*

This completes the proof of the correctness of the translation presented in Definition 16, which provides us with a reduction of local disunification (and thus also of dismatching) to SAT. This SAT reduction has been implemented in our prototype system UEL,[4] which uses SAT4J[5] as external SAT solver. First experiments show that dismatching is indeed helpful for reducing the number and the size of unifiers. The runtime performance of the solver for dismatching problems is comparable to the one for pure unification problems.

7 Related and future work

Since Description Logics and Modal Logics are closely related [26], results on unification in one of these two areas carry over to the other one. In Modal Logics, unification has mostly been considered for expressive logics with all Boolean operators [19, 20, 25]. An important open problem in the area is the question whether unification in the basic modal logic K, which corresponds to the DL $\mathcal{ALC}$, is decidable. It is only known that relatively minor extensions of K have an undecidable unification problem [27]. Disunification also plays an important role in Modal Logics since it is basically the same as the admissibility problem for inference rules [15, 22, 24] (see [4] for details).

Regarding future work, we want to investigate the decidability and complexity of general disunification in $\mathcal{EL}$, and consider also the case where non-ground solutions are allowed. From a more practical point of view, we plan to implement also the goal-oriented algorithm for local disunification, and to evaluate the performance of both presented algorithms on real-world problems.

References

1 Franz Baader, Stefan Borgwardt, Julian Alfredo Mendez, and Barbara Morawska. UEL: Unification solver for $\mathcal{EL}$. In *Proc. DL'12*, volume 846 of *CEUR-WS*, pages 26–36, 2012.

4 version 1.3.0, available at `http://uel.sourceforge.net/`
5 `http://www.sat4j.org/`

2 Franz Baader, Stefan Borgwardt, and Barbara Morawska. Computing minimal $\mathcal{EL}$-unifiers is hard. In *Proc. AiML'12*, 2012.

3 Franz Baader, Stefan Borgwardt, and Barbara Morawska. A goal-oriented algorithm for unification in $\mathcal{EL}$ w.r.t. cycle-restricted TBoxes. In *Proc. DL'12*, volume 846 of *CEUR-WS*, pages 37–47, 2012.

4 Franz Baader, Stefan Borgwardt, and Barbara Morawska. Dismatching and local disunfication in $\mathcal{EL}$. LTCS-Report 15-03, Chair for Automata Theory, TU Dresden, Germany, 2015. See `http://lat.inf.tu-dresden.de/research/reports.html`.

5 Franz Baader, Sebastian Brandt, and Carsten Lutz. Pushing the $\mathcal{EL}$ envelope further. In *Proc. OWLED'08*, 2008.

6 Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.

7 Franz Baader, Ralf Küsters, Alex Borgida, and Deborah L. McGuinness. Matching in description logics. *J. Logic Comput.*, 9(3):411–447, 1999.

8 Franz Baader, Ralf Küsters, and Ralf Molitor. Computing least common subsumers in description logics with existential restrictions. In *Proc. IJCAI'99*, pages 96–101. Morgan Kaufmann, 1999.

9 Franz Baader and Barbara Morawska. SAT encoding of unification in $\mathcal{EL}$. In *Proc. LPAR'10*, volume 6397 of *LNCS*, pages 97–111. Springer, 2010.

10 Franz Baader and Barbara Morawska. Unification in the description logic $\mathcal{EL}$. *Log. Meth. Comput. Sci.*, 6(3), 2010.

11 Franz Baader and Barbara Morawska. Matching with respect to general concept inclusions in the description logic $\mathcal{EL}$. In *Proc. KI'14*, volume 8736 of *LNCS*, pages 135–146. Springer, 2014.

12 Franz Baader and Paliath Narendran. Unification of concept terms in description logics. *J. Symb. Comput.*, 31(3):277–305, 2001.

13 Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1999.

14 Franz Baader and Alexander Okhotin. Solving language equations and disequations with applications to disunification in description logics and monadic set constraints. In *Proc. LPAR'12*, volume 7180 of *LNCS*, pages 107–121. Springer, 2012.

15 Sergey Babenyshev, Vladimir V. Rybakov, Renate Schmidt, and Dmitry Tishkovsky. A tableau method for checking rule admissibility in $S4$. In *Proc. M4M-6*, 2009.

16 Sebastian Brandt. Polynomial time reasoning in a description logic with existential restrictions, GCI axioms, and—what else? In *Proc. ECAI'04*, pages 298–302, 2004.

17 Wray L. Buntine and Hans-Jürgen Bürckert. On solving equations and disequations. *J. of the ACM*, 41(4):591–629, 1994.

18 Hubert Comon. Disunification: A survey. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 322–359. MIT Press, 1991.

19 Silvio Ghilardi. Unification through projectivity. *J. Logic and Computation*, 7(6):733–752, 1997.

20 Silvio Ghilardi. Unification in intuitionistic logic. *J. Logic and Computation*, 64(2):859–880, 1999.

21 Ian Horrocks, Peter F. Patel-Schneider, and Frank van Harmelen. From SHIQ and RDF to OWL: The making of a web ontology language. *J. Web Sem.*, 1(1):7–26, 2003.

22 Rosalie Iemhoff and George Metcalfe. Proof theory for admissible rules. *Ann. Pure Appl. Logic*, 159(1-2):171–186, 2009.

23 Ralf Küsters. Chapter 6: Matching. In *Non-Standard Inferences in Description Logics*, volume 2100 of *LNCS*, pages 153–227. Springer, 2001.

24 Vladimir V. Rybakov. *Admissibility of logical inference rules*, volume 136 of *Studies in Logic and the Foundations of Mathematics*. North-Holland Publishing Co., Amsterdam, 1997.

25 Vladimir V. Rybakov. Multi-modal and temporal logics with universal formula - reduction of admissibility to validity and unification. *J. Logic and Computation*, 18(4):509–519, 2008.

26 Klaus Schild. A correspondence theory for terminological logics: Preliminary report. In *Proc. IJCAI'91*, pages 466–471, 1991.

27 Frank Wolter and Michael Zakharyaschev. Undecidability of the unification and admissibility problems for modal and description logics. *ACM Trans. Comput. Log.*, 9(4), 2008.

Nominal Anti-Unification*

Alexander Baumgartner[1], Temur Kutsia[1], Jordi Levy[2], and Mateu Villaret[3]

1 **Research Institute for Symbolic Computation (RISC)**
Johannes Kepler University, Linz, Austria
{abaumgar,kutsia}@risc.jku.at

2 **Artificial Intelligence Research Institute (IIIA)**
Spanish Council for Scientific Research (CSIC), Barcelona, Spain
levy@iiia.csic.es

3 **Departament d'Informàtica i Matemàtica Aplicada (IMA)**
Universitat de Girona (UdG), Girona, Spain
villaret@ima.udg.edu

Abstract

We study nominal anti-unification, which is concerned with computing least general generalizations for given terms-in-context. In general, the problem does not have a least general solution, but if the set of atoms permitted in generalizations is finite, then there exists a least general generalization which is unique modulo variable renaming and α-equivalence. We present an algorithm that computes it. The algorithm relies on a subalgorithm that constructively decides equivariance between two terms-in-context. We prove soundness and completeness properties of both algorithms and analyze their complexity. Nominal anti-unification can be applied to problems were generalization of first-order terms is needed (inductive learning, clone detection, etc.), but bindings are involved.

1998 ACM Subject Classification F.4.1 [Theory of Computation] Mathematical Logic and Formal Languages – Mathematical Logic, F.2.2 [Theory of Computation] Analysis of Algorithms and Problem Complexity – Nonnumerical Algorithms and Problems

Keywords and phrases Nominal Anti-Unification, Term-in-context, Equivariance

Digital Object Identifier 10.4230/LIPIcs.RTA.2015.57

1 Introduction

Binders are very common in computer science, logic, mathematics, linguistics. Functional abstraction λ, universal quantifier $\forall$, limit lim, integral $\int$ are some well-known examples of binders. To formally represent and study systems with binding, Pitts and Gabbay [15, 13, 14] introduced nominal techniques, based on the idea to give explicit names to bound entities. It makes a syntactic distinction between *atoms,* which can be bound, and *variables,* which can be substituted. This approach led to the development of the theory of nominal sets, nominal logic, nominal algebra, nominal rewriting, nominal logic programming, etc.

Equation solving between nominal terms (maybe together with freshness constraints) has been investigated by several authors, who designed and analyzed algorithms for nominal unification [30, 18, 19, 20, 6, 5], nominal matching [7], equivariant unification [9], and

* This research has been partially supported by the Spanish project HeLo (TIN2012-33042), by the Austrian Science Fund (FWF) with the project SToUT (P 24087-N18), and by the strategic program "Innovatives OÖ 2010plus" by the Upper Austrian Government.

© Alexander Baumgartner, Temur Kutsia, Jordi Levy, and Mateu Villaret;
licensed under Creative Commons License CC-BY

permissive nominal unification [10, 11]. However, in contrast to unification, its dual problem, anti-unification, has not been studied for nominal terms previously.

The anti-unification problem for two terms t_1 and t_2 is concerned with finding a term t that is more general than the original ones, i.e., t_1 and t_2 should be substitutive instances of t. The interesting generalizations are the least general ones, which retain the common structure of t_1 and t_2 as much as possible. Plotkin [23] and Reynolds [25] initiated research on anti-unification in the 1970s, developing generalization algorithms for first-order terms. Since then, anti-unification has been studied in various theories, including some of those with binding constructs: calculus of constructions [22], $M\lambda$ [12], second-order lambda calculus with type variables [21], simply-typed lambda calculus where generalizations are higher-order patterns [3], just to name a few.

The problem we address in this paper is to compute generalizations for nominal terms. More precisely, we consider this problem for nominal *terms-in-context*, which are pairs of a freshness context and a nominal term, aiming at computing their least general generalizations (lgg). However, it turned out that without a restriction, there is no lgg for terms-in-context, in general. Even more, a *minimal* complete set of generalizations does not exist. This is in sharp contrast with the related problem of anti-unification for higher-order patterns, which always have a single lgg [3]. The reason is one can make terms-in-context less and less general by adding freshness constraints for the available (infinitely many) atoms, see Example 2.7. Therefore, we restrict the set of atoms which are permitted in generalizations to be fixed and finite. In this case, there exists a single lgg (modulo α-equivalence and variable renaming) for terms-in-context and we design an algorithm to compute it in $O(n^5)$ time.

There is a close relation between nominal and higher-order pattern unification: One can be translated into the other by the solution-preserving translation defined in [8, 18, 20] or the translation defined for permisive terms in [10, 11]. We show that for anti-unification, this method, in general, is not applicable. Even if one finds conditions under which such a translation-based approach to anti-unification works, due to complexity reasons it is still better to use the direct nominal anti-unification algorithm developed in this paper.

Computation of nominal lgg's requires to solve the equivariance problem: Given two terms s_1 and s_2, find a permutation of atoms which, when applied to s_1, makes it α-equivalent to s_2 (under the given freshness context). This is necessary to guarantee that the computed generalization is *least* general. For instance, if the given terms are $s_1 = f(a, b)$ and $s_2 = f(b, a)$, where a, b are atoms, the freshness context is empty, and the atoms permitted in the generalization are a, b, and c, then the term-in-context $\langle\{c\#X, c\#Y\}, f(X, Y)\rangle$ generalizes $\langle\emptyset, s_1\rangle$ and $\langle\emptyset, s_2\rangle$, but it is not their lgg. To compute the latter, we need to reflect the fact that generalizations of the atoms are related to each other: One can be obtained from the other by swapping a and b. This leads to an lgg $\langle\{c\#X\}, f(X, (a\,b)\cdot X)\rangle$. To compute the permutation $(a\,b)$, an equivariance problem should be solved. Equivariance is already present in α-Prolog [28] and Isabelle [27]. We develop a rule-based algorithm for equivariance problems, which computes in quadratic time the justifying permutation if the input terms are equivariant, and fails otherwise.

Both anti-unification and equivariance algorithms are implemented in the anti-unification algorithm library [2] and can be accessed from `http://www.risc.jku.at/projects/stout/software/`.

Various variants of anti-unification, such as first-order, higher-order, or equational anti-unification have been used in inductive logic programming, logical and relational learning [24], reasoning by analogy [16], program synthesis [26], program verification [21], etc. Nominal anti-unification can, hopefully, contribute in solving similar problems in nominal setting or in first-order settings where bindings play an important role.

In this paper, we mainly follow the notation from [20]. Long proofs can be found in the technical report [4].

2 Nominal Terms

In *nominal signatures* we have *sorts of atoms* (typically ν) and *sorts of data* (typically δ) as disjoint sets. *Atoms* (typically $a, b, \ldots$) have one of the sorts of atoms. *Variables* (typically $X, Y, \ldots$) have a sort of atom or a sort of data, i.e. of the form $\nu \mid \delta$. In nominal terms, variables can be instantiated and atoms can be bound. Nominal function symbols (typically $f, g, \ldots$) have an arity of the form $\tau_1 \times \cdots \times \tau_n \to \delta$, where δ is a sort of data and τ_i are sorts given by the grammar $\tau ::= \nu \mid \delta \mid \langle\nu\rangle\tau$. Abstractions have sorts of the form $\langle\nu\rangle\tau$.

A *swapping* $(a\, b)$ is a pair of atoms of the same sort. A *permutation* is a (possibly empty) sequence of swappings. We use upright Greek letters (e.g., π, ρ) to denote permutations. *Nominal terms* (typically $t, s, u, r, q, \ldots$) are given by the grammar:

$$t ::= f(t_1, \ldots, t_n) \mid a \mid a.t \mid \pi{\cdot}X$$

where f is an n-ary function symbol, a is an atom, π is a permutation, and X is a variable. They are called respectively *application, atom, abstraction,* and *suspension*. The sorts of application and atomic terms are defined as usual, the sort of $a.t$ is $\langle\nu\rangle\tau$ where ν is the sort of a and τ is the sort of t, and the sort of $\pi{\cdot}X$ is the one of X.

The *inverse* of a permutation $\pi = (a_1\, b_1) \ldots (a_n\, b_n)$ is the permutation $(a_n\, b_n) \ldots (a_1\, b_1)$, denoted by π^{-1}. The empty permutation is denoted by *Id*. The *effect of a swapping* over an atom is defined by $(a\, b) \bullet a = b$, $(a\, b) \bullet b = a$ and $(a\, b) \bullet c = c$, when $c \notin \{a, b\}$. It is extended to the rest of terms: $(a\, b) \bullet f(t_1, \ldots, t_n) = f((a\, b) \bullet t_1, \ldots, (a\, b) \bullet t_n)$, $(a\, b) \bullet (c.t) = ((a\, b) \bullet c) \,.\, ((a\, b) \bullet t)$, and $(a\, b) \bullet \pi{\cdot}X = (a\, b)\pi{\cdot}X$, where $(a\, b)\pi$ is the permutation obtained by concatenating $(a\, b)$ and π. The *effect of a permutation* is defined by $(a_1\, b_1) \ldots (a_n\, b_n) \bullet t = (a_1\, b_1) \bullet ((a_2\, b_2) \ldots (a_n\, b_n) \bullet t)$. The effect of the empty permutation is $Id \bullet t = t$. We extend it to suspensions and write X as the shortcut of $Id{\cdot}X$.

The set of variables of a term t is denoted by $\mathrm{Vars}(t)$. A term t is called *ground* if $\mathrm{Vars}(t) = \emptyset$. The set of *atoms* of a term t or a permutation π is the set of all atoms which appear in it and is denoted by $\mathrm{Atoms}(t)$, $\mathrm{Atoms}(\pi)$ respectively. For instance, $\mathrm{Atoms}(f(a.g(a), (b\, c){\cdot}X, d) = \{a, b, c, d\}$. We write $\mathrm{Atoms}(t_1, \ldots, t_n)$ for the set $\mathrm{Atoms}(t_1) \cup \cdots \cup \mathrm{Atoms}(t_n)$.

Positions in terms are defined with respect to their tree representation in the usual way, as strings of integers. However, suspensions are put in a single leaf node. For instance, the tree form of the term $f(a.b.g((a\, b){\cdot}X, a), h(c))$, and the corresponding positions are shown in Fig. 1. The symbol f stands in the position ϵ (the empty sequence). The suspension is put in one node of the tree, at the position 1.1.1.1. The abstraction operator and the corresponding bound atom together occupy one node as well. For any term t, $t|_{\mathsf{p}}$ denotes the *subterm of t at position* p. For instance, $f(a.b.g((a\, b){\cdot}X, a), h(c))|_{1.1} = b.g((a\, b){\cdot}X, a)$.

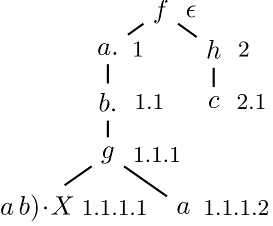


Figure 1 The tree form and positions of the term $f(a.b.g((a\, b){\cdot}X, a), h(c))$.

Every permutation π naturally defines a bijective function from the set of atoms to the sets of atoms, that we will also represent as π. Suspensions are uses of variables with a permutation of atoms waiting to be applied once the variable is instantiated. Occurrences of an atom a are said to be bound if they are in the scope of an abstraction of a, otherwise are said to be free. We denote by $\mathrm{FA}(t)$ the set of all atoms which occur freely in t: $\mathrm{FA}(f(t_1, \ldots, t_n)) = \bigcup_{i=1}^{n} \mathrm{FA}(t_i)$,

$\mathrm{FA}(a) = \{a\}$, $\mathrm{FA}(a.t) = \mathrm{FA}(t) \setminus \{a\}$, and $\mathrm{FA}(\pi{\cdot}X) = \mathrm{Atoms}(\pi)$. $\mathrm{FA}^{\text{-s}}(t)$ is the set of all atoms which occur freely in t ignoring suspensions: $\mathrm{FA}^{\text{-s}}(f(t_1,\dots,t_n)) = \bigcup_{i=1}^{n} \mathrm{FA}^{\text{-s}}(t_i)$, $\mathrm{FA}^{\text{-s}}(a) = \{a\}$, $\mathrm{FA}^{\text{-s}}(a.t) = \mathrm{FA}^{\text{-s}}(t) \setminus \{a\}$, and $\mathrm{FA}^{\text{-s}}(\pi{\cdot}X) = \emptyset$.

The head of a term t, denoted $\mathrm{Head}(t)$, is defined as: $\mathrm{Head}(f(t_1,\dots,t_n)) = f$, $\mathrm{Head}(a) = a$, $\mathrm{Head}(a.t) = .$, and $\mathrm{Head}(\pi{\cdot}X) = X$.

Substitutions are defined in the standard way, as a mapping from variables to terms of the same sort. We use Greek letters $\sigma, \vartheta, \varphi$ to denote substitutions. The identity substitution is denoted by ε. Furthermore, we use the postfix notation for substitution applications, i.e. $t\sigma$ denotes the application of a substitution σ to a term t, and similarly, the composition of two substitutions σ and ϑ is written as $\sigma\vartheta$. Composition of two substitutions is performed as usual. Their application allows atom capture, for instance, $a.X\{X \mapsto a\} = a.a$, and forces the permutation effect: $\pi{\cdot}X\{X \mapsto t\} = \pi \bullet t$, for instance, $(a\,b){\cdot}X\{X \mapsto f(a,(a\,b){\cdot}Y)\} = f(b,(a\,b)(a\,b){\cdot}Y)$. The notions of substitution *domain* and *range* are also standard and are denoted, respectively, by Dom and Ran.

A *freshness constraint* is a pair of the form $a\#X$ stating that the instantiation of X cannot contain free occurrences of a. A *freshness context* is a finite set of freshness constraints. We will use ∇ and Γ to denote freshness contexts. $\mathrm{Vars}(\nabla)$ and $\mathrm{Atoms}(\nabla)$ denote respectively the set of variables and atoms of ∇.

We say that a substitution σ *respects* a freshness context ∇, if for all X, $\mathrm{FA}^{\text{-s}}(X\sigma) \cap \{a \mid a\#X \in \nabla\} = \emptyset$.

The predicate $\approx$, which stands for α-equivalence between terms, and the freshness predicate $\#$ were defined in [29, 30] by the following theory:

$$\frac{}{\nabla \vdash a \approx a} \qquad \frac{\nabla \vdash t \approx t'}{\nabla \vdash a.t \approx a.t'} \qquad \frac{a \neq a' \quad \nabla \vdash t \approx (a\,a') \bullet t' \quad \nabla \vdash a\#t'}{\nabla \vdash a.t \approx a'.t'}$$

$$\frac{a\#X \in \nabla \text{ for all } a \text{ such that } \pi \bullet a \neq \pi' \bullet a}{\nabla \vdash \pi{\cdot}X \approx \pi'{\cdot}X} \qquad \frac{\nabla \vdash t_1 \approx t_1' \quad \cdots \quad \nabla \vdash t_n \approx t_n'}{\nabla \vdash f(t_1,\dots t_n) \approx f(t_1',\dots,t_n')}$$

where the freshness predicate $\#$ is defined by

$$\frac{a \neq a'}{\nabla \vdash a\#a'}(\#\text{-atom}) \qquad \frac{}{\nabla \vdash a\#a.t}(\#\text{-abst-1}) \qquad \frac{a \neq a' \quad \nabla \vdash a\#t}{\nabla \vdash a\#a'.t}(\#\text{-abst-2})$$

$$\frac{(\pi^{-1} \bullet a\#X) \in \nabla}{\nabla \vdash a\#\pi{\cdot}X}(\#\text{-susp.}) \qquad \frac{\nabla \vdash a\#t_1 \quad \cdots \quad \nabla \vdash a\#t_n}{\nabla \vdash a\#f(t_1,\dots t_n)}(\#\text{-application})$$

Their intended meanings are:

1. $\nabla \vdash a\#t$ holds, if for every substitution σ such that $t\sigma$ is a ground term and σ respects the freshness context ∇, we have a is not free in $t\sigma$;
2. $\nabla \vdash t \approx u$ holds, if for every substitution σ such that $t\sigma$ and $u\sigma$ are ground terms and σ respects the freshness context ∇, $t\sigma$ and $u\sigma$ are α-equivalent.

Based on the definition of the freshness predicate, we can design an algorithm, which we call FC, which solves the following problem: Given a set of freshness formulas $\{a_1\#t_1,\dots,a_n\#t_n\}$, compute a *minimal* (with respect to $\subseteq$) freshness context ∇ such that $\nabla \vdash a_1\#t_1,\dots,\nabla \vdash a_n\#t_n$. Such a ∇ may or may not exist, and the algorithm should detect it. The algorithm can be found in the technical report [4]. It is simply a bottom-up application of the rules of the freshness predicate, starting from each of the $\nabla \vdash a_1\#t_1,\dots,\nabla \vdash a_n\#t_n$. It succeeds if each branch of such a derivation tree is either closed (i.e., ends with the application of the #-atom or the #-abst-1 rule), or ends with an application of the #-susp. rule, producing a membership atom of the form $a\#X \in \nabla$ for some a and X. In this case we say that the

desired ∇ is the set of all such $a\#X$ freshness atoms, and write $\mathsf{FC}(\{a_1\#t_1,\dots,a_n\#t_n\}) = \nabla$. (Hence, ∇ is empty when all branches are closed.) It fails if at least one branch of the derivation tree produces $\nabla \vdash a\#a$ for some a, i.e., no rule applies to it. In this case we write $\mathsf{FC}(\{a_1\#t_1,\dots,a_n\#t_n\}) = \bot$. The following theorem is easy to verify:

▶ **Theorem 2.1.** *Let F be a set of freshness formulas and ∇ be a freshness context. Then $\mathsf{FC}(F) \subseteq \nabla$ iff $\nabla \vdash a\#t$ for all $a\#t \in F$.*

▶ **Corollary 2.2.** *$\mathsf{FC}(F) = \bot$ iff there is no freshness context that would justify all formulas in F.*

Given a freshness context ∇ and a substitution σ, we define $\nabla\sigma = \mathsf{FC}(\{a\#X\sigma \mid a\#X \in \nabla\})$. The following lemma is straightforward:

▶ **Lemma 2.3.** *σ respects ∇ iff $\nabla\sigma \neq \bot$.*

When $\nabla\sigma \neq \bot$, we call $\nabla\sigma$ the *instance* of ∇ under σ.

It is not hard to see that (a) if σ respects ∇, then σ respects any $\nabla' \subseteq \nabla$, and (b) if σ respects ∇ and ϑ respects $\nabla\sigma$, then $\sigma\vartheta$ respects ∇ and $(\nabla\sigma)\vartheta = \nabla(\sigma\vartheta)$.

▶ **Definition 2.4.** A *term-in-context* is a pair $\langle\nabla, t\rangle$ of a freshness context and a term. A term-in-context $\langle\nabla_1, t_1\rangle$ is *more general* than a term-in-context $\langle\nabla_2, t_2\rangle$, written $\langle\nabla_1, t_1\rangle \preceq \langle\nabla_2, t_2\rangle$, if there exists a substitution σ, which respects ∇_1, such that $\nabla_1\sigma \subseteq \nabla_2$ and $\nabla_2 \vdash t_1\sigma \approx t_2$.

We write $\nabla \vdash t_1 \preceq t_2$ if there exists a substitution σ such that $\nabla \vdash t_1\sigma \approx t_2$.

Two terms-in-context p_1 and p_2 are *equivalent* (or *equi-general*), written $p_1 \simeq p_2$, iff $p_1 \preceq p_2$ and $p_2 \preceq p_1$. The strict part of $\preceq$ is denoted by $\prec$, i.e., $p_1 \prec p_2$ iff $p_1 \preceq p_2$ and not $p_2 \preceq p_1$. We also write $\nabla \vdash t_1 \simeq t_2$ iff $\nabla \vdash t_1 \preceq t_2$ and $\nabla \vdash t_2 \preceq t_1$.

▶ **Example 2.5.** We give some examples to demonstrate the relations we have just defined:

- $\langle\{a\#X\}, f(a)\rangle \simeq \langle\emptyset, f(a)\rangle$. We can use $\{X \mapsto b\}$ for the substitution applied to the first pair.
- $\langle\emptyset, f(X)\rangle \preceq \langle\{a\#X\}, f(X)\rangle$ (with $\sigma = \varepsilon$), but not $\langle\{a\#X\}, f(X)\rangle \preceq \langle\emptyset, f(X)\rangle$.
- $\langle\emptyset, f(X)\rangle \preceq \langle\{a\#Y\}, f(Y)\rangle$ with $\sigma = \{X \mapsto Y\}$.
- $\langle\{a\#X\}, f(X)\rangle \not\preceq \langle\emptyset, f(Y)\rangle$, because in order to satisfy $\{a\#X\}\sigma \subseteq \emptyset$, the substitution σ should map X to a term t which contains neither a (freely) nor variables. But then $\emptyset \vdash f(t) \approx f(Y)$ does not hold. Hence, together with the previous example, we get $\langle\emptyset, f(Y)\rangle \prec \langle\{a\#X\}, f(X)\rangle$.
- $\langle\{a\#X\}, f(X)\rangle \not\preceq \langle\{a\#X\}, f(a)\rangle$. Notice that $\sigma = \{X \mapsto a\}$ does not respect $\{a\#X\}$.
- $\langle\{b\#X\}, (a\,b){\cdot}X\rangle \preceq \langle\{c\#X\}, (a\,c){\cdot}X\rangle$ with the substitution $\sigma = \{X \mapsto (a\,b)(a\,c){\cdot}X\}$. Hence, we get $\langle\{b\#X\}, (a\,b){\cdot}X\rangle \simeq \langle\{c\#X\}, (a\,c){\cdot}X\rangle$, because the $\succeq$ part can be shown with the help of the substitution $\{X \mapsto (a\,c)(a\,b){\cdot}X\}$.

▶ **Definition 2.6.** A term-in-context $\langle\Gamma, r\rangle$ is called a *generalization* of two terms-in-context $\langle\nabla_1, t\rangle$ and $\langle\nabla_2, s\rangle$ if $\langle\Gamma, r\rangle \preceq \langle\nabla_1, t\rangle$ and $\langle\Gamma, r\rangle \preceq \langle\nabla_2, s\rangle$. It is the *least general generalization,* (lgg in short) of $\langle\nabla_1, t\rangle$ and $\langle\nabla_2, s\rangle$ if there is no generalization $\langle\Gamma', r'\rangle$ of $\langle\nabla_1, t\rangle$ and $\langle\nabla_2, s\rangle$ which satisfies $\langle\Gamma, r\rangle \prec \langle\Gamma', r'\rangle$.

Note that if we have infinite number of atoms in the language, the relation $\prec$ is not well-founded: $\langle\emptyset, X\rangle \prec \langle\{a\#X\}, X\rangle \prec \langle\{a\#X, b\#X\}, X\rangle \prec \cdots$. As a consequence, two terms-in-context may not have an lgg and not even a minimal complete set of generalizations:[1]

[1] Minimal complete sets of generalizations are defined in the standard way. For a precise definition, see, e.g., [1, 17].

▶ **Example 2.7.** Let $p_1 = \langle \emptyset, a_1 \rangle$ and $p_2 = \langle \emptyset, a_2 \rangle$ be two terms-in-context. Then in any complete set of generalizations of p_1 and p_2 there is an infinite chain $\langle \emptyset, X \rangle \prec \langle \{a_3 \# X\}, X \rangle \prec \langle \{a_3 \# X, a_4 \# X\}, X \rangle \prec \cdots$, where $\{a_1, a_2, a_3, \ldots\}$ is the set of all atoms of the language. Hence, p_1 and p_2 do not have a minimal complete set of generalizations.

This example is a proof of the theorem, which characterizes the generalization type of nominal anti-unification:[2]

▶ **Theorem 2.8.** *The problem of anti-unification for terms-in-context is of nullary type.*

However, if we restrict the set of atoms which can be used in the generalizations to be fixed and finite, then the anti-unification problem becomes unitary. (We do not prove this property here, it will follow from the Theorems 6.2 and 6.3 in Sect. 6.)

▶ **Definition 2.9.** We say that a term t (resp., a freshness context ∇) is *based* on a set of atoms A iff $\mathrm{Atoms}(t) \subseteq A$ (resp., $\mathrm{Atoms}(\nabla) \subseteq A$). A term-in-context $\langle \nabla, t \rangle$ is based on A if both t and ∇ are based on it. We extend the notion of A-basedness to permutations, calling π A-based if it contains only atoms from A. Such a permutation defines a bijection, in particular, from A to A. If p_1 and p_2 are A-based terms-in-context, then their *A-based generalizations* are terms-in-context which are generalizations of p_1 and p_2 and are based on A. An *A-based lgg* of A-based terms-in-context p_1 and p_2 is a term-in-context p, which is an A-based generalization of p_1 and p_2 and there is no A-based generalization p' of p_1 and p_2 which satisfies $p \prec p'$.

The problem we would like to solve is the following:

Given: Two nominal terms t and s of the same sort, a freshness context ∇, and a *finite* set of atoms A such that t, s, and ∇ are based on A.
Find: A term r and a freshness context Γ, such that the term-in-context $\langle \Gamma, r \rangle$ is an A-based least general generalization of the terms-in-context $\langle \nabla, t \rangle$ and $\langle \nabla, s \rangle$.

Our anti-unification problem is parametric on the set of atoms we consider as the base, and finiteness of this set is essential to ensure the existence of an lgg.

3 Motivation of Using a Direct Nominal Anti-Unification Algorithm

In [20], relation between nominal unification (NU) and higher-order pattern unification (HOPU) has been studied. In particular, it was shown how to translate NU problems into HOPU problems and how to obtain nominal unifiers back from higher-order pattern unifiers. It is tempting to use the same translation for nominal anti-unification (NAU), using the algorithm from [3] to solve higher-order anti-unification problems over patterns (HOPAU), but it turns out that the generalization computed in this way is not always based on the same set of the atoms as the input:

▶ **Example 3.1.** We consider the following problem: Let the set of atoms be $A_1 = \{a, b\}$. The terms to be generalized are $a.b$ and $b.a$, and the freshness context is $\nabla = \emptyset$. According to [20], translation to higher-order patterns gives the anti-unification problem $\lambda a, b, a.\, b \triangleq \lambda a, b, b.\, a$, whose lgg is $\lambda a, b, c.X(a, b)$. However, we can not translate this lgg back to an A_1-based term-in-context, because it contains more bound variables than there are atoms in A_1.

[2] Generalization types are defined analogously to unification types: unary, finitary, infinitary, and nullary, see [17].

On the other hand, the translation would work for the set of atoms $A_2 = \{a, b, c\}$: Back-translating $\lambda a, b, c.X(a, b)$ gives the A_2-based lgg $\langle\{c\#X\}, c.X\rangle$.

The reason why the translation-based approach does not work for A-based NAU is that A is finite, while in higher-order anti-unification there is an infinite supply of fresh (bound) variables. If we assumed A to be infinite, there would still be a mismatch between NAU and the corresponding HOPAU: NAU, as we saw, is nullary in this case, while HOPAU is unitary. The reason of this contrast is that from infinitely many nominal generalizations, there is only one which is a well-typed higher-order generalization.

One might think that the translation-based approach would still work, if one considers only nominal anti-unification problems where the set of atoms is large enough for the input terms-in-context. However, there is a reason that speaks against NAU-to-HOPAU translation: complexity. The translation approach leads to a quadratic increase of the input size (Lemma 5.6 in [20]). The HOPAU algorithm in [3] runs in cubic time with respect to the size of its input. Hence, the translation-based approach leads to an algorithm with runtime complexity $O(n^6)$. In contrast, the algorithm developed in this paper has runtime complexity $O(n^5)$, and requires no back and forth translations.

4 The Lattice of More General Terms-In-Context

The notion of *more general term* defines an order relation between classes of terms (modulo some notion of *variable renaming*). In most cases, we have actually a meet-semilattice, since, given two terms, there always exists a greatest lower bound (meet) that corresponds to their anti-unifier. On the contrary, the least upper bound (join) of two terms only exists if they are unifiable. For instance, the two first-order terms $f(a, X_1)$ and $f(X_2, b)$ have a meet $f(Y_1, Y_2)$, and, since they are unifiable, also a join $f(a, b)$. Notice that unifiability and existence of a join are equivalent if both terms do not share variables (for instance $f(a, X)$ and $f(X, b)$ are both smaller than $f(a, b)$, hence joinable, but they are not unifiable). With this restriction one do not loose generality: The unification problem $t_1 \approx^? t_2$ (sharing variables), can be reduced to $f(t_1, t_2) \approx^? f(X, X)$ (not sharing variables), where f is some binary symbol and X a fresh variable. Therefore, in the first-order case, the problem of searching a most general unifier is equivalent to the search of the join of two terms, and the search of a least general generalization to the search of the meet. Notice that meet and join are unique up to some notion of *variable renaming*. For instance, the join of $f(a, X, X')$ and $f(Y, b, Y')$ is $f(a, b, Z)$ for any renaming of Z by any variable.

In the nominal case, we consider the set of terms-in-context (modulo variable renaming) with the more general relation. The following lemma establishes a correspondence between joinability and unifiability.

▶ **Lemma 4.1.** *Given two terms-in-context $\langle\nabla_1, t_1\rangle$ and $\langle\nabla_2, t_2\rangle$ with disjoint sets of variables, $\langle\nabla_1, t_1\rangle$ and $\langle\nabla_2, t_2\rangle$ are joinable if, and only if, $\{t_1 \approx^? t_2\} \cup \nabla_1 \cup \nabla_2$ has a solution (is unifiable).*

Like in first-order unification, the previous lemma allows us to reduce any nominal unification problem $P = \{a_1\#u_1, \dots, a_m\#u_m, t_1 \approx s_1, \dots, t_n \approx s_n\}$ into the joinability of the two terms-in-context $\langle\emptyset, f(X, X)\rangle$ and $\langle\mathsf{FC}(\{a_1\#u_1, \dots, a_m\#u_m\}), f(g(t_1, \dots, t_n), g(s_1, \dots, s_n))\rangle$ where f and g are any appropriate function symbols, and X is a fresh variable.

The nominal anti-unification problem is already stated in terms of finding the meet of two terms-in-context, with the only proviso that all terms and contexts must be based on some finite set of atoms.

5 Nominal Anti-Unification Algorithm

The triple $X : t \triangleq s$, where X, t, s have the same sort, is called the *anti-unification triple*, shortly AUT, and the variable X is called a *generalization variable*. We say that a set of AUTs P is based on a finite set of atoms A, if for all $X : t \triangleq s \in P$, the terms t and s are based on A.

▶ **Definition 5.1.** The nominal anti-unification algorithm is formulated in a rule-based way working on tuples P; S; Γ; σ and two global parameters A and ∇, where

- P and S are sets of AUTs such that if $X : t \triangleq s \in P \cup S$, then this is the sole occurrence of X in $P \cup S$;
- P is the set of AUTs to be solved;
- A is a finite set of atoms;
- The freshness context ∇ does not constrain generalization variables;
- S is a set of already solved AUTs (the store);
- Γ is a freshness context (computed so far) which constrains generalization variables;
- σ is a substitution (computed so far) mapping generalization variables to nominal terms;
- P, S, ∇, and Γ are A-based.

We call such a tuple a *state*. The rules below operate on states.

Dec: **Decomposition**

$$\{X : \mathsf{h}(t_1, \dots, t_m) \triangleq \mathsf{h}(s_1, \dots, s_m)\} \uplus P;\ S;\ \Gamma;\ \sigma \\ \Longrightarrow \{Y_1 : t_1 \triangleq s_1, \dots, Y_m : t_m \triangleq s_m\} \cup P;\ S;\ \Gamma;\ \sigma\{X \mapsto \mathsf{h}(Y_1, \dots, Y_m)\},$$

where h is a function symbol or an atom, $Y_1, \dots, Y_m$ are fresh variables of the corresponding sorts, $m \geq 0$.

Abs: **Abstraction**

$$\{X : a.t \triangleq b.s\} \uplus P;\ S;\ \Gamma;\ \sigma \Longrightarrow \{Y : (c\,a) \bullet t \triangleq (c\,b) \bullet s\} \cup P;\ S;\ \Gamma;\ \sigma\{X \mapsto c.Y\},$$

where Y is fresh, $c \in A$, $\nabla \vdash c\#a.t$ and $\nabla \vdash c\#b.s$.

Sol: **Solving**

$$\{X : t \triangleq s\} \uplus P;\ S;\ \Gamma;\ \sigma \Longrightarrow P;\ S \cup \{X : t \triangleq s\};\ \Gamma \cup \Gamma';\ \sigma,$$

if none of the previous rules is applicable, i.e. one of the following conditions hold:

(a) both terms have distinct heads: $\text{Head}(t) \neq \text{Head}(s)$, or

(b) both terms are suspensions: $t = \pi_1 \cdot Y_1$ and $s = \pi_2 \cdot Y_2$, where π_1, π_2 and Y_1, Y_2 are not necessarily distinct, or

(c) both are abstractions and rule Abs is not applicable: $t = a.t'$, $s = b.s'$ and there is no atom $c \in A$ satisfying $\nabla \vdash c\#a.t'$ and $\nabla \vdash c\#b.s'$.

The set Γ' is defined as $\Gamma' := \{a\#X \mid a \in A \ \wedge\ \nabla \vdash a\#t \ \wedge\ \nabla \vdash a\#s\}$.

Mer: **Merging**

$$P;\ \{X : t_1 \triangleq s_1, Y : t_2 \triangleq s_2\} \uplus S;\ \Gamma;\ \sigma \Longrightarrow \\ P;\ \{X : t_1 \triangleq s_1\} \cup S;\ \Gamma\{Y \mapsto \pi \cdot X\};\ \sigma\{Y \mapsto \pi \cdot X\},$$

where π is an $\text{Atoms}(t_1, s_1, t_2, s_2)$-based permutation such that $\nabla \vdash \pi \bullet t_1 \approx t_2$, and $\nabla \vdash \pi \bullet s_1 \approx s_2$.

The rules transform states to states. One can easily observe this by inspecting the rules.

Given a finite set of atoms A, two nominal A-based terms t and s, and an A-based freshness context ∇, to compute A-based generalizations for $\langle \nabla, t \rangle$ and $\langle \nabla, s \rangle$, we start with

$\{X : t \triangleq s\}; \emptyset; \emptyset; \varepsilon$, where X is a fresh variable, and apply the rules as long as possible. We denote this procedure by $\mathfrak{N}$. A *Derivation* is a sequence of state transformations by the rules. The state to which no rule applies has the form $\emptyset; S; \Gamma; \varphi$, where Mer does not apply to S. We call it the *final state.* When $\mathfrak{N}$ transforms $\{X : t \triangleq s\}; \emptyset; \emptyset; \varepsilon$ into a final state $\emptyset; S; \Gamma; \varphi$, we say that the *result computed* by $\mathfrak{N}$ is $\langle \Gamma, X\varphi \rangle$.

Note that the Dec rule works also for the AUTs of the form $X : a \triangleq a$. In the Abs rule, it is important to have the corresponding c in A. If we take $A = A_2$ in Example 3.1, then Abs can transform the AUT between t and s there, but if $A = A_1$ in the same example, then Abs is not applicable. In this case the Sol rule takes over, because the condition (c) of this rule is satisfied.

The condition (b) of Sol helps to compute, e,g, $\langle \emptyset, X \rangle$ for identical terms-in-context $\langle \emptyset, (a\,b)\cdot Y \rangle$ and $\langle \emptyset, (a\,b)\cdot Y \rangle$. Although one might expect that computing $\langle \emptyset, (a\,b)\cdot Y \rangle$ would be more natural, from the generalization point of view it does not matter, because $\langle \emptyset, X \rangle$ is as general as $\langle \emptyset, (a\,b)\cdot Y \rangle$.

▶ **Example 5.2.** We illustrate $\mathfrak{N}$ with the help of some examples:

- Let $t = f(a, b)$, $s = f(b, c)$, $\nabla = \emptyset$, and $A = \{a, b, c, d\}$. Then $\mathfrak{N}$ performs the following transformations:

  $$\{X : f(a, b) \triangleq f(b, c)\}; \emptyset; \emptyset; \varepsilon \Longrightarrow_{\mathsf{Dec}}$$
  $$\{Y : a \triangleq b, Z : b \triangleq c\}; \emptyset; \emptyset; \{X \mapsto f(Y, Z)\} \Longrightarrow^2_{\mathsf{Sol}}$$
  $$\emptyset; \{Y : a \triangleq b, Z : b \triangleq c\}; \{c\#Y, d\#Y, a\#Z, d\#Z\};\ \{X \mapsto f(Y, Z)\} \Longrightarrow_{\mathsf{Mer}}$$
  $$\emptyset; \{Y : a \triangleq b\}; \{c\#Y, d\#Y\}; \{X \mapsto f(Y, (a\,b)(b\,c)\cdot Y)\}$$

  Hence, $p = \langle \{c\#Y, d\#Y\}, f(Y, (a\,b)(b\,c)\cdot Y) \rangle$ is the computed result. It generalizes the input pairs: $p\{Y \mapsto a\} \preceq \langle \nabla, t \rangle$ and $p\{Y \mapsto b\} \preceq \langle \nabla, s \rangle$. The substitutions $\{Y \mapsto a\}$ and $\{Y \mapsto b\}$ can be read from the final store. Note that $\langle \{c\#Y\}, f(Y, (a\,b)(b\,c)\cdot Y) \rangle$ would be also an A-based generalization of $\langle \nabla, t \rangle$ and $\langle \nabla, s \rangle$, but it is strictly more general than p.
- Let $t = f(b, a)$, $s = f(Y, (a\,b)\cdot Y)$, $\nabla = \{b\#Y\}$, and $A = \{a, b\}$. Then $\mathfrak{N}$ computes the term-in-context $\langle \emptyset, f(Z, (a\,b)\cdot Z) \rangle$. It generalizes the input pairs.
- Let $t = f(g(X), X)$, $s = f(g(Y), Y)$, $\nabla = \emptyset$, and $A = \emptyset$. It is a first-order anti-unification problem. $\mathfrak{N}$ computes $\langle \emptyset, f(g(Z), Z) \rangle$. It generalizes the input pairs.
- Let $t = f(a.b, X)$, $s = f(b.a, Y)$, $\nabla = \{c\#X\}$, $A = \{a, b, c, d\}$. Then $\mathfrak{N}$ computes the term-in-context $p = \langle \{c\#Z_1, d\#Z_1\}, f(c.Z_1, Z_2) \rangle$. It generalizes the input pairs: $p\{Z_1 \mapsto b, Z_2 \mapsto X\} = \langle \emptyset, f(c.b, X) \rangle \preceq \langle \nabla, t \rangle$ and $p\{Z_1 \mapsto a, Z_2 \mapsto Y\} = \langle \emptyset, f(c.a, Y) \rangle \preceq \langle \nabla, s \rangle$.

6 Properties of the Nominal Anti-Unification Algorithm

The Soundness Theorem states that the result computed by $\mathfrak{N}$ is indeed an A-based generalization of the input terms-in-context:

▶ **Theorem 6.1** (Soundness of $\mathfrak{N}$). *Given terms t and s and a freshness context ∇, all based on a finite set of atoms A, if $\{X : t \triangleq s\};\ \emptyset;\ \emptyset;\ \varepsilon \Longrightarrow^+ \emptyset;\ S;\ \Gamma;\ \sigma$ is a derivation obtained by an execution of $\mathfrak{N}$, then $\langle \Gamma, X\sigma \rangle$ is an A-based generalization of $\langle \nabla, t \rangle$ and $\langle \nabla, s \rangle$.*

The Completeness Theorem states that for any given A-based generalization of two input terms-in-context, $\mathfrak{N}$ can compute one which is at most as general than the given one.

▶ **Theorem 6.2** (Completeness of $\mathfrak{N}$). *Given terms t and s and freshness contexts ∇ and Γ, all based on a finite set of atoms A. If $\langle \Gamma, r \rangle$ is an A-based generalization of $\langle \nabla, t \rangle$ and $\langle \nabla, s \rangle$, then there exists a derivation $\{X : t \triangleq s\};\ \emptyset;\ \emptyset;\ \varepsilon \Longrightarrow^+ \emptyset;\ S;\ \Gamma';\ \sigma$ obtained by an execution of $\mathfrak{N}$, such that $\langle \Gamma, r \rangle \preceq \langle \Gamma', X\sigma \rangle$.*

Depending on the selection of AUTs to perform a step, there can be different derivations in $\mathfrak{N}$ starting from the same AUT, leading to different generalizations. The next theorem states that all those generalizations are the same modulo variable renaming and α-equivalence.

▶ **Theorem 6.3** (Uniqueness Modulo $\simeq$). *Let t and s be terms and ∇ be a freshness context that are based on the same finite set of atoms. Let $\{X : t \triangleq s\}$; $\emptyset$; $\emptyset$; $\varepsilon \Longrightarrow^{+} \emptyset$; S_1; Γ_1; σ_1 and $\{X : t \triangleq s\}$; $\emptyset$; $\emptyset$; $\varepsilon \Longrightarrow^{+} \emptyset$; S_2; Γ_2; σ_2 be two maximal derivations in $\mathfrak{N}$. Then $\langle \Gamma_1, X\sigma_1 \rangle \simeq \langle \Gamma_2, X\sigma_2 \rangle$.*

Theorems 6.1, 6.2, and 6.3 imply that nominal anti-unification is unitary: For any A-based ∇, t, and s, there exists an A-based lgg of $\langle \nabla, t\rangle$ and $\langle \nabla, s\rangle$, which is unique modulo $\simeq$ and can be computed by the algorithm $\mathfrak{N}$.

Now we study how lgg's of terms-in-context depend on the set of atoms the terms-in-context are based on. The following lemma states the precise dependence.

▶ **Lemma 6.4.** *Let A_1 and A_2 be two finite sets of atoms with $A_1 \subseteq A_2$ such that the A_1-based terms-in-context $\langle \nabla, t\rangle$ and $\langle \nabla, s\rangle$ have an A_1-based lgg $\langle \Gamma_1, r_1\rangle$ and an A_2-based lgg $\langle \Gamma_2, r_2\rangle$. Then $\Gamma_2 \vdash r_1 \preceq r_2$.*

Proof. $\langle \Gamma_1, r_1\rangle$ and $\langle \Gamma_2, r_2\rangle$ are unique modulo $\simeq$. Let D_i be the derivation in $\mathfrak{N}$ that computes $\langle \Gamma_i, r_i\rangle$, $i = 1, 2$. The number of atoms in A_1 and A_2 makes a difference in the rule Abs: If there are not enough atoms in A_1, an Abs step in D_2 is replaced by a Sol step in D_1. It means that for all positions p of r_1, $r_2|_{\mathsf{p}}$ is also defined. Moreover, there might exist a subterm $r_1|_{\mathsf{p}}$, which has a form of suspension, while $r_2|_{\mathsf{p}}$ is an abstraction. For such positions, $r_1|_{\mathsf{p}} \preceq r_2|_{\mathsf{p}}$. For the other positions p' of r_1, $r_1|_{\mathsf{p}'}$ and $r_2|_{\mathsf{p}'}$ may differ only by names of generalization variables or by names of bound atoms.

Another difference might be in the application of Sol in both derivations: It can happen that this rule produces a larger Γ' in D_2 than in D_1, when transforming the same AUT.

Hence, if there are positions $\mathsf{p}_1, \ldots, \mathsf{p}_n$ in r_1 such that $r_1|_{\mathsf{p}_i} = \boldsymbol{\pi}_i \cdot X$, then there exists a substitution φ_X such that $\Gamma_2 \vdash \boldsymbol{\pi}_i \cdot X\varphi \approx r_2|_{\mathsf{p}_i}$, $1 \leq i \leq n$. Taking the union of all φ_X's where $X \in \mathrm{Vars}(r_1)$, we get φ with the property $\Gamma_2 \vdash r_1\varphi \approx r_2$. ◀

Note that, in general, we can not replace $\Gamma_2 \vdash r_1 \preceq r_2$ with $\Gamma_2 \vdash r_1 \simeq r_2$ in Lemma 6.4. The following example illustrates this:

▶ **Example 6.5.** Let $t = a.b$, $s = b.a$, $\nabla = \emptyset$, $A_1 = \{a, b\}$, and $A_2 = \{a, b, c\}$. Then for $\langle \nabla, t\rangle$ and $\langle \nabla, s\rangle$, $\langle \emptyset, X\rangle$ is an A_1-based lgg and $\langle \{c\#X\}, c.X\rangle$ is an A_2-based lgg. Obviously, $\{c\#X\} \vdash X \preceq c.X$ but not $\{c\#X\} \vdash c.X \preceq X$.

This example naturally leads to a question: Under which additional conditions can we have $\Gamma_2 \vdash r_1 \simeq r_2$ instead of $\Gamma_2 \vdash r_1 \preceq r_2$ in Lemma 6.4? To formalize a possible answer to it, we need some notation.

Let the terms t, s and the freshness context ∇ be based on the same set of atoms A. The maximal subset of A, *fresh* for t, s, and ∇, denoted $\mathit{fresh}(A, t, s, \nabla)$, is defined as $A \setminus (\mathrm{Atoms}(t, s) \cup \mathrm{Atoms}(\nabla))$.

If $A_1 \subseteq A_2$ are two sets of atoms such that t, s, ∇ are at the same time based on both A_1 and A_2, then $\mathit{fresh}(A_1, t, s, \nabla) \subseteq \mathit{fresh}(A_2, t, s, \nabla)$.

Let $\|t\|_{\mathsf{Abs}}$ stand for the number of abstraction occurrences in t. $|A|$ stands for the cardinality of the set of atoms A. We say that a set of atoms A is *saturated* for A-based t, s and ∇, if $|\mathit{fresh}(A, t, s, \nabla)| \geq \min\{\|t\|_{\mathsf{Abs}}, \|s\|_{\mathsf{Abs}}\}$.

The following lemma answers the question posed above:

▶ **Lemma 6.6.** *Under the conditions of Lemma 6.4, if* A_1 *is saturated for* t, s, ∇*, then* $\Gamma_2 \vdash r_1 \simeq r_2$.

Proof. Let D_i be the derivation in $\mathfrak{N}$ that computes $\langle \Gamma_i, r_i \rangle$, $i = 1, 2$. Note that in each of these derivations, the number of Abs steps does not exceed $\min\{\|t\|_{\mathsf{Abs}}, \|s\|_{\mathsf{Abs}}\}$. Since A_1 is saturated for t, s, ∇ and $A_1 \subseteq A_2$, A_2 is also saturated for t, s, ∇. Hence, whenever an AUT between two abstractions is encountered in the derivation D_i, there is always $c \in A_1$ available which satisfies the condition of the Abs rule. Therefore, such AU-E's are never transformed by Sol. We can assume without loss of generality that the sequence of steps in D_1 and D_2 are the same. we may also assume that we take the same fresh variables, and the same atoms from $\mathit{fresh}(A_1, t, s, \nabla)$ in the corresponding steps in D_1 and D_2. Then the only difference between these derivations is in the Γ's, caused by the Sol rule which might eventually make Γ_2 larger than Γ_1. The σ's computed by the derivations are the same and, therefore, r_1 and r_2 are the same (modulo the assumptions on the variable and fresh atom names). Hence, $\Gamma_2 \vdash r_1 \simeq r_2$. ◀

In other words, this lemma answers the following pragmatic question: Given t, s and ∇, how to choose a set of atoms A so that (a) t, s, ∇ are A-based and (b) in the A-based lgg $\langle \Gamma, r \rangle$ of $\langle \nabla, t \rangle$ and $\langle \nabla, s \rangle$, the term r generalizes s and t in the "best way", maximally preserving similarities and uniformly abstracting differences between s and t. The answer is: Besides all the atoms occurring in t, s, or ∇, A should contain at least m more atoms, where $m = \min\{\|t\|_{\mathsf{Abs}}, \|s\|_{\mathsf{Abs}}\}$.

Besides that, the lemma also gives the condition when the NAU-to-HOPAU translation can be used for solving NAU problems: The set of permitted atoms should be saturated.

7 Deciding Equivariance

Computation of π in the condition of the rule Mer above requires an algorithm that solves the following problem: Given nominal terms t, s and a freshness context ∇, find an Atoms(t, s)-based permutation π such that $\nabla \vdash \pi \bullet t \approx s$. This is the problem of deciding whether t and s are equivariant with respect to ∇. In this Section we describe a rule-based algorithm for this problem, called $\mathfrak{E}$.

Note that our problem differs from the problem of equivariant unification considered in [9]: We do not solve unification problems, since we do not allow variable substitution. We only look for permutations to *decide equivariance constructively* and provide a dedicated algorithm for that.

The algorithm $\mathfrak{E}$ works on tuples of the form $E;\ \nabla;\ A;\ \pi$ (also called states). E is a set of equivariance equations of the form $t \approx s$ where t, s are nominal terms, ∇ is a freshness context, and A is a finite set of atoms which are available for computing π. The latter holds the permutation to be returned in case of success.

The algorithm is split into two phases. The first one is a simplification phase where function applications, abstractions and suspensions are decomposed as long as possible. The second phase is the permutation computation, where given a set of equivariance equations between atoms of the form $a \approx b$ we compute the permutation which will be returned in case of success. The rules of the first phase are the following:

Dec-E: **Decomposition**

$$\{f(t_1, \ldots, t_m) \approx f(s_1, \ldots, s_m)\} \cup E;\ \nabla;\ A;\ Id \Longrightarrow \{t_1 \approx s_1, \ldots, t_m \approx s_m\} \cup E; \nabla; A; Id.$$

Alp-E: **Alpha Equivalence**

$\{a.t \approx b.s\} \uplus E;\ \nabla;\ A;\ Id \Longrightarrow \{(\acute{c}\ a) \bullet t \approx (\acute{c}\ b) \bullet s\} \cup E;\ \nabla;\ A;\ Id,$

where $\acute{c}$ is a fresh atom of the same sort as a and b.

Sus-E: **Suspension**

$\{\pi_1 \cdot X \approx \pi_2 \cdot X\} \uplus E;\ \nabla;\ A;\ Id \Longrightarrow \{\pi_1 \bullet a \approx \pi_2 \bullet a \mid a \in A \wedge a\#X \notin \nabla\} \cup E;\ \nabla;\ A;\ Id.$

The rules of the second phase are the following:

Rem-E: **Remove**

$\{a \approx b\} \uplus E;\ \nabla;\ A;\ \pi \Longrightarrow E;\ \nabla;\ A \setminus \{b\};\ \pi, \qquad \text{if } \pi \bullet a = b.$

Sol-E: **Solve**

$\{a \approx b\} \uplus E;\ \nabla;\ A;\ \pi \Longrightarrow E;\ \nabla;\ A \setminus \{b\};\ (\pi \bullet a\ \ b)\pi, \qquad \text{if } \pi \bullet a,\ b \in A \text{ and } \pi \bullet a \neq b.$

Note that in Alp-E, $\acute{c}$ is fresh means that $\acute{c} \notin A$ and, therefore, $\acute{c}$ will not appear in π. These atoms are an auxiliary means which play a role during the computation but do not appear in the final result.

Given nominal terms t, s, freshness context ∇, we construct a state $\{t \approx s\};\ \nabla;\ \text{Atoms}(t, s);\ Id$. We will prove that when the rules transform this state into $\emptyset;\ \nabla;\ A;\ \pi$, then π is an $\text{Atoms}(t, s)$-based permutation such that $\nabla \vdash \pi \bullet t \approx s$. When no rule is applicable, and the set of equations is not empty, we will also prove that there is no solution, hence we fail and return $\perp$.

▶ **Example 7.1.** We illustrate the algorithm $\mathfrak{E}$ on examples:

- Consider the equivariance problem $E = \{a \approx a,\ a.(a\,b)(c\,d) \cdot X \approx b.X\}$ and $\nabla = \{a\#X\}$:

$$\begin{aligned}
&\{a \approx a, a.(a\,b)(c\ d) \cdot X \approx b.X\}; \{a\#X\};\ \{a, b, c, d\};\ Id \Longrightarrow_{\text{Alp-E}}\\
&\{a \approx a, (\acute{e}\ a)(a\,b)(c\ d) \cdot X \approx (\acute{e}\ b) \cdot X\}; \{a\#X\};\ \{a, b, c, d\};\ Id \Longrightarrow_{\text{Sus-E}}\\
&\{a \approx a, \acute{e} \approx \acute{e}, c \approx d, d \approx c\}; \{a\#X\};\ \{a, b, c, d\};\ Id \Longrightarrow_{\text{Rem-E}}\\
&\{\acute{e} \approx \acute{e}, c \approx d, d \approx c\}; \{a\#X\};\ \{b, c, d\};\ Id \Longrightarrow_{\text{Rem-E}}\\
&\{c \approx d, d \approx c\}; \{a\#X\};\ \{b, c, d\};\ Id \Longrightarrow_{\text{Sol-E}}\\
&\{d \approx c\};\ \{a\#X\};\ \{b, c\};\ (c\,d) \Longrightarrow_{\text{Rem-E}}\\
&\emptyset;\ \{a\#X\};\ \{b\};\ (c\,d).
\end{aligned}$$

- For $E = \{a.f(b, X) \approx b.f(a, X)\}$ and $\nabla = \{a\#X\}$, $\mathfrak{E}$ returns $\perp$.
- For $E = \{a.f(b, (a\,b) \cdot X) \approx b.f(a, X)\}$ and $\nabla = \{a\#X\}$, $\mathfrak{E}$ returns $(b\,a)$.
- For $E = \{a.b.(a\,b)(a\,c) \cdot X = b.a.(a\,c) \cdot X\}$ and $\nabla = \emptyset$, $\mathfrak{E}$ returns Id.
- For $E = \{a.b.(a\,b)(a\,c) \cdot X = a.b.(b\,c) \cdot X\}$ and $\nabla = \emptyset$, $\mathfrak{E}$ returns $\perp$.

The Soundness Theorem for $\mathfrak{E}$ states that, indeed, the permutation the algorithm computes shows that the input terms are equivariant:

▶ **Theorem 7.2** (Soundness of $\mathfrak{E}$). *Let $\{t \approx s\};\ \nabla;\ A;\ Id \Longrightarrow^* \emptyset;\ \nabla;\ B;\ \pi$ be a derivation in $\mathfrak{E}$, then π is an A-based permutation such that $\nabla \vdash \pi \bullet t \approx s$.*

We now prove an invariant lemma that is used in the proof of completeness Theorem 7.4.

▶ **Lemma 7.3** (Invariant Lemma). *Let A be a finite set of atoms, E_1 be a set of equivariance equations for terms based on A, π_1 be an A-based permutation and $A_1 \subseteq A$. Let $E_1; \nabla; A_1; \pi_1 \Longrightarrow E_2; \nabla; A_2; \pi_2$ be any step performed by a rule in $\mathfrak{E}$. Let $\Gamma = \{\acute{c}\#X \mid X \in \mathrm{Vars}(E_1), \acute{c}$ is a fresh variable$\}$. Let μ be an A-based permutation such that $\nabla \cup \Gamma \vdash \mu \bullet t \approx s$, for all $t \approx s \in E_1$. Then*
1. *$\nabla \cup \Gamma \vdash \mu \bullet t' \approx s'$, for all $t' \approx s' \in E_2$.*
2. *If $\mu^{-1} \bullet b = \pi_1^{-1} \bullet b$, for all $b \in A \setminus A_1$, then $\mu^{-1} \bullet b = \pi_2^{-1} \bullet b$, for all $b \in A \setminus A_2$.*

Proof. By case distinction on the applied rule.

Dec-E: The proposition is obvious.

Alp-E: In this case it follows from the definitions of $\approx$ and permutation application.

Sus-E: In this case $t = \tau_1 \cdot X$, $s = \tau_2 \cdot X$, and by the assumption we have $\nabla \vdash \mu\tau_1 \cdot X \approx \tau_2 \cdot X$. By the definition of $\approx$, it means that we have $a\#X \in \nabla$, for all atoms a such that $\mu\tau_1 \bullet a \neq \tau_2 \bullet a$. Hence, for all $a \in A$ with $a\#X \notin \nabla$ we have $\nabla \vdash \mu\tau_1 \bullet a \approx \tau_2 \bullet a$. This implies that μ also solves the equations in E_2, hence item 1 of the lemma.

Item 2 of the lemma is trivial for these three rules, since $A_1 = A_2$ and $\pi_1 = \pi_2 = Id$.

Rem-E: The item 1 is trivial. To prove the item 2, note that $t = a$, $s = b$, $\pi_1 = \pi_2$ and we only need to show $\mu^{-1} \bullet b = \pi_2^{-1} \bullet b$. By the assumption we have $\nabla \vdash \mu \bullet a \approx b$. Since a and b are atoms, the latter simply means that $\mu \bullet a = b$. From the rule condition we also know that $\pi_1 \bullet a = b$. From these two equalities we get $\mu^{-1} \bullet b = a = \pi_2^{-1} b$.

Sol-E: The item 1 is trivial also in this case. To prove the item 2, note that $t = a$, $s = b$, $\pi_2 = (\pi_1 \bullet a\ b)\pi_1$ and we only need to show $\mu^{-1} \bullet b = \pi_2^{-1} \bullet b$. By the assumption we have $\nabla \vdash \mu \bullet a \approx b$, which means that $\mu \bullet a = b$ and, hence, $a = \mu^{-1} \bullet b$. As for $\pi_2^{-1} \bullet b$, we have $\pi_2^{-1} \bullet b = \pi_1^{-1}(\pi_1 \bullet a\ b) \bullet b = \pi_1^{-1} \bullet (\pi_1 \bullet a) = a$. Hence, we get $\mu^{-1} \bullet b = a = \pi_2^{-1} \bullet b$. ◀

▶ **Theorem 7.4** (Completeness of $\mathfrak{E}$). *Let A be a finite set of atoms, t, s be A-based terms, and ∇ be a freshness context. If $\nabla \vdash \mu \bullet t \approx s$ holds for some A-based permutation μ, then there exists a derivation $\{t \approx s\}; \nabla; A; Id \Longrightarrow^* \emptyset; \Gamma; B; \pi$, obtained by an execution of $\mathfrak{E}$, such that $\pi \bullet a = \mu \bullet a$ for any atom $a \in \mathrm{FA}(t)$.*

8 Complexity Analysis

We represent a permutations π as two hash tables. One for the permutation itself, we call it T_π, and one for the inverse of the permutation, called $T_{\pi^{-1}}$. The key of a hash tables is an atom and we associate another atom, the mapping, with it. For instance the permutation $\pi = (a\,b)(a\,c)$ is represented as $T_\pi = \{a \mapsto c, b \mapsto a, c \mapsto b\}$ and $T_{\pi^{-1}} = \{a \mapsto b, b \mapsto c, c \mapsto a\}$. We write $T_\pi(a)$ to obtain from the hash table T_π the atom which is associated with the key a. If no atom is associated with the key a then $T_\pi(a)$ returns a. We write $T_\pi(a \mapsto b)$, to set the mapping such that $T_\pi(a) = b$. As the set of atoms is small, we can assume a perfect hash function. It follows, that both defined operations are done in constant time, leading to constant time application of a permutation. Swapping application to a permutation $(a\,b)\pi$ is also done in constant time in the following way: Obtain $c = T_{\pi^{-1}}(a)$ and $d = T_{\pi^{-1}}(b)$ and perform the following updates:

(a) $T_\pi(c \mapsto b)$ and $T_\pi(d \mapsto a)$,

(b) $T_{\pi^{-1}}(b \mapsto c)$ and $T_{\pi^{-1}}(a \mapsto d)$.

We also represent set membership of atoms to a set of atoms A with a hash table $\in_A$ from atoms to Booleans such that $\in_A (a) = true$ iff $a \in A$. We also have a list L_A of the atoms representing the entries of the table such that $\in_A (a) = true$ to easily know all atoms in A.

Finally we also represent set membership of freshness constraints to a freshness environment ∇ with a hash table $\in_\nabla$.

▶ **Theorem 8.1.** *Given a set of equivariance equations E, and a freshness context ∇. Let m be the size of ∇, and let n be the size of E. The algorithm $\mathfrak{E}$ has $O(n^2 + m)$ time complexity.*

Proof. Collecting the atoms from E in a separate set A does not affect the space complexity and can be done in time $O(n)$. The freshness environment ∇ will not be modified by rule applications and membership test in the rule Sus-E can be done in constant time. We only have to construct the corresponding hash tables in time $O(m)$. We analyze complexity of both phases.

For the first phase, notice that all rules can be applied only $O(n)$ many times, since Dec-E removes two function symbols and Alp-E two abstraction, and Sus-E two suspensions. The resulting equations after this phase only contain atoms. However, notice that the size of these equations is not necessarily linear. Every time we apply Alp-E a new swapping is applied to both subterms. This swappings may increase the size of suspensions occurring bellow the abstraction. Since there are $O(n)$ many suspensions and $O(n)$ many abstractions, the final size of suspensions is $O(n^2)$. This is the size of the atom equations at the beginning of the second phase. We can see that the application of Dec-E rule has $O(1)$ time complexity (with the appropriate representation of equations).

The application of Alp-E rule requires to find a fresh atom not in A, this can be done in constant time. Later, a swapping has to be applied twice. Swapping application requires traversing the term hence has $O(n)$ time complexity. The application of Sus-E requires to traverse L_A ($O(n)$) and check for freshness membership in $\in_\nabla$ ($O(1)$). Finally it has to add equations like $(\pi_1 \bullet a \approx \pi_2)$, this requires to build T_{π_1} and T_{π_2} that can be done in $O(n)$ time complexity and allow us to build each equation in $O(1)$ time. Summing up, this phase has $O(n^2)$ time complexity.

For the second phase, notice that both rules Rem-E and Sol-E remove an equation and do not introduce any other one. Hence, potentially having $O(n^2)$ many equations in this phase, these equations can be applied $O(n^2)$ may times. We construct a hash table $T_{\boldsymbol{\pi}}$ for $\boldsymbol{\pi}$ that will be maintained and used by both rules. Each application has time complexity $O(1)$. Rem-E uses $T_{\boldsymbol{\pi}}$ to check for applicability and if it is applied, it only removes b from A, hence updating $\in_A$ (notice that we do not care about L_A in this second phase of the algorithm). Sol-E uses $\in_A$ and $T_{\boldsymbol{\pi}}$ to check for applicability and if it is applied, it only removes b from A (hence updating $\in_A$), and updates $T_{\boldsymbol{\pi}}$. Summing up, this phase maintains the overall $O(n^2)$ time complexity. ◀

▶ **Theorem 8.2.** *The nominal anti-unification algorithm $\mathfrak{N}$ has $O(n^5)$ time complexity and $O(n^4)$ space complexity, where n is the input size.*

Proof. By design of the rules and theorem 6.3 we can arrange a maximal derivation like $\{X_0 : t_0 \triangleq s_0\}; \emptyset; \emptyset; \varepsilon \Longrightarrow^*_{\mathsf{Dec,Abs,Sol}} \emptyset; S_l; \Gamma_l; \sigma_l \Longrightarrow^*_{\mathsf{Mer}} \emptyset; S_m; \Gamma_m; \sigma_m$, postponing the application of Mer until the end. Rules Dec, Abs and Sol can be applied $O(n)$ many times. However, notice that every application of Abs may increase the size of every suspension below. Hence, the size of the store S_l is $O(n^2)$, although it only contain $O(n)$ equations, after an exhaustive derivation $\{X_0 : t_0 \triangleq s_0\}; \emptyset; \emptyset; \varepsilon \Longrightarrow^*_{\mathsf{Dec,Abs,Sol}} \emptyset; S_l; \Gamma_l; \sigma_l$.

Now we turn to analyzing the transformation phase $\emptyset; S_l; \Gamma_l; \sigma_l \Longrightarrow^*_{\mathsf{Mer}} \emptyset; S_m; \Gamma_m; \sigma_m$. Let $S_l = \{X_1 : t_1 \triangleq s_1, \ldots, X_k : t_k \triangleq s_k\}$ and n_i be the size of $X_i : t_i \triangleq s_i$, $1 \leq i \leq k$, then $\sum_{i=1}^{k} n_i = O(n^2)$ and $k = O(n)$. From theorem 8.1 we know that solving the equivariance problem for two AUPs $X_i : t_i \triangleq s_i$ and $X_j : t_j \triangleq s_j$ and an arbitrary freshness context ∇ requires $O((n_i + n_j)^2 + m)$ time and space, where m is the size of ∇ with $m = O(n)$.

Merging requires to solve this problem for each pair of AUPs. This leads to the time complexity $\sum_{i=1}^{k}\sum_{j=i+1}^{k} O((n_i+n_j)^2+m) \leq O(\sum_{i=1}^{k}\sum_{j=1}^{k}(n_i+n_j)^2)+O(\sum_{i=1}^{k}\sum_{j=1}^{k} m)$. The second sum is $\sum_{i=1}^{k}\sum_{j=1}^{k} m = k^2m = O(n^3)$. Now we estimate an upper bound for the sum $\sum_{i=1}^{k}\sum_{j=1}^{k}(n_i+n_j)^2 = \sum_{i=1}^{k}\sum_{j=1}^{k} n_i^2 + \sum_{i=1}^{k}\sum_{j=1}^{k} 2n_in_j + \sum_{i=1}^{k}\sum_{j=1}^{k} n_j^2 \leq \sum_{i=1}^{k} kn_i^2 + 2\left(\sum_{i=1}^{k} n_i\right)\left(\sum_{j=1}^{k} n_j\right) + \sum_{i=1}^{k}(\sum_{j=1}^{k} n_j)^2 \leq k(\sum_{i=1}^{k} n_i)^2 + 2O(n^2)O(n^2) + \sum_{i=1}^{k} O(n^2) = kO(n^2)^2 + 2O(n^2)^2 + kO(n^2)^2 = O(n^5)$, resulting into the stated bounds.

The space is bounded by the space required by a single call to the equivariance algorithm with an imput of size $O(n^2)$, hence $O(n^4)$. ◀

9 Conclusion

The problem of anti-unification for nominal terms-in-context is sensitive to the set of atoms permitted in generalizations: If this set is infinite, there is no least general generalization. Otherwise there exists a unique lgg. If this set is finite and satisfies the notion of being saturated, defined in the paper, then the lgg retains the common structure of the input nominal terms maximally.

We illustrated that, similar to some other theories where unification, generalization, and the subsumption relation are defined, the nominal terms-in-contexts form a join-meet lattice with respect to the subsumption relation, where the existence of join is unifiability, and the meet corresponds to least general generalization.

We designed an anti-unification algorithm for nominal terms-in-context. It contains a subalgorithm that constructively decides whether two terms are equivariant with respect to the given freshness context. We proved termination, soundness, and completeness of these algorithms, investigated their complexities, and implemented them. Given a fixed set of atoms A, the nominal anti-unification algorithm computes a least general A-based term-in-context generalization of the given A-based terms-in-context, and requires $O(n^5)$ time and $O(n^4)$ space for that, where n is the size of the input. The computed lgg is unique modulo α-equivalence and variable renaming.

References

1. María Alpuente, Santiago Escobar, José Meseguer, and Pedro Ojeda. A modular equational generalization algorithm. In Michael Hanus, editor, *LOPSTR*, volume 5438 of *LNCS*, pages 24–39. Springer, 2008.
2. Alexander Baumgartner and Temur Kutsia. A library of anti-unification algorithms. In Eduardo Fermé and João Leite, editors, *Logics in Artificial Intelligence - 14th European Conference, JELIA 2014, Funchal, Madeira, Portugal, September 24-26, 2014. Proceedings*, volume 8761 of *Lecture Notes in Computer Science*, pages 543–557. Springer, 2014.
3. Alexander Baumgartner, Temur Kutsia, Jordi Levy, and Mateu Villaret. A variant of higher-order anti-unification. In Femke van Raamsdonk, editor, *RTA*, volume 21 of *LIPIcs*, pages 113–127. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2013.
4. Alexander Baumgartner, Temur Kutsia, Jordi Levy, and Mateu Villaret. Nominal Anti-Unification. Technical Report 15-03, RISC, JKU Linz, April 2015.
5. Christophe Calvès. *Complexity and Implementation of Nominal Algorithms*. PhD thesis, King's College London, 2010.
6. Christophe Calvès and Maribel Fernández. A polynomial nominal unification algorithm. *Theor. Comput. Sci.*, 403(2-3):285–306, 2008.
7. Christophe Calvès and Maribel Fernández. Matching and alpha-equivalence check for nominal terms. *J. Comput. Syst. Sci.*, 76(5):283–301, 2010.

8 James Cheney. Relating nominal and higher-order pattern unification. In Laurent Vigneron, editor, *UNIF'05*, LORIA A05-R-022, pages 105–119, 2005.
9 James Cheney. Equivariant unification. *JAR*, 45(3):267–300, 2010.
10 Gilles Dowek, Murdoch J. Gabbay, and Dominic P. Mulligan. Permissive nominal terms and their unification. In *24th Italian Conference on Computational Logic, CILC'09*, 2009.
11 Gilles Dowek, Murdoch James Gabbay, and Dominic P. Mulligan. Permissive nominal terms and their unification: an infinite, co-infinite approach to nominal techniques. *Logic Journal of the IGPL*, 18(6):769–822, 2010.
12 Cao Feng and Stephen Muggleton. Towards inductive generalization in higher order logic. In Derek H. Sleeman and Peter Edwards, editors, *ML*, pages 154–162. Morgan Kaufmann, 1992.
13 Murdoch Gabbay and Andrew M. Pitts. A new approach to abstract syntax with variable binding. *Formal Asp. Comput.*, 13(3-5):341–363, 2002.
14 Murdoch J. Gabbay. *A Theory of Inductive Definitions with alpha-Equivalence*. PhD thesis, University of Cambridge, UK, 2000.
15 Murdoch J. Gabbay and Andrew M. Pitts. A new approach to abstract syntax involving binders. In *LICS*, pages 214–224. IEEE Computer Society, 1999.
16 Ulf Krumnack, Angela Schwering, Helmar Gust, and Kai-Uwe Kühnberger. Restricted higher-order anti-unification for analogy making. In Mehmet A. Orgun and John Thornton, editors, *Australian Conference on Artificial Intelligence*, volume 4830 of *LNCS*, pages 273–282. Springer, 2007.
17 Temur Kutsia, Jordi Levy, and Mateu Villaret. Anti-unification for unranked terms and hedges. In Manfred Schmidt-Schauß, editor, *RTA*, volume 10 of *LIPIcs*, pages 219–234. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2011.
18 Jordi Levy and Mateu Villaret. Nominal unification from a higher-order perspective. In Andrei Voronkov, editor, *RTA*, volume 5117 of *LNCS*, pages 246–260. Springer, 2008.
19 Jordi Levy and Mateu Villaret. An efficient nominal unification algorithm. In Christopher Lynch, editor, *Proceedings of the 21st International Conference on Rewriting Techniques and Applications, RTA 2010, July 11-13, 2010, Edinburgh, Scottland, UK*, volume 6 of *LIPIcs*, pages 209–226. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2010.
20 Jordi Levy and Mateu Villaret. Nominal unification from a higher-order perspective. *ACM Trans. Comput. Log.*, 13(2):10, 2012.
21 Jianguo Lu, John Mylopoulos, Masateru Harao, and Masami Hagiya. Higher order generalization and its application in program verification. *Ann. Math. Artif. Intell.*, 28(1-4):107–126, 2000.
22 Frank Pfenning. Unification and anti-unification in the calculus of constructions. In *LICS*, pages 74–85. IEEE Computer Society, 1991.
23 Gordon D. Plotkin. A note on inductive generalization. *Machine Intel.*, 5(1):153–163, 1970.
24 Luc De Raedt. *Logical and Relational Learning*. Springer, 2008.
25 John C. Reynolds. Transformational systems and the algebraic structure of atomic formulas. *Machine Intel.*, 5(1):135–151, 1970.
26 Ute Schmid. *Inductive Synthesis of Functional Programs, Universal Planning, Folding of Finite Programs, and Schema Abstraction by Analogical Reasoning*, volume 2654 of *LNCS*. Springer, 2003.
27 Christian Urban. Nominal techniques in Isabelle/HOL. *J. Autom. Reasoning*, 40(4):327–356, 2008.
28 Christian Urban and James Cheney. Avoiding equivariance in alpha-prolog. In Paweł Urzyczyn, editor, *Typed Lambda Calculi and Applications*, volume 3461 of *LNCS*, pages 401–416. Springer Berlin Heidelberg, 2005.

29 Christian Urban, Andrew M. Pitts, and Murdoch J. Gabbay. Nominal unification. In Matthias Baaz and Johann A. Makowsky, editors, *CSL*, volume 2803 of *LNCS*, pages 513–527. Springer, 2003.

30 Christian Urban, Andrew M. Pitts, and Murdoch J. Gabbay. Nominal unification. *Theor. Comput. Sci.*, 323(1–3):473–497, 2004.

A faithful encoding of programmable strategies into term rewriting systems

Horatiu Cirstea, Sergueï Lenglet, and Pierre-Etienne Moreau

Université de Lorraine – LORIA
Campus scientifique – BP 239 – 54506 Vandœuvre-lès-Nancy, France
{firstname.lastname@loria.fr}

Abstract

Rewriting is a formalism widely used in computer science and mathematical logic. When using rewriting as a programming or modeling paradigm, the rewrite rules describe the transformations one wants to operate and declarative rewriting strategies are used to control their application. The operational semantics of these strategies are generally accepted and approaches for analyzing the termination of specific strategies have been studied. We propose in this paper a generic encoding of classic control and traversal strategies used in rewrite based languages such as Maude, Stratego and Tom into a plain term rewriting system. The encoding is proven sound and complete and, as a direct consequence, established termination methods used for term rewriting systems can be applied to analyze the termination of strategy controlled term rewriting systems. The corresponding implementation in Tom generates term rewriting systems compatible with the syntax of termination tools such as AProVE and TTT2, tools which turned out to be very effective in (dis)proving the termination of the generated term rewriting systems. The approach can also be seen as a generic strategy compiler which can be integrated into languages providing pattern matching primitives; this has been experimented for Tom and performances comparable to the native Tom strategies have been observed.

1998 ACM Subject Classification F.4 Mathematical Logic and Formal Languages

Keywords and phrases Programmable strategies, termination, term rewriting systems

Digital Object Identifier 10.4230/LIPIcs.RTA.2015.74

1 Introduction

Rewriting is a very powerful tool used in theoretical studies as well as for practical implementations. It is used, for example, in semantics in order to describe the meaning of programming languages [24], but also in automated reasoning when describing by inference rules a logic, a theorem prover or a constraint solver [20]. It is also used to compute in systems making the notion of rule an explicit and first class object, like Mathematica, Maude [8], or Tom [25, 4]. Rewrite rules, the core concept in rewriting, consist of a pattern that describes a schematic situation and the transformation that should be applied in that particular case. The pattern expresses a potentially infinite number of instances and the application of the rewrite rule is decided locally using a (matching) algorithm which only depends on the pattern and its subject. Rewrite rules are thus very convenient for describing schematically and locally the transformations one wants to operate.

In many situations, the application of a set of rewrite rules to a subject eventually leads to the same final result independently on the way the rules are applied, and in such cases we say that the rewrite rules are *confluent* and *terminating*. When using rewriting as a programming or modeling paradigm it is nevertheless common to consider term rewriting systems (TRS) that are non-confluent or non-terminating. In order to make the concept operational when

© Horatiu Cirstea, Sergueï Lenglet, and Pierre-Etienne Moreau;
licensed under Creative Commons License CC-BY
26th International Conference on Rewriting Techniques and Applications (RTA'15).
Editor: Maribel Fernández; pp. 74–88
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

such rules are employed we need an additional ingredient allowing one to specify which rules should be applied and where (in the subject). This fundamental mechanism allowing the control of rule application is a *rewriting strategy*.

Rule-based languages like ELAN [6], Maude, Stratego [30] or Tom have introduced the notion of programmable strategies to express rule application control in a declarative way. All these languages provide means to define term representations for the objects we want to transform as well as rewrite rules and strategies expressing the way the terms are transformed and offer thus, a clear separation between data structures, function logic and control. Similarly to plain TRS (*i.e.* TRS without strategy), it is interesting to guarantee that the strategy controlled TRS enjoy properties such as confluence and termination. Confluence holds as long as the rewrite rules are deterministic (*i.e.* the corresponding matching is unitary) and all strategy operators are deterministic (or a deterministic implementation is provided).

Termination is more delicate and the normalization under some specific strategy is usually guaranteed by imposing (sufficient) conditions on the corresponding set of rewrite rules. Such conditions have been proposed for the innermost [2, 13, 29, 17], outermost [9, 28, 17, 26], context-sensitive [14, 1, 17], or lazy [15] reduction strategies. Termination under programmable strategies has been studied for ELAN [11] and Stratego [21, 23]. In [11], the authors prove that a programmable strategy is terminating if the system formed with all the rewrite rules the strategy contains is terminating. This criterion is too coarse as it does not take into account how the strategy makes its arguments interact and consequently, the approach cannot be used to prove termination for many terminating strategies. In [21, 23], the termination of some traversal strategies (such as top-down, bottom-up, innermost) is proven, assuming the rewrite rules are measure decreasing, for a notion of measure that combines the depth, and the number of occurrences of a specific constructor in a term.

Contributions. In this paper we propose a more general approach consisting in translating programmable strategies into plain TRS. The interest of this encoding that we show sound and complete is twofold. First, termination analysis techniques [2, 19, 16] and corresponding tools like AProVE [12] and TTT2 [22] that have been successfully used for checking the termination of plain TRS can be used to verify termination in presence of rewriting strategies. Second, the translation can be seen as a generic strategy compiler and thus can be used as a portable implementation of strategies which could be easily integrated in any language providing rewrite rules (or at least pattern matching) primitives. The translation has been implemented in Tom and generates TRS which could be fed into TTT2/AProVE for termination analysis or executed efficiently by Tom.

The paper is organized as follows. The next section introduces the notions of rewriting system and rewriting strategy. Section 3 presents the translation of rewriting strategies into rewriting systems, and its properties are stated together with proof sketches in Section 4. In Section 5 we give some implementation details and present experimental results. We end with conclusions and further work.

2 Strategic rewriting

This section briefly recalls some basic notions of rewriting used in this paper; see [3, 27] for more details on first order terms and term rewriting systems, and [31, 5] for details on rewriting strategies and their implementation in rewrite based languages.

2.1 Term algebra and term rewriting systems

A *signature* Σ consists of an alphabet $\mathcal{F}$ of symbols together with a function *ar* which associates to any symbol f its *arity*. We write $\mathcal{F}^n$ for the subset of symbols of arity n, and $\mathcal{F}^+$ for the symbols of arity $n > 0$. Symbols in $\mathcal{F}^0$ are called *constants*. Given a countable set $\mathcal{X}$ of *variable* symbols, the set of *first-order terms* $\mathcal{T}(\mathcal{F}, \mathcal{X})$ is the smallest set containing $\mathcal{X}$ and such that $f(t_1, \ldots, t_n)$ is in $\mathcal{T}(\mathcal{F}, \mathcal{X})$ whenever $f \in \mathcal{F}^n$ and $t_i \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ for $i \in [1, n]$. We write $\mathcal{V}ar\,(t)$ for the set of variables occuring in $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$. If $\mathcal{V}ar\,(t)$ is empty, t is called a *ground* term; $\mathcal{T}_{\mathcal{F}}$ denotes the set of all ground terms. A *linear* term is a term where every variable occurs at most once. A *substitution* σ is a mapping from $\mathcal{X}$ to $\mathcal{T}(\mathcal{F}, \mathcal{X})$ which is the identity except over a finite set of variables (its *domain*). A substitution extends as expected to an endomorphism of $\mathcal{T}(\mathcal{F}, \mathcal{X})$.

A *position* of a term t is a finite sequence of positive integers describing the path from the root of t to the root of the sub-term at that position. We write ε for the empty sequence, which represents the root of t, $\mathcal{P}os(t)$ for the set of positions of t, and $t_{|\omega}$ for the sub-term of t at position ω. Finally, $t\,[s]_\omega$ is the term t with the sub-term at position ω replaced by s.

A *rewrite rule* (over Σ) is a pair $(l, r) \in \mathcal{T}(\mathcal{F}, \mathcal{X}) \times \mathcal{T}(\mathcal{F}, \mathcal{X})$ (also denoted $l \twoheadrightarrow r$) such that $\mathcal{V}ar\,(r) \subseteq \mathcal{V}ar\,(l)$ and a TRS is a set of rewrite rules $\mathcal{R}$ inducing a *rewriting relation* over $\mathcal{T}_{\mathcal{F}}$, denoted by $\longrightarrow_{\mathcal{R}}$ and such that $t \longrightarrow_{\mathcal{R}} t'$ iff there exist $l \twoheadrightarrow r \in \mathcal{R}$, $\omega \in \mathcal{P}os(t)$, and a substitution σ such that $t_{|\omega} = \sigma(l)$ and $t' = t\,[\sigma(r)]_\omega$. In this case, we say that l matches t and that σ is the solution of the corresponding matching problem. The reflexive and transitive closure of $\longrightarrow_{\mathcal{R}}$ is denoted by $\twoheadrightarrow_{\mathcal{R}}$. In what follows, we generally use the notation $\mathcal{R} \bullet t \longrightarrow t'$ to denote $t \twoheadrightarrow_{\mathcal{R}} t'$. A TRS $\mathcal{R}$ is *terminating* if there exists no infinite rewriting sequence $t_1 \longrightarrow_{\mathcal{R}} t_2 \longrightarrow_{\mathcal{R}} \ldots \longrightarrow_{\mathcal{R}} t_n \longrightarrow_{\mathcal{R}} \ldots$

2.2 Rewriting strategies

Rewriting strategies allow one to specify how rules should be applied. We call the term to which the strategy is applied the subject. The application of a strategy to a subject may diverge, fail, or return a (unique) result.

Taking the same terminology as the one proposed by ELAN and Stratego, a rewrite rule is considered to be an elementary strategy, and a strategy is an expression built over a strategy language. We consider a strategy language over a signature Σ consisting of the main operators used in rewrite based languages like Tom and Stratego:

$$S \;:=\; \mathit{Identity} \mid \mathit{Fail} \mid l \twoheadrightarrow r \mid S\,;S \mid S \leftarrow\!\!\!+ S \mid \mathit{One}(S) \mid \mathit{All}(S) \mid \mu X\,.\,S \mid X$$

with X any variable from the set $\mathcal{X}_{\mathcal{S}}$ of strategy variables and $l \twoheadrightarrow r$ any rewrite rule over Σ. In what follows, we use uppercase for strategy variables and lowercase for term variables.

Before formally defining the strategy semantics, we can already mention that the simplest strategies we can imagine are *Identity* and *Fail*. The *Identity* strategy can be applied to any term without changing it, and thus *Identity* never fails. Conversely, the strategy *Fail* always fails when applied to a term. As mentioned above, a rewrite rule is an elementary strategy which is applied to the root position of its subject. By combining elementary strategies, more complex strategies can be built. In particular, we can apply sequentially two strategies, make a choice between the application of two strategies, apply a strategy to one or to all the immediate sub-terms of the subject, and apply recursively a strategy.

The operational semantics presented in Figure 1 is defined *w.r.t.* a context of the form $X_1\colon S_1 \ldots X_n\colon S_n$ which associates strategy expressions to strategy variables. Indeed, a reduction step is written $\Gamma \vdash S \circ t \Longrightarrow u$, where Γ is the context under which the current

Elementary strategies

$$\frac{}{\Gamma \vdash \mathit{Identity} \circ t \Longrightarrow t}\ (\mathbf{id}) \qquad \frac{}{\Gamma \vdash \mathit{Fail} \circ t \Longrightarrow \mathtt{Fail}}\ (\mathbf{fail})$$

$$\frac{\exists \sigma, \sigma(l) = t}{\Gamma \vdash l \twoheadrightarrow r \circ t \Longrightarrow \sigma(r)}\ (\mathbf{r_1}) \qquad \frac{\nexists \sigma, \sigma(l) = t}{\Gamma \vdash l \twoheadrightarrow r \circ t \Longrightarrow \mathtt{Fail}}\ (\mathbf{r_2})$$

Control combinators

$$\frac{\Gamma \vdash S_1 \circ t \Longrightarrow t'}{\Gamma \vdash (S_1 \nleftrightarrow S_2) \circ t \Longrightarrow t'}\ (\mathbf{choice_1}) \qquad \frac{\Gamma \vdash S_1 \circ t \Longrightarrow \mathtt{Fail} \qquad \Gamma \vdash S_2 \circ t \Longrightarrow u}{\Gamma \vdash (S_1 \nleftrightarrow S_2) \circ t \Longrightarrow u}\ (\mathbf{choice_2})$$

$$\frac{\Gamma \vdash S_1 \circ t \Longrightarrow t' \qquad \Gamma \vdash S_2 \circ t' \Longrightarrow u}{\Gamma \vdash (S_1 \,;S_2) \circ t \Longrightarrow u}\ (\mathbf{seq_1}) \qquad \frac{\Gamma \vdash S_1 \circ t \Longrightarrow \mathtt{Fail}}{\Gamma \vdash (S_1 \,;S_2) \circ t \Longrightarrow \mathtt{Fail}}\ (\mathbf{seq_2})$$

$$\frac{\Gamma; X : S \vdash S \circ t \Longrightarrow u}{\Gamma \vdash \mu X \,.\, S \circ t \Longrightarrow u}\ (\mathbf{mu}) \qquad \frac{\Gamma; X : S \vdash S \circ t \Longrightarrow u}{\Gamma; X : S \vdash X \circ t \Longrightarrow u}\ (\mathbf{muvar})$$

Traversal combinators

$$\frac{\exists i \in [1,n], \Gamma \vdash S \circ t_i \Longrightarrow t'_i \qquad \forall j \in [1, i-1], \Gamma \vdash S \circ t_j \Longrightarrow \mathtt{Fail}}{\Gamma \vdash \mathit{One}(S) \circ f(t_1, \ldots, t_n) \Longrightarrow f(t_1, \ldots, t_{i-1}, t'_i, t_{i+1}, \ldots, t_n)}\ (\mathbf{one_1})$$

$$\frac{\forall i \in [1,n], \Gamma \vdash S \circ t_i \Longrightarrow \mathtt{Fail}}{\Gamma \vdash \mathit{One}(S) \circ f(t_1, \ldots, t_n) \Longrightarrow \mathtt{Fail}}\ (\mathbf{one_2})$$

$$\frac{\forall i \in [1,n], \Gamma \vdash S \circ t_i \Longrightarrow t'_i}{\Gamma \vdash \mathit{All}(S) \circ f(t_1, \ldots, t_n) \Longrightarrow f(t'_1, \ldots, t'_n)}\ (\mathbf{all_1})$$

$$\frac{\exists i \in [1,n], \Gamma \vdash S \circ t_i \Longrightarrow \mathtt{Fail}}{\Gamma \vdash \mathit{All}(S) \circ f(t_1, \ldots, t_n) \Longrightarrow \mathtt{Fail}}\ (\mathbf{all_2})$$

Figure 1 Strategy semantics. The meta variable t denotes a term (which cannot be `Fail`), whereas the meta variable u denotes a result which can be either a well-formed term, or `Fail`.

strategy is applied, S is the strategy to apply, t is the subject, and u is the resulting term or `Fail` (the context may be omitted if empty). The variables in strategy expressions could be thus bound by the recursion operator or by a corresponding assignment in the context: we primarily use the context as an accumulator for the evaluation of recursion strategies, but it can also be used for the evaluation of strategies featuring free variables. As usual, we work modulo *α-conversion* and we adopt Barendregt's *"hygiene-convention"*, *i.e.* free- and bound-variables have different names.

We distinguish three kinds of operators in the strategy language:

- *elementary strategies* consisting of the *Identity* and *Fail* strategies, and rewrite rules which succeed or fail when applied at the root position of the subject.

- *control combinators* that compose strategies but are still applied at the root of the subject. The (left-)choice $S_1 \leftarrow\!\!\!\!+ S_2$ tries to apply S_1 and considers S_2 only if S_1 fails. The sequential application $S_1 \,;S_2$ succeeds if S_1 succeeds on the subject and S_2 succeeds on the subsequent term; it fails if one of the two strategy applications fails. The application of a strategy $\mu X \,.\, S$ (rule **mu**) evaluates S in a context where X is bound to S. If the strategy variable X is applied to a term (rule **muvar**), the strategy S is (re)evaluated, allowing thus recursion. This process can, of course, go on forever but eventually stops if at some point the evaluation of S does not involve X anymore.
- *traversal combinators* that modify the current application position. The operator $One(S)$ tries to apply S to a sub-term of the subject. We have chosen a deterministic semantics for One which looks for the left-most sub-term successfully transformed; the non-deterministic behavior can be easily obtained by removing the second condition in the premises of the inference rule $\mathbf{one_1}$. If S fails on all the sub-terms, then $One(S)$ also fails (rule $\mathbf{one_2}$). In contrast, $All(S)$ applies S to all the sub-terms of the subject (rule $\mathbf{all_1}$) and fails if S fails on one of them (rule $\mathbf{all_2}$). Note that $One(S)$ always fails when applied to a constant while $All(S)$ always succeeds in this case.

The combinators of the strategy language can be used to define more complex ones. For example, we can define a strategy named *Try*, parameterized by a strategy S, which tries to apply S and applies the identity if S fails: $Try(S) = S \leftarrow\!\!\!\!+ Identity$. The $Repeat(S)$ strategy can be defined using recursion: $Repeat(S) = \mu X \,.\, Try(S\,;X)$. In fact, most of the classic reduction strategies can also be defined using the generic traversal operators:

$$\begin{aligned} OnceBottomUp(S) &= \mu X \,.\, One(X) \leftarrow\!\!\!\!+ S \\ BottomUp(S) &= \mu X \,.\, All(X)\,; S \\ OnceTopDown(S) &= \mu X \,.\, S \leftarrow\!\!\!\!+ One(X) \\ TopDown(S) &= \mu X \,.\, S\,; All(X) \end{aligned}$$

The strategy *OnceBottomUp* (denoted *obu* in the following) tries to apply the strategy S once, starting from the leftmost-innermost leaves. *BottomUp* behaves almost like *obu* except that S is applied to all nodes, starting from the leaves. The strategy which applies S as many times as possible, starting from the leaves can be either defined naively as $Repeat(obu(S))$ or using a more efficient approach [31]: $Innermost(s) = \mu X \,.\, All(X)\,; Try(S\,; X)$. Given the rules $R_1, \ldots, R_n$, the strategy $R_1 \leftarrow\!\!\!\!+ \cdots \leftarrow\!\!\!\!+ R_n$ can be used to express an order on the rules.

▶ **Example 1.** Consider the rewrite rules $\mathtt{+}(\mathtt{Z}, x) \twoheadrightarrow x$ and $\mathtt{+}(\mathtt{S}(x), y) \twoheadrightarrow \mathtt{S}(\mathtt{+}(x, y))$ on the signature Σ with $\mathcal{F}^0 = \{\mathtt{Z}\}$, $\mathcal{F}^1 = \{\mathtt{S}\}$, $\mathcal{F}^2 = \{\mathtt{+}, \mathtt{*}\}$.

$\vdash \mathtt{+}(\mathtt{Z}, x) \twoheadrightarrow x \circ \mathtt{Z} \Longrightarrow \mathtt{Fail}$

$\vdash \mathtt{+}(\mathtt{Z}, x) \twoheadrightarrow x \circ \mathtt{+}(\mathtt{Z}, \mathtt{S}(\mathtt{Z})) \Longrightarrow \mathtt{S}(\mathtt{Z})$

$\vdash \mathtt{+}(\mathtt{Z}, x) \twoheadrightarrow x\,; \mathtt{+}(\mathtt{Z}, x) \twoheadrightarrow x \circ \mathtt{+}(\mathtt{Z}, \mathtt{+}(\mathtt{Z}, \mathtt{S}(\mathtt{Z}))) \Longrightarrow \mathtt{S}(\mathtt{Z})$

$\vdash One(\mathtt{+}(\mathtt{Z}, x) \twoheadrightarrow x) \circ \mathtt{+}(\mathtt{S}(\mathtt{Z}), \mathtt{+}(\mathtt{Z}, \mathtt{S}(\mathtt{Z}))) \Longrightarrow \mathtt{+}(\mathtt{S}(\mathtt{Z}), \mathtt{S}(\mathtt{Z}))$

$\vdash TopDown(\mathtt{+}(\mathtt{Z}, x) \twoheadrightarrow x \leftarrow\!\!\!\!+ \mathtt{+}(\mathtt{S}(x), y) \twoheadrightarrow \mathtt{S}(\mathtt{+}(x, y))) \circ \mathtt{+}(\mathtt{S}(\mathtt{Z}), \mathtt{S}(\mathtt{S}(\mathtt{Z}))) \Longrightarrow \mathtt{S}(\mathtt{S}(\mathtt{S}(\mathtt{Z})))$

$\vdash Innermost(\mathtt{+}(\mathtt{Z}, x) \twoheadrightarrow x \leftarrow\!\!\!\!+ \mathtt{+}(\mathtt{S}(x), y) \twoheadrightarrow \mathtt{S}(\mathtt{+}(x, y))) \circ \mathtt{+}(\mathtt{S}(\mathtt{S}(\mathtt{Z})), \mathtt{S}(\mathtt{S}(\mathtt{Z}))) \Longrightarrow \mathtt{S}(\mathtt{S}(\mathtt{S}(\mathtt{S}(\mathtt{Z}))))$

3 Encoding rewriting strategies with rewrite rules

The evaluation of the application of a strategy on a subject consists in setting the "focus" on the active strategy *w.r.t.* the global strategy for control combinators (*e.g.* selecting $S1$ in $S_1 \leftarrow\!\!\!\!+ S_2$ in the inference rule $\mathbf{choice_1}$), in setting the "focus" on the active term(s) *w.r.t.* the global term for traversal combinators (*e.g.* selecting t_i in $f(t_1, \ldots, t_n)$ in the inference rule $\mathbf{one_1}$), and eventually in applying elementary strategies.

The translation function presented in Figure 2 associates to each strategy a set of rewrite rules which encodes exactly this behaviour and preserves the original evaluation: $\mathcal{T}(S) \bullet \varphi_S(t) \longrightarrow u$ whenever $\vdash S \circ t \Longrightarrow u$ (the exact relationship between the strategy and its encoding is formally stated in Section 4). The set $\mathcal{T}(S)$ contains a number of rules whose left-hand sides are headed by φ_S and which encode the behaviour of the strategy S by using, in the right-hand sides, the symbols φ corresponding to the sub-strategies of S and potentially some auxiliary symbols. These φ_S and auxiliary φ symbols are supposed to be freshly generated and uniquely identified, *i.e.* there will be only one φ_S symbol for each encoded (sub-)strategy S and each auxiliary φ symbol can be identified by the strategy it has been generated for. For example, in the encoding $\mathcal{T}(S_1 ; S_2)$, the symbol $\varphi_;$ is just an abbreviation for $\varphi_;^{S_1;S_2}$, *i.e.* the specific $\varphi_;$ used for the encoding of the strategy $S_1 ; S_2$.

The left-hand and right-hand sides of the generated rules are built using the symbols of the original signature, the φ symbols mentioned previously as well as a particular symbol $\bot$ of arity 1 which encodes the failure and whose argument can be used to keep track of the origin of the failure. To keep the presentation of the translation compact and intuitive, we express it using rule schemas which use some special symbols to provide a concise representation of the rewrite rules. We start by introducing these special symbols and we then discuss the translation process.

First, we use the so-called *anti-terms*[1] of the form $!t$ with $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$. Intuitively, an anti-term $!t$ represents all the terms which do not match t and an anti-term $\varphi(!t)$ represents all the terms which do not match $\varphi(t)$; the way the finite representation of these terms is generated is detailed in Section 5. For example, if we consider the signature from Example 1, $!\texttt{+}(\texttt{Z}, x)$ denotes exactly the terms matched by $\texttt{Z}$, $\texttt{S}(x_1)$, $\texttt{+}(\texttt{S}(x_1), x_2)$ or $\texttt{+}(\texttt{+}(x_1, x_2), x_3)$. In this encoding, the semantics of $!t$ with $t \in \mathcal{T}_\mathcal{F}$ are considered *w.r.t.* the terms in $\mathcal{T}_\mathcal{F}$. For example, $!c$ for some constant c does not include $\bot(x)$ or terms of the form $\varphi(x_1, \ldots, x_n)$, because $\bot$ and φ symbols do not belong to the original signature. We can also complement failures but still *w.r.t.* the terms in $\mathcal{T}_\mathcal{F}$ and the pattern $!\bot(x)$ denotes thus all the ground terms $t \in \mathcal{T}_\mathcal{F}$ of the original signature. Terms in the left-hand sides of rules can be aliased, using the symbol "@", by variables which can be then conveniently used in the right-hand sides of the corresponding rules. Moreover, the variable symbol "_" can be used in the left-hand side of a rule to indicate a variable that does not appear in the right-hand side.

For example, the rule schema $\varphi(y \,@\, !\texttt{+}(\texttt{Z}, _)) \twoheadrightarrow \bot(y)$ denotes the set of rewrite rules consisting of $\varphi(\texttt{Z}) \twoheadrightarrow \bot(\texttt{Z})$, $\varphi(\texttt{S}(y_1)) \twoheadrightarrow \bot(\texttt{S}(y_1))$, $\varphi(\texttt{+}(\texttt{S}(y_1), y_2)) \twoheadrightarrow \bot(\texttt{+}(\texttt{S}(y_1), y_2))$ and $\varphi(\texttt{+}(\texttt{+}(y_1, y_2), y_3)) \twoheadrightarrow \bot(\texttt{+}(\texttt{+}(y_1, y_2), y_3))$.

The translation of the *Identity* strategy (**E1**) consists of a rule whose left-hand side matches any term in the signature[2] (contextualized by the corresponding φ symbol) and whose right-hand side is the initial term, and of a rule encoding strict propagation of failure. This latter rule guarantees a faithful encoding of the strategy guided evaluation and is in fact present, in different forms, in the translations of all the strategy operators. Similarly, the translation of the *Fail* strategy (**E2**) contains a failure propagation rule, and a rule whose left-hand side matches any term and whose right-hand side is a failure keeping track of this term.

A rewrite rule (which is an elementary strategy applicable at the root of the subject) is translated (**E3**) by two rules encoding the behaviour in case of respectively a matching success or a failure, together with a rule for failure propagation.

---

1 We restrict here to a limited form of anti-terms; we refer to [7] for the complete semantics of anti-terms.
2 The rule is in fact expanded into n rewrite rules with n the number of symbols in $\mathcal{F}$.

$$
\begin{array}{lrcl}
\textbf{(E1)} & \mathcal{T}(\mathit{Identity}) & = & \{\ \varphi_{\mathit{Identity}}(x \,@\, !\bot(_)) \twoheadrightarrow x, \quad \varphi_{\mathit{Identity}}(\bot(x)) \twoheadrightarrow \bot(x)\ \} \\
\textbf{(E2)} & \mathcal{T}(\mathit{Fail}) & = & \{\ \varphi_{\mathit{Fail}}(x \,@\, !\bot(_)) \twoheadrightarrow \bot(x), \quad \varphi_{\mathit{Fail}}(\bot(x)) \twoheadrightarrow \bot(x)\ \} \\
\textbf{(E3)} & \mathcal{T}(l \twoheadrightarrow r) & = & \{\ \varphi_{l\twoheadrightarrow r}(l) \twoheadrightarrow r, \\
 & & & \quad \varphi_{l\twoheadrightarrow r}(x \,@\, !l) \twoheadrightarrow \bot(x), \quad \varphi_{l\twoheadrightarrow r}(\bot(x)) \twoheadrightarrow \bot(x)\ \ \} \\
 & & & \\
\textbf{(E4)} & \mathcal{T}(S_1\,;S_2) & = & \mathcal{T}(S_1)\ \cup\ \mathcal{T}(S_2) \\
 & & \bigcup & \{\ \varphi_{S_1;S_2}(x \,@\, !\bot(_)) \twoheadrightarrow \varphi_{;}(\varphi_{S_2}(\varphi_{S_1}(x)), x), \quad \varphi_{S_1;S_2}(\bot(x)) \twoheadrightarrow \bot(x), \\
 & & & \quad \varphi_{;}(x \,@\, !\bot(_), _) \twoheadrightarrow x, \quad \varphi_{;}(\bot(_), x) \twoheadrightarrow \bot(x)\ \ \} \\
\textbf{(E5)} & \mathcal{T}(S_1 \leftarrow\!\!\!+ S_2) & = & \mathcal{T}(S_1)\ \cup\ \mathcal{T}(S_2) \\
 & & \bigcup & \{\ \varphi_{S_1 \leftarrow\!\!+ S_2}(x \,@\, !\bot(_)) \twoheadrightarrow \varphi_{\leftarrow\!\!+}(\varphi_{S_1}(x)), \quad \varphi_{S_1 \leftarrow\!\!+ S_2}(\bot(x)) \twoheadrightarrow \bot(x), \\
 & & & \quad \varphi_{\leftarrow\!\!+}(\bot(x)) \twoheadrightarrow \varphi_{S_2}(x), \quad \varphi_{\leftarrow\!\!+}(x \,@\, !\bot(_)) \twoheadrightarrow x\ \ \} \\
\textbf{(E6)} & \mathcal{T}(\mu X\,.\,S) & = & \mathcal{T}(S) \\
 & & \bigcup & \{\ \varphi_{\mu X.S}(x \,@\, !\bot(_)) \twoheadrightarrow \varphi_S(x), \quad \varphi_{\mu X.S}(\bot(x)) \twoheadrightarrow \bot(x), \\
 & & & \quad \varphi_X(x \,@\, !\bot(_)) \twoheadrightarrow \varphi_S(x), \quad \varphi_X(\bot(x)) \twoheadrightarrow \bot(x)\ \ \} \\
\textbf{(E7)} & \mathcal{T}(X) & = & \emptyset \\
 & & & \\
\textbf{(E8)} & \mathcal{T}(\mathit{All}(S)) & = & \mathcal{T}(S) \\
 & & \bigcup & \{\ \varphi_{\mathit{All}(S)}(\bot(x)) \twoheadrightarrow \bot(x)\ \ \} \\
 & & \bigcup_{c\in\mathcal{F}^0} & \{\ \varphi_{\mathit{All}(S)}(c) \twoheadrightarrow c\ \ \} \\
 & & \bigcup_{f\in\mathcal{F}^+} & \{\ \varphi_{\mathit{All}(S)}(f(x_1,\ldots,x_n)) \twoheadrightarrow \varphi_f(\varphi_S(x_1),\ldots,\varphi_S(x_n), f(x_1,\ldots,x_n)), \\
 & & & \quad \varphi_f(x_1 \,@\, !\bot(_),\ldots,x_n \,@\, !\bot(_),_) \twoheadrightarrow f(x_1,\ldots,x_n), \\
 & & & \quad \varphi_f(\bot(_),_,\ldots,_,x) \twoheadrightarrow \bot(x), \\
 & & & \quad \vdots \\
 & & & \quad \varphi_f(_,\ldots,_,\bot(_),x) \twoheadrightarrow \bot(x)\ \ \} \\
\textbf{(E9)} & \mathcal{T}(\mathit{One}(S)) & = & \mathcal{T}(S) \\
 & & \bigcup & \{\ \varphi_{\mathit{One}(S)}(\bot(x)) \twoheadrightarrow \bot(x)\ \ \} \\
 & & \bigcup_{c\in\mathcal{F}^0} & \{\ \varphi_{\mathit{One}(S)}(c) \twoheadrightarrow \bot(c)\ \ \} \\
 & & \bigcup_{f\in\mathcal{F}^+} & \{\ \varphi_{\mathit{One}(S)}(f(x_1,\ldots,x_n)) \twoheadrightarrow \varphi_{f_1}(\varphi_S(x_1),x_2,\ldots,x_n)\ \ \} \\
 & \bigcup_{f\in\mathcal{F}^+} & \bigcup_{1\le i\le ar(f)} & \{\ \varphi_{f_i}(\bot(x_1),\ldots,\bot(x_{i-1}),x_i \,@\, !\bot(_),x_{i+1},\ldots,x_n) \twoheadrightarrow f(x_1,\ldots,x_n)\} \\
 & \bigcup_{f\in\mathcal{F}^+} & \bigcup_{1\le i< ar(f)} & \{\ \varphi_{f_i}(\bot(x_1),\ldots,\bot(x_i),x_{i+1},\ldots,x_n) \twoheadrightarrow \\
 & & & \quad \varphi_{f_{i+1}}(\bot(x_1),\ldots,\bot(x_i),\varphi_S(x_{i+1}),x_{i+2},\ldots,x_n)\ \ \} \\
 & & \bigcup_{f\in\mathcal{F}^+} & \{\ \varphi_{f_n}(\bot(x_1),\ldots,\bot(x_n)) \twoheadrightarrow \bot(f(x_1,\ldots,x_n))\ \ \} \\
 & & & \\
\textbf{(E10)} & \mathcal{B}(\Gamma;X\colon S) & = & \mathcal{B}(\Gamma)\ \cup\ \mathcal{T}(S) \\
 & & \bigcup & \{\ \varphi_X(x \,@\, !\bot(_)) \twoheadrightarrow \varphi_S(x), \quad \varphi_X(\bot(x)) \twoheadrightarrow \bot(x)\ \ \}
\end{array}
$$

Figure 2 Strategy translation.

▶ **Example 2.** The strategy $S_{pz} = \texttt{+}(\texttt{Z}, x) \twoheadrightarrow x$ consisting only of the rewrite rule of Example 1 is encoded by the following rules:

$$\begin{aligned}\mathcal{T}(S_{pz}) = \{\ & \varphi_{pz}(\texttt{+}(\texttt{Z}, x)) \twoheadrightarrow x,\\ & \varphi_{pz}(y \,@\, !\texttt{+}(\texttt{Z}, x)) \twoheadrightarrow \bot(y),\\ & \varphi_{pz}(\bot(x)) \twoheadrightarrow \bot(x)\ \}\end{aligned}$$

which lead, when the anti-terms are expanded *w.r.t.* to the signature, to the TRS:

$$\begin{aligned}\mathcal{T}(S_{pz}) = \{\ & \varphi_{pz}(\texttt{+}(\texttt{Z}, x)) \twoheadrightarrow x,\\ & \varphi_{pz}(\texttt{Z}) \twoheadrightarrow \bot(\texttt{Z}),\\ & \varphi_{pz}(\texttt{S}(y_1)) \twoheadrightarrow \bot(\texttt{S}(y_1)),\\ & \varphi_{pz}(\texttt{+}(\texttt{S}(y_1), y_2)) \twoheadrightarrow \bot(\texttt{+}(\texttt{S}(y_1), y_2)),\\ & \varphi_{pz}(\texttt{+}(\texttt{+}(y_1, y_2), y_3)) \twoheadrightarrow \bot(\texttt{+}(\texttt{+}(y_1, y_2), y_3)),\\ & \varphi_{pz}(\bot(x)) \twoheadrightarrow \bot(x)\ \}\end{aligned}$$

The term $\varphi_{pz}(\texttt{+}(\texttt{Z}, \texttt{S}(\texttt{Z})))$ reduces *w.r.t.* this latter TRS to $\texttt{S}(\texttt{Z})$ and $\varphi_{pz}(\texttt{Z})$ reduces to $\bot(\texttt{Z})$.

The translation of the sequential application of two strategies (**E4**) includes the translation of the respective strategies and some specific rules. A term $\varphi_{S_1;S_2}(t)$ is reduced by the first rule into a term $\varphi_;(\varphi_{S_2}(\varphi_{S_1}(t)), t)$, which guarantees that the rules of the encoding of S_1 are applied before the ones of S_2. Indeed, a term of the form $\varphi(t)$ can be reduced only if $t \in \mathcal{T}_\mathcal{F}$ or $t = \bot(_)$ and thus, the rules for φ_{S_2} can be applied to a term $\varphi_{S_2}(\varphi_{S_1}(_))$ only after $\varphi_{S_1}(_)$ is reduced to a term in $\mathcal{T}_\mathcal{F}$ (or failure). The original subject t is kept during the evaluation (of $\varphi_;$), so that $\bot(t)$ can be returned if the evaluation of S_1 or S_2 fails (*i.e.* produces a $\bot$) at some point. If $\varphi_{S_2}(\varphi_{S_1}(t))$ evaluates to a term $t' \in \mathcal{T}_\mathcal{F}$, then the evaluation of $\varphi_{S_1;S_2}(t)$ succeeds, and t' is the final result. In a similar manner, the translation for the choice operator (**E5**) uses a rule which triggers the application of the rules for S_1. If the corresponding evaluation results in a failure then the application of the rules for S_2 is triggered on the original subject; otherwise the result is returned.

The translation for a strategy $\mu X \,.\, S$ (**E6**) triggers the application of the rules for S at first, and then each time the symbol φ_X is encountered. As in all the other cases, failure is strictly propagated. There is no rewrite rule for the translation of a strategy variable (**E7**) but we should note that the corresponding φ_X symbol could be used when translating the strategy S (in $\mu X \,.\, S$), as we can see in the next example.

▶ **Example 3.** The strategy $S_{rpz} = \mu X.(\texttt{+}(\texttt{Z}, x) \twoheadrightarrow x\ ;\ X) \leftarrow\!\!\!+ \mathit{Identity}$ which applies repeatedly (as long as possible) the rewrite rule from Example 2 is encoded by:

$$\begin{aligned}\mathcal{T}(S_{rpz}) = \ & \{\ \varphi_{rpz}(x \,@\, !\bot(_)) \twoheadrightarrow \varphi_{tpz}(x), && \varphi_{rpz}(\bot(x)) \twoheadrightarrow \bot(x)\ \}\\ \cup\ & \{\ \varphi_X(x \,@\, !\bot(_)) \twoheadrightarrow \varphi_{tpz}(x), && \varphi_X(\bot(x)) \twoheadrightarrow \bot(x)\ \}\\ \cup\ & \{\ \varphi_{tpz}(x \,@\, !\bot(_)) \twoheadrightarrow \varphi_{\leftarrow\!+}(\varphi_{pzX}(x)), && \varphi_{tpz}(\bot(x)) \twoheadrightarrow \bot(x),\\ & \quad \varphi_{\leftarrow\!+}(x \,@\, !\bot(_)) \twoheadrightarrow x, && \varphi_{\leftarrow\!+}(\bot(x)) \twoheadrightarrow \varphi_{\mathit{Identity}}(x)\ \}\\ \cup\ & \{\ \varphi_{\mathit{Identity}}(x \,@\, !\bot(_)) \twoheadrightarrow x, && \varphi_{\mathit{Identity}}(\bot(x)) \twoheadrightarrow \bot(x)\ \}\\ \cup\ & \{\ \varphi_{pzX}(x \,@\, !\bot(_)) \twoheadrightarrow \varphi_;(\varphi_X(\varphi_{pz}(x)), x), && \varphi_{pzX}(\bot(x)) \twoheadrightarrow \bot(x),\\ & \quad \varphi_;(x \,@\, !\bot(_), _) \twoheadrightarrow x, && \varphi_;(\bot(_), x) \twoheadrightarrow \bot(x)\ \}\\ \cup\ & \mathcal{T}(S_{pz})\end{aligned}$$

For presentation purposes, we separated the TRS in sub-sets of rules corresponding to the translation of each operator occurring in the initial strategy. Note that the symbol φ_X used in the rules for the inner sequence can be reduced with the rules generated to handle the recursion operator. The term $\varphi_{rpz}(\texttt{+}(\texttt{Z}, \texttt{+}(\texttt{Z}, \texttt{S}(\texttt{Z}))))$ reduces *w.r.t.* the TRS to $\texttt{S}(\texttt{Z})$.

The rules encoding the traversal operators follow the same principle – the rules corresponding to the translation of the argument strategy S are applied, depending on the traversal operator, to one or all the sub-terms of the subject. For the *All* operator (**E8**), if

the application of S to all the sub-terms succeeds (produces terms in $\mathcal{T_F}$), then the final result is built using the results of each evaluation. If the evaluation of one of the sub-terms produces a $\bot$, a failure with the original subject as origin is returned as a result. Special rules encode the fact that *All* applied to a constant always succeeds; the same behaviour could have been obtained by instantiating the rules for non-constants with $n = 0$, but we preferred an explicit approach for uniformity and efficiency reasons. In the case of the *One* operator (**E9**), if the evaluation for one sub-term results in a failure, then the evaluation of the strategy S is triggered on the next one. If S fails on all sub-terms, a failure with the original subject as origin is returned. The failure in case of constants is necessarily encoded by specific rules.

Finally, each binding $X : S$ of a context (**E10**) is translated by two rules, including the one that propagates failure. The other rule operates as in the recursive case (rule **E6**): applying the strategy variable X to a subject t leads to the application of the rules encoding S to t.

4 Properties of the translation

The goal of the translation is twofold: use well-established methods and tools for plain TRS in order to prove properties of strategy controlled rewrite rules, and offer a generic compiler for user defined strategies. For both items, it is crucial to have a sound and complete translation, and this turns out to be true in our case.

▶ **Theorem 4** (Simulation). *Given a term $t \in \mathcal{T_F}$, a strategy S and a context Γ*

1. $\Gamma \vdash S \circ t \Longrightarrow t'$ *iff* $\mathcal{T}(S) \cup \mathcal{B}(\Gamma) \bullet \varphi_S(t) \longrightarrow t'$, $t' \in \mathcal{T_F}$
2. $\Gamma \vdash S \circ t \Longrightarrow \texttt{Fail}$ *iff* $\mathcal{T}(S) \cup \mathcal{B}(\Gamma) \bullet \varphi_S(t) \longrightarrow \bot(t)$

Proof. The completeness is shown by induction on the height of the derivation tree and the soundness by induction on the length of the reduction. The base cases consisting of the strategies with a constant length reduction – *Identity*, *Fail*, and the rewrite rule – are straightforward to prove since, in particular, the translation of a rule explicitly encodes matching success and failure. Induction is applied for all the other cases and the corresponding proofs rely on some auxiliary properties.

First, the failure is strictly propagated: if $\mathcal{B}(\Gamma) \cup \mathcal{T}(S) \bullet \varphi_S(\bot(t)) \longrightarrow u$, then $u = \bot(t)$. This is essential, in particular, for the sequence case where a failure of the first strategy should be strictly propagated as the final result of the overall sequential strategy.

Second, we note that terms in the signature are in normal form *w.r.t.* the (rules in the) translation of any strategy and that contextualized terms of the form $\varphi_S(t)$ are head-rigid *w.r.t.* to (the translation of) strategies other than S, *i.e.*, they can be reduced at the head position only by the rules obtained for the translation of S and only if t is not contextualized but a term in the signature. More precisely, if for a strategy S' and a context Γ, $\mathcal{B}(\Gamma) \cup \mathcal{T}(S') \bullet \varphi_S(t) \longrightarrow u$ then $t \in \mathcal{T_F}$ and $\mathcal{T}(S) \subseteq \mathcal{B}(\Gamma) \cup \mathcal{T}(S')$ (or $S = X$ and Γ binds X). This guarantees that the steps in the strategy derivation are encoded accurately by the evaluations *w.r.t.* the rules in the translation.

Finally, the origin of the failure is preserved in the sense that if for a $t \in \mathcal{T_F}$, $\varphi_S(t)$ reduces to a failure, then the reduct is necessarily $\bot(t)$. This is crucial in particular for the choice strategy: if the (translation of the) first strategy fails, then the (translation of the) second one should be applied on the initial subject. ◀

As a direct consequence of this property, we obtain that (non-)termination of one system implies the (non-)termination of the other.

▶ **Corollary 5** (Termination). *Given a strategy S and a context Γ, the strategy application $\Gamma \vdash S \circ t$ has a finite derivation for any term $t \in \mathcal{T}_{\mathcal{F}}$ iff $\mathcal{T}(S) \cup \mathcal{B}(\Gamma)$ is terminating.*

The main goal is to prove the termination of some strategy guided system by proving the property for the plain TRS obtained by translation. When termination does not hold, non-terminating TRS reductions correspond to infinite strategy-controlled derivations.

5 Implementation and experimental results

The strategy translation presented in Section 3 has been implemented in a tool called StrategyAnalyser[3], written in Tom, a language that extends Java with high level constructs for pattern matching, rewrite rules and strategies. Given a set of rewrite rules guided by a strategy, the tool generates a plain TRS in AProVE/TTT2 syntax[4] or Tom syntax. In this section we illustrate our approach on two representative examples.

The first one comes from an optimizer for multi-core architectures, a project where abstract syntax trees are manipulated and transformations are expressed using rewrite rules and strategies, and consists of two rewrite rules identified as patterns occurring often in various forms in the project. First, the rewrite rule $g(f(x)) \twoheadrightarrow f(g(x))$ corresponds to the search for an operator g (which can have more than one parameter in the general case) which is pushed down under another operator f (again, this operator may have more than one parameter). This rule is important since the corresponding (innermost) reduction of a term of the form $t_{\mathrm{gf}} = \overbrace{g(\underbrace{f(\cdots(f}_{n}(g(\underbrace{f(\cdots(f}_{n}(g(\underbrace{f(\cdots(f}_{n}(g}^{m}(a)))\cdots)$, with, for example, $n = 10$ and $m = 18$ occurrences of g, involves a lot of computations and could be a performance bottleneck. Second, the rewrite rule $h(x) \twoheadrightarrow g(h(x))$ corresponds to wrapping some parts of a program by some special constructs, like `try/catch` for example, and it is interesting since its uncontrolled application is obviously non-terminating.

At present, a strategy given as input to StrategyAnalyser is written in a simple functional style and a possible strategy for our example could be:

```
let S = signature {a:0, b:0, f:1, g:1, h:1} in
let gfx = { g(f(x)) -> f(g(x)) } in
let hx  = { h(x) -> g(h(x)) } in
let obu(S) = mu X.(one(X) <+ S) in   ## obu stands for OnceBottomUp
let try(S) = S <+ identity in
let repeat(S) = mu X.(try(S ; X)) in  ## naive definition of innermost to
repeat(obu(gfx))                      ## illustrate various possibilites
```

As a second example, we consider the following rewrite rules which implement the distributivity and factorization of symbolic expressions composed of + and * and their application under a specific strategy:

```
let S = signature { Z:0, S:1, +:2, *:2 } in
let dist = { *(x, +(y,z))       -> +(*(x,y),*(x,z)) } in
let fact = { +(*(x,y), *(x,z)) -> *(x,+(y,z)) } in
let innermost(S) = mu X.(all(X) ; ((S ; X) <+ identity)) in
    innermost(dist) ; innermost(fact)
```

[3] source code available at `http://tom.loria.fr/`, directory `jtom/applications`
[4] `http://aprove.informatik.rwth-aachen.de/`

This time the strategy involves rewrite rules which are either non left-linear or non right-linear and which are non-terminating if their application is not guided by a strategy.

The StrategyAnalyser tool is built in a modular way such that the compilation (*i.e.* the translation presented in Section 3) is performed at an abstract level and therefore, new concrete syntaxes and new backends can be easily added.

5.1 Generation of executable TRS

When run with the flag `-tom`, the StrategyAnalyser tool generates a TRS in Tom syntax which can be subsequently compiled into Java code and executed. Moreover, the tool can be configured to generate TRS that use the alias notation or not, and to use the notion of anti-term or not. An encoding using anti-terms and aliasing can be directly used in a Tom program but for languages and tools which do not offer such primitives, aliases and anti-terms have to be expanded into plain rewrite rules. We explain first how this expansion is realized and we discuss then the performances of the obtained executable TRS.

The rules given in Figure 2 can generate two kinds of rules which contain anti-terms. The first family is of the form $\varphi(\ldots, y_i \,@\, !\bot(_), \ldots) \twoheadrightarrow u$ with $y_i \in \mathcal{X}$, and with potentially several occurrences of $!\bot(_)$. These rules can be easily expanded into a family of rules $\varphi(\ldots, y_i \,@\, f(x_1, \ldots, x_n), \ldots) \twoheadrightarrow u$ with such a rule for all $f \in \mathcal{F}$, and with $x_1, \ldots, x_n \in \mathcal{X}$ and $n = ar(f)$. This expansion is performed recursively to eliminate all the instances of $!\bot(_)$. The other rules containing anti-terms come from the translation of rewrite rules (see (E3)) and have the form: $\varphi(y \,@\, !f(t_1, \ldots, t_n)) \twoheadrightarrow \bot(y)$, with $f \in \mathcal{F}^n$, $y \in \mathcal{X}$ and $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{F}, \mathcal{X})$. If the term $f(t_1, \ldots, t_n)$ is linear, then the tool generates two families of rules:

- $\varphi(g(x_1, \ldots, x_m)) \twoheadrightarrow \bot(g(x_1, \ldots, x_m))$ for all $g \in \mathcal{F}$ s.t. $g \neq f$, $x_1, \ldots, x_m \in \mathcal{X}$, $m = ar(g)$,
- $\varphi(f(x_1, \ldots, x_{i-1}, x_i \,@\, !t_i, x_{i+1}, \ldots, x_n)) \twoheadrightarrow \bot(f(x_1, \ldots, x_n))$ for all $i \in [1, n]$ and $t_i \notin \mathcal{X}$,

with the second family of rules recursively expanded, using the same algorithm, until there is no anti-term left. For example, if we consider the signature used for the rule gfx, the rule $\varphi(y \,@\, !\bot(_)) \twoheadrightarrow y$ is expanded into the set of rewrite rules $\{\, \varphi(a) \twoheadrightarrow a,\ \varphi(b) \twoheadrightarrow b,\ \varphi(f(x_1)) \twoheadrightarrow f(x_1),\ \varphi(g(x_1)) \twoheadrightarrow g(x_1),\ \varphi(h(x_1)) \twoheadrightarrow h(x_1) \,\}$ and the rule $\varphi(y @ !g(f(x)) \twoheadrightarrow \bot(y)$ is expanded into the set of rewrite rules $\{\ \varphi(a) \twoheadrightarrow \bot(a),\ \ \varphi(b) \twoheadrightarrow \bot(b),\ \ \varphi(f(x_1)) \twoheadrightarrow \bot(f(x_1)),\ \ \varphi(g(a)) \twoheadrightarrow \bot(g(a)),\ \ \varphi(g(b)) \twoheadrightarrow \bot(g(b)),\ \ \varphi(g(g(x_1))) \twoheadrightarrow \bot(g(g(x_1))),\ \ \varphi(g(h(x_1))) \twoheadrightarrow \bot(g(h(x_1))),\ \varphi(h(x_1)) \twoheadrightarrow \bot(h(x_1)) \,\}$.

This expansion mechanism is more difficult when we want to find a convenient (finite) encoding for non-linear anti-terms and in this case the expansion should be done, in fact, *w.r.t.* the entire translation of a rewrite rule. Given the rules $\varphi(l) \twoheadrightarrow r$ and $\varphi(y \,@\, !l) \twoheadrightarrow \bot(y)$ with $l \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ a non linear term, we consider the linearized version of l, denoted l', with all the variables $x_i \in \mathcal{V}ar\,(l)$ appearing more than once (m_i times, with $m_i > 1$) renamed into $z_i^1, \ldots, z_i^{m_i-1}$ (the first occurrence of x_i is not renamed). Then, these two rules can be translated into:

- $\varphi(y \,@\, !l') \twoheadrightarrow \bot(y)$
- $\varphi(l') \twoheadrightarrow \varphi'(l', x_1 = z_1^1 \wedge \cdots \wedge x_1 = z_1^{m_1-1} \wedge \cdots \wedge x_n = z_n^1 \wedge \cdots \wedge x_n = z_n^{m_n-1})$
- $\varphi'(l', \mathrm{true}) \twoheadrightarrow r$
- $\varphi'(l', \mathrm{false}) \twoheadrightarrow \bot(l')$

with the first rule containing now the linear anti-term $!l'$ expanded as previously. The rules generated for equality and conjunction are as expected.

When considering the rule $+(*(x, y), *(x, z)) \twoheadrightarrow *(x, +(y, z))$ the translation generates the rules $\varphi(+(*(x_1, x_2), *(x_1, x_3))) \twoheadrightarrow *(x_1, +(x_2, x_3))$ and $\varphi(y \,@\, !+(*(x_1, x_2), *(x_1, x_3))) \twoheadrightarrow \bot(y)$,

which are expanded into the following plain TRS:

$$
\begin{array}{lll}
\varphi(\mathtt{Z}) & \twoheadrightarrow & \bot(\mathtt{Z}),\\
\varphi(\mathtt{S}(x_1)) & \twoheadrightarrow & \bot(\mathtt{S}(x_1)),\\
\varphi(*(x_1,x_2)) & \twoheadrightarrow & \bot(*(x_1,x_2)),\\
\varphi(+(x_1,\mathtt{Z})) & \twoheadrightarrow & \bot(+(x_1,\mathtt{Z})),\\
\varphi(+(x_1,\mathtt{S}(x_2))) & \twoheadrightarrow & \bot(+(x_1,\mathtt{S}(x_2))),\\
\varphi(+(x_1,+(x_2,x_3))) & \twoheadrightarrow & \bot(+(x_1,+(x_2,x_3))),\\
\varphi(+(\mathtt{Z},x_2)) & \twoheadrightarrow & \bot(+(\mathtt{Z},x_2)),\\
\varphi(+(\mathtt{S}(x_1),x_2)) & \twoheadrightarrow & \bot(+(\mathtt{S}(x_1),x_2)),\\
\varphi(+(+(x_1,x_2),x_3)) & \twoheadrightarrow & \bot(+(+(x_1,x_2),x_3)),\\
\varphi(+(*(x_1,x_2),*(z_1^1,x_3))) & \twoheadrightarrow & \varphi'(+(*(x_1,x_2),*(z_1^1,x_3)),x_1=z_1^1),\\
\varphi'(+(*(x_1,x_2),*(z_1^1,x_3)),\mathrm{true}) & \twoheadrightarrow & *(x_1,+(x_2,x_3)),\\
\varphi'(+(*(x_1,x_2),*(z_1^1,x_3)),\mathrm{false}) & \twoheadrightarrow & \bot(+(*(x_1,x_2),*(z_1^1,x_3)))
\end{array}
$$

The number of generated rules for a strategy could thus be significant and it is interesting to see how this impacts the efficiency of the execution of such a system.

If we execute a Tom+Java program corresponding to the repeat(obu(gfx)) strategy defined at the beginning of the section and designed using a classic built-in implementation of strategies where strategy failure is implemented by a Java exception, the normalization of the term t_{gf} takes 6.3 s[5] (Table 1, column Tom). When using an alternative built-in implementation with a special encoding of failure which avoids throwing Java exceptions, the computation time decreases to 0.4 s (Table 1, column Tom*). The strategy repeat(obu(gfx)) is translated into an executable TRS containing 90 Tom plain rewrite rules and the normalization takes in this case 0.7 s! More benchmarks for the application of other strategies involving the rules gfx and hx on the same term t_{gf} as well as the application of strategies involving the rules dist and fact on terms containing more than 400 symbols[6] + and * are presented in Table 1.

We observe that, although the number of generated rules could be significant, the execution times of the resulting plain TRS are comparable to those obtained with the native implementation of Tom strategy. This might look somewhat surprising but can be explained when we take a closer look to the way rewriting rules and strategies are generally implemented:

- the implementation of a TRS can be done in an efficient way since the complexity of syntactic pattern matching depends only on the size of the term to be reduce and, thanks to many-to-one matching algorithms [18, 10], the number of rules has almost no impact.
- in Tom, each native strategy constructor is implemented by a Java class with a `visit` method which implements (*i.e.* interprets) the semantics of the corresponding operator. The evaluation of a strategy S on a term t is implemented thus by a call `S.visit(t)` and an exception (`VisitFailure`) is thrown when the application of a strategy fails.

In the generated TRS, the memory allocation involved in the construction of terms headed by the $\bot$ symbol encoding failure appears to be more efficient than the costly Java exception handling. This is reflected by better performances of the plain TRS implementation compared to the exception-based native implementation (especially when the strategy involves a lot of failures). We obtain performances with the generated TRS comparable to an exception-free native implementation of strategies (as we can see with the columns TRS and Tom* in

[5] on a MacPro 3GHz
[6] term of the form $+(t_7,\mathtt{Z})$, with $t_{i+1} = *(\mathtt{Z},+(t_i,t_i))$, and $t_0 = +(\mathtt{Z},\mathtt{Z})$

Table 1 Benchmarks: the column #rules indicates the number of plain rewrite rules generated for the strategy, the column T indicates whether the termination of the rules have been (dis)proven by AProVE, the column TRS indicates the execution time in milliseconds for the executable TRS, the column Tom indicates the execution time of the Tom built-in exception-based implementation and the column Tom* indicates the execution time of the Tom built-in exception-free implementation.

Name	Strategy	#rules	T	TRS	Tom	Tom*
repeat(dist)	$\mu X\,.\,((\mathsf{dist}\,;X) \nleftrightarrow \mathit{Identity})$	60	✓	<5	<5	<5
repeat(fact)	$\mu X\,.\,((\mathsf{fact}\,;X) \nleftrightarrow \mathit{Identity})$	63	✓	<5	13	<5
repeat(dist ; fact)	$\mu X\,.\,(((\mathsf{dist}\,;\mathsf{fact})\,;X) \nleftrightarrow \mathit{Identity})$	83	✗	–	–	–
td(dist)	$\mu X\,.\,((\mathsf{dist} \nleftrightarrow \mathit{Identity})\,;\mathit{All}(X))$	125	✓	39	12	30
obu(fact)	$\mu X\,.\,(\mathit{One}(X) \nleftrightarrow \mathsf{fact})$	73	✓	<5	<5	<5
repeat(obu(fact))	$\mu X\,.\,((\mathsf{obu}(\mathsf{fact})\,;X) \nleftrightarrow \mathit{Identity})$	103	✓	220	2460	120
	td(dist) ; repeat(obu(fact))	218	✓	296	2601	150
rbufact	$\mu X\,.\,(\mathit{All}(X);\ ((\mathsf{fact}\,;\mathit{All}(X)) \nleftrightarrow \mathit{Identity}))$	202	✓	511	427	302
	td(dist) ; rbufact	318	✓	557	453	328
innermost(dist)	$\mu X\,.\,(\mathit{All}(X)\,;((\mathsf{dist}\,;X) \nleftrightarrow \mathit{Identity}))$	135	✓	370	650	230
innermost(fact)	$\mu X\,.\,(\mathit{All}(X)\,;((\mathsf{fact}\,;X) \nleftrightarrow \mathit{Identity}))$	138	✓	345	308	149
	innermost(dist) ; innermost(fact)	138	✓	866	960	340
repeat(td(dist))	$\mu X\,.\,((\mathsf{td}(\mathsf{dist})\,;X) \nleftrightarrow \mathit{Identity})$	155	✗	–	–	–
bu(hx)	$\mu X\,.\,(\mathit{All}(X)\,;(\mathsf{hx} \nleftrightarrow \mathit{Identity}))$	72	✓	5	6	5
td(hx)	$\mu X\,.\,((\mathsf{hx} \nleftrightarrow \mathit{Identity})\,;\mathit{All}(X))$	72	✗	–	–	–
repeat(obu(gfx))	$\mu X\,.\,((\mathsf{obu}(\mathsf{gfx})\,;X) \nleftrightarrow \mathit{Identity})$	90	✓	699	6300	414
innermost(gfx)	$\mu X\,.\,(\mathit{All}(X)\,;((\mathsf{gfx}\,;X) \nleftrightarrow \mathit{Identity}))$	85	✓	565	4180	365
propagate	$\mu X\,.\,(\mathsf{gfx}\,;(\mathit{All}(X) \nleftrightarrow \mathit{Identity}))$	75	✓	<5	<5	<5
bup	$\mu X\,.\,(\mathit{All}(X)\,;(\mathsf{propagate} \nleftrightarrow \mathit{Identity}))$	121	✓	59	46	42

Table 1), because efficient normalization techniques can be used for the plain TRS, since its rewrite rules are not controlled by a programmable strategy.

5.2 Generation of TRS for termination analysis

When run with the flag `-aprove`, the StrategyAnalyser tool generates a TRS in AProVE/TTT2 syntax which can be analyzed by any tool accepting this syntax. In this case, aliases and anti-terms are always completely expanded leading generally to an important number of plain rewrite rules. Fortunately, the number of rules does not seem to be a problem for AProVE and, for example, the termination of the strategy repeat(obu(gfx)), which is translated into 90 rules, is proven in approximately 10 s (using the web interface). Similarly, the termination of the strategy td(dist) ; rbufact, whose definition is given in Table 1, is translated into 318 rules, which can be proven terminating in approximately 75 s.

The termination of some strategies like, for example, repeat(obu(gfx)) might look pretty easy to show for an expert, but termination is less obvious for more complex strategies like, for example, bup, which is a specialized version of repeat(obu(gfx)), or rbufact, which is a variant of bu(fact).

The approach was effective not only in proving termination of some strategies, but also in disproving it when necessary. Once again this might look obvious for some strategies like, for example, td(hx), which involves a non-terminating rewrite rule, but it is less clear for strategies combining terminating rewrite rules or strategies like, *e.g.*, repeat(dist ; fact).

6 Conclusions and further work

We have proposed a translation of programmable strategies into plain rewrite rules that we have proven sound and complete. Well-established termination methods can be thus used to (dis)prove the termination of the obtained TRS and we can deduce, as a direct consequence, the property for the corresponding strategy. Alternatively, the translation can be used as a strategy compiler for languages which do not implement natively such primitives.

The translation has been implemented in Tom and can generate, for the moment, plain TRS using either a Tom or an AProVE/TTT2 syntax. We have experimented with classic strategies and AProVE and TTT2 have been able to (dis)prove the termination even when the number of generated rules was significant. The performances for the generated executable TRS are comparable to the ones of the Tom built-in (exception-free) strategies.

The framework can be of course improved. We expect problems in (dis)proving termination when the number of generated rewrite rules is too big, and we are currently working on a meta-level representation of the strategy translation which abstracts over the signature and considerably decreases the size of the generated TRS compared to the approach of this paper. When termination is disproven and a counter-example can be exhibited, it is interesting to reproduce the corresponding infinite reductions in terms of strategy derivations. Since the TRS reductions corresponding to distinct (sub-)strategy derivations are not interleaved, we think that the back-translation of the counter-examples provided by the termination tools can be automatized.

As far as the executable TRS is concerned, we intend to develop new backends allowing the integration of programmable strategies in other languages than Tom.

References

1 Beatriz Alarcón, Raúl Gutiérrez, and Salvador Lucas. Context-sensitive dependency pairs. *Inf. Comput.*, 208(8):922–968, 2010.

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

3 Franz Baader and Tobias Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

4 Emilie Balland, Paul Brauner, Radu Kopetz, Pierre-Etienne Moreau, and Antoine Reilles. Tom: Piggybacking rewriting on java. In *RTA'07*, volume 4533 of *LNCS*, pages 36–47. Springer-Verlag, 2007.

5 Emilie Balland, Pierre-Etienne Moreau, and Antoine Reilles. Effective strategic programming for Java developers. *Software: Practice and Experience*, 44(2):129–162, 2012.

6 P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and C. Ringeissen. An overview of ELAN. In *WRLA'98*, volume 15. ENTCS, 1998.

7 Horatiu Cirstea, Claude Kirchner, Radu Kopetz, and Pierre-Etienne Moreau. Anti-patterns for rule-based languages. *Journal of Symbolic Computation*, 45(5):523–550, 2010. Symbolic Computation in Software Science.

8 Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. The Maude 2.0 System. In *RTA'03*, volume 2706 of *LNCS*, pages 76–87. Springer-Verlag, 2003.

9 Jörg Endrullis and Dimitri Hendriks. From outermost to context-sensitive rewriting. In *RTA'09*, volume 5595 of *LNCS*, pages 305–319. Springer, 2009.

10 Fabrice Le Fessant and Luc Maranget. Optimizing pattern matching. In *ICFP'01*, pages 26–37. ACM Press, 2001.

11 Olivier Fissore, Isabelle Gnaedig, and Hélène Kirchner. Simplification and termination of strategies in rule-based languages. In *PPDP'03*, pages 124–135. ACM, 2003.

12 Carsten Fuhs, Jürgen Giesl, Michael Parting, Peter Schneider-Kamp, and Stephan Swiderski. Proving termination by dependency pairs and inductive theorem proving. *Journal of Automated Reasoning*, 47(2):133–160, 2011.

13 Jürgen Giesl and Aart Middeldorp. Innermost termination of context-sensitive rewriting. In *DLT'02*, volume 2450 of *LNCS*, pages 231–244. Springer, 2002.

14 Jürgen Giesl and Aart Middeldorp. Transformation techniques for context-sensitive rewrite systems. *J. Funct. Program.*, 14(4):379–427, 2004.

15 Jürgen Giesl, Matthias Raffelsieper, Peter Schneider-Kamp, Stephan Swiderski, and René Thiemann. Automated termination proofs for haskell by term rewriting. *ACM Trans. Program. Lang. Syst.*, 33(2):7, 2011.

16 Jürgen Giesl, René Thiemann, and Stephan Falke. Mechanizing and improving dependency pairs. *Journal of Automated Reasoning*, 37:2006, 2006.

17 Isabelle Gnaedig and Hélène Kirchner. Termination of rewriting under strategies. *ACM Trans. Comput. Log.*, 10(2), 2009.

18 Albert Gräf. Left-to-right tree pattern matching. In *RTA'91*, volume 488 of *LNCS*, pages 323–334. Springer-Verlag, April 1991.

19 Nao Hirokawa and Aart Middeldorp. Automating the dependency pair method. *Information and Computation*, 199(1–2):172–199, 2005.

20 J.-P. Jouannaud and Claude Kirchner. Solving equations in abstract algebras: a rule-based survey of unification. In *Computational Logic. Essays in honor of Alan Robinson*, chapter 8, pages 257–321. The MIT press, 1991.

21 Markus Kaiser and Ralf Lämmel. An Isabelle/HOL-based model of stratego-like traversal strategies. In *PPDP'09*, pages 93–104. ACM, 2009.

22 Martin Korp, Christian Sternagel, Harald Zankl, and Aart Middeldorp. Tyrolean Termination Tool 2. In *RTA'09*, volume 5595 of *LNCS*, pages 295–304. Springer-Verlag, 2009.

23 Ralf Lämmel, Simon J. Thompson, and Markus Kaiser. Programming errors in traversal programs over structured data. *Sci. Comput. Program.*, 78(10):1770–1808, 2013.

24 José Meseguer and Christiano Braga. Modular rewriting semantics of programming languages. In *Algebraic Methodology and Software Technology*, volume 3116 of *LNCS*, pages 364–378. Springer Berlin Heidelberg, 2004.

25 Pierre-Etienne Moreau, Christophe Ringeissen, and Marian Vittek. A Pattern Matching Compiler for Multiple Target Languages. In *CC'03*, volume 2622 of *LNCS*, pages 61–76. Springer-Verlag, 2003.

26 Matthias Raffelsieper and Hans Zantema. A transformational approach to prove outermost termination automatically. *ENTCS*, 237:3–21, 2009.

27 Terese. *Term Rewriting Systems*. Cambridge University Press, 2003. M. Bezem, J. W. Klop and R. de Vrijer, eds.

28 René Thiemann. From outermost termination to innermost termination. In *SOFSEM'09*, volume 5404 of *LNCS*, pages 533–545. Springer, 2009.

29 René Thiemann and Aart Middeldorp. Innermost termination of rewrite systems by labeling. *ENTCS*, 204:3–19, 2008.

30 Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In *RTA'01*, volume 2051 of *LNCS*, pages 357–361. Springer-Verlag, 2001.

31 Eelco Visser, Zine-el-Abidine Benaissa, and Andrew Tolmach. Building program optimizers with rewriting strategies. In *ICFP'98*, pages 13–26. ACM Press, 1998.

Presenting a Category Modulo a Rewriting System*

Florence Clerc[1] **and Samuel Mimram**[2]

1 **CEA, LIST, Software Reliability Laboratory, PC 174, 91191 Gif-sur-Yvette, France**
LIX, École Polytechnique, France
PPS, Université Paris Diderot, France
`florence.clerc@cea.fr`

2 **LIX, École Polytechnique, France**
`samuel.mimram@lix.polytechnique.fr`

Abstract

Presentations of categories are a well-known algebraic tool to provide descriptions of categories by the means of generators, for objects and morphisms, and relations on morphisms. We generalize here this notion, in order to consider situations where the objects are considered modulo an equivalence relation (in the spirit of rewriting modulo), which is described by equational generators. When those form a convergent (abstract) rewriting system on objects, there are three very natural constructions that can be used to define the category which is described by the presentation: one is based on restricting to objects which are normal forms, one consists in turning equational generators into identities (i.e. considering a quotient category), and one consists in formally adding inverses to equational generators (i.e. localizing the category). We show that, under suitable coherence conditions on the presentation, the three constructions coincide, thus generalizing celebrated results on presentations of groups. We illustrate our techniques on a non-trivial example, and hint at a generalization for 2-categories.

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

Keywords and phrases presentation of a category, quotient category, localization, residuation

Digital Object Identifier 10.4230/LIPIcs.RTA.2015.89

1 Introduction

Motivated by generalizing rewriting techniques to the setting of higher-dimensional categories, we introduce here a notion of presentation of a category modulo a rewriting system, in order to be able to present a category as generated by objects and morphisms, quotiented by relations on both morphisms *and objects*. This work can somehow be seen as an extension of traditional techniques of rewriting modulo an equational theory [1], in the case where the equational theory can itself be oriented as a convergent rewriting system, called an equational rewriting system. We provide here coherence conditions on both the original rewriting system and the equational one, so that expected properties hold: for instance, it should be "the same" to rewrite terms modulo the equational theory than to rewrite terms in normal form wrt the equational rewriting system. In this introduction, we expose our motivations, which come from higher-dimensional rewriting theory, however very little knowledge about this setting will be required in the remaining of the article.

* This work was partially supported by CATHRE French ANR project ANR-13-BS02-0005-02.

© Florence Clerc and Samuel Mimram;
licensed under Creative Commons License CC-BY
26th International Conference on Rewriting Techniques and Applications (RTA'15).
Editor: Maribel Fernández; pp. 89–105
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

A *string rewriting system* P consists in an alphabet P_1 and a set $P_2 \subseteq P_1^* \times P_1^*$ of rules. Such a system induces a monoid $\|P\| = P_1^*/\overset{*}{\Leftrightarrow}$ obtained by quotienting the free monoid P_1^* on P_1 by the smallest congruence $\overset{*}{\Leftrightarrow}$ containing the rules in P_2; when the rewriting system is convergent, i.e. both confluent and terminating, normal forms provide canonical representatives of equivalence classes. Given a monoid M, we say that P is a *presentation* of M when M is isomorphic to $\|P\|$: in this case, the elements of P_1 can be seen as *generators* for M, and the elements of P_2 as a complete set of *relations* for M. For instance, the additive monoid $\mathbb{N} \times \mathbb{N}$ admits the presentation P with $P_1 = \{a, b\}$ and $P_2 = \{ba \Rightarrow ab\}$: namely, the string rewriting system is convergent, and its normal forms are words of the form $a^m b^n$, with $(m, n) \in \mathbb{N} \times \mathbb{N}$, from which it is easy to build the required isomorphism.

The notion of presentation is easy to generalize from monoids to categories (a monoid being the particular case of a category with one object): a presentation of category consists in generators for objects and morphisms, together with rules relating morphisms in the free category generated by the generators. Starting from this observation, people have considered a wild generalization of the notion of presentation, in order to present n-categories (computads [14, 13] or polygraphs [5]), thus providing us with a notion of *higher-dimensional rewriting system*. While we will not, in this article, consider much more than presentations of categories, the motivation for this work really comes from a limitation in presentations of 2-categories (and higher-dimensional categories) that we would like to overcome. We shall explain it on a simple example of a monoidal category (which, again, is a particular case of a 2-category with only one 0-cell).

Consider the well-known *simplicial category* Δ whose objects are integers $n \in \mathbb{N}$ and morphisms $f : m \to n$ are increasing functions $f : [m] \to [n]$ where $[m] = \{0, \ldots, m-1\}$. This category is monoidal, with tensor product being given by addition on objects ($m \otimes n = m + n$) and by "juxtaposition" on morphisms, and it is well known that it admits the following presentation as a monoidal category [12, 10]: its objects are generated by one object a, its morphisms are generated by $\mu : a \otimes a \to a$ and $\eta : I \to a$, and the relations are $\alpha : \mu \circ (\mu \otimes \mathrm{id}_a) = \mu \circ (\mathrm{id}_a \otimes \mu)$, $\lambda : \mu \circ (\eta \otimes \mathrm{id}_a) = \mathrm{id}_a$ and $\rho : \mu \circ (\mathrm{id}_a \otimes \eta) = \mathrm{id}_a$. This means that every morphism of Δ can be obtained as a composite of η and μ, and that two such formal composites represent the same morphism precisely when they can be related by the congruence generated by the above relations. As we can see on this example, a presentation P of a monoidal category consists in generators for objects (here $P_1 = \{a\}$), generators for morphisms ($P_2 = \{\eta, \mu\}$) and relations between composite of morphisms ($P_3 = \{\alpha, \lambda, \rho\}$). Notice that such a presentation *does not allow for relations between objects*, and thus is restricted to presenting monoidal categories whose underlying monoid of objects is free.

This limitation can be better understood by trying to present the monoidal category $\Delta \times \Delta$ with tensor product extending componentwise the one of Δ: the underlying monoid of objects is $\mathbb{N} \times \mathbb{N}$, which is not free. If we try to construct a presentation for this monoidal category, seen as consisting of "two copies" of the above category Δ, we are lead to consider a presentation containing "two copies" of the previous presentation: we consider a presentation P with $P_1 = \{a, b\}$ as object generators (where a and b respectively correspond to the objects $(1, 0)$ and $(0, 1)$), with $P_2 = \{\mu_a, \eta_a, \mu_b, \eta_b\}$ as morphism generators (with $\mu_a : a \otimes a \to a$, $\mu_b : b \otimes b \to b$, etc.), and with $P_3 = \{\alpha_a, \lambda_a, \rho_a, \alpha_b, \lambda_b, \rho_b\}$ as relations. If we stop here adding relations, the presented category has $\{a, b\}^*$ as underlying monoid of objects, which is not right: recalling the above presentation for $\mathbb{N} \times \mathbb{N}$, we should moreover add a relation $\gamma : ba = ab$. However, such a relation between objects is not allowed in the usual notion of presentation (only relations between morphisms are usually considered). In order to provide a meaning to it, three constructions are available: restrict P to some canonical representatives of objects

modulo the equivalence generated by γ (typically the words of the form $a^m b^n$), quotient by γ the monoidal category $\|P\|$ presented by P, or formally invert the morphism γ in $\|P\|$. We show that under reasonable assumptions on the presentation, all three constructions coincide, thus providing one with a notion of *coherent* presentation modulo. As a fist step toward this situation, we study here the simpler case of presentations of categories and introduce a notion of presentation modulo for those, leaving the case of 2-categories for future work.

We begin by recalling the notion of presentation of a category (Section 2.1), then we extend it to work modulo a relation on objects (Section 2.2), and consider the quotient and localization wrt to the relation (Section 2.3). In order to compare those constructions, we consider equational rewriting systems equipped with a notion of residuation (Section 3.1) and satisfying a particular "cylinder" property (Section 3.2). We then show that, under suitable coherence conditions, the category of normal forms is isomorphic to the quotient (Section 4.1) and equivalent with the localization (Section 4.2), and illustrate our results on an example (Section 4.3). We finally discuss a possible extension of this work to presentations of 2-categories (Section 5) and conclude (Section 6).

2 Presentations of categories modulo a rewriting system

2.1 Presentations of categories

Recall that a *graph* (P_0, s_0, t_0, P_1) consists of two sets P_0 and P_1, of *vertices* and *edges* respectively, together with two functions $s_0, t_0 : P_1 \to P_0$ associating to an edge its *source* and *target* respectively. Such a graph generates a category with P_0 as objects and the set P_1^* of (directed) paths as morphisms. If we denote by $i_1 : P_1 \to P_1^*$ the coercion of edges to paths of length 1, and $s_0^*, t_0^* : P_1^* \to P_0$ the functions associating to a path its source and target respectively, we thus obtain a diagram as on the left below:

$$
\begin{array}{ccc}
 & & P_1 \\
 & {\scriptstyle s_0,\,t_0}\swarrow\!\!\swarrow & \downarrow{\scriptstyle i_1} \\
P_0 & \overset{s_0^*}{\underset{t_0^*}{\leftleftarrows}} & P_1^*
\end{array}
\qquad\qquad
\begin{array}{ccccc}
 & & P_1 & & P_2 \\
 & {\scriptstyle s_0,\,t_0}\swarrow\!\!\swarrow & \downarrow{\scriptstyle i_1} & \swarrow\!\!\swarrow{\scriptstyle s_1,\,t_1} & \\
P_0 & \overset{s_0^*}{\underset{t_0^*}{\leftleftarrows}} & P_1^* & &
\end{array}
\tag{1}
$$

in **Set** which is commuting in the sense that $s_0^* \circ i_1 = s_0$ and $t_0^* \circ i_1 = t_0$.

▶ **Definition 1.** A *presentation* $P = (P_0, s_0, t_0, P_1, s_1, t_1, P_2)$, as pictured on the right of (1), consists in a graph (P_0, s_0, t_0, P_1) as above, the elements of P_0 (resp. P_1) being called *object* (resp. *morphism*) *generators*, together with a set P_2 of *relations* (or *2-generators*) and two functions $s_1, t_1 : P_2 \to P_1^*$ such that $s_0^* \circ s_1 = s_0^* \circ t_1$ and $t_0^* \circ s_1 = t_0^* \circ t_1$. The category $\|P\|$ *presented* by P is the category obtained from the category generated by the graph (P_0, s_0, t_0, P_1) by quotienting morphisms by the smallest congruence wrt composition identifying any two morphisms f and g such that there exists $\alpha \in P_2$ satisfying $s_1(\alpha) = f$ and $t_1(\alpha) = g$.

In the following, we often simply write (P_0, P_1, P_2) for a presentation as above, leaving the source and target maps implicit. We write $f : x \to y$ for an edge $f \in P_1$ with $s_0(f) = x$ and $t_0(f) = y$, and $\alpha : f \Rightarrow g$ for a relation with f as source and g as target. We sometimes write $\alpha : f \Leftrightarrow g$ to indicate that $\alpha : f \Rightarrow g$ or $\alpha : g \Rightarrow f$ is an element of P_2, and we denote by $\overset{*}{\Leftrightarrow}$ the smallest congruence such that $f \overset{*}{\Leftrightarrow} g$ whenever there exists $\alpha : f \Rightarrow g$ in P_2.

▶ **Example 2.** The monoid $\mathbb{N}/2\mathbb{N}$ (seen as a category with only one object) admits the presentation P with $P_0 = \{x\}$, $P_1 = \{f : x \to x\}$ and $P_2 = \{\varepsilon : f \circ f \Rightarrow \mathrm{id}_x\}$.

▶ Remark. A presentation P generates a 2-category with invertible 2-cells (also called a (2,1)-category), whose set of 2-cells is denoted P_2^*, and the category presented by P is obtained from this 2-category by identifying 1-cells where there is a 2-cell in between [5, 10]. We write $\alpha : f \overset{*}{\Leftrightarrow} g$ for such a 2-cell.

▶ **Lemma 3.** *Any category $\mathcal{C}$ admits a presentation $P^{\mathcal{C}}$, called its* standard presentation*, with $P_0^{\mathcal{C}}$ being the set of objects of $\mathcal{C}$, $P_1^{\mathcal{C}}$ being the set of morphisms of $\mathcal{C}$ and $P_2^{\mathcal{C}}$ being the set of pairs $(f_2 \circ f_1, g) \in P_1^{\mathcal{C}*} \times P_1^{\mathcal{C}*}$ with $f_1, f_2, g \in P_1$ such that $s_0(f_1) = s_0(g)$, $t_0(f_2) = t_0(g)$ and $f_2 \circ f_1 = g$ in $\mathcal{C}$ (with projections as source and target functions).*

By previous lemma, every category admits at least one presentation. In general, it actually admits many presentations. It can be shown that two presentations present the same category if and only if they are related by Tietze transformations: those transformations generate all the operations one can do on a presentation without modifying the presented category [15, 8]. For instance, Knuth-Bendix completions are a particular case of those [9].

▶ **Definition 4.** Given a presentation P, a *Tietze transformation* consists in

- adding (resp. removing) a generator $f \in P_1$ and a relation $\alpha : f \Rightarrow g \in P_2$ with $g \in (P_1 \setminus \{f\})^*$,
- adding (resp. removing) a relation $\alpha : f \Rightarrow g \in P_2$ such that f and g are equivalent wrt the congruence generated by the relations in $P_2 \setminus \{\alpha\}$.

▶ **Proposition 5.** *Two presentations P and P' are related by a finite sequence of Tietze transformations if and only if they present the same category, i.e. $\|P\| \cong \|P'\|$.*

2.2 Presentations modulo

In a presentation P of a category, relations are generated by elements of P_2: the morphisms of the free category on the underlying graph will be quotiented by those in order to obtain the presented category. We now extend this notion in order to also allow for quotienting objects in the process of constructing the presented category.

▶ **Definition 6.** A *presentation modulo* $(P, \tilde{P}_1)$ consists of a presentation $P = (P_0, P_1, P_2)$ together with a set $\tilde{P}_1 \subseteq P_1$, whose elements are called *equational generators*.

The morphisms of $\|P\|$ generated by the equational generators are called *equational morphisms*. Intuitively, the category presented by a presentation modulo should be the "quotient category" $\|P\| / \tilde{P}_1$, as explained in next section, where objects equivalent under $\tilde{P}_1$ (i.e. related by equational morphisms) are identified. We believe that the reason why presentations modulo of categories were not introduced before is that they are unnecessary, in the sense that we can always convert a presentation modulo into a regular presentation, see Lemma 10 below. However, the techniques developed here extend in the case of 2-categories (this will be developed in a subsequent article) and moreover, our framework already enables us to easily obtain interesting results on presented categories, see Section 4.3.

▶ **Definition 7.** Given a presentation modulo $(P, \tilde{P}_1)$, we define the presentation $P/\tilde{P}_1$ as the presentation (P'_0, P'_1, P'_2) where

- $P'_0 = P_0/\cong_1$ where $\cong_1$ is the smallest equivalence such that $x \cong_1 y$ whenever there exists a generator $f : x \to y$ in $\tilde{P}_1$, and we denote $[x]$ the equivalence class of $x \in P_0$,
- the elements of P'_1 are $f : [x] \to [y]$ for $f : x \to y$ in P_1,
- the elements of P'_2 are of the form $\alpha : f \to g$ for $\alpha : f \to g$ in P_2, or $\alpha_f : f \to \mathrm{id}_{[x]}$ for $f : x \to y$ in $\tilde{P}_1$.

We will sometimes need to consider presentations modulo with "arrows reversed":

▶ **Definition 8.** Given a presentation modulo $(P, \tilde{P}_1)$, the presentation modulo $(P^{\mathrm{op}}, \tilde{P}_1^{\mathrm{op}})$ is given by $P^{\mathrm{op}} = (P_0, P_1^{\mathrm{op}}, P_2^{\mathrm{op}})$ where $P_1^{\mathrm{op}} = \{f^{\mathrm{op}} : y \to x \mid f : x \to y \in P_1\}$ and where $P_2^{\mathrm{op}} = \{\alpha^{\mathrm{op}} : f^{\mathrm{op}} \Rightarrow g^{\mathrm{op}} \mid \alpha : f \Rightarrow g\}$ with $f^{\mathrm{op}} = f_1^{\mathrm{op}} \circ ... \circ f_k^{\mathrm{op}}$ for $f = f_k \circ \ldots \circ f_1$ and where $\tilde{P}_1^{\mathrm{op}}$ is the subset of P_1^{op} corresponding to $\tilde{P}_1$.

2.3 Quotient and localization of a presentation modulo

As explained above, we want to quotient our presentations modulo by equational morphisms, in order for the equational morphisms to induce equalities in the presented category. Given a category $\mathcal{C}$ and a set Σ of morphisms, there are essentially two canonical ways to "get rid" of the morphisms of Σ in $\mathcal{C}$: we can either force them to be identities, or to be isomorphisms, giving rise to the two following notions of quotient and localization of a category. These are standard construction in category theory and we recall them below.

▶ **Definition 9.** The *quotient* of a category $\mathcal{C}$ by a set Σ of morphisms of $\mathcal{C}$ is a category $\mathcal{C}/\Sigma$ together with a *quotient functor* $Q : \mathcal{C} \to \mathcal{C}/\Sigma$ sending the elements of Σ to identities, such that for every functor $F : \mathcal{C} \to \mathcal{D}$ sending the elements of Σ to identities, there exists a unique functor $\tilde{F}$ such that $\tilde{F} \circ Q = F$.

Such a quotient category always exists for general reasons [2] and is unique up to isomorphism. Given a presentation modulo $(P, \tilde{P}_1)$, the category presented by the associated (non-modulo) presentation $P/\tilde{P}_1$ described in Definition 7, corresponds to considering the category presented by the (non-modulo) presentation P and quotient it by $\tilde{P}_1$.

▶ **Lemma 10.** *Suppose given a presentation modulo $(P, \tilde{P}_1)$, the categories $\|P\| / \tilde{P}_1$ and $\left\|P/\tilde{P}_1\right\|$ are isomorphic.*

A second, slightly different construction, consists in turning elements of Σ into isomorphisms (instead of identities):

▶ **Definition 11.** The *localization* of a category $\mathcal{C}$ by a set Σ of morphisms is the category $\mathcal{C}[\Sigma^{-1}]$ together with a *localization functor* $L : \mathcal{C} \to \mathcal{C}[\Sigma^{-1}]$ sending the elements of Σ to isomorphisms, such that for every functor $F : \mathcal{C} \to \mathcal{D}$ sending the elements of Σ to isomorphisms, there exists a unique functor $\tilde{F}$ such that $\tilde{F} \circ L = F$.

In the case where the category is presented, its localization admits the following presentation.

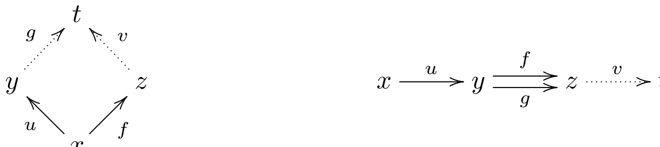
▶ **Lemma 12.** *Given a presentation $P = (P_0, P_1, P_2)$ and a subset Σ of P_1, the category presented by $P' = (P_0, P_1', P_2')$ where $P_1' = P_1 \uplus \left\{\overline{f} : y \to x \;\middle|\; f : x \to y \in \Sigma\right\}$ and where $P_2' = P_2 \uplus \left\{\overline{f} \circ f \Rightarrow \mathrm{id}, f \circ \overline{f} \Rightarrow \mathrm{id} \;\middle|\; f \in \Sigma\right\}$ is a localization of the category $\|P\|$ by Σ.*

▶ **Example 13.** Let us consider the category $\mathcal{C} = x \overset{f}{\underset{g}{\rightrightarrows}} y$ with two objects and two non-trivial morphisms. Its localization by $\Sigma = \{f, g\}$ is equivalent to the category with one object and $\mathbb{Z}$ as set of morphisms (with addition as composition), whereas its quotient by Σ is the category with one object and only identity as morphism. Notice that they are not equivalent.

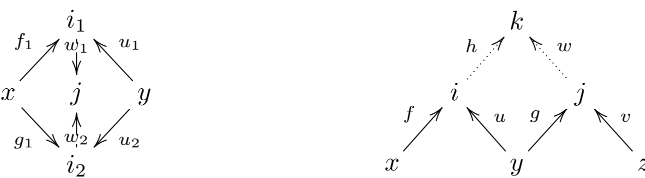
The description of the localization of a category provided by the universal property is often difficult to work with. When the set Σ has nice properties, the localization admits a much more tractable description [7, 4].

▶ **Definition 14.** A set Σ of morphisms of a category $\mathcal{C}$ is a *left calculus of fractions* when

1. the set Σ is closed under composition : for f and g composable morphisms in Σ, $g \circ f$ is in σ.
2. Σ contains the identities id_x for x in P_0.
3. for every pair of coinitial morphisms $u : x \to y$ in Σ and $f : x \to z$ in $\mathcal{C}$, there exists a pair of cofinal morphisms $v : z \to t$ in Σ and $g : y \to t$ in $\mathcal{C}$ such that $v \circ f = g \circ u$.
4. for every morphism $u : x \to y$ in Σ and pair of parallel morphisms $f, g : y \to z$ such that $f \circ u = g \circ u$ there exists a morphism $v : z \to t$ in Σ such that $v \circ f = v \circ g$.



▶ **Theorem 15.** *When Σ is a left calculus of fractions for a category $\mathcal{C}$, the localization $\mathcal{C}[\Sigma^{-1}]$ can be described as the* category of fractions *whose objects are the objects of $\mathcal{C}$ and morphisms from x to y are equivalence classes of pairs of cofinal morphisms (f, u) with $f : x \to i \in \mathcal{C}$ and $u : y \to i \in \Sigma$ under the equivalence relation identifying two such pairs (f_1, u_1) and (f_2, u_2) where there exists two morphisms $w_1, w_2 \in \Sigma$ such that $w_1 \circ u_1 = w_2 \circ u_2$ and $w_1 \circ f_1 = w_2 \circ f_2$, as shown on the left:*



identity on an object x is the equivalence class of $(\mathrm{id}_x, \mathrm{id}_x)$ and composition of two morphisms $(f, u) : x \to y$ and $(g, v) : y \to z$ is the equivalence class of $(h \circ f, w \circ v) : x \to z$ where the morphisms h and w are provided by property 1 of Definition 14.

In such a situation, the following property often enables one to show that there is a full and faithful embedding of the category into its localization [4]:

▶ **Proposition 16.** *Given a left calculus of fractions Σ for a category $\mathcal{C}$, if all the morphisms of Σ are mono then the inclusion functor $F : \mathcal{C} \to \mathcal{C}[\Sigma^{-1}]$ is faithful, where F is the identity on objects and sends a morphism $f : x \to y$ to (f, id_y).*

Given a presentation modulo, when the (abstract) rewriting system on objects given by the equational generators is convergent, normal forms for objects provide canonical representatives of objects modulo equational generators, and therefore we are actually provided with three possible and equally reasonable constructions for the category presented by a presentation modulo $(P, \tilde{P}_1)$:

1. the full subcategory on $\|P\|$ whose objects are normal forms wrt $\tilde{P}_1$,
2. the quotient category $\|P\| / \tilde{P}_1$,
3. the localization $\|P\| [\tilde{P}_1^{-1}]$.

The aim of this article is to provide reasonable assumptions on the presentation modulo ensuring that the two first categories are isomorphic, and equivalent to the third one. We introduce them gradually.

3 Confluence properties

In this section, we introduce a series of local conditions that our presentations modulo should satisfy in order for the constructions recalled above to coincide. These can be seen as a generalization of classical local confluence properties in our context in which rewriting rules correspond to equational generators only, and in which we keep track of 2-cells witnessing local confluence.

3.1 Residuation

We begin by extending to our setting the notion of residual, which is often associated to a confluent rewriting system in order to “keep track” of rewriting steps once others have been performed [11, 3, 6].

▶ **Assumption 1.** We suppose fixed a presentation modulo $(P, \tilde{P}_1)$ such that

1. for every pair of distinct coinitial generators $f : x \to y_1$ in $\tilde{P}_1$ and $g : x \to y_2$ in P_1, there exist a pair of cofinal morphisms $g' : y_1 \to z$ in P_1^* and $f' : y_2 \to z$ in $\tilde{P}_1^*$ and a relation $\alpha : g' \circ f \Leftrightarrow f' \circ g$ in P_2, as shown on the right,
2. there is no infinite path with generators in $\tilde{P}_1$.

$$\begin{array}{ccc} y_1 & \overset{g'}{\dashrightarrow} & z \\ {\scriptstyle f}\uparrow & \overset{\alpha}{\Leftrightarrow} & \uparrow{\scriptstyle f'} \\ x & \underset{g}{\to} & y_2 \end{array}$$

These assumptions ensure in particular that the (abstract) rewriting system on vertices with $\tilde{P}_1$ as set of rules is convergent. Given a vertex $x \in P_0$, we write $\hat{x}$ for the associated normal form. For every pair of distinct morphisms (f, g), as in the first assumption, we suppose fixed an arbitrary choice of a particular triple (g', α, f') associated to it, and write g/f for g', f/g for f' and $\rho_{f,g}$ for α. The morphism g/f (resp. f/g) is as the *residual* of g after f (resp. f after g): intuitively, g/f corresponds to what remains of g once f has been performed. It is natural to extend this definition to paths as follows:

▶ **Definition 17.** Given two coinitial paths $f : x \to y$ and $g : x \to z$ and P_1^* such that either f or g is in $\tilde{P}_1^*$, we define the *residual* g/f of g after f as above when f and g are distinct generators, and by induction with $f/f = \mathrm{id}_y$ and

$$g/\,\mathrm{id}_x = g \qquad \mathrm{id}_x/f = \mathrm{id}_y \qquad (g_2 \circ g_1)/f = (g_2/(f/g_1)) \circ (g_1/f) \qquad g/(f_2 \circ f_1) = (g/f_1)/f_2$$

(by convention the residual g/f is not defined when neither f nor g belongs to $\tilde{P}_1^*$). Graphically,

$$\begin{array}{ccc} \cdot & \xrightarrow{\mathrm{id}} & \cdot \\ {\scriptstyle g}\uparrow & & \uparrow{\scriptstyle g/\mathrm{id}=g} \\ \cdot & \xrightarrow[\mathrm{id}]{} & \cdot \end{array} \qquad \begin{array}{ccc} \cdot & \xrightarrow{f} & \cdot \\ {\scriptstyle \mathrm{id}}\uparrow & & \uparrow{\scriptstyle \mathrm{id}/f=\mathrm{id}} \\ \cdot & \xrightarrow[f]{} & \cdot \end{array} \qquad \begin{array}{ccc} \cdot & \xrightarrow{f/(g_2\circ g_1)} & \cdot \\ {\scriptstyle g_2}\uparrow & & \uparrow{\scriptstyle g_2/(f/g_1)} \\ \cdot & \xrightarrow{f/g_1} & \cdot \\ {\scriptstyle g_1}\uparrow & & \uparrow{\scriptstyle g_1/f} \\ \cdot & \xrightarrow[f]{} & \cdot \end{array} \qquad \begin{array}{ccccc} \cdot & \xrightarrow{f_1/g} & \cdot & \xrightarrow{f_2/(g/f_1)} & \cdot \\ {\scriptstyle g}\uparrow & & \uparrow{\scriptstyle g/f_1} & & \uparrow{\scriptstyle (g/f_1)/f_2} \\ \cdot & \xrightarrow[f_1]{} & \cdot & \xrightarrow[f_2]{} & \cdot \end{array}$$

It can be checked that residuation is well-defined on the morphisms of the free category P_1^* in the sense that it is compatible with associativity and identities, and moreover it does not depend on the order in which rules are applied, see Lemma 20. In order for the definition to be well-founded, and thus always defined, we will make the following additional assumption.

▶ **Assumption 2.** There is a weight function $\omega_1 : P_1 \to \mathbb{N}$, and we still write $\omega_1 : P_1^* \to \mathbb{N}$ for its extension as morphism of category to the category corresponding to the additive monoid $(\mathbb{N}, +)$, such that for every generator $g \in P_1$ and $f \in \tilde{P}_1$, we have $\omega_1(g/f) < \omega_1(g)$.

▶ Remark. In order to simplify the presentation, we did not present the most general axiomatization for the weight function. An important point is that it induces a well-founded ordering on elements of P_1^* and satisfies properties similar to monomial orderings:

- it is compatible with composition: if $\omega_1(g) < \omega_1(g')$ then $\omega_1(h \circ g \circ f) < \omega_1(h \circ g' \circ f)$,
- identities are minimal elements: $\omega_1(\mathrm{id}) < \omega_1(f)$ for every $f \neq \mathrm{id}$; in particular, we have $\omega_1(g) < \omega_1(h \circ g \circ f)$ for $f, h \neq \mathrm{id}$.

In order to study confluence of the rewriting system provided by equational morphisms, through the use of residuals, we first introduce the following category, which allows us to consider, at the same time, both residuals g/f and f/g of two coinitial morphisms f and g.

▶ **Definition 18.** The *zig-zag presentation* associated to the presentation modulo $(P, \tilde{P}_1)$ is the presentation $Z = (Z_0, Z_1, Z_2)$ with $Z_0 = P_0$, $Z_1 = P_1 \uplus \overline{\tilde{P}_1}$ (generators in $\overline{\tilde{P}_1}$ are of the form $\overline{f} : B \to A$ for some generator $f : A \to B$ in $\tilde{P}_1$) and relations in Z_2 are of the form $g \circ \overline{f} \Rightarrow \overline{(f/g)} \circ (g/f)$ or $f \circ \overline{f} \Rightarrow \mathrm{id}_y$ for some pair of distinct coinitial generators $f : x \to y \in \tilde{P}_1$ and $g : x \to z \in P_1$.

▶ **Lemma 19.** *The rewriting system on morphisms in Z_1^* with Z_2 as rules is convergent. Given two coinitial morphisms $f : x \to y$ in $\tilde{P}_1^*$ and $g : x \to z$ in P_1^*, the normal form of $g \circ \overline{f}$ is $\overline{(f/g)} \circ (g/f)$.*

Proof. We extend the weight function of Assumption 2 to morphisms in Z_1^* by setting $\omega_1(\overline{f}) = 0$ for $\overline{f}$ in $\overline{\tilde{P}_1}$. This ensures that the rewriting system on morphisms in Z_1^* with Z_2 as rules is terminating. Moreover, because the left members of rules are of the form $g \circ \overline{f}$ with $g \in P_1$ and $\overline{f} \in \overline{\tilde{P}_1}$, there are no critical pairs, which means that the rewriting system is locally confluent and thus convergent by Newman's lemma. Given two coinitial morphisms $f : x \to y$ in $\tilde{P}_1^*$ and $g : x \to z$ in P_1^*, we prove by recurrence on $\omega_1(g \circ \overline{f})$ that the normal form of $g \circ \overline{f}$ is $\overline{(f/g)} \circ (g/f)$. ◀

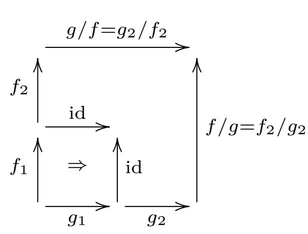
As a direct corollary of the convergence of the rewriting system, one can show that Definition 17 makes sense:

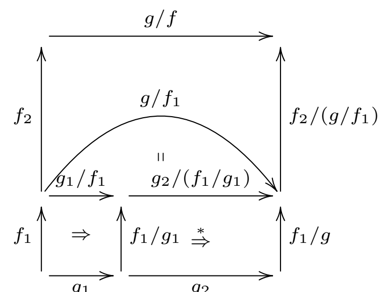
▶ **Lemma 20.** *The residuation operation does not depend on the order in which equalities of Definition 17 are applied.*

Moreover, a "global" version of the residuation property (Assumption 1) holds:

▶ **Proposition 21.** *Given two coinitial morphisms $f : x \to y$ in $\tilde{P}_1^*$ and $g : x \to z$ in P_1^*, there exists a relation $\alpha : (g/f) \circ f \overset{*}{\Leftrightarrow} (f/g) \circ g$.*

Proof. By Lemma 19, there exists a rewriting path $\beta : g \circ \overline{f} \Rightarrow \overline{(f/g)} \circ (g/f)$ in Z_2^*. By induction on its length, we can construct a relation $\alpha : (g/f) \circ f \overset{*}{\Leftrightarrow} (f/g) \circ g$ in the following way. The case where β is empty is immediate, otherwise we have $f = f_2 \circ f_1$ and $g = g_2 \circ g_1$ where f_2 is in $\tilde{P}_1^*$ (resp. g_2 in P_1^*) and f_1 is a generator in $\tilde{P}_1$ (resp. g_1 in P_1). We distinguish two cases depending on the form of the first rule of β:





If $f_1 = g_1$, i.e. the first step of β corresponds to rewriting $g_2 \circ g_1 \circ \overline{f_1} \circ \overline{f_2}$ to $g_2 \circ \overline{f_2}$ by applying the rewriting rule $f_1 \circ \overline{f_1} \Rightarrow$ id of Z_2, then by induction hypothesis, there exists a relation $\hat{\alpha}' : (g_2/f_2) \circ f_2 \overset{*}{\Leftrightarrow} (f_2/g_2) \circ g_2$. Besides, $f_2/g_2 = f/g$ and $g_2/f_2 = g/f$ which means that there exists a relation $(g/f) \circ f \overset{*}{\Leftrightarrow} (f/g) \circ g$. Otherwise $f_1 \neq g_1$, and $g_2 \circ g_1 \circ \overline{f_1} \circ \overline{f_2}$ rewrites to $g_2 \circ \overline{(f_1/g_1)} \circ (g_1/f_1) \circ \overline{f_2}$ by applying the rewriting rule $g_1 \circ \overline{f_1} \Rightarrow \overline{(f_1/g_1)} \circ (g_1/f_1)$ of Z_2. By definition of the relations in Z_2, there exists a relation $(g_1/f_1) \circ f_1 \Leftrightarrow (f_1/g_1) \circ g_1$ in P_2. Moreover, by Lemma 19, the morphism $g_2 \circ \overline{(f_1/g_1)}$ in Z_1^* rewrites to $\overline{(f_1/g)} \circ (g_2/(f_1/g_1))$, and therefore by induction hypothesis, there exists a relation $(g_2/(f_1/g_1)) \circ (f_1/g_1) \overset{*}{\Leftrightarrow} ((f_1/g_1)/g_2) \circ g_2$ in P_2^*. This means that there is a relation in P_2^*

$$(g/f_1) \circ f_1 = (g_2/(f_1/g_1)) \circ (g_1/f_1) \circ f_1 \overset{*}{\Leftrightarrow} ((f_1/g_1)/g_2) \circ g_2 \circ g_1 = (f_1/g) \circ g$$

Similarly, by lemma 19, $(g/f_1) \circ \overline{f_2}$ rewrites to $\overline{(f_2/(g/f_1))} \circ (g/f)$ by rules in Z_2, which means that there exists a relation $(g/f) \circ f_2 \overset{*}{\Leftrightarrow} (f_2/(g/f_1)) \circ (g/f_1)$ in P_2^* and therefore, there exists a relation in P_2^*:

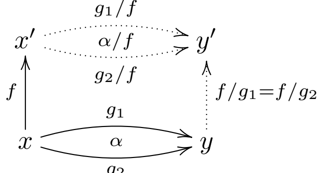
$$(g/f) \circ f = (g/f) \circ f_2 \circ f_1 \overset{*}{\Leftrightarrow} (f_2/(g/f_1)) \circ (f_1/g) \circ g = (f/g) \circ g$$

from which we conclude, as indicated in the above diagram. ◀

3.2 The cylinder property

In previous section, we have studied residuation, which enables one to recover a residual g/f of a morphism g after a coinitial equational morphism f. We now strengthen our hypothesis in order to ensure that if two morphisms are equal (wrt the equivalence generated by P_2^*) then their residuals after a same morphism are equal, i.e. equality is compatible with residuation.

▶ **Assumption 3.** The presentation $(P, \tilde{P}_1)$ satisfies the *cylinder property*: for every triple of coinitial morphisms $f : x \to x'$ in $\tilde{P}_1$ (resp. in P_1) and $g_1, g_2 : x \to y$ in P_1^* (resp. in $\tilde{P}_1^*$) such that there exists a relation $\alpha : g_1 \Leftrightarrow g_2$, we have $f/g_1 = f/g_2$ and there exists a 2-cell $g_1/f \overset{*}{\Leftrightarrow} g_2/f$. We write α/f for an arbitrary choice of such a 2-cell.



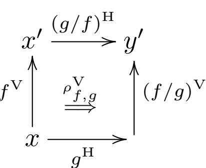
As in previous section, we would like to extend this "local" property (f and α are supposed to be generators) to a "global" one (where f and α can be composite of cells):

▶ **Proposition 22** (Global cylinder property)**.** *Given coinitial morphisms $f : x \to x'$ in $\tilde{P}_1^*$ (resp. in P_1^*) and $g_1, g_2 : x \to y$ in P_1^* (resp. in $\tilde{P}_1^*$) such that there exists a composite relation $\alpha : g_1 \overset{*}{\Leftrightarrow} g_2$, we have $f/g_1 = f/g_2$ and there exists a 2-cell $g_1/f \overset{*}{\Leftrightarrow} g_2/f$.*

The proof of previous proposition requires generalizing, in dimension 2, the termination condition (Assumption 2) and the construction of the zig-zag presentation (Definition 18).

▶ **Definition 23.** The *2-zig-zag presentation* associated to $(P, \tilde{P}_1)$ is $Y = (Y_0, Y_1, Y_2)$ with $Y_0 = P_0$, $Y_1 = P_1^{\mathrm{H}} \uplus P_1^{\mathrm{V}}$ (where the morphisms of P_1^{H} are called *horizontal* of the form $f^{\mathrm{H}} : A \to B$ for some morphism $f : A \to B$ in P_1 and similarly for the morphisms in P_1^{V} which are called *vertical*), and the 2-cells in $Y_2 = Y_2^{\mathrm{H}} \uplus Y_2^{\mathrm{V}}$ are either

- horizontal 2-cells: $Y_2^{\mathrm{H}} = P_2^{\mathrm{H}} \uplus \overline{P_2}^{\mathrm{H}}$ (i.e. relations in P_2 taken forward or backward, and decorated by H)
- vertical 2-cells: given two generators $f : x \to y$ and $g : x \to z$ in P_1 such that f or g belongs to $\tilde{P}_1$, we have a relation $\rho_{f,g}^{\mathrm{V}} : (g/f)^{\mathrm{H}} \circ f^{\mathrm{V}} \Rightarrow (f/g)^{\mathrm{V}} \circ g^{\mathrm{H}}$ in Y_2^{V}.



We consider the following rewriting system on the 2-cells in Y_2^*: for every 2-cell $\alpha : g_1 \Leftrightarrow g_2 : x \to y$ in P_2, for every coinitial 1-cell $f : x \to x'$ in P_1 such that either f or both g_1 and g_2 belong to $\tilde{P}_1^*$, there is a rewriting rule

$$((f/g_1)^{\mathrm{V}} \circ \alpha^{\mathrm{H}}) \bullet \rho^{\mathrm{V}}_{f,g_1} \quad \Rrightarrow \quad \rho^{\mathrm{V}}_{f,g_2} \bullet ((\alpha/f)^{\mathrm{H}} \circ f^{\mathrm{V}}) \tag{2}$$

where $\circ$ (resp. $\bullet$) denotes horizontal (resp. vertical) composition in a 2-category.

In order to ensure the termination of the rewriting system, we suppose the following.

▶ **Assumption 4.** There is a weight function $\omega_2 : P_2^{\mathrm{H}} \to \mathbb{N}$ such that for every $\alpha : g_1 \Rightarrow g_2$ in Y_2^* and f in P_1 such that α/f exists we have $\omega_2(\alpha/f) < \omega_2(\alpha)$. We still write $\omega_2 : \left(P_2^{\mathrm{H}} \uplus \overline{P_2}^{\mathrm{H}}\right)^* \to \mathbb{N}$ for the function such that $\omega_2(\overline{\alpha}) = \omega_2(\alpha)$ and both horizontal and vertical compositions are sent to addition ($\mathbb{N}$ being a commutative additive monoid, this definition is compatible with axioms of 2-categories, such as associativity or exchange law).

▶ **Corollary 24.** *The rewriting system* (2) *is convergent.*

Proposition 22 follows easily, by a reasoning similar to Proposition 21.

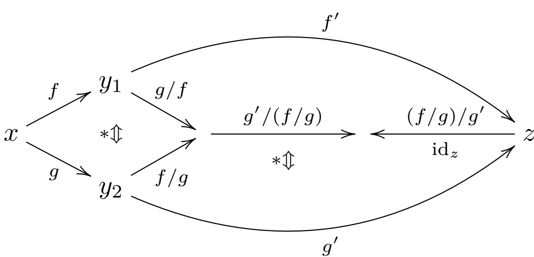
The cylinder property has many interesting consequences for the residuation operation, as we now investigate.

▶ **Proposition 25.** *In the category $\|P\|$, every equational morphism is epi.*

Proof. Suppose given $f : x \to y$ in $\tilde{P}_1^*$, and $g_1, g_2 : y \to z$ in P_1^* such that $g_1 \circ f \overset{*}{\Leftrightarrow} g_2 \circ f$. By Proposition 22, we have $g_1 = (g_1 \circ f)/f \overset{*}{\Leftrightarrow} (g_2 \circ f)/f = g_2$. ◀

▶ **Proposition 26.** *In the category $\|P\|$, every morphism g admits a pushout along a coinitial equational morphism f given by g/f.*

Proof. Suppose given $f : x \to y_1$ in $\tilde{P}_1^*$ and $g : x \to y_2$ in P_1^*. By Proposition 21, we have $(g/f) \circ f \overset{*}{\Leftrightarrow} (f/g) \circ g$ and we now show that $(g/f, f/g)$ forms a universal cocone. Suppose given $f' : y_1 \to z$ and $g' : y_2 \to z$ such that $f' \circ f \overset{*}{\Leftrightarrow} g' \circ g$.



We have $(g'/(f/g)) \circ (f/g) \overset{*}{\Leftrightarrow} ((f/g)/g') \circ g'$, where residuals exist because f/g is in $\tilde{P}_1^*$. Moreover, by applying Proposition 22 to morphism f and 2-cell $f' \circ f \overset{*}{\Leftrightarrow} g' \circ g$, we have $(f/g)/g' = f/(g' \circ g) \overset{*}{\Leftrightarrow} f/(f' \circ f) = \mathrm{id}_z$. Finally, we have $f' \circ f \overset{*}{\Leftrightarrow} g' \circ g \overset{*}{\Leftrightarrow} (g'/(f/g)) \circ (g/f) \circ f$, and by Proposition 25, we have $f' \overset{*}{\Leftrightarrow} (g'/(f/g)) \circ (g/f)$. From which we conclude. ◀

4 Comparing presented categories

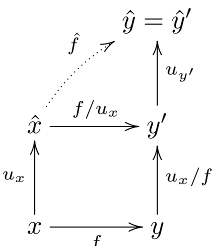
4.1 The category of normal forms

We show here that with our hypothesis on the rewriting system, the quotient category $\|P\| / \tilde{P}_1$ can be recovered as the following subcategory of $\|P\|$, whose objects are those which are in normal form for $\tilde{P}_1$.

▶ **Definition 27.** The *category of normal forms* $\|P\|{\downarrow}\tilde{P}_1$ is the full subcategory of $\|P\|$ whose objects are the normal forms of elements of P_0 wrt rules in $\tilde{P}_1$. We write $I : \|P\|{\downarrow}\tilde{P}_1 \to \|P\|$ for the inclusion functor.

▶ **Theorem 28.** *The category $\|P\|{\downarrow}\tilde{P}_1$ is (isomorphic to) the quotient category $\|P\| / \tilde{P}_1$.*

Proof. Recall that for every object $x \in \|P\|$, the associated normal form wrt rules in $\tilde{P}_1$ is denoted by $\hat{x}$, and we write $u_x : x \to \hat{x}$ for any equational morphism from x to its normal form. In particular, we always have $u_{\hat{x}} = \mathrm{id}_{\hat{x}}$. We define a functor $N : \|P\| \to \|P\|{\downarrow}\tilde{P}_1$ as the functor associating to each object x its normal form $\hat{x}$ under $\tilde{P}_1$, and to each morphism $f : x \to y$, the morphism $\hat{f} : \hat{x} \to \hat{y}$ where $\hat{f} = u_{y'} \circ (f/u_x)$ with y' being the target of f/u_x:



Notice that this definition depends on a choice of a representative in P_1^* for f, and in $\tilde{P}_1^*$ for u_x and $u_{y'}$, in the equivalence classes of morphisms modulo the relations in P_2. The global cylinder property shown in Proposition 22 ensures that the definition is independent of the choice of such representatives. Given two composable morphisms $f : x \to y$ and $g : y \to z$ we have

$$\begin{aligned} Ng \circ Nf &= u_{z'} \circ (g/u_y) \circ u_{y'} \circ (f/u_x) \\ &= u_{z'} \circ (g/(u_{y'} \circ (u_x/f))) \circ u_{y'} \circ (f/u_x) \\ &= u_{z'} \circ (g/(u_x/f))/u_{y'} \circ u_{y'} \circ (f/u_x) \\ &= u_{z'} \circ u_{y'}/(g/(u_x/f)) \circ g/(u_x/f) \circ (f/u_x) \\ &= u_{z''} \circ ((g \circ f)/u_x) \\ &= N(g \circ f) \end{aligned}$$

The image of an equational morphism $u : x \to y$ under the functor N is an identity. Namely, we have $Nu = \hat{u} = u_z \circ (u/u_x)$: since u/u_x is an equational morphism (since it is the residual of an equational morphism) whose source is a normal form, necessarily $u/u_x = \mathrm{id}_{\hat{x}}$, $z = \hat{x}$ and $u_z = \mathrm{id}_{\hat{x}}$. In particular, N preserves identities.

Suppose given a functor $F : \|P\| \to \mathcal{C}$ sending the equational morphisms to identities. In order to obtain the result, we have to show that there exits a unique functor $G : \|P\|{\downarrow}\tilde{P}_1 \to \mathcal{C}$

such that $G \circ N = F$. Writing $I : \|P\| \downarrow \tilde{P}_1 \to \|P\|$ for the inclusion functor, it is easy to show I is a section of F, i.e. $N \circ I = \mathrm{Id}_{\|P\| \downarrow \tilde{P}_1}$. Since F sends equational morphisms to identities, it is easy to check that $G \circ N = F$: given an object x, we have

$$G \circ N(x) = G(\hat{x}) = F \circ I(\hat{x}) = F(\hat{x}) = F(x)$$

the last equality, being due to the fact that $F(u_x) = \mathrm{id}_{F(\hat{x})} = \mathrm{id}_{F(x)}$, and similarly for morphisms. Finally, we check the uniqueness of G. Suppose given another functor $G' : \|P\| \downarrow \tilde{P}_1 \to \mathcal{C}$ such that $G' \circ N = F = G \circ N$. We have $G' = G' \circ N \circ I = G \circ N \circ I = G$. ◀

4.2 Equivalence with localization

We now show that the two previous constructions (quotient and normal forms) also coincide with the third possible construction which consists in formally adding inverses for equational morphisms. First, notice that we can use the description of the localization $\|P\| [\tilde{P}_1^{-1}]$ as a category of fractions given in Theorem 15:

▶ **Lemma 29.** *The set of equational morphisms of $\|P\|$ is a left calculus of fractions.*

Proof. We have to show that the set of equational morphisms satisfies the four conditions of Definition 14: the two first (closure under composition and identities) are immediate, the third one follows from Proposition 21, and the last one is ensured by the fact that all equational morphisms are epi by Proposition 25. ◀

Our proof of the equivalence is based on the embedding of the presented category into the localization provided by Proposition 16. In order for the hypothesis of this proposition to hold, we first need to impose that the same properties hold for the opposite presentation as for the presentation itself:

▶ **Assumption 5.** The presentation modulo $(P^{\mathrm{op}}, \tilde{P}^{\mathrm{op}})$ satisfies Assumptions 1, 2, 3 and 4.

This implies that the duals of previously shown properties hold for $\|P\|$. For instance, by dual of Proposition 25, all equational morphisms are mono, from which follows, by Proposition 16:

▶ **Proposition 30.** *The canonical functor $\|P\| \to \|P\| [\tilde{P}_1^{-1}]$ is faithful.*

▶ Remark. This generalizes Dehornoy's theorem [6] stating that under conditions (which are generalized here), there is an embedding of a monoid into its envelopping groupoid: by localizing wrt all morphisms rather than simply a subset of them, we recover this result. Besides, our hypothesis on relations are weaker (for instance, we only require fixed a choice of residual instead that there is only one possible choice for those).

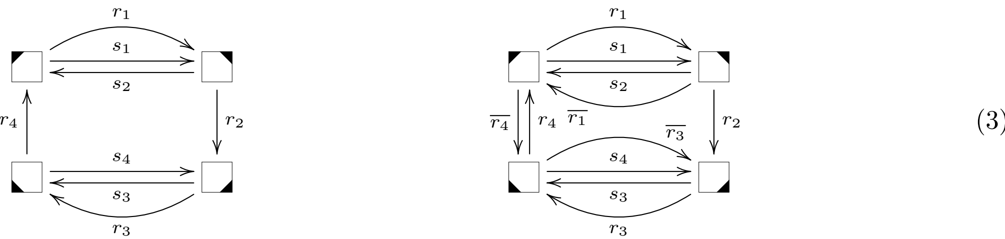
▶ **Definition 31.** A presentation modulo satisfying assumptions 1 to 5 is called *coherent.*

▶ **Theorem 32.** *Given a coherent presentation modulo $(P, \tilde{P}_1)$, the categories $\|P\| / \tilde{P}_1$ and $\|P\| [\tilde{P}_1^{-1}]$ are equivalent.*

Proof. Consider the functor $F : \|P\| \downarrow \tilde{P}_1 \to \|P\| [\tilde{P}_1^{-1}]$ defined as the composite of the inclusion functor $I : \|P\| \downarrow \tilde{P}_1 \to \|P\|$, see Definition 27, with the localization functor $L : \|P\| \to \|P\| [\tilde{P}_1^{-1}]$, see Definition 11. The functor F is faithful since it is the case for both I and L by Proposition 30. It is also full. Namely, by Theorem 15, given any two objects $\hat{x}$ and $\hat{y}$ of $\|P\| \downarrow \tilde{P}_1$, a morphism from $F(\hat{x}) = \hat{x}$ to $F(\hat{y}) = \hat{y}$ in $\|P\| [\tilde{P}_1^{-1}]$ is of the form (f, u) with $f : \hat{x} \to z$ and $u : \hat{y} \to z$ equational. Since $\hat{y}$ is a normal form, we necessarily have $u = \mathrm{id}_{\hat{y}}$ and thus $(f, u) = Ff$. Finally, given an object $y \in \|P\| [\tilde{P}_1^{-1}]$, there is a morphism $u : y \to \hat{y}$ in $\tilde{P}_1^*$ to its normal form which induces an isomorphism $y \cong \hat{y}$ in $\|P\| [\tilde{P}_1^{-1}]$. The functor F is thus an equivalence of categories. ◀

4.3 An example: the dihedral category $D_4^\bullet$

As an illustration of previous properties, we are going to study a presentation of a category which is a variant of the dihedral group. Recall that the *dihedral group* D_n is the group of isometries of the plane preserving a regular polygon with n faces. This group is generated by a rotation r of angle $2\pi/n$ and a reflection s, and can be described as the free group over the two generators r and s quotiented by the congruence generated by the three relations $s^2 = \mathrm{id}$, $r^n = \mathrm{id}$ and $rsr = s$. We consider here a variant of this group: the category $D_n^\bullet$ of isometries of the plane preserving a regular polygon with n faces together with a distinguished vertex (the category thus has n objects). For instance, the category $D_4^\bullet$ is pictured on the left below, the distinguished vertex of the square being pictured by a black triangle:



This category $D_4^\bullet$ admits a presentation P with 4 objects and 8 generating morphisms, as pictured on the left above, satisfying the 12 relations:

$$r_{i+3} \circ r_{i+2} \circ r_{i+1} \circ r_i = \mathrm{id} \qquad s_{j+1} \circ s_j = \mathrm{id} \qquad r_j \circ s_{j+1} \circ r_j = s_j$$
$$s_j \circ s_{j+1} = \mathrm{id} \qquad r_{j+3} \circ s_{j+2} \circ r_{j+1} = s_{j+1}$$

for $i \in \{1, \ldots, 4\}$ and $j \in \{1, 3\}$, where the indices are to be taken modulo 4 so that they lie in $\{1, \ldots, 4\}$.

The methodology introduced earlier can be used to show that by quotienting (resp. localizing) by $\Sigma = \{r_2, r_4\}$, we obtain a category which is isomorphic (resp. equivalent) to $D_2^\bullet$: intuitively, "forgetting" about those rotations quotients the square under symmetry wrt an horizontal axis. We thus consider the presentation modulo $(P, \tilde{P}_1)$ with $\tilde{P}_1 = \Sigma$. Unfortunately, this presentation does not satisfy the assumptions required to apply our results; for instance, there is no residual of r_2 after s_2. It is thus necessary to complete the presentation in order to have the confluence properties (namely, the residuation and cylinder properties). In rewriting theory, when a rewriting system is not confluent, one usually tries to complete it (typically using a Knuth-Bendix completion algorithm) in order for confluence to hold. Similarly, we can transform our presentation using a series of Tietze transformations (Definition 4 and Proposition 5) while preserving the same presented category, in order to obtain another presentation of the same category which satisfies the required assumptions.

We first consider the presentation P' obtained from P by adding the generator $\overline{r}_4 = r_3 \circ r_2 \circ r_1$ and its defining relation, as well as the derivable relations $r_4 \circ \overline{r}_4 = \mathrm{id}$ and $r_4 \circ \overline{r}_4 = \mathrm{id}$. We can now define r_2/s_2 as $\overline{r}_4$, if we consider $\overline{r}_4$ as an equational morphism. Fortunately, following lemma shows that we can quotient, or localize, by $\overline{r}_4$ instead of r_4, and we therefore define $\tilde{P}'_1 = \{r_2, \overline{r}_4\}$:

▶ **Lemma 33.** *Let P be a presentation of category such that there exist f and g in P_1 and two relations $f \circ g \Leftrightarrow \mathrm{id}$ and $g \circ f \Leftrightarrow \mathrm{id}$ in P_2. Let Σ be a subset of P_1 not containing f nor g. Then the quotients (resp. localizations) of $\|P\|$ by $\Sigma \uplus \{f\}$, $\Sigma \uplus \{f, g\}$, and $\Sigma \uplus \{g\}$ are isomorphic.*

In this way, we have transformed the presentation $(P,\tilde{P}_1)$ into a presentation $(P',\tilde{P}'_1)$ for which we can now define the residual r_2/s_2. Similarly, in order for all the required residual to be defined, we modify P' using Tietze transformations by adding generators $\overline{r}_1 = r_4 \circ r_3 \circ r_2$ and $\overline{r}_3 = r_2 \circ r_1 \circ r_4$ and modifying the set of relations. Finally, the presentation we end up with a presentation P'' which has 11 morphism generators $r_i, s_i, \overline{r}_k$, as shown on the right of (3), and 16 relations:

$$\begin{array}{lllll} s_{j+1} \circ s_j = \mathrm{id} & r_1 \circ s_2 \circ r_1 = s_1 & r_k \circ \overline{r}_k = \mathrm{id} & r_2 \circ r_1 = \overline{r}_3 \circ \overline{r}_4 & s_3 \circ r_2 = \overline{r}_4 \circ s_2 \\ s_j \circ s_{j+1} = \mathrm{id} & \overline{r}_3 \circ s_3 \circ \overline{r}_3 = s_4 & \overline{r}_k \circ r_k = \mathrm{id} & r_3 \circ r_2 = \overline{r}_4 \circ \overline{r}_1 & r_2 \circ s_1 = s_4 \circ \overline{r}_4 \end{array}$$

for $i \in \{1,\ldots,4\}$, $j \in \{1,3\}$ and $k \in \{1,3,4\}$, which is considered modulo $\tilde{P}''_1 = \{r_2,\overline{r}_4\}$. This presentation modulo is coherent. It satisfies convergence assumption 1, and residuals are defined by

$$r_2/s_2 = r_2/\overline{r}_1 = \overline{r}_4 \quad \overline{r}_4/s_1 = \overline{r}_4/r_1 = r_2 \quad s_1/\overline{r}_4 = s_4 \quad r_1/\overline{r}_4 = \overline{r}_3 \quad s_2/r_2 = s_3 \quad \overline{r}_1/r_2 = r_3$$

For termination assumption 2, we define ω_1 as equal to 1 on s_1, s_2, $\overline{r}_1$ and r_1 and 0 on other morphism generators. The cylinder assumption 3 follows from considering 5 diagrams. For termination assumption 4 we define ω_2 as 1 on relation generators such that the only morphism generators occurring in the source or the target are r_1, $\overline{r}_1$, s_1 or s_2, and as 0 otherwise. It can be checked similarly that $(P''^{\mathrm{op}},(\tilde{P}''_1)^{\mathrm{op}})$ satisfies the assumptions. Therefore $\|P''\|{\downarrow}\{r_2,\overline{r}_4\}$ is isomorphic to $\|P''\| \,/\, \{r_2,\overline{r}_4\}$ by Theorem 28, and equivalent to $\|P''\|\,[\{r_2,\overline{r}_4\}^{-1}]$ by Theorem 32, the left-to-right part of the equivalence being an embedding by Proposition 30. An explicit (non-modulo) presentation for the quotient can be obtained by Lemma 10, and this presentation is Tietze equivalent to the canonical presentation of $D_2^\bullet$. We finally obtain the following result:

▶ **Theorem 34.** *The category $D_2^\bullet$ is isomorphic to the quotient $D_4^\bullet/\,\{r_2,r_4\}$, embeds fully and faithfully into the category $D_4^\bullet$, and is equivalent to the localization $D_4^\bullet[\{r_2,r_4\}^{-1}]$.*

▶ Remark. In this case, since r_2 and r_4 are already invertible in $\|P\|$, we moreover have $D_4^\bullet[\{r_2,r_4\}^{-1}] \cong D_4^\bullet$.

This illustrates the fact that, even though restricted for now to categories, the tools developed in this article enable one to obtain interesting results about presented categories.

5 Towards an extension to 2-categories

We would like to briefly mention how this work can be extended to presentations of 2-categories, and thus be able to handle examples such as the presentation of the monoidal (i.e. 2-)category $\Delta \times \Delta$ described in the introduction: it should admit a presentation modulo $(P,\tilde{P}_2)$ where $P = (P_0,P_1,P_2,P_3)$ is a presentation of a 2-category and $\tilde{P}_2 \subseteq P_2$ is a set of equational 2-generators, and in particular we should be able to show that the 2-category of normal forms $\|P\|{\downarrow}\tilde{P}_2$ is isomorphic to the quotient 2-category $\|P\|\,/\tilde{P}_2$ and equivalent to the localization $\|P\|\,[\tilde{P}_2^{-1}]$.

While we leave such an extension for future work, we would like to briefly mention some of the adjustments necessary to cover this case. Firstly, since the exchange law in a 2-category ensures that two disjoint rewrites commute, it is enough to impose the existence of suitable residuals for critical pairs only (this is, in our context, a variant of Newman's lemma), and similarly the cylinder property only has to be imposed for triples of coinitial rewriting rules forming a critical triple. Secondly, since in practice not all operations (residuation for

instance in our example) are compatible with exchange law, one actually has to explicitly handle this law and work in the setting of sesquicategories. Thirdly, the precise notion of equivalence between 2-categories is subtle. For instance, the canonical “inclusion” functor $\|P\|\!\downarrow\!\tilde{P}_2 \hookrightarrow \|P\|$, exhibiting the restriction to 1-cells in normal form as a “sub-2-category” of $\|P\|$, is in fact a lax 2-functor: the 0-composition of two 1-cells in normal form is not necessarily a normal form, but always normalizes to one.

6 Conclusion

We have introduced a notion of presentation of a category modulo an “equational” rewriting system, and provided a series of reasonable coherence conditions ensuring that the equational rules are well-behaved wrt the generators. In particular, we show that, under those assumptions, all the three possible natural constructions for the presented category are equivalent. These assumptions are “local” in the sense that they are given directly on the presentations, and can thus be used in practice in order to perform computations, as illustrated in the article. In the future, we would like to investigate more applications, by studying classes of presentations (presentations of monoids and groups are well investigated, but there are fewer studied examples of presentations of categories), and also extend this work to presentations of 2-(and possibly higher-)categories.

References

1. Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge University Press, 1999.
2. Marek A Bednarczyk, Andrzej M Borzyszkowski, and Wieslaw Pawlowski. Generalized Congruences – Epimorphisms in Cat. *Theory and Applications of Categories*, 5(11):266–280, 1999.
3. Marc Bezem, Jan Willem Klop, and Roel de Vrijer. *Term rewriting systems*. Cambridge University Press, 2003.
4. Francis Borceux. *Handbook of Categorical Algebra 1. Basic Category Theory*. Encyclopedia of Mathematics and its Applications. Cambridge Univ. Press, 1994.
5. Albert Burroni. Higher-dimensional word problems with applications to equational logic. *Theoretical computer science*, 115(1):43–62, 1993.
6. Patrick Dehornoy, Francois Digne, Eddy Godelle, Daan Krammer, and Jean Michel. *Foundations of Garside theory*, volume 22 of *EMS tracts in mathematics*. European Mathematical Society, 2015.
7. Peter Gabriel and Michel Zisman. *Calculus of fractions and homotopy theory*, volume 6. Springer, 1967.
8. Yves Guiraud and Philippe Malbos. Polygraphs of finite derivation type. *arXiv:1402.2587*, 2014.
9. Yves Guiraud, Philippe Malbos, Samuel Mimram, et al. A homotopical completion procedure with applications to coherence of monoids. In *RTA-24th International Conference on Rewriting Techniques and Applications-2013*, volume 21, pages 223–238, 2013.
10. Yves Lafont. Towards an algebraic theory of boolean circuits. *Journal of Pure and Applied Algebra*, 184(2):257–310, 2003.
11. J-J Lévy. *Réductions correctes et optimales dans le lambda-calcul*. PhD thesis, Université Paris VII, 1978.
12. Saunders Mac Lane. *Categories for the working mathematician*, volume 5. springer, 1998.
13. A John Power. An n-categorical pasting theorem. In *Category theory*, pages 326–358. Springer, 1991.

14 Ross Street. Limits indexed by category-valued 2-functors. *Journal of Pure and Applied Algebra*, 8(2):149–181, 1976.

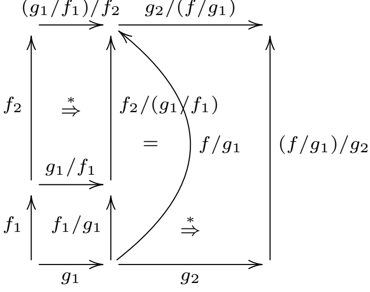
15 Heinrich Tietze. Über die topologischen invarianten mehrdimensionaler mannigfaltigkeiten. *Monatshefte für Mathematik und Physik*, 19(1):1–118, 1908.

A Omitted proofs

Proof of Lemma 10. It is enough to show that $\|P/\tilde{P}_1\|$ is a quotient of $\|P\|$ by $\tilde{P}_1$. We define a quotient functor $Q : \|P\| \to \|P/\tilde{P}_1\|$ on generators by $Q(x) = [x]$ for $x \in P_0$ and $Q(f) = f$ for $f \in P_1$ (this extends to a functor since for every 2-generator $\alpha \in P_2$ there is a corresponding 2-generator in $P/\tilde{P}_1$). For every generator $f \in \tilde{P}_1$, we immediately have $Q(f) = \mathrm{id}$. Suppose given a functor $F : \|P\| \to \mathcal{C}$ sending equational morphisms to identities. We define a functor $\tilde{F} : \|P/\tilde{P}_1\| \to \mathcal{C}$ by $\tilde{F}[x] = Fx$ for an object $[x]$ of $\|P/\tilde{P}_1\|$ (this does not depend on the choice of the representative of class) and, given $f = f_k \circ \ldots \circ f_1$ in $\|P/\tilde{P}_1\|$ with $f_i \in P_1$, we define $\tilde{F}f = Ff_k \circ \ldots \circ Ff_1$ (it can be checked that this is also well-defined). The functor $\tilde{F}$ satisfies $F = \tilde{F} \circ Q$, and it is the only such functor since it has to send elements of $\tilde{P}_1$ to identities. ◀

Proof of Lemma 12. The localization functor L is defined by $Lx = x$ for $x \in P_0$, and $Lf = f$ for $f \in P_1^*$. This functor is well-defined since for any 2-generator $\alpha : f \Rightarrow g$ in P_2, we have that $Lf = f$ and $Lg = g$, and there is a relation $f \Rightarrow g$ in P_2' by definition. Besides, for any f in Σ, $Lf = f$ is an isomorphism since $\overline{f}$ is an inverse for f. Suppose given $F : \|P\| \to \mathcal{C}$ sending the elements of Σ to isomorphisms. We define a functor $\tilde{F} : \|P'\| \to \mathcal{C}$ on the generators by $\tilde{F}x = Fx$ for $x \in P_0$, $\tilde{F}f = Ff$ for $f \in P_1$ and $\tilde{F}\overline{f} = (Ff)^{-1}$. This functor is well-defined, since for any relation $\alpha : f \Rightarrow g$ in $P_2 \subset P_2'$, we have $\tilde{F}f = Ff = Fg = \tilde{F}g$ and $\tilde{F}(f \circ \overline{f}) = Ff \circ F\overline{f} = Ff \circ (Ff)^{-1} = \mathrm{id}$ and similarly $\tilde{F}(\overline{f} \circ f) = \mathrm{id}$. This functor satisfies $\tilde{F} \circ L = F$ and is the unique such functor. ◀

Proof of Lemma 19. We extend the weight function of Assumption 2 to morphisms in Z_1^* by setting $\omega_1(\overline{f}) = 0$ for $\overline{f}$ in $\overline{\tilde{P}_1}$. This ensures that the rewriting system on morphisms in Z_1^* with Z_2 as rules is terminating. Moreover, because the left members of rules are of the form $g \circ \overline{f}$ with $g \in P_1$ and $\overline{f} \in \overline{\tilde{P}_1}$, there are no critical pairs, which means that the rewriting system is locally confluent and thus convergent by Newman's lemma. Given two coinitial morphisms $f : x \to y$ in $\tilde{P}_1^*$ and $g : x \to z$ in P_1^*, we prove by recurrence on $\omega_1(g \circ \overline{f})$ that the normal form of $g \circ \overline{f}$ is $\overline{(f/g)} \circ (g/f)$. If either f or g is an identity, this result is direct. Otherwise, $f = f_2 \circ f_1$ and $g = g_2 \circ g_1$ where f_1, f_2, g_1 and g_2 are non identity-morphisms.



By induction, we have

$$g_1 \circ \overline{f_1} \overset{*}{\Rightarrow} \overline{(f_1/g_1)} \circ (g_1/f_1) \qquad \text{and} \qquad (g_1/f_1) \circ \overline{f_2} \overset{*}{\Rightarrow} \overline{(f_2/(g_1/f_1))} \circ ((g_1/f_1)/f_2)$$

since $\omega_1((g_1/f_1)\circ\overline{f_2}) < \omega_1\left(g_2\circ\left(\overline{(f_1/g_1)}\circ(g_1/f_1)\right)\circ\overline{f}_2\right) < \omega_1(g\circ\overline{f})$. And therefore,

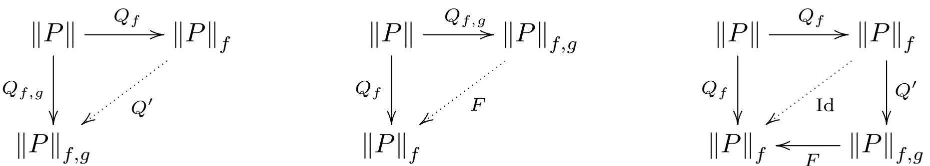
$$\begin{aligned} g\circ\overline{f} &\overset{*}{\Rightarrow} g_2\circ\left(\overline{(f_1/g_1)}\circ(g_1/f_1)\right)\circ\overline{f}_2\\ &\overset{*}{\Rightarrow} g_2\circ\overline{(f_1/g_1)}\circ\left(\overline{(f_2/(g_1/f_1))}\circ((g_1/f_1)/f_2)\right)\\ &\overset{*}{\Rightarrow} g_2\circ\overline{(f/g_1)}\circ(g_1/f)\end{aligned}$$

Similarly $\omega_1(g_2\circ\overline{(f/g_1)}) < \omega_1(g\circ\overline{f})$, therefore $\omega_1(g_2\circ\overline{(f/g_1)}) \overset{*}{\Rightarrow} \overline{((f/g_1)/g_2)}\circ(g_2/(f/g_1))$, and we have

$$\begin{aligned} g\circ\overline{f} &\overset{*}{\Rightarrow} g_2\circ\overline{(f/g_1)}\circ(g_1/f)\\ &\overset{*}{\Rightarrow} \overline{((f/g_1)/g_2)}\circ(g_2/(f/g_1))\circ(g_1/f)\\ &\overset{*}{\Rightarrow} \overline{(f/g)}\circ(g/f)\end{aligned}$$

from which we conclude. ◀

Proof of Lemma 33. The isomorphism of localizations follows from Lemma 12 and the usual proof that a morphism admits at most one inverse in a category. We now consider the case of quotient: we are going to show that the categories $\|P\|_{f,g} = \|P\|/(\Sigma\uplus\{f,g\}$ and $\|P\|_f = \|P\|/(\Sigma\uplus\{f\})$ are isomorphic. We write $Q_{f,g} : \|P\| \to \|P\|_{f,g}$ and $Q_f : \|P\| \to \|P\|_f$ for the quotient functors. By the universal property of Q_f, there exist a unique $Q' : \|P\|_f \to \|P\|_{f,g}$ such that $Q_{f,g} = Q'\circ Q_f$:



Moreover, since $Q_f(f) = \mathrm{id}$, we get that $Q_f(g) = Q_f(g)\circ Q_f(f) = Q_f(g\circ f) = \mathrm{id}$ and therefore, by the universal property of $Q_{f,g}$, there exists a unique functor $F : \|P\|_f \to \|P\|_{f,g}$ such that $Q_f = F\circ Q_{f,g}$. From these equalities, we get that $Q_f = F\circ Q'\circ Q_f$ and that $Q_{f,g} = Q'\circ F\circ Q_{f,g}$. By universal property of Q_f, the identity is the unique endofunctor of $\|P\|_f$ such that $\mathrm{id}\circ Q_f = Q_f$, and therefore $F\circ Q' = \mathrm{Id}$. Similarly, we have $Q'\circ I = \mathrm{Id}$, and therefore the categories $\|P\|_f$ and $\|P\|_{f,g}$ are isomorphic. ◀

Confluence of nearly orthogonal infinitary term rewriting systems*

Łukasz Czajka

Institute of Informatics, University of Warsaw
Banacha 2, 02-097 Warszawa, Poland
lukaszcz@mimuw.edu.pl

Abstract

We give a relatively simple coinductive proof of confluence, modulo equivalence of root-active terms, of nearly orthogonal infinitary term rewriting systems. Nearly orthogonal systems allow certain root overlaps, but no non-root overlaps. Using a slightly more complicated method we also show confluence modulo equivalence of hypercollapsing terms. The condition we impose on root overlaps is similar to the condition used by Toyama in the context of finitary rewriting.

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

Keywords and phrases infinitary rewriting, confluence, coinduction

Digital Object Identifier 10.4230/LIPIcs.RTA.2015.106

1 Introduction

Infinitary term rewriting extends term rewriting by infinite terms and transfinite reductions. This enables the consideration of "limits" of terms under infinite reduction sequences. For instance, in a term rewriting system with the rule

$$f(a) \to c(f(a))$$

the term $f(a)$ "in the limit" reduces to an infinite term c^ω such that $c^\omega = c(c^\omega)$. In fact, c^ω is the normal form of $f(a)$ in an infinitary term rewriting system (iTRS) containing the above single rule.

In this paper we show confluence modulo equivalence of root-active terms of nearly orthogonal iTRSs. This implies that nearly orthogonal iTRSs have the unique normal forms property. Nearly orthogonal iTRSs allow certain root overlaps, but no non-root overlaps. More precisely, for each root critical pair $\langle t_1, t_2 \rangle$ we require that there exists s such that $t_1 \Rightarrow s$ and $t_2 \to^\infty s$, where $\to^\infty$ is strongly convergent infinitary reduction and $\Rightarrow$ is parallel reduction. Since almost orthogonal (i.e. weakly orthogonal with no non-root overlaps) iTRSs are nearly orthogonal, this shows that the failure of the unique normal forms property in weakly orthogonal iTRSs (see [11, 10]) is due to the possibility of non-root overlaps.

Our proof method is different from [21, 22] and it is relatively simple, but it does not easily generalise to confluence modulo equivalence of hypercollapsing terms. Using a bit more complicated method similar to [21] we also prove confluence modulo equivalence of hypercollapsing terms of nearly orthogonal iTRSs. Because of space limits the details of the second proof were moved to an appendix.

Actually, confluence modulo equivalence of root-active terms follows easily from confluence modulo equivalence of hypercollapsing terms. However, the method of the proof of confluence

* Partially supported by NCN grant 2012/07/N/ST6/03398.

© Łukasz Czajka;
licensed under Creative Commons License CC-BY
26th International Conference on Rewriting Techniques and Applications (RTA'15).
Editor: Maribel Fernández; pp. 106–126
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

modulo equivalence of root-active terms is perhaps more interesting than the result itself, and than the second method which is a coinductive adaptation of [21]. The first method is also slightly simpler. For these reasons we chose to present this method in detail, despite it being less general.

In the context of finitary rewriting, confluence of non-orthogonal TRSs was studied by Huet, Toyama, Gramlich and van Oostrom in [18, 34, 17, 35, 36]. The condition we impose on root critical pairs is similar to the conditions used by Toyama. It is not possible to use conditions similar to those from the cited papers for non-root overlaps, because the unique normal forms property fails already for weakly orthogonal iTRSs [11, 10]. As a counterexample, consider the following weakly orthogonal iTRS from [11, 10].

$$P(S(x)) \to x \qquad S(P(x)) \to x$$

Then $S(P^2(S^3(P^4(\ldots))))$ has two distinct normal forms P^ω and S^ω.

1.1 Related work

Infinitary rewriting was introduced in [22]. For an introduction and a general overview see [21, 13]. A coinductive definition of infinitary reductions which corresponds to strongly convergent reductions was introduced in [15] for the infinitary lambda-calculus. The paper [12] introduces a coinductive definition of infinitary reductions in iTRSs, capturing reductions of arbitrary ordinal length. Our coinductive definition of infinitary reductions is based on [15]. Coinductive techniques in infinitary lambda-calculus were investigated in [20]. In [7] confluence, modulo equivalence of root-active terms, of infinitary lambda-calculus was proven coinductively. A simpler proof method for confluence modulo equivalence of terms with no head normal form was later found in [8]. In this paper the proof of confluence modulo equivalence of root-active terms follows a strategy similar to [8]. It also bears some similarity to the proof of the unique normal forms property of orthogonal iTRSs in [28]. The general strategy of the proof of confluence modulo equivalence of hypercollapsing terms, as well as proofs of some lemmas, are similar to [21]. Some other papers related to the methods of the present work are [1, 2, 24, 25, 26, 27, 23, 14].

2 Coinduction

In this section we give a brief explanation of coinduction as it is used in the present paper. Our style of writing coinductive proofs is perhaps not completely standard, but it is similar to how such proofs are presented in e.g. [15, 4, 31, 30, 29]. However, in contrast to some of these papers, we do not claim that our proofs are a paper presentation of proofs formalised in a proof assistant (though they could probably be formalised in such a system).

First, we give an explanation of how our proofs of existential statements should be interpreted. This is the only part that may be non-obvious to someone already well-acquainted with coinduction. Then we shall give an elementary explanation of coinduction. A reader not familiar with coinduction should perhaps skip the following example and return to it after reading the rest of this section.

▶ **Example 1.** Let T be the set of all finite and infinite terms defined coinductively by

$$T ::= V \parallel A(T) \parallel B(T, T)$$

where V is a countable set of variables, and A, B are constructors. By $x, y, \ldots$ we denote variables, and by $t, s, \ldots$ we denote elements of T. Define a binary relation $\to$ on T

coinductively by the following rules.

$$\frac{}{x \to x}\ (1) \qquad \frac{t \to t'}{A(t) \to A(t')}\ (2) \qquad \frac{s \to s' \quad t \to t'}{B(s,t) \to B(s',t')}\ (3) \qquad \frac{t \to t'}{A(t) \to B(t',t')}\ (4)$$

We want to show: for all $s, t, t' \in T$, if $s \to t$ and $s \to t'$ then there exists $s' \in T$ with $t \to s'$ and $t' \to s'$. The idea is to skolemize this statement. So we need to find a Skolem function $f : T^3 \to T$ which will allow us to prove the Skolem normal form:

($\star$) if $s \to t$ and $s \to t'$ then $t \to f(s,t,t')$ and $t' \to f(s,t,t')$.

The rules for $\to$ suggest a definition of f:

$$\begin{array}{rcl} f(x,x,x) & = & x \\ f(A(s), A(t), A(t')) & = & A(f(s,t,t')) \\ f(A(s), A(t), B(t',t')) & = & B(f(s,t,t'), f(s,t,t')) \\ f(A(s), B(t,t), A(t')) & = & B(f(s,t,t'), f(s,t,t')) \\ f(A(s), B(t,t), B(t',t')) & = & B(f(s,t,t'), f(s,t,t')) \\ f(B(s_1,s_2), B(t_1,t_2), B(t_1',t_2')) & = & B(f(s_1,t_1,t_1'), f(s_2,t_2,t_2')) \\ f(s,t,t') & = & \text{some arbitrary term if none of the above matches} \end{array}$$

The definition is guarded, so f is well-defined, i.e., there exists a unique function $f : T^3 \to T$ satisfying the above equations.

We now proceed with a coinductive proof of ($\star$). Assume $s \to t$ and $s \to t'$. If $s = t = t' = x$ then $f(s,t,t') = x$, and $x \to x$ by rule (1). If $s = A(s_1)$, $t = A(t_1)$ and $t' = A(t_1')$ with $s_1 \to t_1$ and $s_1 \to t_1'$, then by the coinductive hypothesis $t_1 \to f(s_1,t_1,t_1')$ and $t_1' \to f(s_1,t_1,t_1')$. We have $f(s,t,t') = A(f(s_1,t_1,t_1'))$. Hence $t = A(t_1) \to f(s,t,t')$ and $t = A(t_1') \to f(s,t,t')$, by rule (2). If $s = B(s_1,s_2)$, $t = B(t_1,t_2)$ and $t' = B(t_1',t_2')$, with $s_1 \to t_1$, $s_1 \to t_1'$, $s_2 \to t_2$ and $s_2 \to t_2'$, then by the coinductive hypothesis we have $t_1 \to f(s_1,t_1,t_1')$, $t_1' \to f(s_1,t_1,t_1')$, $t_2 \to f(s_2,t_2,t_2')$ and $t_2' \to f(s_2,t_2,t_2')$. Hence $t = B(t_1,t_2) \to B(f(s_1,t_1,t_1'), f(s_2,t_2,t_2')) = f(s,t,t')$ by rule (3). Analogously, $t' \to f(s,t,t')$ by rule (3). Other cases are similar.

Usually, it is inconvenient to invent the Skolem function beforehand, because the definition of the Skolem function and the coinductive proof of the Skolem normal form are typically interdependent. Therefore, we adopt an informal style of doing a proof by coinduction of a statement[1]

$$\begin{array}{rcl} \psi(R_1,\ldots,R_m) & = & \forall_{x_1,\ldots,x_n \in T} \,.\, \varphi(\vec{x}) \to \\ & & \quad \exists_{y \in T} . R_1(g_1(\vec{x}),\ldots,g_k(\vec{x}),y) \wedge \ldots \wedge R_m(g_1(\vec{x}),\ldots,g_k(\vec{x}),y) \end{array}$$

with an existential quantifier. We intertwine the corecursive definition of the Skolem function f with a coinductive proof of the Skolem normal form

$$\begin{array}{l} \forall_{x_1,\ldots,x_n \in T} \,.\, \varphi(\vec{x}) \to \\ \quad R_1(g_1(\vec{x}),\ldots,g_k(\vec{x}),f(\vec{x})) \wedge \ldots \wedge R_m(g_1(\vec{x}),\ldots,g_k(\vec{x}),f(\vec{x})) \end{array}$$

[1] Here $\varphi(\vec{x})$ is a statement/formula (whatever it means) with only $x_1,\ldots,x_n$ occuring free. We believe that for explanatory purposes it is not necessary to make this any more precise. In general, we abbreviate $x_1,\ldots,x_n$ with $\vec{x}$. The symbols $R_1,\ldots,R_m$ stand for coinductive relations on T, i.e., relations defined as greatest fixpoints of some monotone functions on the powerset of an appropriate cartesian product of T. The symbols $g_1,\ldots,g_k$ denote some functions of $\vec{x}$. The statement φ may contain $R_1,\ldots,R_m$, but their occurences in φ are not affected by substituting different relations in ψ, e.g., if $\psi(R) = \forall_{x \in T}.R(x) \to R(g(x))$ then $\psi(S) = \forall_{x \in T}.R(x) \to S(g(x))$.

We pretend that the coinductive hypothesis[2] is $\psi(R_1^\alpha, \ldots, R_m^\alpha)$. Each element obtained from the existential quantifier in the coinductive hypothesis is interpreted as a corecursive invocation of the Skolem function. When later we exhibit an element to show the existential subformula of $\psi(R_1^{\alpha+1}, \ldots, R_m^{\alpha+1})$, we interpret this as the definition of the Skolem function in the case specified by the assumptions currently active in the proof. Note that this exhibited element may (or may not) depend on some elements obtained from the existential quantifier in the coinductive hypothesis, i.e., the definition of the Skolem function may involve corecursive invocations.

To illustrate our style of doing coinductive proofs of statements with an existential quantifier, we redo the proof done above. For illustrative purposes, we indicate the arguments of the Skolem function, i.e., we write $s'_{s,t,t'}$ in place of $f(s,t,t')$. These subscripts s, t, t' are normally omitted.

We show by coinduction that if $s \to t$ and $s \to t'$ then there exists $s' \in T$ with $t \to s'$ and $t' \to s'$. Assume $s \to t$ and $s \to t'$. If $s = t = t' = x$ then take $s'_{x,x,x} = x$. If $s = A(s_1)$, $t = A(t_1)$ and $t' = A(t_1')$ with $s_1 \to t_1$ and $s_1 \to t_1'$, then by the coinductive hypothesis we obtain[3] $s'_{s_1,t_1,t_1'}$ with $t_1 \to s'_{s_1,t_1,t_1'}$ and $t_1' \to s'_{s_1,t_1,t_1'}$. Hence $t = A(t_1) \to A(s'_{s_1,t_1,t_1'})$ and $t = A(t_1') \to A(s'_{s_1,t_1,t_1'})$, by rule (2). Thus we may take $s'_{s,t,t'} = A(s'_{s_1,t_1,t_1'})$. If $s = B(s_1, s_2)$, $t = B(t_1, t_2)$ and $t' = B(t_1', t_2')$, with $s_1 \to t_1$, $s_1 \to t_1'$, $s_2 \to t_2$ and $s_2 \to t_2'$, then by the coinductive hypothesis we obtain $s'_{s_1,t_1,t_1'}$ and $s'_{s_2,t_2,t_2'}$ with $t_1 \to s'_{s_1,t_1,t_1'}$, $t_1' \to s'_{s_1,t_1,t_1'}$, $t_2 \to s'_{s_2,t_2,t_2'}$ and $t_2' \to s'_{s_2,t_2,t_2'}$. Hence $t = B(t_1, t_2) \to B(s'_{s_1,t_1,t_1'}, s'_{s_2,t_2,t_2'})$ by rule (3). Analogously, $t' \to B(s'_{s_1,t_1,t_1'}, s'_{s_2,t_2,t_2'})$ by rule (3). Thus we may take $s'_{s,t,t'} = B(s'_{s_1,t_1,t_1'}, s'_{s_2,t_2,t_2'})$. Other cases are similar.

It is quite clear that the above informal proof, when interpreted in the way outlined before, implicitly defines the Skolem function f. It should be kept in mind that in every case the definition of the Skolem function needs to be guarded. We do not explicitly mention this each time, but verifying this is part of verifying the proof.

At this point a foundationally minded reader might wonder what is exactly the coinduction principle employed in our proofs. The answer to this is simple: whichever you like. With enough patience one could, in principle, reformulate all proofs to directly employ the usual coinduction principle in set theory based on the Knaster-Tarski fixpoint theorem [33]. Since all our proofs and corecursive definitions are actually guarded, one could probably[4] formalise them in a proof assistant based on type theory with a syntactic guardedness check, e.g., in Coq [6, 16]. Perhaps the most straightforward, but maybe not the foundationally nicest, way of justifying our proofs is by reducing coinduction to transfinite induction, as outlined below.

Let T and $\to$ be as in Example 1. Formally, the relation $\to$ is the greatest fixpoint of a monotone $F : \mathcal{P}(T \times T) \to \mathcal{P}(T \times T)$ defined by

$$F(R) = \{\langle t_1, t_2\rangle \mid \exists_{x \in V}(t_1 = t_2 = x) \vee \exists_{t,t' \in T}(t_1 = A(t) \wedge t_2 = B(t', t') \wedge R(t, t')) \vee \ldots\}.$$

Alternatively, using the Knaster-Tarski fixpoint theorem, the relation $\to$ may be characterised as the greatest binary relation on T (i.e. the greatest subset of $T \times T$ w.r.t. set

---

[2] We use $R_1^\alpha, \ldots, R_m^\alpha$ for the α-approximants of the coinductive relations $R_1, \ldots, R_m$. A reader confused by this terminology should take a look at our explanation of coinduction after this example.

[3] More precisely: by corecursively applying the Skolem function to s_1, t_1, t_1' we obtain $s'_{s_1,t_1,t_1'}$, and by the coinductive hypothesis we have $t_1 \to s'_{s_1,t_1,t_1'}$ and $t_1' \to s'_{s_1,t_1,t_1'}$.

[4] The author has not paid enough attention to the type theory specific details involved in such a formalisation to claim this with complete certainty.

inclusion) such that $\to \subseteq F(\to)$, i.e., such that for every $t_1, t_2 \in T$ with $t_1 \to t_2$ one of the following holds:

1. $t_1 = t_2 = x$ for some variable $x \in V$,
2. $t_1 = A(t)$, $t_2 = A(t')$ with $t \to t'$,
3. $t_1 = B(s,t)$, $t_2 = B(s',t')$ with $s \to s'$ and $t \to t'$,
4. $t_1 = A(t)$, $t_2 = B(t',t')$ with $t \to t'$.

Yet another way to think about $\to$ is that $t_1 \to t_2$ holds if and only if there exists a *potentially infinite* derivation tree of $t_1 \to t_2$ built using the rules (1) – (4).

The rules (1) – (4) could also be interpreted inductively to yield the least fixpoint of F. This is the conventional interpretation, and it is indicated with a single line in each rule separating premises from the conclusion. A coinductive interpretation is indicated with double lines.

The greatest fixpoint $\to$ of F may be obtained by transfinitely iterating F starting with $T \times T$. More precisely, define an ordinal-indexed sequence $(\to^\alpha)_\alpha$ by:

- $\to^0 = T \times T$,
- $\to^{\alpha+1} = F(\to^\alpha)$,
- $\to^\lambda = \bigcap_{\alpha<\lambda} \to^\alpha$ for a limit ordinal λ.

Then there exists an ordinal ζ such that $\to \;=\; \to^\zeta$. Note also that $\to^\alpha \subseteq \to^\beta$ for $\alpha \geq \beta$ (we often use this fact implicitly). See e.g. [9, Chapter 8]. The relation $\to^\alpha$ is called the α-approximant of $\to$. Note that the α-approximants depend on a particular definition of $\to$ (i.e. on the function F), not solely on the relation $\to$ itself.

It is instructive to note that the coinductive rules for $\to$ may also be interpreted as giving rules for the $\alpha + 1$-approximants, for any ordinal α.

$$\frac{}{x \to^{\alpha+1} x}\ (1) \qquad \frac{t \to^\alpha t'}{A(t) \to^{\alpha+1} A(t')}\ (2) \qquad \frac{s \to^\alpha s' \quad t \to^\alpha t'}{B(s,t) \to^{\alpha+1} B(s',t')}\ (3) \qquad \frac{t \to^\alpha t'}{A(t) \to^{\alpha+1} B(t',t')}\ (4)$$

In this paper we are interested in proving by coinduction statements of the form[5]

$$\psi(R_1, \ldots, R_m) = \forall x_1 \ldots x_n . \varphi(\vec{x}) \to R_1(g_1(\vec{x}), \ldots, g_k(\vec{x})) \wedge \ldots \wedge R_m(g_1(\vec{x}), \ldots, g_k(\vec{x})).$$

Statements with an existential quantifier may be reduced to statements of this form by skolemizing, as explained in Example 1.

To prove $\psi(R_1, \ldots, R_m)$ it suffices to show by transfinite induction that $\psi(R_1^\alpha, \ldots, R_m^\alpha)$ holds for each ordinal $\alpha \leq \zeta$, where R_i^α is the α-approximant of R_i. The reader may easily check that because of the special form of ψ and the fact that R_i^0 is the full relation, the base case $\alpha = 0$ and the cases of α a limit ordinal are trivial. Hence it remains to show the inductive step for α a successor ordinal. It turns out that a coinductive proof of ψ may be interpreted as a proof of this inductive step for a successor ordinal, with the ordinals left implicit and the phrase "coinductive hypothesis" used instead of "inductive hypothesis".

▶ **Example 2.** On terms from T (see Example 1) we define the operation of substitution by guarded corecursion.

$$\begin{array}{rclcrcl} y[t/x] & = & y \quad \text{if } x \neq y & \quad & (A(s))[t/x] & = & A(s[t/x]) \\ x[t/x] & = & t & & (B(s_1,s_2))[t/x] & = & B(s_1[t/x], s_2[t/x]) \end{array}$$

[5] See footnote 1.

We show by coinduction: if $s \to s'$ and $t \to t'$ then $s[t/x] \to s'[t'/x]$, where $\to$ is the relation from Example 1. Formally, the statement we show by transfinite induction on $\alpha \le \zeta$ is: for $s, s', t, t' \in T$, if $s \to s'$ and $t \to t'$ then $s[t/x] \to^{\alpha} s'[t'/x]$. For illustrative purposes, we indicate the α-approximants with appropriate ordinal superscripts, but it is customary to omit these superscripts.

Let us proceed with the proof. The proof is by coinduction with case analysis on $s \to s'$. If $s = s' = y$ with $y \neq x$, then $s[t/x] = y = s'[t'/x]$. If $s = s' = x$ then $s[t/x] = t \to^{\alpha+1} t' = s'[t'/x]$ (note that $\to = \to^{\zeta} \subseteq \to^{\alpha+1}$). If $s = A(s_1)$, $s' = A(s_1')$ and $s_1 \to s_1'$, then $s_1[t/x] \to^{\alpha} s_1'[t'/x]$ by the coinductive hypothesis. Thus $s[t/x] = A(s_1[t/x]) \to^{\alpha+1} A(s_1'[t'/x]) = s'[t'/x]$ by rule (2). If $s = B(s_1, s_2)$, $s' = B(s_1', s_2')$ then the proof is analogous. If $s = A(s_1)$, $s' = B(s_1', s_1')$ and $s_1 \to s_1'$, then the proof is also similar. Indeed, by the coinductive hypothesis we have $s_1[t/x] \to^{\alpha} s_1'[t'/x]$, so $s[t/x] = A(s_1[t/x]) \to^{\alpha+1} B(s_1'[t'/x], s_1'[t'/x]) = s'[t'/x]$ by rule (4).

The reduction of coinduction to transfinite induction outlined here gives a simple criterion to check the correctness of coinductive proofs, using established principles. However, it is perhaps not the best way to understand coinduction intuitively. The author's intuition is that, in the context of the present paper, coinduction formalises the "and so on" arguments quite common when informally explaining proofs of properties of infinite discrete structures.[6] Such intuitions are necessarily vague and can only be shaped through experience.

One thing that remains to be explained is what guarded corecursion is, and why the equations given above define the substitution operation uniquely. However, the author hopes this part is fairly standard and well-understood. Intuitively, guardedness means that each corecursive invocation has to be fed directly as an argument to a constructor, and the result of this cannot be manipulated further.

In practice, when doing proofs by coinduction the following simple but a bit informal criteria need to be kept in mind.

- When we conclude from the coinductive hypothesis that a certain relation $R(t_1, \ldots, t_n)$ holds, this really means that only its approximant $R^{\alpha}(t_1, \ldots, t_n)$ holds. Usually, we need to infer that the next approximant $R^{\alpha+1}(s_1, \ldots, s_n)$ holds (for some other elements $s_1, \ldots, s_n$) by using $R^{\alpha}(t_1, \ldots, t_n)$ as a premise of an appropriate rule. But we cannot, e.g., inspect (do case reasoning on) $R^{\alpha}(t_1, \ldots, t_n)$, use it in any lemmas, or otherwise treat it as $R(t_1, \ldots, t_n)$.
- An element e obtained from an existential quantifier in the coinductive hypothesis is not really the element itself, but a corecursive invocation of the implicit Skolem function. Usually, we need to put it inside some constructor c, e.g. producing $c(e)$, and then exhibit $c(e)$ in the proof of an existential statement. Applying at least one constructor to e is necessary to ensure guardedness of the implicit Skolem function. But we cannot, e.g., inspect e, apply some previously defined functions to it, or otherwise treat it as if it was really given to us.
- In the proofs of existential statements, the implicit Skolem function cannot depend on the ordinal α. However, this is the case as long as we do not violate the first point, because if the ordinals are never mentioned and we do not inspect the approximants obtained from the coinductive hypothesis, then there is no way in which we could possibly introduce a dependency on α.

[6] How does one show that a Böhm tree M of a finite lambda-term does not contain β-redexes? If $M = \bot$ then it is obvious. Otherwise $M = \lambda x_1 \ldots x_n.yM_1 \ldots M_m$ is not a β-redex. And so on, we continue the argument for $M_1, \ldots, M_m$.

The above explanation of coinduction is generalised and elaborated in much more detail in [8]. Also [29] may be helpful as it gives many examples of coinductive proofs written in a style similar to the one used here. The book [33] is an elementary introduction to coinduction and bisimulation (but the proofs there are written in a different style than here). A good way of learning coinduction is by doing non-trivial coinductive proofs. Some people may initially find a proof assistant helpful for this purpose. The chapters [3, 5] explain coinduction in Coq from a practical viewpoint. A reader interested in foundational matters should also consult [19, 32] which deal with the coalgebraic approach to coinduction.

In the rest of this paper we shall freely use coinduction in the style explained above, giving routine coinductive proofs in as much (or as little) detail as it is customary with inductive proofs of analogous difficulty. After all, our aim is to prove results in infinitary rewriting, not to give a mathematically trivial coinduction tutorial. A reader not familiar with coinduction should treat the apparent difficulty of some proofs as an opportunity to learn doing non-trivial proofs by coinduction.

3 Infinitary term rewiting systems

▶ **Definition 3.** A *signature* is a set of symbols with associated arities. By $T(\Sigma)$ we denote the set of finite terms over a signature Σ. By $T^\infty(\Sigma)$ we denote the set of finite and infinite terms over Σ. We denote the set of variables by V. Formally, a term $t \in T^\infty(\Sigma)$ is a partial function from $\mathbb{N}^*$ to $\Sigma \cup V$, satisfying appropriate conditions, see e.g. [21]. A *position* is an element of $\mathbb{N}^*$. A position p is *below* q if q is a prefix of p (not necessarily proper prefix – we allow $p = q$). The subterm at a given position is defined in the standard way. See e.g. [21] for details. The set of terms $T^\infty(\Sigma)$ could also be defined coinductively, giving essentially the same thing.

A *rewrite rule* is a pair $\langle l, r\rangle \in T(\Sigma) \times T^\infty(\Sigma)$ such that l is not a variable and all variables of r are present in l. Note that we require l to be finite. An *infinitary term rewriting system* (iTRS) is a pair $\mathcal{S} = \langle \Sigma, S\rangle$ where Σ is a signature and S a set of rewrite rules. We often confuse $\mathcal{S}$ with S. A *substitution* is a function from V to $T^\infty(\Sigma)$. A substitution σ is extended to a function $\sigma^* : T^\infty(\Sigma) \to T^\infty(\Sigma)$ coinductively.

$$\begin{aligned} \sigma^*(x) &= \sigma(x) \\ \sigma^*(f(t_1, \ldots, t_n)) &= f(\sigma^*(t_1), \ldots, \sigma^*(t_n)) \end{aligned}$$

We often confuse σ^* with σ.

In what follows by a term we mean a member of $T^\infty(\Sigma)$, unless otherwise qualified. We use $=$ to denote identity of terms. Unless otherwise stated, we use $t, s, r, \ldots$ for terms, and $x, y, \ldots$ for variables, and $f, g, \ldots$ for symbols in Σ, and $\sigma, \sigma', \ldots$ for substitutions.

Let S be an iTRS. A term t is an S-*redex* by a rule $\langle l, r\rangle \in S$ with substitution σ, and s is its S-*reduct*, if $\sigma(l) = t$ and $\sigma(r) = s$. We define the relation $\overline{S} \subseteq T^\infty(\Sigma) \times T^\infty(\Sigma)$ by: $\langle t, s\rangle \in \overline{S}$ iff t is an S-redex and s its S-reduct. The *compatible closure* $\to_S$ of S is defined inductively by the following rules.

$$\frac{\langle t, s\rangle \in \overline{S}}{t \to_S s} \qquad \frac{t \to_S s}{f(t_1, \ldots, t_{k-1}, t, t_{k+1}, \ldots, t_n) \to_S f(t_1, \ldots, t_{k-1}, s, t_{k+1}, \ldots, t_n)}$$

By $\to_S^*$ we denote the transitive-reflexive closure of $\to_S$, and by $\to_S^=$ the reflexive closure of $\to_S$. The *parallel closure* $\Rightarrow_S$ of S is defined coinductively.

$$\frac{\langle t, s\rangle \in \overline{S}}{\overline{t \Rightarrow_S s}} \qquad \overline{\overline{x \Rightarrow_S x}} \qquad \frac{t_i \Rightarrow_S t_i' \text{ for } i = 1, \ldots, n}{\overline{f(t_1, \ldots, t_n) \Rightarrow_S f(t_1', \ldots, t_n')}}$$

Given a set $\mathcal{U}$ of terms we define the relation $\sim_{\mathcal{U}}$ analogously to parallel closure except that in the premise of the first rule we use $t, s \in \mathcal{U}$. The *infinitary closure* $\to_S^\infty$ of S is defined coinductively by the following rules.

$$\frac{t \to_S^* x}{t \to_S^\infty x} \qquad \frac{t \to_S^* f(t_1, \ldots, t_n) \quad t_i \to_S^\infty t_i' \text{ for } i = 1, \ldots, n}{t \to_S^\infty f(t_1', \ldots, t_n')}$$

We define $\to_S^{2\infty}$ in an analogous way to $\to_S^\infty$ except that in the first premise of the second rule we use $t \to_S^\infty f(t_1, \ldots, t_n)$.

A term l is *linear* if no variable occurs in l more than once. A rule $\langle l, r \rangle \in S$ is left-linear if l is linear. An iTRS S is left-linear if every rule in S is left-linear. Two rules $\langle l_1, r_1 \rangle$ and $\langle l_2, r_2 \rangle$ *overlap* if l_1 unifies with a non-variable subterm of l_2, or vice versa. We say that $\langle l_1, r_1 \rangle$ and $\langle l_2, r_2 \rangle$ *overlap at the root*, or that they form a *root overlap*, when l_1 unifies with l_2. An iTRS S is *nearly orthogonal* if it is left-linear, there are no non-root overlaps and for all rules $\langle l_1, r_1 \rangle, \langle l_2, r_2 \rangle \in S$ overlapping at the root there is s such that $\sigma(r_1) \to_S^\infty s$ and $\sigma(r_2) \Rightarrow_S s$, where σ is the mgu of l_1 and l_2 (note that this also implies that there is s' with $\sigma(r_1) \Rightarrow_S s'$ and $\sigma(r_2) \to_S^\infty s'$).

A rule $\langle l, r \rangle \in S$ is *collapsing* if r is a variable. A term t is a *collapsing redex* if it is a redex by a collapsing rule. A term t is *collapse-stable* if there is no collapsing redex s with $t \to_S^\infty s$. A term t is *hypercollapsing* if there is no collapse-stable s with $t \to_S^\infty s$. In other words, t is hypercollapsing if for every s with $t \to_S^\infty s$ there is a collapsing redex u such that $s \to_S^\infty u$. A term t is *root-stable* if there is no redex s with $t \to_S^\infty s$. A term t is *root-active* if there is no root-stable s with $t \to_S^\infty s$. By $\mathcal{H}$ we denote the set of hypercollapsing terms, and by $\mathcal{R}$ the set of root-active terms. Let $\mathcal{U}$ be a set of terms. An iTRS S is *confluent modulo* $\sim_{\mathcal{U}}$ when the following condition holds: if $t \sim_{\mathcal{U}} s$, $t \to_S^\infty t'$ and $s \to_S^\infty s'$ then there exist t'', s'' such that $t'' \sim_{\mathcal{U}} s''$, $t' \to_S^\infty t''$ and $s' \to_S^\infty s''$. An iTRS S has the *unique normal forms property* when the following condition holds: if $t \to_S^\infty t'$, $t \to_S^\infty t''$ and t', t'' are in normal form, then $t' = t''$.

The relation $\to_S$ is often called the *contraction relation* of S, and $\to_S^*$ the *reduction relation*. A *root contraction* is a contraction $t \to_S s$ such that t is the contracted redex. A *collapsing contraction* is a contraction of a collapsing redex.

The standard notion of an infinitary reduction is that of a strongly convergent reduction. In an appendix we prove that for left-linear iTRSs our coinductive definition corresponds, in the sense of existence, to strongly convergent reductions. The proof of this fact is a straightforward adaptation of [15, Theorem 3]. As a side-effect, this also yields a proof of the Compression Lemma for left-linear iTRSs.

▶ **Example 4.** Let $t_1 = A(B(t_1))$ and $t_2 = A(C(t_2))$. The following is an example of a nearly orthogonal iTRS. By capital letters we denote function symbols.

$$Z \to t_1 \qquad Z \to t_2 \qquad B(x) \to C(x) \qquad C(x) \to B(x) \qquad B(x) \to x \qquad C(x) \to x$$

Let $s_1 = M(A, s_1)$ and $s_2 = M(B, s_2)$. Here is another example of a nearly orthogonal iTRS.

$$Z \to s_1 \qquad Z \to s_2 \qquad A \to C \qquad B \to C$$

The following iTRS is *not* nearly orthogonal.

$$Z \to t_1 \qquad Z \to t_2 \qquad B(x) \to A(x) \qquad C(x) \to A(x)$$

Neither is this one: $\quad Z \to t_1 \qquad Z \to t_2 \qquad B(x) \to C(x).$

The standard counterexample shows that it is not sufficient to require joinability of root critical pairs: $\quad A \to B \qquad B \to A \qquad A \to C \qquad B \to D.$

The following simple lemma will often be used implicitly.

▶ **Lemma 5.** *Let P be a binary relation on terms, S an iTRS, and $\mathcal{U}$ a set of terms. Then the following conditions hold for all terms t, s, s':*
1. *$t \to_P^\infty t$ and $t \sim_{\mathcal{U}} t$,*
2. *if $t \to_P^* s \to_P^\infty s'$ then $t \to_P^\infty s'$,*
3. *if $t \to_P^* s$ then $t \to_P^\infty s$,*
4. *if $t \Rightarrow_P s$ then $t \to_P^\infty s$,*
5. *if $t \sim_{\mathcal{U}} s$ then $s \sim_{\mathcal{U}} t$,*
6. *if $t \to_S s$ (respectively $t \Rightarrow_S s$ or $t \to_S^\infty s$) then $\sigma(t) \to_S \sigma(s)$ (respectively $\sigma(t) \Rightarrow_S \sigma(s)$ or $\sigma(t) \to_S^\infty \sigma(s)$),*
7. *if $\sigma(x) \to_P^\infty \sigma'(x)$ (respectively $\sigma(x) \Rightarrow_P \sigma'(x)$) for all variables x, then $\sigma(t) \to_P^\infty \sigma'(t)$ (respectively $\sigma(t) \Rightarrow_P \sigma'(t)$).*

Proof. The first point follows by coinduction. The second point follows by case analysis on $s \to_P^\infty s'$. The third point follows from the previous two. Points 4–6 follow by coinduction. The last point follows by coinduction with case analysis on t. Note that the last point does not hold with $\to_P^*$ instead of $\to_P^\infty$, because t may contain infinitely many variables. ◀

4 Confluence

Our aim is to prove the following theorem.

▶ **Theorem 34** (Confluence modulo $\sim_{\mathcal{R}}$ of nearly orthogonal iTRSs)**.**
Let S be a nearly orthogonal iTRS. If $t \sim_{\mathcal{R}} s$, $t \to_S^\infty t'$ and $s \to_S^\infty s'$ then there exist t'', s'' such that $t' \to_S^\infty t''$, $s' \to_S^\infty s''$ and $t'' \sim_{\mathcal{R}} s''$.

Because $\sim_{\mathcal{R}}$ commutes with $\to_S^\infty$ (Lemma 18) and $\sim_{\mathcal{R}}$ is transitive (Lemma 20), it suffices to prove the theorem in the case $t = s$. The general strategy of the proof is illustrated in Figure 1. We show that for every term s' there exists a term w such that $s' \rightsquigarrow_s w$, i.e., s' reduces to s via a certain "standard" auxiliary "normalizing" reduction which disregards root-active subterms (see Definition 25 and Lemma 30). In contrast to infinitary N-reductions from [8], this "standard" reduction need not be unique and it is not really normalizing, but it is "regular" enough to show that it commutes with $\to_S^\infty$ (Lemma 33). The "normal" forms obtained through $\rightsquigarrow_s$ are not really in normal form, but they are closely related to Böhm trees. They may differ only in root-active subterms. Our overall proof strategy is partly similar to the strategy for the proof of the unique normal forms property of orthogonal iTRSs in [28].

Subdiagram (1) in Figure 1 is obtained by showing that $\to_S^\infty$ may be prepended to $\rightsquigarrow_s$ (Corollary 29), i.e., if $t \to_S^\infty s' \rightsquigarrow_s w$ then $t \rightsquigarrow_s w$. Subdiagram (2) follows from commutation of $\to_S^\infty$ and $\rightsquigarrow_s$ (Lemma 33). Subdiagrams (3) and (4) follow from the fact that $\rightsquigarrow_s$ decomposes into $\to^\infty$ and $\sim_{\mathcal{R}}$ (Lemma 31). Subdiagram (5) follows from the commutation of $\to_S^\infty$ and $\sim_{\mathcal{R}}$ (Lemma 18).

We also give a proof of confluence modulo $\sim_{\mathcal{H}}$. In this case a different and a bit more complicated method similar to [21] is necessary. Initially, we show some lemmas used in both proofs. In the rest of this section we fix a nearly orthogonal iTRS $\mathcal{S} = \langle \Sigma, S \rangle$. We write $\to$, $\Rightarrow$, $\to^\infty$, etc., for $\to_S$, $\Rightarrow_S$, $\to_S^\infty$, etc.

▶ **Lemma 6.** *Suppose l is finite and linear. If $t \to^\infty \sigma(l)$ then there is a substitution σ' such that $t \to^* \sigma'(l)$ and $\sigma'(x) \to^\infty \sigma(x)$ for all variables x.*

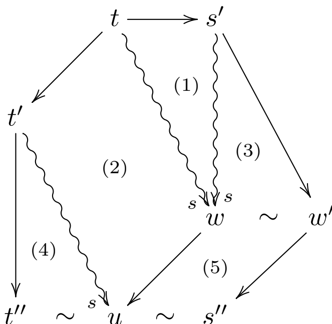


Figure 1 Confluence modulo $\sim_{\mathcal{R}}$ of nearly orthogonal iTRSs.

Proof. This follows from the definition of $\to^\infty$ and the fact that l is finite and linear. Indeed, we just need to go deep enough in the derivation tree of $t \to^\infty \sigma(l)$ to get below variable positions of l, concatenating the $\to^*$ prefixes along the way. ◀

Note that the finiteness of l is crucial in the above lemma. As a counterexample consider $l = A^\omega$, $t = B$ and an iTRS with a single rule $B \to A(B)$.

▶ **Lemma 7.** *If $t \to^\infty s \to u$ then $t \to^\infty u$.*

Proof. By coinduction. If $s = x$ then $u = x$ and thus $t \to^\infty u$. Otherwise $s = f(s_1, \ldots, s_n)$. First assume that s is the redex contracted in $s \to u$. Suppose the contraction is by a rule $\langle l, r\rangle \in S$ with substitution σ. By Lemma 6 there is σ' with $t \to^* \sigma'(l)$ and $\sigma'(x) \to^\infty \sigma(x)$ for all variables x. Then $t \to^* \sigma'(l) \to \sigma'(r) \to^\infty \sigma(r) = u$. Hence $t \to^\infty u$.

So assume that $s \to u$ is not a root contraction. Then $u = f(s_1, \ldots, s_{k-1}, s'_k, s_{k+1}, \ldots, s_n)$ with $s_k \to s'_k$. Also $t \to^* f(t_1, \ldots, t_n)$ with $t_i \to^\infty s_i$ for $i = 1, \ldots, n$. By the coinductive hypothesis $t_k \to^\infty s'_k$. Hence $t \to^\infty u$. ◀

▶ **Lemma 8.** *If $t \to^\infty s \to^\infty u$ then $t \to^\infty u$.*

Proof. By coinduction. If $u = x$ then $t \to^\infty s \to^* u$, so $t \to^\infty u$ by Lemma 7. Otherwise $u = f(u_1, \ldots, u_n)$, $s \to^* f(s_1, \ldots, s_n)$ and $s_i \to^\infty u_i$ for $i = 1, \ldots, n$. By Lemma 7 we have $t \to^\infty f(s_1, \ldots, s_n)$. Thus $t \to^* f(t_1, \ldots, t_n)$ with $t_i \to^\infty s_i$ for $i = 1, \ldots, n$. By the coinductive hypothesis $t_i \to^\infty u_i$. Therefore $t \to^\infty f(u_1, \ldots, u_n) = u$. ◀

▶ **Corollary 9.** *If $t \in \mathcal{H}$ (resp. $t \in \mathcal{R}$, t is collapse-stable, t is root-stable) and $t \to^\infty t'$ then $t' \in \mathcal{H}$ (resp. $t' \in \mathcal{R}$, t' is collapse-stable, t' is root-stable).*

▶ **Corollary 10.** *If $t \Rightarrow^* s$ then $t \to^\infty s$.*

▶ **Lemma 11.** *If $t \to^{2\infty} s$ then $t \to^\infty s$.*

Proof. By coinduction, using Lemma 8. ◀

Note that the proofs of the above lemmas depend only on the left-linearity of S. The next lemma is crucial in our proof. It fails for weakly orthogonal iTRSs. As a counterexample consider the weakly orthogonal iTRS from the introduction and the term $P(S(P^\omega))$.

▶ **Lemma 12.** *If $\langle l, r\rangle \in S$ and $\sigma(l) \to t$ by a non-root contraction, then t is a redex by $\langle l, r\rangle$ with a substitution σ' such that $\sigma(x) \to^= \sigma'(x)$ for every variable x.*

Proof. The contraction $\sigma(l) \to t$ must occur below a variable position of l, because there are no non-root overlaps. Because S is left-linear, t is still a redex by $\langle l, r\rangle$, with a substitution σ' satisfying the requirements of the lemma. ◀

▶ **Lemma 13.** *If $f(t_1, \ldots, t_n)$ is a redex by a rule $\langle f(l_1, \ldots, l_n), r\rangle \in S$ with substitution σ, and $t_i \to^\infty t_i'$ (respectively $t_i \Rightarrow t_i'$) for $i = 1, \ldots, n$, then there is σ' such that $\sigma'(l_i) = t_i'$ for $i = 1, \ldots, n$, and $\sigma(x) \to^\infty \sigma'(x)$ (respectively $\sigma(x) \Rightarrow \sigma'(x)$) for every variable x.*

Proof. This follows from left-linearity and the fact that there are no non-root overlaps: all contractions in $t_i \to^\infty t_i'$ (respectively $t_i \Rightarrow t_i'$) must occur below variable positions of l_i. Formally, one applies the definition of $t_i \to^\infty t_i'$ (respectively $t_i \Rightarrow t_i'$) repeatedly until one reaches variable positions of l_i, using Lemma 12 to show that the contractions in the $\to^*$ prefixes occur below variable positions. ◀

▶ **Lemma 14.** *If $t \Rightarrow t_1$ and $t \Rightarrow t_2$ then there is s with $t_1 \Rightarrow s$ and $t_2 \to^\infty s$.*

Proof. By coinduction. If t is a redex and t_1, t_2 are both its reducts, then there is s with $t_2 \to^\infty s$ and $t_1 \Rightarrow s$, because S is nearly orthogonal. Suppose $t = f(u_1, \ldots, u_n)$ is a redex by a rule $\langle l, r\rangle \in S$ with substitution σ, but $t_2 = f(w_1, \ldots, w_n)$ with $u_i \Rightarrow w_i$ for $i = 1, \ldots, n$. By Lemma 13 there is σ' with $\sigma'(l) = t_2$ and $\sigma(x) \Rightarrow \sigma'(x)$ for every variable x. Then $t_1 = \sigma(r) \Rightarrow \sigma'(r)$ and $t_2 = \sigma'(l) \to \sigma'(r)$, so we may take $s = \sigma'(r)$. The remaining cases, when neither $t \Rightarrow t_1$ nor $t \Rightarrow t_2$ contracts at the root, are trivial or follow directly from the coinductive hypothesis. ◀

▶ **Lemma 15** (Infinitary Parallel Moves Lemma)**.** *If $t \to^\infty t_1$ and $t \Rightarrow t_2$ then there is s with $t_1 \Rightarrow s$ and $t_2 \to^\infty s$.*

Proof. By coinduction we show that if $t \to^\infty t_1$ and $t \Rightarrow t_2$ then there is s with $t_1 \Rightarrow s$ and $t_2 \to^{2\infty} s$. This suffices by Lemma 11. If $t_1 = x$ then $t \to^* t_1$ and the claim follows from Lemma 14 and Lemma 8. Otherwise $t \to^* u = f(u_1, \ldots, u_n)$, $t_1 = f(w_1, \ldots, w_n)$ and $u_i \to^\infty w_i$ for $i = 1, \ldots, n$. By Lemma 14 and Lemma 8 there is t_2' with $t_2 \to^\infty t_2'$ and $u \Rightarrow t_2'$. If $u \Rightarrow t_2'$ is a root contraction by a rule $\langle l, r\rangle \in S$ with substitution σ, then by Lemma 13 there is σ' with $t_1 = \sigma'(l)$ and $\sigma(x) \to^\infty \sigma'(x)$ for all variables x. Then $t_1 = \sigma'(l) \to \sigma'(r)$ and $t_2 \to^\infty t_2' = \sigma(r) \to^\infty \sigma'(r)$, so $t_2 \to^\infty \sigma'(r)$ by Lemma 8. Thus we may take $s = \sigma'(r)$. If $u \Rightarrow t_2'$ does not contract at the root, then $t_2' = f(v_1, \ldots, v_n)$ with $u_i \Rightarrow v_i$ for $i = 1, \ldots, n$. By the coinductive hypothesis we obtain $s_1, \ldots, s_n$ with $v_i \to^{2\infty} s_i$ and $w_i \Rightarrow s_i$ for $i = 1, \ldots, n$. Take $s = f(s_1, \ldots, s_n)$. Then $t_2 \to^{2\infty} s$, because $t_2 \to^\infty f(v_1, \ldots, v_n)$, and $t_1 = f(w_1, \ldots, w_n) \Rightarrow s$. ◀

In the following lemmas $\mathcal{U}$ stands for either $\mathcal{H}$ or $\mathcal{R}$. We say that a term t is *active* if $t \in \mathcal{U}$. We say that a term is *stable* if it is collapse-stable and $\mathcal{U} = \mathcal{H}$, or it is root-stable and $\mathcal{U} = \mathcal{R}$. An *active redex* is a collapsing redex if $\mathcal{U} = \mathcal{H}$, or just a redex if $\mathcal{U} = \mathcal{R}$. Note that:

- t is stable iff there is no active redex s with $t \to^\infty s$,
- t is active ($t \in \mathcal{U}$) iff there is no stable s with $t \to^\infty s$,
- t is active iff for every s with $t \to^\infty s$ there is an active redex s' with $s \to^\infty s'$.

▶ **Lemma 16.** *If $\langle l, r\rangle \in S$ and $s \in \mathcal{U}$ is a proper subterm of $\sigma(l)$, then s occurs in $\sigma(l)$ below a variable position of l.*

Proof. Since $s \in \mathcal{U}$, by Lemma 6 there is a redex u such that $s \to^* u$. Because s is a proper subterm of $\sigma(l)$, by Lemma 12 the term t, which is $\sigma(l)$ with the subterm s replaced with u, is a redex by $\langle l, r\rangle$. But then u (and thus also s) must occur below a variable position of l,

because otherwise there would be a non-root overlap between $\langle l, r\rangle$ and the rule by which u is a redex. ◀

▶ **Lemma 17.** *If $t \to t'$ and $t \sim_{\mathcal{U}} s$ then there is s' with $s \to^{=} s'$ and $t' \sim_{\mathcal{U}} s'$.*

Proof. Induction on $t \to t'$. If $t, s \in \mathcal{U}$ then also $t' \in \mathcal{U}$, so $t' \sim_{\mathcal{U}} s$ and we may take $s' = s$. If $t = x$ then $t' = s' = x$. Otherwise $t = f(t_1, \ldots, t_n)$, $s = f(s_1, \ldots, s_n)$ and $t_i \sim_{\mathcal{U}} s_i$ for $i = 1, \ldots, n$. First assume that $t \to t'$ is a root contraction, i.e., there are $\langle l, r\rangle \in S$ and σ such that $\sigma(l) = t$ and $\sigma(r) = t'$. By Lemma 16 all proper active subterms of t are below variable positions of l. This implies that $s = \sigma'(l)$ with some σ' such that $\sigma(x) \sim_{\mathcal{U}} \sigma'(x)$ for every variable x. Then $s = \sigma'(l) \to \sigma'(r) \sim_{\mathcal{U}} \sigma(r) = t'$. Therefore we may take $s' = \sigma'(r)$. If $t \to t'$ is not a root contraction then the claim follows directly from the inductive hypothesis. ◀

▶ **Lemma 18.** *If $t \to^{\infty} t'$ and $t \sim_{\mathcal{U}} s$ then there is s' with $s \to^{\infty} s'$ and $t' \sim_{\mathcal{U}} s'$.*

Proof. By coinduction. If $t' = x$ then $t \to^{*} x$ and the claim follows from Lemma 17. Otherwise $t \to^{*} u = f(u_1, \ldots, u_n)$, $t' = f(t'_1, \ldots, t'_n)$ and $u_i \to^{\infty} t'_i$ for $i = 1, \ldots, n$. By Lemma 17 there is u' with $s \to^{*} u'$ and $u \sim_{\mathcal{U}} u'$. If $u, u' \in \mathcal{U}$ then $t' \in \mathcal{U}$ by Corollary 9, because $u \to^{\infty} t'$. Hence $t' \sim_{\mathcal{U}} u'$ and we may take $s' = u'$. Otherwise $u' = f(u'_1, \ldots, u'_n)$ with $u_i \sim_{\mathcal{U}} u'_i$ for $i = 1, \ldots, n$. By the coinductive hypothesis we obtain s_i with $u'_i \to^{\infty} s_i$ and $t'_i \sim_{\mathcal{U}} s_i$, for $i = 1, \ldots, n$. Take $s' = f(s_1, \ldots, s_n)$. Then $t' = f(t'_1, \ldots, t'_n) \sim_{\mathcal{U}} s'$ and $s \to^{\infty} s'$. ◀

▶ **Lemma 19.** *If $t \in \mathcal{U}$ and $t \sim_{\mathcal{U}} s$ then $s \in \mathcal{U}$.*

Proof. Assume $s \to^{\infty} s'$. Then by Lemma 18 there is t' with $t \to^{\infty} t' \sim_{\mathcal{U}} s'$. Because $t \in \mathcal{U}$, there is an active redex t'' such that $t' \to^{\infty} t''$. By Lemma 18 there is s'' such that $s' \to^{\infty} s'' \sim_{\mathcal{U}} t''$. If $t'', s'' \in \mathcal{U}$ then there is another active redex u with $s' \to^{\infty} s'' \to^{\infty} u$, so $s' \to^{\infty} u$ by Lemma 8, and thus s' is not stable. Otherwise $t'' = f(t_1, \ldots, t_n)$, $s'' = f(s_1, \ldots, s_n)$ and $t_i \sim_{\mathcal{U}} s_i$ for $i = 1, \ldots, n$. Since t'' is an active redex, there is a rule $\langle l, r\rangle \in S$ and a substitution σ with $\sigma(l) = t''$. By Lemma 16 all proper active subterms of t'' are below variable positions of l. This implies that s'' is also an active redex by the rule $\langle l, r\rangle$. Hence s' is not stable. Since s' was arbitrary with $s \to^{\infty} s'$, we conclude that $s \in \mathcal{U}$. ◀

▶ **Lemma 20.** *If $t \sim_{\mathcal{U}} s \sim_{\mathcal{U}} u$ then $t \sim_{\mathcal{U}} u$.*

Proof. By coinduction, using Lemma 19 when $t, s \in \mathcal{U}$ or $s, u \in \mathcal{U}$. ◀

▶ **Corollary 21.** *The relation $\sim_{\mathcal{U}}$ is an equivalence relation.*

We write $t \to_{\text{ncr}} s$ if $t \to s$ and this is not a collapsing contraction at the root. So $t \to^{*}_{\text{ncr}} s$ if $t \to^{*} s$ and there are no collapsing root contractions in the reduction.

▶ **Lemma 22.** *If $f(t_1, \ldots, t_n) \to^{*}_{\text{ncr}} s$ and $t_i \to^{\infty} t'_i$ for $i = 1, \ldots, n$ then $s = g(s_1, \ldots, s_m)$ and there are $s'_1, \ldots, s'_m$ with $f(t'_1, \ldots, t'_n) \Rightarrow^{*} g(s'_1, \ldots, s'_m)$ and $s_j \to^{\infty} s'_j$ for $j = 1, \ldots, m$.*

Proof. Let $t = f(t_1, \ldots, t_n)$. It suffices to consider the case $t \to_{\text{ncr}} s$. The general case then follows by induction.

If t is the redex contracted in $t \to_{\text{ncr}} s$ then it is contracted by some non-collapsing rule $\langle l, r\rangle \in S$ with substitution σ. Then $r = g(r_1, \ldots, r_m)$. By Lemma 13 there is σ' such that $f(t'_1, \ldots, t'_i) = \sigma'(l)$ and $\sigma(x) \to^{\infty} \sigma'(x)$ for every variable x. Thus $\sigma(r_j) \to^{\infty} \sigma(r'_j)$ for $j = 1, \ldots, m$. Since $s = \sigma(r) = g(\sigma(r_1), \ldots, \sigma(r_m)) \to^{\infty} g(\sigma'(r_1), \ldots, \sigma'(r_m))$ and $f(t'_1, \ldots, t'_i) = \sigma'(l) \to \sigma'(r) = g(\sigma'(r_1), \ldots, \sigma'(r_m))$, we may take $s'_j = \sigma'(r_j)$ for $j = 1, \ldots, m$.

So assume the contraction $t \to_{\mathrm{ncr}} s$ does not occur at the root. Then $s = f(s_1, \ldots, s_n)$ with $t_i \to^{=} s_i$ for $i = 1, \ldots, n$. By Lemma 15 there are $s_1', \ldots, s_n'$ with $t_i' \Rightarrow s_i'$ and $s_i \to^{\infty} s_i'$ for $i = 1, \ldots, n$. Then $f(t_1', \ldots, t_n') \Rightarrow f(s_1', \ldots, s_n')$, so we may take $g = f$ and $m = n$. ◀

▶ **Lemma 23.** *If for every s with $t \to^* s$ there is an active redex u with $s \to^{\infty} u$, then $t \in \mathcal{U}$.*

Proof. Assume t satisfies the antecedent and $t \to^{\infty} t'$. Then $t \to^* f(t_1, \ldots, t_n)$, $t' = f(t_1', \ldots, t_n')$ and $t_i \to^{\infty} t_i'$ for $i = 1, \ldots, n$. By assumption and Lemma 6 there is an active redex $s = g(s_1, \ldots, s_m)$ such that $f(t_1, \ldots, t_n) \to_{\mathrm{ncr}}^* s$. By Lemma 22 there are $s_1', \ldots, s_m'$ such that $t' = f(t_1', \ldots, t_n') \Rightarrow^* g(s_1', \ldots, s_m')$ and $s_j \to^{\infty} s_j'$ for $j = 1, \ldots, m$. Let $s' = g(s_1', \ldots, s_m')$. By Lemma 13 we conclude that s' is a redex by the same rule as s, i.e., s' is an active redex. By Corollary 10 we have $t' \to^{\infty} s'$. Hence t' is not stable. Since t' was arbitrary with $t \to^{\infty} t'$, we conclude that $t \in \mathcal{U}$. ◀

▶ **Lemma 24.** *If $t \to^{\infty} t'$ and $t' \in \mathcal{U}$ then $t \in \mathcal{U}$.*

Proof. Suppose $t \to^* s$. By Lemma 15 there is s' with $s \to^{\infty} s'$ and $t' \Rightarrow^* s'$. We have $t' \to^{\infty} s'$ by Corollary 10. Since also $t' \in \mathcal{U}$, there is an active redex u with $s \to^{\infty} s' \to^{\infty} u$. Then $s \to^{\infty} u$ by Lemma 8. By Lemma 23 this implies $t \in \mathcal{U}$. ◀

4.1 Confluence modulo $\sim_{\mathcal{R}}$

We now proceed to show that nearly orthogonal iTRSs are confluent modulo $\sim_{\mathcal{R}}$. None of the lemmas in this subsection are needed in the proof of confluence modulo $\sim_{\mathcal{H}}$. The method of the present section does not work if $\mathcal{H}$ is used instead of $\mathcal{R}$, because then the proof of Lemma 33 does not go through.

▶ **Definition 25.** The relation $\leadsto_s$ is defined coinductively.

$$\frac{t \to^* x}{t \leadsto_s x} \qquad \frac{t, s \in \mathcal{R}}{t \leadsto_s s}$$

$$\frac{t \to^* f(t_1, \ldots, t_n) \quad t_i \leadsto_s t_i' \text{ for } i = 1, \ldots, n \quad f(t_1, \ldots, t_n) \text{ is root-stable}}{t \leadsto_s f(t_1', \ldots, t_n')}$$

The relation $\leadsto_a$ is defined coinductively in the same way as $\leadsto_s$ except that in the first premise of the last rule we use $t \to^{\infty} f(t_1, \ldots, t_n)$ instead of $t \to^* f(t_1, \ldots, t_n)$.

The relation $\leadsto_s$ denotes a "standard" reduction to "normal" form. The "normal" forms are not really in normal form, but they are closely related to Böhm trees. In fact, it is not difficult to show by coinduction that if $t \leadsto_s s \to^{\infty} s'$ then $s \sim_{\mathcal{R}} s'$. Bahr and Ketema [1, 2, 24] define similar reductions to Böhm-like trees, but they do not seem to use them to obtain new proofs of infinitary confluence. The author has not studied the mentioned papers in enough depth to give a detailed comparison.

The relation $\leadsto_a$, which turns out to be the same as $\leadsto_s$ (Lemma 28), is a technical notion needed to help in some proofs.

▶ **Lemma 26.** *If $t \to^{\infty} s \leadsto_a u$ then $t \leadsto_a u$.*

Proof. If $s, u \in \mathcal{R}$ then also $t \in \mathcal{R}$ by Lemma 24, so $t \leadsto_a u$. If $u = x$ then $s \to^* x$, and thus $t \to^{\infty} x$ by Lemma 7, so $t \leadsto_a u$. Otherwise $u = f(u_1, \ldots, u_n)$, $s \to^{\infty} f(s_1, \ldots, s_n)$, $s_i \leadsto_a u_i$ for $i = 1, \ldots, n$, and $f(s_1, \ldots, s_n)$ is root-stable. By Lemma 8 we have $t \to^{\infty} f(s_1, \ldots, s_n)$. Thus $t \leadsto_a f(u_1, \ldots, u_n) = u$. ◀

▶ **Lemma 27.** *If $s = f(s_1, \ldots, s_n)$ is root-stable and $t_i \to^\infty s_i$ for $i = 1, \ldots, n$, then $t = f(t_1, \ldots, t_n)$ is also root-stable.*

Proof. Suppose t is not root-stable. Hence by Lemma 6 there is a redex u such that $t \to^*_{\mathrm{ncr}} u = g(u_1, \ldots, u_m)$. By Lemma 22 there is $u' = g(u'_1, \ldots, u'_m)$ such that $u_j \to^\infty u'_j$ for $j = 1, \ldots, m$ and $s \Rightarrow^* u'$. By Lemma 13 we conclude that u' is still a redex. Since $s \to^\infty u'$ by Corollary 10, we conclude that s is not root-stable. Contradiction. ◀

▶ **Lemma 28.** *$t \rightsquigarrow_s s$ iff $t \rightsquigarrow_a s$.*

Proof. The implication from left to right follows by straightforward coinduction. We show the other direction by coinduction. If $s = x$ then $t \to^* x$, so $t \rightsquigarrow_s s$. If $t, s \in \mathcal{R}$ then $t \rightsquigarrow_s s$. Otherwise $t \to^\infty f(t_1, \ldots, t_n)$, $s = f(s_1, \ldots, s_n)$, $f(t_1, \ldots, t_n)$ is root-stable, and $t_i \rightsquigarrow_a s_i$ for $i = 1, \ldots, n$. Then $t \to^* f(t'_1, \ldots, t'_n)$ with $t'_i \to^\infty t_i$ for $i = 1, \ldots, n$. By Lemma 27 we conclude that $f(t'_1, \ldots, t'_n)$ is root-stable. By Lemma 26 we have $t'_i \rightsquigarrow_a s_i$ for $i = 1, \ldots, n$. By the coinductive hypothesis $t'_i \rightsquigarrow_s s_i$ for $i = 1, \ldots, n$. Thus $t \rightsquigarrow_s f(s_1, \ldots, s_n) = s$. ◀

▶ **Corollary 29.** *If $t \to^\infty s \rightsquigarrow_s u$ then $t \rightsquigarrow_s u$.*

Proof. Follows from Lemma 26 and Lemma 28. ◀

At this point a reader might conjecture that the following may be easily shown:
($\star$) if $t \rightsquigarrow_s t_1$ and $t \rightsquigarrow_s t_2$ then $t_1 \sim_{\mathcal{R}} r_2$.
However, this is not the case, because a priori t might reduce to two essentially different root-stable terms. Thus it is not clear how to prove ($\star$) coinductively. Using Lemma 14 it is not difficult to show that if $t \to^* s_1$, $t \to^* s_2$ and s_1, s_2 are root-stable then s_1 and s_2 have the same root symbol. But they may still differ below the root.

Note that confluence modulo $\sim_{\mathcal{R}}$ would easily follow from ($\star$), Lemma 30 and Lemma 31. There are two methods which could probably be used to show ($\star$), though the author doubts whether any of them would lead to a much simpler confluence proof than via Lemma 33. The first method would be to adapt the proof of [28, Theorem 15]. The fact that the terms obtained through $\rightsquigarrow_s$ need not be in normal form might complicate this slightly. The second method would be to prove some standardisation result and proceed similarly to [8], using finitary standard reduction to a root-stable term in the definition of $\rightsquigarrow_s$ instead of ordinary finitary reduction. Then the proof of Corollary 29 would become more difficult, because there would be less freedom in the finitary reduction to a root-stable term in $\rightsquigarrow_s$.

▶ **Lemma 30.** *For every term t there is s with $t \rightsquigarrow_s s$.*

Proof. By coinduction. If $t \in \mathcal{R}$ then $t \rightsquigarrow_s t$. Otherwise $t \to^* t'$ for some root-stable t', by Lemma 23. If $t' = x$ then $t \rightsquigarrow_s x$. Otherwise $t' = f(t_1, \ldots, t_n)$. By the coinductive hypothesis we obtain $s_1, \ldots, s_n$ with $t_i \rightsquigarrow_s s_i$ for $i = 1, \ldots, n$. Thus $t \rightsquigarrow_s f(s_1, \ldots, s_n)$. ◀

▶ **Lemma 31.** *If $t \rightsquigarrow_s s$ then there is u with $t \to^\infty u \sim_{\mathcal{R}} s$.*

Proof. By coinduction. If $t, s \in \mathcal{R}$ then $t \sim_{\mathcal{R}} s$ and we may take $u = t$. If $s = x$ then $t \to^* x$, so $t \to^\infty s$ and we may take $u = s$. Otherwise $s = f(s_1, \ldots, s_n)$, $t \to^* f(t_1, \ldots, t_n)$ and $t_i \rightsquigarrow_s s_i$ for $i = 1, \ldots, n$. By the coinductive hypothesis we obtain u_i with $t_i \to^\infty u_i \sim_{\mathcal{R}} s_i$, for $i = 1, \ldots, n$. Take $u = f(u_1, \ldots, u_n)$. Then $t \to^\infty u \sim_{\mathcal{R}} s$. ◀

▶ **Lemma 32.** *If $t \Rightarrow t'$ and $t \rightsquigarrow_s t''$ then there is s with $t' \rightsquigarrow_s s$ and $t'' \Rightarrow s$.*

Proof. By Lemma 28 it suffices to show that if $t \Rightarrow t'$ and $t \rightsquigarrow_s t''$ then there is s with $t' \rightsquigarrow_a s$ and $t'' \Rightarrow s$. We proceed by coinduction. If $t'' = x$ then the claim follows from Lemma 15. If $t, t'' \in \mathcal{R}$ then by Corollary 9 we have $t' \in \mathcal{R}$, so $t' \rightsquigarrow_s t''$ and we may take $s = t''$. Otherwise $t \to^* f(t_1, \ldots, t_n)$, $t'' = f(t''_1, \ldots, t''_n)$, $f(t_1, \ldots, t_n)$ is root-stable and $t_i \rightsquigarrow_s t''_i$ for $i = 1, \ldots, n$. By Lemma 15 there is u with $t' \to^\infty u$ and $f(t_1, \ldots, t_n) \Rightarrow u$. Because $f(t_1, \ldots, t_n)$ is root-stable, u is also root-stable and $u = f(u_1, \ldots, u_n)$ with $t_i \Rightarrow u_i$ for $i = 1, \ldots, n$. By the coinductive hypothesis we obtain $s_1, \ldots, s_n$ with $u_i \rightsquigarrow_a s_i$ and $t''_i \Rightarrow s_i$. Take $s = f(s_1, \ldots, s_n)$. Then $t' \rightsquigarrow_a s$ and $t'' \Rightarrow s$. ◀

The proof of the following lemma fails if $\mathcal{H}$ is used instead of $\mathcal{R}$. This is because a collapse-stable term may contract at the root, in contrast to a root-stable term.

▶ **Lemma 33.** *If $t \rightsquigarrow_s t'$ and $t \to^\infty t''$ then there is s with $t' \to^\infty s$ and $t'' \rightsquigarrow_s s$.*

Proof. By Lemma 11 and Lemma 28 it suffices to show that if $t \rightsquigarrow_s t'$ and $t \to^\infty t''$ then there is s with $t' \to^{2\infty} s$ and $t'' \rightsquigarrow_a s$. We proceed by coinduction. If $t' = x$ then the claim follows from Lemma 15. If $t, t' \in \mathcal{R}$ then also $t'' \in \mathcal{R}$ by Corollary 9, so $t'' \rightsquigarrow_a t'$ and we may take $s = t'$. Otherwise $t \to^* f(t_1, \ldots, t_n)$, $t' = f(t'_1, \ldots, t'_n)$, $f(t_1, \ldots, t_n)$ is root-stable and $t_i \rightsquigarrow_s t'_i$ for $i = 1, \ldots, n$. By Lemma 15 there is u with $t'' \Rightarrow^* u$ and $f(t_1, \ldots, t_n) \to^\infty u$. Hence $f(t_1, \ldots, t_n) \to^* u' = g(u'_1, \ldots, u'_m)$, $u = g(u_1, \ldots, u_m)$ and $u'_j \to^\infty u_j$ for $j = 1, \ldots, m$. Because $f(t_1, \ldots, t_n)$ is root-stable, none of the contractions in $f(t_1, \ldots, t_n) \to^* u'$ may occur at the root. Thus $m = n$, $g = f$ and $t_i \to^* u'_i$ for $i = 1, \ldots, n$. By Lemma 32 there are $w_1, \ldots, w_n$ with $u'_i \rightsquigarrow_s w_i$ and $t'_i \Rightarrow^* w_i$. By the coinductive hypothesis we obtain $s_1, \ldots, s_n$ with $u_i \rightsquigarrow_a s_i$ and $w_i \to^{2\infty} s_i$ for $i = 1, \ldots, n$. Note that $u = f(u_1, \ldots, u_n)$ is root-stable by Corollary 9, because $f(t_1, \ldots, t_n)$ is root-stable and $f(t_1, \ldots, t_n) \to^\infty u$. Since $t'' \Rightarrow^* u$, by Corollary 10 we have $t'' \to^\infty u$, and thus $t'' \rightsquigarrow_a f(s_1, \ldots, s_n)$. Because $t' = f(t'_1, \ldots, t'_n) \Rightarrow^* f(w_1, \ldots, w_n)$, by Corollary 10 we have $t' \to^\infty f(w_1, \ldots, w_n)$, and thus $t' \to^{2\infty} f(s_1, \ldots, s_n)$. So we may take $s = f(s_1, \ldots, s_n)$. ◀

▶ **Theorem 34** (Confluence modulo $\sim_{\mathcal{R}}$ of nearly orthogonal iTRSs).
Let S be a nearly orthogonal iTRS. If $t \sim_{\mathcal{R}} s$, $t \to_S^\infty t'$ and $s \to_S^\infty s'$ then there exist t'', s'' such that $t' \to_S^\infty t''$, $s' \to_S^\infty s''$ and $t'' \sim_{\mathcal{R}} s''$.

Proof. See Figure 1 and the discussion just before it. ◀

▶ **Corollary 35.** *Any nearly orthogonal iTRS has the unique normal forms property.*

4.2 Confluence modulo $\sim_{\mathcal{H}}$

We only mention the following results, delegating the proofs to an appendix.

▶ **Theorem 36** (Confluence modulo $\sim_{\mathcal{H}}$ of nearly orthogonal iTRSs).
Let S be a nearly orthogonal iTRS. If $t \sim_{\mathcal{H}} s$, $t \to_S^\infty t'$ and $s \to_S^\infty s'$ then there exist t'', s'' such that $t' \to_S^\infty t''$, $s' \to_S^\infty s''$ and $t'' \sim_{\mathcal{H}} s''$.

▶ **Corollary 37.** *Any nearly orthogonal iTRS with no collapsing rules is confluent.*

Acknowledgements. The author would like to thank the anonymous referees for helpful comments which led to substantial improvements in the presentation of this paper. The author also thanks Paula Severi for fruitful discussions regarding infinitary lambda calculus and infinitary rewriting, and Bartek Klin for teaching him coinduction.

References

1 Patrick Bahr. Partial order infinitary term rewriting and Böhm trees. In Christopher Lynch, editor, *Proceedings of the 21st International Conference on Rewriting Techniques and Applications, RTA 2010, July 11-13, 2010, Edinburgh, Scottland, UK*, volume 6 of *LIPIcs*, pages 67–84. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2010.

2 Patrick Bahr. Partial order infinitary term rewriting. *Logical Methods in Computer Science*, 10(2), 2014.

3 Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: the Calculus of Inductive Constructions*, chapter 13. Springer, 2004.

4 Marc Bezem, Keiko Nakata, and Tarmo Uustalu. On streams that are finitely red. *Logical Methods in Computer Science*, 8:1–20, 2012.

5 Adam Chlipala. *Certified Programming with Dependent Types*, chapter 5. The MIT Press, 2013.

6 Thierry Coquand. Infinite objects in type theory. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs, International Workshop TYPES'93, Nijmegen, The Netherlands, May 24-28, 1993, Selected Papers*, volume 806 of *Lecture Notes in Computer Science*, pages 62–78. Springer, 1993.

7 Łukasz Czajka. A coinductive confluence proof for infinitary lambda-calculus. In Gilles Dowek, editor, *Rewriting and Typed Lambda Calculi – Joint International Conference, RTA-TLCA 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8560 of *Lecture Notes in Computer Science*, pages 164–178. Springer, 2014.

8 Łukasz Czajka. Coinductive techniques in infinitary lambda-calculus. In preparation, preprint available at `http://arxiv.org/abs/1501.04354`, 2015.

9 Brian A. Davey and Hilary A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 2nd edition, 2002.

10 Jörg Endrullis, Clemens Grabmayer, Dimitri Hendriks, Jan Willem Klop, and Vincent van Oostrom. Unique normal forms in infinitary weakly orthogonal rewriting. In C. Lynch, editor, *Proceedings of the Conference on Rewriting Techniques and Applications (RTA 2010)*, volume 6 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 85–102. Dagstuhl, 2010.

11 Jörg Endrullis, Clemens Grabmayer, Dimitri Hendriks, Jan Willem Klop, and Vincent van Oostrom. Infinitary term rewriting for weakly orthogonal systems: properties and examples. *Logical Methods in Computer Science*, 10:1–33, 2014.

12 Jörg Endrullis, Helle Hvid Hansen, Dimitri Hendriks, Andrew Polonsky, and Alexandra Silva. A coinductive treatment of infinitary rewriting. Unpublished, available at `http://arxiv.org/abs/1306.6224`, 2013.

13 Jörg Endrullis, Dimitri Hendriks, and Jan Willem Klop. Highlights in infinitary rewriting and lambda calculus. *Theoretical Computer Science*, 464:48–71, 2012.

14 Jörg Endrullis, Dimitri Hendriks, Jan Willem Klop, and Andrew Polonsky. Discriminating lambda-terms using clocked Böhm trees. *Logical Methods in Computer Science*, 10(2), 2014.

15 Jörg Endrullis and Andrew Polonsky. Infinitary rewriting coinductively. In Nils Anders Danielsson and Bengt Nordström, editors, *TYPES*, volume 19 of *LIPIcs*, pages 16–27. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2011.

16 Eduardo Giménez. Codifying guarded definitions with recursive schemes. In Peter Dybjer, Bengt Nordström, and Jan M. Smith, editors, *TYPES*, volume 996 of *Lecture Notes in Computer Science*, pages 39–59. Springer, 1994.

17 Bernhard Gramlich. Confluence without termination via parallel critical pairs. In Hélène Kirchner, editor, *Trees in Algebra and Programming – CAAP'96, 21st International Col-*

loquium, Linköping, Sweden, April, 22-24, 1996, Proceedings, volume 1059 of *Lecture Notes in Computer Science*, pages 211–225. Springer, 1996.

18 Gérard P. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *J. ACM*, 27(4):797–821, 1980.

19 Bart Jacobs and Jan J.M.M. Rutten. An introduction to (co)algebras and (co)induction. In *Advanced Topics in Bisimulation and Coinduction*, pages 38–99. Cambridge University Press, 2011.

20 Felix Joachimski. Confluence of the coinductive [lambda]-calculus. *Theoretical Computer Science*, 311(1-3):105–119, 2004.

21 Richard Kennaway and Fer-Jan de Vries. Infinitary rewriting. In Terese, editor, *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*, chapter 12, pages 668–711. Cambridge University Press, 2003.

22 Richard Kennaway, Jan Willem Klop, M. Ronan Sleep, and Fer-Jan de Vries. Transfinite reductions in orthogonal term rewriting systems. *Information and Computation*, 119(1):18–38, 1995.

23 Richard Kennaway, Vincent van Oostrom, and Fer-Jan de Vries. Meaningless terms in rewriting. *Journal of Functional and Logic Programming*, 1:1–35, 1999.

24 Jeroen Ketema. Böhm-like trees for term rewriting systems. In Vincent van Oostrom, editor, *Rewriting Techniques and Applications, 15th International Conference, RTA 2004, Aachen, Germany, June 3-5, 2004, Proceedings*, volume 3091 of *Lecture Notes in Computer Science*, pages 233–248. Springer, 2004.

25 Jeroen Ketema. Comparing Böhm-like trees. In Ralf 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 239–254. Springer, 2009.

26 Jeroen Ketema. Reinterpreting compression in infinitary rewriting. In Ashish Tiwari, editor, *23rd International Conference on Rewriting Techniques and Applications (RTA'12) , RTA 2012, May 28 – June 2, 2012, Nagoya, Japan*, volume 15 of *LIPIcs*, pages 209–224. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2012.

27 Jeroen Ketema and Jakob Grue Simonsen. Infinitary combinatory reduction systems. *Information and Computation*, 209(6):893–926, 2011.

28 Jan Willem Klop and Roel C. de Vrijer. Infinitary normalization. In Sergei N. Artëmov, Howard Barringer, Artur S. d'Avila Garcez, Luís C. Lamb, and John Woods, editors, *We Will Show Them! Essays in Honour of Dov Gabbay, Volume Two*, pages 169–192. College Publications, 2005.

29 Dexter Kozen and Alexandra Silva. Practical coinduction. Draft, 2014.

30 Xavier Leroy and Hervé Grall. Coinductive big-step operational semantics. *Information and Computation*, 207(2):284–304, 2009.

31 Keiko Nakata and Tarmo Uustalu. Resumptions, weak bisimilarity and big-step semantics for while with interactive I/O: An exercise in mixed induction-coinduction. In L. Aceto and P. Sobociński, editors, *Seventh Workshop on Structural Operational Semantics (SOS'10)*, pages 57–75, 2010.

32 Jan J.M.M. Rutten. Universal coalgebra: a theory of systems. *Theoretical Computer Science*, 249(1):3–80, 2000.

33 Davide Sangiorgi. *Introduction to Bisimulation and Coinduction.* Cambridge University Press, 2012.

34 Yoshihito Toyama. Commutativity of term rewriting systems. In K. Fuchi and L. Kott, editors, *Programming of Future Generation Computers II*, pages 393–407. North-Holland, 1988.

35 Vincent van Oostrom. Development closed critical pairs. In Gilles Dowek, Jan Heering, Karl Meinke, and Bernhard Möller, editors, *HOA*, volume 1074 of *Lecture Notes in Computer Science*, pages 185–200. Springer, 1995.
36 Vincent van Oostrom. Developing developments. *Theoretical Computer Science*, 175(1):159–181, 1997.

A Proof of Theorem 36

It turns out that because nearly orthogonal iTRSs allow no non-root overlaps, all root overlaps with a collapsing rule must have a special form.

▶ **Lemma 38.** *Let $\langle l_1, x\rangle, \langle l_2, r_2\rangle \in S$ and let σ be the mgu of l_1 and l_2. Then $\sigma(r_2) \Rightarrow \sigma(x)$.*

Proof. Because S is nearly orthogonal, there is s such that $\sigma(r_2) \Rightarrow s$ and $\sigma(x) \to^\infty s$. It suffices to show that $s = \sigma(x)$. If $\sigma(x)$ is a variable then this is obvious. Otherwise, because σ is an mgu of two linear terms, we may assume that $\sigma(x)$ is a proper subterm of l_2 (we may assume the subterm is proper because l_1 is not a variable). But then $\sigma(x)$ cannot contain any redexes, because they would constitute a non-root overlap with the rule $\langle l_2, r_2\rangle$. Hence $\sigma(x) = s$. ◀

▶ **Definition 39.** A *hypercollapsing sequence* for a term t is an infinite sequence $(t_n)_{n\in\mathbb{N}}$ of terms satisfying:

- $t \to^\infty t_0$, and
- for each $n \in \mathbb{N}$ there is a collapsing rule $\langle l, x\rangle \in S$ and a substitution σ such that $t_n = \sigma(l) \to \sigma(x) \to^\infty t_{n+1}$.

The following lemma was shown for orthogonal iTRSs in [21, Lemma 12.8.4], by essentially the same proof.

▶ **Lemma 40.** *If there exists a hypercollapsing sequence for t then $t \in \mathcal{H}$.*

Proof. Assume that $(t_n)_{n\in\mathbb{N}}$ is a hypercollapsing sequence for t. It suffices to show that if $t \to s$ then there is a hypercollapsing sequence for s. Then it will follow from Lemma 23 that $t \in \mathcal{H}$.

Assume $t \to s$. We describe the construction of a hypercollapsing sequence $(s_n)_{n\in\mathbb{N}}$ for s. Assume the elements $s_0, \ldots, s_{n-1}$ of the sequence have been defined, and u, v are such that $u \Rightarrow v$, $u \to^\infty t_n$. In the base case $n = 0$ we take $u = t$ and $v = s$. By Lemma 15 there is v' with $v \to^\infty v'$ and $t_n \Rightarrow v'$. By the definition of a hypercollapsing sequence there are $\langle l, x\rangle \in S$ and σ such that $t_n = \sigma(l) \to \sigma(x) \to^\infty t_{n+1}$. If $t_n \Rightarrow v'$ by a root contraction, then $v' \to^\infty \sigma(x)$ by Lemma 38. Hence $v \to^\infty v' \to^\infty \sigma(x) \to^\infty t_{n+1}$, so $v \to^\infty t_{n+1}$ by Lemma 8. Then take $s_m = t_{m+1}$ for $m \geq n$ and finish the construction. So assume $t_n \Rightarrow v'$ is not a root contraction. Then $t_n = f(w_1, \ldots, w_k)$, $v' = f(w_1', \ldots, w_k')$ and $w_i \Rightarrow w_i'$ for $i = 1, \ldots, k$. By Lemma 13 there is σ' such that $v' = \sigma'(l)$ and $\sigma(x) \Rightarrow \sigma'(x)$. Hence $v \to^\infty v' \to \sigma'(x)$. Take $s_n = v'$ and continue the construction with $u := \sigma(x)$ and $v := \sigma'(x)$.

It follows by construction that $(s_n)_{n\in\mathbb{N}}$ is a hypercollapsing sequence for s. ◀

▶ **Definition 41.** The relation $\rightsquigarrow$ is defined coinductively.

$$\frac{t \to^* x}{t \rightsquigarrow x} \qquad \frac{t \to^* f(t_1, \ldots, t_n) \quad t_i \rightsquigarrow t_i' \text{ for } i = 1, \ldots, n}{t \rightsquigarrow f(t_1', \ldots, t_n')} \qquad \frac{t, s \in \mathcal{H}}{t \rightsquigarrow s}$$

The relation $\rightsquigarrow^\infty$ is defined coinductively in the same way as $\rightsquigarrow$ except that in the first premise of the second rule we use $t \to^\infty f(t_1, \ldots, t_n)$ instead of $t \to^* f(t_1, \ldots, t_n)$.

The intuitive interpretation of $\rightsquigarrow$ is quite different from the intuitive interpretation of $\rightsquigarrow_s$ in Section 4.1. If $t \rightsquigarrow s$ then s need not be "normal" in any sense. The crucial difference is that in the second rule we do not require $f(t_1, \ldots, t_n)$ to be collapse-stable. Essentially, $t \rightsquigarrow s$ means that t infinitarily reduces to s, up to equivalence of hypercollapsing subterms. This intuition is validated by the following lemma.

▶ **Lemma 42.** *The following conditions are equivalent:*
1. $t \to^\infty u \sim_{\mathcal{H}} s$ *for some term* u,
2. $t \rightsquigarrow s$,
3. $t \rightsquigarrow^\infty s$.

Proof.
($1 \Rightarrow 2$) By coinduction, analysing $u \sim_{\mathcal{H}} s$. If $u, s \in \mathcal{H}$ then $t \in \mathcal{H}$ by Lemma 24, so $t \rightsquigarrow s$. If $u = s = x$ then $t \to^* x$, so $t \rightsquigarrow s$. If $u = f(u_1, \ldots, u_n)$, $s = f(s_1, \ldots, s_n)$ and $u_i \sim_{\mathcal{H}} s_i$ for $i = 1, \ldots, n$, then $t \to^* f(t_1, \ldots, t_n)$ with $t_i \to^\infty u_i$. By the coinductive hypothesis $t_i \rightsquigarrow s_i$ for $i = 1, \ldots, n$. Thus $t \rightsquigarrow f(s_1, \ldots, s_n) = s$.
($2 \Rightarrow 3$) Straightforward coinduction.
($3 \Rightarrow 1$) We show by coinduction that if $t \rightsquigarrow^\infty s$ then there is u with $t \to^{2\infty} u \sim_{\mathcal{H}} s$. This suffices by Lemma 11. If $s = x$ then $t \to^* x \sim_{\mathcal{H}} x$, so we may take $u = x$. If $t, s \in \mathcal{H}$ then $t \sim_{\mathcal{H}} s$ and we may take $u = t$. Otherwise $t \to^\infty f(t_1, \ldots, t_n)$, $s = f(t'_1, \ldots, t'_n)$ and $t_i \rightsquigarrow^\infty t'_i$ for $i = 1, \ldots, n$. By the coinductive hypothesis we obtain $u_1, \ldots, u_n$ with $t_i \to^{2\infty} u_i \sim_{\mathcal{H}} t'_i$. Take $u = f(u_1, \ldots, u_n)$. Then $t \to^{2\infty} u \sim_{\mathcal{H}} s$. ◀

▶ **Lemma 43.** *If* $s \sim_{\mathcal{H}} t \Rightarrow t'$ *then there is* s' *with* $s \Rightarrow s' \sim_{\mathcal{H}} t'$.

Proof. By coinduction. If $t, s \in \mathcal{H}$ then $t' \in \mathcal{H}$ by Corollary 9, so $t' \sim_{\mathcal{H}} s$ and we may take $s' = s$. If $t = x$ then $t' = x$ and we may take $s' = s$. Otherwise $s = f(s_1, \ldots, s_n)$, $t = f(t_1, \ldots, t_n)$ and $s_i \sim_{\mathcal{H}} t_i$ for $i = 1, \ldots, n$. If $t \Rightarrow t'$ is a root contraction then the claim follows from Lemma 17. If $t \Rightarrow t'$ does not contract at the root, then the claim follows directly from the coinductive hypothesis. ◀

▶ **Lemma 44.** *If* $t \Rightarrow t_1$ *and* $t \rightsquigarrow t_2$ *then there is* s *with* $t_1 \rightsquigarrow s$ *and* $t_2 \Rightarrow s$.

Proof. Follows from Lemma 42, Lemma 15 and Lemma 43. ◀

The construction of a hypercollapsing sequence in the proof of the following lemma is similar to the construction in [21, Lemma 12.8.14].

▶ **Lemma 45.** *If* $t \notin \mathcal{H}$, $t \to^\infty t'$ *and* $t \rightsquigarrow u$ *then one of the following holds:*
1. $t' \to^* x$ *and* $u \to^* x$ *for some variable* x, *or*
2. *there are* $s = f(s_1, \ldots, s_n)$, $u' = f(u_1, \ldots, u_n)$ *and* $w = f(w_1, \ldots, w_n)$ *such that* $t \to^* s$, $t' \to^\infty w$, $u \to^\infty u'$, $s_i \to^\infty w_i$ *and* $s_i \rightsquigarrow u_i$ *for* $i = 1, \ldots, n$.

Proof. By Lemma 40 it suffices to show that if neither 1 nor 2 holds then a hypercollapsing sequence $(v_k)_{k \in \mathbb{N}}$ for t may be constructed.

If $t' = x$ then by Lemma 44 we have $u \to^* x$, so 1 holds. If t' is not a variable then we have $t' = f(t'_1, \ldots, t'_n)$. Because $t \to^\infty t'$, there are $t_1, \ldots, t_n$ with $t \to^* t_0 = f(t_1, \ldots, t_n)$ and $t_i \to^\infty t'_i$ for $i = 1, \ldots, n$. By Lemma 44 and Corollary 10 there is u' with $u \to^\infty u'$ and $t_0 \rightsquigarrow u'$. If $u' = x$ then $u \to^* x$ and $t_0 \to^* x$, and thus $t' \to^* x$, by Lemma 15 and Corollary 10, so point 1 is true. Hence assume $u' = g(u_1, \ldots, u_m)$. Because $t \notin \mathcal{H}$, also $t_0 \notin \mathcal{H}$ by Lemma 24. Thus $t_0 \to^* s = g(s_1, \ldots, s_m)$ with $s_j \rightsquigarrow u_j$ for $j = 1, \ldots, m$.

If $t_0 \to^*_{\mathrm{ncr}} s$ then by Lemma 22 and Corollary 10 there is $w = g(w_1, \ldots, w_m)$ with $t' \to^\infty w$ and $s_j \to^\infty w_j$ for $j = 1, \ldots, m$. Since also $t \to^* s$, $u \to^\infty u'$ and $s_j \to^\infty u_j$ for $j = 1, \ldots, m$, then point 2 is true.

So suppose there is a collapsing root contraction in the reduction $t_0 \to^* s$, i.e., $t_0 \to^* \sigma(l) \to \sigma(x) \to^* s$ for some collapsing rule $\langle l, x\rangle \in S$ and some substitution σ. Since $t_0 \to^* \sigma(x)$ and $t_0 \to^\infty t'$, by Lemma 15 and Corollary 10 there is t'' with $t' \to^\infty t''$ and $\sigma(x) \to^\infty t''$. Note that also $\sigma(x) \notin \mathcal{H}$ and $\sigma(x) \rightsquigarrow u'$, by Lemma 24 because $t \to^* \sigma(x)$. Note that if the points 1-2 hold for $\sigma(x), t'', u'$ then they also hold for t, t', u, by Lemma 8. So we may take $v_k = \sigma(l)$ as the next element of the hypercollapsing sequence, and continue the construction with $t := \sigma(x)$, $t' := t''$ and $u := u'$.

Ultimately, we will either conclude that 1 or 2 holds, or we will construct a hypercollapsing sequence $(v_k)_{k\in\mathbb{N}}$ for t. ◀

▶ **Lemma 46.** *If $t \to^\infty t_1$ and $t \rightsquigarrow t_2$ then there is s with $t_1 \rightsquigarrow^\infty s$ and $t_2 \to^{2\infty} s$.*

Proof. By coinduction. If $t \in \mathcal{H}$ then $t_1, t_2 \in \mathcal{H}$ by Corollary 9, Lemma 42 and Lemma 19, so $t_1 \rightsquigarrow^\infty t_2$ and we may take $s = t_2$. So assume $t \notin \mathcal{H}$. Then by Lemma 45 either $t_1, t_2 \to^* x$ for some variable x, and then we may take $s = x$, or there are $v = f(v_1, \ldots, v_n)$, $u = f(u_1, \ldots, u_n)$ and $w = f(w_1, \ldots, w_n)$ such that $t_1 \to^\infty w$, $t_2 \to^\infty u$, and $v_i \to^\infty w_i$ and $v_i \rightsquigarrow u_i$ for $i = 1, \ldots, n$. By the coinductive hypothesis we obtain $s_1, \ldots, s_n$ with $w_i \rightsquigarrow^\infty s_i$ and $u_i \to^{2\infty} s_i$ for $i = 1, \ldots, n$. Take $s = f(s_1, \ldots, s_n)$. Then $t_1 \rightsquigarrow^\infty s$ and $t_2 \to^{2\infty} s$. ◀

▶ **Theorem 36** (Confluence modulo $\sim_{\mathcal{H}}$ of nearly orthogonal iTRSs)**.**
Let S be a nearly orthogonal iTRS. If $t \sim_{\mathcal{H}} s$, $t \to^\infty_S t'$ and $s \to^\infty_S s'$ then there exist t'', s'' such that $t' \to^\infty_S t''$, $s' \to^\infty_S s''$ and $t'' \sim_{\mathcal{H}} s''$.

Proof. Assume $t \sim_{\mathcal{H}} s$, $t \to^\infty t'$ and $s \to^\infty s'$. By Lemma 18 there is u with $s \to^\infty u \sim_{\mathcal{H}} t'$. Hence $s \rightsquigarrow t'$ by Lemma 42. By Lemma 46, Lemma 42 and Lemma 11 there are t'', s'' with $t' \to^\infty t''$ and $s' \to^\infty s'' \sim_{\mathcal{H}} t''$. ◀

B Strongly convergent reductions

In this section we prove that for left-linear iTRSs the existence of coinductive infinitary reductions is equivalent to the existence of strongly convergent reductions. As a corollary, this also yields ω-compression of strongly convergent reductions. The equivalence proof is virtually the same as in [15]. The notion of strongly convergent reductions is the standard notion of infinitary reductions used in non-coinductive treatments of infinitary rewriting. See e.g. [21] for details. In the rest of this section we fix a left-linear iTRS $\mathcal{S} = \langle \Sigma, S\rangle$.

▶ **Definition 47.** On the set of terms we define a metric d by

$$d(t, s) = \inf\{2^{-n} \mid t^{\upharpoonright n} = s^{\upharpoonright n}\}$$

where $r^{\upharpoonright n}$ for $r \in T^\infty(\Sigma)$ is defined as the term obtained by replacing all subterms of r at depth n by a fresh constant $\bot$. This defines a metric topology on the set of terms. Let α be an ordinal. A map $\phi : \{\beta \leq \alpha\} \to T^\infty(\Sigma)$ together with contraction steps $\sigma_\beta : \phi(\beta) \to_S \phi(\beta+1)$ for $\beta < \alpha$ is a *strongly convergent S-reduction sequence of length α from $\phi(0)$ to $\phi(\alpha)$* if the following conditions hold:

1. if $\gamma \leq \alpha$ is a limit ordinal then $f(\gamma)$ is the limit in the metric topology on infinitary terms of the ordinal-indexed sequence $(\phi(\beta))_{\beta<\gamma}$,

2. if $\gamma \leq \alpha$ is a limit ordinal then for every $d \in \mathbb{N}$ there exists $\beta < \gamma$ such that for all β' with $\beta \leq \beta' < \gamma$ the redex contracted in the step $\sigma_{\beta'}$ occurs at depth greater than d.

We write $s \xrightarrow{Q,\alpha}_S t$ if Q is a strongly convergent S-reduction sequence of length α from s to t.

▶ **Theorem 48.**

1. *If $s \to_S^\infty t$ then there exists a strongly convergent R-reduction sequence from s to t of length at most ω.*
2. *If there exists a strongly convergent S-reduction sequence from s to t then $s \to_S^\infty t$.*

Proof. The proof is a straightforward adaptation of the proof of Theorem 3 in [15].

Suppose that $s \to_S^\infty t$. By traversing the infinite derivation tree of $s \to_S^\infty t$ and accumulating the finite prefixes by concatenation, we obtain a reduction sequence of length at most ω which satisfies the depth requirement by construction.

For the other direction, by induction on α we show that if $s \xrightarrow{Q,\alpha}_S t$ then $s \to_S^{2\infty} t$, which suffices for $s \to_S^\infty t$ by Lemma 11 (recall that the proofs of lemmas 6-11 depended only on the left-linearity of S). There are three cases.

- $\alpha = 0$. If $s \xrightarrow{Q,0}_S t$ then $s = t$, so $s \to_S^{2\infty} t$.
- $\alpha = \beta + 1$. If $s \xrightarrow{S,\beta+1}_S t$ then $s \xrightarrow{Q',\beta}_S s' \to_S t$. Hence $s \to_S^{2\infty} s'$ by the inductive hypothesis. Then $s \to_S^\infty s' \to_S t$ by Lemma 11. So $s \to_S^\infty t$ by Lemma 7.
- α is a limit ordinal. By coinduction we show that if $s \xrightarrow{Q,\alpha}_S t$ then $s \to_S^{2\infty} t$. By the depth condition there is $\beta < \alpha$ such that for every $\gamma \geq \beta$ the redex contracted in S at γ occurs at depth greater than zero. Let t_β be the term at index β in Q. Then by the inductive hypothesis we have $s \to_S^{2\infty} t_\beta$, and thus $s \to_S^\infty t_\beta$ by Lemma 11. There are two cases.
  - $t_\beta = x$. This is impossible because then there can be no contraction of t_β at depth greater than zero.
  - $t_\beta = f(t_1, \ldots, t_n)$. Then $t = f(u_1, \ldots, u_n)$ and the tail of the reduction S past β may be split into n parts: $t_i \xrightarrow{Q_i,\delta_i}_S u_i$ with $\delta_i \leq \alpha$ for $i = 1, \ldots, n$. Then $t_i \to_S^{2\infty} u_i$ by the inductive and/or the coinductive hypothesis. Since $s \to_S^\infty f(t_1, \ldots, t_n)$ we obtain $s \to_S^{2\infty} f(u_1, \ldots, u_n) = t$.

◀

▶ **Corollary 49** (ω-compression)**.** *If there exists a strongly convergent S-reduction sequence from s to t then there exists such a sequence of length at most ω.*

No complete linear term rewriting system for propositional logic

Anupam Das[1] and Lutz Straßburger[2]

1 **ENS de Lyon and Inria, France, anupam.das@ens-lyon.fr**
2 **INRIA Saclay – Île-de-France, France, and Laboratoire d'Informatique (LIX), Palaiseau, France, lutz@lix.polytechnique.fr**

Abstract

Recently it has been observed that the set of all sound linear inference rules in propositional logic is already **coNP**-complete, i.e. that every Boolean tautology can be written as a (left- and right-) linear rewrite rule. This raises the question of whether there is a rewriting system on linear terms of propositional logic that is sound and complete for the set of all such rewrite rules. We show in this paper that, as long as reduction steps are polynomial-time decidable, such a rewriting system does not exist unless **coNP** = **NP**.

We draw tools and concepts from term rewriting, Boolean function theory and graph theory in order to access the required intermediate results. At the same time we make several connections between these areas that, to our knowledge, have not yet been presented and constitute a rich theoretical framework for reasoning about linear TRSs for propositional logic.

1998 ACM Subject Classification F.4 Mathematical Logic and Formal Languages

Keywords and phrases Linear rules, Term rewriting, Propositional logic, Proof theory, Deep inference

Digital Object Identifier 10.4230/LIPIcs.RTA.2015.127

1 Introduction

Linear inferences, as defined in [9] and also known as "balanced" tautologies (e.g. in [24]) or linear rules (e.g. in deep inference [2], [3] [12], [13]), are sound implications in classical propositional logic (CPL), each of whose variables occur exactly once in both the premiss and the conclusion. From the point of view of term rewriting they are rewrite rules that are non-erasing, left- and right-linear, and such that the Boolean function computed by the left hand side logically implies that computed by the right hand side.[1]

The reason why this is an interesting set of rewrite rules is due to the observation that *all* Boolean tautologies can be written in this form, by means of a polynomial-time translation [24]. In this work we ask whether one can derive all of CPL *internally* to this fragment; i.e. is there a set of linear inferences (satisfying certain conditions) that is complete, under term rewriting, for the set of all linear inferences (denoted L henceforth)?

It was previously shown that such a set could not be finite [9, 24], via an encoding of instances of the pigeonhole principle as linear inferences. However in this work we consider any system whose reduction steps can be checked efficiently, i.e. form a polynomial-time decidable set. The motivation behind this generality is that such a set would constitute

[1] For generality and ease of presentation, we later drop the "non-erasing" criterion for linear inferences in this work.

© Anupam Das and Lutz Straßburger;
licensed under Creative Commons License CC-BY
26th International Conference on Rewriting Techniques and Applications (RTA'15).
Editor: Maribel Fernández; pp. 127–142
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

a sound and complete *proof system*[2] for CPL with no meaningful duplication, creation or destruction of formulae,[3] in stark contrast to the traditional approach of *structural* proof theory, based on rules exhibiting precisely such behaviour.

In this work we show that no such linear system exists, unless **coNP** = **NP**. In a little more detail, we show that any such system would admit a derivation of each valid linear inference of polynomial length (and so polynomial-size, by linearity). This would imply that **coNP** is contained in **NP** as follows:

1. There is an **NP**-algorithm for L: simply guess the correct derivation in some sound and complete linear system.
2. Since *TAUT* is polynomial-time reducible to L there is also an **NP**-algorithm for *TAUT*.
3. By the Cook-Levin theorem that *SAT* is **NP**-complete [5, 20], we have that *TAUT* is **coNP**-complete, and so there is a **NP**-algorithm for **coNP**.

Functions computed by linear terms of CPL have been studied in Boolean function theory, and more specifically circuit complexity, for decades, where they are called "read-once functions" (e.g. in [7]).[4] They are closely related to positional games (first mentioned in [15]) and have been used in amplification of approximation circuits, (first in [26], more generally in [11]) as well amongst other areas. Their equivalence classes under associativity and commutativity of $\wedge$ and $\vee$ can also be represented as the set of "cographs", or "P_4-free" graphs, essentially what we call "relation webs" in this work, following [13] and [23].

In this paper we work in both the Boolean function theoretic and graph theoretic settings, as well as that of term rewriting, presenting novel interplays between them. In particular, the proof of our main result, Thm. 30, crucially uses concepts from all three settings, which we hope is clear from the exposition.

We develop connections and applications of concepts about read-once functions, e.g., Prop. 13 and Thm. 19, that seem to be novel, as results on such concepts have appeared before only in the setting of isolated Boolean functions, rather than in a logical setting where we care futhermore about logical relations between functions, in particular, when one function implies another.

From the point of view of rewriting theory, logic has always been a motivational domain of applications. For example, "tautology checking" is used as one of the three motivating examples in the Terese book, *Term Rewriting Systems* [25]. Rewriting systems for propositional logic can be recovered from axiom systems for Boolean algebras and Boolean rings, e.g. as in [10] and [18]. While this area has been well studied, our 'deep inference' style approach is more general in scope due to our handling of negation: by dealing with terms in negation normal form we can reason about systems that are not purely equational, but consisting of arbitrary sound rules, due to the absence of negative contexts. Notice that complete equational theories for CPL cannot possibly be linear, e.g. due to Thm. 9, and so such a question is only pertinent in our more general setting.

The organisation of this paper is as follows. In Sects. 2 and 3 we present the basics on term rewriting in CPL and usual Boolean interpretations. In Sect. 4 we define relation webs and give graph-theoretic versions of various logical concepts. In Sect. 5 we present a normal form of linear derivations, which we ultimately use in Sect. 6 to prove our main

---

[2] Recall that proof systems are usually required to be efficiently (i.e. polynomial-time) checkable [6].

[3] The only duplication would occur in the reduction from *TAUT* to L where its complexity is bounded by some fixed polynomial.

[4] These have been studied in various forms and under different names. The first appearance we are aware of is in [4], and also the seminal paper of [14] characterising these functions. The book we reference presents an excellent and comprehensive introduction to the area.

result, polynomial-time weak normalisation. In Sect. 7 we apply previous results to deduce and conjecture forms of *canonicity* of certain linear rules prominent in deep inference proof theory, and in Sect. 8 we make some concluding remarks.

2 Preliminaries on rewriting theory

We generally work in the first-order term rewriting setting defined in the Terese textbook, *Term Rewriting Systems* [25]. We will, in fact, use the same notation for all symbols except the connectives, for which we use more standard notation from proof theory. In particular we will use $\bot$ and $\top$ for the truth constants, reserving 0 and 1 for the inputs and outputs of Boolean functions, introduced later.

We adopt two particular conventions which differ from usual definitions in the literature:

1. A TRS is usually defined as an arbitrary set of rewrite rules. Here we insist that the set of instances of these rules, or reduction steps, is polynomial-time decidable.
2. Rewriting modulo an equivalence relation usually places no restriction on the source and target of a reduction step. Here we insist that they must be *distinct* modulo the equivalence relation.

The motivation for (1) is that we wish to be as general as possible without admitting trivial results. If we allowed all sets then a complete system could be specified quite easily indeed. Furthermore, that an inference rule is easily or feasibly checkable is a usual requirement in proof theory, and in proof complexity this is formalised by the same condition (1) on inference rules, essentially due to the fact that *TAUT* is **coNP**-complete. Perhaps it would be better to call these 'polynomial' TRSs, however we drop this prefix for presentation reasons throughout this article.

The motivation for (2) is that we fundamentally care about weak normalisation, e.g. Cor. 31, but it will be useful to make statements resembling strong normalisation under this notion of rewriting modulo, e.g. Thm. 30. All the equivalence relations we will work with are polynomial-time decidable, and so this convention is consistent with (1). The same notion of rewriting modulo was also used in previous work [9].

Propositional logic in the term rewriting setting

Our language is built from the connectives $\bot, \top, \wedge, \vee$ and a set *Var* of propositional variables, typically denoted $x, y, z, \ldots$. The set *Var* is equipped with an involution (i.e. self-inverse function) $\bar{\cdot} : Var \to Var$. We call $\bar{x}$ the *dual* of x and, for each pair of dual variables, we arbitrarily choose one to be *positive* and the other to be *negative*.

The set *Ter* of formulae, or *terms*, is built freely from this signature in the usual way. Terms are typically denoted by $s, t, u, \ldots$, and term and variable symbols may occur with superscripts and subscripts if required.

In this setting $\top$ and $\bot$ are considered the constant symbols of our language. We say that a term t is *constant-free* if $\top$ and $\bot$ do not occur in t.

We do not include a symbol for negation in our language. This is due to the fact that soundness of a rewrite step is only preserved under *positive* contexts. Instead we simply consider terms in negation normal form (NNF), which can be generated for arbitrary terms from positive and negative variables by the De Morgan laws:

$$\bar{\top} = \bot \qquad \bar{\bot} = \top \qquad \bar{\bar{x}} = x \qquad \overline{A \vee B} = \bar{A} \wedge \bar{B} \qquad \overline{A \wedge B} = \bar{A} \vee \bar{B}$$

We say that a term is *negation-free* if it does not contain any negative variables. We write $Var(t)$ to denote the set of variables occurring in t. We say that a term t is *linear* if, for

each $x \in \mathit{Var}(t)$, there is exactly one occurrence of x in t. The *size* of a term t, denoted $|t|$, is the total number of variable and function symbols occurring in t. A *substitution* is a mapping $\sigma\colon \mathit{Var} \to \mathit{Ter}$ from the set of variables to the set of terms such that $\sigma(x) \neq x$ for only finitely many x. The notion of substitution is extended to all terms, i.e. a map $\mathit{Ter} \to \mathit{Ter}$, in the usual way. A (one-hole) *context* is a term with a single 'hole' $\square$ occurring in place of a subterm. For example consider the following:

$$C_1[\square] := y \wedge (z \vee \square) \qquad C_2[\square] := \square \vee (w \wedge x) \qquad C_3[\square] := (w \wedge x) \vee (y \wedge (z \vee \square))$$

We may write $C_i[t]$ to denote the term obtained by replacing the occurrence of $\square$ in $C_i[\square]$ with t. We may also replace holes with other contexts to derive new contexts. For example, notice that $C_3[\square]$ is equivalent, modulo commutativity of $\vee$, to $C_2[C_1[\square]]$.

▶ **Definition 1** (Rewrite rules). A *rewrite rule* is an expression $l \to r$, where l and r are terms. We write $\rho : l \to r$ to express that the rule $l \to r$ is called ρ. In this rule we call l the left hand side (LHS) of ρ, and r the right hand side (RHS).

We say that ρ is *left-linear* (resp. *right-linear*) if l (resp. r) is a linear term. We say that ρ is *linear* if it is both left- and right-linear.

We write $s \underset{\rho}{\to} t$ to express that $s \to t$ is a *reduction step* of ρ, i.e. that $s = C[\sigma(l)]$ and $t = C[\sigma(r)]$ for some substitution σ and context $C[\square]$.

▶ **Definition 2** (Term rewriting systems). A *term rewriting system* (TRS) is a set of rewrite rules whose reduction steps are decidable in polynomial time. The *one-step* reduction relation of a TRS R is $\underset{R}{\to}$, where $s \underset{R}{\to} t$ if $s \underset{\rho}{\to} t$ for some $\rho \in R$.

A *linear* (term rewriting) system is a TRS, all of whose rules are linear.

▶ **Definition 3** (Derivations). A *derivation* under a binary relation $\underset{R}{\to}$ on Ter is a sequence $\pi : t_0 \underset{R}{\to} t_1 \underset{R}{\to} \cdots \underset{R}{\to} t_l$. In this case we say that π has *length* l.

We also write $\overset{*}{\underset{R}{\to}}$ to denote the reflexive transitive closure of $\underset{R}{\to}$.

▶ **Definition 4** (Rewriting modulo). For an equivalence relation $\sim$ on Ter and a TRS R, we define the relation $\underset{R/\sim}{\longrightarrow}$ by $s \underset{R/\sim}{\longrightarrow} t$ if there are s', t' such that $s \sim s' \underset{R}{\to} t' \sim t$ such that $s' \nsim t'$. An $R/\sim$ derivation is also called an R-derivation *modulo* $\sim$.

In this work we consider linear equivalence relations, like associativity and commutivity of $\wedge$ and $\vee$, denoted AC. We also have linear equations for the truth constants, the system U:

$$x \vee \bot = x = \bot \vee x \quad , \quad x \wedge \top = x = \top \wedge x \quad , \quad \top \vee \top = \top \quad , \quad \bot \wedge \bot = \bot$$

We denote by ACU the combined system of AC and U. For certain reasons it will also be useful to consider the system U' that extends U by the following rules:[5]

$$x \vee \top = \top = \top \vee x \quad , \quad x \wedge \bot = \bot = \bot \wedge x$$

We denote by ACU' the combined system of AC and U'. It turns out that this equivalence relation relates precisely those linear terms that compute the same Boolean function, as we discuss in the next section.

▶ Remark (On the use of '$\to$'). To avoid possible confusion, notice that we are using the $\to$ symbol both for a formal expression, e.g. the rewrite rule $s \to t$, and with annotations to express a relation between two terms, e.g. the reduction step $s \underset{\rho}{\to} t$.

[5] Notice that these are not linear in the sense of [9], but are considered linear in our more general setting.

3 Preliminaries on Boolean functions

In this section we introduce the usual Boolean function models for terms of propositional logic.

A *Boolean function* on a (finite) set of variables $X \subseteq Var$ is a map $f\colon \{0,1\}^X \to \{0,1\}$. We identify $\{0,1\}^X$ with $\mathcal{P}(X)$, the powerset of X, i.e. we may specify an argument of a Boolean function by the subset of its variables assigned to 1.

A little more formally, a function $\nu : X \to \{0,1\}$ is specified by the set X_ν it indicates, i.e. $x \in X_\nu$ just if $\nu(x) = 1$. For this reason we may quantify over the arguments of a Boolean function by writing $Y \subseteq X$ rather than $\nu \in \{0,1\}^X$, i.e., we write $f(Y)$ to denote the value of f if the input is 1 for the variables in Y and 0 for the variables in $X \setminus Y$. Similarly, we write $f(\overline{Y})$ for the value of f when the variables in Y are 0 and the variables in $X \setminus Y$ are 1.

3.1 Boolean semantics of terms

A term t computes a Boolean function $\{0,1\}^{Var(t)} \to \{0,1\}$ in the usual way.

For Boolean functions $f, g : \{0,1\}^X \to \{0,1\}$ we write $f \leq g$ if $\forall Y \subseteq X$ we have that $f(Y) \leq g(Y)$. Notice that the following can easily be shown to be equivalent:

1. $f \leq g$.
2. $f(Y) = 1 \Rightarrow g(Y) = 1$.
3. $g(Y) = 0 \Rightarrow f(Y) = 0$.

We also write $f < g$ if $f \leq g$ but $f(Y) \neq g(Y)$ for some $Y \subseteq X$.

▶ **Definition 5** (Soundness)**.** We say that a rewrite rule $s \to t$ is *sound* if s and t compute Boolean functions f and g, respectively, such that $f \leq g$. We say that a TRS is sound if all its rules are sound. A *linear inference* is a sound linear rewrite rule. The set of all linear inferences is denoted by L.

▶ Notation 6. To switch conveniently between the settings of terms and Boolean functions, we freely interchange notations, e.g. writing $s \leq t$ to denote that $s \to t$ is sound, and saying $f \to g$ is sound when $f \leq g$.

▶ Remark. We point out that, here, our definition of "linear inference" differs slightly from that occurring in [9]. Namely, we insist only that the LHS and RHS are linear, but not necessarily that they have the same variable set. We choose this more general definition since it seems more natural in the setting of term rewriting. Furthermore, since it is indeed a more general definition, the same result carries over for the previous notion too. In fact, in later sections, we will restrict our attention to the former notion of linear inference due to the fact that any erasure or introduction[6] of variables in a linear rule would constitute what we call a "triviality" in Section 5, where we also elaborate on and address this issue.

Finally we give one of the key motivations for this work, essentially from [24]:

▶ **Proposition 7.** L *is* **coNP**-*complete.*

This result is the reason, from the point of proof theory, why one might restrict attention to only linear inferences at all: every Boolean tautology can be written as a linear inference. As we can see from the proof that follows, the translation is not very complicated. However,

[6] We point out that in many settings, indeed in [25], a rewrite rule is not allowed to introduce new variables. I.e. all variables occurring on the RHS must also occur in the LHS. In our setting it seems more natural and symmetric to allow such behaviour and, again, this yields a more general result.

it does induce an at most quadratic blowup in size from an input tautology to a linear inference.

We include a proof below, for completeness, and since the statement here differs slightly from that in [24].

Proof of Proposition 7. That L is in **coNP** is due to the fact that checking soundness of a rewrite rule $s \to t$ can be reduced to checking validity of the formula $\bar{s} \vee t$. To prove **coNP**-hardness, we can reduce validity of general tautologies to soundness of linear rewrite rules. We let t' be the term obtained from t (which is assumed to be in NNF) by doing the following for each positive variable x: let n be the number of occurrences of x in t, and let m be the number of occurrences of $\bar{x}$ in t. If $n = 0$ replace every occurence of $\bar{x}$ by $\bot$, and if $m = 0$ replace every occurrence of x by $\bot$. Otherwise, introduce $2mn$ fresh (positive) variables $x'_{i,j}, x''_{i,j}$ for $1 \leq i \leq n$ and $1 \leq j \leq m$. Now, for $1 \leq i \leq n$, replace the ith occurrence of x by $x'_{i,1} \vee \ldots \vee x'_{i,m}$ and, for $1 \leq j \leq m$, replace the jth occurrence of $\bar{x}$ by $x''_{1,j} \vee \ldots \vee x''_{n,j}$.

Now t' is a linear term (without negation), and its size is quadratic in the size of t. Let s' be the conjunction of all pairs $x' \vee x''$ of variables introduced in the construction of t'. Clearly $\mathit{Var}(s') = \mathit{Var}(t')$ and s' is also a linear term of the same size as t'. Furthermore, t is a tautology if and only if $s' \to t'$ is sound. To see this, let s'' and t'' be obtained from s' and t', respectively, by replacing each x'' by $\bar{x}'$. Then s'' always evaluates to 1, and t'' is a tautology if and only if t is a tautology. ◀

3.2 Read-once functions and linear terms

Linear terms compute what are known as "read-once" Boolean functions, and we survey some of their theory in this section.

▶ **Definition 8** (Read-once functions)**.** A Boolean function is *read-once* if it is computed by some linear term (of propositional logic).

It is not exactly clear when the following result first appeared, although we refer to a discussion in [7] where it is stated that results directly implying this were first mentioned in [19]. The result also occurs in [14], and is generalised to certain other bases in [16] and [17].

▶ **Theorem 9.** *Constant-free negation-free linear terms compute the same (read-once) Boolean function if and only if they are equivalent modulo AC.*

A proof of this can easily be derived from results in Sect. 4, by the presentation of equivalence classes modulo AC as relation webs and the graph-theoretic definition of soundness.

The following consequences of Thm. 9 appear in [9], where detailed proofs may be found.

▶ **Corollary 10.** *Negation-free linear terms compute the same (read-once) Boolean function if and only if they are equivalent modulo ACU'.*

Proof idea. The result essentially follows from the observation that every negation-free term is ACU'-equivalent to $\bot$, $\top$ or a unique constant-free term [8]. ◀

▶ **Corollary 11.** *Any sound negation-free linear TRS, modulo ACU', is terminating in exponential-time.*

Proof. The result follows by Boolean semantics and the preceding corollary: each consequent term must compute a distinct Boolean function that is strictly bigger, under $\leq$, and the graph of $\leq$ has length 2^n, where n is the number of variables in the input term. ◀

3.3 Minterms and maxterms

In this section we restrict our attention to *monotone* Boolean functions, i.e., those functions $f\colon \{0,1\}^X \to \{0,1\}$ such that $Y \subseteq Y' \subseteq X$ implies $f(Y) \leq f(Y')$. We point out the observation that negation-free terms compute monotone Boolean functions.

Minterms and maxterms correspond to minimal DNF and CNF representations, respectively, of a monotone Boolean function. We refer the reader to [7] for an introduction to their theory. In this work we use them in a somewhat different way to Boolean function theory, in that we devise definitions of logical concepts, such as soundness and, later in Sect. 5, what we call "triviality". The reason for this is to take advantage of the purely function-theoretic results stated in this section (e.g. Gurvich's Thm. 14 below) to derive our main results.

▶ **Definition 12.** Let f be a monotone Boolean function on a variable set X. A set $Y \subseteq X$ is a *minterm* (resp. *maxterm*) for f if it is a minimal set such that $f(Y) = 1$ (resp. $f(\overline{Y}) = 0$). The set of all minterms (resp. maxterms) of f is denoted $MIN(f)$ (resp. $MAX(f)$).

Using these notions, we can now give an alternative definition of soundness.

▶ **Proposition 13** (Soundness via minterms or maxterms). *For monotone Boolean functions f, g on the same variable set, the following are equivalent:*
1. $f \leq g$.
2. $\forall S \in MIN(f).\ \exists S' \in MIN(g).\ S' \subseteq S$.
3. $\forall T \in MAX(g).\ \exists T' \in MAX(f).\ T' \subseteq T$.

Proof. $1 \implies 2$. Let $f \leq g$ and suppose there is an $S \in MIN(f)$ such that there is no $S' \in MIN(g)$ with $S' \subseteq S$. Then $f(S) = 1$ and $g(S) = 0$, contradicting $f \leq g$.

$2 \implies 1$. Let Y be such that $f(Y) = 1$. Then there is a minterm $S \in MIN(f)$ with $S \subseteq Y$. By 2, there is a minterm $S' \in MIN(g)$ with $S' \subseteq S$, and therefore $S' \subseteq Y$. Therefore $g(Y) = 1$, by monotonicity, and so $f \leq g$.

$1 \implies 3$ and $3 \implies 1$ are proved similarly. ◀

The following classical result is due to Gurvich in [14], but has appeared in various presentations. In particular, the proof appearing in [7] uses the notion of *cooccurrence* graph, to which our "relation webs" in the next section essentially amounts.[7]

▶ **Theorem 14** (Gurvich). *A monotone Boolean function f is read-once if and only if*

$$\forall S \in MIN(f).\ \forall T \in MAX(f).\ |S \cap T| = 1 \quad .$$

4 Relation webs

In this section we restrict our attention to negation-free constant-free linear terms. It will be useful for us to consider not only the Boolean semantics of terms but also their syntactic structure, in the form of *relation webs* [13, 23]. It turns out that many of the same concepts that we have seen in the previous sections can be defined in this setting and the interplay between the two settings is something that we will take advantage of in later results.

[7] Indeed, by the end of Sect. 4 we will have developed enough technology to give a self-contained proof of this result, but that is beyond the scope of this work.

4.1 Preliminary material

We make use of *labelled graphs* with their standard terminology. For a graph G we denote its *vertex set* or set of *nodes* as $V(G)$, and the set of its *labelled edges* as $E(G)$.

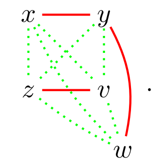
For graphs G and H such that $V(G) \subseteq V(H)$, we say "G in H" to assert that G is an (induced) subgraph[8] of H. In particular we say "$x \overset{\star}{\text{——}} y$ in G" to express that the edge $\{x, y\}$ is labelled $\star$ in the graph G.

We say that a set $X \subseteq V(G)$ is a $\star$-*clique* if every pair $x, y \in X$ has a $\star$-labelled edge between them. A *maximal* $\star$-clique is a $\star$-clique that is not contained in any larger $\star$-clique.

Analysing the term tree of a negation-free constant-free linear term, notice that for each pair of variables x, y, there is a unique connective $\star \in \{\wedge, \vee\}$ at the root of the smallest subtree containing the (unique) occurrences of x and y. Let us call this the *first common connective* of x and y in t.

▶ **Definition 15** (Relation webs). The *(relation) web* $\mathcal{W}(t)$ of a constant-free negation-free linear term t is the complete graph whose vertex set is $\mathit{Var}(t)$, such that the edge between two variables x and y is labelled by their first common connective in t.

As a convention we will write x —— y if the edge $\{x, y\}$ is labelled by $\wedge$, and we write x ⋯⋯ y if it is labelled by $\vee$.

▶ **Example 16.** The term $([x \vee w] \wedge y) \vee (z \wedge v)$ has the relation web  .

▶ Remark (Labels). We point out that, instead of using labelled complete graphs, we could have also used unlabelled arbitrary graphs, since we have only two connectives ($\wedge$ and $\vee$) and so one could be specified by the lack of an edge. This is indeed done in some settings, e.g. the cooccurrence graphs of [7]. However, we use the current formulation in order to maintain consistency with the previous literature, e.g. [13] and [23], and since it helps write certain arguments, e.g. in Sect. 7, where we need to draw graphs with incomplete information.

One of the reasons for considering relation webs is the following proposition, which allows to reason about equivalence classes modulo AC easily. It follows immediately from the definition and that AC preserves first common connectives.

▶ **Proposition 17.** *Constant-free negation-free linear terms are equivalent modulo AC if and only if they have the same web.*

An important property of webs is that they have no minimal paths of length > 2. More precisely, we have the following proposition:

▶ **Proposition 18.** *A complete $\{\wedge, \vee\}$-labelled graph on X is the web of some negation-free constant-free linear term on X if and only if it contains no induced subgraphs of the form:*

(1)

A proof of this property can be found, for example, in [21], [22], [1], or [13]. It is called P_4*-freeness* or *Z-freeness* or *N-freeness*, depending on the viewpoint. We will make crucial use of it when later reasoning with webs.

[8] In fact, since all graphs we deal with are complete, all subgraphs are implicitly induced.

4.2 Relationships to minterms and maxterms

Essentially one can think of relation webs as a graph-theoretic formulation of minterms and maxterms, as opposed to the set-theoretic formulation earlier, in light of the following result:

▶ **Theorem 19.** *A set of variables is a minterm (resp. maxterm) of a negation-free constant-free linear term t if and only if it is a maximal $\wedge$-clique (resp. maximal $\vee$-clique) in $\mathcal{W}(t)$.*

The proof of this follows easily from the following alternative definition of minterms and maxterms, based on structural induction on a term:

▶ **Proposition 20** (Inductive definition of minterms and maxterms)**.** *Let t be a linear term. A set $S \subseteq Var(t)$ is a minterm of t if and only if:*
- $t = x$ *and* $S = \{x\}$.
- $t = t_1 \vee t_2$ *and* S *is a minterm of* t_1 *or of* t_2.
- $t = t_1 \wedge t_2$ *and* $S = S_1 \cup S_2$ *where each* S_i *is a minterm of* t_i.

Dually, a set $T \subseteq Var(t)$ is a maxterm of t if and only if:
- $t = x$ *and* $T = \{x\}$.
- $t = t_1 \vee t_2$ *and* $T = T_1 \cup T_2$ *where each* T_i *is a maxterm of* t_i.
- $t = t_1 \wedge t_2$ *and* T *is a maxterm of* t_1 *or of* t_2.

5 Dealing with constants, negation, erasure and trivialities

In this section we show that we need not deal with linear rules that contain constants or negation when looking for a complete linear system, or linear rules all of whose variables do not occur on both sides. The fundamental concept here is that of "triviality", first introduced in [9] as "semantic triviality". This turns out also to be precisely the concept which allows us to polynomially restrict the length of linear derivations for our main result in Sect. 6.

Many of the following results appeared in [9], so we present only brief arguments here.

5.1 Triviality

The idea behind triviality of a variable in some linear inference is that the inference is "independent" of the behaviour of that variable.

▶ **Definition 21** (Triviality)**.** Let f and g be Boolean functions on a set of variables X, and let $x \in X$. We say $f \to g$ is *trivial* at x if for all $Y \subseteq X$, we have $f(Y \cup \{x\}) \leq g(Y \setminus \{x\})$. We say simply that f is 'trivial' if it is trivial at one of its variables.

▶ Remark (Hereditariness of triviality)**.** Notice that the triviality relation is somehow hereditary: if a sound sequence $f_0 \to f_1 \to \ldots \to f_l$ of Boolean functions is trivial at some point $f_i \to f_{i+1}$ for $0 \leq i < l$ then $f_1 \to f_n$ is trivial. However the converse does not hold: if the first and last function of a sound sequence constitutes a trivial pair it may be that there is no local triviality in the sequence. E.g. the endpoints of the derivation,

$$(w \wedge x) \vee (y \wedge z) \to [w \vee y] \wedge [x \vee z] \to w \vee x \vee (y \wedge z)$$

form a pair that is trivial at w (or trivial at x), but no local step witnesses this. In these cases we call the sequence *globally* trivial. This notion is fundamental later in Lemma 33, on which our main result crucially relies.

In a similar way as we could express soundness with minterms or maxterms in Prop. 13, we can also define triviality with minterms or maxterms.

▶ **Proposition 22.** *The following are equivalent:*
1. $f \to g$ *is trivial at* x*.*
2. $\forall S \in MIN(f).\ \exists S' \in MIN(g).\ S' \subseteq S \setminus \{x\}$*.*
3. $\forall T \in MAX(g).\ \exists T' \in MAX(f).\ T' \subseteq T \setminus \{x\}$*.*

Proof. We first show that $1 \implies 2$. Assume $f \to g$ is trivial at x, and let $S \in MIN(f)$. We have $f(S) = 1$, and hence also $f(S \cup \{x\}) = 1$. By way of contradiction assume there is no $S' \in MIN(g)$ with $S' \subseteq S \setminus \{x\}$. Therefore $g(S \setminus \{x\}) = 0$, contradicting triviality at x. Next, we show $2 \implies 1$. For this, let Y be such that $f(Y \cup \{x\}) = 1$. Then there is a minterm $S \in MIN(f)$ with $S \subseteq Y \cup \{x\}$. By 2, there is a minterm $S' \in MIN(g)$ with $S' \subseteq S \setminus \{x\}$. Hence $S' \subseteq Y \setminus \{x\}$. Therefore $g(Y \setminus \{x\}) = 1$, and thus $f \to g$ is trivial at x. To show $1 \implies 3$ and $3 \implies 1$ we proceed analogously. ◀

We now present a series of results illustrating that we need not consider trivial derivations in any linear system containing certain rules. These results are then used to show that constants and negation are similarly unimportant.

▶ **Definition 23.** We define the following rules:

$$\mathsf{s} : x \wedge [y \vee z] \to (x \wedge y) \vee z \quad , \quad \mathsf{m} : (w \wedge x) \vee (y \wedge z) \to [w \vee y] \wedge [x \vee z]$$

We call the former *switch* and the latter *medial* [2].

In what follows we implicitly assume that rewriting is conducted modulo ACU.

▶ **Lemma 24.** *If* s, t *are negation-free linear terms on* $x_1, \dots, x_n$ *and* $s \leq t$*, then there are terms* s', t', u *such that:*
1. *There are derivations* $s \xrightarrow[\mathsf{s,m}]{*} s' \vee u$ *and* $t' \vee u \xrightarrow[\mathsf{s,m}]{*} t$ *of length* $O(n^2)$*.*
2. $s' \to t'$ *is sound and nontrivial.*

Proof. See [9]. Briefly, the idea is that u is obtained by repeatedly 'moving aside' trivial variables, using s, m and ACU, until there are no trivialities remaining in $s' \to t'$. ◀

▶ **Theorem 25.** *Let* R *be a complete linear system. If* $s \xrightarrow[R]{*} t$ *then there is an* R*-derivation from* s *to* t *with only* $O(|s|^2)$*-many steps whose redex and contractum constitute a triviality.*

Proof. Apply the lemma above to generate terms s', t', u as above. Since R is complete there must be a derivation of $s' \to t'$, and this cannot contain any trivialities by the hereditariness property (cf. Rmk. 5.1) and the fact that $s' \to t'$ is nontrivial.

Therefore the only steps whose redex and contractum form a trivial pair are those generated by 1 in Lemma 24 above, whence we know that the number of such steps is quadratic in the number of variables. ◀

5.2 Erasing and introducing rules

A left- and right-linear rewrite rule may still erase or introduce variables, i.e. there may be variables on one side that do not occur on the other. However, notice that any such situation must constitute a triviality at such a variable, since the soundness of the step is not dependent on the value of that variable.

▶ **Proposition 26.** *Suppose* $\rho : l \to r$ *is linear, and there is some variable* x *occurring in only one of* l *and* r*. Then* ρ *is trivial at* x*.*

5.3 Negation

If a (positive) variable x occurs negatively on both sides of a linear rule then $\bar{x}$ can be replaced soundly by x on both sides. Otherwise, if x occurs positively on one side and negatively on the other, it must be that we have a triviality at x.

▶ **Proposition 27.** *For each linear rule ρ either there is a negation-free linear rule that is equivalent to ρ (i.e. with the same reduction steps), or ρ is trivial.*

5.4 Constants

Let us assume in this subsection that terms are negation-free, in light of Prop. 27 above.

Recall that ACU' preserves the Boolean function computed by a term, and that every linear term is equivalent to $\bot$, $\top$ or a unique constant-free linear term.

▶ **Theorem 28.** *Let R be a complete linear system. Then any constant-free nontrivial linear inference $s \to t$ has a constant-free R/ACU'-derivation.*

Proof. By completeness there is an R-derivation of $s \to t$. Now reduce every line by ACU' to a constant-free term or $\bot$ or $\top$ (e.g. as shown in [9]). If some line reduces to $\bot$ or $\top$ and another does not, then $s \to t$ is trivial, and if every line reduces to $\bot$ or every line reduces to $\top$ then the derivation collapses and is no longer constant-free. ◀

5.5 Putting it together

Combining the various results of this section we obtain the following:

▶ **Theorem 29.** *The following are equivalent:*
1. *There is a sound linear system complete for* L.
2. *There is a sound constant-free negation-free nontrivial linear system, whose rules have the same variables on both sides, complete for the set of such inferences.*

6 Main results

In light of Thm. 29 in the previous section, we assume the following throughout this section:

Terms are constant-free, negation-free and linear on a variable set X of size n.

The following is our main result.

▶ **Main Theorem 30.** *For every sequence of terms $s = t_0 < t_1 < \cdots < t_l = t$ we have that:*
1. *$l = O(n^4)$; or,*
2. *$s \to t$ is trivial.*

Before giving a proof, we show how this implies that there is no sound and complete linear system, modulo hardness assumptions.

▶ **Corollary 31.** *If there is a sound and complete linear system, then there is one that has a $O(n^4)$-length derivation for each linear inference on n variables.*

Proof. This follows from Thm. 30, Lemma 24 and Thm. 29. ◀

▶ **Corollary 32.** *There is no sound linear system complete for* L *unless* **coNP** = **NP**.

Proof. L is **coNP**-complete, by Prop. 7, and so Cor. 31 induces an **NP** decision procedure for L for any such system R: guess a correct sequence of R-steps to derive $s \to t$. ◀

In the next section we give the crucial lemma that allows us to obtain a proof of our main theorem. The argument itself is outlined in the section thereafter.

6.1 Critical minterms and maxterms

For this section, let us fix a sequence $f = f_0 < f_1 < \cdots < f_l = g$ of strictly increasing read-once Boolean functions on a variable set X.

Here we show that, unless $f \to g$ is trivial, for each variable $x \in X$ we must be able to associate a minterm S^x of f such that, for any $S \subseteq S^x$ that is a minterm of some f_i, it must be that $S \ni x$. We simultaneously show the dual property for maxterms.

▶ **Lemma 33** (Subset and intersection lemma). *Suppose $f \to g$ is not trivial. For every variable $x \in X$, there is a minterm S^x of f and a maxterm T^x of g such that:*
1. $\forall S_i \in MIN(f_i).S_i \subseteq S^x \implies x \in S_i$.
2. $\forall T_i \in MAX(g_i).T_i \subseteq T^x \implies x \in T_i$.
3. $\forall S_i \in MIN(f_i), \forall T_i \in MAX(g_i).S_i \subseteq S^x, T_i \subseteq T^x \implies S_i \cap T_i = \{x\}$.

Proof. Suppose that, for some variable x no minterm of f has property 1. In other words, for every minterm S^x of f containing x there is some minterm S_i of some f_i that is a subset of S^x yet does not contain x. Since $f_i \to f_l$ is sound for every i we have that, by Prop. 13, for every minterm S^x of f containing x there is some minterm S_l of $f_l = g$ that is a subset of S^x not containing x. I.e. $f \to g$ is trivial, by Prop. 22, which is a contradiction. Property 2 is proved analogously. Finally, Property 3 is proved by appealing to read-onceness. Any such S_i and T_i must contain x by properties 1 and 2, yet their intersection must be a singleton by Thm. 14 since all f_i are read-once, whence the result follows. ◀

We notice that, since some S_i and T_i must exist for all i, by soundness, we can build a chain[9] of such minterms and maxterms preserving the intersection point. For a given derivation, let us call a choice of such minterms and maxterms *critical*.

6.2 Proof of the main result, Thm. 30

Throughout this section let us fix a sound (negation-free constant-free) linear system R, which we assume to contain s, m,[10] whose reduction relation, modulo AC, is $\underset{R}{\longrightarrow}$.

Recall that $s \underset{R}{\longrightarrow} t$ implies that s, t are distinct modulo AC so compute distinct Boolean functions by Thm. 14 and have distinct relation webs. Let us fix a nontrivial R-derivation,

$$\pi \quad : \qquad s = t_0 \underset{R}{\longrightarrow} t_1 \underset{R}{\longrightarrow} \cdots \underset{R}{\longrightarrow} t_l = t$$

Now, let us fix for each $x \in X$ and $0 \leq i \leq l$ choices S_i^x and T_i^x of critical minterms and maxterms, respectively, of t_i, by Lemma 33. I.e. we have that, for each $x \in X$:

1. $S_i^x \cap T_i^x = \{x\}$ for each $i \leq l$.
2. $S_0^x \supseteq S_1^x \supseteq \cdots \supseteq S_l^x$.
3. $T_0^x \subseteq T_1^x \subseteq \cdots \subseteq T_l^x$.

---

[9] More generally we can build lattices of these terms since the properties are universally quantified.
[10] If a linear system is complete, then it must derive s and m with fixed size derivations.

First, we give a definition of the measures we will use to deduce the bound of Thm. 30.

▶ **Definition 34** (Measures). For each term t_i in π we define the following measures:
1. $e_\wedge(t_i)$ (resp. $e_\vee(t_i)$) is the number of $\wedge$- (resp. $\vee$-) labelled edges in $\mathcal{W}(t)$.[11]
2. $\nu^x(t_i)$ (resp. $\mu^x(t_i)$) is the size of the citical minterm (resp. maxterm) of x at t_i, i.e. $|S_i^x|$ (resp. $|T_i^x|$).
3. $\nu(t_i) := \sum_{x\in X} \nu^x(t_i)$ and $\mu(t_i) := \sum_{x\in X} \mu^x(t_i)$.

We point out some simple properties of these measures.

▶ **Proposition 35.** *Let $e := \frac{1}{2}n(n-1)$. We have the following:*
1. *$e_\wedge, e_\vee \le e$, and $e_\wedge + e_\vee = e$.*
2. *For each $x \in X$ we have that $\nu^x, \mu^x \le n$, so $\nu, \mu \le n^2$.*

Proof. 1 follows from the fact that there are only e edges in a web, all of which must be labelled $\wedge$ or $\vee$. For 2, simply observe that a minterm or maxterm has size at most n. ◀

We show that, whenever an $\wedge$-edge becomes labelled $\vee$, some minterm strictly decreases.

▶ **Proposition 36.** *Suppose, for some $i < l$, we have that x —— y in $\mathcal{W}(t_i)$ and $x \cdots\cdots y$ in $\mathcal{W}(t_{i+1})$. Then there is a minterm S of t_i, and a minterm S' of t_{i+1} such that $S' \subsetneq S$.*

Proof. Take any maximal $\wedge$-clique in $\mathcal{W}(t_i)$ containing x and y, of which there must be at least one. This must have a $\wedge$-subclique which is maximal in $\mathcal{W}(t_{i+1})$, by Prop. 13 and Thm. 19. This subclique cannot contain both x and y, so the inclusion must be strict. ◀

We show that, if a minterm strictly decreases in size, some critical maxterm must strictly increase in size.

▶ **Proposition 37.** *Suppose for $j > i$ there is some minterm S_i of t_i and some minterm S_j of t_j such that $S_j \subsetneq S_i$. Then, for some variable $x \in X$, we have that $T_i^x \subsetneq T_j^x$.*

Proof. We let x be some variable in $x \in S_i \setminus S_j$, which must be nonempty by hypothesis. By Thm. 14 we have that $|T_i^x \cap S_i| = 1$, so it must be that $T_i^x \cap S_i = \{x\}$ by construction.

On the other hand we also have that $|T_j^x \cap S_j| = 1$, and so there is some (unique) $y \in T_j^x \cap S_j$. Now, since $S_i \supsetneq S_j$ we must have $y \in S_i$. However we cannot have $y \in T_i^x$ since that would imply that $\{x, y\} \subset T_i^x \cap S_i$, contradicting the above.

Finally, by soundness, we have that $T_i^x \supsetneq T_j^x$ as required. ◀

Recall that all $\mathcal{W}(t_i)$ are distinct, so both $e_\wedge$ and $e_\vee$ must change at each step of π.

▶ **Lemma 38** (Increasing measure). *The lexicographical product $\mu \times e_\wedge$ is strictly increasing at each step of π.*

Proof. Notice that, by Lemma 33.2, we have that $T_0^x \subseteq T_1^x \subseteq \cdots \subseteq T_l^x$, i.e. μ is non-decreasing. So let us consider the case that $e_\wedge$ decreases at some step and show that μ must strictly increase. If $e_\wedge(t_i) > e_\wedge(t_{i+1})$ then we must have that some edge is labelled $\wedge$ in $\mathcal{W}(t_i)$ and labelled $\vee$ in $\mathcal{W}(t_{i+1})$. Hence, by Prop. 36 some minterm has strictly decreased in size and so by Prop. 37 some critical maxterm must have strictly increased in size. ◀

From here we can finally give a simple proof of our main result:

[11] Of course, these measures can more generally be defined for any linear term.

Proof of Thm. 30. By Prop. 35 we have that $\mu = O(n^2) = e_\wedge$ and so, since $s \to t$ is nontrivial, it must be that the length l of π is $O(n^4)$, as required. ◀

Notice that, while the various settings exhibit a symmetry between $\wedge$ and $\vee$, it is the property of soundness that induces the necessary asymmetry required to achieve this result.

7 Canonicity

We show that the medial rule is somehow “canonical”: it is the *only* linear inference that, on relation webs, preserves $\wedge$-edges (up to reflexive transitive closure modulo AC).

On the other hand, the switch rule is not canonical, in the sense that it is not the only rule that preserves $\vee$-edges, and we give an example of this from previous work. However we conjecture a weaker form of canonicity for the switch rule.

7.1 Canonicity of medial

▶ **Definition 39.** Let s and t be linear terms on a set X of variables. We write $s \lhd\!\!\rhd t$ if:

1. Whenever x —— y in $\mathcal{W}(s)$ we have that x —— y in $\mathcal{W}(t)$.
2. Whenever x ······ y in $\mathcal{W}(s)$ and x —— y in $\mathcal{W}(t)$, there are $w, z \in X$ such that,

$$\begin{array}{ccc} w & \text{——} & x \\ \vdots & \text{(dotted cross)} & \vdots \\ y & \text{——} & z \end{array} \text{ in } \mathcal{W}(s) \quad \text{and} \quad \begin{array}{ccc} w & \text{——} & x \\ \vdots & \text{(solid cross)} & \vdots \\ y & \text{——} & z \end{array} \text{ in } \mathcal{W}(t).$$

The following result appeared in [23], where a detailed proof may be found.

▶ **Proposition 40** (Medial criterion). *$s \lhd\!\!\rhd t$ if and only if $s \xrightarrow[\mathsf{m}]{*} t$.*

▶ **Definition 41.** If t is a linear term with $x, y, z \in Var(t)$, we say that y *separates* x *from* z in $\mathcal{W}(t)$ if x —— y in $\mathcal{W}(t)$ and y ······ z in $\mathcal{W}(t)$.

▶ **Theorem 42.** *Let s and t be linear terms on a variable set X. The following are equivalent:*

1. *$s \leq t$ and for all $x, y \in X$ we have x —— y in $\mathcal{W}(s)$ implies x —— y in $\mathcal{W}(t)$.*
2. *$s \lhd\!\!\rhd t$.*
3. *$s \xrightarrow[\mathsf{m}]{*} t$.*

Proof. We have that $2 \Longrightarrow 3$ by Prop. 40 and $3 \Longrightarrow 1$ by inspection of medial, so it suffices to show $1 \Longrightarrow 2$. For this, assume 1 and suppose x ······ y in $\mathcal{W}(s)$ and x —— y in $\mathcal{W}(t)$, and let S be a minterm of s containing x. We must have $S \supsetneq \{x\}$ since x —— y in $\mathcal{W}(t)$ and $s \to t$ is sound.[12] Similarly there must be a maxterm T of t containing y such that $T \supsetneq \{y\}$. Now, by 1, it must be that S (resp. T) is also a minterm (resp. maxterm) of t (resp. s),[13] and so, by Thm. 14, there is some (unique) $z \in S \cap T$ which, by definition, separates x from y in both $\mathcal{W}(s)$ and $\mathcal{W}(t)$. By a symmetric argument we obtain a w separating y

[12] Recall that, by Prop. 13 and Thm. 19, there must a subset of S which is a maximal $\wedge$-clique in $\mathcal{W}(t)$.

[13] Since by 1, $\wedge$-edges (resp. $\vee$-edges) are preserved left-to-right (resp. right-to-left) and so $\wedge$-cliques (resp. $\vee$-cliques) must be preserved (resp. reflected). Of course, these must be maximal by soundness.

from x in both $\mathcal{W}(s)$ and $\mathcal{W}(t)$. By construction, w and z must be distinct, so we have the following situation,

$$\begin{array}{ccc} x & \text{—} & z \\ \vdots & \ddots & \vdots \\ w & \text{—} & y \end{array} \text{ in } \mathcal{W}(s) \quad \text{and} \quad \begin{array}{ccc} x & \text{—} & z \\ \vdots & \diagdown & \vdots \\ w & \text{—} & y \end{array} \text{ in } \mathcal{W}(t).$$

whence 2 follows by P_4-freeness. ◀

▶ **Corollary 43.** *The bound in Thm. 30.1 can be improved to $O(n^3)$.*

For the proof, let us first define $\#_\wedge(t)$ to be the number of $\wedge$ symbols occurring in t.

Proof of Cor. 43. Instead of using $e_\wedge$ in Lemma 38, use $\#_\wedge$, which is linear in the size of the term. If no $\wedge$-edge becomes labelled $\vee$, $\#_\wedge$ must have strictly decreased by Thm. 42. ◀

7.2 Towards canonicity of switch

Switch is not canonical in the same sense, due to the following example appearing in [9]:

$$\begin{array}{rl} & ([z \vee v] \wedge [x \vee (z' \wedge v')]) \vee ([(y \wedge u) \vee w] \wedge [y' \vee u']) \\ \to & [x \vee (y \wedge y')] \wedge [(z \wedge z') \vee (u \wedge u')] \wedge [(v \wedge v') \vee w] \end{array} \tag{2}$$

For this inference, no $\vee$-edge becomes a $\wedge$-edge, but it is not derivable by switch and medial, as shown in [9]. However, we conjecture that a weaker form of canonicity applies.

▶ **Conjecture 44.** *If $s \to t$ is sound and nontrivial, every $\vee$-edge in $\mathcal{W}(s)$ is also labelled $\vee$ in $\mathcal{W}(t)$, $s \to t$, and $\#_\wedge(s) = \#_\wedge(t)$, then $s \xrightarrow[\mathsf{s}]{*} t$.*

8 Final remarks

Conjecture 44 above is inspired by the observation that the only nontrivial linear inference we know of that preserves $\#_\wedge$ is s. There are known trivial examples (e.g. "supermix" from [9] : $x \wedge (y_1 \vee \cdots \vee y_k) \to x \vee (y_1 \wedge \cdots \wedge y_k)$) that *increase* $\#_\wedge$ but every nontrivial rule we know of, including the rule (2) above, strictly decreases it.

Notice that, the stronger conjecture that s is the only nontrivial rule that preserves $\#_\wedge$ already implies our main result, since $\#_\wedge \times e_\wedge$ would be a strictly decreasing measure.

We point out that this measure is used for the usual proof of termination of $\{\mathsf{s}, \mathsf{m}\}$ (modulo AC), e.g. in [9], and also yields a cubic bound on termination. In this work we have matched that bound for *all* linear derivations in the case of weak normalisation, and in the case of strong normalisation for derivations (modulo ACU) that are not globally trivial.

Finally, some preliminary research has shown that the length-bound for termination of $\{\mathsf{s}, \mathsf{m}\}$ can be improved to a quadratic. We conjecture that such an improvement is also possible in the case of (nontrivial) linear derivations in general.

References

1 Denis Bechet, Philippe de Groote, and Christian Retoré. A complete axiomatisation of the inclusion of series-parallel partial orders. In H. Common, editor, *Rewriting Techniques and Applications, RTA 1997*, volume 1232 of *LNCS*, pages 230–240. Springer, 1997.

2 Kai Brünnler and Alwen F. Tiu. A local system for classical logic. In R. Nieuwenhuis and A. Voronkov, editors, *LPAR 2001*, volume 2250 of *LNCS*, pages 347–361. Springer, 2001.

3 Paola Bruscoli and Alessio Guglielmi. On the proof complexity of deep inference. *ACM Transactions on Computational Logic*, 10(2):1–34, 2009. Article 14.
4 Michael Chein. Algorithmes d'écriture de fonctions booléennes croissantes en sommes et produits. *Revue Française d'Informatique et de Recherche Opérationnelle*, 1:97–105, 1967.
5 Stephen Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.
6 Stephen Cook and Robert Reckhow. On the lengths of proofs in the propositional calculus (preliminary version). In *Proceedings of the 6th annual ACM Symposium on Theory of Computing*, pages 135–148. ACM Press, 1974.
7 Yves Crama and Peter L Hammer. *Boolean functions: Theory, algorithms, and applications.* Cambridge University Press, 2011.
8 Anupam Das. On the proof complexity of cut-free bounded deep inference. In K. Brünnler and G. Metcalfe, editors, *Tableaux 2011*, volume 6793 of *LNAI*, pages 134–148, 2011.
9 Anupam Das. Rewriting with linear inferences in propositional logic. In Femke van Raamsdonk, editor, *RTA'13*, volume 21 of *LIPIcs*, pages 158–173, 2013.
10 Nachum Dershowitz and Jieh Hsiang. Rewrite methods for clausal and non-clausal theorem proving. In *Automata, Languages and Programming*, pages 331–346. Springer, 1983.
11 Moshe Dubiner and Uri Zwick. Amplification by read-once formulas. *SIAM Journal on Computing*, 26(1):15–38, 1997.
12 A. Guglielmi and L. Straßburger. Non-commutativity and MELL in the calculus of structures. In L. Fribourg, editor, *CSL 2001*, volume 2142 of *LNCS*, pages 54–68, 2001.
13 Alessio Guglielmi. A system of interaction and structure. *ACM Transactions on Computational Logic*, 8(1):1–64, 2007.
14 V. A. Gurvich. Repetition-free boolean functions. *Uspekhi Matematicheskikh Nauk*, 32(1):183–184, 1977.
15 V. A. Gurvich. On the normal form of positional games. In *Soviet math. dokl*, volume 25, pages 572–574, 1982.
16 Rafi Heiman, Ilan Newman, and Avi Wigderson. On read-once threshold formulae and their randomized decision tree complexity. In *Theoret. Comp. Science*, pages 78–87, 1994.
17 Lisa Hellerstein and Marek Karpinski. Computational complexity of learning read-once formulas over different bases. Technical report, University of Bonn, 1990.
18 Jieh Hsiang. Refutational theorem proving using term-rewriting systems. *Artificial Intelligence*, 25(3):255–300, 1985.
19 Aleksandr Vasilevich Kuznetsov. Non-repeating contact schemes and non-repeating superpositions of functions of algebra of logic. *Trudy Matematicheskogo Instituta im. VA Steklova*, 51:186–225, 1958.
20 Leonid A. Levin. Universal sequential search problems. *Problemy Peredachi Informatsii*, 9(3):115–116, 1973.
21 Rolf H. Möhring. Computationally tractable classes of ordered sets. In I. Rival, editor, *Algorithms and Order*, pages 105–194. Kluwer Acad. Publ., 1989.
22 Christian Retoré. *Réseaux et Séquents Ordonnés.* PhD thesis, Université Paris VII, 1993.
23 Lutz Straßburger. A characterisation of medial as rewriting rule. In Franz Baader, editor, *RTA 2007*, volume 4533 of *LNCS*, pages 344–358. Springer-Verlag, 2007.
24 Lutz Straßburger. Extension without cut. *Ann. Pure Appl. Logic*, 163(12):1995–2007, 2012.
25 Terese. *Term rewriting systems.* Cambridge University Press, 2003.
26 L. G. Valiant. Short monotone formulae for the majority function. *Journal of Algorithms*, 5(3):363 – 366, 1984.

A Coinductive Framework for Infinitary Rewriting and Equational Reasoning

Jörg Endrullis[1], Helle Hvid Hansen[2], Dimitri Hendriks[1], Andrew Polonsky[3], and Alexandra Silva[4]

1 **Department of Computer Science, VU University Amsterdam, The Netherlands,** `{j.endrullis | r.d.a.hendriks}@vu.nl`
2 **Department of Engineering Systems and Services, Delft University of Technology, The Netherlands,** `h.h.hansen@tudelft.nl`
3 **Institut Galilée, Université Paris 13, France,** `andrew.polonsky@gmail.com`
4 **Department of Computer Science, Radboud University Nijmegen, The Netherlands,** `alexandra@cs.ru.nl`

Abstract

We present a coinductive framework for defining infinitary analogues of equational reasoning and rewriting in a uniform way. We define the relation $\overset{\infty}{=}$, a notion of infinitary equational reasoning, and $\to^{\infty}$, the standard notion of infinitary rewriting as follows:

$$\overset{\infty}{=} \;:=\; \nu R.\, (=_{\mathcal{R}} \cup \overline{R})^{*}$$
$$\to^{\infty} \;:=\; \mu R.\, \nu S.\, (\to_{\mathcal{R}} \cup \overline{R})^{*} \circ \overline{S}$$

where μ and ν are the least and greatest fixed-point operators, respectively, and where

$$\overline{R} \;:=\; \{\, \langle f(s_1,\ldots,s_n),\, f(t_1,\ldots,t_n)\rangle \mid f \in \Sigma,\, s_1 \mathrel{R} t_1,\ldots,s_n \mathrel{R} t_n \,\} \cup \mathrm{Id}\,.$$

The setup captures rewrite sequences of arbitrary ordinal length, but it has neither the need for ordinals nor for metric convergence. This makes the framework especially suitable for formalizations in theorem provers.

1998 ACM Subject Classification D.1.1 Applicative (Functional) Programming, D.3.1 Formal Definitions and Theory, F.4.1 Mathematical Logic, F.4.2 Grammars and Other Rewriting Systems, I.1.1 Expressions and Their Representation, I.1.3 Languages and Systems

Keywords and phrases Infinitary rewriting, coinduction

Digital Object Identifier 10.4230/LIPIcs.RTA.2015.143

1 Introduction

We present a coinductive framework for defining infinitary equational reasoning and infinitary rewriting in a uniform way. The framework is free of ordinals, metric convergence and partial orders which have been essential in earlier definitions of the concept of infinitary rewriting [11, 26, 29, 25, 24, 3, 2, 4, 18].

Infinitary rewriting is a generalization of the ordinary finitary rewriting to infinite terms and infinite reductions (including reductions of ordinal length greater than ω). For the definition of rewrite sequences of ordinal length, there is a design choice concerning the exclusion of jumps at limit ordinals, as illustrated in the ill-formed rewrite sequence

$$\underbrace{\mathsf{a} \to \mathsf{a} \to \mathsf{a} \to \cdots}_{\omega\text{-many steps}}\; \mathsf{b} \to \mathsf{b}$$

© Jörg Endrullis, Helle Hvid Hansen, Dimitri Hendriks, Andrew Polonsky, and Alexandra Silva;
licensed under Creative Commons License CC-BY
26th International Conference on Rewriting Techniques and Applications (RTA'15).
Editor: Maribel Fernández; pp. 143–159
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

where the rewrite system is $\mathcal{R} = \{\, \mathsf{a} \to \mathsf{a},\, \mathsf{b} \to \mathsf{b} \,\}$. The rewrite sequence remains for ω steps at a and in the limit step 'jumps' to b. To ensure connectedness at limit ordinals, the usual choices are:

(i) *weak convergence* (also called 'Cauchy convergence'), where it suffices that the sequence of terms converges towards the limit term, and

(ii) *strong convergence*, which additionally requires that the 'rewriting activity', i.e., the depth of the rewrite steps, tends to infinity when approaching the limit.

The notion of strong convergence incorporates the flavor of 'progress', or 'productivity', in the sense that there is only a finite number of rewrite steps at every depth. Moreover, it leads to a more satisfactory metatheory where redex occurrences can be traced over limit steps.

While infinitary rewriting has been studied extensively, notions of infinitary equational reasoning have not received much attention. One of the few works in this area is [24] by Kahrs, see *Related Work* below. The reason is that the usual definition of infinitary rewriting is based on ordinals to index the rewrite steps, and hence the rewrite direction is incorporated from the start. This is different for the framework we propose here, which enables us to define several natural notions: infinitary equational reasoning, bi-infinite rewriting, and the standard concept of infinitary rewriting. All of these have strong convergence 'built-in'.

We define *infinitary equational reasoning* with respect to a system of equations $\mathcal{R}$, as a relation $\stackrel{\infty}{=}$ on potentially infinite terms by the following mutually coinductive rules:

$$\frac{s\ (=_{\mathcal{R}} \cup \stackrel{\infty}{\leftrightarrows})^*\ t}{s \stackrel{\infty}{=} t} \qquad\qquad \frac{s_1 \stackrel{\infty}{=} t_1 \quad \cdots \quad s_n \stackrel{\infty}{=} t_n}{f(s_1, s_2, \ldots, s_n) \stackrel{\infty}{\leftrightarrows} f(t_1, t_2, \ldots, t_n)} \tag{1}$$

The relation $\stackrel{\infty}{\leftrightarrows}$ stands for infinitary equational reasoning below the root. The coinductive nature of the rules means that the proof trees need not be well-founded. Reading the rules bottom-up, the first rule allows for an arbitrary, but finite, number of rewrite steps at any finite depth (of the term tree). The second rule enforces that we eventually proceed with the arguments, and hence the activity tends to infinity.

▶ **Example 1.1.** Let $\mathcal{R}$ consist of the equation $\mathsf{C}(\mathsf{a}) = \mathsf{a}$. We write C^ω to denote the infinite term $\mathsf{C}(\mathsf{C}(\mathsf{C}(\ldots)))$, the solution of the equation $X = \mathsf{C}(X)$. Using the rules (1), we can derive $\mathsf{C}^\omega \stackrel{\infty}{=} \mathsf{a}$ as shown in Figure 1. This is an infinite proof tree as indicated by the loop $\cdots\!>$ in which the sequence $\mathsf{C}^\omega \stackrel{\infty}{\leftrightarrows} \mathsf{C}(\mathsf{a}) =_{\mathcal{R}} \mathsf{a}$ is written by juxtaposing $\mathsf{C}^\omega \stackrel{\infty}{\leftrightarrows} \mathsf{C}(\mathsf{a})$ and $\mathsf{C}(\mathsf{a}) =_{\mathcal{R}} \mathsf{a}$.

$$\frac{\dfrac{\dfrac{\vdots}{\mathsf{C}^\omega \stackrel{\infty}{=} \mathsf{a}}}{\mathsf{C}^\omega \stackrel{\infty}{\leftrightarrows} \mathsf{C}(\mathsf{a})} \quad \mathsf{C}(\mathsf{a}) =_{\mathcal{R}} \mathsf{a}}{\mathsf{C}^\omega \stackrel{\infty}{=} \mathsf{a}}$$

Figure 1 Derivation of $\mathsf{C}^\omega \stackrel{\infty}{=} \mathsf{a}$.

Using the greatest fixed-point constructor ν, we can define $\stackrel{\infty}{=}$ equivalently as follows:

$$\stackrel{\infty}{=} \ := \ \nu R.\, (=_{\mathcal{R}} \cup \overline{R})^*\,, \tag{2}$$

where $\overline{R}$, corresponding to the second rule in (1), is defined by

$$\overline{R} \ := \ \{\, \langle f(s_1, \ldots, s_n),\ f(t_1, \ldots, t_n) \rangle \mid f \in \Sigma,\ s_1\ R\ t_1,\ \ldots, s_n\ R\ t_n \,\} \cup \mathrm{Id}\,. \tag{3}$$

This is a new and interesting notion of infinitary (strongly convergent) equational reasoning.

Now let $\mathcal{R}$ be a term rewriting system (TRS). If we use $\to_{\mathcal{R}}$ instead of $=_{\mathcal{R}}$ in the rules (1), we obtain what we call *bi-infinite rewriting* $\stackrel{\infty}{\rightrightarrows}$:

$$\frac{s\ (\to_{\mathcal{R}} \cup \stackrel{\infty}{\rightharpoondown})^*\ t}{s \stackrel{\infty}{\rightrightarrows} t} \qquad\qquad \frac{s_1 \stackrel{\infty}{\rightrightarrows} t_1 \quad \cdots \quad s_n \stackrel{\infty}{\rightrightarrows} t_n}{f(s_1, s_2, \ldots, s_n) \stackrel{\infty}{\rightharpoondown} f(t_1, t_2, \ldots, t_n)} \tag{4}$$

corresponding to the following fixed-point definition:

$$\overset{\infty}{\to} \;:=\; \nu R.\, (\to_{\mathcal{R}} \cup \overline{R})^* \,. \tag{5}$$

We write $\overset{\infty}{\to}$ to distinguish bi-infinite rewriting from the standard notion $\to^{\infty}$ of (strongly convergent) infinitary rewriting [32]. The symbol ∞ is centered above $\to$ in $\overset{\infty}{\to}$ to indicate that bi-infinite rewriting is 'balanced', in the sense that it allows rewrite sequences to be extended infinitely forwards, but also infinitely backwards. Here backwards does *not* refer to reversing the arrow $\leftarrow_{\varepsilon}$. For example, for $\mathcal{R} = \{\, \mathsf{C}(\mathsf{a}) \to \mathsf{a} \,\}$ we have the backward-infinite rewrite sequence $\cdots \to \mathsf{C}(\mathsf{C}(\mathsf{a})) \to \mathsf{C}(\mathsf{a}) \to \mathsf{a}$ and hence $\mathsf{C}^{\omega} \overset{\infty}{\to} \mathsf{a}$. The proof tree for $\mathsf{C}^{\omega} \overset{\infty}{\to} \mathsf{a}$ has the same shape as the proof tree displayed in Figure 1; the only difference is that $\overset{\infty}{=}$ is replaced by $\overset{\infty}{\to}$ and $\overset{\infty}{\leftrightharpoons}$ by $\overset{\infty}{\rightharpoondown}$. In contrast, the standard notion $\to^{\infty}$ of infinitary rewriting only takes into account forward limits and we do *not* have $\mathsf{C}^{\omega} \to^{\infty} \mathsf{a}$.

We have the following strict inclusions:

$$\to^{\infty} \;\subsetneq\; \overset{\infty}{\to} \;\subsetneq\; \overset{\infty}{=} \,.$$

In our framework, these inclusions follow directly from the fact that the proof trees for $\to^{\infty}$ (see below) are a restriction of the proof trees for $\overset{\infty}{\to}$ which in turn are a restriction of the proof trees for $\overset{\infty}{=}$. It is also easy to see that each inclusion is strict. For the first, see above. For the second, just note that $\overset{\infty}{\to}$ is not symmetric.

Finally, by a further restriction of the proof trees, we obtain the standard concept of (strongly convergent) infinitary rewriting $\to^{\infty}$. Using least and greatest fixed-point operators, we define:

$$\to^{\infty} \;:=\; \mu R.\, \nu S.\, (\to \cup \overline{R})^* \circ \overline{S} \,, \tag{6}$$

where $\circ$ denotes relational composition. Here R is defined inductively, and S is defined coinductively. Thus only the last step in the sequence $(\to \cup \overline{R})^* \circ \overline{S}$ is coinductive. This corresponds to the following fact about reductions σ of ordinal length: every strict prefix of σ must be shorter than σ itself, while strict suffixes may have the same length as σ.

If we replace μ by ν in (6), we get a definition equivalent to $\overset{\infty}{\to}$ defined by (5). To see that it is at least as strong, note that $\mathrm{Id} \subseteq \overline{S}$.

Conversely, $\to^{\infty}$ can be obtained by a restriction of the proof trees obtained by the rules (4) for $\overset{\infty}{\to}$. Assume that in a proof tree using the rules (4), we mark those occurrences of $\overset{\infty}{\rightharpoondown}$ that are followed by another step in the premise of the rule (i.e., those that are not the last step in the premise). Thus we split $\overset{\infty}{\rightharpoondown}$ into $\rightharpoondown^{\infty}$ and $\overset{<}{\rightharpoondown}^{\infty}$. Then the restriction to obtain the relation $\to^{\infty}$ is to forbid infinite nesting of marked symbols $\overset{<}{\rightharpoondown}^{\infty}$. This marking is made precise in the following rules:

$$\frac{s \;(\to \cup \overset{<}{\rightharpoondown}^{\infty})^* \circ \rightharpoondown^{\infty}\; t}{s \to^{\infty} t} \qquad \frac{s_1 \to^{\infty} t_1 \quad \cdots \quad s_n \to^{\infty} t_n}{f(s_1, s_2, \ldots, s_n) \overset{(<)}{\rightharpoondown}^{\infty} f(t_1, t_2, \ldots, t_n)} \qquad \frac{}{s \overset{(<)}{\rightharpoondown}^{\infty} s} \tag{7}$$

Here $\rightharpoondown^{\infty}$ stands for infinitary rewriting below the root, and $\overset{<}{\rightharpoondown}^{\infty}$ is its marked version. The symbol $\overset{(<)}{\rightharpoondown}^{\infty}$ stands for both $\rightharpoondown^{\infty}$ and $\overset{<}{\rightharpoondown}^{\infty}$. Correspondingly, the rule in the middle is an abbreviation for two rules. The axiom $s \rightharpoondown^{\infty} s$ serves to 'restore' reflexivity, that is, it models the identity steps in $\overline{S}$ in (6). Intuitively, $s \overset{<}{\rightharpoondown}^{\infty} t$ can be thought of as an infinitary rewrite sequence below the root, shorter than the sequence we are defining.

We have an infinitary strongly convergent rewrite sequence from s to t if and only if $s \to^{\infty} t$ can be derived by the rules (7) in a (not necessarily well-founded) proof tree without infinite nesting of $\overset{<}{\rightharpoondown}^{\infty}$, that is, proof trees in which all paths (ascending through

the proof tree) contain only finitely many occurrences of $\stackrel{\le}{\to}{}^{\infty}$. The depth requirement in the definition of strong convergence arises naturally in the rules (7), in particular the middle rule pushes the activity to the arguments.

The fact that the rules (7) capture the infinitary rewriting relation $\to^{\infty}$ is a consequence of a result due to [26] which states that every strongly convergent rewrite sequence contains only a finite number of steps at any depth $d \in \mathbb{N}$, in particular only a finite number of root steps $\to_{\varepsilon}$. Hence every strongly convergent reduction is of the form $(\stackrel{\le}{\to}{}^{\infty} \circ \to_{\varepsilon})^{*} \circ \to^{\infty}$ as in the premise of the first rule, where the steps $\stackrel{\le}{\to}{}^{\infty}$ are reductions of shorter length.

We conclude with an example of a TRS that allows for a rewrite sequence of length beyond ω.

▶ **Example 1.2.** We consider the term rewriting system with the following rules:

$$\mathsf{f}(x,x) \to \mathsf{D} \qquad\qquad \mathsf{a} \to \mathsf{C}(\mathsf{a}) \qquad\qquad \mathsf{b} \to \mathsf{C}(\mathsf{b})\,.$$

We then have $\mathsf{a} \to^{\infty} \mathsf{C}^{\omega}$, that is, an infinite reduction from a to C^{ω} in the limit:

$$\mathsf{a} \to \mathsf{C}(\mathsf{a}) \to \mathsf{C}(\mathsf{C}(\mathsf{a})) \to \mathsf{C}(\mathsf{C}(\mathsf{C}(\mathsf{a}))) \to \cdots \to^{\omega} \mathsf{C}^{\omega}\,.$$

Using the proof rules (7), we can derive $\mathsf{a} \to^{\infty} \mathsf{C}^{\omega}$ as shown in Figure 2.

The proof tree in Figure 2 can be described as follows: We have an infinitary rewrite sequence from a to C^{ω} since we have a root step from a to $\mathsf{C}(\mathsf{a})$, and an infinitary reduction below the root from $\mathsf{C}(\mathsf{a})$ to C^{ω}. The latter reduction $\mathsf{C}(\mathsf{a}) \mathrel{\overline{\to}}^{\infty} \mathsf{C}^{\omega}$ is in turn witnessed by the infinitary rewrite sequence $\mathsf{a} \to^{\infty} \mathsf{C}^{\omega}$ on the direct subterms.

$$\dfrac{\mathsf{a} \to_{\varepsilon} \mathsf{C}(\mathsf{a}) \qquad \dfrac{\dfrac{}{\mathsf{a} \to^{\infty} \mathsf{C}^{\omega}}}{\mathsf{C}(\mathsf{a}) \mathrel{\overline{\to}}^{\infty} \mathsf{C}^{\omega}}}{\mathsf{a} \to^{\infty} \mathsf{C}^{\omega}}$$

Figure 2 A reduction $\mathsf{a} \to^{\infty} \mathsf{C}^{\omega}$.

We also have the following reduction, now of length $\omega + 1$:

$$\mathsf{f}(\mathsf{a},\mathsf{b}) \to \mathsf{f}(\mathsf{C}(\mathsf{a}),\mathsf{b}) \to \mathsf{f}(\mathsf{C}(\mathsf{a}),\mathsf{C}(\mathsf{b})) \to \cdots \to^{\omega} \mathsf{f}(\mathsf{C}^{\omega},\mathsf{C}^{\omega}) \to \mathsf{D}\,.$$

That is, after an infinite rewrite sequence of length ω, we reach the limit term $\mathsf{f}(\mathsf{C}^{\omega},\mathsf{C}^{\omega})$, and we then continue with a rewrite step from $\mathsf{f}(\mathsf{C}^{\omega},\mathsf{C}^{\omega})$ to D.

Figure 3 shows how this rewrite sequence $\mathsf{f}(\mathsf{a},\mathsf{b}) \to^{\infty} \mathsf{D}$ can be derived in our setup. We note that the rewrite sequence $\mathsf{f}(\mathsf{a},\mathsf{b}) \to^{\infty} \mathsf{D}$ cannot be 'compressed' to length ω. So there is no reduction $\mathsf{f}(\mathsf{a},\mathsf{b}) \to^{\le\omega} \mathsf{D}$.

$$\dfrac{\dfrac{\dfrac{\text{like Figure 2}}{\mathsf{a} \to^{\infty} \mathsf{C}^{\omega}} \qquad \dfrac{\text{like Figure 2}}{\mathsf{b} \to^{\infty} \mathsf{C}^{\omega}}}{\mathsf{f}(\mathsf{a},\mathsf{b}) \stackrel{\le}{\to}{}^{\infty} \mathsf{f}(\mathsf{C}^{\omega},\mathsf{C}^{\omega})} \qquad \mathsf{f}(\mathsf{C}^{\omega},\mathsf{C}^{\omega}) \to_{\varepsilon} \mathsf{D}}{\mathsf{f}(\mathsf{a},\mathsf{b}) \to^{\infty} \mathsf{D}}$$

Figure 3 A reduction $\mathsf{f}(\mathsf{a},\mathsf{b}) \to^{\infty} \mathsf{D}$.

1.1 Related Work

While a coinductive treatment of infinitary rewriting is not new [7, 22, 19], the previous approaches only capture rewrite sequences of length at most ω. The coinductive framework that we present here captures all strongly convergent rewrite sequences of arbitrary ordinal length.

From the topological perspective, various notions of infinitary rewriting and infinitary equational reasoning have been studied in [24]. The closure operator S_E from [24] is closely related to our notion of infinitary equational reasoning $\stackrel{\infty}{=}$. The operator S_E is defined by

$S_E(R) = (S \circ E)^*(R)$ where $E(R)$ is the equivalence closure of R, and $S(R)$ is the strongly convergent rewrite relation obtained from (single steps) R. Thus $S_E(\to)$ is the repeated closure under equivalence and strongly convergent reduction of $\to$. Although defined in very different ways, we conjecture that the relations $S_E(\to)$ and $\overset{\infty}{\cong}$ typically coincide, and only in rare cases there is a strict inclusion $S_E(\to) \subsetneq \overset{\infty}{\cong}$.

Martijn Vermaat has formalized infinitary rewriting using metric convergence (in place of strong convergence) in the Coq proof assistant [33], and proved that weakly orthogonal infinitary rewriting does not have the property UN of unique normal forms, see [17]. While his formalization could be extended to strong convergence, it remains to be investigated to what extent it can be used for the further development of the theory of infinitary rewriting.

Ketema and Simonsen [27] introduce the notion of 'computable infinite reductions' [27], where terms as well as reductions are computable, and provide a Haskell implementation of the Compression Lemma for this notion of reduction.

1.2 Outline

In Section 2 we introduce infinitary rewriting in the usual way based on ordinals, and with convergence at every limit ordinal. Section 3 is a short explanation of (co)induction and fixed-point rules. The two new definitions of infinitary rewriting $\to^\infty$ based on mixing induction and coinduction, as well as their equivalence, are spelled out in Section 4. Then, in Section 5, we prove the equivalence of these new definitions of infinitary rewriting with the standard definition. In Section 6 we present the above introduced relations $\overset{\infty}{\cong}$ and $\overset{\infty}{\leftrightarrows}$ of infinitary equational reasoning and bi-infinite rewriting. In Section 7 we compare the three relations $\overset{\infty}{\cong}$, $\overset{\infty}{\leftrightarrows}$ and $\to^\infty$. As an application, we show in Section 8 that our framework is suitable for formalizations in theorem provers. We conclude in Section 9.

2 Preliminaries on Term Rewriting

We give a brief introduction to infinitary rewriting. For further reading on infinitary rewriting we refer to [29, 32, 6, 18], for an introduction to finitary rewriting to [28, 32, 1, 5].

A *signature* Σ is a set of symbols f each having a fixed arity $ar(f) \in \mathbb{N}$. Let $\mathcal{X}$ be an infinite set of variables such that $\mathcal{X} \cap \Sigma = \varnothing$. The set $Ter^\infty(\Sigma, \mathcal{X})$ of (finite and) *infinite terms* over Σ and $\mathcal{X}$ is coinductively defined by the following grammar:

$$T ::=^{\text{co}} x \mid f(\underbrace{T, \dots, T}_{ar(f) \text{ times}})\ (x \in \mathcal{X},\ f \in \Sigma)\,.$$

This means that $Ter^\infty(\Sigma, \mathcal{X})$ is defined as the largest set T such that for all $t \in T$, either $t \in \mathcal{X}$ or $t = f(t_1, t_2, \dots, t_n)$ for some $f \in \Sigma$ with $ar(f) = n$ and $t_1, t_2, \dots, t_n \in T$. So the grammar rules may be applied an infinite number of times, and equality on the terms is bisimilarity. See further Section 3 for a brief introduction to coinduction.

We write Id for the identity relation on terms, $\mathrm{Id} := \{\langle s, s\rangle \mid s \in Ter^\infty(\Sigma, \mathcal{X})\}$.

▶ **Remark.** Alternatively, the set $Ter^\infty(\Sigma, \mathcal{X})$ arises from the set of finite terms, $Ter(\Sigma, \mathcal{X})$, by metric completion, using the well-known distance function d defined by $\mathrm{d}(t, s) = 2^{-n}$ if the n-th level of the terms $t, s \in Ter(\Sigma, \mathcal{X})$ (viewed as labeled trees) is the first level where a difference appears, in case t and s are not identical; furthermore, $\mathrm{d}(t, t) = 0$. It is standard that this construction yields $\langle Ter(\Sigma, \mathcal{X}), \mathrm{d}\rangle$ as a metric space. Now infinite terms are obtained by taking the completion of this metric space, and they are represented by infinite

trees. We will refer to the complete metric space arising in this way as $\langle \mathit{Ter}^\infty(\Sigma, \mathcal{X}), \mathrm{d}\rangle$, where $\mathit{Ter}^\infty(\Sigma, \mathcal{X})$ is the set of finite and infinite terms over Σ.

Let $t \in \mathit{Ter}^\infty(\Sigma, \mathcal{X})$ be a finite or infinite term. The set of *positions* $\mathcal{P}os(t) \subseteq \mathbb{N}^*$ *of* t is defined by: $\varepsilon \in \mathcal{P}os(t)$, and $ip \in \mathcal{P}os(t)$ whenever $t = f(t_1, \dots, t_n)$ with $1 \le i \le n$ and $p \in \mathcal{P}os(t_i)$. For $p \in \mathcal{P}os(t)$, the *subterm* $t|_p$ of t at position p is defined by $t|_\varepsilon = t$ and $f(t_1, \dots, t_n)|_{ip} = t_i|_p$. The set of *variables* $\mathcal{V}ar(t) \subseteq \mathcal{X}$ *of* t is $\mathcal{V}ar(t) = \{x \in \mathcal{X} \mid \exists p \in \mathcal{P}os(t).\, t|_p = x\}$.

A *substitution* σ is a map $\sigma : \mathcal{X} \to \mathit{Ter}^\infty(\Sigma, \mathcal{X})$; its domain is extended to $\mathit{Ter}^\infty(\Sigma, \mathcal{X})$ by corecursion: $\sigma(f(t_1, \dots, t_n)) = f(\sigma(t_1), \dots, \sigma(t_n))$. For a term t and a substitution σ, we write $t\sigma$ for $\sigma(t)$. We write $x \mapsto s$ for the substitution defined by $\sigma(x) = s$ and $\sigma(y) = y$ for all $y \neq x$. Let $\Box$ be a fresh variable. A *context* C is a term $\mathit{Ter}^\infty(\Sigma, \mathcal{X} \cup \{\Box\})$ containing precisely one occurrence of $\Box$. For contexts C and terms s we write $C[s]$ for $C(\Box \mapsto s)$.

A *rewrite rule* $\ell \to r$ over Σ and $\mathcal{X}$ is a pair (ℓ, r) of terms $\ell, r \in \mathit{Ter}^\infty(\Sigma, \mathcal{X})$ such that the left-hand side ℓ is not a variable ($\ell \notin \mathcal{X}$), and all variables in the right-hand side r occur in ℓ, $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(\ell)$. Note that we require neither the left-hand side nor the right-hand side of a rule to be finite.

A *term rewriting system (TRS)* $\mathcal{R}$ over Σ and $\mathcal{X}$ is a set of rewrite rules over Σ and $\mathcal{X}$. A TRS induces a rewrite relation on the set of terms as follows. For $p \in \mathbb{N}^*$ we define $\to_{\mathcal{R},p} \subseteq \mathit{Ter}^\infty(\Sigma, \mathcal{X}) \times \mathit{Ter}^\infty(\Sigma, \mathcal{X})$, a *rewrite step at position* p, by $C[\ell\sigma] \to_{\mathcal{R},p} C[r\sigma]$ if C is a context with $C|_p = \Box$, $\ell \to r \in \mathcal{R}$, and $\sigma : \mathcal{X} \to \mathit{Ter}^\infty(\Sigma, \mathcal{X})$. We write $\to_\varepsilon$ for *root steps*, $\to_\varepsilon = \{\, (\ell\sigma, r\sigma) \mid \ell \to r \in \mathcal{R},\ \sigma \text{ a substitution} \,\}$. We write $s \to_{\mathcal{R}} t$ if $s \to_{\mathcal{R},p} t$ for some $p \in \mathbb{N}^*$. A *normal form* is a term without a redex occurrence, that is, a term that is not of the form $C[\ell\sigma]$ for some context C, rule $\ell \to r \in \mathcal{R}$ and substitution σ.

A natural consequence of this construction is the notion of *weak convergence*: we say that $t_0 \to t_1 \to t_2 \to \cdots$ is an infinite reduction sequence with limit t, if t is the limit of the sequence $t_0, t_1, t_2, \dots$ in the usual sense of metric convergence. We use *strong convergence*, which in addition to weak convergence, requires that the depth of the redexes contracted in the successive steps tends to infinity when approaching a limit ordinal from below. So this rules out the possibility that the action of redex contraction stays confined at the top, or stagnates at some finite level of depth.

▶ **Definition 2.1.** A *transfinite rewrite sequence* (of ordinal length α) is a sequence of rewrite steps $(t_\beta \to_{\mathcal{R},p_\beta} t_{\beta+1})_{\beta<\alpha}$ such that for every limit ordinal $\lambda < \alpha$ we have that if β approaches λ from below, then

(i) the distance $\mathrm{d}(t_\beta, t_\lambda)$ tends to 0 and, moreover,

(ii) the depth of the rewrite action, i.e., the length of the position p_β, tends to infinity.

The sequence is called *strongly convergent* if α is a successor ordinal, or there exists a term t_α such that the conditions 1 and 2 are fulfilled for every limit ordinal $\lambda \le \alpha$; we then write $t_0 \to^\infty_{\mathit{ord}} t_\alpha$. The subscript *ord* is used in order to distinguish $\to^\infty_{\mathit{ord}}$ from the equivalent relation $\to^\infty$ as defined in Definition 4.4. We sometimes write $t_0 \to^\alpha_{\mathit{ord}} t_\alpha$ to explicitly indicate the length α of the sequence. The sequence is called *divergent* if it is not strongly convergent.

There are several reasons why strong convergence is beneficial; the foremost being that in this way we can define the notion of *descendant* (also *residual*) over limit ordinals. Also the well-known Parallel Moves Lemma and the Compression Lemma fail for weak convergence, see [31] and [11] respectively.

3 (Co)induction and Fixed Points

We briefly introduce the relevant concepts from (co)algebra and (co)induction that will be used later throughout this paper. For a more thorough introduction, we refer to [21]. There will be two main points where coinduction will play a role, in the definition of terms and in the definition of term rewriting.

Terms are usually defined with respect to a type constructor F. For instance, consider the type of lists with elements in a given set A, given in a functional programming style:

```
type List a = Nil | Cons a (List a)
```

The above grammar corresponds to the type constructor $F(X) = 1 + A \times X$ where the 1 is used as a placeholder for the empty list `Nil` and the second component represents the `Cons` constructor. Such a grammar can be interpreted in two ways: The *inductive* interpretation yields as terms the set of finite lists, and corresponds to the *least fixed point* of F. The *coinductive* interpretation yields as terms the set of all finite or infinite lists, and corresponds to the *greatest fixed point* of F. More generally, the inductive interpretation of a type constructor yields finite terms (with well-founded syntax trees), and dually, the coinductive interpretation yields possibly infinite terms. For readers familiar with the categorical definitions of algebras and coalgebras, these two interpretations amount to defining finite terms as the *initial F-algebra*, and possibly infinite terms as the *final F-coalgebra*.

Formally, term rewriting is a relation on a set T of terms, and hence an element of the complete lattice $L := \mathcal{P}(T \times T)$, the powerset of $T \times T$. Relations on terms can thus be defined using least and greatest fixed points of monotone operators on L. In this setting, an inductively defined relation is a least fixed point $\mu X. F(X)$ of a monotone $F : L \to L$. Dually, a coinductively defined relation is a greatest fixed point $\nu X. F(X)$ of a monotone $F : L \to L$. Coinduction, and similarly induction, can be formulated as proof rules:

$$\frac{X \le F(X)}{X \le \nu Y. F(Y)}(\nu\text{-rule}) \qquad\qquad \frac{F(X) \le X}{\mu Y. F(Y) \le X}(\mu\text{-rule}) \tag{8}$$

These rules express the fact that $\nu Y. F(Y)$ is the greatest post-fixed point of F, and $\mu Y. F(Y)$ is the least pre-fixed point of F.

4 New Definitions of Infinitary Term Rewriting

We present two new definitions of infinitary rewriting $s \to^\infty t$, based on mixing induction and coinduction, and prove their equivalence. In Section 5 we show they are equivalent to the standard definition based on ordinals. We summarize the definitions:

(a) *Derivation Rules.* First, we define $s \to^\infty t$ via a syntactic restriction on the proof trees that arise from the coinductive rules (7). The restriction excludes all proof trees that contain ascending paths with an infinite number of marked symbols.

(b) *Mixed Induction and Coinduction.* Second, we define $s \to^\infty t$ based on mutually mixing induction and coinduction, that is, least fixed points μ and greatest fixed points ν.

In contrast to previous coinductive definitions [7, 22, 19], the setup proposed here captures all strongly convergent rewrite sequences (of arbitrary ordinal length).

Throughout this section, we fix a signature Σ and a term rewriting system $\mathcal{R}$ over Σ. We also abbreviate $T := \mathit{Ter}^\infty(\Sigma, \mathcal{X})$.

▶ **Notation 4.1.** Instead of introducing separate derivation rules for transitivity, we write a reduction of the form $s_0 \rightsquigarrow s_1 \rightsquigarrow \cdots \rightsquigarrow s_n$ as a sequence of single steps:

$$\frac{s_0 \rightsquigarrow s_1 \quad s_1 \rightsquigarrow s_2 \quad \cdots \quad s_{n-1} \rightsquigarrow s_n}{\text{conclusion}}$$

This allows us to write the subproof immediately above a single step.

▶ **Definition 4.2.** For a relation $R \subseteq T \times T$ we define its *lifting* $\overline{R}$ by

$$\overline{R} \;:=\; \{\, \langle f(s_1, \ldots, s_n),\, f(t_1, \ldots, t_n)\rangle \mid f \in \Sigma,\ ar(f) = n\,, s_1 \mathrel{R} t_1, \ldots, s_n \mathrel{R} t_n \,\} \cup \mathrm{Id}\,.$$

4.1 Derivation Rules

▶ **Definition 4.3.** We define the relation $\to^\infty \subset T \times T$ as follows. We have $s \to^\infty t$ if there exists a (finite or infinite) proof tree δ deriving $s \to^\infty t$ using the following five rules:

$$\frac{s\ (\to_\varepsilon \cup \overset{<}{\rightharpoondown}{}^\infty)^* \circ \rightharpoondown^\infty t}{s \to^\infty t}\ \mathsf{split} \qquad \frac{s_1 \to^\infty t_1 \quad \cdots \quad s_n \to^\infty t_n}{f(s_1, s_2, \ldots, s_n) \overset{(<)}{\rightharpoondown}{}^\infty f(t_1, t_2, \ldots, t_n)}\ \mathsf{lift} \qquad \frac{}{s \overset{(<)}{\rightharpoondown}{}^\infty s}\ \mathsf{id}$$

such that δ does not contain an infinite nesting of $\overset{<}{\rightharpoondown}{}^\infty$, that is, such that there exists no path ascending through the proof tree that meets an infinite number of symbols $\overset{<}{\rightharpoondown}{}^\infty$. The symbol $\overset{(<)}{\rightharpoondown}{}^\infty$ stands for $\rightharpoondown^\infty$ or $\overset{<}{\rightharpoondown}{}^\infty$; so the second rule is an abbreviation for two rules; similarly for the third rule.

We give some intuition for the rules in Definition 4.3. The relations $\overset{<}{\rightharpoondown}{}^\infty$ and $\rightharpoondown^\infty$ are infinitary reductions below the root. We use $\overset{<}{\rightharpoondown}{}^\infty$ for constructing parts of the prefix (between root steps), and $\rightharpoondown^\infty$ for constructing a suffix of the reduction that we are defining. When thinking of ordinal indexed rewrite sequences σ, a suffix of σ can have length equal to σ, while the length of every prefix of σ must be strictly smaller than the length of σ. The five rules (split, and the two versions of lift and id) can be interpreted as follows:

(i) The split-rule: the term s rewrites infinitarily to t, $s \to^\infty t$, if s rewrites to t using a finite sequence of (a) root steps, and (b) infinitary reductions $\rightharpoondown^\infty$ below the root (where infinitary reductions preceding root steps must be shorter than the derived reduction).

(ii) The lift-rules: the term s rewrites infinitarily to t below the root, $s \overset{(<)}{\rightharpoondown}{}^\infty t$, if the terms are of the shape $s = f(s_1, s_2, \ldots, s_n)$ and $t = f(t_1, t_2, \ldots, t_n)$ and there exist reductions on the arguments: $s_1 \to^\infty t_1$, …, $s_n \to^\infty t_n$.

(iii) The id-rules allow for the rewrite relations $\overset{(<)}{\rightharpoondown}{}^\infty$ to be reflexive, and this in turn yields reflexivity of $\to^\infty$. For variable-free terms, reflexivity can already be derived using the other rules. For terms with variables, this rule is needed (unless we treat variables as constant symbols).

For an example of a proof tree, we refer to Example 1.2 in the introduction.

4.2 Mixed Induction and Coinduction

The next definition is based on mixing induction and coinduction. The inductive part is used to model the restriction to finite nesting of $\overset{<}{\rightharpoondown}{}^\infty$ in the proofs in Definition 4.3. The induction corresponds to a least fixed point μ, while a coinductive rule to a greatest fixed point ν.

▶ **Definition 4.4.** We define the relation $\to^\infty \subseteq T \times T$ by

$$\to^\infty \;\;:=\;\; \mu R.\, \nu S.\, (\to_\varepsilon \cup \overline{R})^* \circ \overline{S}\,.$$

We argue why $\to^\infty$ is well-defined. Let $L := \mathcal{P}(T \times T)$ be the set of all relations on terms. Define functions $G : L \times L \to L$ and $F : L \to L$ by

$$G(R, S) := (\to_\varepsilon \cup \overline{R})^* \circ \overline{S} \quad \text{and} \quad F(R) := \nu S.\, G(R, S) = \nu S.\, (\to_\varepsilon \cup \overline{R})^* \circ \overline{S}\,. \tag{9}$$

Then we have $\to^\infty \;=\; \mu R.\, F(R) \;=\; \mu R.\, \nu S.\, G(R, S) \;=\; \mu R.\, \nu S.\, (\to_\varepsilon \cup \overline{R})^* \circ \overline{S}$. It can easily be verified that F and G are monotone (in all their arguments). Recall that a function H over sets is monotone if $X \subseteq Y$ implies $H(\ldots, X, \ldots) \subseteq H(\ldots, Y, \ldots)$. Hence F and G have unique least and greatest fixed points.

4.3 Equivalence

We show equivalence of Definitions 4.3 and 4.4. Intuitively, the μR in the fixed point definition corresponds to the nesting restriction in the definition using derivation rules. If one thinks of Definition 4.4 as $\mu R.\, F(R)$ with $F(R) = \nu S.\, G(R, S)$ (see equation (9)), then $F^{n+1}(\varnothing)$ are all infinite rewrite sequences that can be derived using proof trees where the nesting depth of the marked symbol $\overset{<}{\to}{}^\infty$ is at most n.

To avoid confusion we write $\to^\infty_{der}$ for the relation $\to^\infty$ defined in Definition 4.3, and $\to^\infty_{fp}$ for the relation $\to^\infty$ defined in Definition 4.4. We show $\to^\infty_{der} = \to^\infty_{fp}$. Definition 4.3 requires that the nesting structure of $\overset{<}{\to}{}^\infty_{der}$ in proof trees is well-founded. As a consequence, we can associate to every proof tree a (countable) ordinal that allows to embed the nesting structure in an order-preserving way. We use ω_1 to denote the first uncountable ordinal, and we view ordinals as the set of all smaller ordinals (then the elements of ω_1 are all countable ordinals).

▶ **Definition 4.5.** Let δ be a proof tree as in Definition 4.3, and let α be an ordinal. An *α-labeling of δ* is a labeling of all symbols $\overset{<}{\to}{}^\infty_{der}$ in δ with elements from α such that each label is strictly greater than all labels occurring in the subtrees (all labels above).

▶ **Lemma 4.6.** *Every proof tree as in Definition 4.3 has an α-labeling for some $\alpha \in \omega_1$.*

Proof. Let δ be a proof tree and let $L(\delta)$ be the set positions of symbols $\overset{<}{\to}{}^\infty_{der}$ in t. For positions $p, q \in L(\delta)$ we write $p < q$ if p is a strict prefix of q. Then we have that $>$ is well-founded, that is, there is no infinite sequence $p_0 < p_1 < p_2 < \cdots$ with $p_i \in L(\delta)$ $(i \geq 0)$ as a consequence of the nesting restriction on $\overset{<}{\to}{}^\infty_{der}$. The the extension of this well-founded order on $L(t)$ to a total, well-founded order is isomorphic to an ordinal α, and $\alpha < \omega_1$ since $L(t)$ is countable. ◀

▶ **Definition 4.7.** Let δ be a proof tree as in Definition 4.3. We define the *nesting depth* of δ as the least ordinal $\alpha \in \omega_1$ such that δ admits an α-labeling. For every $\alpha \leq \omega_1$, we define a relation $\to^\infty_{\alpha,der} \subseteq \to^\infty_{der}$ as follows: $s \to^\infty_{\alpha,der} t$ whenever $s \to^\infty_{der} t$ can be derived using a proof with nesting depth $< \alpha$. Likewise we define relations $\rightharpoondown^\infty_{\alpha,der}$ and $\overset{<}{\to}{}^\infty_{\alpha,der}$.

As a direct consequence of Lemma 4.6 we have:

▶ **Corollary 4.8.** *We have $\to^\infty_{\omega_1,der} = \to^\infty_{der}$.*

▶ **Theorem 4.9.** *Definitions 4.3 and 4.4 define the same relation, $\to^\infty_{der} = \to^\infty_{fp}$.*

Proof. We begin with $\to^\infty_{fp} \subseteq \to^\infty_{der}$. Recall that $F(\to^\infty_{der})$ is the greatest fixed point of $G(\to^\infty_{der}, _)$, see (9). Also, we have $\rightharpoondown^\infty_{der} = \overset{<}{\to}{}^\infty_{der} = \overline{\to^\infty_{der}}$, and hence

$$F(\to^\infty_{der}) = (\to_\varepsilon \cup \overline{\to^\infty_{der}})^* \circ \overline{F(\to^\infty_{der})} = (\to_\varepsilon \cup \overset{<}{\to}{}^\infty_{der})^* \circ \overline{F(\to^\infty_{der})} \tag{10}$$

$$\overline{F(\to^\infty_{der})} = \mathrm{Id} \cup \{\, \langle f(\vec{s}),\, f(\vec{t}) \rangle \mid \vec{s}\ F(\to^\infty_{der})\ \vec{t} \,\} \tag{11}$$

where $\vec{s}$, $\vec{t}$ abbreviate $s_1, \ldots, s_n$ and $t_1, \ldots, t_n$, respectively, and we write $\vec{s}\ R\ \vec{t}$ if we have $s_1\ R\ t_1, \ldots, s_n\ R\ t_n$. Employing the μ-rule from (8), it suffices to show that $F(\to^{\infty}_{der}) \subseteq \to^{\infty}_{der}$. Assume $\langle s, t\rangle \in F(\to^{\infty}_{der})$. Then $\langle s, t\rangle \in (\to_{\varepsilon} \cup \overset{<}{\rightharpoondown}{}^{\infty}_{der})^* \circ \overline{F(\to^{\infty}_{der})}$. Then there exists s' such that $s\ (\to_{\varepsilon} \cup \overset{<}{\rightharpoondown}{}^{\infty}_{der})^*\ s'$ and $s'\ \overline{F(\to^{\infty}_{der})}\ t$. Now we distinguish cases according to (11):

$$\dfrac{s\ (\to_{\varepsilon} \cup \overset{<}{\rightharpoondown}{}^{\infty}_{der})^*\ t \qquad \dfrac{}{t \rightharpoondown^{\infty} t}\ \mathsf{id}}{s \to^{\infty} t}\ \mathsf{split} \qquad\qquad \dfrac{s\ (\to_{\varepsilon} \cup \overset{<}{\rightharpoondown}{}^{\infty}_{der})^*\ s' \qquad \dfrac{T_1 \quad \cdots \quad T_n}{s' \rightharpoondown^{\infty} t}\ \mathsf{lift}}{s \to^{\infty} t}\ \mathsf{split}$$

Here, for $i \in \{1, \ldots, n\}$, T_i is the proof tree for $s_i \to^{\infty} t_i$ obtained from $s_i\ F(\to^{\infty}_{der})\ t_i$ by corecursively applying the same procedure.

Next we show that $\to^{\infty}_{der} \subseteq \to^{\infty}_{fp}$. By Corollary 4.8 it suffices to show $\to^{\infty}_{\omega_1,der} \subseteq \to^{\infty}_{fp}$. We prove by well-founded induction on $\alpha \leq \omega_1$ that $\to^{\infty}_{\alpha,der} \subseteq \to^{\infty}_{fp}$. Since $\to^{\infty}_{fp}$ is a fixed point of F, we obtain $\to^{\infty}_{fp} = F(\to^{\infty}_{fp})$, and since $F(\to^{\infty}_{fp})$ is a greatest fixed point, using the ν-rule from (8), it suffices to show that $(*)\ \to^{\infty}_{\alpha,der} \subseteq G(\to^{\infty}_{fp}, \to^{\infty}_{\alpha,der})$. Thus assume that $s \to^{\infty}_{\alpha,der} t$, and let δ be a proof tree of nesting depth $\leq \alpha$ deriving $s \to^{\infty}_{\alpha,der} t$. The only possibility to derive $s \to^{\infty}_{der} t$ is an application of the split-rule with the premise $s\ (\to_{\varepsilon} \cup \overset{<}{\rightharpoondown}{}^{\infty}_{der})^* \circ \rightharpoondown^{\infty}_{der}\ t$. Since $s \to^{\infty}_{\alpha,der} t$, we have $s\ (\to_{\varepsilon} \cup \overset{<}{\rightharpoondown}{}^{\infty}_{\alpha,der})^* \circ \rightharpoondown^{\infty}_{\alpha,der}\ t$. Let τ be one of the steps $\overset{<}{\rightharpoondown}{}^{\infty}_{\alpha,der}$ displayed in the premise. Let u be the source of τ and v the target, so $\tau : u \overset{<}{\rightharpoondown}{}^{\infty}_{\alpha,der} v$. The step τ is derived either via the id-rule or the lift-rule. The case of the id-rule is not interesting since we then can drop τ from the premise. Thus let the step τ be derived using the lift-rule. Then the terms u, v are of form $u = f(u_1, \ldots, u_n)$ and $v = f(v_1, \ldots, v_n)$ and for every $1 \leq i \leq n$ we have $u_i \to^{\infty}_{\beta,der} v_i$ for some $\beta < \alpha$. Thus by induction hypothesis we obtain $u_i \to^{\infty}_{fp} v_i$ for every $1 \leq i \leq n$, and consequently $u\ \overline{\to^{\infty}_{fp}}\ v$. We then have $s\ (\to_{\varepsilon} \cup \overline{\to^{\infty}_{fp}})^* \circ \rightharpoondown^{\infty}_{\alpha,der}\ t$, and hence $s\ G(\to^{\infty}_{fp}, \to^{\infty}_{\alpha,der})\ t$. This concludes the proof. ◀

5 Equivalence with the Standard Definition

In this section we prove the equivalence of the coinductively defined infinitary rewrite relations $\to^{\infty}$ from Definitions 4.3 (and 4.4) with the standard definition based on ordinal length rewrite sequences with metric and strong convergence at every limit ordinal (Definition 2.1). The crucial observation is the following theorem from [29]:

▶ **Theorem 5.1** (Theorem 2 of [29]). *A transfinite reduction is divergent if and only if for some $n \in \mathbb{N}$ there are infinitely many steps at depth n.*

We are now ready to prove the equivalence of both notions:

▶ **Theorem 5.2.** *We have $\to^{\infty} = \to^{\infty}_{ord}$.*

Proof. We write $\rightharpoondown^{\infty}_{ord}$ to denote a reduction $\to^{\infty}_{ord}$ without root steps, and we write $\to^{\alpha}_{ord}$ and $\rightharpoondown^{\alpha}_{ord}$ to indicate the ordinal length α.

We begin with the direction $\to^{\infty}_{ord} \subseteq \to^{\infty}$. We define a function $\mathfrak{T}$ (and $\mathfrak{T}'_{(<)}$) by guarded corecursion [8], mapping rewrite sequences $s \to^{\alpha}_{ord} t$ (and $s \rightharpoondown^{\alpha}_{ord} t$) to infinite proof trees derived using the rules from Definition 4.3. This means that every recursive call produces a constructor, contributing to the construction of the infinite tree. Note that the arguments of $\mathfrak{T}$ (and $\mathfrak{T}'_{(<)}$) are not required to be structurally decreasing.

We do case distinction on the ordinal α. If $\alpha = 0$, then $t = s$ and we define

$$\mathfrak{T}(s \to^{0}_{ord} s) \quad = \quad \dfrac{\mathfrak{T}'(s \rightharpoondown^{0}_{ord} s)}{s \to^{\infty} s}\ \mathsf{split} \qquad\qquad \mathfrak{T}'_{(<)}(s \rightharpoondown^{0}_{ord} s) \quad = \quad \dfrac{}{s \overset{(<)}{\rightharpoondown}{}^{\infty} s}\ \mathsf{id}$$

If $\alpha > 0$, then, by Theorem 5.1 the rewrite sequence $s \to_{ord}^{\alpha} t$ contains only a finite number of root steps. As a consequence, it is of the form:

$$s = s_0 \rightsquigarrow s_1 \cdots \rightsquigarrow s_{n-1} \rightsquigarrow s_n = t$$

where for every $i \in \{0, \dots, n-1\}$, $s_i \rightsquigarrow s_{i+1}$ is either a root step $s_i \to_{\varepsilon} s_{i+1}$, or an infinite reduction below the root $s_i \rightharpoondown_{ord}^{\leq\alpha} s_{i+1}$ where $s_i \rightharpoondown_{ord}^{<\alpha} s_{i+1}$ if $i < n-1$. In the latter case, the length of $s_i \rightharpoondown_{ord} s_{i+1}$ is smaller than α because every strict prefix must be shorter than the sequence itself. We define

$$\mathfrak{T}(s \to_{ord}^{\alpha} t) \quad = \quad \frac{T_0 \quad T_1 \quad \cdots \quad T_{n-1}}{s \to^{\infty} t} \; \textsf{split}$$

where, for $0 \leq i < n$,

$$T_i = \begin{cases} s_i \to_{\varepsilon} s_{i+1} & \text{if } s_i \rightsquigarrow s_{i+1} \text{ is a root step,} \\ \mathfrak{T}'_{<}(s_i \rightharpoondown_{ord}^{\beta} s_{i+1}) & \text{if } i < n-1 \text{ and } s_i \rightharpoondown_{ord}^{\beta} s_{i+1} \text{ for some } \beta < \alpha, \\ \mathfrak{T}'(s_i \rightharpoondown_{ord}^{\beta} s_{i+1}) & \text{if } i = n-1 \text{ and } s_i \rightharpoondown_{ord}^{\beta} s_{i+1} \text{ for some } \beta \leq \alpha. \end{cases}$$

For rewrite sequences $s \rightharpoondown_{ord}^{\alpha} t$ with $\alpha > 0$ we have that $s = f(s_1, \dots, s_n)$ and $t = f(t_1, \dots, t_n)$ for some $f \in \Sigma$ of arity n and terms $s_1, \dots, s_n, t_1, \dots, t_n \in \mathit{Ter}^{\infty}(\Sigma, \mathcal{X})$, and there is a rewrite sequence $s_i \to_{ord}^{\leq\alpha} t_i$ for every i with $1 \leq i \leq n$. We define the two rules:

$$\mathfrak{T}'_{(<)}(s \rightharpoondown_{ord}^{\alpha} t) \quad = \quad \frac{\mathfrak{T}(s_1 \to_{ord}^{\leq\alpha} t_1) \quad \cdots \quad \mathfrak{T}(s_n \to_{ord}^{\leq\alpha} t_n)}{s \stackrel{(<)}{\rightharpoondown}{}^{\infty} t} \; \textsf{lift}$$

The obtained proof tree $\mathfrak{T}(s \to_{ord}^{\alpha} t)$ derives $s \to^{\infty} t$. To see that the requirement that there is no ascending path through this tree containing an infinite number of symbols $\stackrel{<}{\rightharpoondown}{}^{\infty}$ is fulfilled, we note the following. The symbol $\stackrel{<}{\rightharpoondown}{}^{\infty}$ is produced by $\mathfrak{T}'_{<}(s \rightharpoondown_{ord}^{\beta} t)$ which is invoked in $\mathfrak{T}(s \to_{ord}^{\alpha} t)$ for a β that is strictly smaller than α. By well-foundedness of $<$ on ordinals, no such path exists.

We now show $\to^{\infty} \subseteq \to_{ord}^{\infty}$. We prove by well-founded induction on $\alpha \leq \omega_1$ that $\to_{\alpha}^{\infty} \subseteq \to_{ord}^{\infty}$. This suffices since $\to^{\infty} = \to_{\omega_1}^{\infty}$. Let $\alpha \leq \omega_1$ and assume that $s \to_{\alpha}^{\infty} t$. Let δ be a proof tree of nesting depth $< \alpha$ witnessing $s \to_{\alpha}^{\infty} t$. The only possibility to derive $s \to^{\infty} t$ is an application of the split-rule with the premise $s \; (\to_{\varepsilon} \cup \stackrel{<}{\rightharpoondown}{}^{\infty})^{*} \circ \rightharpoondown^{\infty} t$. Since $s \to_{\alpha}^{\infty} t$, we have $s \; (\to_{\varepsilon} \cup \stackrel{<}{\rightharpoondown}{}_{\alpha}^{\infty})^{*} \circ \rightharpoondown_{\alpha}^{\infty} t$. By induction hypothesis we have $s \; (\to_{\varepsilon} \cup \to_{ord}^{\infty})^{*} \circ \rightharpoondown_{\alpha}^{\infty} t$, and thus $s \to_{ord}^{\infty} \circ \rightharpoondown_{\alpha}^{\infty} t$. We have $\rightharpoondown_{\alpha}^{\infty} = \overline{\to_{\alpha}^{\infty}}$, and consequently $s \to_{ord}^{\infty} s_1 \; \overline{\to_{\alpha}^{\infty}} \; t$ for some term s_1. Repeating this argument on $s_1 \; \overline{\to_{\alpha}^{\infty}} \; t$, we get $s \to_{ord}^{\infty} s_1 \; \overline{\to_{ord}^{\infty}} \; s_2 \; \overline{\overline{\to_{\alpha}^{\infty}}} \; t$. After n iterations, we obtain

$$s \to_{ord}^{\infty} s_1 \; \overline{\to_{ord}^{\infty}} \; s_2 \; \overline{\overline{\to_{ord}^{\infty}}} \; s_3 \; \overline{\overline{\overline{\to_{ord}^{\infty}}}} \; s_4 \cdots (\to_{\alpha}^{\infty})^{-(n-1)} \; s_n \; (\to_{\alpha}^{\infty})^{-n} \; t$$

where $(\to_{\alpha}^{\infty})^{-n}$ denotes the nth iteration of $x \mapsto \overline{x}$ on $\to_{\alpha}^{\infty}$.

Clearly, the limit of $\{s_n\}$ is t. Furthermore, each of the reductions $s_n \to_{ord}^{\infty} s_{n+1}$ are strongly convergent and take place at depth greater than or equal to n. Thus, the infinite concatenation of these reductions yields a strongly convergent reduction from s to t (there is only a finite number of rewrite steps at every depth n). ◀

$$\dfrac{\mathsf{a} \to_\varepsilon \mathsf{f}(\mathsf{a}) \qquad \dfrac{\dfrac{\mathsf{a} \to_\varepsilon \mathsf{f}(\mathsf{a}) \quad \dfrac{}{\mathsf{f}(\mathsf{a}) \overset{\infty}{\leftrightarrows} \mathsf{f}^\omega}}{\mathsf{a} \overset{\infty}{=} \mathsf{f}^\omega}}{\mathsf{f}(\mathsf{a}) \overset{\infty}{\leftrightarrows} \mathsf{f}^\omega} \qquad \dfrac{\dfrac{\dfrac{}{\mathsf{f}^\omega \overset{\infty}{\leftrightarrows} \mathsf{f}(\mathsf{b})} \quad \mathsf{f}(\mathsf{b}) \leftarrow_\varepsilon \mathsf{b}}{\mathsf{f}^\omega \overset{\infty}{=} \mathsf{b}}}{\mathsf{f}^\omega \overset{\infty}{\leftrightarrows} \mathsf{f}(\mathsf{b})} \qquad \mathsf{f}(\mathsf{b}) \leftarrow_\varepsilon \mathsf{b}}{\mathsf{a} \overset{\infty}{=} \mathsf{b}}$$

$$\dfrac{\dfrac{\dfrac{\text{(as above)}}{\mathsf{a} \overset{\infty}{=} \mathsf{b}}}{\mathsf{C}(\mathsf{a}) \overset{\infty}{\leftrightarrows} \mathsf{C}(\mathsf{b})} \qquad \mathsf{C}(\mathsf{b}) \to_\varepsilon \mathsf{C}(\mathsf{C}(\mathsf{a})) \qquad \dfrac{\dfrac{}{\mathsf{C}(\mathsf{a}) \overset{\infty}{=} \mathsf{C}^\omega}}{\mathsf{C}(\mathsf{C}(\mathsf{a})) \overset{\infty}{\leftrightarrows} \mathsf{C}^\omega}}{\mathsf{C}(\mathsf{a}) \overset{\infty}{=} \mathsf{C}^\omega}$$

Figure 4 An example of infinitary equational reasoning, deriving $\mathsf{C}(\mathsf{a}) \overset{\infty}{=} \mathsf{C}^\omega$ in the TRS $\mathcal{R}$ of Example 6.2. Recall Notation 4.1.

6 Infinitary Equational Reasoning and Bi-Infinite Rewriting

6.1 Infinitary Equational Reasoning

▶ **Definition 6.1.** Let $\mathcal{R}$ be a TRS over Σ, and let $T = \mathit{Ter}^\infty(\Sigma, \mathcal{X})$. We define *infinitary equational reasoning* as the relation $\overset{\infty}{=} \subseteq T \times T$ by the mutually coinductive rules:

$$\dfrac{s\ (\leftarrow_\varepsilon \cup \to_\varepsilon \cup \overset{\infty}{\leftrightarrows})^*\ t}{s \overset{\infty}{=} t} \qquad\qquad \dfrac{s_1 \overset{\infty}{=} t_1 \quad \cdots \quad s_n \overset{\infty}{=} t_n}{f(s_1, s_2, \ldots, s_n) \overset{\infty}{\leftrightarrows} f(t_1, t_2, \ldots, t_n)}$$

where $\overset{\infty}{\leftrightarrows} \subseteq T \times T$ stands for infinitary equational reasoning below the root.

Note that, in comparison with the rules (1) for $\overset{\infty}{=}$ from the introduction, we now have used $\leftarrow_\varepsilon \cup \to_\varepsilon$ instead of $=_\mathcal{R}$. It is not difficult to see that this gives rise to the same relation. The reason is that we can 'push' non-root rewriting steps $=_\mathcal{R}$ into the arguments of $\overset{\infty}{\leftrightarrows}$.

▶ **Example 6.2.** Let $\mathcal{R}$ be a TRS consisting of the following rules:

$$\mathsf{a} \to \mathsf{f}(\mathsf{a}) \qquad\qquad \mathsf{b} \to \mathsf{f}(\mathsf{b}) \qquad\qquad \mathsf{C}(\mathsf{b}) \to \mathsf{C}(\mathsf{C}(\mathsf{a}))\,.$$

Then we have $\mathsf{a} \overset{\infty}{=} \mathsf{b}$ as derived in Figure 4 (top), and $\mathsf{C}(\mathsf{a}) \overset{\infty}{=} \mathsf{C}^\omega$ as in Figure 4 (bottom).

Definition 6.1 of $\overset{\infty}{=}$ can also be defined using a greatest fixed point as follows:

$$\overset{\infty}{=} \quad := \quad \nu R.\, (\leftarrow_\varepsilon \cup \to_\varepsilon \cup \overline{R})^*\,,$$

where $\overline{R}$ was defined in Definition 4.2. The equivalence of these definitions can be established in a similar way as in Theorem 4.9. It is easy to verify that the function $R \mapsto (\leftarrow_\varepsilon \cup \to_\varepsilon \cup \overline{R})^*$ is monotone, and consequently the greatest fixed point exists.

We note that, in the presence of collapsing rules (i.e., rules $\ell \to r$ where $r \in \mathcal{X}$), everything becomes equivalent: $s \overset{\infty}{=} t$ for all terms s, t. For example, having a rule $\mathsf{f}(x) \to x$ we obtain that $s \overset{\infty}{=} \mathsf{f}(s) \overset{\infty}{=} \mathsf{f}^2(s) \overset{\infty}{=} \cdots \overset{\infty}{=} \mathsf{f}^\omega$ for every term s. This can be overcome by forbidding certain infinite terms and certain infinite limits.

6.2 Bi-Infinite Rewriting

Another notion that arises naturally in our setup is that of bi-infinite rewriting, allowing rewrite sequences to extend infinitely forwards and backwards. We emphasize that each of the steps $\to_\varepsilon$ in such sequences is a forward step.

▶ **Definition 6.3.** Let $\mathcal{R}$ be a term rewriting system over Σ, and let $T = \mathit{Ter}^\infty(\Sigma, \mathcal{X})$. We define the *bi-infinite rewrite relation* $\overset{\infty}{\rightrightarrows} \subseteq T \times T$ by the following coinductive rules

$$\frac{s\ (\to_\varepsilon \cup \overset{\infty}{\rightharpoonup})^*\ t}{s \overset{\infty}{\rightrightarrows} t} \qquad\qquad \frac{s_1 \overset{\infty}{\rightrightarrows} t_1 \quad \cdots \quad s_n \overset{\infty}{\rightrightarrows} t_n}{f(s_1, s_2, \ldots, s_n) \overset{\infty}{\rightharpoonup} f(t_1, t_2, \ldots, t_n)}$$

where $\overset{\infty}{\rightharpoonup} \subseteq T \times T$ stands for bi-infinite rewriting below the root.

If we replace $\overset{\infty}{\equiv}$ and $\to^\infty$ by $\overset{\infty}{\rightrightarrows}$, and $\overset{\infty}{\leftrightharpoons}$ and $\rightharpoonup^\infty$ by $\overset{\infty}{\rightharpoonup}$, then Examples 1.1 and 1.2 are illustrations of this rewrite relation.

Again, like $\overset{\infty}{\equiv}$, the relation $\overset{\infty}{\rightrightarrows}$ can also be defined using a greatest fixed point:

$$\overset{\infty}{\rightrightarrows} \quad := \quad \nu R.\, (\to_\varepsilon \cup \overline{R})^* \,.$$

Monotonicity of $R \mapsto (\to_\varepsilon \cup \overline{R})^*$ is easily verified, so that the greatest fixed point exists. Also, the equivalence of Definition 6.3 with this ν-definition can be established similarly.

7 Relating the Notions

▶ **Lemma 7.1.** *Each of the relations $\to^\infty$, $\overset{\infty}{\rightrightarrows}$ and $\overset{\infty}{\equiv}$ is reflexive and transitive. The relation $\overset{\infty}{\equiv}$ is also symmetric.*

Proof. Follows immediately from the fact that the relations are defined using the reflexive-transitive closure in each of their first rules. ◀

▶ **Theorem 7.2.** *For every TRS $\mathcal{R}$ we have the following inclusions:*

$$\begin{array}{ccccccc} \to^\infty & \subseteq & \overset{\infty}{\rightrightarrows} & \subseteq & (\overset{\infty}{\leftleftarrows} \cup \overset{\infty}{\rightrightarrows})^* & \subseteq & \overset{\infty}{\equiv} \\ & \subsetneq & & \subsetneq & & & \\ & & ({}^\infty\!\leftarrow \cup \to^\infty)^* & & & & \end{array}$$

Moreover, for each of these inclusions there exists a TRS for which the inclusion is strict.

Proof. The inclusions $\to^\infty \subsetneq \overset{\infty}{\rightrightarrows} \subsetneq \overset{\infty}{\equiv}$ have already been established in the introduction. The inclusion $\to^\infty \subsetneq ({}^\infty\!\leftarrow \cup \to^\infty)^*$ is well-known (and obvious). Also $\overset{\infty}{\rightrightarrows} \subsetneq (\overset{\infty}{\leftleftarrows} \cup \overset{\infty}{\rightrightarrows})^*$ is immediate since $\overset{\infty}{\rightrightarrows}$ is not symmetric.

The inclusion $({}^\infty\!\leftarrow \cup \to^\infty)^* \subseteq (\overset{\infty}{\leftleftarrows} \cup \overset{\infty}{\rightrightarrows})^*$ is immediate since $\to^\infty \subseteq \overset{\infty}{\rightrightarrows}$. Example 1.1 witnesses strictness of this inclusion. The reason is that, for this example, $\to^\infty = \to^*$ as the system does not admit any forward limits. Hence $({}^\infty\!\leftarrow \cup \to^\infty)^*$ is just finite conversion on potentially infinite terms. Thus $\mathsf{C}^\omega \overset{\infty}{\rightrightarrows} \mathsf{a}$, but not $\mathsf{C}^\omega\ ({}^\infty\!\leftarrow \cup \to^\infty)^*\ \mathsf{a}$.

The inclusion $(\overset{\infty}{\leftleftarrows} \cup \overset{\infty}{\rightrightarrows})^* \subseteq \overset{\infty}{\equiv}$ follows from the fact that $\overset{\infty}{\equiv}$ includes $\overset{\infty}{\rightrightarrows}$ and is symmetric and transitive. Example 6.2 witnesses strictness: $\mathsf{C}(\mathsf{a}) = \mathsf{C}^\omega$ can only be derived by a rewrite sequence of the form:

$$\mathsf{C}(\mathsf{a}) \overset{\infty}{\rightrightarrows} \mathsf{C}(\mathsf{f}^\omega) \overset{\infty}{\leftleftarrows} \mathsf{C}(\mathsf{b}) \to \mathsf{C}(\mathsf{C}(\mathsf{a})) \overset{\infty}{\rightrightarrows} \mathsf{C}(\mathsf{C}(\mathsf{f}^\omega)) \overset{\infty}{\leftleftarrows} \mathsf{C}(\mathsf{C}(\mathsf{b})) \to \mathsf{C}(\mathsf{C}(\mathsf{C}(\mathsf{a}))) \overset{\infty}{\rightrightarrows} \cdots$$

and hence we need to change rewriting directions infinitely often whereas $(\overset{\infty}{\leftleftarrows} \cup \overset{\infty}{\rightrightarrows})^*$ allows to change the direction only a finite number of times. ◀

Concerning, the rewrite relations introduced in [23] it is not difficult to see that $\overset{\infty}{\rightrightarrows} \subsetneq \twoheadrightarrow_t$ where $\twoheadrightarrow_t$ is the topological graph closure of $\to$.

8 A Formalization in Coq

The standard definition of infinitary rewriting, using ordinal length rewrite sequences and strong convergence at limit ordinals, is difficult to formalize. The coinductive framework we propose, is easy to formalize and work with in theorem provers.

In Coq, the coinductive definition of infinitary strongly convergent reductions can be defined as follows:

```
Inductive ired : relation term :=
  | Ired :
      forall R I : relation term,
      subrel I ired ->
      subrel R ((root_step (+) lift I)* ;; lift R) ->
      subrel R ired.
```

Here `term` is the set of coinductively defined terms, `;;` is relation composition, `(+)` is the union of relations, `*` the reflexive-transitive closure, `lift R` is $\overline{R}$, and `root_step` is the root step relation.

Let us briefly comment on this formalization. Recall that $\to^{\infty} \;:=\; \mu R.\, \nu S.\, G(R, S)$ where $G(R, S) = (\to_{\varepsilon} \cup \overline{R})^{*} \circ \overline{S}$. The inductive definition of `ired` corresponds to the least fixed point μR. Coq has no support for mutual inductive and coinductive definitions. Therefore, instead of the explicit coinduction, we use the ν-rule from (8). For every relation T that fulfills $T \subseteq G(R, T)$, we have that $T \subseteq \nu S.\, G(R, S)$. Moreover, we know that $\nu S.\, G(R, S)$ is the union of all these relations T. Finally, we introduce an auxiliary relation `I` to help Coq generate a good induction principle. One can think of `I` as consisting of those pairs for which the recursive call to `ired` is invoked. Replacing `lift I` by `lift ired` is correct, but then the induction principle that Coq generates for `ired` is useless.

On the basis of the above definition we proved the Compression Lemma: whenever there is an infinite reduction from s to t ($s \to^{\infty} t$) then there exists a reduction of length at most ω from s to t ($s \to^{\leq \omega} t$). The Compression Lemma holds for left-linear TRSs with finite left-hand sides. To characterize rewrite sequences $\to^{\leq \omega}$ in Coq, we define:

```
Inductive ored : relation (term F X) :=
| Ored :
    forall R : relation (term F X),
    subrel R (mred ;; lift R) ->
    forall s t, R s t -> ored s t.
```

Here `mred` are finite rewrite sequences $\to^{*}$. The definition can be understood as follows. We want the relation `ored` to be the greatest fixed point of H defined by $H(R) = \to^{*} \circ \overline{R}$. So we allow a finite rewrite sequence after which the rewrite activity has to go 'down' to the arguments. Again, as above for `ired`, we avoid the use of coinduction and define `ored` inductively as the union of all relations R with $R \subseteq H(R)$.

To the best of our knowledge this is the first formal proof of this well-known lemma. The formalization is available at `http://dimitrihendriks.com/coq/compression`.

9 Conclusion

We have proposed a coinductive framework which gives rise to several natural variants of infinitary rewriting in a uniform way:

(a) infinitary equational reasoning $\overset{\infty}{=} \;:=\; \nu y.\, (\leftarrow_{\varepsilon} \cup \to_{\varepsilon} \cup \overline{y})^{*}$,

(b) bi-infinite rewriting $\overset{\infty}{\to} \;:=\; \nu y.\, (\to_{\varepsilon} \cup \overline{y})^{*}$, and

(c) infinitary rewriting $\to^{\infty} \;:=\; \mu x.\, \nu y.\, (\to_{\varepsilon} \cup \overline{x})^{*} \circ \overline{y}\,.$

We believe that (a) and (b) are new. As a consequence of the coinduction over the term structure, these notions have the strong convergence built-in, and thus can profit from the well-developed techniques (such as tracing) in infinitary rewriting.

We have given a mixed inductive/coinductive definition of infinitary rewriting and established a bridge between infinitary rewriting and coalgebra. Both fields are concerned with infinite objects and we would like to understand their relation better. In contrast to previous coinductive treatments, the framework presented here captures rewrite sequences of arbitrary ordinal length, and paves the way for formalizing infinitary rewriting in theorem provers (as illustrated by our proof of the Compression Lemma in Coq).

Concerning proof trees/terms for infinite reductions, let us mention that an alternative approach has been developed in parallel by Lombardi, Ríos and de Vrijer [30]. While we focus on proof terms for the reduction relation and abstract from the order of steps in parallel subterms, they use proof terms for modeling the fine-structure of the infinite reductions themselves. Another difference is that our framework allows for non-left-linear systems. We believe that both approaches are complementary. Theorems for which the fine-structure of rewrite sequences is crucial, must be handled using [30]. (But note that we can capture standard reductions by a restriction on proof trees and prove standardization using proof tree transformations, see [19]). If the fine-structure is not important, as for instance for proving confluence, then our system is more convenient to work with due to simpler proof terms.

Our work lays the foundation for several directions of future research:

(i) The coinductive treatment of infinitary λ-calculus [19] has led to elegant, significantly simpler proofs [9, 10] of some central properties of the infinitary λ-calculus. The coinductive framework that we propose enables similar developments for infinitary term rewriting with reductions of arbitrary ordinal length.

(ii) The concepts of bi-infinite rewriting and infinitary equational reasoning are novel. We would like to study these concepts, in particular since the theory of infinitary equational reasoning is still underdeveloped. For example, it would be interesting to compare the Church–Rosser properties

$$\overset{\infty}{=} \;\subseteq\; \to^{\infty} \circ {}^{\infty}\!\leftarrow \qquad \text{and} \qquad ({}^{\infty}\!\leftarrow \circ \to^{\infty})^{*} \;\subseteq\; \to^{\infty} \circ {}^{\infty}\!\leftarrow\,.$$

(iii) The formalization of the proof of the Compression Lemma in Coq is just the first step towards the formalization of all major theorems in infinitary rewriting.

(iv) It is interesting to investigate whether and how the coinductive framework can be extended to other notions of infinitary rewriting, for example reductions where root-active terms are mapped to $\perp$ in the limit [3, 2, 4, 18].

(v) We believe that the coinductive definitions will ease the development of new techniques for automated reasoning about infinitary rewriting. For example, methods for proving (local) productivity [13, 15, 35], for (local) infinitary normalization [34, 14, 12], for (local) unique normal forms [17], and for analysis of infinitary reachability and infinitary confluence. Due to the coinductive definitions, the implementation and formalization of these techniques could make use of circular coinduction [20, 16].

Acknowledgments. We thank Patrick Bahr, Jeroen Ketema, and Vincent van Oostrom for fruitful discussions and comments on earlier versions of this paper.

References

1. F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge Univ. Press, 1998.
2. P. Bahr. Abstract Models of Transfinite Reductions. In *Proc. Conf. on Rewriting Techniques and Applications (RTA 2010)*, volume 6 of *Leibniz International Proceedings in Informatics*, pages 49–66. Schloss Dagstuhl, 2010.
3. P. Bahr. Partial Order Infinitary Term Rewriting and Böhm Trees. In *Proc. Conf. on Rewriting Techniques and Applications (RTA 2010)*, volume 6 of *Leibniz International Proceedings in Informatics*, pages 67–84. Schloss Dagstuhl, 2010.
4. P. Bahr. Infinitary Term Graph Rewriting is Simple, Sound and Complete. In *Proc. Conf. on Rewriting Techniques and Applications (RTA 2012)*, volume 15 of *Leibniz International Proceedings in Informatics*, pages 69–84. Schloss Dagstuhl, 2012.
5. H.P. Barendregt. The Type Free Lambda Calculus. In *Handbook of Mathematical Logic*, pages 1091–1132. Nort-Holland Publishing Company, Amsterdam, 1977.
6. H.P. Barendregt and J.W. Klop. Applications of Infinitary Lambda Calculus. *Information and Computation*, 207(5):559–582, 2009.
7. C. Coquand and Th. Coquand. On the Definition of Reduction for Infinite Terms. *Comptes Rendus de l'Académie des Sciences. Série I*, 323(5):553–558, 1996.
8. Th. Coquand. Infinite objects in type theory. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs, International Workshop TYPES'93, Nijmegen, The Netherlands, May 24–28, 1993, Selected Papers*, volume 806 of *LNCS*, pages 62–78. Springer, 1994.
9. Ł. Czajka. A Coinductive Confluence Proof for Infinitary Lambda-Calculus. In *Rewriting and Typed Lambda Calculi (RTA-TLCA 2014)*, volume 8560 of *Lecture Notes in Computer Science*, pages 164–178. Springer, 2014.
10. Ł. Czajka. Coinductive Techniques in Infinitary Lambda-Calculus. *ArXiv e-prints*, 2015.
11. N. Dershowitz, S. Kaplan, and D.A. Plaisted. Rewrite, Rewrite, Rewrite, Rewrite, Rewrite,.... *Theoretical Computer Science*, 83(1):71–96, 1991.
12. J. Endrullis, R. C. de Vrijer, and J. Waldmann. Local Termination: Theory and Practice. *Logical Methods in Computer Science*, 6(3), 2010.
13. J. Endrullis, C. Grabmayer, and D. Hendriks. Complexity of Fractran and Productivity. In *Proc. Conf. on Automated Deduction (CADE 22)*, volume 5663 of *LNCS*, pages 371–387, 2009.
14. J. Endrullis, C. Grabmayer, D. Hendriks, J.W. Klop, and R.C de Vrijer. Proving Infinitary Normalization. In *Postproc. Int. Workshop on Types for Proofs and Programs (TYPES 2008)*, volume 5497 of *LNCS*, pages 64–82. Springer, 2009.
15. J. Endrullis and D. Hendriks. Lazy Productivity via Termination. *Theoretical Computer Science*, 412(28):3203–3225, 2011.
16. J. Endrullis, D. Hendriks, and M. Bodin. Circular Coinduction in Coq Using Bisimulation-Up-To Techniques. In *Proc. Conf. on Interactive Theorem Proving (ITP)*, volume 7998 of *LNCS*, pages 354–369. Springer, 2013.
17. J. Endrullis, D. Hendriks, C. Grabmayer, J.W. Klop, and V. van Oostrom. Infinitary term rewriting for weakly orthogonal systems: Properties and counterexamples. *Logical Methods in Computer Science*, 10(2:7):1–33, 2014.
18. J. Endrullis, D. Hendriks, and J.W. Klop. Highlights in Infinitary Rewriting and Lambda Calculus. *Theoretical Computer Science*, 464:48–71, 2012.
19. J. Endrullis and A. Polonsky. Infinitary Rewriting Coinductively. In *Proc. Types for Proofs and Programs (TYPES 2012)*, volume 19 of *Leibniz International Proceedings in Informatics*, pages 16–27. Schloss Dagstuhl, 2013.
20. J. Goguen, K. Lin, and G. Roşu. Circular Coinductive Rewriting. In *Proc. of Automated Software Engineering*, pages 123–131. IEEE, 2000.

21 B. Jacobs and J.J.M.M. Rutten. An Introduction to (Co)Algebras and (Co)Induction. In *Advanced Topics in Bisimulation and Coinduction*, pages 38–99. Cambridge University Press, 2011.

22 F. Joachimski. Confluence of the Coinductive Lambda Calculus. *Theoretical Computer Science*, 311(1-3):105–119, 2004.

23 S. Kahrs. Infinitary Rewriting: Foundations Revisited. In *Proc. Conf. on Rewriting Techniques and Applications (RTA 2010)*, volume 6 of *Leibniz International Proceedings in Informatics*, pages 161–176. Schloss Dagstuhl, 2010.

24 S. Kahrs. Infinitary Rewriting: Closure Operators, Equivalences and Models. *Acta Informatica*, 50(2):123–156, 2013.

25 J.R. Kennaway and F.-J. de Vries. *Infinitary Rewriting*, chapter 12. Cambridge University Press, 2003. in [32].

26 J.R. Kennaway, J.W. Klop, M.R. Sleep, and F.-J. de Vries. Transfinite Reductions in Orthogonal Term Rewriting Systems. *Information and Computation*, 119(1):18–38, 1995.

27 J. Ketema and J.G. Simonsen. Computing with Infinite Terms and Infinite Reductions. Unpublished manuscript.

28 J.W. Klop. Term Rewriting Systems. In *Handbook of Logic in Computer Science*, volume II, pages 1–116. Oxford University Press, 1992.

29 J.W. Klop and R.C de Vrijer. Infinitary Normalization. In *We Will Show Them: Essays in Honour of Dov Gabbay (2)*, pages 169–192. College Publications, 2005.

30 C. Lombardi, A. Ríos, and R.C de Vrijer. Proof Terms for Infinitary Rewriting. In *Rewriting and Typed Lambda Calculi (RTA-TLCA 2014)*, volume 8560 of *Lecture Notes in Computer Science*, pages 303–318. Springer, 2014.

31 J.G. Simonsen. On Confluence and Residuals in Cauchy Convergent Transfinite Rewriting. *Information Processing Letters*, 91(3):141–146, 2004.

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

33 M. Vermaat. Infinitary Rewriting in Coq. Available at url `http://martijn.vermaat.name/master-project/`.

34 H. Zantema. Normalization of Infinite Terms. In *Proc. Conf. on Rewriting Techniques and Applications (RTA 2008)*, number 5117 in LNCS, pages 441–455, 2008.

35 H. Zantema and M. Raffelsieper. Proving Productivity in Infinite Data Structures. In *Proc. Conf. on Rewriting Techniques and Applications (RTA 2010)*, volume 6 of *Leibniz International Proceedings in Informatics*, pages 401–416. Schloss Dagstuhl, 2010.

Proving non-termination by finite automata

Jörg Endrullis[1] and Hans Zantema[2,3]

1 **Department of Computer Science, VU University Amsterdam, 1081 HV Amsterdam, The Netherlands, email:** j.endrullis@vu.nl
2 **Department of Computer Science, TU Eindhoven, P.O. Box 513, 5600 MB Eindhoven, The Netherlands, email:** H.Zantema@tue.nl
3 **Institute for Computing and Information Sciences, Radboud University Nijmegen, P.O. Box 9010, 6500 GL Nijmegen, The Netherlands**

Abstract

A new technique is presented to prove non-termination of term rewriting. The basic idea is to find a non-empty regular language of terms that is closed under rewriting and does not contain normal forms. It is automated by representing the language by a tree automaton with a fixed number of states, and expressing the mentioned requirements in a SAT formula. Satisfiability of this formula implies non-termination. Our approach succeeds for many examples where all earlier techniques fail, for instance for the S-rule from combinatory logic.

1998 ACM Subject Classification D.1.1 Applicative (Functional) Programming, D.3.1 Formal Definitions and Theory, F.4.1 Mathematical Logic, F.4.2 Grammars and Other Rewriting Systems, I.1.1 Expressions and Their Representation, I.1.3 Languages and Systems

Keywords and phrases Non-termination, finite automata, regular languages

Digital Object Identifier 10.4230/LIPIcs.RTA.2015.160

1 Introduction

A basic approach for proving that a term rewriting system (TRS) is non-terminating is to prove that it admits a *loop*, that is, a reduction of the shape $t \to^+ C[t\sigma]$, see [26]. Indeed, such a loop gives rise to an infinite reduction $t \to^+ C[t\sigma] \to^+ C[(C[t\sigma])\sigma] \to \cdots$ in which in every step t is replaced by $C[t\sigma]$. In trying to prove non-termination, several tools ([1, 2]) search for a loop. An extension from [7], implemented in [1] goes a step further: it searches for reductions of the shape $t\sigma^n\mu \to^+ C[t\sigma^{f(n)}\mu\tau]$ for every n for a linear increasing function f, and some extensions. All of these patterns are chosen to be extended to an infinite reduction in an obvious way, hence proving non-termination. However, many non-terminating TRSs exist not admitting an infinite reduction of this regular shape, or the technique from [7] fails to find it.

A crucial example is the S-rule $a(a(a(S,x),y),z) \to a(a(x,z),a(y,z))$, one of the building blocks of Combinatory Logic. Although being only one single rule, and having nice properties like orthogonality, non-termination of this system is a hard issue. Infinite reductions are known, but are of a complicated shape, see [31]. So developing a general technique that can prove non-termination of the S-rule automatically is a great challenge. In this paper we succeed in presenting such a technique, and we describe a SAT-based approach by which non-termination of many TRSs, including the S-rule, is proved fully automatically.

The underlying idea is quite simple: non-termination immediately follows from the existence of a non-empty set of terms that is closed under rewriting and does not contain normal forms. Our approach is to find such a set being the language accepted by a finite tree automaton, and find this tree automaton from the satisfying assignment of a SAT formula

© Jörg Endrullis and Hans Zantema;
licensed under Creative Commons License CC-BY
26th International Conference on Rewriting Techniques and Applications (RTA'15).
Editor: Maribel Fernández; pp. 160–176
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

describing the above requirements. Hence the goal is to describe the requirements, namely non-emptiness, closed under rewriting, and not containing normal forms, in a SAT formula.

We want to stress that having quick methods for proving non-termination of term rewriting also may be fruitful for proving termination. In a typical search for a termination proof, like using the dependency pair framework, the original problem is transformed in several ways to other termination problems that not all need to be terminating. Being able to quickly recognize non-termination of some of them makes a further search for termination proofs redundant, which may speed up the overall search for a termination proof.

We note that, like termination, non-termination is an undecidable property. However, while termination is Π^0_2-complete, non-termination is Σ^0_2-complete [11, 10].

The paper is organized as follows. In Section 2 we present our basic approach in the setting of abstract reduction systems on a set T, in which the language is just a subset of T. Surprisingly, being not weakly normalizing corresponds to (strongly) closed under rewriting, and being not strongly normalizing corresponds to weakly closed under rewriting. In Section 3 we give preliminaries on tree automata and show how string automata can be seen as an instance of tree automata. In Section 4 we present our basic methods, starting by how the requirements are expressed in SAT, and next how this is used to disprove weak normalization and strong normalization. In Section 5 we strengthen our approach by labeling the states of the tree automata by sets of rewrite rules and exploiting this in the method. In Section 6 we present experimental results of our implementation. We conclude in Section 7.

1.1 Related Work

The paper [26] introduces the notion of loops and investigates necessary conditions for the existence of them. The work [34] employs SAT solvers to find loops, [35] uses forward closures to find loops efficiently, and [33] introduces 'compressed loops' to find certain forms of very long loops. Non-termination beyond loops has been investigated in [29] and [7]. There the basic idea is the search for a particular generalization of loops, like a term t and substitutions σ, τ such that for every n there exist C, μ such that $t\sigma^n\tau$ rewrites to $C[t\sigma^{f(n)}\tau\mu]$, for some ascending linear function f. Although the S-rule admits such reductions, these techniques fail to find them. For other examples for which not even reductions exist of the shape studied in [29] and [7], we will be able to prove non-termination fully automatically.

Our approach can be summarized as searching for non-termination proofs based on regular (tree) automata. Regular (tree) automata have been fruitfully applied to a wide rage of properties of term rewriting systems: for proving termination [25, 21, 27], infinitary normalization [12], liveness [28], and for analyzing reachability and deciding the existence of common reducts [23, 13]. Local termination on regular languages, has been investigated in [9].

2 Abstract Rewriting

An *abstract reduction system* (ARS) is a binary relation $\to$ on a set T. We write $\to^+$ for the transitive closure, and $\to^*$ for the reflexive, transitive closure of $\to$.

Let $\to$ be an ARS on T. The ARS $\to$ is called *terminating* or *strongly normalizing* (SN) if no infinite sequence $t_0, t_1, t_2, \ldots \in T$ exists such that $t_i \to t_{i+1}$ for all $i \geq 0$. A *normal form* with respect to $\to$ is an element $t \in T$ such that no $u \in T$ exists satisfying $t \to u$. The set of all normal forms with respect to $\to$ is denoted by $\mathsf{NF}(\to)$. The ARS $\to$ is called *weakly normalizing* (WN) if for every $t \in T$ a normal form $u \in T$ exists such that $t \to^* u$.

▶ **Definition 1.** A set $L \subseteq T$ is called

- *closed under* $\to$ if for all $t \in L$ and all $u \in T$ satisfying $t \to u$ it holds $u \in L$, and
- *weakly closed under* $\to$ if for all $t \in L \setminus \mathsf{NF}(\to)$ there exists $u \in L$ such that $t \to u$.

It is straightforward from these definitions that if L is closed under $\to$, then L is weakly closed under $\to$ as well. The following theorems relate these notions to SN and WN.

▶ **Theorem 2.** *An ARS $\to$ on T is not* SN *if and only if a non-empty $L \subseteq T$ exists such that $L \cap \mathsf{NF}(\to) = \varnothing$ and L is weakly closed under $\to^+$.*

Proof. If $\to$ is not SN then an infinite sequence $t_0, t_1, t_2, \ldots \in T$ exists such that $t_i \to t_{i+1}$ for all $i \geq 0$. Then $L = \{t_i \mid i \geq 0\}$ satisfies the required properties.

Conversely, assume L satisfies the given properties. Since L is non-empty we can choose $t_0 \in L$, and using the other properties for $i = 0, 1, 2, 3, \ldots$ we can choose $t_{i+1} \in L$ such that $t_i \to^+ t_{i+1}$, proving that $\to$ is not SN. ◀

▶ **Theorem 3.** *An ARS $\to$ on T is not* WN *if and only if a non-empty $L \subseteq T$ exists such that $L \cap \mathsf{NF}(\to) = \varnothing$ and L is closed under $\to$.*

Proof. If $\to$ is not WN then $t \in T$ exists such that $L \cap \mathsf{NF}(\to) = \varnothing$ for $L = \{u \in T \mid t \to^* u\}$. Then L satisfies the required properties.

Conversely, assume L satisfies the given properties. Since L is non-empty we can choose $t_0 \in L$. Assume that $\to$ is WN, then $t_0 \to t_1 \to \cdots \to t_n$ exists such that $t_n \in \mathsf{NF}(\to)$. Since L is closed under $\to$ we obtain $t_i \in L$ for $i = 1, 2, \ldots, n$, contradicting $L \cap \mathsf{NF}(\to) = \varnothing$. ◀

A variant of Theorem 2, where $\to^+$ is replaced by $\to$, has been observed in [5]. To the best knowledge of the authors, Theorem 3 has not been observed in the literature.

3 Tree Automata

▶ **Definition 4.** A *(non-deterministic finite) tree automaton A over a signature* Σ is a tuple $A = \langle Q, \Sigma, F, \delta \rangle$ where

(i) Q is a finite set of *states*,
(ii) $F \subseteq Q$ is a set of *accepting states*, and
(iii) δ a set of rewrite rules, called *transition rules*, of the shape

$$f(q_1, \ldots, q_n) \rightsquigarrow q$$

where n is the arity of $f \in \Sigma$ and $q_1, \ldots, q_n, q \in Q$. We write $\rightsquigarrow$ for the rewrite relation generated by the rules δ.

Note that we use $\rightsquigarrow$ to distinguish automata transitions from term rewriting $\to$ with respect to some TRS R.

▶ **Definition 5.** The *language* $\mathcal{L}(A)$ *accepted by* A is the set

$$\mathcal{L}(A) = \{\, t \mid t \in T(\Sigma, \varnothing),\ q \in F,\ t \rightsquigarrow^* q \,\}$$

of ground terms that rewrite to a final state.

The kind of tree automata considered here is called *bottom up* in the literature. Sometimes in the definition of bottom-up tree automaton the right hand side q in the rule has arguments and the acceptance criterion is rewriting to a term with a final state as root. However, when

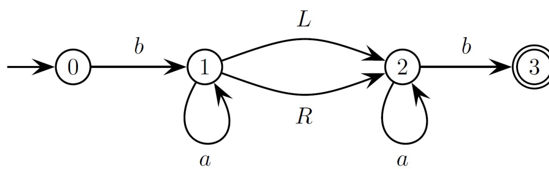
tree automata are only used for defining (term) languages as is the case in this paper, these definitions coincide.

Tree automata can be seen as a generalization of string automata as follows. For a string automaton (= NFA) S define the tree automaton A by

- taking the same sets of states and accepting states, and
- taking as signature the same signature in which all symbols are unary, extended by a single constant ε, and
- taking as transition rules $\varepsilon \rightsquigarrow q_0$ for q_0 being the initial state of S, and for every transition $q \xrightarrow{a} q'$ in S the rule $a(q) \rightsquigarrow q'$.

Form this definition it is immediate that a string $a_1 a_2 \cdots a_n$ is accepted by S if and only if $a_n(a_{n-1}(\cdots(a_1(\varepsilon))\cdots))$ is accepted by A. Here we assume that S reads the string from left to right (otherwise there is no need to reverse the order of the letters).

▶ **Example 6.** To define a tree automaton accepting the language $b\ a^*\ (L|R)\ a^*\ b$, that is, all words that start with b, end with b, contain one L or R and otherwise only a, we start by its corresponding string automaton



The above construction yields the tree automaton $A_{LR} = \langle Q, \Sigma, F, \delta \rangle$ where $\Sigma = \{b, L, R, a, \varepsilon\}$ in which b, L, R, a are unary and ε is a constant, $Q = \{0, 1, 2, 3\}$, $F = \{3\}$ and δ consists of the rules

$$\varepsilon \rightsquigarrow 0 \qquad a(1) \rightsquigarrow 1 \qquad b(0) \rightsquigarrow 1 \qquad R(1) \rightsquigarrow 2 \qquad L(1) \rightsquigarrow 2$$
$$a(2) \rightsquigarrow 2 \qquad b(2) \rightsquigarrow 3$$

▶ **Example 7.** The following is a tree automaton for the signature $\Sigma = \{a, S\}$ where a is binary and S is a constant. Let $A_S = \langle Q, \Sigma, F, \delta \rangle$ where $Q = \{0, 1, 2, 3, 4\}$, $F = \{4\}$ and

$$S \rightsquigarrow 0 \qquad a(0,0) \rightsquigarrow 1 \qquad a(1,0) \rightsquigarrow 2 \qquad a(2,2) \rightsquigarrow 3 \qquad a(3,3) \rightsquigarrow 3$$
$$a(0,2) \rightsquigarrow 2 \qquad a(2,3) \rightsquigarrow 3 \qquad a(3,3) \rightsquigarrow 4$$
$$a(0,3) \rightsquigarrow 2$$

As is usual in combinatory logic, ground terms are represented by omitting the a symbol and writing $uvw = (uv)w$. We show that this automaton accepts the term $SSS(SSS)(SSS(SSS))$:

$$\begin{aligned} SSS(SSS)(SSS(SSS)) &\rightsquigarrow^{12} 000(000)(000(000)) \\ &\rightsquigarrow^{4} 10(10)(10(10)) \rightsquigarrow^{4} 22(22) \rightsquigarrow^{2} 33 \rightsquigarrow^{1} 4 \end{aligned}$$

Since $4 \in F$ the term is accepted by the automaton.

This automaton has been found automatically by our tool, and its language is closely related to the QQQ-criterion of Waldmann [31, 3]. Roughly speaking, the language recognized by this automaton can be described as follows:

- state 0 accepts only the term S,
- state 1 accepts only the term SS,
- state 2 corresponds to terms that contain at least *one* occurrence of SSS,
- state 3 corresponds to terms that contain at least *two* occurrence of SSS, and
- state 4 accepts terms MN for which both M and N contain two occurrences of SSS.

4 Basic Methods

In this section, we are concerned with automating the abstract non-termination methods from Section 2. To this end, we use finite tree automata giving rise to regular tree languages. We first develop methods for disproving weak normalization and then for disproving strong normalization.

4.1 SAT Encoding of Properties

In this section, we collect decision procedures for the main properties of tree automata that we employ for proving non-termination, and we describe how we encode these procedures as Boolean satisfiability problems (SAT).

▶ **Remark 8** (SAT encoding of tree automata)**.** We encode the search for a tree automaton $A = \langle Q, \Sigma, F, \delta \rangle$ over a signature Σ as a satisfiability problem as follows. We pick the number of states $n \in \mathbb{N}$ the automaton should have; the set of states is $Q = \{s_1, \ldots, s_n\}$. While the set of states Q is fix, we represent the final states $F \subseteq Q$ by n fresh Boolean variables

$$v_{F,s_1},\ v_{F,s_2},\ v_{F,s_3}, \ldots, v_{F,s_n}$$

and, for every $f \in \Sigma$, we represent the transition relation δ by fresh variables

$$v_{f,q_1,\ldots,q_{\#(f)},q} \qquad \text{for every } q_1, \ldots, q_{\#(f)}, q \in Q$$

For the moment, there are no constraints (formulas) and the interpretation of these variables can be chosen freely. The intention is that v_{F,s_i} is true if and only if s_i is a final state, and $v_{f,q_1,\ldots,q_{\#(f)},q}$ is true if and only if $f(q_1, \ldots, q_{\#(f)}) \rightsquigarrow q$ is a transition rule in δ.

▶ **Definition 9.** A state $q \in Q$ of a tree automaton $A = \langle Q, \Sigma, F, \delta \rangle$ is called *reachable* if there exists a ground term $t \in T(\Sigma, \varnothing)$ such that $t \rightsquigarrow^* q$.

We assume, without loss of generality, that all states are reachable. Note that requiring that all states are reachable is not a restriction since we can always replace unreachable states by 'copies' of reachable states. We guarantee reachability as follows.

▶ **Lemma 10.** *Let $A = \langle Q, \Sigma, F, \delta \rangle$ be a tree automaton. Then all states of A are reachable if and only if there exists a total well-founded order $<$ on the states Q such that for every $q \in Q$ there exists $f \in \Sigma$ and states $q_1 < q$, $q_2 < q$, $\ldots$, $q_{\#(f)} < q$ with $f(q_1, \ldots, q_{\#(f)}) \rightsquigarrow q$.*

Proof. The reachable states are the smallest set $Q' \subseteq Q$ that is closed under δ, that is, $q \in Q'$ whenever $f(q_1, \ldots, q_{\#(f)}) \rightsquigarrow q$ for some $f \in \Sigma$ and states $q_1, \ldots, q_{\#(f)} \in Q'$.

For the 'if'-part, assume that there was a non-reachable state. Let $q \in Q$ be the smallest non-reachable state with respect to the order $<$. By assumption there exist $f \in \Sigma$ and states $q_1 < q$, $q_2 < q$, $\ldots$, $q_{\#(f)} < q$ with $f(q_1, \ldots, q_{\#(f)}) \rightsquigarrow q$. By choice of q it follows that all states $q_1, q_2, \ldots, q_{\#(f)}$ are reachable, and hence q is reachable, contradicting the assumption.

For the 'only if'-part, assume that all states are reachable. Then Q is the result of stepwise closing $\varnothing$ under δ. There exists a sequence of states $\varnothing = Q_0 \subseteq Q_1 \subseteq \ldots \subseteq Q_{|Q|} = Q$ such that for every $0 \leq i < |Q|$ we have $Q_{i+1} = Q_i \cup \{q_i\}$ for some $q_i \in Q \setminus Q_i$ such that there are $f_i \in \Sigma$ and states $q_{i,1}, \ldots, q_{i,\#(f_i)} \in Q_i$ with $f_i(q_{i,1}, \ldots, q_{i,\#(f)}) \rightsquigarrow q_i$. The order $<$ induced by $q_0 < q_1 < \ldots < q_{|Q|-1}$ is a total order on the states with the desired property. ◀

▶ **Remark 11** (SAT encoding of reachability of all states)**.** We extend the encoding of tree automata as described in Remark 8. We want to guarantee that all states are reachable by

employing Lemma 10. However, instead of encoding an arbitrary well-founded relation, we make use of the fact that the names of states are irrelevant. Hence, without loss of generality (modulo renaming of states), we may assume that $s_1 < s_2 < \ldots < s_n$. We then encode the condition of Lemma 10 by formulas

$$\bigvee_{f \in \Sigma,\ q_1 < q, \ldots,\ q_{\#(n)} < q} v_{f,q_1,\ldots,q_n,q}$$

for every $q \in Q$.

The following lemma is immediate.

▶ **Lemma 12.** *Let $A = \langle Q, \Sigma, F, \delta \rangle$ be a tree automaton such that all states are reachable. Then $\mathcal{L}(A) \neq \varnothing$ if and only if $F \neq \varnothing$.*

▶ **Remark 13** (SAT encoding of $\mathcal{L}(A) \neq \varnothing$)**.** In a setting where all states are reachable, the encoding of $\mathcal{L}(A) \neq \varnothing$ as satisfiability problem trivializes to: $\bigvee_{q \in Q} v_{F,q}$.

The following lemma gives a simple criterion for closure under rewriting.

▶ **Lemma 14** (Genet [24, Proposition 12])**.** *Let $A = \langle Q, \Sigma, F, \delta \rangle$ be a tree automaton and R a left-linear term rewriting system. Then $\mathcal{L}(A)$ is closed under rewriting with respect to R if for every $\ell \to r \in R$, $\alpha : \mathcal{X} \to Q$ and $q \in Q$ we have $\ell\alpha \rightsquigarrow_A^* q \implies r\alpha \rightsquigarrow_A^* q$.*

Note that left-linearity of R is crucial for the Lemma 14 since A can be a non-deterministic automaton. If R would contain non-left-linear rules $\ell \to r$ then we would need to check set-assignments $\alpha : \mathcal{X} \to \wp(Q)$ instead $\alpha : \mathcal{X} \to Q$. That is, we would need to take into account, that a non-deterministic automaton can interpret the same term by different states.

For terms t, we use $\mathit{Var}(t)$ to denote the set of variables occurring in t.

▶ **Remark 15** (SAT encoding of closure under rewriting)**.** We encode the conditions of Lemma 14. Let the automaton A be encoded as in Remark 8. Let U be the set of all non-variable subterms of left-hand sides and right-hand sides of rules in R. For every $t \in U$, assignment $\alpha : \mathit{Var}(t) \to Q$ and $q \in Q$ we introduce a fresh variable

$$v_{t,\alpha,q} \qquad \text{with the intended meaning: } v_{t,\alpha,q} \text{ is true} \iff t\alpha \rightsquigarrow^* q\ .$$

We ensure this meaning by the following formulas: for terms $t = f(t_1, \ldots, t_n) \in U$

$$v_{t,\alpha,q} \longleftrightarrow \bigvee_{q_1,\ldots,q_n \in Q} \left(v_{t_1,\alpha_1,q_1} \wedge \ldots \wedge v_{t_n,\alpha_n,q_n} \wedge v_{f,q_1,\ldots,q_n,q} \right)$$

where α_i is the restriction of α to the domain $\mathit{Var}(t_i)$. For variables $x \in U$, we stipulate $v_{x,\alpha,q} \iff \alpha(x) = q$; note that we can immediately evaluate and fill in these truth values. Finally, we encode $\ell\alpha \rightsquigarrow_A^* q \implies r\alpha \rightsquigarrow_A^* q$ by formulas

$$v_{\ell,\alpha,q} \to v_{r,\alpha,q}$$

for every $\ell \to r \in R$, $\alpha : \mathit{Var}(\ell) \to Q$ and $q \in Q$.

The following modification of Lemma 14 gives a simple criterion for weak closure under rewriting. The requirement $r\alpha \rightsquigarrow_A^* q$ of Lemma 14 is weakened to $t\alpha \rightsquigarrow_A^* q$ for some reduct t the left-hand side ℓ.

▶ **Lemma 16.** *Let $A = \langle Q, \Sigma, F, \delta \rangle$ be a tree automaton and R a left-linear term rewriting system. Then $\mathcal{L}(A)$ is weakly closed under rewriting with respect to R if for every $\ell \to r \in R$, $\alpha : \mathcal{X} \to Q$ and $q \in Q$ we have $\ell\alpha \leadsto_A^* q \implies t\alpha \leadsto_A^* q$ for some term t such that $\ell \to_R^+ t$.*

Proof. Let $s \in \mathcal{L}(A) \setminus \mathsf{NF}(\to_R)$. Then $s = C[\ell\sigma]$ for a context C, rewrite rule $\ell \to r \in R$ and substitution $\sigma : \mathcal{X} \to T(\Sigma, \varnothing)$. Since $s \in \mathcal{L}(A)$ there exists $q \in Q$ such that $\ell\sigma \leadsto^* q$ and $C[q] \leadsto^* q'$ with $q' \in F$. By left-linearity ℓ does not contain duplicated occurrences of variables. As a consequence, there exists $\alpha : \mathcal{X} \to Q$ such that $\sigma(x) \leadsto^* \alpha(x)$ and $\ell\alpha \leadsto^* q$. By the assumptions of the lemma, a term t exists such that $\ell \to_R^+ t$ and $t\alpha \leadsto^* q$. Hence $t\sigma \leadsto^* q$ and $C[t\sigma] \leadsto^* C[q] \leadsto^* q'$. Thus $C[t\sigma] \in \mathcal{L}(A)$. Since $s = C[\ell\sigma] \to_R^+ C[t\sigma]$, this proves that $\mathcal{L}(A)$ is weakly closed under rewriting with respect to R. ◀

▶ **Remark 17** (SAT encoding of Lemma 16)**.** The conditions of Lemma 16 can be encoded similar to Lemma 14 (described in Remark 15). In Lemma 16 the condition $r\alpha \leadsto_A^* q$ is weakened to: $t\alpha \leadsto_A^* q$ for some reduct t the left-hand side ℓ. We can pick a finite set of reducts $U \subseteq \{t \mid \ell \to^+ t\}$ of the left-hand side ℓ, and encode the disjunction $\bigvee_{t \in U} t\alpha \leadsto_A^* q$ as a Boolean satisfiability problem. Note that $U \neq \varnothing$ since $r \in U$.

Next, we want to guarantee that the language $\mathcal{L}(A)$ contains no normal forms, in other words, that every term in the language contains a redex. For left-linear term rewriting systems R, we can reduce this problem to language inclusion $\mathcal{L}(A) \subseteq \mathcal{L}(B)$ where B is a tree automaton that accepts the language of reducible terms. If R is a left-linear rewrite system, then the set of ground terms containing redex occurrences is a regular tree language. A deterministic automaton B for this language can be constructed using the overlap-closure of subterms of left-hand sides, see further [15, 16]. Here, we do not repeat the construction, but state the lemma that we will employ:

▶ **Lemma 18.** *Let $\{\ell_1, \ldots, \ell_n\}$ be a set of linear terms over Σ. Then we can construct a deterministic and complete automaton $B = \langle Q, \Sigma, F, \delta \rangle$ and sets $F_{\ell_1}, \ldots, F_{\ell_n} \subseteq Q$ such that for every term $t \in T(\Sigma, \varnothing)$ and $i \in \{1, \ldots, n\}$ we have:*

- *$t \leadsto^* q$ with $q \in F_{\ell_i}$ if and only if t is an instance of ℓ_i.*

Note that by choosing $F = F_{\ell_1} \cup \ldots \cup F_{\ell_n}$ we obtain: $t \leadsto^* q$ with $q \in F$ if and only if t is an instance ℓ_i for some $i \in \{1, \ldots, n\}$.

▶ **Example 19.** The following tree automaton $B_S = \langle Q, \Sigma, F, \delta \rangle$ accepts the language of ground terms that contain a redex occurrence with respect to the S-rule $a(a(a(S, x), y), z) \to a(a(x, z), a(y, z))$. Here $Q = \{0, 1, 2, 3\}$, $\Sigma = \{a, S\}$, $F = \{3\}$ and

$$S \leadsto 0 \qquad a(0, q) \leadsto 1 \qquad a(1, q) \leadsto 2 \qquad a(2, q) \leadsto 3 \qquad a(3, q) \leadsto 3 \qquad a(q', 3) \leadsto 3$$

for all $q \in \{0, 1, 2\}$ and $q' \in \{0, 1, 2, 3\}$.

Since the automaton B_S is deterministic and complete, we can obtain an automaton $\overline{B_S} = \langle Q, \Sigma, \overline{F}, \delta \rangle$ that accepts the complement of the language (the language of ground normal forms) by taking the complement $\overline{F} = \{0, 1, 2\}$ of the set of final states.

The following is crucial for feasibility of our approach. Deciding language inclusion of non-deterministic automata is known to be EXPTIME complete, see [30]. However, to guarantee that a language contains no normal forms, it suffices to check whether two non-deterministic automata have a non-empty intersection. This property can be decided in polynomial time by constructing the product automaton and considering the reachable states.

▶ **Definition 20.** The *product* $A \times B$ of tree automata $A = \langle Q, \Sigma, F, \delta \rangle$ and $B = \langle Q', \Sigma, F', \delta' \rangle$ is the tree automaton $C = \langle Q \times Q', \Sigma, F \times F', \gamma \rangle$ where the transition relation γ is given by

$$f(\,(q_1, p_1), \ldots, (q_n, p_n)\,) \rightsquigarrow_\gamma (q', p') \iff f(q_1, \ldots, q_n) \rightsquigarrow_\delta q' \;\wedge\; f(p_1, \ldots, p_n) \rightsquigarrow_{\delta'} p'$$

for every $f \in \Sigma$ of arity n and states $q_1, \ldots, q_n, q' \in Q$ and $p_1, \ldots, p_n, p' \in Q'$.

▶ **Lemma 21.** *Let $A = \langle Q, \Sigma, F, \delta \rangle$ and $B = \langle Q', \Sigma, F', \delta' \rangle$ be tree automata. Then we have $\mathcal{L}(A) \cap \mathcal{L}(B) = \varnothing$ if and only if in $A \times B$ no state in $F \times F'$ is reachable.*

Proof. Let $A \times B = \langle Q \times Q', \Sigma, \varnothing, \gamma \rangle$. For the 'if'-part, assume that $\mathcal{L}(A) \cap \mathcal{L}(B) \neq \varnothing$. Let $t \in \mathcal{L}(A) \cap \mathcal{L}(B)$. Then $t \rightsquigarrow_\delta^* q$ for some $q \in F$ and $t \rightsquigarrow_{\delta'}^* q'$ for some $q' \in F'$. But then $t \rightsquigarrow_\gamma^* (q, q')$ and hence $(q, q') \in F \times F'$ is reachable in $A \times B$; this contradicts the assumption.

For the 'only if'-part, assume, for a contradiction, that $t \rightsquigarrow_\gamma^* (q, q')$ in $A \times B$ with $q \in F$ and $q' \in F'$. Then this directly translates to $t \rightsquigarrow_\delta^* q$ in A and $t \rightsquigarrow_{\delta'}^* q'$ in B. Hence $t \in \mathcal{L}(A)$ and $t \in \mathcal{L}(B)$, contradicting $\mathcal{L}(A) \cap \mathcal{L}(B) = \varnothing$. ◀

We can use Lemma 21 to check that the language $\mathcal{L}(A)$ of an automaton A does not contain normal forms. To this end, we only need an automaton B that accepts all ground normal forms. Then $\mathcal{L}(A)$ contains no normal forms if $\mathcal{L}(A) \cap \mathcal{L}(B) = \varnothing$.

▶ **Example 22.** The reachable states of the product $A_S \times \overline{B_S}$ of the automata A_S from Example 7 and $\overline{B_S}$ from Example 19 are $(0,0), (1,1), (2,2), (2,1), (3,3), (3,2), (2,3), (4,3)$. The only state (q, q') such that q is accepting in A_S is $(4,3)$ and 3 is not an accepting state of $\overline{B_S}$. The conditions of Lemma 21 are fulfilled and hence $\mathcal{L}(A_S) \cap \mathcal{L}(\overline{B_S}) = \varnothing$. Recall that $\overline{B_S}$ accepts all ground normal forms, and thus every term accepted by A_S contains a redex.

▶ **Remark 23** (SAT encoding of language inclusion)**.** Let $A = \langle Q, \Sigma, F, \delta \rangle$ and $B = \langle Q', \Sigma, F', \delta' \rangle$ be tree automata. Let $A \times B = \langle Q \times Q', \Sigma, \varnothing, \gamma \rangle$.

First, note that reachability of all states in the automata A and B does not imply that all states in the product automaton $A \times B$ are reachable. As a consequence, we have to 'compute' the set of reachable states using Boolean satisfiability problems. For this purpose, we reformulate Lemma 21 in the following equivalent way: ..., then $\mathcal{L}(A) \cap \mathcal{L}(B) = \varnothing$ if and only if there exists a set of states $P \subseteq Q \times Q'$ such that

(a) P is *closed under transitions* in $A \times B$, that is, $q \in P$ whenever $f(q_1, \ldots, q_n) \rightsquigarrow_\gamma q$ for some $q_1, \ldots, q_n \in P$, and
(b) for all $(q, q') \in P$ it holds that $q \in F$ implies $q' \notin F'$.

Note that this statement is equivalent to Lemma 21. Item (a) guarantees that P contains all reachable states, and hence (b) is required for at least the reachable states. Thus the conditions imply those of Lemma 21. On the other hand, we can take P to be precisely the set of reachable states, and then the conditions are exactly those of Lemma 21.

The idea is that the reformulated statement has a much more efficient encoding as Boolean satisfiability problem. We only need to encode the closure of P under transitions, but there is no longer the need for encoding the property that P is the smallest such set (which is a statement of second-order logic).

Assume that we have a SAT encoding of the automata A and B as in Remark 8; we write $v_{A,\ldots}$ for the variables encoding A, and $v_{B,\ldots}$ for the variables encoding B. To represent the set P, we introduce variables $p_{(q,q')}$ for every $(q, q') \in Q \times Q'$ and the properties are translated into the following formulas:

(a) for every $f \in \Sigma$ with arity n and $(q_1, q_1'), \ldots, (q_n, q_n'), (q, q') \in Q \times Q'$:

$$(v_{A,f,q_1,\ldots,q_n,q} \;\wedge\; v_{B,f,q_1',\ldots,q_n',q'} \;\wedge\; p_{(q_1,q_1')} \;\wedge\; p_{(q_2,q_2')} \;\wedge\; \ldots \;\wedge\; p_{(q_n,q_n')}) \;\to\; p_{(q,q')}$$

(b) for every $(q, q') \in Q \times Q'$: $(p_{(q,q')} \wedge v_{A,F,q}) \to \neg v_{B,F',q'}$.

Each of these formulas simplifies to a single clause (a disjunction of literals).

We remark that we will employ this translation for the case that B consists of the set of terms containing redex occurrences with respect to a given rewrite system R. Then B is known and fixed before the translation to a satisfiability problem. As a consequence, we know the truth values of $v_{B,f,q_1',\ldots,q_n',q'}$ and $v_{B,F',q'}$ in the formulas above, and can immediate skip the generation of formulas that are trivially true (the large majority in case (a)).

▶ **Remark 24** (Complexity of the SAT encoding)**.** While the encoding is efficient for string rewriting systems, it suffers from an 'encoding explosion' for term rewriting systems containing symbols of higher arity. The problem arises from the SAT encoding of the recursive computation of the interpretation of terms (described in Remark 15). The computation of the interpretation of a term $f(t_1, \ldots, t_n)$ containing m variables needs $O(|Q|^{m+n+1})$ clauses: m for the quantification over the variable assignments, n for the possible states of $t_1, \ldots, t_n$ and 1 for the possible result states. To some extend, this problem can be overcome by 'uncurrying' the system, that is, for every symbol f of arity $n > 2$ we introduce fresh symbols $f_1, \ldots, f_{n-1}$ of arity 2 and then replace all occurrences of $f(t_1, \ldots, t_n)$ by $f_{n-1}(\ldots f_2(f_1(t_1, t_2), t_3) \ldots, t_n)$. This transformation helps to bring the complexity down to $O(|Q|^{m+3})$. Nevertheless, for example for the S-rule, which only contains binary symbols, we still need $|Q|^6$ clauses. We note that after the uncurrying transformation, an automaton with more states may be needed to generate 'the same' language.

4.2 Disproving Weak Normalization

We are now ready to use Theorem 3 in combination with tree automata for automatically disproving weak normalization. The language L in the theorem is described by a non-deterministic tree automaton. In the previous section, we have seen how the relevant properties of tree automata can be checked. Here, we summarize the procedure:

▶ **Technique 25.** Let R be a left-linear TRS. We search for a tree automaton $A = \langle Q, \Sigma, F, \delta \rangle$ such that $\mathcal{L}(A)$ fulfills the properties of Theorem 3:

(i) We guarantee $\mathcal{L}(A) \cap \mathsf{NF}(\to) = \varnothing$ by the following steps:
- We employ Lemma 18 to construct a deterministic, complete automaton $B = \langle Q, \Sigma, F, \delta \rangle$ that accepts the set of terms containing redex occurrences with respect to R.
- Then the automaton $\overline{B} = \langle Q, Q \setminus \Sigma, F, \delta \rangle$ accepts all ground normal forms.
- We use Lemma 21 to check that $\mathcal{L}(A) \cap \mathcal{L}(\overline{B}) = \varnothing$ (thus $\mathcal{L}(A) \subseteq \mathcal{L}(B)$).

(ii) We guarantee that $\mathcal{L}(A)$ is closed under $\to$ by Lemma 14.

(iii) We use Lemma 12 to ensure that $\mathcal{L}(A) \neq \varnothing$.

These conditions can be encoded as satisfiability problems as described in Remarks 8, 11, 23, 13 and 15. This enables us to utilize SAT solvers to search for suitable automata A.

▶ **Remark 26.** The technique 25 can be slightly strengthened by first eliminating collapsing rules, that is, rules of the form $\ell \to x$ with $x \in \mathcal{X}$. Assume that the TRS R contains a collapsing rule $\ell \to x$. For every $f \in \Sigma$ we define the substitution $\sigma_f : \mathcal{X} \to T(\Sigma, \mathcal{X})$ by $\sigma_f(x) = f(x_1, \ldots, x_{\#(f)})$ for fresh variables $x_1, \ldots, x_{\#(f)}$ and $\sigma_f(y) = y$ for all $y \neq x$. We define $R' = (R \setminus \{\ell \to x\}) \cup \{\ell\sigma_f \to x\sigma_f \mid f \in \Sigma\}$. Then $\to_R$ and $\to_{R'}$ coincide on ground terms and hence R is (weakly) ground normalizing if and only if R is.

▶ **Remark 27.** We note the combination of Technique 25 with Remark 26 is complete with respect to disproving weak normalization on regular languages: if there exists a regular language L fulfilling the conditions of Theorem 3, then weak normalization can be disproved using Technique 25 after eliminating collapsing rules as in Remark 26.

This can be seen as follows. In the work [8, 9] a generalized method for ensuring closure of the language of automata under rewriting has been proposed. Thereby the condition $\ell\alpha \rightsquigarrow_A^* q \implies r\alpha \rightsquigarrow_A^* q$ of Lemma 14 is weakened to

$$\ell\alpha \rightsquigarrow_A^* q \implies r\alpha \rightsquigarrow_A^* p \quad \text{for some } p \geq q \,. \tag{1}$$

Here $\leq$ is a quasi-order on the states Q and the automaton must be monotonic with respect to this order, see Definition 34. The monotonicity guarantees that the language of the automaton is closed under rewriting.

In [23] it has been shown that this monotonicity property is strong enough to characterize and decide the closure of the regular languages under rewriting. In particular, the language of a deterministic tree automaton is closed under rewriting if and only if there exists such a monotonic quasi-order on the states.

Let R be a TRS such that there exists a regular language that satisfies the conditions of Theorem 3. Then there exists a deterministic, complete automaton A accepting this language and a quasi-order $\leq$ on the states satisfying (1) and monotonicity. Let R' be obtained from R by eliminating collapsing rules as described in Remark 26. We obtain a non-deterministic automaton A' that fulfils the requirements of Technique 25 for R' by closing the transition relation of A under $\leq$: we add $f(q_1,\ldots,q_n) \rightsquigarrow q$ whenever $q \leq p$ and $f(q_1,\ldots,q_n) \rightsquigarrow p$. As a consequence of monotonicity and using induction over the term structure, we obtain for all terms $t \in T(\Sigma,\mathcal{X})$ with $t \notin \mathcal{X}$ and $\alpha : \mathcal{X} \to Q$ that

($\star$) $t \rightsquigarrow_{A'}^* q$ if and only if $t \rightsquigarrow_A^* p$ for some p with $q \leq p$.

As a consequence of ($\star$) and monotonicity we have $\mathcal{L}(A') = \mathcal{L}(A)$ (roughly speaking, if $q \leq p$, then q accepts a subsets of the language of p). Thus $\mathcal{L}(A') \cap \mathsf{NF}(\to) = \varnothing$ and $\mathcal{L}(A') \neq \varnothing$ are guaranteed. Finally, we show that Lemma 14 is applicable for R' and A'. Let $\ell \to r \in R'$, $\alpha : \mathcal{X} \to Q$ and $q \in Q$ such that $\ell\alpha \rightsquigarrow_{A'}^* q$. Then by ($\star$) we get $\ell\alpha \rightsquigarrow_A^* q'$ for some $q' \in Q$ with $q \leq q'$. By (1) we have that $r\alpha \rightsquigarrow_A^* q''$ for some $q'' \in Q$ with $q' \leq q''$. Again by ($\star$) we obtain that $r\alpha \rightsquigarrow_A^* q$. Hence the conditions of Technique 25 are fulfilled for R' and A'.

▶ **Example 28.** We consider the following string rewriting system:

$$aL \to La \qquad Ra \to aR \qquad bL \to bR \qquad Rb \to Lab$$

This rewrite system is neither strongly nor weakly normalizing, but does not admit looping reductions, that is, reductions of the form $s \to^+ \ell sr$. An example of an infinite reduction is:

$$bLb \to bRb \to bLab \to bRab \to baRb \to baLab \to bLaab \to bRaab \to \cdots$$

It is easy to check that the automaton A_{LR} from Example 6 fulfills the requirements of Technique 25. Hence, the system is not weakly normalizing.

▶ **Example 29.** We consider the S-rule from combinatory logic:

$$a(a(a(S,x),y),z) \to a(a(x,z),a(y,z))$$

For the S-rule it is known that there are no reductions $t \to^* C[t]$ for ground terms t, see [31]. For open terms t the existence of reductions $t \to^* C[t\sigma]$ is open.

It is straightforward to verify that the automaton A_S from Example 7 fulfills the requirements of Technique 25, and hence the S-rule, and in particular the term $SSS(SSS)(SSS(SSS))$, are not weakly normalizing.

▶ **Example 30.** The δ-rule (known as Owl in Combinatory Logic) is even simpler:

$$\delta xy \to y(xy), \qquad \text{or equivalently} \qquad a(a(\delta, x), y) \to a(y, a(x, y))\ .$$

As shown in [4], this rule does not admit loops, , and the techniques in [7] fail for this system. The Technique 25 can be applied to automatically disprove weak-normalization for this rule. Our tool finds a tree automaton that has 3 states and accepts the language of all ground terms with two occurrences of $\delta\delta$. In fact, this is precisely the language of non-terminating ground δ-terms, see further [4].

In all examples until now infinite reductions exist of the regular shape based on $t\sigma^n\tau$ rewriting to a term having $t\sigma^{f(n)}\tau\mu$ as a sub-term, for every n, for some term t and substitutions σ, τ, μ and an ascending linear function f. For instance, the S rule (Example 29) admits an infinite reduction implied by $t\sigma^n\tau$ rewriting to a super-term of $t\sigma^{n+1}\tau$, for $t = a(x, x)$, $\sigma(x) = Ax$, $\tau(x) = SA(SAA)$, for $A = SSS$.

▶ **Example 31.** The following example does not have an infinite reduction of this regular shape, neither of the more general patterns from [29] and [7].

$$aL \to La \qquad Raa \to aaaR \qquad bL \to bRa \qquad Rb \to Lb \qquad Rab \to Lab.$$

In this system bRa^nb rewrites to $bRa^{f(n)}b$ for f defined by $f(2n) = 3n + 1$ and $f(2n + 1) = 3n + 2$ for all n. This obviously yields an infinite reduction, but f is not linear, by which this example is outside the scope of [29] and [7]. In our approach a proof of non-termination and even non-weak-normalization is extremely simple: $ba^*(L \mid R)a^*b$ is non-empty, closed under rewriting and does not contain normal forms.

4.3 Disproving Strong Normalization

For disproving strong normalization based on Theorem 2, the only difference with Technique 25 is that checking that L is closed under $\to$ by Lemma 14 has to be replaced by checking that L is weakly closed under $\to$ by Lemma 16. The technique is applicable to string and term rewriting systems, and can be automated as described in Technique 25 and Remark 17.

▶ **Example 32.** Let us consider the rewrite system

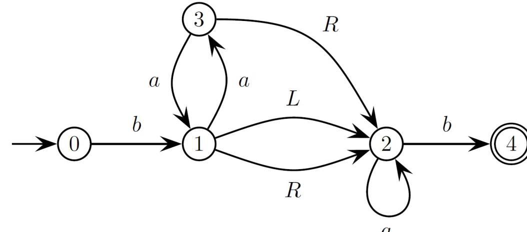
$$aaL \to Laa \qquad Ra \to aR \qquad bL \to bR \qquad Rb \to Lab \qquad Rb \to aLb$$

This system is non-looping and non-terminating. However, in contrast to Example 28, this system is weakly normalizing, since by always choosing the fourth rule the last rule is never used, and the first four rules are terminating. Hence the Technique 25 is not applicable for this TRS. However, the following pattern extends to an infinite reduction

$$bRa^{2n}b \to^{2n} ba^{2n}Rb \to ba^{2n}Lab \to^{n} bLa^{2n+1}b \to$$

$$bRa^{2n+1}b \to^{2n+1} ba^{2n+1}Rb \to ba^{2n+2}Lb \to^{n+1} bLa^{2n+2}b \to bRa^{2n+2}b.$$

Instead of finding this pattern explicitly, non-termination is also concluded from checking that $b(aa)^*(L \mid R \mid aR)a^*b$ describes a language satisfying all conditions from Theorem 2. A corresponding automaton is given on the right.



The conditions of Theorem 2 are now checked as follows. Non-emptiness follows from the existence of a path from state 0 to state 4. Every path from 0 to 4 either contains one of the patterns aaL, Ra, bL or Rb, so it remains to show weakly closedness under rewriting by Lemma 16. In the setting of string automata this means that for every left hand side ℓ and every ℓ-path from a state p to a state q we should find a u-path from p to q for a string u such that ℓ rewrites to u in one or more steps. For $\ell = aaL$ the only path is from 1 to 2, for which there is also an Laa path. For $\ell = Ra$ there is a path from 1 to 2, for which there is also an aR path via 3. The only other option for $\ell = Ra$ is a path from 3 to 2, for which there is also an aR path via 1. For $\ell = bL$ the only path is from 0 to 2, for which there is also a bR path. Finally, for $\ell = Rb$ there is a path from 1 to 4, for which there is also an Lab path and a path from 3 to 4, for which there is also an aLb path, by which all conditions have been verified. Note that for the Rb-path from 1 to 4 it is essential to use the 4th rule, while for the Rb-path from 3 to 4 it is essential to use the last rule.

This example can also be treated by the technique introduced in the following section.

5 Improved Methods for Disproving Strong Normalization

In this section, we improve the method for proving non-termination. The methods introduced so far are not able to handle the following example.

▶ **Example 33.** We consider the following string rewriting system:

$$zL \to Lz \qquad Rz \to zR \qquad zLL \to zLR \qquad RRz \to LzRz$$

This rewrite system is weakly normalizing but not strongly normalizing. The non-termination criteria introduced in the previous sections are not applicable for this system. Let us consider the first steps of an infinite reduction:

$$\begin{aligned} & \underline{zLL}zzRz \\ \to\ & zLRzzRz \to zLzRzRz \to zLzzRRz \\ \to\ & zLzzLzRz \to zLzLzzRz \to \underline{zLL}zzzRz \\ \to\ & \ldots \end{aligned}$$

Note the underlined occurrences of zLL. Due to the rule $zLL \to zLR$, the word zL is the marker for 'turning' on the left; However, this marker zL is itself a redex. To obtain an infinite reduction, this marker must not be reduced.

The idea for proving non-termination of systems like Example 33 is to let the automaton determine which redex to contract. To this end, we introduce a 'redex selection' function

$$\xi : Q \to \mathcal{P}(R)$$

that maps states of the automaton to sets of rules that may be contracted at the corresponding position in the term. The idea is that a redex $\ell\sigma$ in a term $C[\ell\sigma]$ with respect to a rule $\ell \to r$ is allowed to be contracted if $\ell\sigma \rightsquigarrow^* q$ with $\ell \to r \in \xi(q)$. In this way, the automaton determines what redexes are to be contracted. Then the automaton only needs to fulfill the property $\ell\alpha \rightsquigarrow_A^* q \implies r\alpha \rightsquigarrow_A^* q$ for the *selected rules*:

$$\ell \to r \in \xi(q) \ \wedge\ \ell\alpha \rightsquigarrow_A^* q \quad \implies \quad r\alpha \rightsquigarrow_A^* q$$

for every rule $\ell \to r \in R$, state $q \in Q$ and $\alpha : \mathcal{X} \to Q$. Moreover, as proposed in [8, 9, 23], we weaken the requirement $r\alpha \rightsquigarrow_A^* q$ to $r\alpha \rightsquigarrow_A^* p$ for some $p \geq q$. Here $\leq$ is a quasi-order on the states and the automaton must be monotonic with respect to this order (see Definition 34). The monotonicity guarantees that the language of the automaton is closed under rewriting. For the present paper, this closure property holds only for the rules selected by ξ.

▶ **Definition 34** (Monotonicity). A tree automaton $A = \langle Q, \Sigma, F, \delta \rangle$ is *monotonic* with respect to a quasi-order $\leq$ on the states Q if the following properties hold:

(i) For all $f \in \Sigma$ with arity n and states $a_1 \leq b_1,\ a_2 \leq b_2,\ \dots,\ a_n \leq b_n$, it holds

$$f(a_1, \dots, a_n) \rightsquigarrow_A q \quad \Longrightarrow \quad f(b_1, \dots, b_n) \rightsquigarrow_A p \ \text{ for some } p \in Q \text{ with } q \leq p$$

(ii) Whenever $q \in F$ and $q \leq p$, then $p \in F$.

The following lemma is immediate by induction on the size of the context.

▶ **Lemma 35.** *Let $A = \langle Q, \Sigma, F, \delta \rangle$ be a tree automaton that is monotonic with respect to a quasi-order $\leq$ on the states Q. Let $a, b \in Q$ with $a \leq b$. Then for all ground contexts C we have that $C[a] \rightsquigarrow a'$ with $a' \in Q$ implies that $C[b] \rightsquigarrow b'$ for some $b' \in Q$ with $a' \leq b'$.*

▶ **Definition 36** (Runs). Let $A = \langle Q, \Sigma, F, \delta \rangle$ be a tree automaton and $t \in T(\Sigma, \varnothing)$. A *run* of A on t is a function $\rho : \mathit{Pos}(t) \to Q$ such that for every $p \in \mathit{Pos}(t)$ and $t(p) = f \in \Sigma$ there is a rule $f(\rho(p1), \dots, \rho(pn)) \rightsquigarrow \rho(p)$ in δ. The run ρ is accepting if $\rho(\varepsilon) \in F$.

Note that there is a direct correspondence between runs on t and rewrite sequences $t \rightsquigarrow^* q$. We are now ready to state the generalized theorem for disproving strong normalization.

▶ **Theorem 37.** *Let R be a left-linear TRS. Let $A = \langle Q, \Sigma, F, \delta \rangle$ be a tree automaton with $\mathcal{L}(A) \neq \varnothing$, $\leq$ a quasi-order on the states Q, and $\xi : Q \to \mathcal{P}(R)$ a function, called* redex selection function*. Assume that the following properties hold:*

(a) *The automaton A is monotonic with respect to $\leq$.*

(b) *For every state $q \in Q$, rule $\ell \to r \in \xi(q)$ and $\alpha : \mathcal{X} \to Q$ it holds that:*

$$\ell\alpha \rightsquigarrow_A^* q \quad \Longrightarrow \quad (\exists p \in Q.\ q \leq p \ \wedge\ r\alpha \rightsquigarrow_A^* p) \vee$$
$$(\exists r' \trianglelefteq r.\ \exists q' \in F.\ r'\alpha \rightsquigarrow_A^* q')$$

(c) *For every term $t \in T(\Sigma, \varnothing)$ and accepting run ρ on t there is a position p such that $t|_p$ is an instance of the left-hand side of a rule $\ell \to r \in \xi(\rho(p))$.*

Then R is not strongly normalizing.

Proof. Assume that the conditions of the theorem are fulfilled. To disprove strong normalization of $\to$ it suffices to disprove strong normalization of $\to \circ \trianglerighteq$ where $\trianglerighteq$ is the (non-strict) sub-term relation. We show that $\mathcal{L}(A)$ and $\to \circ \trianglerighteq$ fulfill the requirements of Theorem 2. Let $t \in \mathcal{L}(A)$. Then there exists an accepting run ρ of A on t. By item 3 there exists a position $p \in \mathit{Pos}(t)$ and a rule $\ell \to r \in \xi(\rho(p))$ such that $t|_p$ is an instance of ℓ. Then $t|_p = \ell\sigma$ for some substitution σ. By left-linearity, we can define $\alpha : \mathit{Var}(\ell) \to Q$ by $\alpha(x) = \rho(pp')$ whenever $\ell|_{p'} \in \mathcal{X}$. Then $\ell\alpha \rightsquigarrow^* \rho(p)$ and we distinguish cases according to item 2:

1. There exists $q \in Q$ with $\rho(p) \leq q$ and $r\alpha \rightsquigarrow^* q$. We know that $t[\ell\sigma]_p = t \rightsquigarrow^* \rho(\varepsilon)$ and define $t' = t[r\sigma]_p$. Note that $t \to t'$ and $t \to \circ \trianglerighteq t'$. We have $t \rightsquigarrow t[\rho(p)] \rightsquigarrow \rho(\varepsilon)$ and $\rho(p) \leq q$. By Lemma 35 we have $t[q] \rightsquigarrow q'$ for some $q' \geq \rho(\varepsilon)$ and by monotonicity $q' \in F$. Hence $t' = t[r\sigma]_p \rightsquigarrow t[q] \rightsquigarrow^* q'$. Thus $t' \in \mathcal{L}(A)$ and $t \to \circ \trianglerighteq t'$.
2. There exist $r' \trianglelefteq r$ and $q \in F$ such that $r'\alpha \rightsquigarrow^* q$. Then $r'\sigma \rightsquigarrow^* q$ and hence $r'\sigma \in \mathcal{L}(A)$. Moreover, $t \to \circ \trianglerighteq r'\sigma$.

This shows that $\mathcal{L}(A)$ contains no normal forms and is weakly closed under $\to \circ \trianglerighteq$. By Theorem 2, $\to \circ \trianglerighteq$ is not strongly normalizing and hence $\to$ is not strongly normalizing. ◀

▶ **Remark 38** (SAT encoding of the conditions of Theorem 37). The encoding of condition 1 of Theorem 37 is straightforward, and condition 2 is an easy extension of Remark 15. The requirement 3 can be encoded similar to Remark 23, as follows. Let $\ell_1, \dots, \ell_n$ be the left-hand sides of rules in R. Let B be the automaton and $F_{\ell_1}, \dots, F_{\ell_n}$ the sets of states obtained from Lemma 18. We construct the product automaton $A \times B$, and then we compute those states that are reachable without passing states (q, q') for which there exists $\ell \to r \in \xi(q)$ such that $q' \in F_\ell$ (that is, the rule $\ell \to r$ is selected by A and B confirms that the term is an instance of ℓ).

6 Experimental Results

We have implemented the improved method for disproving strong normalization (Theorem 37) presented in this paper. For the purpose of evaluating our techniques, the tool applies only the methods presented in this paper, and no other non-termination method like loop checks. The SAT solver employed for the evaluation results in this section is MiniSat [6]. Our tool can be downloaded from `http://joerg.endrullis.de/non-termination/`.

Our tool can automatically prove non-termination of all examples in this paper, including the S-rule and the δ-rule. The following table shows the size of the automata that are found by the tool as witnesses for non-termination for the examples in our paper:

Example	28	29	30	31	32	33
Number of states	4	5	3	4	5	6

Each of these automata has been found within less than a second on a dual core laptop.

We have also evaluated our methods on the database used in [7], consisting of 58 non-terminating term rewriting systems that do not admit loops. The tool AProVE recognizes 44 systems as non-terminating; an impressive 76%. An extension of AProVE with our method would increase the recognition by 8.5% to 84.5%. In other words, our method succeeds on 36% (that is 5 systems) of the remaining 14 systems for which AProVE did not find a proof. These 5 systems are:

- `nonloop/TRS/emmes/ex3_4.trs`
- `nonloop/TRS/own/challenge_fab.trs`
- `nonloop/TRS/own/downfrom.trs`
- `nonloop/TRS/own/ex_payet.trs`
- `nonloop/TRS/own/isList-List.trs`

In total, our tool succeeds for 26 of the 58 non-looping examples from [7]. The results suggest that our method and that of [7] are complementary and should be combined for maximum strength. The paper [7] explicitly mentions that the following example is beyond their techniques (this example is not part of the database above):

$$
\begin{aligned}
f(\mathrm{true}, \mathrm{true}, x, \mathrm{s}(y)) &\to f(\mathit{isNat}(x), \mathit{isNat}(y), \mathrm{s}(x), \mathit{double}(\mathrm{s}(y))) \\
\mathit{isNat}(0) &\to \mathrm{true} \\
\mathit{isNat}(\mathrm{s}(x)) &\to \mathit{isNat}(x) \\
\mathit{double}(0) &\to 0 \\
\mathit{double}(\mathrm{s}(x)) &\to \mathrm{s}(\mathrm{s}(\mathit{double}(x)))
\end{aligned}
$$

Our non-termination techniques can handle this system: the tool finds an automaton with 6 states within 3 seconds (using the transformation from Remark 24).

Finally, we have evaluated the tool on the termination problem database (TPDB). We have run our tool on all string and term rewriting systems (of the standard categories) that remained unsolved during the last full run of all tools in December 2013. For string rewriting, our tool was able to disprove termination for 13, and for term rewriting, for 8 systems of the unsolved systems. This corresponds to an increase of strength of 11.5% (114 + 13) for string rewriting and of 3% (274 + 8) for term rewriting. Let us mention that many of the 13 string rewriting systems actually admit loops, but very complicated ones, that are not found by the standard tools. These loops have been found in previous competitions by the tools Matchbox [32] and Knocked for Loops [36].

7 Conclusions and Future Work

In this paper, we have employed regular languages for proving non-termination. Instead of searching for an infinite reduction explicitly we search for a regular language with properties from which non-termination easily follows. After encoding these properties in a propositional formula, the actual search is done by a SAT solver. In some examples, like Example 31, a very simple corresponding regular language is quickly found by our approach, while the actual infinite reductions have a non-linear pattern being beyond earlier approaches.

For future work, it is interesting to investigate whether this approach can be extended to context-free (tree) languages; such an approach could potentially also generalize [7]. The question is whether there are efficient criteria to check the conditions of Theorem 2. For example, consider the following string rewriting system:

$$\begin{array}{ccccc} & bB \to Bb & bcd \to BcD & Dd \to dD & \\ aX \to abb & BX \to Xb & bcd \to XcY & YD \to dY & Ye \to dde \end{array}$$

This system admits for every $n > 1$ reductions of the form

$$a\,b^n\,c\,d^n\,e \to^* a\,B^{n-1}\,bcd\,D^{n-1}\,e \to^* a\,B^{n-1}\,XcY\,D^{n-1}\,e \to^* a\,b^{n+1}\,c\,d^{n+1}\,e$$

As a description of this pattern needs a context-free language, it is unlikely that a regular language exists that fulfills the requirements of Theorem 2.

As described in Remark 24, the SAT encoding of (non-deterministic) automata is not efficient for symbols of higher arity. We think that these problems can be overcome by more efficient encodings of automata. For example, the uncurrying transformation mentioned in Remark 24 can be seen as a restriction of the shape of the automata (the transition is computed argument by argument) instead of a transformation on the system. It would be interesting to investigate what other restrictions would lead to a more efficient representation of automata as Boolean satisfiability problems. Results in this direction can be of interest in various areas where automata are applied.

We think that it is also interesting to investigate whether the characterization of strong and weak normalization (Theorems 2 and 3) can be adapted to the setting of infinitary rewriting [20, 22, 14] with infinite terms and ordinal-length reductions; the interesting properties then are infinitary strong and weak normalization.

Finally, we note that equality of streams [17, 18, 37, 19] (infinite sequences of symbols) can be rendered as a non-termination problem (a comparison program running indefinitely if the streams are equal, and terminating as soon as a difference is found). It remains to be investigated whether non-termination techniques can be employed fruitfully for proving stream equality.

References

1. Homepage of AProVE, 2015. http://aprove.informatik.rwth-aachen.de.
2. Homepage of TTT2, 2015. http://cl-informatik.uibk.ac.at/software/ttt2/.
3. H.P. Barendregt, J. Endrullis, J.W. Klop, and J. Waldmann. Dance of the Starlings. 2015. To be published on arXiv.org.
4. H.P. Barendregt, J. Endrullis, J.W. Klop, and J. Waldmann. Songs of the Starling and the Owl. 2015. To be published on arXiv.org.
5. B. Cook. Priciples of program termination. http://research.microsoft.com/en-us/um/cambridge/projects/terminator/principles.pdf.
6. N. Eén and A. Biere. Effective Preprocessing in SAT through Variable and Clause Elimination. In *Proc. Conf. on Theory and Applications of Satisfiability Testing (SAT '05)*, volume 3569 of *LNCS*, pages 61–75. Springer, 2005.
7. F. Emmes, T. Enger, and J. Giesl. Proving Non-looping Non-termination Automatically. In *International Joint Conference on Automated Reasoning (IJCAR 2012)*, volume 7364 of *LNCS*, pages 225–240. Springer, 2012.
8. J. Endrullis, R. C. de Vrijer, and J. Waldmann. Local Termination. In *Proc. Conf. on Rewriting Techniques and Applications (RTA)*, volume 5595 of *LNCS*, pages 270–284. Springer, 2009.
9. J. Endrullis, R.C. de Vrijer, and J. Waldmann. Local Termination: Theory and Practice. *Logical Methods in Computer Science*, 6(3), 2010.
10. J. Endrullis, H. Geuvers, J. G. Simonsen, and H. Zantema. Levels of Undecidability in Rewriting. *Information and Computation*, 209(2):227–245, 2011.
11. J. Endrullis, H. Geuvers, and H. Zantema. Degrees of Undecidability in Term Rewriting. In *Proc. Int. Workshop on Computer Science Logic (CSL 2009)*, volume 5771 of *LNCS*, pages 255–270. Springer, 2009.
12. J. Endrullis, C. Grabmayer, D. Hendriks, J.W. Klop, and R.C. de Vrijer. Proving Infinitary Normalization. In *Postproc. Int. Workshop on Types for Proofs and Programs (TYPES 2008)*, volume 5497 of *LNCS*, pages 64–82. Springer, 2009.
13. J. Endrullis, C. Grabmayer, J.W. Klop, and V. van Oostrom. On Equal μ-Terms. *Theoretical Computer Science*, 412(28):3175–3202, 2011.
14. J. Endrullis, H. Hvid Hansen, D. Hendriks, A. Polonsky, and A. Silva. A Coinductive Framework for Infinitary Rewriting and Equational Reasoning. In *Proc. Conf. on Rewriting Techniques and Applications (RTA 2015)*, Leibniz International Proceedings in Informatics. Schloss Dagstuhl, 2015.
15. J. Endrullis and D. Hendriks. Transforming Outermost into Context-Sensitive Rewriting. *Logical Methods in Computer Science*, 6(2), 2010.
16. J. Endrullis and D. Hendriks. Lazy Productivity via Termination. *Theoretical Computer Science*, 412(28):3203–3225, 2011.
17. J. Endrullis, D. Hendriks, and R. Bakhshi. On the Complexity of Equivalence of Specifications of Infinite Objects. In *ACM SIGPLAN International Conference on Functional Programming (ICFP 2012)*, pages 153–164. ACM, 2012.
18. J. Endrullis, D. Hendriks, R. Bakhshi, and G. Roşu. On the complexity of stream equality. *Journal of Functional Programming*, 24(2–3):166–217, 2014.
19. J. Endrullis, D. Hendriks, and M. Bodin. Circular Coinduction in Coq Using Bisimulation-Up-To Techniques. In *Proc. Conf. on Interactive Theorem Proving (ITP)*, volume 7998 of *LNCS*, pages 354–369. Springer, 2013.
20. J. Endrullis, D. Hendriks, and J.W. Klop. Highlights in Infinitary Rewriting and Lambda Calculus. *Theoretical Computer Science*, 464:48–71, 2012.
21. J. Endrullis, D. Hofbauer, and J. Waldmann. Decomposing Terminating Rewrite Relations. In *Proc. Workshop on Termination (WST '06)*, pages 39–43, 2006.

22 J. Endrullis and A. Polonsky. Infinitary Rewriting Coinductively. In *Proc. Types for Proofs and Programs (TYPES 2012)*, volume 19 of *Leibniz International Proceedings in Informatics*, pages 16–27. Schloss Dagstuhl, 2013.

23 B. Felgenhauer and R. Thiemann. Reachability Analysis with State-Compatible Automata. In *LATA*, volume 8370 of *LNCS*, pages 347–359. Springer, 2014.

24 T. Genet. Decidable Approximations of Sets of Descendants and Sets of Normal Forms. In *Proc. Conf. on Rewriting Techniques and Applications (RTA '98)*, volume 1379 of *LNCS*, pages 151–165. Springer, 1998.

25 A. Geser, D. Hofbauer, J. Waldmann, and H. Zantema. On tree automata that certify termination of left-linear term rewriting systems. *Information and Computation*, 205(4):512–534, 2007.

26 A. Geser and H. Zantema. Non-looping String Rewriting. *RAIRO Theoretical Informatics and Applications*, 33(3):279–302, 1999.

27 M. Korp and A. Middeldorp. Match-bounds revisited. *Information and Computation*, 207(11):1259–1283, 2009.

28 M. Mousazadeh, B. T. Ladani, and H. Zantema. Liveness verification in trss using tree automata and termination analysis. *Computing and Informatics*, 29(3):407–426, 2010.

29 M. Oppelt. Automatische Erkennung von Ableitungsmustern in nichtterminierenden Wortersetzungssystemen. Technical report, HTWK Leipzig, Germany, 2008. Diploma Thesis.

30 Helmut Seidl. Deciding equivalence of finite tree automata. In *STACS 89*, volume 349 of *LNCS*, pages 480–492. Springer, 1989.

31 J. Waldmann. The Combinator S. *Information and Computation*, 159(1–2):2–21, 2000.

32 J. Waldmann. Matchbox: A Tool for Match-Bounded String Rewriting. In *Proc. Conf. on Rewriting Techniques and Applications (RTA '04)*, volume 3091 of *LNCS*, pages 85–94. Springer, 2004.

33 J. Waldmann. Compressed loops (draft), 2012.

34 H. Zankl and A. Middeldorp. Nontermination of String Rewriting using SAT, 2007.

35 H. Zankl, C. Sternagel, D. Hofbauer, and A. Middeldorp. Finding and certifying loops. In *Proc. Conf. on Theory and Practice of Computer Science (SOFSEM 2010)*, volume 5901 of *LNCS*, pages 755–766. Springer, 2010.

36 H. Zankl, C. Sternagel, D. Hofbauer, and A. Middeldorp. Finding and certifying loops. In *SOFSEM 2010: Theory and Practice of Computer Science*, volume 5901 of *LNCS*, pages 755–766. Springer, 2010.

37 H. Zantema and J. Endrullis. Proving Equality of Streams Automatically. In *Proc. Conf. on Rewriting Techniques and Applications (RTA 2011)*, pages 393–408, 2011.

Reachability Analysis of Innermost Rewriting

Thomas Genet and Yann Salmon

IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France
{Thomas.Genet,Yann.Salmon}@irisa.fr

Abstract

We consider the problem of inferring a grammar describing the output of a functional program given a grammar describing its input. Solutions to this problem are helpful for detecting bugs or proving safety properties of functional programs and, several rewriting tools exist for solving this problem. However, known grammar inference techniques are not able to take evaluation strategies of the program into account. This yields very imprecise results when the evaluation strategy matters. In this work, we adapt the Tree Automata Completion algorithm to approximate accurately the set of terms reachable by rewriting under the innermost strategy. We prove that the proposed technique is sound and precise w.r.t. innermost rewriting. The proposed algorithm has been implemented in the Timbuk reachability tool. Experiments show that it noticeably improves the accuracy of static analysis for functional programs using the call-by-value evaluation strategy.

1998 ACM Subject Classification I.2.3 Deduction and Theorem Proving, F.4.2 Grammars and Other Rewriting Systems, D.2.4 Software/Program Verification

Keywords and phrases term rewriting systems, strategy, innermost strategy, tree automata, functional program, static analysis

Digital Object Identifier 10.4230/LIPIcs.RTA.2015.177

1 Introduction and motivations

If we define by a grammar the set of inputs of a functional program, is it possible to infer the grammar of its output? Some strongly typed functional programming languages (like Haskell, OCaml, Scala and F#) have a type inference mechanism. This mechanism, among others, permits to automatically detect some kinds of errors in the programs. In particular, when the inferred type is not the expected one, this suggests that there may be a bug in the function. To prove properties stronger than well typing of a program, it is possible to define properties and, then, to prove them using a proof assistant or an automatic theorem prover. However, defining those properties with logic formulas (and do the proof) generally requires a strong expertise.

Here, we focus on a restricted family of properties: regular properties on the structures manipulated by those programs. Using a grammar, we define the set of data structures given as input to a function and we want to infer the grammar that can be obtained as output (or an approximation). Like in the case of type inference, the output grammar can suggest that the program contains a bug, or on the opposite, that it satisfies a regular property.

The family of properties that can be shown in this way is restricted, but it strictly generalises standard typing as used in languages of the ML family[1]. There are other approaches where the type system is enriched by logic formulas and arithmetic like [29, 5], but they

[1] Standard types can easily be expressed as grammars. The opposite is not true. For instance, with a grammar one can distinguish between an empty and a non empty list.

© Thomas Genet and Yann Salmon;
licensed under Creative Commons License CC-BY
26th International Conference on Rewriting Techniques and Applications (RTA'15).
Editor: Maribel Fernández; pp. 177–193
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

generally require to annotate the output of the function for type checking to succeed. The properties we consider here are intentionally simpler so as to limit as much as possible the need for annotations. The objective is to define a *lightweight* formal verification technique. The verification is *formal* because it *proves* that the results have a particular form. But, the verification is *lightweight* for two reasons. First, the proof is carried out automatically: no interaction with a prover or a proof assistant is necessary. Second, it is not necessary to state the property on the output of the function using complex logic formulas or an enriched type system but, instead, only to observe and check the result of an abstract computation.

With regards to the grammar inference technique itself, many works are devoted to this topic in the functional programming community [20, 25][2] as well as in the rewriting community [10, 27, 3, 14, 23, 2, 12]. In [12], starting from a term rewriting system (TRS for short) encoding a function and a tree automaton recognising the inputs of a function, it is possible to automatically produce a tree automaton *over-approximating* as precisely as possible the outputs. Note that a similar reasoning can be done on higher-order programs [16] using a well-known encoding of higher order functions into first-order TRS [20]. However, for the sake of simplicity, examples used in this paper will only be first order functions. This is implemented in the Timbuk tool [13]. Thus, we are close to building an *abstract interpreter*, evaluating a function on an (unbounded) regular set of inputs, for a real programming language. However, none of the aforementioned grammar inference techniques takes the evaluation strategy into account, though every functional programming language has one. As a consequence, those techniques produce very poor results as soon as the evaluation strategy matters or, as we will see, as soon as the program is not terminating. This paper proposes a grammar inference technique for the innermost strategy:

- overcoming the precision problems of [20, 25] and [10, 27, 3, 14, 23, 2, 12] on the analysis of functional programs using call-by-value strategy
- whose accuracy is not only shown on a practical point of view but also formally proved. This is another improvement w.r.t. other grammar inference techniques (except [14]).

1.1 Towards an abstract OCaml interpreter

In the following, we assume that we have an abstract OCaml interpreter. This interpreter takes a regular expression as an input and outputs another regular expression. In fact, all the computations presented in this way have been performed with Timbuk (and latter with TimbukSTRAT), but on a TRS and a tree automaton rather than on an OCaml function and a regular expression. We made this choice to ease the understanding of input and output languages, since regular expressions are far more easier to read and to understand than tree automata. Assume that we have a notation, inspired by regular expressions, to define regular languages of lists. Let us denote by `[a*]` (resp. `[a+]`) the language of lists having 0 (resp. 1) or more occurrences of symbol `a`. We denote by `[(a|b)*]` any list with 0 or more occurrences of `a` and `b` (in any order). Now, in OCaml, we define a function deleting all the occurrences of an element in a list. Here is a first (bugged) version of this function:

```
let rec delete x l= match l with
  | [] -> []
  | h::t -> if h=x then t else h::(delete x t);;
```

Of course, one can perform tests on this function using the usual OCaml interpreter:

[2] Note that the objective of other papers like [4, 21] is different. They aim at predicting the control flow of a program rather than estimating the possible results of a function (data flow).

```
# delete 2 [1;2;3]);;
-:int list= [1; 3]
```

With an *abstract* OCaml interpreter dealing with grammars, we could ask the following question: what is the set of the results obtained by applying `delete` to `a` and to any list of `a` and `b`?

```
# delete a [(a|b)*];;
-:abst list= [(a|b)*]
```

The obtained result is not the expected one. Since all occurrences of `a` should have been removed, we expected the result `[b*]`. Since the abstract interpreter results into a grammar *over-approximating* the set of outputs, this does not *show* that there is a bug, it only suggests it (like for type inference). Indeed, in the definition of `delete` there is a missing recursive call in the `then` branch. If we correct this mistake, we get:

```
# delete a [(a|b)*];;
-:abst list= [b*]
```

This result proves that `delete` deletes all occurrences of an element in a list. This is only one of the expected properties of `delete`, but shown automatically and without complex formalisation. Here is, in Timbuk syntax, the TRS R and tree automata that are given to Timbuk to achieve the above proof.

```
Ops delete:2 cons:2 nil:0 a:0 b:0 ite:3 true:0 false:0 eq:2
Vars X Y Z
TRS R
eq(a,a)->true        eq(a,b)->false     eq(b,a)->false     eq(b,b)->true
delete(X,nil)->nil   ite(true,X,Y)->X   ite(false,X,Y)->Y
delete(X,cons(Y,Z))->ite(eq(X,Y),delete(X,Z),cons(Y,delete(X,Z)))
Automaton A0  States qf qa qb qlb qlab qnil  Final States qf
Transitions  delete(qa,qlab)->qf  a->qa  b->qb  nil->qlab
cons(qa,qlab)->qlab    cons(qb,qlab)->qlab
```

The resulting automaton computed by Timbuk is the following. It is not minimal but its recognised language is equivalent to `[b*]`.

```
States q0 q6 q8 Final States q6
Transitions cons(q8,q0)->q0  nil->q0  b->q8  cons(q8,q0)->q6  nil->q6
```

1.2 What is the problem with evaluation strategies?

Let us consider the function `sum(x)` which computes the sum of the `x` first natural numbers.

```
let rec sumList x y=                  let rec nth i (x::l)=
  (x+y)::(sumList (x+y) (y+1))           if i<=0 then x else nth (i-1) l
let sum x= nth x (sumList 0 0)
```

This function is terminating with call-by-need (used in Haskell) but not with call-by-value strategy (used in OCaml). Hence, any call to `sum` for any number `i` will not terminate because of OCaml's evaluation strategy. Thus the result of the abstract interpreter on `sum s*(0)` (*i.e.* `sum` applied to any natural number `0`, `s(0)`, ...) should be an empty grammar meaning that there is an empty set of results. However, if we use any of the techniques mentioned in the introduction to infer the output grammar, it will fail to show this. All those techniques compute reachable term grammars that do not take evaluation strategy into account. In particular, the inferred grammars will also contain all call-by-need evaluations. Thus, an abstract interpreter built on those techniques will produce a result of the form `s*(0)`, which is a very rough approximation. In this paper, we propose to improve

the accuracy of such approximations by defining a language inference technique taking the call-by-value evaluation strategy into account.

1.3 Computing over-approximations of innermost reachable terms

Call-by-value evaluation strategy of functional programs is strongly related to innermost rewriting. The problem we are interested in is thus to compute (or to over-approximate) the set of innermost reachable terms. For a TRS R and a set of terms $L_0 \subseteq T(\Sigma)$, the set of reachable terms is $R^*(L_0) = \{t \in T(\Sigma) \mid \exists s \in L_0, s \to_R^* t\}$. This set can be computed for specific classes of R but, in general, it has to be approximated. Most of the techniques compute such approximations using tree automata (and not grammars) as the core formalism to represent or approximate the (possibly) infinite set of terms $R^*(L_0)$. Most of them also rely on a Knuth-Bendix completion-like algorithm to produce an automaton $\mathcal{A}^*$ recognising exactly, or over-approximating, the set of reachable terms. As a result, these techniques can be referred to as *tree automata completion* techniques [10, 27, 3, 14, 23].

Surprisingly, very little effort has been paid to computing or over-approximating the set $R^*_{strat}(L_0)$, *i.e.* set of reachable terms when R is applied with a strategy *strat*. To the best of our knowledge, Pierre Réty and Julie Vuotto's work [26] is the first one to have tackled this goal. They give some sufficient conditions on L_0 and R for $R^*_{strat}(L_0)$ to be recognised by a tree automaton $\mathcal{A}^*$, where *strat* can be the innermost or the outermost strategy. Innermost reachability for shallow TRSs was studied in [9]. However, in both cases, the restrictions on R are strong and generally incompatible with functional programs seen as TRS. Moreover, the proposed techniques are not able to over-approximate reachable terms when the TRSs does not satisfy the restrictions.

In this paper, we concentrate on the innermost strategy and define a tree automata completion algorithm over-approximating the set $R^*_{\text{in}}(L_0)$ (innermost reachable terms) for any left-linear TRS R and any regular set of input terms L_0. As the completion algorithm of [14], it is parameterized by a set of term equations E defining the precision of the approximation. We prove the soundness of the algorithm: for all set of equation E, if completion terminates then the resulting automaton $\mathcal{A}^*$ recognises an over-approximation of $R^*_{\text{in}}(L_0)$. Then, we prove a precision theorem: $\mathcal{A}^*$ recognises no more terms than terms reachable by innermost rewriting with R modulo equations of E. Finally, we show on examples that the precision of innermost completion noticeably improves the accuracy of the static analysis of functional programs.

This paper is organised as follows. Section 2 recalls some basic notions about TRSs and tree automata.

Section 3 exposes innermost completion. Section 4 states and proves the soundness of this method. Section 5 states the precision theorem. Section 6 demonstrates how our new technique can effectively give more precise results on functional programs thanks to the tool TimbukSTRAT, an implementation of our method in the Timbuk reachability tool [13].

2 Preliminaries

We use the same basic definitions and notions as in [1] and [28] for TRS and as in [6] for tree automata.

For a set of functions Σ and a set of variables $\mathcal{X}$, we denote signatures by $(\Sigma, \mathcal{X})$, $T(\Sigma, \mathcal{X})$ for the set of terms and $T(\Sigma)$ for the set of ground terms over $(\Sigma, \mathcal{X})$. Given a signature Σ and $k \in \mathbb{N}$, the set of its function symbols of arity k is denoted by Σ_k.

▶ **Definition 1** (Rewriting rule, term rewriting system)**.** A rewriting rule over $(\Sigma, \mathcal{X})$ is a couple $(\ell, r) \in T(\Sigma, \mathcal{X}) \times T(\Sigma, \mathcal{X})$, denoted by $\ell \to r$, such that ℓ is not a variable and any variable appearing in r also appears in ℓ. A term rewriting system (TRS) over $(\Sigma, \mathcal{X})$ is a set of rewriting rules over $(\Sigma, \mathcal{X})$.

The set of normal forms of a rewriting system R (*i.e.* terms that are not reducible by R) is $\textsc{Irr}(R)$. A term t is linear when no variable appears twice in t; a TRS is left-linear if the lhs of each of its rules is linear.

▶ **Definition 2** (Set of reachable terms)**.** Given a signature $(\Sigma, \mathcal{X})$, a TRS R over it and a set of terms $L \subseteq T(\Sigma)$, we denote $R(L) = \{t \in T(\Sigma) \mid \exists s \in L, s \to_R t\}$ and $R^*(L) = \{t \in T(\Sigma) \mid \exists s \in L, s \to_R^* t\}$.

2.1 Equations

▶ **Definition 3** (Equivalence relation, congruence)**.** A binary relation is an equivalence relation if it is reflexive, symmetric and transitive.

An equivalence relation $\equiv$ over $T(\Sigma)$ is a congruence if for all $k \in \mathbb{N}$, for all $f \in \Sigma_k$, for all $t_1, \ldots, t_k, s_1, \ldots, s_k \in T(\Sigma)$ such that $\forall i = 1 \ldots k$, $t_i \equiv s_i$, we have $f(t_1, \ldots, t_k) \equiv f(s_1, \ldots, s_k)$.

▶ **Definition 4** (Equation, $\equiv_E$)**.** An equation over $(\Sigma, \mathcal{X})$ is a pair of terms $(s, t) \in T(\Sigma, \mathcal{X}) \times T(\Sigma, \mathcal{X})$, denoted by $s = t$. A set E of equations over $(\Sigma, \mathcal{X})$ induces a congruence $\equiv_E$ over $T(\Sigma)$ which is the smallest congruence over $T(\Sigma)$ such that for all $s = t \in E$ and for all substitution $\theta : \mathcal{X} \to T(\Sigma)$, $s\theta \equiv_E t\theta$. The equivalence classes of $\equiv_E$ are denoted with $[\cdot]_E$.

▶ **Definition 5** (Rewriting modulo E)**.** Given a TRS R and a set of equations E both over $(\Sigma, \mathcal{X})$, we define the R modulo E rewriting relation, $\to_{R/E}$, as follows. For any $u, v \in T(\Sigma)$, $u \to_{R/E} v$ if and only if there exist $u', v' \in T(\Sigma)$ such that $u' \equiv_E u$, $v' \equiv_E v$ and $u' \to_R v'$. We define $\to_{R/E}^*$ as the reflexive and transitive closure of $\to_{R/E}$ and $(R/E)(L)$ and $(R/E)^*(L)$ in the same way as $R(L)$ and $R^*(L)$ where $\to_{R/E}$ replaces $\to_R$.

2.2 Tree automata

▶ **Definition 6** (Tree automaton, delta-transition, epsilon-transition, new state)**.** An automaton over Σ is some $\mathcal{A} = (\Sigma, Q, Q_F, \Delta)$ where Q is a finite set of states (symbols of arity 0 such that $\Sigma \cap Q = \varnothing$), Q_F is a subset of Q whose elements are called final states and Δ a finite set of transitions. A delta-transition is of the form $f(q_1, \ldots, q_k) \rightarrowtail q'$ where $f \in \Sigma_k$ and $q_1, \ldots, q_k, q' \in Q$. An epsilon-transition is of the form $q \rightarrowtail q'$ where $q, q' \in Q$. A configuration of $\mathcal{A}$ is a term in $T(\Sigma, Q)$.

A state $q \in Q$ that appears nowhere in Δ is called a new state. A configuration is elementary if each of its sub-configurations at depth 1 (if any) is a state.

▶ **Definition 7.** Let $\mathcal{A} = (\Sigma, Q, Q_F, \Delta)$ be an automaton and let c, c' be configurations of $\mathcal{A}$. We say that $\mathcal{A}$ recognises c into c' in one step, and denoted by $c \underset{\mathcal{A}}{\rightarrowtail} c'$ if there a transition $\tau \rightarrowtail \rho$ in $\mathcal{A}$ and a context C over $T(\Sigma, Q)$ such that $c = C[\tau]$ and $c' = C[\rho]$. We denote by $\underset{\mathcal{A}}{\overset{*}{\rightarrowtail}}$ the reflexive and transitive closure of $\underset{\mathcal{A}}{\rightarrowtail}$ and, for any $q \in Q$, $\mathscr{L}(\mathcal{A}, q) = \left\{ t \in T(\Sigma) \;\middle|\; t \underset{\mathcal{A}}{\overset{*}{\rightarrowtail}} q \right\}$. We extend this definition to subsets of Q and denote it by $\mathscr{L}(\mathcal{A}) = \mathscr{L}(\mathcal{A}, Q_F)$. A sequence of configurations $c_1, \ldots, c_n$ such that $t \underset{\mathcal{A}}{\rightarrowtail} c_1 \underset{\mathcal{A}}{\rightarrowtail} \cdots \underset{\mathcal{A}}{\rightarrowtail} c_n \underset{\mathcal{A}}{\rightarrowtail} q$ is called a recognition path

for t (into q) in $\mathcal{A}$. When $q \underset{\mathcal{A}}{\rightarrowtail} q'$ and $q' \underset{\mathcal{A}}{\rightarrowtail} q$, this is denoted by $q \underset{\mathcal{A}}{\rightleftarrows} q'$. A state q of $\mathcal{A}$ is accessible if $\mathscr{L}(\mathcal{A}, q) \neq \varnothing$. An automaton is accessible if all of its states are.

▶ **Example 8.** Let Σ be defined with $\Sigma_0 = \{n, 0\}$, $\Sigma_1 = \{s, a, f\}$, $\Sigma_2 = \{c\}$ where 0 is meant to represent integer zero, s the successor operation on integers, a the predecessor ('antecessor') operation, n the empty list, c the constructor of lists of integers and f is intended to be the function on lists that filters out integer zero. Let $R = \{f(n) \to n, f(c(s(X), Y)) \to c(s(X), f(Y)), f(c(a(X), Y)) \to c(a(X), f(Y)), f(c(0, Y)) \to f(Y), a(s(X)) \to X, s(a(X)) \to X\}$. Let $\mathcal{A}_0$ be the tree automaton with final state q_f and transitions $\{n \rightarrowtail q_n, 0 \rightarrowtail q_0, s(q_0) \rightarrowtail q_s, a(q_s) \rightarrowtail q_a, c(q_a, q_n) \rightarrowtail q_c, f(q_c) \rightarrowtail q_f\}$. We have $\mathscr{L}(\mathcal{A}_0, q_f) = \{f(c(a(s(0)), n))\}$ and $R(\mathscr{L}(\mathcal{A}_0, q_f)) = \{f(c(0, n)), c(a(s(0)), f(n))\}$.

▶ Remark. Automata transitions may have 'colours', like $\mathfrak{R}$ for transition $q \overset{\mathfrak{R}}{\rightarrowtail} q'$. We will use colours $\mathfrak{R}$ and $\mathfrak{E}$ for transitions denoting either rewrite or equational steps.

▶ **Definition 9.** Given an automaton $\mathcal{A}$ and a colour $\mathfrak{R}$, we denote by $\mathcal{A}^{\not\mathfrak{R}}$ the automaton obtained from $\mathcal{A}$ by removing all transitions coloured with $\mathfrak{R}$.

2.3 Pair automaton

We now give notations used for pair automaton, the archetype of which is the product of two automata.

▶ **Definition 10** (Pair automaton)**.** An automaton $\mathcal{A} = (\Sigma, Q, Q_F, \Delta)$ is said to be a pair automaton if there exists some sets Q_1 and Q_2 such that $Q = Q_1 \times Q_2$.

▶ **Definition 11** (Product automaton [6])**.** Let $\mathcal{A} = (\Sigma, Q, Q_F, \Delta_{\mathcal{A}})$ and $\mathcal{B} = (\Sigma, P, P_F, \Delta_{\mathcal{B}})$ be two automata. The product automaton of $\mathcal{A}$ and $\mathcal{B}$ is $\mathcal{A} \times \mathcal{B} = (\Sigma, Q \times P, Q_F \times P_F, \Delta)$ where $\Delta = \{f(\langle q_1, p_1 \rangle, \ldots, \langle q_k, p_k \rangle) \rightarrowtail \langle q', p' \rangle \mid f(q_1, \ldots, q_k) \rightarrowtail q' \in \Delta_{\mathcal{A}} \wedge f(p_1, \ldots, p_k) \rightarrowtail p' \in \Delta_{\mathcal{B}}\} \cup \{\langle q, p \rangle \rightarrowtail \langle q', p \rangle \mid p \in P,\ q \rightarrowtail q' \in \Delta_{\mathcal{A}}\} \cup \{\langle q, p \rangle \rightarrowtail \langle q, p' \rangle \mid q \in Q,\ p \rightarrowtail p' \in \Delta_{\mathcal{B}}\}$

▶ **Definition 12** (Projections)**.** Let $\mathcal{A} = (\Sigma, Q, Q_F, \Delta)$ be a pair automaton, let $\tau \rightarrowtail \rho$ be one of its transitions and $\langle q, p \rangle$ be one of its states. We define $\Pi_1(\langle q, p \rangle) = q$ and extend $\Pi_1(\cdot)$ to configurations inductively: $\Pi_1(f(\gamma_1, \ldots, \gamma_k)) = f(\Pi_1(\gamma_1), \ldots, \Pi_1(\gamma_k))$. We define $\Pi_1(\tau \rightarrowtail \rho) = \Pi_1(\tau) \rightarrowtail \Pi_1(\rho)$. We define $\Pi_1(\mathcal{A}) = (\Sigma, \Pi_1(Q), \Pi_1(Q_F), \Pi_1(\Delta))$. $\Pi_2(\cdot)$ is defined on all these objects in the same way for the right component.

▶ Remark. Using $\Pi_1(\mathcal{A})$ amounts to forgetting the precision given by the right component of the states. As a result, $\mathscr{L}(\Pi_1(\mathcal{A}), q) \supseteq \bigcup_{p \in P} \mathscr{L}(\mathcal{A}, \langle q, p \rangle)$.

2.4 Innermost strategy

In general, a strategy over a TRS R is a set of (computable) criteria to describe a certain sub-relation of $\to_R$. In this paper, we will be interested in innermost strategies. In these strategies, commonly used to execute functional programs ('call-by-value'), terms are rewritten by always contracting one of the lowest reducible subterms. If $s \to_R t$ and rewriting occurs at a position p of s, $s|_p$ is called the *redex*.

▶ **Definition 13** (Innermost strategy)**.** Given a TRS R and two terms s, t, we say that s can be rewritten into t by R with an innermost strategy, denoted by $s \to_{R_{\text{in}}} t$, if $s \to_R t$ and each strict subterm of the redex in s is a R-normal form. We define $R_{\text{in}}(L)$ and $R^*_{\text{in}}(L)$ in the same way as $R(L)$, $R^*(L)$ where $\to_{R_{\text{in}}}$ replaces $\to_R$.

▶ **Example 14.** We continue on Example 8. We have $R_{\text{in}}(\mathscr{L}(\mathcal{A}_0, q_f)) = \{f(c(0, n))\}$ because the rewriting step $f(c(a(s(0)), n)) \rightarrow_R c(a(s(0)), f(n))$ is not innermost since the subterm $a(s(0))$ of the redex $f(c(a(s(0)), n))$ is not in normal form.

To deal with innermost strategies, we have to discriminate normal forms. When R is left-linear, it is possible to compute a tree automaton recognising normal forms.

▶ **Theorem 15** ([7]). *Let R be a left-linear TRS. There is a deterministic and complete tree automaton $\mathcal{A}_{\mathcal{IRR}}(R)$ whose states are all final except one, denoted by p_{red} and such that $\mathscr{L}(\mathcal{A}_{\mathcal{IRR}}(R)) = \text{Irr}(R)$ and $\mathscr{L}(\mathcal{A}_{\mathcal{IRR}}(R), p_{\text{red}}) = T(\Sigma) \smallsetminus \text{Irr}(R)$.*

▶ Remark. Since $\mathcal{A}_{\mathcal{IRR}}(R)$ is deterministic, for any state $p \neq p_{\text{red}}$, $\mathscr{L}(\mathcal{A}_{\mathcal{IRR}}(R), p) \subseteq \text{Irr}(R)$.

▶ Remark. If a term s is reducible, any term having s as a subterm is also reducible. Thus any transition of $\mathcal{A}_{\mathcal{IRR}}(R)$ where p_{red} appears in the left-hand side will necessarily have p_{red} as its right-hand side. Thus, for brevity, these transitions will always be left implicit when describing the automaton $\mathcal{A}_{\mathcal{IRR}}(R)$ for some TRS R.

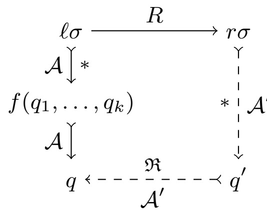
▶ **Example 16.** In Example 8, $\mathcal{A}_{\mathcal{IRR}}(R)$ needs, in addition to p_{red}, a state p_{list} to recognise lists of integers, a state p_a for terms of the form $a(\dots)$, a state p_s for $s(\dots)$, a state p_0 for 0 and a state p_{var} to recognise terms that are not subterms of lhs of R, but may participate in building a reducible term by being instances of variables in a lhs. We note $P = \{p_{list}, p_0, p_a, p_s, p_{var}\}$ and $P_{int} = \{p_0, p_a, p_s\}$. The interesting transitions are thus $0 \rightarrowtail p_0$, $\bigcup_{p \in P \smallsetminus \{p_a\}} \{s(p) \rightarrowtail p_s\}$, $\bigcup_{p \in P \smallsetminus \{p_s\}} \{a(p) \rightarrowtail p_a\}$; $n \rightarrowtail p_{list}$, $\bigcup_{p \in P_{int}, p' \in P} \{c(p, p') \rightarrowtail p_{list}\}$; $f(p_{list}) \rightarrowtail p_{\text{red}}$, $a(p_s) \rightarrowtail p_{\text{red}}$, $s(p_a) \rightarrowtail p_{\text{red}}$. Furthermore, as remarked above, any configuration that contains a p_{red} is recognised into p_{red}. Finally, some configurations are not covered by the previous cases: they are recognised into p_{var}.

3 Innermost equational completion

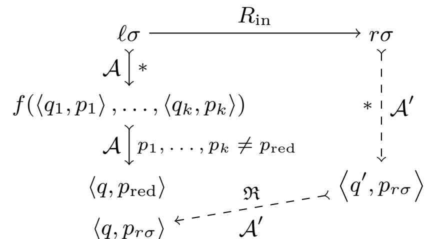
Our first contribution is an adaptation of the classical equational completion of [14], which is an iterative process on automata. Starting from a tree automaton $\mathcal{A}_0$ it iteratively computes tree automata $\mathcal{A}_1, \mathcal{A}_2, \dots$ until a fixpoint automaton $\mathcal{A}_*$ is found. Each iteration comprises two parts: (exact) completion itself (Subsection 3.1), then equational merging (Subsection 3.2). The former tends to incorporate descendants by R of already recognised terms into the recognised language; this leads to the creation of new states. The latter tends to merge states in order to ease termination of the overall process, at the cost of precision of the computed result. Some transition added by equational completion will have colours $\mathfrak{R}$ or $\mathfrak{E}$. We will use colours $\mathfrak{R}$ and $\mathfrak{E}$ for transitions denoting either rewrite or equational steps; it is assumed that the transitions of the input automaton $\mathcal{A}_0$ do not have any colour and that $\mathcal{A}_0$ does not have any epsilon-transition.

The equational completion of [14] is blind to strategies. To make it innermost-strategy-aware, we equip each state of the studied automata with a state from the automaton $\mathcal{A}_{\mathcal{IRR}}(R)$ (see Theorem 15) to keep track of normal and reducible forms. Let $\mathcal{A}_{init}$ be an automaton recognising the initial language. Completion will start with $\mathcal{A}_0 = \mathcal{A}_{init} \times \mathcal{A}_{\mathcal{IRR}}(R)$. This automaton enjoys the following property.

▶ **Definition 17** (Consistency with $\mathcal{A}_{\mathcal{IRR}}(R)$). A pair automaton $\mathcal{A}$ is said to be consistent with $\mathcal{A}_{\mathcal{IRR}}(R)$ if, for any configuration c and any state $\langle q, p\rangle$ of $\mathcal{A}$, $\Pi_2(c)$ is a configuration of $\mathcal{A}_{\mathcal{IRR}}(R)$ and p is a state of $\mathcal{A}_{\mathcal{IRR}}(R)$, and if $c \overset{*}{\underset{\mathcal{A}}{\rightarrowtail}} \langle q, p\rangle$ then $\Pi_2(c) \overset{*}{\underset{\mathcal{A}_{\mathcal{IRR}}(R)}{\rightarrowtail}} p$.



(a) For comparison: Classical completion.



(b) Innermost completion: we added information about normal forms.

Figure 1 Comparison of classical and innermost critical pairs.

3.1 Exact completion

The first step of equational completion incorporates descendants by R of terms recognised by $\mathcal{A}_i$ into $\mathcal{A}_{i+1}$. The principle is to search for critical pairs between $\mathcal{A}_i$ and R. In classical completion, a critical pair is triple $(\ell \to r, \sigma, q)$ such that $l\sigma \xrightarrow[\mathcal{A}_i]{*} q$, $l\sigma \to_R r\sigma$ and $r\sigma \not\xrightarrow[\mathcal{A}_i]{*} q$. Such a critical pair denotes a rewriting position of a term recognised by $\mathcal{A}_i$ such that the rewritten term is not recognised by $\mathcal{A}_i$. For the innermost strategy, the critical pair notion is slightly refined since it also needs that every subterm γ at depth 1 in $\ell\sigma$ is in normal form. This corresponds to the third case of the following definition where $\gamma \xrightarrow[\mathcal{A}]{*} \langle q_\gamma, p_\gamma \rangle$ and $p_\gamma \neq p_{red}$ ensures that γ is irreducible. See Figure 1.

▶ **Definition 18** (Innermost critical pair)**.** Let $\mathcal{A}$ be a pair automaton. A tuple $(\ell \to r, \sigma, \langle q, p \rangle)$ where $\ell \to r \in R$, $\sigma : \mathcal{X} \to Q_\mathcal{A}$ and $\langle q, p \rangle \in Q_\mathcal{A}$ is called a critical pair if

1. $\ell\sigma \xrightarrow[\mathcal{A}]{*} \langle q, p \rangle$,
2. there is no p' such that $r\sigma \xrightarrow[\mathcal{A}]{*} \langle q, p' \rangle$ and
3. for each sub-configuration γ at depth 1 of $\ell\sigma$, the state $\langle q_\gamma, p_\gamma \rangle$ such that $\gamma \xrightarrow[\mathcal{A}]{*} \langle q_\gamma, p_\gamma \rangle$ in the recognition path of condition 1 is with $p_\gamma \neq p_{\mathrm{red}}$.

▶ Remark. Because a critical pair denotes a rewriting situation, the p of Definition 18 is necessarily p_{red} as long as $\mathcal{A}$ is consistent with $\mathcal{A}_{\mathcal{IRR}}(R)$.

▶ **Example 19.** In the situation of Examples 8 and 16, consider the rule $f(c(a(X), Y)) \to c(a(X), f(Y))$, the substitution $\sigma_1 = \{X \mapsto \langle q_s, p_s \rangle, Y \mapsto \langle q_n, p_n \rangle\}$ and the state $\langle q_f, p_{\mathrm{red}} \rangle$: this is not an innermost critical pair because the recognition path is:

$$f(c(a(\langle q_s, p_s \rangle), \langle q_n, p_n \rangle)) \rightarrowtail f(c(\langle q_a, p_{\mathrm{red}} \rangle, \langle q_n, p_n \rangle)) \rightarrowtail f(\langle q_c, p_{\mathrm{red}} \rangle) \rightarrowtail \langle q_f, p_{\mathrm{red}} \rangle$$

and so there is a p_{red} at depth 1. But there is an innermost critical pair in $\mathcal{A}_0$ with the rule $a(s(X)) \to X$, the substitution $\sigma_2 = \{X \mapsto \langle q_0, p_0 \rangle\}$ and the state $\langle q_a, p_{\mathrm{red}} \rangle$.

Once a critical pair is found, the completion algorithm needs to resolve it: it adds the necessary transitions for $r\sigma$ to be recognised by the completed automaton. Classical completion adds the necessary transitions so that $r\sigma \xrightarrow[\mathcal{A}']{*} q$, where $\mathcal{A}'$ is the completed automaton. In innermost completion this is more complex. The state q is, in fact, a pair of the form $\langle q, p_{red} \rangle$ and adding transitions so that $r\sigma \xrightarrow[\mathcal{A}']{*} \langle q, p_{red} \rangle$ may jeopardise consistency of $\mathcal{A}'$ with $\mathcal{A}_{\mathcal{IRR}}$ if $r\sigma$ is not reducible. Thus the diagram is closed in a different way preserving consistency with $\mathcal{A}_{\mathcal{IRR}}$ (see Figure 1). However, like in classical completion, this

can generally not be done in one step, as $r\sigma$ might be a non-elementary configuration. We have to split the configuration into elementary configurations and to introduce new states to recognise them: this is what *normalisation* (denoted by $Norm_{\mathcal{A}}$) does. Given an automaton $\mathcal{A}$, a configuration c of $\mathcal{A}$ and a new state $\langle q, p\rangle$, we denote by $Norm_{\mathcal{A}}(c, \langle q, p\rangle)$ the set of transitions (with new states) that we add to $\mathcal{A}$ to ensure that c is recognised into $\langle q, p\rangle$. The $Norm_{\mathcal{A}}$ operation is parameterized by $\mathcal{A}$ because it reuses transitions of $\mathcal{A}$ whenever it is possible. On an example, we show how normalisation behaves. For a formal definition see [15].

▶ **Example 20.** With a suitable signature, suppose that automaton $\mathcal{A}$ consists of the transitions $c \rightarrowtail \langle q_1, p_c\rangle$ and $f(\langle q_1, p_c\rangle) \rightarrowtail \langle q_2, p_{f(c)}\rangle$ and we want to normalise $f(g(\langle q_2, p_{f(c)}\rangle, c))$ to the new state $\langle q_N, p_{f(g(f(c),c))}\rangle$. We first have to normalise under g: $\langle q_2, p_{f(c)}\rangle$ is already a state, so it does not need to be normalised; c has to be normalised to a state: since $\mathcal{A}$ already has transition $c \rightarrowtail \langle q_1, p_c\rangle$, we add no new state and it remains to normalise $g(\langle q_2, p_{f(c)}\rangle, \langle q_1, p_c\rangle)$. Since $\mathcal{A}$ does not contain a transition for this configuration, we must add a new state $\langle q', p_{g(f(c),c)}\rangle$ and the transition $g(\langle q_2, p_{f(c)}\rangle, \langle q_1, p_c\rangle) \rightarrowtail \langle q', p_{g(f(c),c)}\rangle$. Finally, we add $f(\langle q', p_{g(f(c),c)}\rangle) \rightarrowtail \langle q_N, p_{f(g(f(c),c))}\rangle$. Note that due to consistency with $\mathcal{A}_{\mathcal{IRR}}(R)$, whenever we add a new transition $c' \rightarrowtail \langle q', p'\rangle$, only the q' is arbitrary: the p' is always the state of $\mathcal{A}_{\mathcal{IRR}}(R)$ such that $\Pi_2(c) \underset{\mathcal{A}_{\mathcal{IRR}}(R)}{\rightarrowtail} p'$, in order to preserve consistency with $\mathcal{A}_{\mathcal{IRR}}(R)$.

Completion of a critical pair is done in two steps. The first set of operations formalises 'closing the square' (see Figure 1), *i.e.* if $l\sigma \overset{*}{\underset{\mathcal{A}}{\rightarrowtail}} \langle q, p_{red}\rangle$ then we add transitions $r\sigma \overset{*}{\underset{\mathcal{A}'}{\rightarrowtail}} \langle q', p_{r\sigma}\rangle \overset{\mathfrak{R}}{\rightarrowtail} \langle q, p_{r\sigma}\rangle$. The second step adds the necessary transitions for any context $C[r\sigma]$ to be recognised in the tree automaton if $C[l\sigma]$ was. Thus if the the recognition path for $C[l\sigma]$ is of the form $C[l\sigma] \overset{*}{\underset{\mathcal{A}}{\rightarrowtail}} C[\langle q, p_{red}\rangle] \overset{*}{\underset{\mathcal{A}}{\rightarrowtail}} \langle q_c, p_{red}\rangle$, we add the necessary transitions for $C[\langle q, p_{r\sigma}\rangle]$ to be recognised into $\langle q_c, p_c\rangle$ where p_c is the state of $\mathcal{A}_{\mathcal{IRR}}(R)$ recognising $C[r\sigma]$.

▶ **Definition 21** (Completion of an innermost critical pair)**.** A critical pair $(\ell \to r, \sigma, \langle q, p\rangle)$ in automaton $\mathcal{A}$ is completed by first computing $N = Norm_{\mathcal{A}\not{\mathcal{R}}}(r\sigma, \langle q', p_{r\sigma}\rangle)$ where q' is a new state and $\Pi_2(r\sigma) \overset{*}{\underset{\mathcal{A}_{\mathcal{IRR}}(R)}{\rightarrowtail}} p_{r\sigma}$, then adding to $\mathcal{A}$ the new states and the transitions appearing in N as well as the transition $\langle q', p_{r\sigma}\rangle \overset{\mathfrak{R}}{\rightarrowtail} \langle q, p_{r\sigma}\rangle$. If $r\sigma$ is a trivial configuration (*i.e.* r is just a variable, and thus $\Pi_2(r\sigma)$ is a state), only transition $r\sigma \overset{\mathfrak{R}}{\rightarrowtail} \langle q, \Pi_2(r\sigma)\rangle$ is added. Afterwards, we execute the following supplementary operations. For any new transition $f(\dots, \langle q, p_{\text{red}}\rangle, \dots) \rightarrowtail \langle q'', p''\rangle$, we add a transition $f(\dots, \langle q, p_{r\sigma}\rangle, \dots) \rightarrowtail \langle q'', p'''\rangle$ with $f(\dots, p_{r\sigma}, \dots) \underset{\mathcal{A}_{\mathcal{IRR}}(R)}{\rightarrowtail} p'''$. These new transitions are in turn recursively considered for the supplementary operations[3].

▶ **Definition 22** (Innermost completion step)**.** Let PC be the set of all innermost critical pairs of $\mathcal{A}_i$. For $pc \in PC$, let N_{pc} be the set of new states and transitions needed under Definition 21 to complete pc, and $\mathcal{A} \cup N_{pc}$ the automaton $\mathcal{A}$ completed by states and transitions of N_{pc}. Then $\mathcal{A}_{i+1} = \mathcal{A}_i \cup \bigcup_{pc \in PC} N_{pc}$.

[3] Those supplementary operations add new pairs, but the element of each pair are not new. So, this necessarily terminates.

▶ **Lemma 23.** *Let $\mathcal{A}$ be an automaton obtained from some $\mathcal{A}_{init} \times \mathcal{A}_{\mathcal{IRR}}(R)$ after some steps of innermost completion. $\mathcal{A}$ is consistent with $\mathcal{A}_{\mathcal{IRR}}(R)$.*

Due to space constraints, the full proofs can be found in [15].

3.2 Equational simplification

▶ **Definition 24.** Given two states q, q' of some automaton $\mathcal{A}$ and a colour $\mathfrak{E}$, we note $q \overset{\mathfrak{E}}{\underset{\mathcal{A}}{\rightleftarrows}} q'$ when we have both $q \overset{\mathfrak{E}}{\underset{\mathcal{A}}{\rightarrowtail}} q'$ and $q' \overset{\mathfrak{E}}{\underset{\mathcal{A}}{\rightarrowtail}} q$.

▶ **Definition 25** (Situation of application of an equation)**.** Given an equation $s = t$, an automaton $\mathcal{A}$, a substitution $\theta : \mathcal{X} \to Q_{\mathcal{A}}$ and states $\langle q_1, p_1\rangle$ and $\langle q_2, p_2\rangle$, we say that $(s = t, \theta, \langle q_1, p_1\rangle, \langle q_2, p_2\rangle)$ is a situation of application in $\mathcal{A}$ if

1. $s\theta \overset{*}{\underset{\mathcal{A}}{\rightarrowtail}} \langle q_1, p_1\rangle$,
2. $t\theta \overset{*}{\underset{\mathcal{A}}{\rightarrowtail}} \langle q_2, p_2\rangle$,
3. $\langle q_1, p_1\rangle \not\overset{\mathfrak{E}}{\underset{\mathcal{A}}{\rightleftarrows}} \langle q_2, p_2\rangle$
4. $p_1 = p_2$.

Note that when $p_1 \neq p_2$, this is not a situation of application for an equation. This is the only difference with the situation of application in classical completion. This restriction avoids, in particular, to apply an equation between reducible and irreducible terms. Such terms will be recognised by states having two distinct second components. On the opposite, when a situation of application arises, we 'apply' the equation, *i.e.* add the necessary transitions to have $\langle q_1, p_1\rangle \overset{\mathfrak{E}}{\underset{\mathcal{A}}{\rightleftarrows}} \langle q_2, p_2\rangle$ and supplementary transitions to lift this property to any embedding context. We apply equations until there are no more situation of application on the automaton (this is guaranteed to happen because we add no new state in this part).

▶ **Definition 26** (Application of an equation)**.** Given $(s = t, \theta, \langle q_1, p_1\rangle, \langle q_2, p_1\rangle)$ a situation of application in $\mathcal{A}$, applying the underlying equation in it consists in adding transitions $\langle q_1, p_1\rangle \overset{\mathfrak{E}}{\rightarrowtail} \langle q_2, p_1\rangle$ and $\langle q_2, p_1\rangle \overset{\mathfrak{E}}{\rightarrowtail} \langle q_1, p_1\rangle$ to $\mathcal{A}$. We also add the supplementary transitions $\langle q_1, p_1'\rangle \overset{\mathfrak{E}}{\rightarrowtail} \langle q_2, p_1'\rangle$ and $\langle q_2, p_1'\rangle \overset{\mathfrak{E}}{\rightarrowtail} \langle q_1, p_1'\rangle$ where $\langle q_1, p_1'\rangle$ and $\langle q_2, p_1'\rangle$ occur in the automaton.

▶ **Lemma 27.** *Applying an equation preserves consistency with $\mathcal{A}_{\mathcal{IRR}}(R)$.*

3.3 Innermost completion and equations

▶ **Definition 28** (Step of innermost equational completion)**.** Let R be a left-linear TRS, $\mathcal{A}_{init}$ a tree automaton, E a set of equations and $\mathcal{A}_0 = \mathcal{A}_{init} \times \mathcal{A}_{\mathcal{IRR}}(R)$. The automaton $\mathcal{A}_{i+1}$ is obtained, from $\mathcal{A}_i$, by applying an innermost completion step on $\mathcal{A}_i$ (Definition 21) and solving all situations of applications of equations of E (Definition 25).

4 Correctness

▶ **Definition 29** (Correct automaton)**.** An automaton $\mathcal{A}$ is correct w.r.t. R_{in} if for all states $\langle q, p_{\text{red}}\rangle$ of $\mathcal{A}$, for all $u \in \mathscr{L}(\mathcal{A}, \langle q, p_{\text{red}}\rangle)$ and for all $v \in R_{\text{in}}(u)$, either there is a state p of $\mathcal{A}_{\mathcal{IRR}}(R)$ such that $v \in \mathscr{L}(\mathcal{A}, \langle q, p\rangle)$ or there is a critical pair $(\ell \to r, \sigma, \langle q_0, p_0\rangle)$ in $\mathcal{A}$ for some $\langle q_0, p_0\rangle$ and a context C on $T(\Sigma)$ such that $u \overset{*}{\underset{\mathcal{A}}{\rightarrowtail}} C[\ell\sigma] \overset{*}{\underset{\mathcal{A}}{\rightarrowtail}} C[\langle q_0, p_{\text{red}}\rangle] \overset{*}{\underset{\mathcal{A}}{\rightarrowtail}} \langle q, p_{\text{red}}\rangle$ and $v \overset{*}{\underset{\mathcal{A}}{\rightarrowtail}} C[r\sigma]$.

▶ **Lemma 30.** *Any automaton produced by innermost equational completion starting from some $\mathcal{A}_{init} \times \mathcal{A}_{\mathcal{IRR}}(R)$ is correct w.r.t. R_{in}.*

▶ **Theorem 31** (Correctness). *Assuming R is left-linear, the innermost equational completion procedure defined above produces a correct result whenever it terminates and produces some fixpoint $\mathcal{A}_{\text{in}*}$:*

$$\mathscr{L}(\mathcal{A}_{\text{in}*}) \supseteq R^*_{\text{in}}(\mathscr{L}(\mathcal{A}_{init} \times \mathcal{A}_{\mathcal{IRR}}(R))).$$

Proof. $\mathcal{A}_{\text{in}*}$ is correct w.r.t. R_{in}, but the case of Definition 29 where there remains a critical pair cannot occur, because it is a fixpoint. ◀

5 Precision theorem

We just showed that the approximation is correct. Now we investigate its accuracy on a theoretical point of view. This theorem is technical and difficult to prove (details can be found in [15]). But, this result is crucial because producing an over-approximation of reachable terms is easy (the tree automaton recognising $T(\Sigma)$ is a correct over-approximation) but producing an accurate approximation is hard. To the best of our knowledge, no other work dealing with abstract interpretation of functional programs or computing approximations of regular languages can provide such a formal precision guarantee (except [14] but in the case of general rewriting). Like in [14], we formally quantify the accuracy w.r.t. rewriting modulo E, replaced here by *innermost* rewriting modulo E. The relation of innermost rewriting modulo E, denoted by $\rightarrow_{R_{\text{in}}/E}$, is defined as rewriting modulo E where $\rightarrow_{R_{\text{in}}}$ replaces $\rightarrow_R$. We also define $(R_{\text{in}}/E)(L)$ and $(R_{\text{in}}/E)^*(L)$ in the same way as $R(L)$, $R^*(L)$ where $\rightarrow_{R_{\text{in}}/E}$ replaces $\rightarrow_R$.

The objective of the proof is to show that the completed tree automaton recognises no more terms than those reachable by R_{in}/E rewriting. The accuracy relies on the R_{in}/E-coherence property of the completed tree automaton, defined below. Roughly, a tree automaton $\mathcal{A}$ is R_{in}/E-coherent if $\overset{*}{\underset{\mathcal{A}}{\rightarrowtail}}$ is coherent w.r.t. R innermost rewriting steps and E equational steps. More precisely if $s \overset{*}{\underset{\mathcal{A}}{\rightarrowtail}} q$ and $t \overset{*}{\underset{\mathcal{A}}{\rightarrowtail}} q$ with no epsilon transitions with colour $\mathfrak{R}$, then $s =_E t$ (this is called separation of E-classes for $\mathcal{A}^{\not\mathfrak{R}}$). And, if $t \overset{*}{\underset{\mathcal{A}}{\rightarrowtail}} q$ with at least one epsilon transitions with colour $\mathfrak{R}$, then $s \rightarrow^*_{R_{\text{in}}/E} t$ (this is called R_{in}-coherence of $\mathcal{A}$). Roughly, a tree automaton separates E-classes if all terms recognized by a state are E-equivalent. Later, we will require this property on $\mathcal{A}_0$ and then propagate it on $\mathcal{A}_i^{\not\mathfrak{R}}$, for all completed automata $\mathcal{A}_i$.

▶ **Definition 32** (Separation of E-classes). The pair automaton $\mathcal{A}$ separates the classes of E if for any $q \in \Pi_1(Q_{\mathcal{A}})$, there is a term s such that for all $p \in \Pi_2(Q_{\mathcal{A}})$, $\mathscr{L}(\mathcal{A}, \langle q, p\rangle) \subseteq [s]_E$. We denote by $[q]^{\mathcal{A}}_E$ the common class of terms in $\mathscr{L}(\mathcal{A}, \langle q, \cdot\rangle)$, and extend this to configurations. We say the separation of classes by $\mathcal{A}$ is total if $\Pi_1(\mathcal{A})$ is accessible.

▶ **Definition 33** (R_{in}/E-coherence). An automaton $\mathcal{A}$ is R_{in}/E-coherent if

1. $\mathcal{A}^{\not\mathfrak{R}}$ totally separates the classes of E,
2. $\mathcal{A}$ is accessible, and
3. for any state $\langle q, p\rangle$ of $\mathcal{A}$, $\mathscr{L}(\mathcal{A}, \langle q, p\rangle) \subseteq (R_{\text{in}}/E)^* \left([q]^{\mathcal{A}^{\not\mathfrak{R}}}_E\right)$.

Then, the objective is to show that the two basic elements of innermost equational completion: completing a critical pair and applying an equation preserve R_{in}/E-coherence. This is the purpose of the two following lemmas.

▶ **Lemma 34.** *Completion of an innermost critical pair preserves R_{in}/E-coherence.*

▶ **Lemma 35.** *Equational simplification preserves R_{in}/E-coherence.*

This shows that, under the assumption that $\mathcal{A}_0$ separates the classes of E, innermost equational completion will never never add to the computed approximation a term that is not a descendant of $\mathscr{L}(\mathcal{A}_0)$ through R_{in} modulo E rewriting. This permits to state the main theorem, which formally defines the precision of the completed tree automaton.

▶ **Theorem 36** (Precision)**.** *Let E be a set of equations. Let $\mathcal{A}_0 = \mathcal{A}_{init} \times \mathcal{A}_{IRR}(R)$, where $\mathcal{A}_{init}$ has designated final states. We prune $\mathcal{A}_0$ of its non-accessible states. Suppose $\mathcal{A}_0$ separates the classes of E. Let R be any left-linear TRS. Let $\mathcal{A}_i$ be obtained from $\mathcal{A}_0$ after some steps of innermost equational completion. Then*

$$\mathscr{L}(\mathcal{A}_i) \subseteq (R_{\text{in}}/E)^*(\mathscr{L}(\mathcal{A}_0))).$$

Proof. (Sketch) We know that $\mathcal{A}_0$ is R_{in}/E-coherent because (1) $\mathcal{A}_0^{\not\varepsilon}$ separates the classes of E ($\mathcal{A}_0$ separates the classes of E and $\mathcal{A}_0 = \mathcal{A}_0^{\not\varepsilon}$ since none of $\mathcal{A}_{init}$ and $\mathcal{A}_{IRR}$ have epsilon transitions), and (2) $\mathcal{A}_0$ is accessible. Condition (3) of Definition 33 is trivially satisfied since $\mathcal{A}_0$ separates classes of E, meaning that for all states q, there is a term s s.t. $\mathscr{L}(\mathcal{A}_0, \langle q, p\rangle) \subseteq [s]_E$, *i.e.* all terms recognized by q are E-equivalent to s which is a particular case of case (3) in Definition 33. Then, during successive completion steps, by Lemma 34 and 35, we know that each basic transformation applied on $\mathcal{A}_0$ (completion or equational step) will preserve the R_{in}/E-coherence of $\mathcal{A}_0$. Thus, $\mathcal{A}_i$ is R_{in}/E-coherent. Finally, case (3) of R_{in}/E-coherence of $\mathcal{A}_i$ entails the result. ◀

Note that the fact that $\mathcal{A}_0$ needs to separate the classes of E is not a strong restriction. In the particular case of functional TRS (TRS encoding first order typed functional programs [12]), there always exists a tree automaton recognising a language equal to $\mathscr{L}(\mathcal{A}_0)$ and which separates the classes of E, see [11] for details.

6 Improving accuracy of static analysis of functional programs

We just showed accuracy of the approximation on a theoretical side. Now we investigate the accuracy on a practical point of view. There is a recent and renewed interest for Data flow analysis of higher-order functional programs [25, 22] that was initiated by [20]. None of those techniques is strategy-aware: on Example 8, they all consider the term $c(a(s(0)), f(n))$ as reachable, though it is not with innermost strategy. Example 8 also shows that this is not the case with innermost completion.

We made an alpha implementation of innermost equational completion. This new version of Timbuk, named TimbukSTRAT, is available at [13] along with several examples. On those examples, innermost equational completion runs within milliseconds. Sets of approximation equations, when needed, are systematically defined using [12]. Roughly, the idea is to define a set E such that the set of equivalence classes of $T(\Sigma)$ w.r.t. E is finite. Now, we show that accuracy of innermost equational completion can benefit to static analysis of functional programs. As soon as one of the analysed functions is not terminating (intentionally or because of a bug), not taking the evaluation strategy into account may result into an imprecise analysis. Consider the following OCaml program:

```
let hd= function x::_ -> x;;          let tl= function _::l -> l;;
let rec delete e l=
 if (l=[]) then [] else if (hd l=e) then tl l else (hd l)::(delete e l);;
```

It is faulty: the recursive call should be `(hd l)::(delete e (tl l))`. Because of this error, any call `(delete e l)` will not terminate if `l` is not empty and `hd l` is not `e`. We can encode the above program into a TRS R. Furthermore, if we consider only two elements in lists (`a` and `b`), the language L of calls to `(delete a l)`, where `l` is any non empty list of `b`, is regular. Thus, standard completion can compute an automaton over-approximating $R^*(L)$. Besides, the automaton $\mathcal{A}_{\mathcal{IRR}}(R)$ recognising normal forms of R can be computed since R is left-linear. Then, by computing the intersection between the two automata, we obtain the automaton recognising an over-approximation of the set of reachable terms in normal form[4]. Assume that we have an abstract OCaml interpreter performing completion and intersection with $\mathcal{A}_{\mathcal{IRR}}(R)$:

```
# delete a [b+];;
-:abst list= empty
```

The result `empty` reflects the fact that the `delete` function does not compute any *result*, *i.e.* it is not terminating on all the given input values. Thus the language of results is empty. Now, assume that we consider calls like `hd(delete e l)`. In this case, any analysis technique ignoring the call-by-value evaluation strategy of OCaml will give imprecise results. This is due to the fact that, for any non empty list `l` starting with an element `e'` different from `e`, `(delete e l)` rewrites into `e'::(delete e l)`, and so on. Thus `hd(delete e l)`, can be rewritten into `e'` with an outermost rewrite strategy. Thus, if we use an abstract OCaml interpreter built on the standard completion, we will have the following interaction:

```
# hd (delete a [b+]);;
-:abst list= b
```

The result provided by the abstract interpreter is imprecise. It fails to reveal the bug in the delete function since it totally hides the fact that the delete function does not terminate! Using innermost equational completion and TimbukSTRAT on the same example would permit to have the expected result which is[5]:

```
# hd (delete a [b+]);;
-:abst list= empty
```

We can perform the same kind of analysis for the program `sum` given in the introduction. This program does not terminate with call-by-value (for any input) but it terminates with call-by-name strategy. Again, strategy-unaware methods cannot show this: there are (outermost) reachable terms that are in normal form: the integer results obtained with a call-by-need or lazy evaluation. An abstract OCaml interpreter unaware of strategies would say:

```
# sum s*(0);;
-:abst nat= s*(0)
```

where a more precise and satisfactory answer would be `-:abst nat= empty`. Using TimbukSTRAT, we can get this answer. To over-approximate the set of results of the function `sum` for all natural numbers `i`, we can start innermost equational completion with the initial regular language $\{sum(s^*(0))\}$. Let $\mathcal{A} = (\Sigma, Q, Q_f, \Delta)$ with $Q_f = \{q_1\}$ and $\Delta = \{0 \rightarrowtail q_0, s(q_0) \rightarrowtail q_0, sum(q_0) \rightarrowtail q_1\}$ be an automaton recognising this language. Timbuk[13] can compute the automaton $\mathcal{A}_{\mathcal{IRR}}(R)$. Innermost equational completion with TimbukSTRAT terminates on an automaton (see [15]) where the only product state labelled by q_1 is $\langle q_1, p_{\text{red}}\rangle$. This means that terms of the form $sum(s^*(0))$ have no innermost normal form, i.e. the function `sum` is *not terminating* with call-by-value for all input values. On all

[4] Computing $\mathcal{A}_{\mathcal{IRR}}(R)$ and the intersection can be done using Timbuk.
[5] Details in [15]; see files `nonTerm1` and `nonTerm1b` in the TimbukSTRAT distribution at [13].

those examples, we used initial automata $\mathcal{A}$ that were not separating equivalences classes of E. On those particular examples the precision of innermost completion was already sufficient for our verification purpose. Yet, if accuracy is not sufficient, it is possible to refine $\mathcal{A}$ into an equivalent automaton separating equivalences classes of E, see [11]. When necessary, this permits to exploit the full power of the precision Theorem 36 and get an approximation of innermost reachable terms, as precise as possible, w.r.t. E.

On the same example, all aforementioned techniques [25, 22, 20], as well as all standard completion techniques [27, 14, 23], give a more coarse approximation and are unable to prove strong non-termination with call-by-value. Indeed, those techniques approximate all reachable terms, independently of the rewriting strategy. Their approximation will, in particular, contain the integer results that are reachable by the call-by-need evaluation strategy.

7 Related work

No tree automata completion-like techniques [10, 27, 3, 14, 23] take evaluation strategies into account. They compute over-approximations of *all* reachable terms.

Dealing with reachable terms and strategies was first addressed in [26] in the exact case for innermost and outermost strategies but only for some restricted classes of TRSs, and also in [9]. As far as we know, the technique we propose is the first to over-approximate terms reachable by innermost rewriting for *any* left-linear TRSs. For instance, Example 8 and examples of Section 6 are in the scope of innermost equational completion but are outside of the classes of [26, 9]. For instance, the `sum` example is outside of classes of [26, 9] because a right-hand side of a rule has two nested defined symbols and is not shallow.

Data flow analysis of higher-order functional programs is a long standing and very active research topic [25, 22, 20]. Used techniques ranges from tree grammars to specific formalisms: HORS, PMRS or ILTGs and can deal with higher-order functions. Higher-order functions are not in the scope of the work presented here, though it is possible with tree automata completion in general [16]. None of [25, 22, 20], takes evaluation strategies into account and analysis results are thus coarse when program execution rely on a specific strategy.

8 Conclusion

In this paper, we have proposed a sound and precise algorithm over-approximating the set of terms reachable by innermost rewriting. As far as we know this is the first algorithm solving this problem for any left linear TRS and any regular initial set of terms. It is based on tree automata completion and equational abstractions with a set E of approximation equations. The algorithm also minimises the set of added transitions by completing the product automaton (between $\mathcal{A}_{init}$ and $\mathcal{A}_{\mathcal{IRR}}(R)$). We proposed TimbukSTRAT [13], a prototype implementation of this method.

The precision of the approximations have been shown on a theoretical and a practical point of view. On a theoretical point of view, we have shown that the approximation automaton recognises no more terms than those effectively reachable by innermost rewriting modulo the approximation E. On the practical side, unlike other techniques used to statically analyse functional programs [25, 22, 20], innermost equational completion can take the call-by-value strategy into account. As a result, for programs whose semantics highly depend on the evaluation strategy, innermost equational completion yields more accurate results. This should open new ways to statically analyse functional programs by taking evaluation strategies into account.

Approximations of sets of ancestors or descendants can also improve existing termination techniques [17, 24]. In the dependency pairs setting, such approximations can remove edges in a dependency graph by showing that there is no rewrite derivation from a pair to another. Besides, it has been shown that dependency pairs can prove innermost termination [19]. In this case, *innermost* equational completion can more strongly prune the dependency graph: it can show that there is no *innermost* derivation from a pair to another. For instance, on the TRS:

$$\begin{array}{l|l|l} \mathit{choice}(X,Y) \to X & \mathit{choice}(X,Y) \to Y & \mathit{eq}(s(X),s(Y)) \to \mathit{eq}(X,Y) \\ \mathit{eq}(0,0) \to \mathit{tt} & \mathit{eq}(s(X),0) \to \mathit{ff} & \mathit{eq}(0,s(Y)) \to \mathit{ff} \\ g(0,X) \to \mathit{eq}(X,X) & g(s(X),Y) \to g(X,Y) & f(\mathit{ff},X,Y) \to f(g(X,\mathit{choice}(X,Y)),X,Y) \end{array}$$

We can prove that any term of the form $f(g(t_1, \mathit{choice}(t_2,t_3)), t_4, t_5)$ cannot be rewritten (innermost) to a term of the form $f(\mathit{ff}, t_6, t_7)$ (for all terms $t_i \in T(\Sigma)$, $i = 1 \dots 7$). This proves that, in the dependency graph, there is no cycle on this pair. This makes the termination proof of this TRS simpler than what AProVE [18] does: it needs more complex techniques, including proofs by induction. Simplification of termination proofs using innermost equational completion should be investigated more deeply.

For further work, we want to improve and expand our implementation of innermost equational completion in order to design a strategy-aware and higher-order-able static analyser for a reasonable subset of a real functional programming language with call-by-value like OCaml, F#, Scala, Lisp or Scheme. On examples taken from [25], we already showed in [16] that completion can handle some higher-order functions. We also want to study if the innermost completion covers the TRS classes preserving regularity of [26, 9], like standard completion does for many decidable classes [8].

Another objective is to extend this completion technique to other strategies. It should be easy to extend those results to the case of *leftmost* or *rightmost* innermost strategy. This should be a simple refinement of the second phase of completion of innermost critical pairs, when supplementary transitions are added. To encode leftmost (resp. rightmost) innermost, for each transition $f(q_1, \dots, q_{i-1}, \langle q, p_{red}\rangle, q_{i+1}, \dots, q_n) \rightarrowtail \langle q'', p''\rangle$, we should add a new transition $f(q_1, \dots, q_{i-1}, \langle q, p_{r\sigma}\rangle, q_{i+1}, \dots, q_n) \rightarrowtail \langle q'', p'''\rangle$, only if all states $q_1, \dots, q_{i-1}$ (resp. $q_{i+1}, \dots, q_n$) have a p component that is not p_{red}. Another strategy of interest for completion is the outermost strategy. This would improve the precision of static analysis of functional programming language using call-by-need evaluation strategy, like Haskell. Extension of this work to the outermost case is not straightforward but it may use similar principles, such as running completion on a pair automaton rather than on single automaton. States in tree automata are closely related to positions in terms. To deal with the innermost strategy, in states $\langle q, p\rangle$, the p component tells us if terms s (or subterms of s) recognised by the state $\langle q, p\rangle$ are reducible or not. This is handy for innermost completion because we can decide if a tuple $(\ell \to r, \sigma, \langle q', p'\rangle)$ is an *innermost* critical pair we checking if the p components of the states recognising strict subterms of $\ell\sigma$ are different from p_{red}. For the outermost case, this is exactly the opposite: a tuple $(\ell \to r, \sigma, \langle q', p'\rangle)$ is an *outermost* critical pair only if all the *contexts* $C[\,]$ such that $C[\ell\sigma]$ is recognised, are irreducible contexts. If it is possible to encode in the p' component (using an automaton or something else) whether all contexts embedding $\langle q', p'\rangle$ are irreducible or not, we should be able to define outermost critical pairs and, thus, outermost completion in a similar manner.

Acknowledgements. The authors thank René Thiemann for providing the example of innermost terminating TRS for AProVE, Thomas Jensen, Luke Ong, Jonathan Kochems, Robin Neatherway and the anonymous referees for their comments.

References

1. F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.
2. Y. Boichut, J. Chabin, and P. Réty. Over-approximating descendants by synchronized tree languages. In *RTA'13*, volume 21 of *LIPIcs*, pages 128–142. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2013.
3. Y. Boichut, R. Courbis, P.-C. Héam, and O. Kouchnarenko. Handling non left-linear rules when completing tree automata. *IJFCS*, 20(5), 2009.
4. C. H. Broadbent, A. Carayol, M. Hague, and O. Serre. C-shore: a collapsible approach to higher-order verification. In *ICFP'13*. ACM, 2013.
5. G. Castagna, K. Nguyen, Z. Xu, H. Im, S. Lenglet, and L. Padovani. Polymorphic functions with set-theoretic types: part 1: syntax, semantics, and evaluation. In *POPL'14*. ACM, 2014.
6. H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, C. Löding, S. Tison, and M. Tommasi. Tree automata techniques and applications. `http://tata.gforge.inria.fr`, 2008.
7. H. Comon and Jean-Luc Rémy. How to characterize the language of ground normal forms. Technical Report 676, INRIA-Lorraine, 1987.
8. G. Feuillade, T. Genet, and V. Viet Triem Tong. Reachability Analysis over Term Rewriting Systems. *Journal of Automated Reasonning*, 33 (3-4):341–383, 2004.
9. A. Gascon, G. Godoy, and F. Jacquemard. Closure of Tree Automata Languages under Innermost Rewriting. In *WRS'08*, volume 237 of *ENTCS*, pages 23–38. Elsevier, 2008.
10. T. Genet. Decidable Approximations of Sets of Descendants and Sets of Normal Forms. In *RTA'98*, volume 1379 of *LNCS*, pages 151–165. Springer, 1998.
11. T. Genet. A note on the Precision of the Tree Automata Completion. Technical report, INRIA, 2014. `https://hal.inria.fr/hal-01091393`.
12. T. Genet. Towards Static Analysis of Functional Programs using Tree Automata Completion. In *WRLA'14*, volume 8663 of *LNCS*. Springer, 2014.
13. T. Genet, Y. Boichut, B. Boyer, V. Murat, and Y. Salmon. Reachability Analysis and Tree Automata Calculations. IRISA / Université de Rennes 1. `http://www.irisa.fr/celtique/genet/timbuk/`.
14. T. Genet and R. Rusu. Equational tree automata completion. *Journal of Symbolic Computation*, 45:574–597, 2010.
15. T. Genet and Y. Salmon. Reachability Analysis of Innermost Rewriting. Technical report, INRIA, 2013. `http://hal.archives-ouvertes.fr/hal-00848260/PDF/main.pdf`.
16. T. Genet and Y. Salmon. Tree Automata Completion for Static Analysis of Functional Programs. Technical report, INRIA, 2013. `http://hal.archives-ouvertes.fr/hal-00780124/PDF/main.pdf`.
17. A. Geser, D. Hofbauer, J. Waldmann, and H. Zantema. On tree automata that certify termination of left-linear term rewriting systems. In *RTA'05*, volume 3467 of *LNCS*, pages 353–367. Springer, 2005.
18. J. Giesl, M. Brockschmidt, F. Emmes, F. Frohn, C. Fuhs, C. Otto, M. Plücker, P. Schneider-Kamp, T. Ströder, S. Swiderski, and R. Thiemann. Proving termination of programs automatically with aprove. In *IJCAR'14*, volume 8562 of *LNCS*, pages 184–191. Springer, 2014.
19. J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Mechanizing and improving dependency pairs. *Journal of Automated Reasonning*, 37(3):155–203, 2006.
20. N. D. Jones and N. Andersen. Flow analysis of lazy higher-order functional programs. *Theoretical Computer Science*, 375(1-3):120–136, 2007.

21 N. Kobayashi. Model Checking Higher-Order Programs. *Journal of the ACM*, 60.3(20), 2013.

22 J. Kochems and L. Ong. Improved Functional Flow and Reachability Analyses Using Indexed Linear Tree Grammars. In *RTA'11*, volume 10 of *LIPIcs*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2011.

23 A. Lisitsa. Finite Models vs Tree Automata in Safety Verification. In *RTA'12*, volume 15 of *LIPIcs*, pages 225–239, 2012.

24 A. Middeldorp. Approximations for strategies and termination. *ENTCS*, 70(6):1–20, 2002.

25 L. Ong and S. Ramsay. Verifying higher-order functional programs with pattern-matching algebraic data types. In *POPL'11*. ACM, 2011.

26 P. Réty and J. Vuotto. Regular Sets of Descendants by some Rewrite Strategies. In *RTA'02*, volume 2378 of *LNCS*. Springer, 2002.

27 T. Takai, Y. Kaji, and H. Seki. Right-linear finite-path overlapping term rewriting systems effectively preserve recognizability. In *RTA'11*, volume 1833 of *LNCS*. Springer, 2000.

28 Terese. *Term Rewriting Systems*. Cambridge University Press, 2003.

29 N. Vazou, P. Rondon, and R. Jhala. Abstract Refinement Types. In *ESOP'13*, volume 7792 of *LNCS*. Springer, 2013.

Network Rewriting II: Bi- and Hopf Algebras

Lars Hellström

Division of Applied Mathematics, The School of Education, Culture and Communication
Mälardalen University, Box 883, 721 23 Västerås, Sweden
lars.hellstrom@residenset.net

Abstract

Bialgebras and their specialisation Hopf algebras are algebraic structures that challenge traditional mathematical notation, in that they sport two core operations that defy the basic functional paradigm of taking zero or more operands as input and producing one result as output. On the other hand, these peculiarities do not prevent studying them using rewriting techniques, if one works within an appropriate network formalism rather than the traditional term formalism. This paper restates the traditional axioms as rewriting systems, demonstrating confluence in the case of bialgebras and finding the (infinite) completion in the case of Hopf algebras. A noteworthy minor problem solved along the way is that of constructing a quasi-order with respect to which the rules are compatible.

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

Keywords and phrases confluence, network, PROP, Hopf algebra

Digital Object Identifier 10.4230/LIPIcs.RTA.2015.194

1 Introduction

Bialgebras and Hopf algebras are rarely mentioned in first (or second) abstract algebra courses, but many familiar algebraic and combinatorial [5] structures possess a Hopf algebra structure, which may be viewed as giving a more complete picture of the basic thing than the mere algebra would. For polynomials in one variable x over a field $\mathcal{K}$, one may define the coproduct $\Delta\colon \mathcal{K}[x] \longrightarrow \mathcal{K}[x] \otimes \mathcal{K}[x]$, the counit $\varepsilon\colon \mathcal{K}[x] \longrightarrow \mathcal{K}$, and the antipode $S\colon \mathcal{K}[x] \longrightarrow \mathcal{K}[x]$ as the linear maps which satisfy

$$\Delta(x^n) = \sum_{k=0}^{n} \binom{n}{k} x^k \otimes x^{n-k}, \qquad \varepsilon(x^n) = \begin{cases} 1 & \text{if } n = 0, \\ 0 & \text{otherwise,} \end{cases} \qquad S(x^n) = (-1)^n x^n$$

for all $n \geqslant 0$; this turns $\mathcal{K}[x]$ into a Hopf algebra. For any group G, the corresponding group algebra $\mathcal{K}[G]$ is similarly endowed with coproduct $\Delta\colon \mathcal{K}[G] \longrightarrow \mathcal{K}[G] \otimes \mathcal{K}[G]$, counit $\varepsilon\colon \mathcal{K}[G] \longrightarrow \mathcal{K}$, and antipode $S\colon \mathcal{K}[G] \longrightarrow \mathcal{K}[G]$ defined by

$$\Delta(g) = g \otimes g, \qquad \varepsilon(g) = 1, \qquad S(g) = g^{-1}$$

for all $g \in G$ and then extended to the whole of $\mathcal{K}[G]$ by linearity, that turn $\mathcal{K}[G]$ into a Hopf algebra. If G is finite, then the linear dual of $\mathcal{K}[G]$ will moreover also be a Hopf algebra. Hopf algebras are thus close at hand, but they can for *syntactic* reasons be awkward to work with abstractly.

The simplest way to fully formalise the Hopf algebra concept of coproduct in a classical computational context would be that it is a function which returns a generator object for a finite sequence of pairs of algebra elements, because the basic way to encode a general

© Lars Hellström;
licensed under Creative Commons License CC-BY
26th International Conference on Rewriting Techniques and Applications (RTA'15).
Editor: Maribel Fernández; pp. 194–208
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

element in a vector space tensor product $V \otimes V$ is as a finite sum $a_1 \otimes b_1 + \cdots + a_n \otimes b_n$ where $a_i, b_i \in V$ for all $i = 1, \ldots, n$; note however that neither the length of this sum nor any particular term of it is uniquely determined by the element that the sum encodes. For example, in the polynomial Hopf algebra one can express $\Delta(x)$ as $1 \otimes x + x \otimes 1$, but equally well as $1 \otimes (x+1) + (x-1) \otimes 1$ or $(x+1) \otimes (x+1) - 1 \otimes 1 - x \otimes x$, since these are all the same element of $\mathcal{K}[x] \otimes \mathcal{K}[x]$. That the overall result of a computation should be independent of how such a tensor product element happened to get encoded places far-reaching constraints on what one may do to the a_is and b_is; in particular, all the a_i must be processed in the same way, and all the b_i must be processed in the same way, although a_is need *not* be processed in the same way as the b_i. Hence a more intuitive syntactic interpretation of the coproduct Δ is that it is like a subroutine with one in-parameter and two out-parameters, one of which is the "sequence" of a_i and the other being the corresponding "sequence" of b_i. In a composite expression, the left and the right results of a coproduct may then be used in quite separate places of the expression as a whole.

The counit ε is syntactically even stranger, as it also takes one operand as input, but produces *no* result as output, although it contributes a global factor to the final result of any composite expression of which it is part. Getting a grip on Δ and ε is difficult, and the main reason for this is precisely that they in the natural interpretation go beyond one of the fundamental principles of mathematical notation, namely that each expression is either atomic or a combination of *independent* subexpressions that each contributes one intermediate result to the final combining operation, thus giving every expression an underlying rooted tree structure. The two output results of a coproduct can instead create a syntactic dependence between what from the root looks like separate subexpressions, and the no output results of a counit can leave an expression syntactically disconnected, in both cases invalidating the traditional presumption that an expression is structured like a tree.

One approach for working with bialgebras has been to device special notational extensions to traditional notation, such as the Sweedler [10] notation which however has the drawback of having the bialgebra axioms built in; it cannot be used if one wishes to study the bialgebra axioms themselves. Another approach has been to give up on traditional expressions for equational reasoning, to rather work in the formalism of category theory: instead of an equational proof, one has a huge commutative diagram, where the various paths correspond to expressions, and the facets correspond to applications of axioms. An awkward trait of this approach is that it places considerable emphasis on such elementary issues as the domains of intermediate results at the expense of more structural aspects (like saying

$$\mathbb{R} \xrightarrow{\text{sin}} [-1,1] \xrightarrow{\text{sqr}} [0,1] \xrightarrow{t \mapsto 1-t} [0,1] \xrightarrow{\text{sqrt}} [0,1]$$

instead of $\sqrt{1 - \sin^2 x}$ while aiming to do basic calculus) and a significant disadvantage is that it requires many steps for trivial rearrangements of parentheses.

The approach followed here, to the end of examining bi- and Hopf algebras using techniques of rewriting, is instead to adopt a more general expression (formal term) concept, where the underlying structure is a DAG rather than a simple tree. This kind of generalisation is known from the works of for example Hasegawa [2], Lafont [6], and Mimram [8], but the exact realisation of it that will be used here is that of [3]. This *network* concept of more general expressions can be transcribed in terms of the categorical primitives of morphism composition, tensor product, and component permutation, but it is graphical (primarily as in graph, only secondarily as in graphics) and thus more accessible to the human eye. Even better, the matter of whether two categorical expressions are equal modulo the axioms of a symmetric monoidal category (the "rearrangement of parentheses" mentioned above) turns

out to be exactly the same as whether the corresponding networks are isomorphic (as graphs with some extra structure). There is much formal nonsense, but once the *many* minutiae of establishing the more general expression concept have been taken care of [3, Secs. 4, 5, 7], rewriting behaves *very much* as we're used to, even if some new phenomena pop up.

What is not a new phenomenon, but deserves to be stated explicitly, is that the rewriting operates on a higher level of abstraction than is usual in applications of rewriting to abstract algebra; the objects being rewritten do not represent/evaluate to elements of the algebra in question, but are rather expressions that could be applied to some tuple of algebra elements. More technically, the objects being rewritten represent (could be taken as evaluating to) multilinear maps $\mathcal{H}^{\otimes n} \longrightarrow \mathcal{H}^{\otimes m}$ which live in the PROP (symmetric monoidal category) of such maps that is generated by the five Hopf algebra operations. Rewrite theories that describe more specific Hopf algebras, for example "the free Hopf algebra generated by a given coalgebra", can be had by extending the generic system described below with extra constant operations representing the generating elements and extra rewrite rules representing how the coproduct and counit act on these generators; the resulting Hopf algebra is then the no-input-one-output component of the generated PROP.

Section 2 gives an introduction to the network formalism for bi- and Hopf algebras. Section 3 presents the axiom system for bialgebras and shows that it constitutes a confluent system of rewrite rules. Section 4 presents the axiom system for Hopf algebras and derives the additional rules needed to make the rewrite system complete. Section 5 shows that the system of the previous section indeed is confluent. Section 6 takes care of a technical detail left aside in the earlier critical pairs/completion oriented sections, namely that of how to construct a compatible order on the set of networks, to ensure termination.

2 Network formalism for bi- and Hopf algebras

In a Hopf algebra $\mathcal{H}$ over a field $\mathcal{K}$, there are five multilinear operations:

multiplication $\mu\colon \mathcal{H} \otimes \mathcal{H} \longrightarrow \mathcal{H}$ | antipode $S\colon \mathcal{H} \longrightarrow \mathcal{H}$ | unit $u\colon \mathcal{K} \longrightarrow \mathcal{H}$

coproduct $\Delta\colon \mathcal{H} \longrightarrow \mathcal{H} \otimes \mathcal{H}$ | counit $\varepsilon\colon \mathcal{H} \longrightarrow \mathcal{K}$

A bialgebra is not required to have an antipode. The graphic symbols shown are used to denote these operations in network notation expressions (see [3, Sec. 5] for the formal definition of network notation). These networks will be directed acyclic graphs where each inner vertex is one of the above five, and all edges by convention are directed downwards; no arrowheads are drawn. Edges beginning at the top of the network correspond to inputs and edges ending at the bottom correspond to outputs; together, these constitute the *legs* of the network. A network may be interpreted as a "circuit" performing Hopf algebra operations; any antichain k-edge-cut separating input side from output side is then the location of an intermediate result of the circuit; technically such an intermediate result is an element of the tensor power $\mathcal{H}^{\otimes k}$. When occurring as parts of a larger mathematical formula, network expressions are for clarity framed in brackets, like so:

$$[\ \ ] - [\ \ ] - [\ \ ] + [\,|\,|\,]$$

The rightmost of these networks is the identity map $\mathrm{id}^{\otimes 2} = \mathrm{id} \otimes \mathrm{id} : \mathcal{H}^{\otimes 2} \longrightarrow \mathcal{H}^{\otimes 2}$, whereas the first three in categorical notation rather would be $\Delta \circ \mu \circ (S \otimes S)$, $(\mathrm{id} \otimes \mu) \circ (\Delta \otimes \mathrm{id})$,

and $(\mu \otimes \mathrm{id}) \circ (\mathrm{id} \otimes \Delta)$. The rewriting formalism applied operates on linear combinations of networks, but the bialgebra and Hopf algebra axioms are all binomial, so the reader may for this paper ignore that aspect. (Rewrite rules always have simple networks, as opposed to formal linear combinations of networks, as left hand sides. Critical pairs/ambiguities thus only arise at simple networks, even though their resolutions might involve linear combinations if there are rules introducing such.)

Rewrite rules act on networks in the pictorially intuitive way of removing a subnetwork isomorphic to the left hand side and instead splicing in a subnetwork isomorphic to the right hand side, making sure that corresponding legs of the left and right hand sides are spliced into the same edge of the network being rewritten. Thus the rule

[ ] → [ ] can change [ ] into [ ] ;

it makes no mathematical difference that the crossing of two edges is shown above the two antipodes in the right hand side of the rule but below them in the spliced network (rightmost), as in this formalism the crossing of two edges is merely a presentational artifact that arises when the abstract network (a graph not given with an embedding) has to be depicted on a two-dimensional page.

Critical pairs (the formal term used in [3] is *decisive ambiguities*) arise when the left hand sides of two rules occur as overlapping subnetworks of some network that they cover completely, at least in the case of the rewrite systems considered here. (In some more general cases, there can be an ambiguity even when there is not an overlap.)

3 The bialgebra axioms and rewriting system

The bialgebra axioms are straightforward to state as network rewrite rules. First, there are the axioms for an associative unital algebra

associativity — s_1: [ ] → [ ] — $\mu \circ (\mathrm{id} \otimes \mu) \to \mu \circ (\mu \otimes \mathrm{id})$

left unit — s_2: [ ] → [|] — $\mu \circ (u \otimes \mathrm{id}) \to \mathrm{id}$

right unit — s_3: [ ] → [|] — $\mu \circ (\mathrm{id} \otimes u) \to \mathrm{id}$

then the dual axioms (obtained from the above by exchanging the roles of inputs and outputs) for a coassociative counital coalgebra

coassociativity — s_4: [ ] → [ ] — $(\mathrm{id} \otimes \Delta) \circ \Delta \to (\Delta \otimes \mathrm{id}) \circ \Delta$

left counit — s_5: [ ] → [|] — $(\varepsilon \otimes \mathrm{id}) \circ \Delta \to \mathrm{id}$

right counit — s_6: [ ] → [|] — $(\mathrm{id} \otimes \varepsilon) \circ \Delta \to \mathrm{id}$

and finally the axioms relating co- and non-co operations

s_7: [ ] → [ ] — $\varepsilon \circ u \to \mathrm{id}^{\otimes 0}$

s_8: [ ] → [ ] — $\Delta \circ u \to u \otimes u$

s_9: [ ] → [ ] — $\varepsilon \circ \mu \to \varepsilon \otimes \varepsilon$

s_{10}: [ ] → [ ] — $\Delta \circ \mu \to \mu \otimes \mu \circ$ $\mathrm{id} \otimes \tau \otimes \mathrm{id} \circ$ $\Delta \otimes \Delta$

where $\mathrm{id}^{\otimes 0}$ is the neutral element with respect to the tensor product operation $\otimes$ and τ is a "twist" map defined by $\tau(x \otimes y) = y \otimes x$ for all x and y; it is usually when reaching expressions the size of the right hand side of s_{10} that one starts to appreciate the network notation (and its less formalised kin, such as 'shorthand diagrams' [7] and Penrose's pictorial notation [9]) as an improvement over the raw categorical notation. In more traditional algebraic presentations, axioms s_7 and s_9 are often combined into the claim that ε is a unital algebra homomorphism, whereas axioms s_8 and s_{10} combine into the same claim about Δ. The crossing of edges (or τ twist) in the right hand side of s_{10} is then swept under the rug as a detail of how the multiplication operation of a tensor product algebra $\mathcal{H} \otimes \mathcal{H}$ is defined, but it is an important feature which deserves to be made explicit.

A sequence of rewriting steps modulo the system $\{s_1, s_2, \dots, s_{10}\}$ is

$$[\ ] \xrightarrow{s_{10}} [\ ] \xrightarrow{s_9} [\ ] \xrightarrow{s_5} [\ ] \xrightarrow{s_5} [\ ]$$

and that is also half of the resolution of the critical pair formed by rules s_{10} and s_5; the other half amounts to just one application of s_5. The full list[1] of networks being sites of critical pairs for this rewriting system is

and these all resolve in a quite straightforward manner. This list was compiled by enumerating all networks that satisfy the conditions of [3, Lemma 10.15]. Together with the quasi-order discussed in Section 6, this meets the conditions of the network rewriting diamond lemma [3, Th. 10.24], and so it follows that:

▶ **Theorem 1.** *The rewriting system $\{s_k\}_{k=1}^{10}$ is terminating and confluent.*

Remark on proof. This may seem abrupt, but proofs of confluence using a diamond lemma admit a degree of stylisation that almost render them redundant. Recall that a diamond lemma is a theorem of the form that if certain prerequisites are met, then various claims are equivalent; one of these claims is that a rewriting system is confluent and another that the rewriting system is locally confluent at each critical pair. Proofs relying upon it therefore tends to have two parts: first check that the prerequisites are met, which among other things

[1] This list of critical pairs, resolutions of these critical pairs, ditto for the extensions of the rewriting system treated below, and all drawings of networks shown in this paper, were computed using a utility for completion in network rewriting that was written by the author. The homepage of that utility, where its sources are available for download, is currently `http://www.mdh.se/ukk/personal/maa/lhm03/sw/rewriting`

establishes termination, second check the local confluence. But when termination holds, the matter of local confluence becomes algorithmically decidable, so recording the details of those calculations is not essential for the proof. The second part can therefore be abbreviated pretty much to the point of being omitted entirely, and the first part is often completely standard. (For this particular theorem, it is not so standard; the argument underlying termination can be found in Section 6, since the same argument would be used for all rewriting systems in this paper.)

What, on the other hand, may require a great deal of ingenuity and calculations is the construction of the confluent rewrite system. But the rewrite system must be included already in the statement of the theorem, so in a sense such claims tell the reader how to prove them. ◀

Networks which are built from μ, u, Δ, and ε vertices and moreover are on normal form with respect to $\{s_k\}_{k=1}^{10}$ can be fully characterised. Let $M_0 = u$, $M_1 = \mathrm{id}$, $M_{i+2} = M_{i+1} \circ (\mu \otimes \mathrm{id}^{\otimes i})$ for $i \geqslant 0$. Dually let $D_0 = \varepsilon$, $D_1 = \mathrm{id}$, and $D_{i+2} = (\Delta \otimes \mathrm{id}^{\otimes i}) \circ D_{i+1}$ for $i \geqslant 0$. Then the networks on normal form consist of three layers and have the overall form $A \circ B \circ C$, where $A = \bigotimes_{k=1}^{m} M_{p_k}$ for some numbers $\{p_k\}_{k=1}^{m} \subseteq \mathbb{N}$, the middle B part is a permutation, and $C = \bigotimes_{k=1}^{n} D_{q_k}$ for some numbers $\{q_k\}_{k=1}^{n} \subseteq \mathbb{N}$. In other words, the A part contains all the μ and u, whereas the C part contains all the Δ and ε, and both the A part and the C part are written on left-leaning form.

4 The Hopf algebra axioms and rewriting system

The situation for Hopf algebras is far more complicated. The traditional axiom system for these adds just two axioms to the ten of a bialgebra, namely

$f_{a0}\colon$ [diagram] $\to$ [diagram] and $f_{b0}\colon$ [diagram] $\to$ [diagram].

Logically, these two are all that is needed, but in practical calculations one needs to employ a number of derived rules. In particular, there are four rules describing interaction of an antipode with one of the four bialgebra operations:

$s_{11}\colon$ [diagram] $\to$ [diagram] $s_{12}\colon$ [diagram] $\to$ [diagram] $s_{13}\colon$ [diagram] $\to$ [diagram] $s_{14}\colon$ [diagram] $\to$ [diagram]

The rules s_{13} and s_{14} for how an antipode interacts with a counit and unit are fairly straightforward; they are among the first things an automated completion procedure discovers when given the Hopf algebra axioms as input, and the derivation of s_{14} is merely

[diagram] $\overset{s_7}{\leftarrow}$ [diagram] $\overset{f_{b0}}{\leftarrow}$ [diagram] $\overset{s_8}{\rightarrow}$ [diagram] $\overset{s_3}{\rightarrow}$ [diagram].

The rules s_{11} and s_{12} for how an antipode interacts with the multiplication and coproduct are on the other hand among the last spurious rules such a procedure discovers; a derivation

of s_{11} with some steps combined is

$$\xleftarrow{s_3,s_6} \xleftarrow{f_{a0}} \xrightarrow{s_1,s_4} \xleftarrow{s_3,s_6} \xleftarrow{f_{a0}} \xrightarrow{s_4} \xrightarrow{s_4} \xleftrightarrow{s_1} \xleftarrow{s_{10}} \xrightarrow{f_{b0}} \xrightarrow{s_9} \xrightarrow{s_2,s_5} \quad (1)$$

and the derivation of s_{12} is just the vertical flip (exchanging inputs and outputs, multiplication and coproduct, and unit and counit) of this one.

Given rules s_{11} and s_{12}, it is easy to see that these will form critical pairs with the axioms f_{a0} and f_{b0} that lead to the failed resolutions

$$\xleftarrow{s_{11}} \xrightarrow{f_{a0}} \xrightarrow{s_{14}} \quad \text{and} \quad \xleftarrow{s_{11}} \xrightarrow{f_{a0}} \xrightarrow{s_{14}} .$$

This thus calls for the introduction of two derived rules f_{c0} and f_{d0}, which are themselves involved in similar critical pairs, that in turn call for another two derived rules with an extra pair of antipodes and an extra crossing. Since crossing twice takes one back to the original uncrossed state, this second pair of derived rules may be called f_{a1} and f_{b1} as they look just like f_{a0} and f_{b0}, except with two extra antipodes on each of the two paths between coproduct and multiplication:

f_{a1}: $\rightarrow$ $\qquad$ f_{b1}: $\rightarrow$

Continuing this way, one will generate four infinite families of rules, where the members of a family differ only in how many extra antipodes are inserted between the coproduct and the multiplication in the left hand side, but all rules in a family have the same right hand side. To present this succinctly in network notation, it becomes convenient to introduce a special *double antipode sequence* vertex ⊞, that denotes a path of some $2n$ antipode vertices; the number n will appear as an index in the rule name. Note especially that all double antipode sequence vertices in a single network denote the *same* even number of antipodes. Using this, the f_{an}, f_{bn}, f_{cn}, and f_{dn} families of rules are what is shown in the top row of Figure 1.

Families a–d also form critical pairs with rules s_1 and s_4, that do not resolve using the rules mentioned so far; one example is

$$\xleftarrow{s_1} \xrightarrow{f_{an}} \xrightarrow{s_3} .$$

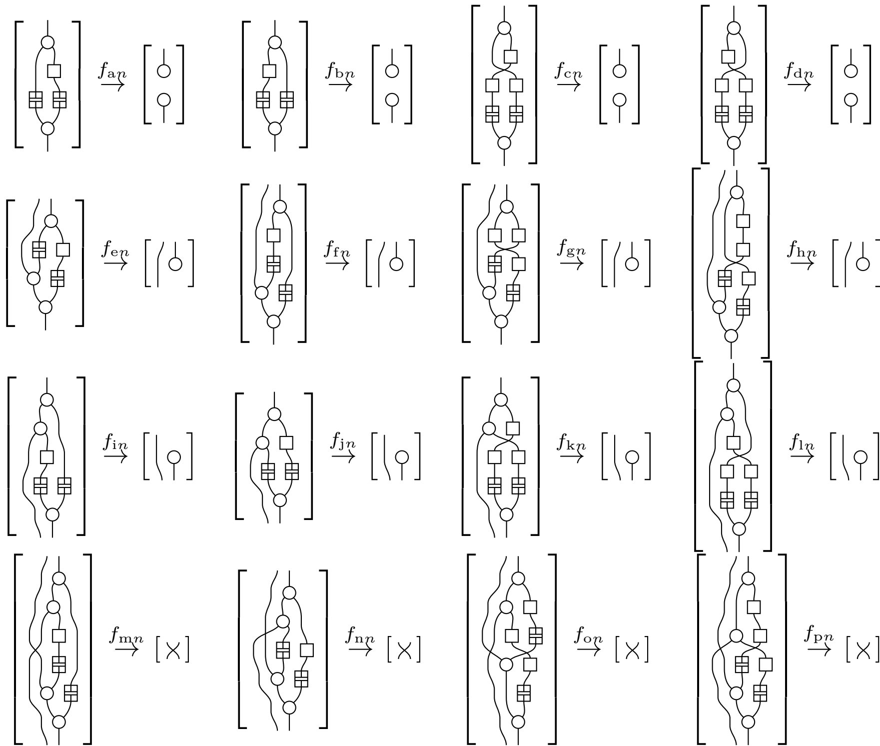


Figure 1 The sixteen infinite families of rewrite rules for Hopf algebras. The letters a–p in the index correspond to the subequation labels in [3, Eq. 1.2], where this system of rewrite rules was first announced.

These failed resolutions thus give rise to additional families of derived rules; with s_1 one gets $f_{\mathrm{e}n}$ through $f_{\mathrm{h}n}$ and with s_4 one gets $f_{\mathrm{i}n}$ through $f_{\mathrm{l}n}$, also defined in Figure 1. Furthermore families e–h form critical pairs with s_4 that give rise to another four families m–p, and those same four families also arise from critical pairs of s_1 and a member of families i–l. But after these last four that bring the total up to sixteen families $\{f_{\mathrm{a}n}\}_{n=0}^{\infty}$ through $\{f_{\mathrm{p}n}\}_{n=0}^{\infty}$, there are no more derived rules to discover; the rewrite system is, as shall be shown in Section 5, complete.

The last four families, the simplest member of which is

$$f_{\mathrm{n}0}\colon \quad [\;\cdots\;] \to [\times], \qquad \text{so that} \quad [\;\cdots\;] \to [\;\cdots\;]$$

(the filled vertex in that formula is to be read as a placeholder for an arbitrary network expression) do however exhibit an interesting property: they apply even in places where the first input is reachable from the first output. Note that since networks are by definition DAGs, a rewrite formalism for networks may not perform any surgery that would introduce cycles. A simple condition to that effect would be that left hand sides of rules may only be identified with networks in such a way that no directed path exists from a left hand side output to a left hand side input, because that ensures the result of applying the rule is also acyclic no

matter what the right hand side looks like. Though simple, this *convexity* condition turns out to be a bit too restrictive in practice, and a more appropriate condition is that a rewrite rule should not introduce any *new* connections; the right hand side should not have a path from input j to output i unless there was such a path in the left hand side.

Remember that a derived rule can be considered a bottled sequence of proof steps exercising more elementary rules; any application of a derived rule can, as far as equational reasoning is concerned, be replaced by a sequence of more elementary steps. When considering such a sequence of steps in the context of a larger network, there is no a priori reason why the union of the subnetworks operated upon in the more elementary steps should always be convex, and in general it will indeed not be. What matters for the validity of a derived rule is merely that the elementary steps it is a parcel for can be carried out in every context where the rule is claimed to apply, and that is certainly the case with rule families m–p.

Experience with completing a modification of the Hopf axiom system suggests that rules will typically become nonconvex as soon as they grow complicated enough. An open problem in the more general case is however that there may exist more connections in the intermediate steps of a rule derivation than there are in neither the final left or right hand sides. In this case the rule is *non-sharp* [3, Def. 10.3], and it may be involved in critical pairs other than the decisive ambiguities, a matter which requires further research [4]. The rewrite rules considered in this paper are however all sharp, so that is not a concern for the results stated here.

5 Confluence of the Hopf system

For the matter of proving confluence of the system of Hopf algebra rules derived in the previous section, one may begin with the system of the fourteen spurious rules s_1 through s_{14}. Since this is a superset of the bialgebra system, all the critical pairs of that system arise again, but they can also be resolved in exactly the same way as there. The additional critical pairs that arise are at the sites

[ ] , [ ] , [ ] , [ ] , [ ] , [ ] , [ ] , [ ] , [ ] , [ ] ,

and these also resolve through straightforward calculations. Again putting aside until Section 6 the technical details concerning the construction of a compatible order on the set of networks, it may now be claimed that:

▶ **Theorem 2.** *The rewriting system* $\{s_k\}_{k=1}^{14}$ *is terminating and confluent.*

The normal form modulo $\{s_k\}_{k=1}^{14}$ of a Hopf algebra expression is a three-layered $A \circ B \circ C$ as in the case of a bialgebra, but with the difference that B in this case may contain antipodes. Hence rather than being a simple matching (as permutations are), the B network is in general a disjoint union of paths where each path may contain any number (including 0) of antipode vertices. Adding the sixteen infinite families to the system will reduce that slightly, but not very much.

Continuing with that full rewriting system $F = \{s_k\}_{k=1}^{14} \cup \{f_{\mathrm{a}n}\}_{n=0}^{\infty} \cup \cdots \cup \{f_{\mathrm{p}n}\}_{n=0}^{\infty}$, one may first observe that the spurious rules s_3, s_6, s_7, s_{13}, and s_{14} do not form any critical pairs with the family rules. Rules s_8 and s_9 form critical pairs, but these all resolve very easily as a unit or counit will effectively gobble any vertex to which it becomes adjacent. Rules s_1, s_2, s_4, s_5, s_{11}, and s_{12} are another matter, as they get involved in a rather complicated

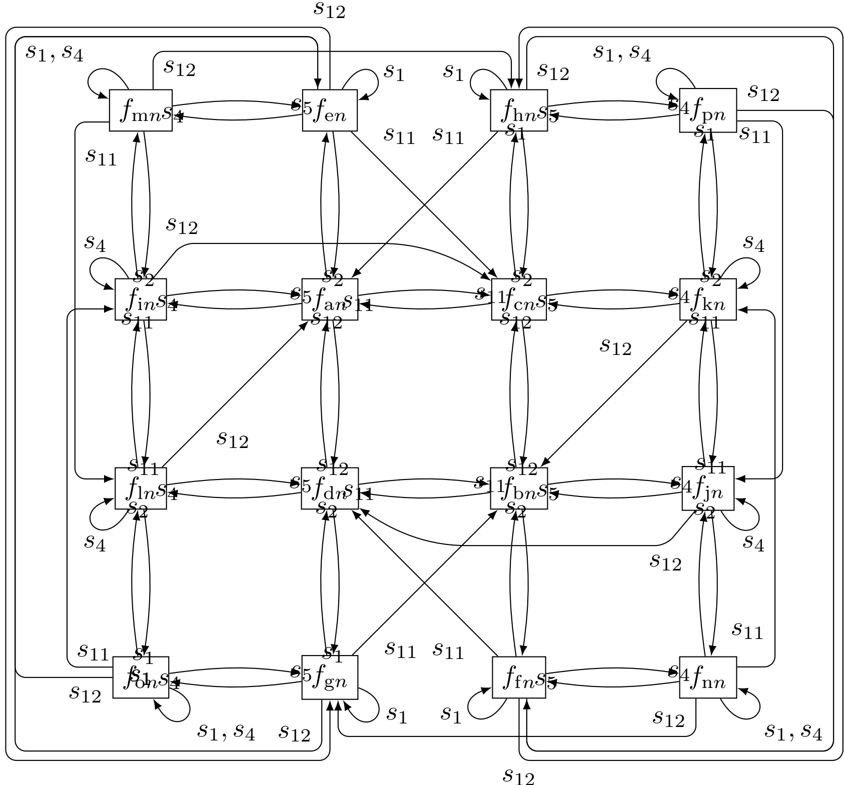


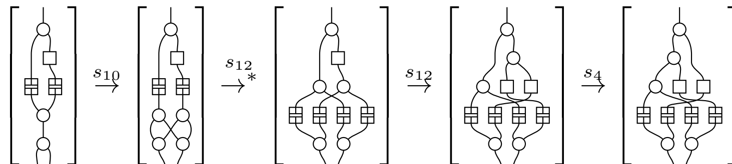
Figure 2 The effect of spurious rules on family rules. Critical pairs formed by one spurious rule from $\{s_1, s_2, s_4, s_5, s_{11}, s_{12}\}$ and some rule f_{xn} from one of the sixteen infinite families resolve in a number of spurious rule steps and one family rule step; in several but not all cases, these resolutions can alternatively be used as derivations of that family rule. The head of an arrow point at the family of which a member might be derived, whereas the family at the tail and the arrow label correspond to the rules that would be involved in the critical pair.

dance of transforming rules of one family into rules of another family (or sometimes the same family); Figure 2 gives an overview of how the families connect. If not for the fact that all rules in a family form the same kind of critical pair with a spurious rule, and also that the resolutions are (within each family) all trivial variations on each other, it would be very hard to verify that the resolutions all succeed. The sheer volume of calculations that are needed is such that one appreciates also having obtained a computer verification[2] of them (for the first couple of rules from each family), even though it remains within the realm of what can be carried out manually.

The final spurious rule s_{10} is not only the one most prolific in forming critical pairs with family rules (once for the coproduct at the top, once for the multiplication at the bottom), but also the one which creates the most complicated intermediate steps in their resolutions.

[2] Using the utility mentioned in a previous footnote.

A typical sequence is



where now a rule from family $f_{\mathrm{j}n}$ applies on the subnetwork containing the right multiplication, and then a rule from family $f_{\mathrm{a}n}$ resolves the rest. In the middle step it looks unlikely that any family rule can apply, because the crossing introduced by rule s_{10} means that two paths of antipodes that are adjacent on the coproduct side are not adjacent on the multiplication side, but what makes everything fit together is that one of the original paths between coproduct and multiplication has an even number of antipodes whereas the other has an odd number, and thus there is an extra twist at the end which makes one pair of paths adjacent on both sides, after which the resolution becomes straightforward. So even though rule s_{10} creates a lot of noise as far as critical pairs are concerned, it does not really contribute anything interesting here. (Note however that rule s_{10} plays a crucial role at one point in the derivation (1) of rule s_{11}.)

The final case of critical pair would be one formed by two family rules, and although that happens (for example between $f_{\mathrm{a}n}$ and $f_{\mathrm{m}n}$), it does not happen very often. The main reason is that the overlap has to take the form of a path starting in a coproduct, passing some number of antipodes, and ending in a multiplication; this places a strong restriction on the n values that might be involved, as both rules must have such paths with the same number of antipode vertices. Considering in addition that twisted families (c, d, g, h, k, l, o, and p) cannot form overlaps with the straight families (a, b, e, f, i, j, m, and n), one ends up with the conclusion that the n values of both rules involved must in fact be equal, and then it follows that all these critical pairs have trivial resolutions. Thus we have, again in anticipation of the technical details that will be dealt with in the next section, the main claim that:

▶ **Theorem 3.** *The full Hopf rewriting system F is terminating and confluent.*

As before, networks on normal form with respect to F can be written as $A \circ B \circ C$ where all the μ and u are in A, all the antipodes S are in B, and all Δ and ε are in C. What is new in the full system is that those arrangements of coproduct, antipodes, and multiplication upon which one of the family rules would act are forbidden, but which arrangements are those? Define a *mid-section path* of a network to be one that begins in a coproduct, have antipodes as inner vertices, and ends in a multiplication. Two mid-section paths are said to be *adjacent* on the coproduct or multiplication side if their outermost vertices on that side are adjacent or coincide. Clearly, the family rules may only apply to pairs of paths that are adjacent on both sides. Moreover, the number of antipodes on the paths in the pair must differ by 1, and the paths must cross (or not cross) depending on whether it is the path with the even number of antipodes that is the longer (or shorter, respectively).

6 Compatible ordering of networks

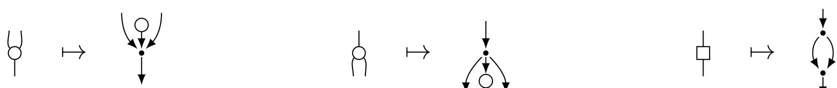
The diamond lemma in [3] is a descendant of Bergman's diamond lemma for associative algebras [1], so it requires a well-founded quasi-order P on the set of networks, that on one hand is compatible with the rules of the rewriting system, and on the other is strictly

preserved under composition of networks. This turns out to not be entirely trivial to construct in this setting.

The main complication is the coproduct–multiplication rule s_{10}, since this has the unhelpful property of *increasing* the number of vertices in a network; were it not for this (and to a lesser extent rules s_{11} and s_{12}), one could have ensured well-foundedness simply by first ordering networks by the number of vertices in them. A generalisation to weighted vertex counts achieves nothing, since one would need $w_\mu + w_u \geqslant 0$ for rules s_2 and s_3, $w_\Delta + w_\varepsilon \geqslant 0$ for rules s_5 and s_6, $w_\mu \geqslant w_\varepsilon$ for rule s_9, $w_\Delta \geqslant w_u$ for rule s_8, and $0 \geqslant w_\mu + w_\Delta$ for rule s_{10}, all of which taken together merely imply that we have equality in all those inequalities. Beyond weighted vertex counts, it is not easy to come up with an ordering principle that is preserved under composition; most elementary suggestions of orders one can make up that do take the structure of an expression into account tend to fail at being preserved under composition.

Might it be better to orient some rules the other way? But no, this is the natural orientation; rules s_8, s_9, s_{10}, s_{11} and s_{12} expand things, whereas most of the others remove superfluous operations, and only the orientations of s_1 and s_4 are really arbitrary. It is natural that s_{11} changes expressions so that antipodes are applied before the multiplication rather than after, so how would one formalise this intuition? Obviously the order in which operations are performed matters, but how does one express that when comparing networks, as the structure of one network can be quite different from the structure of another? One possibility is to compare the sequence in which different vertex types occur along paths from input to output, because in the left hand side of s_{10} each path passes first a μ vertex and then a Δ vertex, whereas in the right hand side it is Δ first and μ second; the same kind of condition works for s_{11} and s_{12}. The only catch is that in order to be preserved under composition of networks, these comparisons must be performed separately for each pair of input and output, and also separately for paths that begin or end within a network.

In the end, it turns out to be sufficient to compare the *number* of paths through a network, provided that vertices are replaced by suitable gadgets, as follows:



Unfilled circles (like for the unit u and counit ε) may serve as start or end point of a path, but the filled dots may only occur as inner vertices on a path; hence a multiplication μ counts as having three types of paths: paths from the left input passing through, paths from the right input passing through, and paths beginning here and continuing through the output. With these substitutions, it turns out that both the left and right hand sides of s_{10} contribute the same number of paths reaching the boundary of the network (despite the right hand side having 3 edges more), but the left hand side in addition has a path that both starts and ends within the network (starting at the μ, ending at the Δ), which the right hand side does not (since every μ there, where such a path might start, comes after the Δs where it would have to end), and *therefore* the left hand side achieves a greater number of paths than the right hand side. Similarly in rule s_{11}, the antipode doubles the number of paths passing through it, but it is only in the left hand side that the antipode also doubles the number of paths beginning at the multiplication. This is how these rules can be oriented from greater to smaller, even though there are more vertices in the right hand side than in the left hand side.

That is however the intuitive explanation. The formal nonsense is rather that a suitable quasi-order is constructed by pulling back the standard [3, Constr. 6.1] PROP quasi-order [3, Def. 3.1] on the biaffine PROP Baff($\mathbb{N}$) [3, Ex. 2.15] along a cleverly chosen PROP homo-

morphism, namely that g which satisfies

$$g(\mu) = \begin{pmatrix} 1 & \mathit{0} & \mathit{0} & \mathit{0} \\ 0 & 1 & 0 & 0 \\ 0 & \mathit{1} & \mathit{1} & \mathit{1} \end{pmatrix} \quad g(u) = \begin{pmatrix} 1 & \mathit{0} \\ 0 & 1 \\ 0 & \mathit{1} \end{pmatrix} \quad g(\Delta) = \begin{pmatrix} 1 & \mathit{0} & \mathit{1} \\ 0 & 1 & 0 \\ 0 & \mathit{0} & \mathit{1} \\ 0 & \mathit{0} & \mathit{1} \end{pmatrix} \quad g(S) = \begin{pmatrix} 1 & \mathit{0} & \mathit{0} \\ 0 & 1 & 0 \\ 0 & \mathit{0} & \mathit{2} \end{pmatrix}$$

$$g(\varepsilon) = \begin{pmatrix} 1 & \mathit{0} & \mathit{1} \\ 0 & 1 & 0 \end{pmatrix};$$

these conditions uniquely determine a homomorphism, because the set of all networks is the free PROP [3, Th. 8.4]. The italic matrix entries are those that are not fixed for elements of the biaffine PROP and thus may be chosen, although for the resulting PROP order to be strict it is necessary that there is at least one positive element in each row and at least one positive element in each column [3, Cor. 6.5]. The relevant interpretation of an element of $\mathrm{Baff}(\mathbb{N})$ is that the $(i+2, j+2)$ entry keeps track of the number of paths going from input j to output i of a network, whereas entry $(i+2, 2)$ keeps track of the number of paths which begin inside the network and reach output i, entry $(1, j+2)$ keeps track of the number of paths which come from input j but end inside the network, and entry $(1, 2)$ keeps track of the number of paths which both begin and end inside the network. Thus the above argument about counting paths in a gadgetified s_{10} corresponds to the observation that

$$g\left(\left[\;\cdots\;\right]\right) = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix} > \begin{pmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix} = g\left(\left[\;\cdots\;\right]\right).$$

What is not distinguished by the order pulled back over that g are the left and right hand sides of rules s_1 and s_4; since these impose a left–right asymmetry, they would interact poorly with the left–right swaps introduced by rules s_{11} and s_{12}. To orient also these, one introduces a secondary comparison criterion (technically makes a lexicographic composition [3, Constr. 3.7] of quasi-orders) by pulling back along a second cleverly chosen homomorphism g_2, for example that which has

$$g_2(\mu) = \begin{pmatrix} 1 & \mathit{0} & \mathit{0} & \mathit{1} \\ 0 & 1 & 0 & 0 \\ 0 & \mathit{1} & \mathit{1} & \mathit{1} \end{pmatrix} \qquad g_2(\Delta) = \begin{pmatrix} 1 & \mathit{0} & \mathit{1} \\ 0 & 1 & 0 \\ 0 & \mathit{0} & \mathit{1} \\ 0 & \mathit{1} & \mathit{1} \end{pmatrix}$$

In the path-counting interpretation, this amounts to adding the opportunity to end a path entering through the right input of a μ and begin a path leaving through the right output of a Δ. This causes the third input of a right-leaning $\mu \circ (\mathrm{id} \otimes \mu)$ to offer two chances for a path to end, whereas the third input of a left-leaning $\mu \circ (\mu \otimes \mathrm{id})$ only sees one; this suffices for making the left hand side of s_1 strictly larger than the right hand side.

Two additional results of [3] that are important in verifying that the quasi-order constructed as described above meet the conditions for the diamond lemma (Th. 10.24) are Corollary 9.16 and Lemma 9.18. Arguably also Lemma 3.5 on well-foundedness, but that result is on the other hand quite standard.

7 Final remarks

An anonymous reviewer has asked about the difficulty of finding redexes in network rewriting. Since network rewriting is nondeterministic, this ends up being a search problem, but the

search is in practice fairly constrained: most of the time, a redex is uniquely determined by the correspondence of one vertex in the rule left hand side to one vertex in the network being reduced, because each edge incident with a vertex occupies a distinct "port" on that vertex (e.g. multiplication μ has a left incoming factor, a right incoming factor, and an outgoing result, no two of which are interchangeable). Therefore it is feasible to pick one vertex in the left hand side and try to match it against each vertex of the network to reduce; when things don't match, one tends to discover that early, and the only case in which the search might need to backtrack would be for a left hand side that is not connected. There are some additional complications related to keeping track of the *transference types* of rules [3, Defs. 6.14, 10.3], which can prevent something from being a redex even though the networks match, but that boils down to doing some extra bookkeeping.

A somewhat tougher problem is how to check which rules in a rewrite system might apply to a given network, especially when the rewrite system is large and in flux due to a (Knuth–Bendix style) completion being in progress. The author has implemented a system where each network is assigned a "signature" that counts occurrences of various small subgraphs therein, to the end of only considering rules whose signature is dominated by that of the target network. The performance has been good enough that reduction is not perceived as a problem when running large completions.

Acknowledgements. This work was begun in the spring of 2004, during my postdoc stay at the Mittag-Leffler institute as part of the NOG (Noncommutative Geometry) programme, funded by the European Science Foundation and the Royal Swedish Academy of Sciences. Much research has also been done while the author was associated with the Department of Mathematics and Mathematical Statistics at Umeå University. Final write-up was supported by The School of Education, Culture and Communication at Mälardalen University.

References

1 George M. Bergman. The diamond lemma for ring theory. *Adv. in Math.*, 29(2):178–218, 1978.

2 Masahito Hasegawa. *Models of sharing graphs.* CPHC/BCS Distinguished Dissertations. Springer-Verlag London, Ltd., London, 1999. A categorical semantics of let and letrec, Dissertation, University of Edinburgh, Edinburgh.

3 Lars Hellström. Network Rewriting I: The Foundation. *ArXiv e-prints*, 2012. arXiv:1204.2421 [math.RA].

4 Lars Hellström. Critical pairs in network rewriting. In Takahito Aoto and Delia Kesner, editors, *IWC 2014, 3rd International Workshop on Confluence*, pages 9–13, 2014. `http://www.nue.riec.tohoku.ac.jp/iwc2014/iwc2014.pdf`.

5 S. A. Joni and G.-C. Rota. Coalgebras and bialgebras in combinatorics. *Stud. Appl. Math.*, 61(2):93–139, 1979.

6 Yves Lafont. Towards an algebraic theory of Boolean circuits. *J. Pure Appl. Algebra*, 184(2-3):257–310, 2003.

7 Shahn Majid. Cross products by braided groups and bosonization. *J. Algebra*, 163(1):165–190, 1994.

8 Samuel Mimram. Computing critical pairs in 2-dimensional rewriting systems. In *RTA 2010: Proceedings of the 21st International Conference on Rewriting Techniques and Applications*, volume 6 of *LIPIcs. Leibniz Int. Proc. Inform.*, pages 227–241. Schloss Dagstuhl. Leibniz-Zent. Inform., Wadern, 2010.

9 Roger Penrose. Applications of negative dimensional tensors. In *Combinatorial Mathematics and its Applications (Proc. Conf., Oxford, 1969)*, pages 221–244. Academic Press, London, 1971.

10 Moss E. Sweedler. *Hopf algebras.* Mathematics Lecture Note Series. W. A. Benjamin, Inc., New York, 1969.

Leftmost Outermost Revisited*

Nao Hirokawa[1], Aart Middeldorp[2], and Georg Moser[2]

1 **School of Information Science**
JAIST, Japan
`hirokawa@jaist.ac.jp`

2 **Institute of Computer Science**
University of Innsbruck, Austria
`{aart.middeldorp|georg.moser}@uibk.ac.at`

Abstract

We present an elementary proof of the classical result that the leftmost outermost strategy is normalizing for left-normal orthogonal rewrite systems. Our proof is local and extends to hyper-normalization and weakly orthogonal systems. Based on the new proof, we study basic normalization, i.e., we study normalization if the set of considered starting terms is restricted to basic terms. This allows us to weaken the left-normality restriction. We show that the leftmost outermost strategy is hyper-normalizing for basically left-normal orthogonal rewrite systems. This shift of focus greatly extends the applicability of the classical result, as evidenced by the experimental data provided.

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

Keywords and phrases term rewriting, strategies, normalization

Digital Object Identifier 10.4230/LIPIcs.RTA.2015.209

1 Introduction

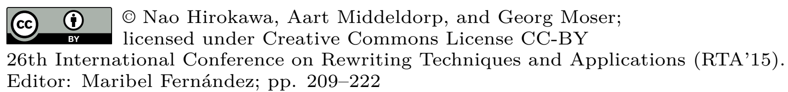
The (hyper-)normalization of the leftmost outermost strategy is a fundamental result in Combinatory Logic and λ-calculus. The importance of hyper-normalization as opposed to normalization stems from the fact that this property is essential to show that all partial recursive functions are definable in λ-calculus and Combinatory Logic. Consequently, numerous (hyper-)normalization proofs can be found in the literature and the result is of foundational interest.

On the other hand, as already observed by O'Donnell [20] effective normalization results are of significant practical interest. Functional programming languages need to be efficiently implemented. For this it is mandatory to study computable and normalizing strategies. Our motivation is mainly concerned with such practical considerations. Consider the term rewrite system (TRS for short) consisting of the rewrite rules

$$\begin{array}{rlrl}
\mathsf{primes}(n) & \to \mathsf{take}(n, \mathsf{sieve}(\mathsf{from}(\mathsf{s}(\mathsf{s}(\mathsf{0}))))) & \mathsf{filter}(\mathsf{0}, y : z, w) & \to \mathsf{0} : \mathsf{filter}(w, z, w) \\
\mathsf{sieve}(\mathsf{0} : y) & \to \mathsf{sieve}(y) & \mathsf{filter}(\mathsf{s}(x), y : z, w) & \to y : \mathsf{filter}(x, z, w) \\
\mathsf{sieve}(\mathsf{s}(x) : y) & \to \mathsf{s}(x) : \mathsf{sieve}(\mathsf{filter}(x, y, x)) & \mathsf{take}(\mathsf{0}, y) & \to \mathsf{nil} \\
\mathsf{from}(x) & \to x : \mathsf{from}(\mathsf{s}(x)) & \mathsf{take}(\mathsf{s}(n), x : y) & \to x : \mathsf{take}(n, y)
\end{array}$$

encoding the sieve of Eratosthenes.

* This research is supported by JSPS KAKENHI Grant Number 25730004, JSPS Core to Core Program, and the Austrian Science Fund (FWF) project P 25781-N18.



© Nao Hirokawa, Aart Middeldorp, and Georg Moser;
licensed under Creative Commons License CC-BY

When normalizing a term like primes(s(s(s(0)))), it is important to adopt a good evaluation strategy in order to avoid getting trapped in an infinite computation, which happens for instance with any innermost strategy. Efficiency is another desirable property of a strategy.

Since the TRS is orthogonal, we know that the maximal (parallel) outermost strategy is normalizing (see O'Donnell [20]) but it is also known that the maximal outermost strategy is not optimal in the sense that redexes are contracted which do not contribute to the normal form. Needed reduction is an optimal one-step normalizing strategy for orthogonal TRSs but in general not computable [15]. As a matter of fact, our example TRS happens to be strongly sequential and hence admits a computable optimal one-step strategy [16]. This strategy can be implemented using advanced data structures (matching dags [16], definitional trees [2] for constructor TRSs). Moreover, showing strong sequentiality of orthogonal TRSs is a non-trivial matter [7].

What about a popular and easy to implement strategy like leftmost outermost? Since left-normality is not satisfied, we actually do not know whether the leftmost outermost strategy is normalizing for this TRS. In this situation, we propose a shift of focus. Instead of contemplating normalization of *all* terms, we restrict our attention to *specific* starting terms, following the example of the term primes(s(s(s(0)))). That is, for practical considerations it seems sufficient if we restrict the set of starting terms to basic terms, which are terms that contain exactly one defined symbol, at the root position. This allows us to replace left-normality by a significant weaker restriction. This restriction, which we name *basic left-normality*, is satisfied for the above and many other non-left-normal TRSs, as witnessed by the experimental data that we present in this paper.

The proof is based on *usable replacement maps*, which were originally introduced by Fernández [10] for innermost termination analysis and adapted for complexity analysis of (full) rewriting in [14]. Effective computation of an approximation of usable replacement maps based on unification and fixed point computation is established in [14]. Employing this approximation in the context of basic left-normality yields an easily decidable criterion that ensures the normalization of the leftmost outermost strategy. Furthermore the strategy itself is very easy to implement.

There is a strong and ongoing trend to certify well-established results in all areas of rewriting. Certification not only helps to identify bugs in automated tools, but may also reveal mistakes in the underlying theory. Moreover, the huge body of research in this area shows that it is not unrealistic to aim for the certification of competitive tools. As our motivation is practical, certifiability of strategy tools is clearly of interest to us. Thus in addition to providing a simple and easy to implement strategy for basically left-normal TRS, we provide a formal foundation that is eventually machine-checkable, in order to yield certified evaluations.

Contribution. In this paper we introduce the class of basically left-normal weakly orthogonal TRSs, for which we establish the fundamental result of hyper-normalization for the leftmost outermost strategy starting from basic terms. Along the way we present an elementary proof of the hyper-normalization of the leftmost outermost strategy for the class of left-normal weakly orthogonal TRSs, which is a known result (Toyama [25]). Our proof is based on abstract quasi-commutation properties in connection with a careful analysis of the interplay of single leftmost outermost steps and parallel non-leftmost outermost steps. This gives rise to a local proof which lends itself better to future formalization efforts. We provide experimental evidence which clearly shows the applicability of our result.

Organization. The remainder of the paper is organized as follows. In the next section we recall some rewriting preliminaries and we present the abstract results on which our hyper-normalization proof is based. Section 3 contains the new proof that the leftmost outermost strategy is hyper-normalizing for left-normal weakly orthogonal TRSs. This result is extended to basic hyper-normalization in Section 4, where we also report on our experiments. In Section 5, we discuss related work. Finally, in Section 6, we conclude, where we also discuss future work.

2 Preliminaries

We assume familiarity with term rewriting and all that (e.g., [3]) and only shortly recall notions that are used in the following.

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 rules of $\mathcal{R}$ without common variables, a position $p \in \mathcal{P}\mathsf{os}_\mathcal{F}(\ell_2)$, 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 each other. The pair $((\ell_2\mu)[r_1\mu]_p, r_2\mu)$ is called a critical pair of $\mathcal{R}$. A pair (s,t) is trivial if $s = t$. Left-linear TRSs without critical pairs are called *orthogonal*. A left-linear TRS is *weakly orthogonal* if all critical pairs are trivial.

▶ **Definition 1.** The relations $<_\mathsf{l}$ and $<_\mathsf{lo}$ on positions is inductively defined as follows:

$$\frac{i < j}{i\,p <_\mathsf{l} j\,q} \qquad \frac{p <_\mathsf{l} q}{i\,p <_\mathsf{l} i\,q} \qquad \frac{p \neq \epsilon}{\epsilon <_\mathsf{lo} p} \qquad \frac{i < j}{i\,p <_\mathsf{lo} j\,q} \qquad \frac{p <_\mathsf{lo} q}{i\,p <_\mathsf{lo} i\,q}$$

Here i and j are positive integers and p and q are positions. Distinct positions are called *parallel* if they are comparable with $<_\mathsf{l}$. For a set of parallel positions Q we write $p <_\mathsf{lo} Q$ if $p <_\mathsf{lo} q$ for all $q \in Q$.

We have $p <_\mathsf{l} q$ if and only if position p is to the left of q. It is easy to see that $<_\mathsf{lo}$ is a total order on positions. It is the union of $<_\mathsf{l}$ and the standard prefix order $<$. So $p <_\mathsf{lo} q$ if and only if p is to the left of q or p is strictly above q.

▶ **Definition 2.** The rewrite step that contracts a redex at a position p is denoted by $\to_p$. We write $\mathsf{lo}(t)$ for the smallest redex position with respect to $<_\mathsf{lo}$. A redex at position $\mathsf{lo}(t)$ is called *leftmost-outermost* and we write $t \xrightarrow{\mathsf{lo}} u$ for $t \to_{\mathsf{lo}(t)} u$.

▶ **Definition 3.** A term t is called *left-normal* if function symbols precede variables when t is written in prefix notation. Formally, if $p \in \mathcal{P}\mathsf{os}_\mathcal{V}(t)$ and $q \in \mathcal{P}\mathsf{os}(t)$ with $p <_\mathsf{lo} q$ then $q \in \mathcal{P}\mathsf{os}_\mathcal{V}(t)$. A TRS $\mathcal{R}$ is *left-normal* if the left-hand side ℓ of every rule $\ell \to r \in \mathcal{R}$ is a left-normal term.

▶ **Definition 4.** Parallel rewriting is inductively defined by the following three clauses:

- $t \mathrel{+\!\!\!\!\!\to}_\varnothing t$ for every term t,
- $\ell\sigma \mathrel{+\!\!\!\!\!\to}_{\{\epsilon\}} r\sigma$ for every rewrite rule $\ell \to r$ and substitution σ,
- $f(s_1,\dots,s_n) \mathrel{+\!\!\!\!\!\to}_P f(t_1,\dots,t_n)$ if $s_i \mathrel{+\!\!\!\!\!\to}_{P_i} t_i$ for all $1 \leqslant i \leqslant n$ and $P = \{i\,p \mid 1 \leqslant i \leqslant n$ and $p \in P_i\}$.

We write $s \mathrel{+\!\!\!\!\!\to} t$ if $s \mathrel{+\!\!\!\!\!\to}_P t$ for some set of positions P.

We conclude this section by stating the abstract results that will be used in Section 3. We deal with abstract rewrite systems (ARSs) $\langle A, \to\rangle$ whose relation $\to$ is decomposed into two, not necessarily disjoint, parts $\to_\alpha$ and $\to_\beta$. In order to reduce the number of arrows, we denote the individual relations $\to_\alpha$ and $\to_\beta$ of such an ARS $\langle A, \{\to_\alpha, \to_\beta\}\rangle$ simply by α and β.

Let $\mathcal{A} = \langle A, \{\alpha, \beta\}\rangle$ be an ARS. We say that α *quasi-commutes* over β if the inclusion $\to_\beta \cdot \to_\alpha \subseteq \to_\alpha \cdot \to^*$ holds. The following easy result was first stated in [12].

▶ **Lemma 5.** *Let $\mathcal{A} = \langle A, \{\alpha, \beta\}\rangle$ be an ARS. If $\to_\beta \cdot \to_\alpha \subseteq \to_\alpha^* \cdot \to_\beta^=$ then $\to^* \subseteq \to_\alpha^* \cdot \to_\beta^*$.*

Proof. From the assumption we obtain $\to_\beta \cdot \to_\alpha^* \subseteq \to_\alpha^* \cdot \to_\beta^=$ by a straightforward induction proof. We show $\to^n \subseteq \to_\alpha^* \cdot \to_\beta^*$ by induction on $n \geqslant 0$. If $n = 0$ the inclusion holds trivially. Suppose $a \to b \to^n c$. The induction hypothesis yields $a \to b \to_\alpha^* \cdot \to_\beta^* c$. We distinguish two cases. If $a \to_\alpha b$ then $a \to_\alpha^* \cdot \to_\beta^* c$ holds without further ado. If $a \to_\beta b$ then we obtain $a \to_\alpha^* \cdot \to_\beta^= \cdot \to_\beta^* c$ from the strengthened assumption and thus also $a \to_\alpha^* \cdot \to_\beta^* c$. ◀

The following result is due to Bachmair and Dershowitz [4]. Here α/β denotes the relation $\to_\beta^* \cdot \to_\alpha \cdot \to_\beta^*$ and α/β-termination is perhaps better known as the termination of α relative to β.

▶ **Lemma 6.** *Let $\mathcal{A} = \langle A, \{\alpha, \beta\}\rangle$ be an ARS. If α quasi-commutes over β then every α-terminating element is α/β-terminating.*

Proof. From the quasi-commutation assumption we obtain $\to_\beta^* \cdot \to_\alpha \subseteq \to_\alpha \cdot \to^*$ by a straightforward induction argument. So α quasi-commutes over β^*. We prove that every α-terminating element $a \in A$ is α/β-terminating by well-founded induction on the restriction of $\to_\alpha$ to α-terminating elements, which is a well-founded relation. If $a \in \mathsf{NF}(\alpha/\beta)$ then the claim is trivial. Consider an arbitrary step $a \to_{\alpha/\beta} b$, i.e., $a \to_\beta^* \cdot \to_\alpha \cdot \to_\beta^* b$. Using the quasi-commutation of α over β, the latter sequence can be written as $a \to_\alpha a' \to^* b$. The element a' is α-terminating because a is α-terminating and $a \to_\alpha a'$. Hence the induction hypothesis yields that a' is α/β-terminating. Since $\to^* = \to_\beta^* \cup \to_{\alpha/\beta}^*$ and α/β-terminating elements are preserved under $\to_\beta$, it follows that b is α/β-terminating. Because this holds for any step $a \to_{\alpha/\beta} b$, element a is α/β-terminating. ◀

A rewrite strategy $\mathcal{S}$ for an ARS $\mathcal{A} = \langle A, \to_\mathcal{A}\rangle$ is a relation $\to_\mathcal{S}$ such that $\to_\mathcal{S} \subseteq \to_\mathcal{A}^+$ and $\mathsf{NF}(\to_\mathcal{S}) = \mathsf{NF}(\mathcal{A})$. A *one-step* strategy $\mathcal{S}$ satisfies $\to_\mathcal{S} \subseteq \to_\mathcal{A}$. We say that a strategy $\mathcal{S}$ is *deterministic* if $a = b$ whenever $a \;{}_\mathcal{S}{\leftarrow} \cdot \to_\mathcal{S} b$. A rewrite strategy $\mathcal{S}$ for an ARS $\mathcal{A}$ is *normalizing* if every $\mathcal{A}$-normalizing element is $\mathcal{S}$-terminating. Here an element a is $\mathcal{A}$-normalizing if $a \to_\mathcal{A}^* b$ for some $b \in \mathsf{NF}(\mathcal{A})$. We call $\mathcal{S}$ *hyper-normalizing* if every $\mathcal{A}$-normalizing element is $\mathcal{S}/\mathcal{A}$-terminating. Normalization is the property that by repeatedly performing steps according the strategy a normal form will be computed, provided the starting term has a normal form. Hyper-normalization is a much stronger property. It guarantees that normal forms will still be computed even if between successive strategy steps arbitrary but finitely many other steps are performed.

▶ **Lemma 7.** *A deterministic rewrite strategy $\mathcal{S}$ for an ARS $\mathcal{A}$ is normalizing if there is an ARS $\mathcal{B}$ with $\to_\mathcal{A}^* \subseteq \to_\mathcal{S}^* \cdot \to_\mathcal{B}^*$ and $\mathsf{NF}(\mathcal{A}) \subseteq \mathsf{NF}(\mathcal{B}^{-1})$.*

Proof. If $a \to_\mathcal{A}^! b$ then $a \to_\mathcal{S}^* c \to_\mathcal{B}^* b$ for some c. Because $b \in \mathsf{NF}(\mathcal{A}) \subseteq \mathsf{NF}(\mathcal{B}^{-1})$, $c = b$ and thus $a \to_\mathcal{S}^! b$. Since $\mathcal{S}$ is deterministic, it is terminating on a. ◀

▶ **Theorem 8.** *A normalizing rewrite strategy $\mathcal{S}$ for an ARS $\mathcal{A}$ is hyper-normalizing if $\mathcal{S}$ quasi-commutes over $\mathcal{A}$.*

Proof. Since every $\mathcal{S}$-terminating element is $\mathcal{S}/\mathcal{A}$-terminating according to Lemma 6, the result follows from the definitions of normalization and hyper-normalization. ◀

3 Left-Normal Weakly Orthogonal Rewrite Systems

In this section we present a simple proof of the hyper-normalization of the leftmost outermost strategy for the class of left-normal weakly orthogonal TRSs. The result is well-known and different proofs can be found in the literature, especially for Combinatory Logic and left-normal orthogonal TRSs, cf. the remarks on related work in Section 5. We give full proof details in order to ease future certification efforts. Let $\mathcal{R}$ be a TRS and let $\to$ denote the induced rewrite relation.

▶ **Definition 9.** A position p in t *overlaps with* q (from above) if there are a rule $\ell \to r \in \mathcal{R}$ and $p' \in \mathcal{P}\mathsf{os}_{\mathcal{F}}(\ell)$ such that $t|_p$ is an instance of ℓ and $q = pp'$.

The key to our proof is the following restriction of parallel rewriting.

▶ **Definition 10.** We write $s \xrightarrow{\neg\mathsf{lo}}\!\!\!\!\!\!\!\!\mathrel{+\!\!\!\!\!\!\longrightarrow} t$ if $s \mathrel{+\!\!\!\!\!\!\longrightarrow}_P t$ such that $\mathsf{lo}(s)$ overlaps with none of the positions in P. If P is a singleton set, we may write $s \xrightarrow{\neg\mathsf{lo}} t$.

For *orthogonal* TRSs we have $s \xrightarrow{\neg\mathsf{lo}}\!\!\!\!\!\!\!\!\mathrel{+\!\!\!\!\!\!\longrightarrow} t$ if and only if $s \mathrel{+\!\!\!\!\!\!\longrightarrow}_P t$ with $\mathsf{lo}(s) \notin P$. We start our analysis with a number of results that do not rely on left-normality.

The following lemma is obvious from the definition of weak orthogonality, and will be used silently throughout the remainder of this section.

▶ **Lemma 11.** *If $s \to_p t$, $s \to_q u$, and p overlaps with q then $t = u$.*

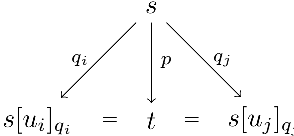
▶ **Lemma 12.** *The identity $\to \;=\; \xrightarrow{\mathsf{lo}} \cup \xrightarrow{\neg\mathsf{lo}}$ holds for every weakly orthogonal TRS.*

Proof. The inclusion from right to left is obvious. Suppose $s \to_p t$. If $p = \mathsf{lo}(s)$ then $s \xrightarrow{\mathsf{lo}} t$ by definition. If $\mathsf{lo}(s)$ overlaps with p then $s \to_{\mathsf{lo}(s)} t$ by weak orthogonality and thus also $s \xrightarrow{\mathsf{lo}} t$. In the remaining case we have $s \xrightarrow{\neg\mathsf{lo}} t$ by definition. ◀

The next result is essentially due to Takahashi [22].

▶ **Lemma 13.** *Suppose $s \to_p t$ and $s \mathrel{+\!\!\!\!\!\!\longrightarrow}_Q u$ such that p overlaps with all positions in Q. If $|Q| \geqslant 2$ then $s = t = u$.*

Proof. Let $Q = \{q_1, \ldots, q_n\}$ with $n \geqslant 2$. For $1 \leqslant i \leqslant n$ we denote the subterm of u at position q_i by u_i. So $s \to_{q_i} s[u_i]_{q_i}$. Now for $i \neq j$ we obtain



Since q_i and q_j are parallel, we have $u_i = (s[u_i]_{q_i})|_{q_i} = (s[u_j]_{q_j})|_{q_i} = s|_{q_i}$. Consequently $s \to_{q_i} s[u_i]_{q_i} = s$ and thus $s = t$. Since this holds for all $1 \leqslant i \leqslant n$ we obtain $u = s$. ◀

▶ **Lemma 14.** *The inclusion $\mathrel{+\!\!\!\!\!\!\longrightarrow} \;\subseteq\; \xrightarrow{\mathsf{lo}}{}^{*} \cdot \xrightarrow{\neg\mathsf{lo}}\!\!\!\!\!\!\!\!\mathrel{+\!\!\!\!\!\!\longrightarrow}$ holds for every weakly orthogonal TRS.*

Proof. Suppose $s \mathrel{+\!\!\!\!\!\!\longrightarrow}_P t$. We use induction on $|P|$. Let $P_1 = \{p \in P \mid \mathsf{lo}(s) \text{ overlaps with } p\}$ and $P_2 = P \setminus P_1$. There exists a term u such that $s \mathrel{+\!\!\!\!\!\!\longrightarrow}_{P_1} u \mathrel{+\!\!\!\!\!\!\longrightarrow}_{P_2} t$. If $P_1 = \varnothing$ then $s = u$ and if $|P_1| \geqslant 2$ then $s = u$ follows from Lemma 13, and thus we have $s \xrightarrow{\neg\mathsf{lo}}\!\!\!\!\!\!\!\!\mathrel{+\!\!\!\!\!\!\longrightarrow}_{P_2} t$. If $|P_1| = 1$ then we obtain $s \xrightarrow{\mathsf{lo}} u$ from weak orthogonality. The induction hypothesis yields $u \xrightarrow{\mathsf{lo}}{}^{*} \cdot \xrightarrow{\neg\mathsf{lo}}\!\!\!\!\!\!\!\!\mathrel{+\!\!\!\!\!\!\longrightarrow} t$ and thus also $s \xrightarrow{\mathsf{lo}}{}^{*} \cdot \xrightarrow{\neg\mathsf{lo}}\!\!\!\!\!\!\!\!\mathrel{+\!\!\!\!\!\!\longrightarrow} t$. ◀

▶ **Lemma 15.** *The leftmost outermost strategy is deterministic for every weakly orthogonal TRS.*

Proof. Since the leftmost outermost redex position is unique in any reducible term, the statement holds for any TRS without non-trivial overlays. In particular, it holds for every weakly orthogonal TRS. ◀

Because of the inclusion $\twoheadrightarrow\!\!\!\!\!+ \;\subseteq\; \to^*$, which holds for arbitrary TRSs, every parallel rewrite step can be serialized into a sequence of rewrite steps. For orthogonal TRSs it can be shown that every $\xrightarrow{\neg\mathsf{lo}}\!\!\!\!\!\!+$ step can be serialized into a sequence of $\xrightarrow{\neg\mathsf{lo}}$ steps. However, the following (original) example shows that serialization of $\xrightarrow{\neg\mathsf{lo}}\!\!\!\!\!\!+$ does not extend to weakly orthogonal TRSs.

▶ **Example 16.** Consider the weakly-orthogonal TRS $\mathcal{R}$ consisting of the four rewrite rules

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

$$\mathsf{g}(x,y) \to \mathsf{g}(\mathsf{b},\mathsf{b}) \qquad \mathsf{f}(\mathsf{g}(\mathsf{b},\mathsf{a})) \to \mathsf{f}(\mathsf{g}(\mathsf{b},\mathsf{b}))$$

Note that $\mathcal{R}$ is left-normal. Let $s = \mathsf{f}(\mathsf{g}(\mathsf{a},\mathsf{a}))$ and $t = \mathsf{f}(\mathsf{g}(\mathsf{b},\mathsf{b}))$. We have $\mathsf{lo}(s) = 1$ and $s \xrightarrow{\neg\mathsf{lo}}\!\!\!\!\!\!+\; t$ since $s \twoheadrightarrow\!\!\!\!\!+_{\{11,12\}} t$ and position 1 does not overlap with 11 or 12 in s. From s we can perform two different $\xrightarrow{\neg\mathsf{lo}}$ steps:

$$s \xrightarrow{\neg\mathsf{lo}} \mathsf{f}(\mathsf{g}(\mathsf{a},\mathsf{b})) \qquad s \xrightarrow{\neg\mathsf{lo}} \mathsf{f}(\mathsf{g}(\mathsf{b},\mathsf{a}))$$

In both cases we obtain a term which is in normal form with respect to $\xrightarrow{\neg\mathsf{lo}}$ because the created redex at the root position equals a left-hand side.

For the results that follow we need the restriction to *left-normal* weakly orthogonal TRSs.

▶ **Definition 17.** A position p is said to be *below* a term t if $p \geqslant q$ for some $q \in \mathcal{P}\mathsf{os}_{\mathcal{V}}(t)$.

▶ **Lemma 18.** *Let t be a left-normal term and σ a substitution. If $p, q \in \mathcal{P}\mathsf{os}(t\sigma)$ such that p is below t and $p <_{\mathsf{lo}} q$ then q is below t.*

Proof. We have $p \geqslant q'$ for some position $q' \in \mathcal{P}\mathsf{os}_{\mathcal{V}}(t)$. If $q \geqslant q'$ then q is below t. If $q \geqslant q'$ does not hold then we must have $p <_{\mathsf{l}} q$ and $q' <_{\mathsf{l}} q$. If $q \in \mathcal{P}\mathsf{os}(t)$ then $q \in \mathcal{P}\mathsf{os}_{\mathcal{V}}(t)$ by left-normality. Otherwise $q > q''$ for some position $q'' \in \mathcal{P}\mathsf{os}_{\mathcal{V}}(t)$. In both cases we conclude that q is below t. ◀

The next result can be viewed as a special case of the Parallel Moves Lemma. Although the statement appears plausible, the proof is subtle because of weak orthogonality. Below, we denote $i \cdot P$ by $\{i\,p \mid p \in P\}$, and write $\sigma \twoheadrightarrow\!\!\!\!\!+ \tau\ [X]$ if $x\sigma \twoheadrightarrow\!\!\!\!\!+ x\tau$ holds for all $x \in X$.

▶ **Lemma 19.** *If $\mathcal{R}$ is a left-normal weakly orthogonal TRS then $\xleftarrow{\mathsf{lo}} \cdot \xrightarrow{\neg\mathsf{lo}}\!\!\!\!\!\!+ \;\subseteq\; \twoheadrightarrow\!\!\!\!\!+ \cdot \xleftarrow{\mathsf{lo}}$.*

Proof. Let $s \xrightarrow{\neg\mathsf{lo}}\!\!\!\!\!\!+_Q\; t$ and $s \xrightarrow{\mathsf{lo}}_p u$. By induction on the sum $\|Q\|$ of the lengths of the positions in Q we show the existence of a term v such that $t \xrightarrow{\mathsf{lo}} v$ and $u \twoheadrightarrow\!\!\!\!\!+ v$. If $Q = \varnothing$ then $s = t$ and we simply take $u = v$. If $\epsilon \in Q$ then $Q = \{\epsilon\}$ and so there is nothing to show since the assumption $s \xrightarrow{\neg\mathsf{lo}}\!\!\!\!\!\!+_Q\; t$ is violated. In the remaining case we have $s = f(s_1,\ldots,s_n)$, $t = f(t_1,\ldots,t_n)$, $s_i \twoheadrightarrow\!\!\!\!\!+_{Q_i} t_i$ for all $1 \leqslant i \leqslant n$, and $Q = \{i\,q \mid 1 \leqslant i \leqslant n \text{ and } q \in Q_i\}$. We distinguish two further cases, depending on the position p.

$$p = j \cdot p_j \qquad p <_{\mathsf{lo}} j \cdot Q_j \qquad p' = j \cdot \mathsf{lo}(t_j) \qquad p' <_{\mathsf{l}} Q' \qquad p' <_{\mathsf{lo}} j \cdot Q_j$$

$$s' = f(s_1, \ldots, s_{j-1}, t_j, s_{j+1}, \ldots, s_n) \qquad u' = f(s_1, \ldots, s_{j-1}, v_j, s_{j+1}, \ldots, s_n)$$

$$u = f(s_1, \ldots, s_{j-1}, u_j, s_{j+1}, \ldots, s_n) \qquad v = f(s_1, \ldots, s_{j-1}, v_j, t_{j+1}, \ldots, t_n)$$

Figure 1 The critical case in the proof of Lemma 19.

- If $p = \epsilon$ then there exist a rewrite rule $\ell \to r$ and a substitution σ such that $s = \ell\sigma$ and $u = r\sigma$. Since the root symbol of s is f, we may write $\ell = f(\ell_1, \ldots, \ell_n)$. Fix $i \in \{1, \ldots, n\}$. We have $s_i = \ell_i\sigma \mathrel{+\!\!\!\!\!\!\to} t_i$. Since $\mathsf{lo}(s) = \epsilon$ does not overlap with positions in Q, all steps in $s \xrightarrow{\neg\mathsf{lo}}_{Q} t$ take place in the substitution σ. Using the linearity of ℓ, it is not difficult to prove the existence of a substitution τ_i such that $t_i = \ell_i\tau_i$ and $\sigma \mathrel{+\!\!\!\!\!\!\to} \tau_i$ $[\mathcal{V}\mathsf{ar}(\ell_i)]$. We assume without loss of generality that $\mathsf{dom}(\tau_i) \subseteq \mathcal{V}\mathsf{ar}(\ell_i)$. Otherwise, we can always consider the restriction of τ_i to $\mathcal{V}\mathsf{ar}(\ell_i)$ due to the linearity of ℓ. Thus the substitution $\tau = \tau_1 \cup \cdots \cup \tau_n$ is well-defined and satisfies $\ell\tau = t$ and $\sigma \mathrel{+\!\!\!\!\!\!\to} \tau$ $[\mathcal{V}\mathsf{ar}(\ell)]$. Let $v = r\tau$. Since parallel rewriting is closed under substitutions, we obtain $u = r\sigma \mathrel{+\!\!\!\!\!\!\to} r\tau = v$. Furthermore, we obviously have $t = \ell\tau \to_p r\tau = v$ with $\mathsf{lo}(t) = \epsilon = p$.
- If $p \neq \epsilon$ then $p = j\,p_j$ for some $1 \leqslant j \leqslant n$ and position $p_j \in \mathcal{P}\mathsf{os}(s_j)$. This case is illustrated in Figure 1. We have $\|Q_j\| < \|Q\|$ and $Q_i = \varnothing$ for all $1 \leqslant i < j$ because $p <_{\mathsf{lo}} Q$. Moreover, $s_j \xrightarrow{\neg\mathsf{lo}}_{Q_j} t_j$ follows from $s \xrightarrow{\neg\mathsf{lo}}_{Q} t$. Hence we can apply the induction hypothesis, yielding a term v_j such that $t_j \xrightarrow{\mathsf{lo}} v_j$ and $u_j \mathrel{+\!\!\!\!\!\!\to} v_j$. Let $s' = s[t_j]_j$, $u' = s[v_j]_j$, and $p' = j \cdot \mathsf{lo}(t_j)$. We have $u \mathrel{+\!\!\!\!\!\!\to} u'$, $s' \to_{p'} u'$, and $s' \mathrel{+\!\!\!\!\!\!\to}_{Q'} t$ with $Q' = Q \setminus Q_j = \{i\,q \mid j < i \leqslant n \text{ and } q \in Q_i\}$. Obviously, $p' <_{\mathsf{l}} Q'$ and thus there exists a term v such that $u' \mathrel{+\!\!\!\!\!\!\to}_{Q'} v$ and $t \to_{p'} v$. The parallel steps $u \mathrel{+\!\!\!\!\!\!\to} u'$ and $u' \mathrel{+\!\!\!\!\!\!\to}_{Q'} v$ can be combined into a single parallel step $u \mathrel{+\!\!\!\!\!\!\to} v$ because the redexes contracted in $u \mathrel{+\!\!\!\!\!\!\to} u'$ are below position j and thus to the left of all positions in Q'. It remains to show that $t \xrightarrow{\mathsf{lo}} v$. This is obvious if $p' = \mathsf{lo}(t)$. So suppose $p' \neq \mathsf{lo}(t)$, which implies $\mathsf{lo}(t) = \epsilon$. So $t = \ell\sigma$ for some rewrite rule $\ell = f(\ell_1, \ldots, \ell_n) \to r$ and substitution σ. We distinguish two further cases.
  - If p' is not below ℓ then we obtain $t \xrightarrow{\mathsf{lo}} v$ from weak orthogonality.
  - If p' is below ℓ then, since $p' <_{\mathsf{l}} Q'$, all positions in Q' are below ℓ according to Lemma 18. Since

    $$p' = j \cdot \mathsf{lo}(t_j) \leqslant j \cdot p_j = p \leqslant_{\mathsf{lo}} j \cdot Q_j$$

    the same holds for the positions in $j \cdot Q_j$. It follows that s is an instance of ℓ, contradicting $\mathsf{lo}(s) \neq \epsilon$. ◀

The following example shows the necessity of left-normality in Lemma 19.

▶ **Example 20.** Consider the orthogonal TRS consisting of the rewrite rules

$$\mathsf{a} \to \mathsf{b} \qquad\qquad \mathsf{f}(x, \mathsf{b}) \to \mathsf{c}$$

and the term $s = \mathsf{f}(\mathsf{a}, \mathsf{a})$. We have $s \xrightarrow{\mathsf{lo}} \mathsf{f}(\mathsf{b}, \mathsf{a})$ and $s \xrightarrow{\neg\mathsf{lo}} \mathsf{f}(\mathsf{a}, \mathsf{b})$ but there is no term t such that $\mathsf{f}(\mathsf{b}, \mathsf{a}) \mathrel{\mathrlap{\;+\!\!+}\longrightarrow} t$ and $\mathsf{f}(\mathsf{a}, \mathsf{b}) \xrightarrow{\mathsf{lo}} t$.

▶ **Lemma 21.** *The inclusion* $\overset{\neg\mathsf{lo}}{\mathrlap{\;+\!\!+}\longrightarrow} \subseteq \xrightarrow{\mathsf{lo}} \cdot \mathrel{\mathrlap{\;+\!\!+}\longrightarrow} \cdot \xleftarrow{\mathsf{lo}} \cup =$ *holds for every left-normal weakly orthogonal TRS.*

Proof. Let $s \overset{\neg\mathsf{lo}}{\mathrlap{\;+\!\!+}\longrightarrow}_P t$. If $P = \varnothing$ then $s = t$. If $P \neq \varnothing$ then s is reducible and thus $s \xrightarrow{\mathsf{lo}} u$ for some term u. We obtain $u \mathrel{\mathrlap{\;+\!\!+}\longrightarrow} \cdot \xleftarrow{\mathsf{lo}} t$ from Lemma 19 and thus $s \xrightarrow{\mathsf{lo}} \cdot \mathrel{\mathrlap{\;+\!\!+}\longrightarrow} \cdot \xleftarrow{\mathsf{lo}} t$ as desired. ◀

Combining Lemmata 14, 15 and 21 gives the following result.

▶ **Corollary 22.** *The inclusion* $\overset{\neg\mathsf{lo}}{\mathrlap{\;+\!\!+}\longrightarrow} \cdot \xrightarrow{\mathsf{lo}} \subseteq \xrightarrow{\mathsf{lo}}{}^{+} \cdot \overset{\neg\mathsf{lo}}{\mathrlap{\;+\!\!+}\longrightarrow}$ *holds for every left-normal weakly orthogonal TRS.*

Proof. We have

$$\begin{aligned}
\overset{\neg\mathsf{lo}}{\mathrlap{\;+\!\!+}\longrightarrow} \cdot \xrightarrow{\mathsf{lo}} &\subseteq (\xrightarrow{\mathsf{lo}} \cdot \mathrel{\mathrlap{\;+\!\!+}\longrightarrow} \cdot \xleftarrow{\mathsf{lo}} \cup =) \cdot \xrightarrow{\mathsf{lo}} && \text{(Lemma 21)}\\
&= \xrightarrow{\mathsf{lo}} \cdot \mathrel{\mathrlap{\;+\!\!+}\longrightarrow} \cdot \xleftarrow{\mathsf{lo}} \cdot \xrightarrow{\mathsf{lo}} \cup = \cdot \xrightarrow{\mathsf{lo}} \\
&\subseteq \xrightarrow{\mathsf{lo}} \cdot \mathrel{\mathrlap{\;+\!\!+}\longrightarrow} \cdot = \cup \xrightarrow{\mathsf{lo}} && \text{(Lemma 15)}\\
&= \xrightarrow{\mathsf{lo}} \cdot \mathrel{\mathrlap{\;+\!\!+}\longrightarrow} \\
&\subseteq \xrightarrow{\mathsf{lo}} \cdot \xrightarrow{\mathsf{lo}}{}^{*} \cdot \overset{\neg\mathsf{lo}}{\mathrlap{\;+\!\!+}\longrightarrow} && \text{(Lemma 14)}\\
&= \xrightarrow{\mathsf{lo}}{}^{+} \cdot \overset{\neg\mathsf{lo}}{\mathrlap{\;+\!\!+}\longrightarrow}
\end{aligned}$$

◀

▶ **Corollary 23.** *The relation* $\xrightarrow{\mathsf{lo}}$ *quasi-commutes over* $\overset{\neg\mathsf{lo}}{\mathrlap{\;+\!\!+}\longrightarrow}$ *for every left-normal weakly orthogonal TRS.*

Proof. This follows from the preceding corollary and the inclusion $\xrightarrow{\mathsf{lo}}{}^{+} \cdot \overset{\neg\mathsf{lo}}{\mathrlap{\;+\!\!+}\longrightarrow} \subseteq \xrightarrow{\mathsf{lo}} \cdot \to^{*}$. ◀

▶ **Corollary 24.** *The inclusion* $\to^{*} \subseteq \xrightarrow{\mathsf{lo}}{}^{*} \cdot \overset{\neg\mathsf{lo}}{\mathrlap{\;+\!\!+}\longrightarrow}{}^{*}$ *holds for every left-normal weakly orthogonal TRS.*

Proof. This follows from Corollary 22 in connection with Lemmata 12 and 5 (with $\to_\alpha = \xrightarrow{\mathsf{lo}}$ and $\to_\beta = \overset{\neg\mathsf{lo}}{\mathrlap{\;+\!\!+}\longrightarrow}$). ◀

We arrive at the (hyper-)normalization theorem.

▶ **Theorem 25.** *The leftmost outermost strategy is hyper-normalizing for every left-normal weakly orthogonal TRS.*

Proof. Normalization of $\xrightarrow{\mathsf{lo}}$ is obtained from Lemma 7, Corollary 24, and the inclusion $\mathsf{NF}(\mathcal{R}) \subseteq \mathsf{NF}(\xleftarrow{\neg\mathsf{lo}})$ which follows from Lemma 19: If $t \notin \mathsf{NF}(\xleftarrow{\neg\mathsf{lo}})$ then $s \xrightarrow{\neg\mathsf{lo}} t$ for some term s and thus $s \xrightarrow{\mathsf{lo}} u$ for some term u and hence $u \mathrel{\mathrlap{\;+\!\!+}\longrightarrow} \cdot \xleftarrow{\mathsf{lo}} t$, so $t \notin \mathsf{NF}(\mathcal{R})$. By combining the normalization of $\xrightarrow{\mathsf{lo}}$ with Theorem 8 and Corollary 23, hyper-normalization of $\xrightarrow{\mathsf{lo}}$ is concluded. ◀

4 Basic Normalization

We recall a few notions from context-sensitive rewriting.

▶ **Definition 26.** A *replacement map* associates every n-ary function symbol to a subset of $\{1,\dots,n\}$. Let μ be a replacement map. The set $\mathcal{P}\mathsf{os}^\mu(t)$ of *active* positions in t is inductively defined as follows:

$$\mathcal{P}\mathsf{os}^\mu(t) = \begin{cases} \{\epsilon\} & \text{if } t \text{ is a variable} \\ \{\epsilon\} \cup \{ip \mid i \in \mu(f),\ 1 \leqslant i \leqslant n, \text{ and } p \in \mathcal{P}\mathsf{os}^\mu(t_i)\} & \text{if } t = f(t_1,\dots,t_n) \end{cases}$$

The set $\mathcal{P}\mathsf{os}_\mathcal{V}(t) \cap \mathcal{P}\mathsf{os}^\mu(t)$ is abbreviated to $\mathcal{P}\mathsf{os}^\mu_\mathcal{V}(t)$.

We introduce basic normalization. Let $\mathcal{R}$ be a TRS and $\mathcal{D} = \{\mathsf{root}(\ell) \mid \ell \to r \in \mathcal{R}\}$ be the set of all defined symbols in $\mathcal{R}$. Terms t with $\mathcal{P}\mathsf{os}_\mathcal{D}(t) = \{\epsilon\}$ are called *basic terms*.

▶ **Definition 27.** A rewrite strategy $\mathcal{S}$ for a TRS $\mathcal{R}$ is *basically normalizing* if every $\mathcal{R}$-normalizing basic term is $\mathcal{S}$-terminating.

By recasting the basic term condition of basic normalization in a replacement map for context-sensitive rewriting, we establish a powerful criterion for basic normalization. As a basis of this formulation, we use usable replacement maps [10, 14].

▶ **Definition 28.** A term t is *accessible* if $s \to^* t$ for some basic term s. A replacement map μ for a TRS $\mathcal{R}$ is *usable* if all redex positions in every accessible term t are included in $\mathcal{P}\mathsf{os}^\mu(t)$.

An effective technique for computation of usable replacement maps based on unification and fixed point computation is suggested in [14]. The method adapts the cap-function ICAP suitably [11]. The key observation is that through the cap-function usable arguments can be delineated which is formalizable as a monotone operator $\Upsilon^\mathcal{R}$. Applying Tarski's fixed point theorem we conclude the existence of a least fixed point which yields the desired approximation, cf. [14, Definition 9].

We define basically left-normal TRSs. The notion of μ-left-normality in the following definition was introduced in [18] for normalization of context-sensitive rewriting.

▶ **Definition 29.** A term t is called left-normal with respect to a replacement map μ, or simply *μ-left-normal*, if $p \in \mathcal{P}\mathsf{os}^\mu_\mathcal{V}(t)$ and $q \in \mathcal{P}\mathsf{os}^\mu(t)$ with $p <_{\mathsf{lo}} q$ imply $q \in \mathcal{P}\mathsf{os}_\mathcal{V}(t)$. Let $\mathcal{R}$ be a TRS with a usable replacement map μ. The TRS $\mathcal{R}$ is *basically left-normal* if the left-hand side ℓ of every rule $\ell \to r \in \mathcal{R}$ is μ-left-normal.

▶ **Example 30.** Consider the orthogonal TRS in the introduction. The map μ given by

$$\mu(\mathsf{sieve}) = \{1\} \qquad \mu(\mathsf{primes}) = \mu(\mathsf{from}) = \mu(\mathsf{s}) = \mu(\mathsf{0}) = \varnothing$$
$$\mu(\mathsf{filter}) = \mu(\mathsf{take}) = \mu(:) = \{2\}$$

is a usable replacement map. For example, consider $t = \mathsf{take}(\mathsf{s}(n), x : y)$. There are no active positions $p \in \mathcal{P}\mathsf{os}^\mu_\mathcal{V}(t) = \{22\}$ and $q \in \mathcal{P}\mathsf{os}^\mu(t) = \{\epsilon, 2, 22\}$ such that $p <_{\mathsf{lo}} q$. So t is μ-left-normal. In a similar way one can verify basic left-normality of all other left-hand sides of the TRS left-normal. Hence, the TRS is basic left-normal with respect to μ.

We show that the leftmost outermost strategy is basically normalizing for basically left-normal TRSs.

Table 1 Experimental results on 161 weakly orthogonal TRSs.

	left-normal	basically left-normal
# of TRSs	75	150
total time (in seconds)	0.63	1.03

▶ **Lemma 31.** *Let μ be a replacement map, t a μ-left-normal term, and σ a substitution. If $p, q \in \mathcal{P}\mathsf{os}^{\mu}(t\sigma)$ such that p is below t and $p <_{\mathsf{lo}} q$ then q is below t.*

Proof. The proof of Lemma 18 goes through after replacing $\mathcal{P}\mathsf{os}_{\mathcal{V}}(t)$ with $\mathcal{P}\mathsf{os}^{\mu}_{\mathcal{V}}(t)$: We have $p \geqslant q'$ for some position $q' \in \mathcal{P}\mathsf{os}_{\mathcal{V}}(t)$. Hence $q' \in \mathcal{P}\mathsf{os}^{\mu}_{\mathcal{V}}(t)$ follows from $p \in \mathcal{P}\mathsf{os}^{\mu}(t\sigma)$. If $q \geqslant q'$ then q is below t. If $q \geqslant q'$ does not hold then we must have $p <_{\mathsf{l}} q$ and $q' <_{\mathsf{l}} q$ for otherwise the assumption $p <_{\mathsf{lo}} q$ would be violated. If $q \in \mathcal{P}\mathsf{os}^{\mu}(t)$ then $q \in \mathcal{P}\mathsf{os}_{\mathcal{V}}(t)$ by μ-left-normality. Otherwise $q > q''$ for some position $q'' \in \mathcal{P}\mathsf{os}^{\mu}_{\mathcal{V}}(t) \subseteq \mathcal{P}\mathsf{os}_{\mathcal{V}}(t)$. In both cases we conclude that q is below t. ◀

Let $\rightsquigarrow$ be a relation on terms. The *accessible version* $\rightsquigarrow_{\mathsf{a}}$ is defined as follows: $s \rightsquigarrow_{\mathsf{a}} t$ if s is an accessible term and $s \rightsquigarrow t$. In Section 3 we used left-normality to show Lemma 19. The next lemma is its counterpart for basic hyper-normalization.

▶ **Lemma 32.** *Let $\mathcal{R}$ be a weakly orthogonal TRS with a usable replacement map μ. If $\mathcal{R}$ is basically left-normal then the inclusion ${}_{\mathsf{a}}\!\xleftarrow{\mathsf{lo}} \cdot \xrightarrow{\neg\mathsf{lo}}\!\!\!\!\!\!\!\!+\!\!\!\!+\;_{\mathsf{a}} \subseteq \;\twoheadrightarrow\!\!\!\!\!\!+\!\!+_{\mathsf{a}} \cdot {}_{\mathsf{a}}\!\xleftarrow{\mathsf{lo}}$ holds.*

Proof. Let $s \xrightarrow{\neg\mathsf{lo}}_{\mathsf{a}} t$ (multi-step) and $s \xrightarrow{\mathsf{lo}}_{\mathsf{a}} u$. Since s is accessible and μ is usable, $s \xrightarrow{\neg\mathsf{lo}}_{Q} t$ and $s \xrightarrow{\mathsf{lo}}_{p} u$ hold for some $Q \subseteq \mathcal{P}\mathsf{os}^{\mu}(s)$ and $p \in \mathcal{P}\mathsf{os}^{\mu}(s)$. We obtain $t \twoheadrightarrow\!\!\!\!\!\!+\!\!+ \cdot \xleftarrow{\mathsf{lo}} u$ as in the proof of Lemma 19, provided that Lemma 31 is used instead of Lemma 18. Since accessibility of s carries over to t and u, we conclude $t \twoheadrightarrow\!\!\!\!\!\!+\!\!+_{\mathsf{a}} \cdot {}_{\mathsf{a}}\!\xleftarrow{\mathsf{lo}} u$. ◀

By using Lemma 32, one can lift all statements (and proofs) in Lemma 21 and Corollaries 22, 23, and 24 to the accessible version in a straightforward way.

▶ **Theorem 33.** *The leftmost outermost strategy is basically hyper-normalizing for every basically left-normal weakly orthogonal TRS.* ◀

We implemented and tested Theorems 8 and 33 on a collection of 161 weakly orthogonal TRSs, consisting of the 153 systems for innermost, outermost, and context-sensitive rewriting in version 8.0.7 of the Termination Problem Data Base (TPDB)[1] and 9 systems from van de Pol's examples for strategy annotations [21].[2] Usable replacement maps for Theorem 33 are estimated by the fixed point computation of [14, Definition 9]. Table 1 summarizes the results.[3]

5 Related Work

The (hyper-)normalization of the leftmost outermost strategy is a fundamental result in Combinatory Logic and λ-calculus. The importance of hyper-normalization as opposed to

[1] http://termcomp.uibk.ac.at/
[2] http://wwwhome.cs.utwente.nl/~vdpol/jitty/
[3] Details are available at: http://www.jaist.ac.jp/~hirokawa/15bn/

normalization stems from the fact that this property is essential to show that all partial recursive functions are definable in λ-calculus and Combinatory Logic. Consequently, numerous (hyper-)normalization proofs can be found in the literature. Below we comment on some of them.

Combinatory Logic and λ-calculus. The usual argument that the leftmost outermost strategy is (hyper-)normalizing in Combinatory Logic or λ-calculus employs the standardization theorem [8, 5], which in itself is based on the study of residuals. Avoiding residuals, Kashima presents in [17] an interesting inductive treatment of standardization and thus provides a simple proof of hyper-normalization of the leftmost outermost strategy for the λ-calculus.

It is perhaps worthy of note that the proof of the (hyper-)normalization theorem is absent from the well-known textbook by Hindley and Seldin [13]. The other recent book covering Combinatory Logic by Bimbó [6, Lemma 2.2.8] contains the following short and *incomplete* proof of the normalization theorem (here $\rhd_1$ refers to the rewrite relation of Combinatory Logic, "lmrs" stands for the leftmost (outermost) strategy, and Theorem 2.1.14 is the Church-Rosser theorem):

> *If a CL-term has an nf, then there is a $\rhd_1$ reduction sequence of finite length. If the leftmost reduction sequence is a finite $\rhd_1$ reduction sequence, then by theorem 2.1.14, the last term is the nf of the starting term.*
> *The other possibility is that the lmrs gives us an infinite $\rhd_1$ reduction sequence. Therefore, if some redex other than the leftmost one is reduced in another $\rhd_1$ reduction sequence, then the leftmost redex remains in the term as long as the lmrs is not followed, i.e., the term will not reduce to its nf.* *(qed)*

Left-normal (orthogonal) term rewrite systems. O'Donnell [20] introduced left-normality and proved the normalization of the leftmost outermost strategy for left-normal orthogonal TRSs. A modern account of his proof, which is based on residual theory and cofinality of leftmost-fair reductions, can be found in [23, Section 4.9].

Hyper-normalization of the leftmost outermost strategy for left-normal orthogonal TRSs is obtained by van Oostrom and de Vrijer in [23, Theorem 9.3.21] as a corollary of the more general statement that the leftmost outermost strategy is a *needed* strategy for left-normal orthogonal TRSs.

Extending upon [24] Toyama proves in [25] that external reduction, which is a variation of needed reduction, is a normalizing strategy for the class of left-linear root balanced joinable external TRSs. Note that the studied TRSs may be ambiguous. As a corollary he obtains the hyper-normalization of the leftmost outermost strategy for the class of left-linear left-normal root balanced joinable TRSs. The latter class includes all left-normal weakly orthogonal TRSs. It is an open problem whether our proof method extends to left-linear left-normal root balanced joinable TRSs.

All these proofs can be characterized as global in the sense that definitions refer to properties of rewrite sequences rather than single (parallel) steps, which are manipulated throughout the proof. We believe this will hamper formalization efforts. In contrast, our proof is elementary and local, as it makes essential use of abstract commutation properties.

Commutation properties In [1] Accattoli introduces an abstract framework for factorization, relying on commutation properties similar to those exploited in Lemma 5 and 6 above. The framework is general in the sense that it applies to a multitude of explicit substitutions calculi. However, its applicability in our context is less straightforward. In order to employ

a *square factorization system* [1, Definition 3.3] to prove our Corollary 24, we would need to decompose the relations $\xrightarrow{\mathsf{lo}}$ and $\overset{\neg\mathsf{lo}}{\twoheadrightarrow}$ into four relations ($\xrightarrow{\mathsf{lo}}_1$, $\xrightarrow{\mathsf{lo}}_2$, $\overset{\neg\mathsf{lo}}{\twoheadrightarrow}_1$, $\overset{\neg\mathsf{lo}}{\twoheadrightarrow}_2$), such that $\xrightarrow{\mathsf{lo}}_1$ and $\overset{\neg\mathsf{lo}}{\twoheadrightarrow}_1$ are terminating; a non-trivial task.

Dershowitz argues in [9, Note 20] that quasi-commutation applies to Combinatory Logic and orthogonal TRSs, in connection with the inclusion $\xleftarrow{\mathsf{lo}} \cdot \xrightarrow{\neg\mathsf{lo}} \subseteq \xrightarrow{\neg\mathsf{lo}}{}^* \cdot \xleftarrow{\mathsf{lo}}$. Apart from the missing left-normality condition, the inclusion does not hold, not even for Combinatory Logic as $\mathsf{I}x \xleftarrow{\mathsf{lo}} \mathsf{II}x \xrightarrow{\neg\mathsf{lo}} \mathsf{I}x$ but not $\mathsf{I}x \xrightarrow{\neg\mathsf{lo}}{}^* \cdot \xleftarrow{\mathsf{lo}} \mathsf{I}x$.

6 Conclusion

In this paper we have presented an elementary proof of the classical result that the leftmost outermost strategy is normalizing for left-normal orthogonal rewrite systems. Our proof is local and extends to hyper-normalization and weakly orthogonal systems. Our interest in leftmost outermost stems from the observation that archetypical TRSs often fail the definition of left-normality, while morally these TRSs are left-normal, if the set of starting terms is suitable restricted.

Based on this observation, we introduced basic normalization, i.e., normalization if the set of considered starting terms is restricted to basic terms. This allowed us to weaken the left-normality restriction. Building upon our new proof, we have shown that the leftmost outermost strategy is hyper-normalizing for basically left-normal weakly orthogonal rewrite systems. This provides a simple and easy to implement strategy for basic terms in a surprisingly large number of cases, as evidenced by the experimental data provided.

Despite the technical challenges found in the generalization of our result to weakly orthogonal systems, we have striven for an elementary proof, which we believe offers itself to future formalization within an interactive theorem prover. Hopefully our results pave the way for future certification efforts in the area of strategies.

In future work we will pursue an experimental and theoretical comparison of our results with the normalizing strategies induced by strongly/inductively [16, 2] sequential TRSs. Furthermore, we seek a simple proof of (hyper-)normalization of maximal (parallel) outermost for weakly orthogonal TRSs (cf. [20, 19]).

Acknowledgements. We are grateful for the comments by the anonymous reviewers which greatly helped us to improve the presentation of the paper.

References

1 B. Accattoli. An abstract factorization theorem for explicit substitutions. In A. Tiwari, editor, *Proc. 23rd International Conference on Rewriting Techniques and Applications*, volume 15 of *LIPIcs*, pages 6–21. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2012. doi: 10.4230/LIPIcs.RTA.2012.6.

2 S. Antoy. Definitional trees. In H. Kirchner and G. Levi, editors, *Proc. 3rd International Conference on Algebraic and Logic Programming*, volume 632 of *LNCS*, pages 143–157. Springer, 1992. doi: 10.1007/BFb0013825.

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

4 L. Bachmair and N. Dershowitz. Commutation, transformation, and termination. In J.H. Siekmann, editor, *Proc. 8th International Conference on Automated Deduction*, volume 230 of *LNCS*, pages 5–20, 1986. doi: 10.1007/3-540-16780-3_76.

5. H.P. Barendregt. *The Lambda Calculus, its Syntax and Semantics*, volume 103 of *Studies in Logic*. North-Holland, 2nd edition, 1984.
6. K. Bimbó. *Combinatory Logic: Pure, Applied and Typed*. CRC Press, 2012.
7. H. Comon. Sequentiality, monadic second-order logic and tree automata. *Information and Computation*, 157(1-2):25–51, 2000. doi: 10.1006/inco.1999.2838.
8. H.B. Curry, J.R. Hindley, and J.P. Seldin. *Combinatory Logic, Volume II*, volume 65 of *Studies in Logic*. North-Holland, 1972.
9. N. Dershowitz. On lazy commutation. In O. Grumberg, M. Kaminski, S. Katz, and S. Wintner, editors, *Languages: From Formal to Natural: Essays Dedicated to Nissim Francez on the Occasion of His 65th Birthday*, volume 5533 of *LNCS*, pages 59–82. Springer, 2009. doi: 10.1007/978-3-642-01748-3_5.
10. M.-L. Fernández. Relaxing monotonicity for innermost termination. *Information Processing Letters*, 93(1):117–123, 2005. doi: 10.1016/j.ipl.2004.10.005.
11. J. Giesl, R. Thiemann, and P. Schneider-Kamp. Proving and disproving termination of higher-order functions. In B. Gramlich, editor, *Proc. 5th International Symposium on Frontiers of Combining Systems*, volume 3717 of *Lecture Notes in Artificial Intelligence*, pages 216–231. Springer, 2005. doi: 10.1007/11559306_12.
12. J.R. Hindley. *The Church-Rosser Property and a Result in Combinatory Logic*. PhD thesis, University of Newcastle-upon-Tyne, 1964.
13. J.R. Hindley and J.P. Seldin. *Lambda-Calculus and Combinators, an Introduction*. Cambridge University Press, 2008.
14. N. Hirokawa and G. Moser. Automated complexity analysis based on context-sensitive rewriting. In G. Dowek, editor, *Proc. Joint 25th International Conference on Rewriting Techniques and Applications and 12th International Conference on Typed Lambda Calculi and Applications*, volume 8560 of *LNCS (ARCoSS)*, pages 257–271. Springer, 2014. doi: 10.1007/978-3-319-08918-8_18.
15. G. Huet and J.-J. Lévy. Computations in orthogonal rewriting systems, I. In J.L. Lassez and G. Plotkin, editors, *Computational Logic – Essays in Honor of Alan Robinson*, pages 395–414. The MIT Press, 1991.
16. G. Huet and J.-J. Lévy. Computations in orthogonal rewriting systems, II. In J.L. Lassez and G. Plotkin, editors, *Computational Logic – Essays in Honor of Alan Robinson*, pages 415–443. The MIT Press, 1991.
17. R. Kashima. A proof of the standardization theorem in λ-calculus. Research Reports on Mathematical and Computing Sciences C-145, Tokyo Institute of Technology, 2000.
18. S. Lucas. Needed reductions with context-sensitive rewriting. In M. Hanus, J. Heering, and K. Meinke, editors, *Proc. 6th International Conference on Algebraic and Logic Programming*, volume 1298 of *LNCS*, pages 129–143, 1997. doi: 10.1007/BFb0027007.
19. V. van Oostrom. Normalisation in weakly orthogonal rewriting. In P. Narendran and M. Rusinowitch, editors, *Proc. 10th International Conference on Rewriting Techniques and Applications*, volume 1631 of *LNCS*, pages 60–74. Springer, 1999. doi: 10.1007/3-540-48685-2_5.
20. M.J. O'Donnell. *Computing in Systems Described by Equations*, volume 58 of *LNCS*. Springer, 1977. doi: 10.1007/3-540-08531-9.
21. J. van de Pol. JITty: A rewriter with strategy annotations. In S. Tison, editor, *Proc. 13th International Conference on Rewriting Techniques and Applications*, volume 2378 of *LNCS*, pages 367–370. Springer, 2002. doi: 10.1007/3-540-45610-4_26.
22. M. Takahashi. Lambda-calculi with conditional rules. In M. Bezem and J.F. Groote, editors, *Proc. International Conference on Typed Lambda Calculi and Applications*, volume 664 of *LNCS*, pages 406–417. Springer, 1993. doi: 10.1007/BFb0037121.

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

24 Y. Toyama. Strong sequentiality of left-linear overlapping term rewriting systems. In *Proc. 7th Annual Symposium on Logic in Computer Science*, pages 274–284. IEEE Computer Society, 1992. doi: 10.1109/LICS.1992.185540.

25 Y. Toyama. Reduction strategies for left-linear term rewriting systems. In A. Middeldorp, V. van Oostrom, F. van Raamsdonk, and R.C. de Vrijer, editors, *Processes, Terms and Cycles: Steps on the Road to Infinity, Essays Dedicated to Jan Willem Klop, on the Occasion of his 60th Birthday*, volume 3838 of *LNCS*, pages 198–223, 2005. doi: 10.1007/11601548_13.

Conditional Complexity*

Cynthia Kop, Aart Middeldorp, and Thomas Sternagel

Institute of Computer Science
University of Innsbruck, Austria
{cynthia.kop|aart.middeldorp|thomas.sternagel}@uibk.ac.at

Abstract

We propose a notion of complexity for oriented conditional term rewrite systems. This notion is realistic in the sense that it measures not only successful computations but also partial computations that result in a failed rule application. A transformation to unconditional context-sensitive rewrite systems is presented which reflects this complexity notion, as well as a technique to derive runtime and derivational complexity bounds for the latter.

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

Keywords and phrases conditional term rewriting, complexity

Digital Object Identifier 10.4230/LIPIcs.RTA.2015.223

1 Introduction

Conditional term rewriting is a well-known computational paradigm. First studied in the eighties and early nineties of the previous century, in more recent years transformation techniques have received a lot of attention and automatic tools for (operational) termination [8, 16, 25] as well as confluence [27] were developed.

In this paper we are concerned with the following question: What is the length of a longest derivation to normal form in terms of the size of the starting term? For unconditional rewrite systems this question has been investigated extensively and numerous techniques have been developed that provide an upper bound on the resulting notions of derivational and runtime complexity (e.g. [5, 11, 12, 19, 20]). Tools that support complexity methods ([2, 22, 30]) are under active development and compete annually in the complexity competition.[1]

We are not aware of any techniques or tools for conditional (derivational and runtime) complexity—or indeed, even of a *definition* for conditional complexity. This may be for a good reason, as it is not obvious what such a definition should be. Of course, simply counting (top-level) steps will not do. Taking the conditions into account when counting successful rewrite steps is a natural idea and transformations from conditional term rewrite systems to unconditional ones exist (e.g., unravelings [24]) that do justice to this two-dimensional view [15, 16]. However, we will argue that this still gives rise to an unrealistic notion of complexity. Modern rewrite engines like Maude [6] that support conditional rewriting can spend significant resources on evaluating conditions that in the end prove to be useless for rewriting the term at hand. This should be taken into account when defining complexity.

Contribution. We propose a new notion of conditional complexity for a large class of reasonably well-behaved conditional term rewrite systems. This notion aims to capture the maximal number of rewrite steps that can be performed when reducing a term to normal

* This research is supported by the Austrian Science Fund (FWF) project I963.
[1] http://cbr.uibk.ac.at/competition/

© Cynthia Kop, Aart Middeldorp, and Thomas Sternagel;
licensed under Creative Commons License CC-BY

form, including the steps that were computed but ultimately not useful. In order to reuse existing methodology, we present a transformation into unconditional rewrite systems that can be used to estimate the conditional complexity. The transformed system is context-sensitive (Lucas [13, 14]), which is not yet supported by current complexity tools, but ignoring the corresponding restrictions, we still obtain an upper bound on the conditional complexity.

Organization. The remainder of the paper is organized as follows. In the next section we recall some preliminaries. Based on the analysis of conditional complexity in Section 3, we introduce our new notion formally in Section 4. Section 5 presents a transformation to context-sensitive rewrite systems, and in Section 6 we present an interpretation-based method targeting the resulting systems. Even though we are far removed from tool support, examples are given to illustrate that manual computations are feasible. Related work is discussed in Section 7 before we conclude in Section 8 with suggestions for future work.

2 Preliminaries

We assume familiarity with (conditional) term rewriting and all that (e.g., [4, 28, 24]) and only shortly recall important notions that are used in the following.

In this paper we consider *oriented* conditional term rewrite systems (CTRSs for short). Given a CTRS $\mathcal{R}$, a substitution σ, and a list of conditions $c\colon s_1 \approx t_1, \dots, s_k \approx t_k$, let $\mathcal{R} \vdash c\sigma$ denote $s_i\sigma \to_{\mathcal{R}}^* t_i\sigma$ for all $1 \leqslant i \leqslant k$. We have $s \to_{\mathcal{R}} t$ if there exist a position p in s, a rule $\ell \to r \Leftarrow c$ in $\mathcal{R}$, and a substitution σ such that $s|_p = \ell\sigma$, $t = s[r\sigma]_p$, and $\mathcal{R} \vdash c\sigma$. We may write $s \xrightarrow{\epsilon} t$ for a rewrite step at the root position and $s \xrightarrow{>\epsilon} t$ for a non-root step.

Given a (C)TRS $\mathcal{R}$ over a signature $\mathcal{F}$, the root symbols of left-hand sides of rules in $\mathcal{R}$ are called *defined* and every other symbol in $\mathcal{F}$ is a *constructor*. These sets are denoted by $\mathcal{F}_{\mathcal{D}}$ and $\mathcal{F}_{\mathcal{C}}$, respectively. For a defined symbol f, we write $\mathcal{R}{\restriction}f$ for the set of rules in $\mathcal{R}$ that define f. A constructor term consists of constructors and variables. A basic term is a term $f(t_1, \dots, t_n)$ with $f \in \mathcal{F}_{\mathcal{D}}$ and constructor terms $t_1, \dots, t_n$.

Context-sensitive rewriting, as used in Section 5, restricts the positions in a term where rewriting is allowed. A (C)TRS is combined with a *replacement map* μ, which assigns to every n-ary symbol $f \in \mathcal{F}$ a subset $\mu(f) \subseteq \{1, \dots, n\}$. A position p is *active* in a term t if either $p = \epsilon$, or $p = i\,q$, $t = f(t_1, \dots, t_n)$, $i \in \mu(f)$, and q is active in t_i. The set of active positions in a term t is denoted by $\mathcal{P}\mathsf{os}_\mu(t)$, and t may only be reduced at active positions.

Given a terminating and finitely branching TRS $\mathcal{R}$ over a signature $\mathcal{F}$, the *derivation height* of a term t is defined as $\mathsf{dh}(t) = \max\{n \mid t \to^n u \text{ for some term } u\}$. This leads to the notion of *derivational complexity* $\mathsf{dc}_{\mathcal{R}}(n) = \max\{\mathsf{dh}(t) \mid |t| \leqslant n\}$. If we restrict the definition to basic terms t we get the notion of *runtime complexity* $\mathsf{rc}_{\mathcal{R}}(n)$ [10].

Rewrite rules $\ell \to r \Leftarrow c$ of CTRSs are classified according to the distribution of variables among ℓ, r, and c. In this paper we consider 3-CTRSs, where the rules satisfy $\mathcal{V}\mathsf{ar}(r) \subseteq \mathcal{V}\mathsf{ar}(\ell, c)$. A CTRS $\mathcal{R}$ is *deterministic* if for every rule $\ell \to r \Leftarrow s_1 \approx t_1, \dots, s_k \approx t_k$ in $\mathcal{R}$ we have $\mathcal{V}\mathsf{ar}(s_i) \subseteq \mathcal{V}\mathsf{ar}(\ell, t_1, \dots, t_{i-1})$ for $1 \leqslant i \leqslant k$. A deterministic 3-CTRS $\mathcal{R}$ is *quasi-decreasing* if there exists a well-founded order $>$ with the subterm property that extends $\to_{\mathcal{R}}$, such that $\ell\sigma > s_i\sigma$ for all $\ell \to r \Leftarrow s_1 \approx t_1, \dots, s_k \approx t_k \in \mathcal{R}$, $1 \leqslant i \leqslant k$, and substitutions σ with $s_j\sigma \to_{\mathcal{R}}^* t_j\sigma$ for $1 \leqslant j < i$. Quasi-decreasingness ensures termination and, for finite CTRSs, computability of the rewrite relation. Quasi-decreasingness coincides with *operational termination* [15]. We call a CTRS *constructor-based* if the right-hand sides of conditions as well as the arguments of left-hand sides of rules are constructor terms.

Limitations. We restrict ourselves to left-linear constructor-based deterministic 3-CTRSs, where moreover all right-hand sides of conditions are linear, and use only variables not occurring in the left-hand side or in earlier conditions. That is, for every rule $f(\ell_1, \dots, \ell_n) \to r \Leftarrow s_1 \approx t_1, \dots, s_k \approx t_k \in \mathcal{R}$:

- $\ell_1, \dots, \ell_n, t_1, \dots, t_k$ are linear constructor terms without common variables,
- $\mathcal{V}\mathsf{ar}(s_i) \subseteq \mathcal{V}\mathsf{ar}(\ell_1, \dots, \ell_n, t_1, \dots, t_{i-1})$ for $1 \leqslant i \leqslant k$ and $\mathcal{V}\mathsf{ar}(r) \subseteq \mathcal{V}\mathsf{ar}(\ell_1, \dots, \ell_n, t_1, \dots, t_k)$.

We will call such systems *CCTRSs* in the sequel. Furthermore, we restrict our attention to *quasi-decreasing* and *confluent* CCTRSs. While these latter restrictions are not needed for the formal development in this paper, without them the complexity notion that we propose is either undefined or not meaningful, as argued below.

To appreciate the limitations, note that in CTRSs which are not deterministic, 3-CTRSs or quasi-decreasing, the rewrite relation is undecidable in general, which makes it hard to define what complexity means. The restriction to linear constructor-TRSs is common in rewriting, and the restrictions on the conditions are a natural extension of this. Most importantly, with these restrictions computation is unambiguous: To evaluate whether a term $\ell\sigma$ reduces with a rule $\ell \to r \Leftarrow s_1 \approx t_1, \dots, s_k \approx t_k$, we start by reducing $s_1\sigma$ and, finding an instance of t_1, extend σ to the new variables in t_1 resulting in σ', continue with $s_2\sigma'$, and so on. If any extension of σ satisfies all conditions then this procedure will find one, no matter how we reduce. However, if confluence, quasi-decreasingness or any of the restrictions on the conditions were dropped, this would no longer be the case and we might be unable to verify whether a rule applied without enumerating all possible reducts of its conditions. The restrictions on the ℓ_i are needed to obtain Lemma 5, which will be essential to justify the way we handle failure.

▶ **Example 1.** The CTRS $\mathcal{R}$ consisting of the rewrite rules

$$\mathsf{0} + y \to y \qquad \mathsf{fib}(\mathsf{0}) \to \langle \mathsf{0}, \mathsf{s}(\mathsf{0}) \rangle$$
$$\mathsf{s}(x) + y \to \mathsf{s}(x + y) \qquad \mathsf{fib}(\mathsf{s}(x)) \to \langle z, w \rangle \;\Leftarrow\; \mathsf{fib}(x) \approx \langle y, z \rangle,\; y + z \approx w$$

is a quasi-deterministic and confluent CCTRS. The requirements for quasi-decreasingness are satisfied (e.g.) by the lexicographic path order with precedence $\mathsf{fib} > \langle \cdot, \cdot \rangle > + > \mathsf{s}$.

3 Analysis

We start our analysis with a deceivingly simple CCTRS to illustrate that the notion of complexity for conditional systems is not obvious.

▶ **Example 2.** The CCTRS $\mathcal{R}_{\mathsf{even}}$ consists of the following six rewrite rules:

$$\mathsf{even}(\mathsf{0}) \to \mathsf{true} \quad (1) \qquad \mathsf{odd}(\mathsf{0}) \to \mathsf{false} \quad (4)$$
$$\mathsf{even}(\mathsf{s}(x)) \to \mathsf{true} \;\Leftarrow\; \mathsf{odd}(x) \approx \mathsf{true} \quad (2) \qquad \mathsf{odd}(\mathsf{s}(x)) \to \mathsf{true} \;\Leftarrow\; \mathsf{even}(x) \approx \mathsf{true} \quad (5)$$
$$\mathsf{even}(\mathsf{s}(x)) \to \mathsf{false} \;\Leftarrow\; \mathsf{even}(x) \approx \mathsf{true} \quad (3) \qquad \mathsf{odd}(\mathsf{s}(x)) \to \mathsf{false} \;\Leftarrow\; \mathsf{odd}(x) \approx \mathsf{true} \quad (6)$$

If, like in the unconditional case, we count the number of steps needed to normalize a term, then a term $t_n = \mathsf{even}(\mathsf{s}^n(\mathsf{0}))$ has derivation height 1, since $t_n \to \mathsf{false}$ in a single step. To reflect actual computation, the rewrite steps to verify the condition should be taken into account. Viewed like this, normalizing t_n takes $n + 1$ rewrite steps.

Table 1 Number of steps required to normalize $\mathsf{even}(\mathsf{s}^n(0))$ and $\mathsf{odd}(\mathsf{s}^n(0))$ in Maude.

n	0	1	2	3	4	5	6	7	8	9	10	11	12
$2^{n+1}-1$	1	3	7	15	31	63	127	255	511	1023	2047	4095	8191
$\mathsf{even}(\mathsf{s}^n(0))$	1	3	3	11	5	37	7	135	9	521	11	2059	13
$\mathsf{odd}(\mathsf{s}^n(0))$	1	2	6	4	20	6	70	8	264	10	1034	12	4108
$\mathsf{even}(\mathsf{s}^n(0))$	1	2	7	8	31	32	127	128	511	512	2047	2048	8191
$\mathsf{odd}(\mathsf{s}^n(0))$	1	3	4	15	16	63	64	255	256	1023	1024	4095	4096

However, this still seems unrealistic, since a rewriting engine cannot know in advance which rule to attempt first. For example, when rewriting t_9, rule (2) may be tried first, which requires normalizing $\mathsf{odd}(\mathsf{s}^8(0))$ to verify the condition. After finding that the condition fails, rule (3) is attempted. Thus, for $\mathcal{R}_{\mathsf{even}}$, a realistic engine would select a rule with a failing condition about half the time. If we assume a worst possible selection strategy and count all rewrite steps performed during the computation, we need $2^{n+1}-1$ steps to normalize t_n.

Although this exponential upper bound may come as a surprise, a powerful rewrite engine like Maude [6] does not perform much better, as can be seen from the data in Table 1. For rows three and four we presented the rules to Maude in the order given in Example 2. Changing the order to (4), (6), (5), (1), (3), (2) we obtain the last two rows. For no order on the rules is the optimal linear bound on the number of steps obtained for all tested terms.

From the above we conclude that *a realistic definition of conditional complexity should take failed computations into account.* This opens new questions, which are best illustrated on a different (admittedly artificial) CTRS.

▶ **Example 3.** The CCTRS $\mathcal{R}_{\mathsf{fg}}$ consists of the following two rewrite rules:

$$\mathsf{f}(x) \to x \qquad\qquad \mathsf{g}(x) \to \mathsf{a} \Leftarrow x \approx \mathsf{b}$$

How many steps does it take to normalize $t_{n,m} = \mathsf{f}^n(\mathsf{g}(\mathsf{f}^m(\mathsf{a})))$? As we have not imposed an evaluation strategy, one approach for evaluating this term could be as follows. We use the second rule on the subterm $\mathsf{g}(\mathsf{f}^m(\mathsf{a}))$. This fails in m steps. With the first rule at the root position we obtain $t_{n-1,m}$. We again attempt the second rule, failing in m steps. Repeating this scenario results in $n \cdot m$ rewrite steps before we reach the term $t_{0,m}$.

In the above example we keep attempting—and failing—to rewrite an unmodified copy of a subterm we tried before, with the same rule. Even though the position of the subterm $\mathsf{g}(\mathsf{f}^m(\mathsf{a}))$ changes, we already know that this reduction will fail. Hence it is reasonable to assume that once we fail a conditional rule on given subterms, we should not try the same rule again on (copies of) the same subterms. This idea will be made formal in Section 4.

▶ **Example 4.** Continuing with the term $t_{0,m}$ from the preceding example, we could try to use the second rule, which fails in m steps. Next, the first rule is applied on a subterm, and we obtain $t_{0,m-1}$. Again we try the second rule, failing after executing $m-1$ steps. Repeating this alternation results eventually in the normal form $t_{0,0}$, but not before computing $\frac{1}{2}(m^2 + 3m)$ rewrite steps in total.

Like in Example 3, we keep coming back to a subterm which we have already tried before in an unsuccessful attempt. The difference is that the subterm has been rewritten between successive attempts. According to the following general result, we do not need to reconsider a failed attempt to apply a conditional rewrite rule if only the arguments were changed.

▶ **Lemma 5.** *Given a CCTRS $\mathcal{R}$, suppose $s \xrightarrow{>\epsilon}{}^{*} t$ and let $\rho\colon \ell \to r \Leftarrow c$ be a rule such that s is an instance of ℓ. If $t \xrightarrow{\epsilon}_{\rho} u$ then there exists a term v such that $s \xrightarrow{\epsilon}_{\rho} v$ and $v \to^{*} u$.*

So if we can rewrite a term at the root position eventually, and the term already matches the left-hand side of the rule with which we can do so, then we can rewrite the term with this rule immediately and obtain the same result.

Proof. Let σ be a substitution such that $s = \ell\sigma$ and $\mathsf{dom}(\sigma) \subseteq \mathcal{V}\mathsf{ar}(\ell)$. Because ℓ is a basic term, all steps in $s \xrightarrow{>\epsilon}{}^{*} t$ take place in the substitution part σ of $\ell\sigma$. Since ℓ is a linear term, we have $t = \ell\tau$ for some substitution τ such that $\mathsf{dom}(\tau) \subseteq \mathcal{V}\mathsf{ar}(\ell)$ and $\sigma \to^{*} \tau$. Because the rule ρ applies to t at the root position, there exists an extension τ' of τ such that $\mathcal{R} \vdash c\tau'$. We have $u = r\tau'$. Define the substitution σ' as follows:

$$\sigma'(x) = \begin{cases} \sigma(x) & \text{if } x \in \mathcal{V}\mathsf{ar}(\ell) \\ \tau'(x) & \text{if } x \notin \mathcal{V}\mathsf{ar}(\ell) \end{cases}$$

We have $s = \ell\sigma = \ell\sigma'$ and $\sigma' \to^{*} \tau'$. Let $a \approx b$ be a condition in c. From $\mathcal{V}\mathsf{ar}(b) \cap \mathcal{V}\mathsf{ar}(\ell) = \varnothing$ we infer $a\sigma' \to^{*} a\tau' \to^{*} b\tau' = b\sigma'$. It follows that $\mathcal{R} \vdash c\sigma'$ and thus $s \xrightarrow{\epsilon}_{\rho} r\sigma'$. Hence we can take $v = r\sigma'$ as $r\sigma' \to^{*} r\tau' = u$. ◀

From these observations we see that we can mark occurrences of defined symbols with the rules we have tried without success or, symmetrically, with the rules we have yet to try.

4 Conditional Complexity

To formalize the ideas from Section 3, we label defined function symbols by subsets of the rules used to define them.

▶ **Definition 6.** Let $\mathcal{R}$ be a CCTRS over a signature $\mathcal{F}$. The labeled signature $\mathcal{F}'$ is defined as $\mathcal{F}_{\mathcal{C}} \cup \{f_R \mid f \in \mathcal{F}_{\mathcal{D}} \text{ and } R \subseteq \mathcal{R}{\restriction}f\}$. A labeled term is a term in $\mathcal{T}(\mathcal{F}', \mathcal{V})$.

Intuitively, the label R in f_R records the defining rules for f which have not yet been attempted.

▶ **Definition 7.** Let $\mathcal{R}$ be a CCTRS over a signature $\mathcal{F}$. The mapping $\mathsf{label}\colon \mathcal{T}(\mathcal{F}, \mathcal{V}) \to \mathcal{T}(\mathcal{F}', \mathcal{V})$ labels every defined symbol f with $\mathcal{R}{\restriction}f$. The mapping $\mathsf{erase}\colon \mathcal{T}(\mathcal{F}', \mathcal{V}) \to \mathcal{T}(\mathcal{F}, \mathcal{V})$ removes the labels of defined symbols.

We obviously have $\mathsf{erase}(\mathsf{label}(t)) = t$ for every $t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$. The identity $\mathsf{label}(\mathsf{erase}(t)) = t$ holds for constructor terms t but not for arbitrary terms $t \in \mathcal{T}(\mathcal{F}', \mathcal{V})$.

▶ **Definition 8.** A *labeled normal form* is a term in $\mathcal{T}(\mathcal{F}_{\mathcal{C}} \cup \{f_{\varnothing} \mid f \in \mathcal{F}_{\mathcal{D}}\}, \mathcal{V})$.

The relation $\rightharpoonup$ is designed in such a way that a ground labeled term can be reduced if and only if it is not a labeled normal form. First, with Definition 9 we can remove a rule from a label if that rule will never be applicable due to an impossible matching problem.

▶ **Definition 9.** We write $s \overset{\perp}{\rightharpoonup} t$ if there exist a position $p \in \mathcal{P}\mathsf{os}(s)$ and a rewrite rule $\rho\colon f(\ell_1, \ldots, \ell_n) \to r \Leftarrow c$ such that

1. $s|_p = f_R(s_1, \ldots, s_n)$ with $\rho \in R$,
2. $t = s[f_{R \setminus \{\rho\}}(s_1, \ldots, s_n)]_p$, and
3. there exist a linear labeled normal form u with fresh variables, a substitution σ, and an index $1 \leqslant i \leqslant n$ such that $s_i = u\sigma$ and $\mathsf{erase}(u)$ does not unify with ℓ_i.

The last item ensures that rewriting s strictly below position p cannot give a reduct that matches ℓ, since $s_i = u\sigma$ can only reduce to instances $u\sigma'$ of u and thus not to an instance of ℓ_i. Furthermore, by the linearity of $\ell = f(\ell_1, \dots, \ell_n)$ we also have that, if $s_1, \dots, s_n$ are labeled normal forms then either $f(s_1, \dots, s_n)$ is an instance of ℓ or $\overset{\perp}{\rightharpoonup}$ applies.

Second, Definition 10 describes how to "reduce" labeled terms in general.

▶ **Definition 10.** A *complexity-conscious reduction* is a sequence $t_1 \rightharpoonup t_2 \rightharpoonup \cdots \rightharpoonup t_m$ of labeled terms where $s \rightharpoonup t$ if either $s \overset{\perp}{\rightharpoonup} t$ or there exist a position $p \in \mathcal{P}\mathsf{os}(s)$, rewrite rule $\rho\colon f(\ell_1, \dots, \ell_n) \to r \Leftarrow a_1 \approx b_1, \dots, a_k \approx b_k$, substitution σ, and index $1 \leqslant j \leqslant k$ such that

1. $s|_p = f_R(s_1, \dots, s_n)$ with $\rho \in R$ and $s_i = \ell_i\sigma$ for all $1 \leqslant i \leqslant n$,
2. $\mathsf{label}(a_i)\sigma \rightharpoonup^* b_i\sigma$ for all $1 \leqslant i \leqslant j$,

and either

3. $j = k$ and $t = s[\mathsf{label}(r)\sigma]_p$

in which case we speak of a *successful* step, or

4. $j < k$ and there exist a linear labeled normal form u and a substitution τ such that $\mathsf{label}(a_{j+1})\sigma \rightharpoonup^* u\tau$, u does not unify with b_{j+1}, and $t = s[f_{R \setminus \{\rho\}}(s_1, \dots, s_n)]_p$,

which is a *failed* step.

It is easy to see that for all ground labeled terms s which are not labeled normal forms, a term t exists such that $s \overset{\perp}{\rightharpoonup} t$ or there are p, ρ, σ such that $s|_p$ "matches" ρ in the sense that the first requirement in Definition 10 is satisfied. As all b_i are linear constructor terms on fresh variables and conditions are evaluated from left to right, $\mathsf{label}(a_i)\sigma \rightharpoonup b_i\sigma$ simply indicates that $a_i\sigma$—with labels added to allow reducing defined symbols in a_i—reduces to an instance of b_i.

A successful reduction occurs when we manage to reduce each $\mathsf{label}(a_i)\sigma$ to $b_i\sigma$. A failed reduction happens when we start reducing $\mathsf{label}(a_i)\sigma$ and obtain a term that will never reduce to an instance of b_i. As discussed after Definition 9, this is what happens in case 4.

▶ **Definition 11.** The *cost* of a complexity-conscious reduction is the sum of the costs of its steps. The cost of a step $s \rightharpoonup t$ is 0 if $s \overset{\perp}{\rightharpoonup} t$,

$$1 + \sum_{i=1}^{k} \mathsf{cost}(\mathsf{label}(a_i)\sigma \rightharpoonup^* b_i\sigma)$$

in case of a successful step $s \rightharpoonup t$, and

$$\sum_{i=1}^{j} \mathsf{cost}(\mathsf{label}(a_i)\sigma \rightharpoonup^* b_i\sigma) + \mathsf{cost}(\mathsf{label}(a_{j+1})\sigma \rightharpoonup^* u\tau)$$

in case of a failed step $s \rightharpoonup t$. The *conditional derivational complexity* of a CCTRS $\mathcal{R}$ is defined as $\mathsf{cdc}_{\mathcal{R}}(n) = \max\{\mathsf{cost}(t \rightharpoonup^* u) \mid |t| \leqslant n \text{ and } t \rightharpoonup^* u \text{ for some term } u\}$. If we restrict t to basic terms we arrive at the *conditional runtime complexity* $\mathsf{crc}_{\mathcal{R}}(n)$.

Note that the cost of a failed step is the cost to evaluate its conditions and conclude failure, while for a successful step we add one for the step itself.

The following result connects the relations $\rightharpoonup$ and $\to$ to each other.

▶ **Lemma 12.** *Let $\mathcal{R}$ be a CCTRS.*

1. *If* $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ *and* $s \to^* t$ *then* $\mathsf{label}(s) \rightharpoonup^* \mathsf{label}(t)$*.*
2. *If* $s, t \in \mathcal{T}(\mathcal{F}', \mathcal{V})$ *and* $s \rightharpoonup^* t$ *then* $\mathsf{erase}(s) \to^* \mathsf{erase}(t)$*.*

Proof. **1.** We use induction on the number of rewrite steps needed to derive $s \to^* t$. If $s = t$ then the result is obvious, so let $s \to u \to^* t$. The induction hypothesis yields $\mathsf{label}(u) \rightharpoonup^* \mathsf{label}(t)$, so it suffices to show $\mathsf{label}(s) \rightharpoonup^* \mathsf{label}(u)$. There exist a position $p \in \mathcal{P}\mathsf{os}(s)$, a rule $\rho\colon \ell \to r \Leftarrow a_1 \approx b_1, \ldots, a_k \approx b_k$, and a substitution σ such that $s|_p = \ell\sigma$, $u = s[r\sigma]_p$, and $a_i\sigma \to^* b_i\sigma$ for all $1 \leqslant i \leqslant n$. Let σ' be the (labeled) substitution $\mathsf{label} \circ \sigma$. Fix $1 \leqslant i \leqslant k$. We have $\mathsf{label}(a_i\sigma) = \mathsf{label}(a_i)\sigma'$ and $\mathsf{label}(b_i\sigma) = b_i\sigma'$ (as b_i is a constructor term). Because $a_i\sigma \to^* b_i\sigma$ is used in the derivation of $s \to^* t$ we can apply the induction hypothesis, resulting in $\mathsf{label}(a_i\sigma) \rightharpoonup^* \mathsf{label}(b_i\sigma)$. Furthermore, writing $\ell = f(\ell_1, \ldots, \ell_n)$, we obtain $\mathsf{label}(\ell) = f_{\mathcal{R}\restriction f}(\ell_1, \ldots, \ell_n)$. Hence

$$\mathsf{label}(s) = \mathsf{label}(s)[\mathsf{label}(\ell)\sigma']_p \rightharpoonup \mathsf{label}(s)[\mathsf{label}(r)\sigma']_p = \mathsf{label}(u)$$

because conditions (1)–(4) in Definition 10 are satisfied.

2. We use induction on the pair $(\mathsf{cost}(s \rightharpoonup^* t), \|s\|)$ where $\|s\|$ denotes the sum of the sizes of the labels of defined symbols in s, ordered lexicographically. The result is obvious if $s = t$, so let $s \rightharpoonup u \rightharpoonup^* t$. Clearly $\mathsf{cost}(s \rightharpoonup^* t) \geqslant \mathsf{cost}(u \rightharpoonup^* t)$. We distinguish two cases.

- Suppose $s \overset{\perp}{\rightharpoonup} u$ or $s \rightharpoonup u$ by a failed step. In either case we have $\mathsf{erase}(s) = \mathsf{erase}(u)$ and $\|s\| = \|u\| + 1$. The induction hypothesis yields $\mathsf{erase}(u) \to^* \mathsf{erase}(t)$.
- Suppose $s \rightharpoonup u$ is a successful step. So there exist a position $p \in \mathcal{P}\mathsf{os}(s)$, a rule $\rho\colon \ell \to r \Leftarrow a_1 \approx b_1, \ldots, a_k \approx b_k$ in $\mathcal{R}$, a substitution σ, and terms $\ell', a_1', \ldots, a_k'$ such that $s|_p = \ell'\sigma$ with $\mathsf{erase}(\ell') = \ell$, $a_i'\sigma \rightharpoonup^* b_i\sigma$ with $\mathsf{erase}(a_i') = a_i$ for all $1 \leqslant i \leqslant k$, and $u = s[\mathsf{label}(r)\sigma]_p$. Let σ' be the (unlabeled) substitution $\mathsf{erase} \circ \sigma$. We have $\mathsf{erase}(s) = \mathsf{erase}(s)[\ell\sigma']_p$ and $\mathsf{erase}(u) = \mathsf{erase}(s)[r\sigma']_p$. Since $\mathsf{cost}(s \rightharpoonup u) > \mathsf{cost}(a_i'\sigma \rightharpoonup^* b_i\sigma)$ we obtain $a_i\sigma' = \mathsf{erase}(a_i'\sigma) \to^* \mathsf{erase}(b_i\sigma) = b_i\sigma'$ from the induction hypothesis, for all $1 \leqslant i \leqslant k$. Hence $\mathsf{erase}(s) \to \mathsf{erase}(u)$. Finally, $\mathsf{erase}(u) \to^* \mathsf{erase}(t)$ by another application of the induction hypothesis. ◀

5 Complexity Transformation

The notion of complexity introduced in the preceding section has the downside that we cannot easily reuse existing complexity results and tools. Therefore, we will consider a transformation to unconditional rewriting where, rather than tracking rules in the labels of the defined function symbols, we will keep track of them in separate arguments, but restrict reduction by adopting a suitable context-sensitive replacement map.

▶ **Definition 13.** Let $\mathcal{R}$ be a CCTRS over a signature $\mathcal{F}$. For $f \in \mathcal{F}_{\mathcal{D}}$, let m_f be the number of rules in $\mathcal{R}\restriction f$. The context-sensitive signature $(\mathcal{G}, \mu)$ is defined as follows:

- $\mathcal{G}$ contains two constants $\perp$ and $\top$,
- for every constructor symbol $g \in \mathcal{F}_{\mathcal{C}}$ of arity n, $\mathcal{G}$ contains the symbol g with the same arity and $\mu(g) = \{1, \ldots, n\}$,
- for every defined symbol $f \in \mathcal{F}_{\mathcal{D}}$ of arity n, $\mathcal{G}$ contains two symbols f and f_{a} of arity $n + m_f$ with $\mu(f) = \{1, \ldots, n\}$ and $\mu(f_{\mathsf{a}}) = \{n+1, \ldots, n+m_f\}$,
- for every defined symbol $f \in \mathcal{F}_{\mathcal{D}}$ of arity n, rewrite rule $\rho\colon \ell \to r \Leftarrow c_1, \ldots, c_k$ in $\mathcal{R}\restriction f$, and $1 \leqslant i \leqslant k$, $\mathcal{G}$ contains a symbol c_ρ^i of arity $n + i$ with $\mu(c_\rho^i) = \{n + i\}$.

Fixing an order $\mathcal{R}\restriction f = \{\rho_1, \ldots, \rho_{m_f}\}$, terms in $\mathcal{T}(\mathcal{G}, \mathcal{V})$ that are involved in reducing $f(s_1, \ldots, s_n) \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ will have one of two forms: $f(s_1, \ldots, s_n, t_1, \ldots, t_{m_f})$ with each $t_i \in \{\top, \bot\}$, indicating that rule ρ_i has been attempted (and failed) if and only if $t_i = \bot$, and $f_{\mathsf{a}}(s_1, \ldots, s_n, t_1, \ldots, c_{\rho_i}^{j+1}(s_1, \ldots, s_n, b_1, \ldots, b_j, u_{j+1}), \ldots, t_{m_f})$ indicating that rule ρ_i is currently being evaluated and the first j conditions of ρ_i have succeeded. The reason for passing the terms $s_1, \ldots, s_n$ to $c_{\rho_i}^{j+1}$ is that it allows for easier complexity methods.

▶ **Definition 14.** The maps $\xi_\star \colon \mathcal{T}(\mathcal{F}, \mathcal{V}) \to \mathcal{T}(\mathcal{G}, \mathcal{V})$ with $\star \in \{\bot, \top\}$ are inductively defined:

$$\xi_\star(t) = \begin{cases} t & \text{if } t \text{ is a variable,} \\ f(\xi_\star(t_1), \ldots, \xi_\star(t_n)) & \text{if } t = f(t_1, \ldots, t_n) \text{ and } f \text{ is a constructor symbol,} \\ f(\xi_\star(t_1), \ldots, \xi_\star(t_n), \star, \ldots, \star) & \text{if } t = f(t_1, \ldots, t_n) \text{ and } f \text{ is a defined symbol.} \end{cases}$$

Linear terms in the set $\{\xi_\bot(t) \mid t \in \mathcal{T}(\mathcal{F}, \mathcal{V})\}$ are called *$\bot$-patterns.*

In the transformed system that we will define, a ground term is in normal form if and only if it is a $\bot$-pattern. This allows for syntactic "normal form" tests. Most importantly, it allows for purely syntactic anti-matching tests: If s does not reduce to an instance of some linear constructor term t, then $s \to^* u\sigma$ for some substitution σ and $\bot$-pattern u that does not unify with t. What is more, we only need to consider a finite number of $\bot$-patterns u.

▶ **Definition 15.** Let t be a linear constructor term. The set of *anti-patterns* $\mathsf{AP}(t)$ is inductively defined as follows. If t is a variable then $\mathsf{AP}(t) = \varnothing$. If $t = f(t_1, \ldots, t_n)$ then $\mathsf{AP}(t)$ consists of the following $\bot$-patterns:

- $g(x_1, \ldots, x_m)$ for every m-ary constructor symbol g different from f,
- $g(x_1, \ldots, x_m, \bot, \ldots, \bot)$ for every defined symbol g of arity m in $\mathcal{F}$, and
- $f(x_1, \ldots, x_{i-1}, u, x_{i+1}, \ldots, x_n)$ for all $1 \leqslant i \leqslant n$ and $u \in \mathsf{AP}(t_i)$.

Here $x_1, \ldots, x_{m(n)}$ are fresh and pairwise distinct variables.

▶ **Example 16.** Consider the CCTRS of Example 1. The set $\mathsf{AP}(\langle z, w\rangle)$ consists of the $\bot$-patterns $\mathsf{0}$, $\mathsf{s}(x)$, $\mathsf{fib}(x, \bot, \bot)$, and $+(x, y, \bot, \bot)$.

The straightforward proof of the following lemma is omitted.

▶ **Lemma 17.** *Let s be a $\bot$-pattern and t a linear constructor term with $\mathcal{V}\mathsf{ar}(s) \cap \mathcal{V}\mathsf{ar}(t) = \varnothing$. If s and t are not unifiable then s is an instance of an anti-pattern in $\mathsf{AP}(t)$.*

We are now ready to define the transformation from a CCTRS $(\mathcal{F}, \mathcal{R})$ to a context-sensitive TRS $(\mathcal{G}, \mu, \Xi(\mathcal{R}))$. Here, we will use the notation $\langle t_1, \ldots, t_n\rangle[u]_i$ to denote the sequence $t_1, \ldots, t_{i-1}, u, t_{i+1}, \ldots, t_n$ and we occasionally write $\vec{t}$ for a sequence $t_1, \ldots, t_n$.

▶ **Definition 18.** Let $\mathcal{R}$ be a CCTRS over a signature $\mathcal{F}$. For every defined symbol $f \in \mathcal{F}_{\mathcal{D}}$ we fix an order on the m_f rules that define f: $\mathcal{R}\restriction f = \{\rho_1, \ldots, \rho_{m_f}\}$. The context-sensitive TRS $\Xi(\mathcal{R})$ is defined over the signature $(\mathcal{G}, \mu)$ as follows. Let $\rho\colon f(\ell_1, \ldots, \ell_n) \to r \Leftarrow a_1 \approx b_1, \ldots, a_k \approx b_k$ be the i-th rule in $\mathcal{R}\restriction f$.

- If $k = 0$ then $\Xi(\mathcal{R})$ contains the rule

$$f(\ell_1, \ldots, \ell_n, \langle x_1, \ldots, x_{m_f}\rangle[\top]_i) \to \xi_\top(r) \tag{1_ρ}$$

- If $k > 0$ then $\Xi(\mathcal{R})$ contains the rules

$$f(\vec{\ell}, \langle x_1, \ldots, x_{m_f} \rangle[\top]_i) \to f_{\mathsf{a}}(\vec{\ell}, \langle x_1, \ldots, x_{m_f} \rangle[c^1_\rho(\vec{\ell}, \xi_\top(a_1))]_i) \qquad (2_\rho)$$

$$f_{\mathsf{a}}(\vec{\ell}, \langle x_1, \ldots, x_{m_f} \rangle[c^k_\rho(\vec{y}, b_1, \ldots, b_k)]_i) \to \xi_\top(r) \qquad (3_\rho)$$

  and the rules

$$f_{\mathsf{a}}(\vec{\ell}, \langle x_1, \ldots, x_{m_f} \rangle[c^j_\rho(\vec{y}, b_1, \ldots, b_j)]_i) \to f_{\mathsf{a}}(\vec{\ell}, \langle x_1, \ldots, x_{m_f} \rangle[c^{j+1}_\rho(\vec{y}, b_1, \ldots, b_j, \xi_\top(a_{j+1}))]_i) \qquad (4_\rho)$$

  for all $1 \leqslant j < k$, and the rules

$$f_{\mathsf{a}}(\vec{\ell}, \langle x_1, \ldots, x_{m_f} \rangle[c^j_\rho(\vec{y}, b_1, \ldots, b_{j-1}, v)]_i) \to f(\vec{\ell}, \langle x_1, \ldots, x_{m_f} \rangle[\bot]_i) \qquad (5_\rho)$$

  for all $1 \leqslant j \leqslant k$ and $v \in \mathsf{AP}(b_j)$.
- Regardless of k, $\Xi(\mathcal{R})$ contains the rules

$$f(\langle y_1, \ldots, y_n \rangle[v]_j, \langle x_1, \ldots, x_{m_f} \rangle[\top]_i) \to f(\langle y_1, \ldots, y_n \rangle[v]_j, \langle x_1, \ldots, x_{m_f} \rangle[\bot]_i) \qquad (6_\rho)$$

  for all $1 \leqslant j \leqslant n$ and $v \in \mathsf{AP}(\ell_j)$.

Here $x_1, \ldots, x_{m_f}, y_1, \ldots, y_n$ are fresh and pairwise distinct variables. A step using rule (1_ρ) or rule (3_ρ) has cost 1; other rules—also called *administrative rules*—have cost 0.

Rule (1_ρ) simply adds the $\top$ labels to the right-hand sides of unconditional rules. To apply a conditional rule ρ, we "activate" the current function symbol with rule (2_ρ) and start evaluating the first condition of ρ by steps inside the last argument of c^1_ρ. With rules (4_ρ) we move to the next condition and, after all conditions have succeeded, an application of rule (3_ρ) results in the right-hand side with $\top$ labels. If a condition fails (5_ρ) or the left-hand side of the rule does not match and will never match (6_ρ), then we simply replace the label for ρ by $\bot$, indicating that we do not need to try it again.

These rules carry some redundant information. For example, all c^i_ρ are passed the parameters $\ell_1, \ldots, \ell_n$ of the corresponding rule. This is done to make it easier to orient the resulting rules with interpretations, as we will see in Section 6. Also, instead of passing $b_1, \ldots, b_j$ to each c^{j+1}_ρ, and $\ell_1, \ldots, \ell_n$ to f_{a}, it would suffice to pass along their *variables*. This was left in the current form to simplify the presentation.

Note that the rules that do not produce the right-hand side of the originating conditional rewrite rule are administrative and hence do not contribute to the cost of a reduction. The anti-pattern sets result in many rules (5_ρ) and (6_ρ), but all of these are simple. We could generalize the system by replacing each $\star_i$ by a fresh variable; the complexity of the resulting (smaller) TRS gives an upper bound for the original complexity.

▶ **Example 19.** The (context-sensitive) TRS $\Xi(\mathcal{R}_{\mathsf{even}})$ consists of the following rules:

$$\mathsf{even}(0, \top, y, z) \to \mathsf{true} \qquad (1_1)$$

$$\mathsf{even}(\star_1, \top, y, z) \to \mathsf{even}(\star_1, \bot, y, z) \qquad (6_1)$$

$$\mathsf{even}(\mathsf{s}(x), y, \top, z) \to \mathsf{even}_{\mathsf{a}}(\mathsf{s}(x), y, \mathsf{c}^1_2(\mathsf{s}(x), \mathsf{odd}(x, \top, \top, \top)), z) \qquad (2_2)$$

$$\mathsf{even}_{\mathsf{a}}(\mathsf{s}(x), y, \mathsf{c}^1_2(y', \mathsf{true}), z) \to \mathsf{true} \qquad (3_2)$$

$$\mathsf{even}_{\mathsf{a}}(\mathsf{s}(x), y, \mathsf{c}^1_2(y', \star_2), z) \to \mathsf{even}(\mathsf{s}(x), y, \bot, z) \qquad (5_2)$$

$$\mathsf{even}(\star_3, y, \top, z) \to \mathsf{even}(\star_3, y, \bot, z) \qquad (6_2)$$

$$\begin{aligned}
\mathsf{even}(\mathsf{s}(x), y, z, \top) &\to \mathsf{even_a}(\mathsf{s}(x), y, z, \mathsf{c}_3^1(\mathsf{s}(x), \mathsf{even}(x, \top, \top, \top))) && (2_3)\\
\mathsf{even_a}(\mathsf{s}(x), y, z, \mathsf{c}_3^1(y', \mathsf{true})) &\to \mathsf{false} && (3_3)\\
\mathsf{even_a}(\mathsf{s}(x), y, z, \mathsf{c}_3^1(y', \star_2)) &\to \mathsf{even}(\mathsf{s}(x), y, z, \bot) && (5_3)\\
\mathsf{even}(\star_3, y, z, \top) &\to \mathsf{even}(\star_3, y, z, \bot) && (6_3)\\
\mathsf{odd}(0, \top, y, z) &\to \mathsf{false} && (1_4)\\
\mathsf{odd}(\star_1, \top, y, z) &\to \mathsf{odd}(\star_1, \bot, y, z) && (6_4)\\
\mathsf{odd}(\mathsf{s}(x), y, \top, z) &\to \mathsf{odd_a}(\mathsf{s}(x), y, \mathsf{c}_5^1(\mathsf{s}(x), \mathsf{odd}(x, \top, \top, \top)), z) && (2_5)\\
\mathsf{odd_a}(\mathsf{s}(x), y, \mathsf{c}_5^1(y', \mathsf{true}), z) &\to \mathsf{false} && (3_5)\\
\mathsf{odd_a}(\mathsf{s}(x), y, \mathsf{c}_5^1(y', \star_2), z) &\to \mathsf{odd}(\mathsf{s}(x), y, \bot, z) && (5_5)\\
\mathsf{odd}(\star_3, y, \top, z) &\to \mathsf{odd}(\star_3, y, \bot, z) && (6_5)\\
\mathsf{odd}(\mathsf{s}(x), y, z, \top) &\to \mathsf{odd_a}(\mathsf{s}(x), y, z, \mathsf{c}_6^1(\mathsf{s}(x), \mathsf{even}(x, \top, \top, \top))) && (2_6)\\
\mathsf{odd_a}(\mathsf{s}(x), y, z, \mathsf{c}_6^1(y', \mathsf{true})) &\to \mathsf{true} && (3_6)\\
\mathsf{odd_a}(\mathsf{s}(x), y, z, \mathsf{c}_6^1(y', \star_2)) &\to \mathsf{odd}(\mathsf{s}(x), y, z, \bot) && (5_6)\\
\mathsf{odd}(\star_3, y, z, \top) &\to \mathsf{odd}(\star_3, y, z, \bot) && (6_6)
\end{aligned}$$

for all

$$\begin{aligned}
&\star_1 \in \mathsf{AP}(0) = \{\mathsf{true}, \mathsf{false}, \mathsf{s}(x), \mathsf{even}(x, \bot, \bot, \bot), \mathsf{odd}(x, \bot, \bot, \bot)\}\\
&\star_2 \in \mathsf{AP}(\mathsf{true}) = \{\mathsf{false}, 0, \mathsf{s}(x), \mathsf{even}(x, \bot, \bot, \bot), \mathsf{odd}(x, \bot, \bot, \bot)\}\\
&\star_3 \in \mathsf{AP}(\mathsf{s}(x)) = \{\mathsf{true}, \mathsf{false}, 0, \mathsf{even}(x, \bot, \bot, \bot), \mathsf{odd}(x, \bot, \bot, \bot)\}
\end{aligned}$$

Below we relate complexity-conscious reductions with $\mathcal{R}$ to context-sensitive reductions in $\Xi(\mathcal{R})$. The following definition explains how we map terms in $\mathcal{T}(\mathcal{F}', \mathcal{V})$ to terms in $\mathcal{T}(\mathcal{G}, \mathcal{V})$. It resembles the earlier definition of $\xi_\star$.

▶ **Definition 20.** For $t \in \mathcal{T}(\mathcal{F}', \mathcal{V})$ we define

$$\zeta(t) = \begin{cases} t & \text{if } t \in \mathcal{V},\\ f(\zeta(t_1), \ldots, \zeta(t_n)) & \text{if } t = f(t_1, \ldots, t_n) \text{ with } f \text{ a constructor symbol,}\\ f(\zeta(t_1), \ldots, \zeta(t_n), c_1, \ldots, c_{m_f}) & \text{if } t = f_R(t_1, \ldots, t_n) \text{ with } R \subseteq \mathcal{R}{\restriction}f \end{cases}$$

where $c_i = \top$ if the i-th rule of $\mathcal{R}{\restriction}f$ belongs to R and $c_i = \bot$ otherwise, for $1 \leqslant i \leqslant m_f$. For a substitution $\sigma \in \Sigma(\mathcal{F}', \mathcal{V})$ we denote the substitution $\zeta \circ \sigma$ by σ_ζ.

It is easy to see that $p \in \mathcal{P}\mathsf{os}_\mu(\zeta(t))$ if and only if $p \in \mathcal{P}\mathsf{os}(t)$ if and only if $\zeta(t)|_p \notin \{\bot, \top\}$, for any $t \in \mathcal{T}(\mathcal{F}', \mathcal{V})$. The easy induction proof of the following lemma is omitted.

▶ **Lemma 21.** *If $t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ then $\zeta(\mathsf{label}(t)) = \xi_\top(t)$. If $t \in \mathcal{T}(\mathcal{F}', \mathcal{V})$ and $\sigma \in \Sigma(\mathcal{F}', \mathcal{V})$ then $\zeta(t\sigma) = \zeta(t)\sigma_\zeta$. Moreover, if t is a labeled normal form then $\zeta(t) = \xi_\bot(\mathsf{erase}(t))$.*

▶ **Theorem 22.** *Let $\mathcal{R}$ be a CCTRS. If $s \rightharpoonup^* t$ is a complexity-conscious reduction with cost N then there exists a context-sensitive reduction $\zeta(s) \to^*_{\Xi(\mathcal{R}),\mu} \zeta(t)$ with cost N.*

Proof. We use induction on the number of steps in $s \rightharpoonup^* t$. The result is obvious when this number is zero, so let $s \rightharpoonup u \rightharpoonup^* t$ with M the cost of the step $s \rightharpoonup u$ and $N - M$ the cost of $u \rightharpoonup^* t$. The induction hypothesis yields a context-sensitive reduction $\zeta(u) \to^*_{\Xi(\mathcal{R}),\mu} \zeta(t)$ of cost $N - M$ and so it remains to show that there exists a context-sensitive reduction $\zeta(s) \to^*_{\Xi(\mathcal{R}),\mu} \zeta(u)$ of cost M. Let $\rho\colon f(\ell_1, \ldots, \ell_n) \to r \Leftarrow a_1 \approx b_1, \ldots, a_k \approx b_k$ be the rule

in $\mathcal{R}$ that gives rise to the step $s \rightharpoonup u$ and let i be its index in $\mathcal{R}\restriction f$. There exist a position $p \in \mathcal{P}\mathsf{os}(s)$, terms $s_1, \ldots, s_n$, and a subset $R \subseteq \mathcal{R}\restriction f$ such that $s|_p = f_R(s_1, \ldots, s_n)$ and $\rho \in R$. We have $\zeta(s)|_p = \zeta(s|_p) = f_R(\zeta(s_1), \ldots, \zeta(s_n), c_1, \ldots, c_{m_f})$ where $c_j = \top$ if the j-th rule of $\mathcal{R}\restriction f$ belongs to R and $c_j = \bot$ otherwise, for $1 \leqslant j \leqslant m_f$. In particular, $c_i = \top$. Note that p is an active position in $\zeta(s)$. We distinguish three cases.

- First suppose that $s \xrightharpoonup{\bot} u$. So $M = 0$, $u = s[f_{R\setminus\{\rho\}}(s_1, \ldots, s_n)]_p$, and there exist a linear labeled normal form v, a substitution σ, and an index $1 \leqslant j \leqslant n$ such that $s_j = v\sigma$ and $\mathsf{erase}(v)$ does not unify with ℓ_j. By Lemma 21, $\zeta(s_j) = \zeta(v\sigma) = \zeta(v)\sigma_\zeta = \xi_\bot(\mathsf{erase}(v))\sigma_\zeta$. By definition, $\xi_\bot(\mathsf{erase}(v))$ is a $\bot$-pattern, which cannot unify with ℓ_j because $\mathsf{erase}(v)$ does not. From Lemma 17 we obtain an anti-pattern $v' \in \mathsf{AP}(\ell_j)$ such that $\xi_\bot(\mathsf{erase}(v))$ is an instance of v'. Hence $\zeta(s) = \zeta(s)[f(\zeta(s_1), \ldots, \zeta(s_n), c_1, \ldots, c_{m_f})]_p$ with $\zeta(s_j)$ an instance of $v' \in \mathsf{AP}(\ell_j)$ and $c_i = \top$. Consequently, $\zeta(s)$ reduces to $\zeta(s)[f(\zeta(s_1), \ldots, \zeta(s_n), \langle c_1, \ldots, c_{m_f}\rangle[\bot]_i)]_p$ by an application of rule (6_ρ), which has cost zero. The latter term equals $\zeta(s[f_{R\setminus\{\rho\}}(s_1, \ldots, s_n)]_p) = \zeta(u)$, and hence we are done.
- Next suppose that $s \rightharpoonup u$ is a successful step. So there exists a substitution σ such that $\mathsf{label}(a_i)\sigma \rightharpoonup^* b_i\sigma$ with cost M_i for all $1 \leqslant i \leqslant k$, and $M = 1 + M_1 + \cdots + M_k$. The induction hypothesis yields reductions $\zeta(\mathsf{label}(a_i)\sigma) \to^*_{\Xi(\mathcal{R}),\mu} \zeta(b_i\sigma)$ with cost M_i. By Lemma 21, $\zeta(\mathsf{label}(a_i)\sigma) = \zeta(\mathsf{label}(a_i))\sigma_\zeta = \xi_\top(a_i)\sigma_\zeta$ and $\zeta(b_i\sigma) = b_i\sigma_\zeta$. Moreover, $\zeta(s)|_p = \zeta(s|_p) = f(\vec{\ell}, \langle c_1, \ldots, c_{m_f}\rangle[\top]_i)\sigma_\zeta$ and $\zeta(u) = \zeta(s)[\zeta(\mathsf{label}(r)\sigma)]_p$ with $\zeta(\mathsf{label}(r))\sigma_\zeta = \xi_\top(r)\sigma_\zeta$ by Lemma 21. So it suffices if $f(\vec{\ell}, \langle c_1, \ldots, c_{m_f}\rangle[\top]_i)\sigma_\zeta \to^*_{\Xi(\mathcal{R}),\mu} \xi_\top(r)\sigma_\zeta$ with cost M. If $k = 0$ we can use rule (1_ρ). Otherwise, we use the reductions $\xi_\top(a_i)\sigma_\zeta \to^*_{\Xi(\mathcal{R}),\mu} b_i\sigma_\zeta$, rules (2_ρ) and (3_ρ), and $k-1$ times a rule of type (4_ρ) to obtain

$$\begin{aligned}
f(\vec{\ell}, \langle c_1, \ldots, c_{m_f}\rangle[\top]_i)\sigma_\zeta &\to_{\Xi(\mathcal{R}),\mu} f_{\mathsf{a}}(\vec{\ell}, \langle c_1, \ldots, c_{m_f}\rangle[c^1_\rho(\vec{\ell}, \xi_\top(a_1))]_i)\sigma_\zeta \\
&\to^*_{\Xi(\mathcal{R}),\mu} f_{\mathsf{a}}(\vec{\ell}, \langle c_1, \ldots, c_{m_f}\rangle[c^1_\rho(\vec{\ell}, b_1)]_i)\sigma_\zeta \\
&\to_{\Xi(\mathcal{R}),\mu} f_{\mathsf{a}}(\vec{\ell}, \langle c_1, \ldots, c_{m_f}\rangle[c^2_\rho(\vec{\ell}, b_1, \xi_\top(a_2))]_i)\sigma_\zeta \\
&\to^*_{\Xi(\mathcal{R}),\mu} \cdots \\
&\to_{\Xi(\mathcal{R}),\mu} f_{\mathsf{a}}(\vec{\ell}, \langle c_1, \ldots, c_{m_f}\rangle[c^k_\rho(\vec{\ell}, b_1, \ldots, b_k)]_i)\sigma_\zeta \\
&\to_{\Xi(\mathcal{R}),\mu} \xi_\top(r)\sigma_\zeta
\end{aligned}$$

  Note that all steps take place at active positions, and that the steps with rules 2_ρ and 4_ρ are administrative. Therefore, the cost of this reduction equals M.
- The remaining case is a failed step $s \rightharpoonup u$. So there exist substitutions σ and τ, an index $1 \leqslant j < k$, and a linear labeled normal form v which does not unify with b_{j+1} such that $\mathsf{label}(a_i)\sigma \rightharpoonup^* b_i\sigma$ with cost M_i for all $1 \leqslant i \leqslant j$ and $\mathsf{label}(a_{j+1})\sigma \rightharpoonup^* v\tau$ with cost M_{j+1}. We obtain $\zeta(\mathsf{label}(a_i)\sigma) = \xi_\top(a_i)\sigma_\zeta$, $\zeta(b_i\sigma) = b_i\sigma_\zeta$, and $\zeta(s)|_p = f(\vec{\ell}, \langle c_1, \ldots, c_{m_f}\rangle[\top]_i)\sigma_\zeta$ like in the preceding case. Moreover, like in the first case, we obtain an anti-pattern $v' \in \mathsf{AP}(b_{j+1})$ such that $\xi_\bot(\mathsf{erase}(v))$ is an instance of v'. We have $\zeta(v\tau) = \zeta(v)\tau_\zeta = \xi_\bot(\mathsf{erase}(v))\tau_\zeta$ by Lemma 21. Hence $\zeta(v\tau)$ is an instance of v'. Consequently,

$$\begin{aligned}
f(\vec{\ell}, \langle c_1, \ldots, c_{m_f}\rangle[\top]_i)\sigma_\zeta &\to^*_{\Xi(\mathcal{R}),\mu} f_{\mathsf{a}}(\vec{\ell}, \langle c_1, \ldots, c_{m_f}\rangle[c^{j+1}_\rho(\vec{\ell}, b_1, \ldots, b_j, \xi_\top(a_{j+1}))]_i)\sigma_\zeta \\
&\to^*_{\Xi(\mathcal{R}),\mu} f_{\mathsf{a}}(\vec{\ell}, \langle c_1, \ldots, c_{m_f}\rangle[c^{j+1}_\rho(\vec{\ell}, b_1, \ldots, b_j, \zeta(v\tau))]_i)\sigma_\zeta \\
&\to_{\Xi(\mathcal{R}),\mu} f(\vec{\ell}, \langle c_1, \ldots, c_{m_f}\rangle[\bot]_i)\sigma_\zeta
\end{aligned}$$

  where the last step uses an administrative rule of type (5_ρ). Again, all steps take place at active positions. Note that $f(\vec{\ell}, \langle c_1, \ldots, c_{m_f}\rangle[\bot]_i)\sigma_\zeta = \zeta(f_{R\setminus\{\rho\}}(s_1, \ldots, s_n)) = \zeta(u|_p)$.

Hence $\zeta(s) \to^*_{\Xi(\mathcal{R}),\mu} \zeta(u)$ as desired. The cost of this reduction is $M_1 + \cdots + M_{j+1}$, which coincides with the cost M of the step $s \rightharpoonup u$. ◀

Theorem 22 provides a way to establish conditional complexity: If $\Xi(\mathcal{R})$ has complexity $O(\varphi(n))$ then the conditional complexity of $\mathcal{R}$ is at most $O(\varphi(n))$.[2] Although there are no complexity tools yet which take context-sensitivity into account, we can obtain an upper bound by simply ignoring the replacement map. Similarly, although existing tools do not accommodate administrative rules, we can count all rules equally. Since for every non-administrative step reducing a term $f_R(\cdots)$ at the root position, at most (number of rules) $\times$ (greatest number of conditions + 1) administrative steps at the root position can be done, the difference is only a constant factor. Moreover, these rules are an instance of *relative rewriting*, for which advanced complexity methods do exist. Thus, it is likely that there will be direct tool support in the future.

6 Interpretations in $\mathbb{N}$

A common method to derive complexity bounds for a TRS is to use interpretations in $\mathbb{N}$. Such an interpretation $\mathcal{I}$ maps function symbols of arity n to functions from $\mathbb{N}^n$ to $\mathbb{N}$, giving a value $[t]_\mathcal{I}$ for ground terms t, which is shown to decrease in every reduction step. The method is easily adapted to take context-sensitivity and administrative rules into account.

Unfortunately, standard interpretation techniques like polynomial interpretations are ill equipped to deal with exponential bounds. Furthermore, to handle the interleaving behavior of f and f_a, the natural choice for an interpretation is to map both symbols to the same function. However, compatibility with rule (2_ρ) then gives rise to the constraint $[\top]_\mathcal{I} \geqslant [c^1_\rho(\ell_1, \ldots, \ell_n, \xi_\top(a_1))]^\alpha_\mathcal{I}$ regardless of the assignment α for the variables in a_1, which is virtually impossible to satisfy. Therefore, we propose a new interpretation-based method which is not subject to this weakness.

Let $\mathbb{B} = \{0, 1\}$. We define relations $\geqslant$ and $>$ on $\mathbb{N} \times \mathbb{B}$ as follows: for $\circ \in \{\geqslant, >\}$, $(n', b') \circ (n, b)$ if $n' \circ n$ and $b' \geqslant b$. Moreover, let π_1 and π_2 be the projections $\pi_1((n, b)) = n$ and $\pi_2((n, b)) = b$.

▶ **Definition 23.** A context-sensitive interpretation over $\mathbb{N} \times \mathbb{B}$ is a function $\mathcal{I}$ mapping each symbol $f \in \mathcal{F}$ of arity n to a function $\mathcal{I}_f$ from $(\mathbb{N} \times \mathbb{B})^n$ to $\mathbb{N} \times \mathbb{B}$, such that $\mathcal{I}_f$ is strictly monotone in its i-th argument for all $i \in \mu(f)$. Given a valuation α mapping each variable to an element of $\mathbb{N} \times \mathbb{B}$, the value $[t]^\alpha_\mathcal{I} \in \mathbb{N} \times \mathbb{B}$ of a term t is defined as usual ($[x]^\alpha_\mathcal{I} = \alpha(x)$ and $[f(t_1, \ldots, t_n)]^\alpha_\mathcal{I} = \mathcal{I}_f([t_1]^\alpha_\mathcal{I}, \ldots, [t_n]^\alpha_\mathcal{I})$). We say $\mathcal{I}$ is compatible with $\mathcal{R}$ if for all $\ell \to r \in \mathcal{R}$ and valuations α, $[\ell]^\alpha_\mathcal{I} > [r]^\alpha_\mathcal{I}$ if $\ell \to r \in \mathcal{R}$ is non-administrative and $[\ell]^\alpha_\mathcal{I} \geqslant [r]^\alpha_\mathcal{I}$ otherwise.

The primary purpose of the second component of (n, b) is to allow more sophisticated choices for $\mathcal{I}$. We easily see that if $s \to_{\mathcal{R}\,\mu} t$ then $[s]^\alpha_\mathcal{I} \geqslant [t]^\alpha_\mathcal{I}$ and $[s]^\alpha_\mathcal{I} > [t]^\alpha_\mathcal{I}$ if the employed rule is non-administrative. Consequently, $\mathsf{dh}(s, \to_{\mathcal{R},\mu}) \leqslant \pi_1([s]^\alpha_\mathcal{I})$ for any valuation α.

[2] We suspect that the equivalence goes both ways: If the derivation height of a term s in $\Xi(\mathcal{R})$ is N then a complexity-conscious reduction of length at least N exists starting at $\mathsf{label}(s)$. However, we have not yet confirmed this proposition as the proof—which is based on swapping rule applications at different positions—is non-trivial and Theorem 22 provides the direction which is most important in practice.

▶ **Example 24.** Continuing Example 19, we define

$$\mathcal{I}_\top = (0,1) \qquad \mathcal{I}_\bot = I_{\mathsf{true}} = \mathcal{I}_{\mathsf{false}} = \mathcal{I}_{\mathsf{0}} = (0,0) \qquad \mathcal{I}_{\mathsf{s}}((x,b)) = (x+1,0)$$
$$\mathcal{I}_{\mathsf{even}}((x,b),(y_1,b_1),(y_2,b_2),(y_3,b_3)) = (1+x+y_1+y_2+y_3+b_2\cdot 3^x + b_3 \cdot 3^x, 0)$$
$$\mathcal{I}_{\mathsf{odd}} = \mathcal{I}_{\mathsf{even_a}} = \mathcal{I}_{\mathsf{odd_a}} = \mathcal{I}_{\mathsf{even}} \qquad \mathcal{I}_{\mathsf{c}^1_i}((x,b),(y,d)) = (y,0) \quad \text{for all } i \in \{2,3,5,6\}$$

One easily checks that $\mathcal{I}$ satisfies the required monotonicity constraints. Moreover, all rewrite rules in $\Xi(\mathcal{R}_{\mathsf{even}})$ are oriented as required. For instance, for rule (2_2) we obtain

$$\begin{aligned} &1 + (x+1) + y + 0 + z + 1\cdot 3^{x+1} + b_z \cdot 3^{x+1} \\ &\qquad \geqslant 1 + (x+1) + y + (1 + x + 0 + 0 + 0 + 3^x + 3^x) + z + 0 + b_z \cdot 3^{x+1} \end{aligned}$$

which holds for all $x,y,z \in \mathbb{N}$ and $b_z \in \mathbb{B}$. All constructor terms are interpreted by linear polynomials with coefficients in $\{0,1\}$ and hence $\pi_1([s]_\mathcal{I}) \leqslant |t|$ for all ground constructor terms t. Therefore, the conditional runtime complexity $\mathsf{crc}_{\mathcal{R}_{\mathsf{even}}}(n)$ is bound by

$$\begin{aligned} &\max(\{\pi_1(\mathcal{I}_f((x_1,b_1),\ldots,(x_4,b_4))) \mid f \in \mathcal{F}_\mathcal{D} \text{ and } x_1+x_2+x_3+x_4 < n\}) \\ = \;&\max(\{1 + x_1 + x_2 + x_3 + x_4 + 2\cdot 3^{x_1} \mid x_1 + x_2 + x_3 + x_4 < n\}) \leqslant 3^n \end{aligned}$$

As observed before, the actual runtime complexity for this system is $O(2^n)$. In order to obtain this more realistic bound, we might observe that, starting from a basic term s, if $s \to^* t$ then the first argument of even and odd anywhere in t cannot contain defined symbols. Therefore, $\mathcal{I}_{\mathsf{even}}$ does not need to be monotone in its first argument. This observation is based on a result in [11], which makes it possible to impose a stronger replacement map μ.

As to derivational complexity, we observe that $[t]_\mathcal{I} \leqslant {}^n3$ (tetration, or $3\uparrow\uparrow n$ in Knuth's up-arrow notation) when t is an arbitrary ground term of size n. To obtain a more elementary bound, we will need more sophisticated methods, for instance assigning a compatible sort system and using the fact that all terms of sort int are necessarily constructor terms.

The interpretations in Example 24 may appear somewhat arbitrary, but in fact there is a recipe that we can most likely apply to many TRSs obtained from a CCTRS. The idea is to define the interpretation $\mathcal{I}$ as an extension of a "basic" interpretation $\mathcal{J}$ over $\mathbb{N}$. To do so, we choose for every symbol f of arity n in the original signature $\mathcal{F}$ interpretation functions $\mathcal{J}_f^0, \ldots, \mathcal{J}_f^{m_f} : \mathbb{N}^n \to \mathbb{N}$ such that $\mathcal{J}_f^0$ is strictly monotone in all its arguments, and the other $\mathcal{J}_f^j$ are weakly monotone. Similarly, for each rule ρ with $k > 0$ conditions we fix interpretation functions $\mathcal{J}_{c,\rho}^1, \ldots, \mathcal{J}_{c,\rho}^k$ with $\mathcal{J}_{c,\rho}^i : \mathbb{N}^{n+i} \to \mathbb{N}$ if c_ρ^i has arity $n+i$. These functions must be strictly monotone in the last argument. Based on these interpretations, we fix an interpretation for $\mathcal{G}$: $\mathcal{I}_\top = (0,1)$ and $\mathcal{I}_\bot = (0,0)$,

$$\mathcal{I}_f((x_1,b_1),\ldots,(x_n,b_n)) = (\mathcal{J}_f^0(x_1,\ldots,x_n),0)$$

for every $f \in \mathcal{F}_\mathcal{C}$ of arity n,

$$\begin{aligned} &\mathcal{I}_f((x_1,b_1),\ldots,(x_n,b_n),(y_1,d_1),\ldots,(y_{m_f},d_{m_f})) \\ &\qquad = (\mathcal{J}_f^0(x_1,\ldots,x_n) + y_1 + \cdots + y_{m_f} + \sum_{i=1}^{m_f}(d_i \cdot \mathcal{J}_f^i(x_1,\ldots,x_n)),0) \end{aligned}$$

and $\mathcal{I}_{f_\mathsf{a}} = \mathcal{I}_f$ for every $f \in \mathcal{F}_\mathcal{D}$ of arity n, and

$$\mathcal{I}_{c_\rho^i}((x_1,b_1),\ldots,(x_{n+i},b_{n+i})) = (\mathcal{J}_{c,\rho}^i(x_1,\ldots,x_{n+i}),0)$$

for every symbol c^i_ρ. It is not hard to see that $\mathcal{I}$ satisfies the monotonicity requirements. In practice, this interpretation assigns to a $\top$ symbol in a term $f(s_1,\dots,s_n,\dots,\top,\dots)$ the value $\mathcal{J}^i_f(s_1,\dots,s_n)$ and ignores the $\mathbb{B}$ component otherwise. Using this interpretation for the rules in Definition 18, the inequalities we obtain can be greatly simplified. Obviously, $[\ell]^\alpha_\mathcal{I} \geqslant [r]^\alpha_\mathcal{I}$ is satisfied for all rules obtained from clauses (5_ρ) and (6_ρ). For the other clauses we obtain the following requirements for each rule ρ: $f(\ell_1,\dots,\ell_n) \to r \Leftarrow a_1 \approx b_1,\dots,a_k \approx b_k$ in the original system $\mathcal{R}$, with ρ the i-th rule in $\mathcal{R}{\restriction}f$:

$$\mathcal{J}^0_f([\ell_1]),\dots,[\ell_n]) + \mathcal{J}^i_f([\ell_1],\dots,[\ell_n]) > \pi_1([\xi_\top(r)]^\alpha_\mathcal{I}) \tag{1_ρ}$$

$$\mathcal{J}^i_f([\ell_1],\dots,[\ell_n]) \geqslant \mathcal{J}^1_{c,\rho}([\ell_1],\dots,[\ell_n],\pi_1([\xi_\top(r)]^\alpha_\mathcal{I})) \tag{2_ρ}$$

$$\mathcal{J}^0_f([\ell_1],\dots,[\ell_n]) + \mathcal{J}^k_{c,\rho}([\ell_1],\dots,[\ell_n],[b_1],\dots,[b_k]) > \pi_1([\xi_\top(r)]^\alpha_\mathcal{I}) \tag{3_ρ}$$

$$\mathcal{J}^j_f([\ell_1],\dots,[\ell_n],[b_1],\dots,[b_j]) \geqslant \mathcal{J}^{j+1}_f([\ell_1],\dots,[\ell_n],[b_1],\dots,[b_j],\pi_1([\xi_\top(r)]^\alpha_\mathcal{I})) \tag{4_ρ}$$

for the same cases of k and j as in Definition 18. Here, $[l_j]$ and $[b_j]$ are short-hand notation for $\pi_1([l_j]^\alpha_\mathcal{I})$ and $\pi_1([b_j]^\alpha_\mathcal{I})$, respectively. Note that $[f(t_1,\dots,t_n)] = \mathcal{J}_f([t_1],\dots,[t_n])$ for constructor terms $f(s_1,\dots,s_n)$. Additionally observing that

$$\pi_1([\xi_\top(f(t_1,\dots,t_n))]^\alpha_\mathcal{I}) = \sum_{i=0}^{m_f} \mathcal{J}^i_f(\pi_1([\xi_\top(t_1)]^\alpha_\mathcal{I}),\dots,\pi_1([\xi_\top(t_n)]^\alpha_\mathcal{I}))$$

we can obtain bounds for the derivation height without ever calculating $\xi_\top(t)$.

▶ **Example 25.** We derive an upper bound for the runtime complexity of $\mathcal{R}_{\mathsf{fib}}$. Following the recipe explained above, we fix $\mathcal{J}^1_+(x,y) = \mathcal{J}^2_+(x,y) = \mathcal{J}^1_{\mathsf{fib}}(x) = 0$. Writing $K = \mathcal{J}^0_{\mathsf{0}}$, $S = \mathcal{J}^0_{\mathsf{s}}$, $P = \mathcal{J}^0_+$, $F = \mathcal{J}^0_{\mathsf{fib}}$, $G = \mathcal{J}^2_{\mathsf{fib}}$, $A = \mathcal{J}^0_{\langle\cdot\rangle}$, $C = \mathcal{J}^1_{c,4}$, and $D = \mathcal{J}^2_{c,4}$, we get the constraints

$$P(K,y) > y \qquad P(S(x),y) > S(P(x,y)) \qquad F(0) > A(0,S(0))$$

for the unconditional rules of $\mathcal{R}_{\mathsf{fib}}$ and

$$\begin{aligned} G(S(x)) &\geqslant C(S(x),F(x)+G(x)) \\ F(S(x)) + D(S(x),A(y,z),w) &> A(z,w) \\ C(S(x),A(y,z)) &\geqslant D(S(x),A(y,z),P(y,z)) \end{aligned}$$

for the conditional rule $\mathsf{fib}(\mathsf{s}(x)) \to \langle z,w\rangle \ \Leftarrow\ \mathsf{fib}(x) \approx \langle y,z\rangle,\ y+z \approx w$. The functions P, F, A, and S must be strictly monotone in all arguments, whereas for C and D strict monotonicity is required only for the last argument. Choosing $K = 0$, $S(x) = x+1$, $P(x,y) = 2x+y+1$, $A(x,y) = x+y+1$, $C(x,y) = 3y$, and $D(x,y,z) = y+z$ to eliminate as many arguments as possible, the constraints simplify to

$$F(0) > 3 \qquad G(x+1) \geqslant 3\,F(x) + 3\,G(x) \qquad F(x+1) > 0$$

Choosing $F(x) = x+4$ leaves the constraint $G(x+1) \geqslant 3x + 3\,G(x) + 12$, which is satisfied (e.g.) by taking $G(x) = 4^{x+1}$, which results in a conditional runtime complexity of $O(4^x)$.

As in Example 24, we can obtain a more precise bound using [11], by observing that runtime complexity is not altered if we impose a replacement map μ with $\mu(\mathsf{fib}) = \varnothing$, which allows us to choose a non-monotone function for F. More sophisticated methods may lower the bound further.

7 Related Work

We are not aware of any attempt to study the complexity of conditional rewriting, but numerous transformations from CTRSs to TRSs have been proposed in the literature. They can roughly be divided into so-called *unravelings* and *structure-preserving* transformations. The former were coined by Marchiori [17] and have been extensively investigated (e.g. [18, 21, 23, 24, 26]), mainly to establish (operational) termination and confluence of the input CTRS. The latter originate from Viry [29] and improved versions were proposed in [1, 7, 9].

The transformations that are known to transform CTRSs into TRSs such that (simple) termination of the latter implies quasi-decreasingness of the former, are natural candidates for study from a complexity perspective. We observe that unravelings are not suitable in this regard. For instance, the unraveling from [18] transforms the CCTRS $\mathcal{R}_{\mathsf{even}}$ into

$$\begin{array}{lll} \mathsf{even}(0) \to \mathsf{true} & \mathsf{even}(\mathsf{s}(x)) \to \mathsf{U}_1(\mathsf{odd}(x), x) & \mathsf{U}_1(\mathsf{true}, x) \to \mathsf{true} \\ & \mathsf{even}(\mathsf{s}(x)) \to \mathsf{U}_2(\mathsf{even}(x), x) & \mathsf{U}_2(\mathsf{true}, x) \to \mathsf{false} \\ \mathsf{odd}(0) \to \mathsf{false} & \mathsf{odd}(\mathsf{s}(x)) \to \mathsf{U}_3(\mathsf{odd}(x), x) & \mathsf{U}_3(\mathsf{true}, x) \to \mathsf{false} \\ & \mathsf{odd}(\mathsf{s}(x)) \to \mathsf{U}_4(\mathsf{even}(x), x) & \mathsf{U}_4(\mathsf{true}, x) \to \mathsf{true} \end{array}$$

This TRS has a linear runtime complexity, which is readily confirmed by a complexity tool like TCT [2]. As the conditional runtime complexity is exponential, the transformation is not suitable for measuring conditional complexity. The same holds for the transformation in [3].

We do not know whether structure-preserving transformations can be used for conditional complexity. If we apply the transformations from [7] and [9] to $\mathcal{R}_{\mathsf{even}}$ we obtain TRSs for which complexity tools fail to establish an upper bound on the runtime and derivational complexities. The latter is also true for the TRS that we obtain from $\Xi(\mathcal{R}_{\mathsf{even}})$ by lifting the context-sensitive restriction, but this is solely due to the (current) lack of support in complexity tools for techniques that yield non-polynomial upper bounds.

8 Conclusion and Future Work

In this paper we have defined a first notion of complexity for conditional term rewriting, which takes failed calculations into account as any automatic rewriting engine would. We have also defined a transformation to unconditional context-sensitive TRSs, and shown how this transformation can be used to find upper bounds for conditional complexity using traditional interpretation-based methods.

There are several possible directions to continue our research.

Weakening restrictions. An obvious direction for future research is to broaden the class of CTRSs we consider. This requires careful consideration. The correctness of the transformation Ξ depends on the limitations that we impose on CCTRSs. However, it may be possible to weaken the restrictions and still obtain at least a sound (if perhaps not complete) transformation. More importantly, though, as discussed in Section 2, the restrictions on the conditions are needed to justify our complexity notion. For the same reason, Lemma 5 (which relies on the left-hand sides being linear basic terms) needs to be preserved.

Alternatively, we might consider different strategies for the evaluation of conditions. For example, if all restrictions except for variable freshness in the conditions are satisfied, we could impose the strategy that conditions are always evaluated to normal form. For instance, given a CTRS with a rule

$$\mathsf{f}(y, z) \to r \;\Leftarrow\; y \approx \mathsf{g}(x), z \approx x$$

rewriting a term $\mathsf{f}(\mathsf{g}(0+0),0)$ would then bind 0 rather than $0+0$ to x in the first condition (assuming sensible rules for $+$), allowing the second condition to succeed. This approach could also be used to handle non-left-linear rules, for instance transforming a rule $\mathsf{f}(\mathsf{g}(x),x) \to r$ into the left-linear rule above.

It would take a little more effort to handle non-confluent systems in a way that does not allow us to give up on rule applications when it is still possible that their conditions can be satisfied. As a concrete example, consider the CCTRS

$$\mathsf{a}(x) \to 0 \qquad \mathsf{a}(x) \to 1 \qquad \mathsf{f}(x) \to \mathsf{g}(y,y) \Leftarrow \mathsf{a}(x) \approx y \qquad \mathsf{h}(x) \to x \Leftarrow \mathsf{f}(x) \approx \mathsf{g}(0,1)$$

Even though $\mathsf{f}(0) \to^* \mathsf{g}(0,0)$ and $\mathsf{g}(0,0)$ is an instance of $\mathsf{g}(x,0) \in \mathsf{AP}(\mathsf{g}(0,1))$, we should not conclude that $\mathsf{h}(0)$ cannot be reduced. We could instead attempt to calculate all normal forms in order to satisfy conditions, but if we do this for the third rule, we would not find the desired reduction $\mathsf{f}(0) \to \mathsf{g}(\mathsf{a}(0),\mathsf{a}(0))$. Thus, to confirm that $\mathsf{h}(0)$ can be reduced, we need to determine *all* (or at least, all *most general*) reducts of the left-hand sides of conditions. This will likely give very high complexity bounds, however. It would be interesting to investigate how real conditional rewrite engines like Maude handle this problem.

Rules with branching conditions. Consider the following variant of $\mathcal{R}_{\mathsf{even}}$:

$$\begin{array}{rlr rlr}
\mathsf{even}(0) \to \mathsf{true} & & (1) & \mathsf{odd}(0) \to \mathsf{false} & & (4)\\
\mathsf{even}(\mathsf{s}(x)) \to \mathsf{true} & \Leftarrow \mathsf{odd}(x) \approx \mathsf{true} & (2) & \mathsf{odd}(\mathsf{s}(x)) \to \mathsf{true} & \Leftarrow \mathsf{even}(x) \approx \mathsf{true} & (5)\\
\mathsf{even}(\mathsf{s}(x)) \to \mathsf{false} & \Leftarrow \mathsf{odd}(x) \approx \mathsf{false} & (3) & \mathsf{odd}(\mathsf{s}(x)) \to \mathsf{false} & \Leftarrow \mathsf{even}(x) \approx \mathsf{false} & (6)
\end{array}$$

Evaluating $\mathsf{even}(\mathsf{s}^9(0))$ with rule (2) causes the calculation of the normal form false of $\mathsf{odd}(\mathsf{s}^8(0))$, before concluding that the rule does not apply. In our definitions (of $\rightharpoonup$ and Ξ), and conform to the behavior of Maude, we would dismiss the result and continue trying the next rule. In this case, that means recalculating the normal form of $\mathsf{odd}(\mathsf{s}^8(0))$, but now to verify whether rule (3) applies. There is clearly no advantage in treating the rules (2) and (3) separately. Instead, we could consider rules such as these to be *grouped*; if the left-hand side matches, we try the corresponding condition, and the result determines whether we proceed with (2), (3), or fail. Future definitions of complexity for *sensible* conditional rewriting should take optimizations like these into account.

Improving the transformation. With regard to the transformation Ξ, it should be possible to obtain smaller resulting systems using various optimizations, such as reducing the set AP of anti-patterns using typing considerations, or leaving defined symbols untouched when they are only defined by unconditional rules. As observed in footnote 2, either proving that Ξ preserves complexity, or improving it so that it does, would be interesting.

Complexity methods. While the interpretation recipe from Section 6 has the advantage of immediately eliminating rules (5_ρ) and (6_ρ), it is not strictly necessary to always map f and f_{a} to the same function. With alternative recipes, we may be able to more fully take advantage of the context-sensitivity of the transformed system, and handle different examples.

Besides interpretations into $\mathbb{N}$, there are many other complexity techniques which could possibly be adapted to context-sensitivity and to handle the $\top$ symbols appearing in $\Xi(\mathcal{R})$. As for tool support, we believe that it should not be hard to integrate support for conditional rewriting into existing tools. We hope that, in the future, developers of complexity tools will branch out to context-sensitive rewriting and not shy away from exponential upper bounds.

Acknowledgments. We thank the anonymous reviewers, whose constructive comments have helped to improve the presentation.

References

1. S. Antoy, B. Brassel, and M. Hanus. Conditional narrowing without conditions. In *Proc. 5th PPDP*, pages 20–31, 2003. doi:10.1145/888251.888255.
2. M. Avanzini and G. Moser. Tyrolean complexity tool: Features and usage. In *Proc. 24th RTA*, volume 21 of *Leibniz International Proceedings in Informatics*, pages 71–80, 2013. doi:10.4230/LIPIcs.RTA.2013.71.
3. J. Avenhaus and C. Loría-Sáenz. On conditional rewrite systems with extra variables and deterministic logic programs. In *Proc. 5th LPAR*, volume 822 of *LNAI*, pages 215–229, 1994. doi:10.1007/3-540-58216-9_40.
4. F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.
5. G. Bonfante, A. Cichon, J.-Y. Marion, and H. Touzet. Algorithms with polynomial interpretation termination proof. *JFP*, 11(1):33–53, 2001.
6. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude – A High-Performance Logical Framework*, volume 4350 of *LNCS*. Springer, 2007.
7. T.F. Şerbănuţă and G. Roşu. Computationally equivalent elimination of conditions. In *Proc. 17th RTA*, volume 4098 of *LNCS*, pages 19–34, 2006. doi:10.1007/11805618_3.
8. J. Giesl, M. Brockschmidt, F. Emmes, F. Frohn, C. Fuhs, C. Otto, M. Plücker, P. Schneider-Kamp, S. Swiderski, and R. Thiemann. Proving termination of programs automatically with AProVE. In *Proc. 7th IJCAR*, volume 8562 of *LNCS*, pages 184–191, 2014. doi:10.1007/978-3-319-08587-6_13.
9. K. Gmeiner and N. Nishida. Notes on structure-preserving transformations of conditional term rewrite systems. In *Proc. 1st WPTE*, volume 40 of *OASICS*, pages 3–14, 2014. doi:10.4230/OASIcs.WPTE.2014.3.
10. 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. doi:10.1007/978-3-540-71070-7_32.
11. N. Hirokawa and G. Moser. Automated complexity analysis based on context-sensitive rewriting. In *Proc. Joint 25th RTA and 12th TLCA*, volume 8560 of *LNCS*, pages 257–271, 2014. doi:10.1007/978-3-319-08918-8_18.
12. D. Hofbauer and C. Lautemann. Termination proofs and the length of derivations (preliminary version). In *Proc. 3rd RTA*, volume 355 of *LNCS*, pages 167–177, 1989. doi:10.1007/3-540-51081-8_107.
13. S. Lucas. Context-sensitive computations in functional and functional logic programs. *Journal of Functional and Logic Programming*, 1998(1), 1998.
14. S. Lucas. Context-sensitive rewriting strategies. *Information and Computation*, 178(1):294–343, 2002. doi:10.1006/inco.2002.3176.
15. S. Lucas, C. Marché, and J. Meseguer. Operational termination of conditional term rewriting systems. *Information Processing Letters*, 95(4):446–453, 2005. doi:10.1016/j.ipl.2005.05.002.
16. S. Lucas and J. Meseguer. 2D dependency pairs for proving operational termination of CTRSs. In *Proc. 10th WRLA*, volume 8663 of *LNCS*, pages 195–212, 2014. doi:10.1007/978-3-319-12904-4_11.
17. M. Marchiori. Unravelings and ultra-properties. In M. Hanus and M. Rodríguez-Artalejo, editors, *Proc. 5th ICALP*, volume 1139 of *LNCS*, pages 107–121. Springer, 1996. doi:10.1007/3-540-61735-3_7.

18 M. Marchiori. On deterministic conditional rewriting. Computation Structures Group Memo 405, MIT Laboratory for Computer Science, 1997.
19 A. Middeldorp, G. Moser, F. Neurauter, J. Waldmann, and H. Zankl. Joint spectral radius theory for automated complexity analysis of rewrite systems. In *Proc. 4th CAI*, volume 6742 of *LNCS*, pages 1–20, 2011. doi:10.1007/978-3-642-21493-6_1.
20 G. Moser and A. Schnabl. The derivational complexity induced by the dependency pair method. *Logical Methods in Computer Science*, 7(3), 2011. doi:10.2168/LMCS-7(3:1)2011.
21 N. Nishida, M. Sakai, and T. Sakabe. Soundness of unravelings for conditional term rewriting systems via ultra-properties related to linearity. *Logical Methods in Computer Science*, 8:1–49, 2012. doi:10.2168/LMCS-8(3:4)2012.
22 L. Noschinski, F. Emmes, and J. Giesl. Analyzing innermost runtime complexity of term rewriting by dependency pairs. *Journal of Automated Reasoning*, 51(1):27–56, 2013. doi:10.1007/s10817-013-9277-6.
23 E. Ohlebusch. Transforming conditional rewrite systems with extra variables into unconditional systems. In *Proc. 6th LPAR*, volume 1705 of *LNCS*, pages 111–130, 1999. doi:10.1007/3-540-48242-3_8.
24 E. Ohlebusch. *Advanced Topics in Term Rewriting*. Springer, 2002. doi:10.1007/978-1-4757-3661-8.
25 F. Schernhammer and B. Gramlich. VMTL – a modular termination laboratory. In *Proc. 20th RTA*, volume 5595 of *LNCS*, pages 285–294, 2009. doi:10.1007/978-3-642-02348-4_20.
26 F. Schernhammer and B. Gramlich. Characterizing and proving operational termination of deterministic conditional term rewriting systems. *Journal of Logic and Algebraic Programming*, 79(7):659–688, 2010. doi:10.1016/j.jlap.2009.08.001.
27 T. Sternagel and A. Middeldorp. Conditional confluence (system description). In *Proc. Joint 25th RTA and 12th TLCA*, volume 8560 of *LNCS*, pages 456–465, 2014. doi:10.1007/978-3-319-08918-8_31.
28 Terese. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.
29 P. Viry. Elimination of conditions. *Journal of Symbolic Computation*, 28(3):381–401, 1999. doi:10.1006/jsco.1999.0288.
30 H. Zankl and M. Korp. Modular complexity analysis for term rewriting. *Logical Methods in Computer Science*, 10(1:19):1–33, 2014. doi:10.2168/LMCS-10(1:19)2014.

Constructing Orthogonal Designs in Powers of Two: Gröbner Bases Meet Equational Unification

Ilias Kotsireas[1], Temur Kutsia[2], and Dimitris E. Simos[*3]

1 **Wilfrid Laurier University**
Waterloo, Ontario, Canada
ikotsire@wlu.ca

2 **RISC, Johannes Kepler University**
Altenbergerstrasse 69, A-4040 Linz, Austria
kutsia@risc.jku.at

3 **SBA Research**
Favoritenstrasse 16, A-1040 Vienna, Austria
dsimos@sba-research.org

Abstract

In the past few decades, design theory has grown to encompass a wide variety of research directions. It comes as no surprise that applications in coding theory and communications continue to arise, and also that designs have found applications in new areas. Computer science has provided a new source of applications of designs, and simultaneously a field of new and challenging problems in design theory. In this paper, we revisit a construction for orthogonal designs using the multiplication tables of Cayley-Dickson algebras of dimension 2^n. The desired orthogonal designs can be described by a system of equations with the aid of a Gröbner basis computation. For orders greater than 16 the combinatorial explosion of the problem gives rise to equations that are unfeasible to be handled by traditional search algorithms. However, the structural properties of the designs make this problem possible to be tackled in terms of rewriting techniques, by equational unification. We establish connections between central concepts of design theory and equational unification where equivalence operations of designs point to the computation of a minimal complete set of unifiers. These connections make viable the computation of some types of orthogonal designs that have not been found before with the aforementioned algebraic modelling.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems, F.4.1 Mathematical Logic, G.2.1 Combinatorics

Keywords and phrases Orthogonal designs, unification theory, algorithms, Gröbner bases.

Digital Object Identifier 10.4230/LIPIcs.RTA.2015.241

1 Introduction

Orthogonal designs are an important class of combinatorial designs. They are of great interest in applications for wireless communication [15] and in statistics [11]. Even though there exist many combinatorial constructions for orthogonal designs [6], ones that originate from Cayley-Dickson algebras [7,8] have not been explored enough. In particular, as we exemplify in this work, these algebras can provide a general framework for obtaining orthogonal designs

* The work of the third author was carried out during the tenure of an ERCIM "Alain Bensoussan" Fellowship Programme.

© Ilias Kotsireas, Temur Kutsia, and Dimitris E. Simos;
licensed under Creative Commons License CC-BY

for powers of two. Designs in these orders are also of theoretical interest due to their connection to the asymptotic existence of orthogonal designs [6].

Contribution. In this paper, after revisiting past methods we formulate orthogonal design problems in terms of equational unification. In particular, the Cayley-Dickson formulation gives rise to a polynomial system of equations of a specific form, that due to its size cannot be handled by traditional search algorithms. By establishing and proving connections between central concepts of the theory of orthogonal designs and equational unification, we are able to completely tackle these systems of equations, where each solution of them gives rise to an orthogonal design. The efficiency of the unification algorithms needed to solve the corresponding orthogonal design problems is evident also by the fact that we found some types of orthogonal designs, that were not known before with this algebraic modelling of Cayley-Dickson algebras. Our approach not only reports the orthogonal designs, but also constructs the corresponding design matrices. In this way, we *always* give a *constructive* solution to the problem which is not always the case with other approaches used in design theory as we explain in the last section. Last but not least, we would like to emphasize the novel connections we established between base orthogonal designs, a notion introduced in this paper, and minimal complete sets of unifiers, as a means to advance the knowledge in the field of design theory (orthogonal design equivalence among other topics) and also benefit from the algorithmic notions of unification theory as we applied them in this paper.

Structure of the paper. In Section 2 we give some details regarding orthogonal designs and list some of their applications. Afterwards, in Section 3 we detail the algebraic framework for constructing orthogonal designs via computation algebra where we also introduce some new terms for designs. Some first connections with unification theory are also shown. In the subsequent section we give some basic notions of unification theory while in Section 5 we establish additional connections of designs with concepts of unification theory that allow us to formulate orthogonal design problems as unification problems. In Section 6 we describe the unification algorithms we developed for solving the unification problems and in the last section, we translate the solutions obtained via unifiers back to orthogonal designs.

2 Orthogonal Designs

In this section, we give some details regarding orthogonal designs. We provide the necessary definitions and related concepts that will be needed for our approach and list also some applications of orthogonal designs that are of broader interest.

2.1 Definitions and Related Concepts

An *orthogonal design* of order n and type $(s_1, s_2, \ldots, s_u)$ $(s_i > 0)$, denoted $OD(n; s_1, s_2, \ldots, s_u)$, on the commuting variables $x_1, x_2, \ldots, x_u$, is an $n \times n$ matrix D with entries from $\{0, \pm x_1, \pm x_2, \ldots, \pm x_u\}$ such that

$$DD^T = \left(\sum_{i=1}^{u} s_i x_i^2 \right) I_n$$

where by I_n we denote the identity matrix of order n. Alternatively, the rows of D are formally orthogonal and each row has precisely s_i entries of the type $\pm x_i$. The design matrix D, may be considered as a matrix with entries in the field of quotients of the integral domain

$\mathbb{Z}[x_1, x_2, \ldots, x_u]$. In [5], where this was first defined, it was mentioned that

$$D^T D = \left(\sum_{i=1}^{u} s_i x_i^2\right) I_n$$

and so our alternative description of D applies equally well to the columns of D. It was also shown in [5] that $u \leq \rho(n)$, where $\rho(n)$ (Radon's function) is defined by $\rho(n) = 8c + 2^d$, when $n = 2^a b$, b odd, $a = 4c + d$, $0 \leq d < 4$. D will be called a *full orthogonal design*, if $n = s_1 + s_2 + \ldots + s_u$. Due to the Equating-Killing Lemma, given below, which is of central importance in the theory of Orthogonal Designs, one is interested in full orthogonal designs.

▶ **Lemma 1** (The Equating and Killing Lemma, Geramita and Seberry [6]). *If D is an orthogonal design $OD(n; s_1, s_2, \ldots, s_u)$ on the commuting variables $\{0, \pm x_1, \pm x_2, \ldots, \pm x_u\}$ then there exists an orthogonal design:*
(i) *$OD(n;\ s_1, s_2, \ldots, s_i + s_j, \ldots, s_u)$ ($s_i = s_j$, equating variables)*
(ii) *$OD(n; s_1, s_2, \ldots, s_{j-1}, s_{j+1}, \ldots, s_u)$ ($s_j = 0$, killing variables)*
on the $u - 1$ commuting variables $\{0, \pm x_1, \pm x_2, \ldots, \pm x_{j-1}, \pm x_{j+1}, \ldots, \pm x_u\}$.

We also list the Doubling Lemma, which will be needed in the last section of the paper.

▶ **Lemma 2** (The Doubling Lemma, Geramita and Seberry [6]). *If there exists an orthogonal design of order n and type $(s_1, s_2, \ldots, s_u)$, then there exists orthogonal designs of type*
(i) *$(e_1 s_1, e_2 s_2, \ldots, e_u s_u)$ where $e_i = 1$ or 2,*
(ii) *$(s_1, s_1, f s_2, \ldots, f s_u)$ where $f = 1$ or 2.*

▶ **Example 3.** We give an example of some small orthogonal designs, and how we can obtain one from another due to Lemma 1 and related equivalence operations.

$$\underset{OD(2;1,1)}{\begin{bmatrix} x_1 & x_2 \\ x_2 & -x_1 \end{bmatrix}}, \underset{OD(4;1,1,1,1)}{\begin{bmatrix} x_1 & -x_2 & -x_3 & -x_4 \\ x_2 & x_1 & -x_4 & x_3 \\ x_3 & x_4 & x_1 & -x_2 \\ x_4 & -x_3 & x_2 & x_1 \end{bmatrix}}, \underset{OD(4;1,1,2)}{\begin{bmatrix} x_1 & x_2 & x_2 & x_4 \\ -x_2 & x_1 & x_4 & -x_2 \\ -x_2 & -x_4 & x_1 & x_2 \\ -x_4 & x_2 & -x_2 & x_1 \end{bmatrix}}, \underset{OD(4;1,1)}{\begin{bmatrix} x_1 & 0 & -x_3 & 0 \\ 0 & x_1 & 0 & x_3 \\ c & 0 & x_1 & 0 \\ 0 & -x_3 & 0 & x_1 \end{bmatrix}}$$

- $OD(4; 1, 1, 2)$ can be obtained from $OD(4; 1, 1, 1, 1)$ by setting $x_3 = -x_2$ in its design matrix.
- $OD(4; 1, 1)$ can be obtained from $OD(4; 1, 1, 1, 1)$ by setting $x_2 = x_4 = 0$ in its design matrix.

It is important to note here that in the first case the transformation is composed by the equating operation of the Equating and Killing Lemma and also changing the sign of the variable. The last operation leaves invariant the type of the design, however changes the design matrix. We describe more formally *equivalence of orthogonal designs* taken from [18].

Given two designs D_1 and D_2 of the same order, we say that D_2 is a *variant* of D_1, if it is obtained from D_1 by the following operations, performed in any order and any number of times:

1. Multiply one row (one column) by -1.
2. Swap two rows (columns).
3. Rename or negate a variable throughout the design.

It is easy to prove that the relation of being a variant is an equivalence relation. Below we write $D_1 \simeq D_2$ to express this fact. Note also that if $D_1 \simeq D_2$, then D_1 and D_2 have the same type. This follows directly from the definition of orthogonal design.

The general discussion of equivalence of orthogonal designs is very difficult because of the lack of a nice canonical form. It also means that it is quite difficult to decide whether or not two given orthogonal designs of the same order are equivalent. To the best of our knowledge, there has been little effort contributing at this point. In [18], where the above mentioned notion of equivalence was introduced, some designs for small orders have been classified by hand.

The approach proposed in this paper, besides providing a systematic search method for orthogonal designs in order of powers of two, also exhibits some interesting connections between the Equating and Killing Lemma and equivalence of orthogonal designs on the one hand, and fundamental concepts of unification theory such as subsumption and equi-generality on the other hand, as we can see below in Section 5.

2.2 Applications of Orthogonal Designs

We give some references to works describing applications of orthogonal designs. We do not aim to provide a comprehensive, or by all means complete, treatment of the subject, as this is not the purpose of the present paper. We are merely interesting in giving a flavor of the many different application areas involved, in order to exhibit that while orthogonal designs are specialized types of combinatorial structures their applications are of a broader interest.

As first noted in [11], orthogonal designs are used in statistics where they generate optimal statistical designs used in weighing experiments. A special case of orthogonal designs, the so called *Hadamard matrices* play an important role also in coding theory where they have been used to generate the so called Hadamard codes ([10]), i.e. error-correcting codes that correct the maximum number of errors. It is worthwhile to note that, a Hadamard code was used during the 1971 space probe Mariner 9 mission by NASA to correct for picture transmission error. The Mariner 9 mission and the Coding Theory used in that project are the subjects of [12] and [16]. Recently, complex orthogonal designs were used in [15] to generate space-time block codes, a relatively new paradigm for communication over Rayleigh fading channels using multiple transmit antennas. In this case, the orthogonal structure of the space-time block code derived by the orthogonal design gives a maximum-likelihood decoding algorithm which is based only on linear processing at the receiver.

Orthogonal designs are also used in telecommunications where they generate sequences used in digital communications and in optics for the improvement of the quality and resolution of image scanners. More details, regarding their applications in communications and signal/image processing can be found in [6, 13, 17].

3 Orthogonal Designs via Computational Algebra

In this section, we revisit a construction for orthogonal designs based on the multiplication tables of algebras of order n. These multiplication tables are used to construct right multiplication matrices that in the sequel are used to construct orthogonal designs. Using the right multiplication operator is a way to overcome the obstacle of non-associativity of the algebra. Non-associativity is an obstacle, because it is incompatible with the existence of matrix representations, that we could use directly to construct orthogonal designs. To circumvent this obstacle we use the right multiplication operator, as it seems that left multiplication is not suitable for our purposes.

First, we give an account of the classical Williamson construction for orthogonal designs [2], from the point of view of quaternions, following Baumert and Hall to be able to use it as reference in subsequent constructions.

A basis for quaternions is given by the four elements $1, i, j, k$, having the properties

$$i^2 = -1,\ j^2 = -1,\ k^2 = -1,\ ij = k,\ ji = -k,\ ik = -j,\ ki = j,\ jk = i,\ kj = -i.$$

These properties are enough to specify the full multiplication table for the four basis elements. We note that quaternion multiplication is not commutative.

To associate a 4×4 matrix to each basis element, we use the right multiplication operator on the column vector $v = \begin{bmatrix} 1 & i & j & k \end{bmatrix}^t$. Then the right multiplications $v \cdot 1$, $v \cdot i$, $v \cdot j$, $v \cdot k$, give rise to the following four 4×4 matrices respectively:

$$q_1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad q_2 = \begin{bmatrix} 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 \\ 0 & 0 & 1 & 0 \end{bmatrix},$$

$$q_3 = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \end{bmatrix}, \quad q_4 = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \end{bmatrix}.$$

Let A, B, C, D be commuting variables. Then the sum

$$Aq_1 + Bq_2 + Cq_3 + Dq_4$$

is equal to the classical Williamson array

$$H_4 = \begin{pmatrix} A & B & C & D \\ -B & A & -D & C \\ -C & D & A & -B \\ -D & -C & B & A \end{pmatrix}$$

which has the property

$$H_4 H_4^T = (A^2 + B^2 + C^2 + D^2) I_4.$$

The matrix H is the design matrix of an $OD(4; 1, 1, 1, 1)$.

The Cayley-Dickson process allows us to obtain an algebra of dimension $2n$ from an algebra of dimension n, see [4]. One limitation of this process is that restricts our method to study ODs in powers of two. By applying the Cayley-Dickson process successively to the algebras of quaternions we get octonions and sedenions [7]. Repeating the Cayley-Dickson process to the algebra of sedenions one obtains a Cayley-Dickson algebra of dimension 32 and doing the same for the latter algebra we can obtain a Cayley-Dickson algebra of order 64 [8].

3.1 Cayley-Dickson Orthogonal Designs

It is important to note that the Cayley-Dickson process essentially constructs the multiplication tables we need to model orthogonal designs. Now we describe a generic formulation of

the algebraic modelling with multiplication tables of appropriate Cayley-Dickson algebras of order n to obtain orthogonal designs of order n.

Take a Cayley-Dickson algebra of dimension n with basis $e_0 = 1, e_1, \ldots, e_{n-1}$. To associate an $n \times n$ matrix to each basis element, we use the right multiplication operator on the column vector $v = \begin{bmatrix} 1 & e_1 & \ldots & e_{n-1} \end{bmatrix}^t$. Then the n right multiplications $v \cdot e_0,\ v \cdot e_1,\ \ldots,\ v \cdot e_{n-1}$ give rise to n matrices $q_0, \ldots, q_{n-1}$ of order n. Let $A_1, \ldots, A_n$ be commuting variables. Then the sum $A = \sum_{i=0}^{n-1} A_{i+1} q_i$ is equal to an $n \times n$ matrix with the property that the diagonal elements of AA^T are all equal to $\sum_{i=1}^{n} A_i^2$, but whose other elements are not necessarily all zero.

By requiring that all elements of AA^T (except the diagonal ones) are equal to zero, we obtain a polynomial system of equations in the set of variables $\{A_1, \ldots, A_n\}$. We define this problem as the *Cayley-Dickson Orthogonal Design* (CDOD) problem.

To represent solutions, we introduce a special kind of mapping that we call *substitution mapping* or, simply, a *substitution*. Formally, a substitution from a set S_1 to a set $S_2 \supseteq S_1$ is a mapping from S_1 to S_2 which is identity almost everywhere. We use lower case Greek letters to denote them. The identity substitution is denoted by ε. The *domain* and the *range* of a substitution σ are defined, respectively, as $dom(\sigma) := \{u \mid u \in S_1, u \neq \sigma(u)\}$ and $ran(\sigma) := \cup_{u \in dom(\sigma)} \{\sigma(u)\}$. A substitution, usually, is represented as a function by a finite set of bindings of variables in its domain. For instance, a substitution σ is represented as $\{u \mapsto \sigma(u) \mid u \in dom(\sigma)\}$.

It is important to highlight that we seek solutions of CDOD's in an endomorphic form, i.e., substitutions from a set S to itself. For a CDOD of order n, this set is $\{A_1, \ldots, A_n\}$. Moreover, the domain and the range of such substitutions should be disjoint, i.e., the substitutions should be idempotent. These requirements are justified by the following:

- Mapping of variables A_i to variables A_j, for $i, j \in \{1, \ldots, n\}$, is due to the fact that we force the matrix A to be an orthogonal design and by definition the diagonal elements give rise to a quadratic form that is a sum of squares.
- In particular, if several variables map to the same variable, it is the analogue of the equating operation of the Equating-Killing Lemma for orthogonal designs for the equations that are produced by the algebraic modelling. It is clear from the context that equating variables in the polynomial system of equations, is the same as equating variables in the design matrix representation.

▶ **Theorem 4.** *Let $n = 2^m$ for some $m > 0$. Any endomorphic idempotent solution to* CDOD *of order n gives rise to an orthogonal design of order n, which we call a Cayley-Dickson orthogonal design of order n.*

Proof. Let σ be an endomorphic idempotent solution of the CDOD of order n over the set of variables $\{A_1, \ldots, A_n\}$. From σ, we associate with each A_i a number s_i as follows:

- If $A_i \in dom(\sigma)$ and $A_i \notin ran(\sigma)$, then $s_i = 0$.
- If $A_i \notin dom(\sigma)$ and $A_i \notin ran(\sigma)$, then $s_i = 1$.
- If $A_i \notin dom(\sigma)$ and $A_i \in ran(\sigma)$, then $s_i = m + 1$, where m is the number of variables that map to A_i by sigma.

These s_i's, together with the corresponding A_i, give a matrix A with the property $AA^T = (\sum_{j=1}^{k} s_j A_j^2) I_n$, for $k \leq \rho(n)$, and $\rho(n)$ is the Radon function that gives an upper bound on the

number of variables that can appear in a design. This is by definition an orthogonal design of order n and type $(s_1, \ldots, s_k)$. ◀

Now, it is important to note that the `CDOD` problem is instantiated for orders of power of two, since in these orders we are able to construct the multiplication tables of the respective algebras by using successively the Cayley-Dickson process on the construction of designs via quaternions of Baumert and Hall. We are interested in Cayley-Dickson orthogonal designs in orders 16, 32 and 64.

- `CDOD16`: An instance of the `CDOD` problem for order 16, consists of a polynomial system of 42 equations in 14 variables.
- `CDOD32`: An instance of the `CDOD` problem for order 32, consists of a polynomial system of 252 equations in 30 variables.
- `CDOD64`: An instance of the `CDOD` problem for order 64, consists of a polynomial system of 1182 equations in 62 variables.

We emphasize here the computational difficulty of retrieving *all* endomorphic solutions of the previous three problems. We have used Gröbner bases to verify the computations of [7] and [8], for orders 16 and 32, 64, respectively. In particular, we have computed in Magma V2.12-14 a reduced Gröbner basis (for a total degree reverse lexicographical ordering) for the polynomial systems of the `CDOD16` and `CDOD32` problem. For order 64 we have not managed to compute a Gröbner basis due to its enormous computational cost. Clearly, a solution of the reduced polynomial system obtained by a Gröbner basis corresponds to a solution of the original system. We formulate the `CDOD` problems in terms of Gröbner bases, below.

- `CDODGB16`: A reduced Gröbner basis of the `CDOD` problem for order 16, consists of a polynomial system of 21 equations in 14 variables.
- `CDODGB32`: A reduced Gröbner basis of the `CDOD` problem for order 32, consists of a polynomial system of 290 equations in 30 variables.

Gröbner bases give some insight how to locate endomorphic solutions due to the fact that binomial terms of the polynomial system could be written in a canonical form. However, this is not sufficient to compute all required solutions as there is no indication for the structure of substitution of different variables. Moreover, using this property that distills from Gröbner bases in [7] and [8], it was feasible only to compute a handful of solutions and respectively orthogonal designs.

It is clear that a specialized equation solver is needed to retrieve all endomorphic solutions for the previous five problems. Performing some post-processing on the structure of the polynomial systems we obtained for these problems, we observe that each equation consists of the *same* number of positive and negative monomial terms, and within each equation, all monomials have the same degree. This property, together with some statistics for the structure of the equations, makes the `CDOD` problems and their Gröbner basis counterpart very suitable to be attacked by equational unification as we later explain in Sections 5 and 6.

4 Equational Unification

Unification theory [1] studies *unification problems:* sets of equations between *terms.* The latter, as usual, are constructed by a set of function symbols $\mathcal{F}$ and a (countably infinite) set of variables $\mathcal{V}$. We denote the set of terms over $\mathcal{F}$ and $\mathcal{V}$ by $\mathcal{T}(\mathcal{F}, \mathcal{V})$. Variables are denoted by x, y, z, function symbols by f, g, and terms by s, t, r.

Our substitutions are a special case of the substitutions defined in Section 3.1, mapping variables to terms. An *application* of a substitution σ to a term t, denoted $t\sigma$, is defined as follows: If $t = x$, then $t\sigma := \sigma(x)$. If $t = f(s_1, \ldots, s_n)$, $n \geq 0$, then $t\sigma := f(s_1\sigma, \ldots, s_n\sigma)$. *Composition* of two substitutions σ and φ, written as $\sigma\varphi$, is defined as $t\sigma\varphi := (t\sigma)\varphi$ for any t.

An *equational theory*, defined by a set equational axioms $E \subseteq \mathcal{T}(\mathcal{F}, \mathcal{V}) \times \mathcal{T}(\mathcal{F}, \mathcal{V})$, is the least congruence relation on $\mathcal{T}(\mathcal{F}, \mathcal{V})$, that is closed under substitution application and contains E. It is denoted by $\doteq_E$. If $s \doteq_E t$, then we say that s and t are *equal modulo* E. The axioms (i.e., the elements of E) are written as $s \approx t$. For instance, $E = \{f(x, f(y, z)) \approx f(f(x, y), z),\ f(x, y) \approx f(y, x)\}$ defines the equational theory of associativity and commutativity of f.

Given an E and a set of variables $\mathcal{X}$, the substitution σ is *more general modulo* E *on* $\mathcal{X}$ than the substitution φ, written $\sigma \preccurlyeq_E^{\mathcal{X}} \varphi$, iff there exists a substitution ϑ such that $x\sigma\vartheta \doteq_E x\varphi$ for all $x \in \mathcal{X}$. The relation $\preccurlyeq_E^{\mathcal{X}}$ is a quasi-order, and the induced equivalence is denoted by $\simeq_E^{\mathcal{X}}$.

Given an E and a set of function symbols $\mathcal{F}$, an E-unification problem Γ over $\mathcal{F}$ is a finite set of equations between terms over $\mathcal{F}$ and a countable infinite set of variables $\mathcal{V}$, written as $\Gamma := \{s_1 \doteq_E^? t_1, \ldots, s_n \doteq_E^? t_n\}$. An E-unifier of Γ is a substitution σ such that $s_i\sigma \doteq_E t_i\sigma$ for all $1 \leq i \leq n$.

Let Γ be an E-unification problem over $\mathcal{F}$ and let $\mathcal{X}$ be the set of all variables that occur in Γ. A *minimal complete set of unifiers* (*mcsu,* in short) of Γ, denoted $mcsu(\Gamma)$, is the set of substitutions such that the following three conditions are satisfied:

- Correctness: Each element of $mcsu(\Gamma)$ is an E-unifier of Γ.
- Completeness: For each unifier φ of Γ there exists $\sigma \in mcsu(\Gamma)$ such that $\sigma \preccurlyeq_E^{\mathcal{X}} \varphi$.
- Minimality: For all $\sigma_1, \sigma_2 \in mcsu(\Gamma)$, if $\sigma_1 \preccurlyeq_E^{\mathcal{X}} \sigma_2$, then $\sigma_1 = \sigma_2$.

The *signature* of an equational theory E, denoted by $sig(E)$, is the set of all function symbols that appear in the axioms of E. An E-unification problem Γ over $\mathcal{F}$ is *elementary*, if $\mathcal{F} \setminus sig(E) = \emptyset$. It is a problem with *constants*, if $\mathcal{F} \setminus sig(E)$ is a set of constants. It is called a *general* problem, if $\mathcal{F} \setminus sig(E)$ may contain arbitrary function symbols.

When we are interested in E-unification problems of a special form, we talk about a *fragment* of E-unification. When solutions only of a special form are needed, then we say that a *variant* of E-unification is considered.

5 Orthogonal Designs Meet Equational Unification

In this section, we establish the connections between orthogonal designs and equational unification. In particular, we show that Cayley-Dickson orthogonal designs defined in Section 3.1 can be constructed from unifiers of certain unification problems.

Recall that, as we observed, each equation in a CDOD consists of an *equal* number of positive and negative monomial terms. Moreover, within an equation, all monomials have the *same* degree. That means that the equations have the form $A_{11} \cdots A_{1n} + \cdots + A_{m1} \cdots A_{mn} - B_{11} \cdots B_{1n} - \cdots - B_{m1} \cdots B_{mn} = 0$ with $n, m > 0$. By changing the design variables with unification variables (A with x, B with y), making the multiplication explicit, and placing negative monomials on the other side of equation, we obtain a unification problem of the form $x_{11} * \cdots * x_{1n} + \cdots + x_{m1} * \cdots * x_{mn} \doteq_{\mathsf{AC}(+,*)}^? y_{11} * \cdots * y_{1n} + \cdots + y_{m1} * \cdots * y_{mn}$, where $*$ and $+$ are associative and commutative (and the subscript $\mathsf{AC}(+,*)$ indicates this fact). We refer to the unification problem obtained from an CDOD (of order n) in this way as $\mathtt{CDOD}_{\mathcal{U}}$ (of order n). The important property, that is straightforward to see, is that there is a

direct correspondence between endomorphic solutions of unification equations in the $\mathtt{CDOD}_{\mathcal{U}}$ and those of the corresponding polynomial equations in the given CDOD.

▶ **Theorem 5.** *Let $n = 2^m$ for some $m > 0$. If there exists an endomorphic idempotent unifier for an $\mathtt{CDOD}_{\mathcal{U}}$ problem of order n, then there exists a Cayley-Dickson orthogonal design of order n.*

Proof. $\mathtt{CDOD}_{\mathcal{U}}$ of order n has an idempotent endomorphic unifier iff the corresponding CDOD of order n has an idempotent endomorphic solution. By Theorem 4, the latter implies the existence of an Cayley-Dickson orthogonal design of order n. ◀

For an endomorphic idempotent solution σ of the CDOD, the corresponding Cayley-Dickson orthogonal design is denoted by $\mathtt{CDOD}(\sigma)$.

In the theory of orthogonal designs, as we have already mentioned the Equating-Killing Lemma plays a pivotal role, as it can produce a vast number of orthogonal designs from any given one. It is natural to distinguish between orthogonal designs that can be produced or not by the Equating-Killing Lemma.

Given two ODs of the same order, D_1 and D_2, we say D_1 is more general than D_2 and write $D_1 \trianglerighteq D_2$, if there exists an OD D_3 of the same order as D_1 and D_2 such that $D_1 \simeq D_3$ and D_2 is obtained from D_3 by equating zero or more variables. Strictly more generality relation is written $D_1 \triangleright D_2$ and requires equating one or more variables to get D_3 from D_1.

▶ **Definition 6** (Basis)**.** Let $\mathcal{D}$ be a set of orthogonal designs of order n. A *basis* for $\mathcal{D}$ is a set $\mathcal{B} \subseteq \mathcal{D}$ such that for each $D \in \mathcal{D}$, there is $B \in \mathcal{B}$ such that $B \trianglerighteq D$.

A trivial basis for $\mathcal{D}$ is $\mathcal{D}$ itself. The interesting ones are *reduced bases* defined below:

▶ **Definition 7** (Reduced Basis, Base OD)**.** Let $\mathcal{B}$ be a basis of the set $\mathcal{D}$ of orthogonal designs of the same order. $\mathcal{B}$ is a *reduced basis* of $\mathcal{D}$, written $rb(\mathcal{D})$, if $\mathcal{B}$ does not contain two elements B_1, B_2 such that $B_1 \trianglerighteq B_2$. The elements of $rb(\mathcal{D})$ are called the *base orthogonal designs* for $\mathcal{D}$.

This notion of *base orthogonal designs* introduced here for the first time, exhibits a remarkable connection with unification theory.

▶ **Theorem 8.** *Consider a CDOD problem of order n and the corresponding $\mathtt{CDOD}_{\mathcal{U}}$ unification problem. Let σ be an element of the minimal complete set of endomorphic idempotent unifiers of $\mathtt{CDOD}_{\mathcal{U}}$. Assume $\mathcal{D}$ is a set of Cayley-Dickson orthogonal designs of order n (i.e, the solutions of the CDOD problem). Then $\mathtt{CDOD}(\sigma) \in \mathcal{D}$ is a base orthogonal design for $\mathcal{D}$.*

Proof. Let χ be the set of variables of $\mathtt{CDOD}_{\mathcal{U}}$ and A be the set of variables of CDOD. The theorem follows from the following fact: For two endomorphic idempotent unifiers φ_1 and φ_2 of $\mathtt{CDOD}_{\mathcal{U}}$, if $\varphi_1 \preccurlyeq^{\chi}_{\mathsf{AC}(+,*)} \varphi_2$, then $\mathtt{CDOD}(\varphi_1) \trianglerighteq \mathtt{CDOD}(\varphi_2)$. Since φ_1 and φ_2 are endomorphic, $\varphi_1 \preccurlyeq^{\chi}_{\mathsf{AC}(+,*)} \varphi_2$ means that for some ϑ, $x\varphi_1\vartheta = x\varphi_2$ for all $x \in \chi$. Hence, ϑ is also endomorphic on χ and can be decomposed into $\vartheta_1\vartheta_2$, where ϑ_1 is a permutation (a bijective mapping from $dom(\vartheta)$ to $dom(\vartheta)$), and ϑ_2 is an arbitrary endomorphic substitution. Then from $\mathtt{CDOD}(\varphi_1)$ we first can obtain an OD D by renaming variables that correspond to ϑ_1. It gives $\mathtt{CDOD}(\varphi_1) \simeq D$. Afterwards, from D we can perform variable equating according to ϑ_2, which will give $\mathtt{CDOD}(\varphi_2)$. By the definition of $\trianglerighteq$, we get $\mathtt{CDOD}(\varphi_1) \trianglerighteq \mathtt{CDOD}(\varphi_2)$. ◀

The connections between orthogonal designs and unification theory presented in this section are essential for translating CDOD problems to unification problems, and in addition provide some concrete guidelines on how to efficiently perform a systematic solving of the respective polynomial systems.

6 Solving Unification Problems

Our unification problem Γ contains only equations in the flattened form $x_1^1 * \cdots * x_n^1 + \cdots + x_1^m * \cdots * x_n^m \doteq^?_{\mathsf{AC}(+,*)} y_1^1 * \cdots * y_n^1 + \cdots + y_1^m * \cdots * y_n^m$ for some $n, m > 0$, where $+$ and $*$ are the AC symbols. We call it a *balanced* fragment of AC-unification. We are looking for AC-unifiers of Γ that map variables of Γ to variables of Γ, i.e., both domain and range of unifiers should be subsets of $var(\Gamma)$. We call such variants *endomorphic*. Hence, the problem we would like to solve is an *endomorphic variant of a balanced fragment of the elementary AC-unification*. For brevity, we refer to it as an $\mathrm{AC_{EB}}$-unification problem.

Note that this problem always has a unifier: Just map all variables to one of them, and it will be a solution. What we are looking for is the minimal complete set of unifiers.

AC-unification problems are solved by reducing them to systems of linear Diophantine equations, see, e.g., [3, 14]. However, it is pretty easy to formulate a direct algorithm that computes a complete set of unifiers for $\mathrm{AC_{EB}}$-unification problems. In fact, as we will see, the four rules below are sufficient to construct it. The rules transform systems (pairs $\Gamma; \sigma$ of an unification problem and a substitution) into systems. The symbol $\uplus$ stands for disjoint union. The subscript $\mathsf{AC}(+, *)$ is omitted, as well as the symbol $*$.

T: **Trivial**

$$\{x \doteq^? x\} \uplus \Gamma';\ \sigma \Longrightarrow \Gamma';\ \sigma.$$

D-sum: **Decomposition for Sums**

$$\{s_1 + \cdots + s_n \doteq^? t_1 + \cdots + t_n\} \uplus \Gamma';\ \sigma \Longrightarrow \{s_1 \doteq^? \pi(t_1), \ldots, s_n \doteq^? \pi(t_n)\} \cup \Gamma';\ \sigma,$$

where $n > 1$ and π is a permutation of the multiset $\{t_1, \ldots, t_n\}$.

D-prod: **Decomposition for Products**

$$\{x_1 \cdots x_n \doteq^? y_1 \cdots y_n\} \uplus \Gamma';\ \sigma \Longrightarrow \{x_1 \doteq^? \pi(y_1), \ldots, x_n \doteq^? \pi(y_n)\} \cup \Gamma';\ \sigma,$$

where $n > 1$ and π is a permutation of the multiset $\{t_1, \ldots, t_n\}$.

S: **Solve**

$$\{x \doteq^? y\} \uplus \Gamma';\ \sigma \Longrightarrow \Gamma'\{x \mapsto y\};\ \sigma\{x \mapsto y\}, \qquad \text{where } x \neq y.$$

We call a system $\Gamma; \sigma$ a *balanced system*, if Γ is a balanced AC-unification problem. By inspecting the rules, it is easy to see that the rules transform balanced systems into balanced systems. Note that any balanced system $\Gamma; \sigma$, where $\Gamma \neq \emptyset$, can be transformed, and each selected equation can be transformed by only one rule.

To solve an unification problem Γ, we create the initial system $\Gamma; \varepsilon$ and apply the rules exhaustively. Let **BF** denote this algorithm, to indicate that it is a brute force approach, i.e., **BF** $:= (\mathsf{T} \mid \mathsf{D\text{-}sum} \mid \mathsf{D\text{-}prod} \mid \mathsf{S})^*$, where $\mid$ stands for choice and * for iteration. The terminal systems have the form $\emptyset; \sigma$. We say in this case that the algorithm computes σ. Given a balanced Γ, the set of all substitutions computed by **BF** is denoted by $\Sigma_{\mathbf{BF}}(\Gamma)$. This set is finite, because there can be finitely many terminal systems (since rules that produce all possible permutations lead to finite branching).

▶ **Theorem 9.** *Given a balanced AC-unification problem Γ, the algorithm **BF** terminates and computes $\Sigma_{\mathbf{BF}}(\Gamma)$, which is a complete set of endomorphic idempotent AC-unifiers of Γ.*

Proof. (Sketch) To prove termination, we first define the size of an equation as the number of symbol occurrences in it (including the $*$ that is omitted in the rules). Next, we associate to each AC-unification problem its measure: the multiset of sizes of equations in it. Then we

can see that each rule strictly decreases this measure. For the rules T and S it is obvious. For the other two rules it follows from the condition $n > 1$, which implies that the resulting set of equations reduces the number of occurrences of $+$ or $*$, while the rest does not increase. These facts, together with the observation that the number of branching alternatives the rules produce is finite, imply termination.

The S rule guarantees that the computed substitutions are endomorphic and idempotent. Each rule preserves the set of unifiers for the problems it transforms. Hence the computed substitutions are endomorphic idempotent unifiers. Completeness is implied by the fact that the permutations in the decomposition rules generate all possible branchings in the search tree. ◀

The set $\Sigma_{\mathsf{BF}}(\Gamma)$ is not minimal, in general. This is not surprising, since $\mathrm{AC}_{\mathrm{EB}}$-unification is, in fact, variadic commutative (aka orderless) unification [9]. No algorithm is known that would directly compute minimal complete set of unifiers for commutative unification problems. There is an additional minimization step required.

Our main challenge, however, was related to the size of the problem. The unification problems contain hundreds of equations and the brute-force approach of **BF** usually is not feasible. We need to keep the alternatives as small as possible. For this purpose, we elaborated several heuristics. Two of them concern equation selection, and two more unification problem simplification:

Sel1: For transformation, select an equation with the minimal number of arguments. For instance, if the unification problem is $\{x_1x_2 + y_3x_3 \doteq^? x_3y_3 + y_2y_1, x_1 \doteq^? y_1\}$, the equation $x_1 \doteq^? y_1$ will be selected and transformed by the rule Solve.

Sel2: In the decomposition rules, permute that side of the selected equation that generates fewer permutations (i.e., the side that has more repeated arguments). It reduces the branching factor, but completeness is not violated, since equality is symmetric.

Simp1: Given an ordering on variables that is extended lexicographically to products and sums, rearrange unordered subterms in equations in the ordered form. For instance, if $x_1 > x_2 > x_3 > y_1 > y_2 > y_3$, then the equation $x_1x_2 + y_3x_3 \doteq^? x_3y_3 + y_2y_1$ would be transformed in one step to, e.g., $x_1x_2 + x_3y_3 \doteq^? x_3y_3 + y_2y_1$ and in two steps to $x_1x_2 + x_3y_3 \doteq^? x_3y_3 + y_1y_2$.

Simp2: Remove all common arguments from both sides of equations. This is a well-known technique used in AC-unification. It reduces, for instance, the equation $x_1x_2 + x_3y_3 \doteq^? x_3y_3 + y_1y_2$ in one step to $x_1x_2 \doteq^? y_1y_2$. Two applications of this strategy would reduce the equation $x_1x_2y_1 \doteq^? y_1x_2y_2$ to $x_1 \doteq^? y_2$.

Let T-s and S-s be the variations of the T and S rules, respectively, where equations are selected according to Sel1. Similarly, D-sum-s and D-prod-s stand for the variants of D-sum and D-prod rules, where the equation is selected according to Sel1, and the permutation side is selected according to Sel2. Simp1 should be used in combination with Simp2 to detect common arguments in the sides of equations. Then we define the refined algorithm **Ref** with the following strategy ($\circ$ stands for composition, $\mid$ for choice, * for iteration):

$$\mathbf{Ref} := \big((\mathsf{Simp1} \mid \mathsf{Simp2})^* \circ (\mathsf{T\text{-}s} \mid \mathsf{S\text{-}s} \mid \mathsf{D\text{-}sum\text{-}s} \mid \mathsf{D\text{-}prod\text{-}s})\big)^*.$$

In words, it means that **Ref** works with a set of systems, selects one of them nondeterministically, normalizes it with respect to the Simp1 and Simp2, transforms the obtained system into new ones with one of the rules T-s, S-s, D-sum-s, or D-prod-s, and iterates.

Since unification problems are sets, simplification steps may decrease the number of equations, when several equations simplify to the same one. It is not hard to see that the

selection and simplification heuristics affect neither soundness nor completeness. Therefore, based on Theorem 9 we have that $\Sigma_{\mathsf{Ref}}(\Gamma)$ is a complete set of endomorphic idempotent unifiers of Γ.

As it turns out, Simp2 plays an important role in reducing the number of computed unifiers. For instance, for an unification problem Γ originated from `CDODGB16`, $\Sigma_{\mathsf{Ref}}(\Gamma)$ contains 7 unifiers. For a Γ coming from `CDODGB32`, this number is 33. If we skipped the Simp2 step in **Ref**, then we would get 45 unifiers for `CDODGB16`, and 1574 for `CDODGB32`. Similarly, for an unification problem Γ originated from `CDOD16`, $\Sigma_{\mathsf{Ref}}(\Gamma)$ contains 65 unifiers. For a Γ coming from `CDOD32`, this number is 6935. If we again skip the Simp2 step in **Ref**, then we get 264 unifiers for `CDOD16`.

The set computed by **Ref** is complete but not minimal. A minimal and complete algorithm **ACEB** for $\mathrm{AC_{EB}}$-unification problems can be formulated as

$$\mathbf{ACEB}(\Gamma) := \mathit{minimize}\big(\Sigma_{\mathsf{Ref}}(\Gamma)\big),$$

where *minimize* is a function that minimizes a set of substitutions. Therefore, we have the following theorem:

▶ **Theorem 10.** *ACEB*$(\Gamma) = mcsu(\Gamma)$.

For efficiency reasons, it makes sense to have an incremental version of the algorithm **ACEB**: Instead of working with the entire set of equations at once, we split this set into smaller subsets of some fixed size size. After **ACEB** computes an mcsu $\mathcal{U}$ of one such subset, we generate all possible instances of the next subset with respect to the unifiers in $\mathcal{U}$, and proceed further in a similar way for each new set. Such early minimization efforts reduce the number of redundant potential solutions. This method is sensitive to the choice of size. It should be not too small not to trigger frequent calls of the expensive *minimize* function, and not too big not to postpone minimization too much. As experiments showed, a good strategy for the unification problems originated from the original polynomials is, for instance, to set size close to the number of equations of the smallest size. For instance, in `CDOD32`, the polynomials of the smallest size are those that contain 4 monomials, each of degree 2. There are 42 such polynomials (out of 252) there. Setting size to 42 led to the fastest computation of the result. However, for equations coming from the polynomials in Gröbner bases, we could not observe such a pattern.

Now we give the elements of **ACEB**(Γ) for unification problems Γ that originate from `CDOD16`, `CDODGB16`, `CDOD32`, `CDODGB32` and `CDOD64` problems:

`CDOD16` and `CDODGB16`:

$$\sigma_1^{16} = \{x_{10} \to x_2,\, x_{11} \to x_3,\, x_{12} \to x_4,\, x_{13} \to x_5,\, x_{14} \to x_6,\, x_{15} \to x_7,\, x_{16} \to x_8\}$$

$$\sigma_2^{16} = \{x_2 \to x_8,\, x_3 \to x_8,\, x_4 \to x_8,\, x_5 \to x_8,\, x_6 \to x_8,\, x_7 \to x_8,\, x_{10} \to x_{16},\, x_{11} \to x_{16},\, x_{12} \to x_{16},\, x_{13} \to x_{16},\, x_{14} \to x_{16},\, x_{15} \to x_{16}\}$$

`CDOD32` and `CDODGB32`:

$$\sigma_1^{32} = \{x_2 \to x_8,\, x_3 \to x_8,\, x_4 \to x_8,\, x_5 \to x_8,\, x_6 \to x_8,\, x_7 \to x_8,\, x_{10} \to x_{32},\, x_{11} \to x_{32},\, x_{12} \to x_{32},\, x_{13} \to x_{32},\, x_{14} \to x_{32},\, x_{15} \to x_{32},\, x_{16} \to x_{32},\, x_{18} \to x_8,\, x_{19} \to x_8,\, x_{20} \to x_8,\, x_{21} \to x_8,\, x_{22} \to x_8,\, x_{23} \to x_8,\, x_{24} \to x_8,\, x_{25} \to x_9,\, x_{26} \to x_{32},\, x_{27} \to x_{32},\, x_{28} \to x_{32},\, x_{29} \to x_{32},\, x_{30} \to x_{32},\, x_{31} \to x_{32}\}$$

$$\sigma_2^{32} = \{x_2 \to x_{26},\, x_{10} \to x_{26},\, x_{11} \to x_3,\, x_{12} \to x_4,\, x_{13} \to x_5,\, x_{14} \to x_6,\, x_{15} \to x_7,$$

$$x_{16} \to x_8,\ x_{18} \to x_{26},\ x_{19} \to x_3,\ x_{20} \to x_4,\ x_{21} \to x_5,\ x_{22} \to x_6,\ x_{23} \to x_7,$$
$$x_{24} \to x_8,\ x_{25} \to x_9,\ x_{27} \to x_3,\ x_{28} \to x_4,\ x_{29} \to x_5,\ x_{30} \to x_6,\ x_{31} \to x_7,$$
$$x_{32} \to x_8\}$$

$$\sigma_3^{32} = \{x_2 \to x_9,\ x_3 \to x_9,\ x_4 \to x_9,\ x_5 \to x_9,\ x_6 \to x_9,\ x_7 \to x_9,\ x_8 \to x_9,$$
$$x_{10} \to x_9,\ x_{11} \to x_9,\ x_{12} \to x_9,\ x_{13} \to x_9,\ x_{14} \to x_9,\ x_{15} \to x_9,\ x_{16} \to x_9,$$
$$x_{18} \to x_{32},\ x_{19} \to x_{32},\ x_{20} \to x_{32},\ x_{21} \to x_{32},\ x_{22} \to x_{32},\ x_{23} \to x_{32},$$
$$x_{24} \to x_{32},\ x_{25} \to x_{32},\ x_{26} \to x_{32},\ x_{27} \to x_{32},\ x_{28} \to x_{32},\ x_{29} \to x_{32},$$
$$x_{30} \to x_{32},\ x_{31} \to x_{32}\}$$

CDOD64:

$$\sigma_1^{64} = \{x_2 \to x_9,\ x_3 \to x_9,\ x_4 \to x_9,\ x_5 \to x_9,\ x_6 \to x_9,\ x_7 \to x_9,\ x_8 \to x_9,$$
$$x_{10} \to x_9,\ x_{11} \to x_9,\ x_{12} \to x_9,\ x_{13} \to x_9,\ x_{14} \to x_9,\ x_{15} \to x_9,\ x_{16} \to x_9,$$
$$x_{17} \to x_9,\ x_{18} \to x_9,\ x_{19} \to x_9,\ x_{20} \to x_9,\ x_{21} \to x_9,\ x_{22} \to x_9,\ x_{23} \to x_9,$$
$$x_{24} \to x_9,\ x_{25} \to x_9,\ x_{26} \to x_9,\ x_{27} \to x_9,\ x_{28} \to x_9,\ x_{29} \to x_9,\ x_{30} \to x_9,$$
$$x_{31} \to x_9,\ x_{32} \to x_9,\ x_{34} \to x_{64},\ x_{35} \to x_{64},\ x_{36} \to x_{64},\ x_{37} \to x_{64},$$
$$x_{38} \to x_{64},\ x_{39} \to x_{64},\ x_{40} \to x_{64},\ x_{41} \to x_{64},\ x_{42} \to x_{64},\ x_{43} \to x_{64},$$
$$x_{44} \to x_{64},\ x_{45} \to x_{64},\ x_{46} \to x_{64},\ x_{47} \to x_{64},\ x_{48} \to x_{64},\ x_{49} \to x_{64},$$
$$x_{50} \to x_{64},\ x_{51} \to x_{64},\ x_{52} \to x_{64},\ x_{53} \to x_{64},\ x_{54} \to x_{64},\ x_{55} \to x_{64},$$
$$x_{56} \to x_{64},\ x_{57} \to x_{64},\ x_{58} \to x_{64},\ x_{59} \to x_{64},\ x_{60} \to x_{64},\ x_{61} \to x_{64},$$
$$x_{62} \to x_{64},\ x_{63} \to x_{64}\}$$

$$\sigma_2^{64} = \{x_2 \to x_8,\ x_3 \to x_8,\ x_4 \to x_8,\ x_5 \to x_8,\ x_6 \to x_8,\ x_7 \to x_8,\ x_{10} \to x_{64},$$
$$x_{11} \to x_{64},\ x_{12} \to x_{64},\ x_{13} \to x_{64},\ x_{14} \to x_{64},\ x_{15} \to x_{64},\ x_{16} \to x_{64},$$
$$x_{17} \to x_{49},\ x_{18} \to x_8,\ x_{19} \to x_8,\ x_{20} \to x_8,\ x_{21} \to x_8,\ x_{22} \to x_8,\ x_{23} \to x_8,$$
$$x_{24} \to x_8,\ x_{25} \to x_9,\ x_{26} \to x_{64},\ x_{27} \to x_{64},\ x_{28} \to x_{64},\ x_{29} \to x_{64},$$
$$x_{30} \to x_{64},\ x_{31} \to x_{64},\ x_{32} \to x_{64},\ x_{34} \to x_8, x_{35} \to x_8,\ x_{36} \to x_8,$$
$$x_{37} \to x_8,\ x_{38} \to x_8,\ x_{39} \to x_8,\ x_{40} \to x_8,\ x_{41} \to x_9,\ x_{42} \to x_{64},\ x_{43} \to x_{64},$$
$$x_{44} \to x_{64},\ x_{45} \to x_{64},\ x_{46} \to x_{64},\ x_{47} \to x_{64},\ x_{48} \to x_{64},\ x_{50} \to x_8,$$
$$x_{51} \to x_8,\ x_{52} \to x_8,\ x_{53} \to x_8,\ x_{54} \to x_8,\ x_{55} \to x_8,\ x_{56} \to x_8,\ x_{57} \to x_9,$$
$$x_{58} \to x_{64},\ x_{59} \to x_{64},\ x_{60} \to x_{64},\ x_{61} \to x_{64}, x_{62} \to x_{64},\ x_{63} \to x_{64}\}$$

$$\sigma_3^{64} = \{x_2 \to x_{58},\ x_3 \to x_{59},\ x_4 \to x_{60},\ x_5 \to x_{61},\ x_6 \to x_{62},\ x_{10} \to x_{58},\ x_{11} \to x_{59},$$
$$x_{12} \to x_{60},\ x_{13} \to x_{61},\ x_{14} \to x_{62},\ x_{15} \to x_7,\ x_{16} \to x_8,\ x_{17} \to x_{49},\ x_{18} \to x_{58},$$
$$x_{19} \to x_{59},\ x_{20} \to x_{60},\ x_{21} \to x_{61},\ x_{22} \to x_{62},\ x_{23} \to x_7,\ x_{24} \to x_8,\ x_{25} \to x_9,$$
$$x_{26} \to x_{58},\ x_{27} \to x_{59},\ x_{28} \to x_{60},\ x_{29} \to x_{61},\ x_{30} \to x_{62},\ x_{31} \to x_7,\ x_{32} \to x_8,$$
$$x_{34} \to x_{58},\ x_{35} \to x_{59},\ x_{36} \to x_{60},\ x_{37} \to x_{61},\ x_{38} \to x_{62},\ x_{39} \to x_7,\ x_{40} \to x_8,$$
$$x_{41} \to x_9,\ x_{42} \to x_{58},\ x_{43} \to x_{59},\ x_{44} \to x_{60},\ x_{45} \to x_{61},\ x_{46} \to x_{62},\ x_{47} \to x_7,$$
$$x_{48} \to x_8,\ x_{50} \to x_{58},\ x_{51} \to x_{59},\ x_{52} \to x_{60},\ x_{53} \to x_{61},\ x_{54} \to x_{62},\ x_{55} \to x_7,$$
$$x_{56} \to x_8,\ x_{57} \to x_9,\ x_{63} \to x_7,\ x_{64} \to x_8\}$$

$$\sigma_4^{64} = \{x_2 \to x_9,\ x_3 \to x_9,\ x_4 \to x_9,\ x_5 \to x_9,\ x_6 \to x_9,\ x_7 \to x_9,\ x_8 \to x_9,\ x_{10} \to x_9,$$
$$x_{11} \to x_9,\ x_{12} \to x_9,\ x_{13} \to x_9, x_{14} \to x_9,\ x_{15} \to x_9,\ x_{16} \to x_9,\ x_{17} \to x_{49},$$
$$x_{18} \to x_{64},\ x_{19} \to x_{64},\ x_{20} \to x_{64},\ x_{21} \to x_{64},\ x_{22} \to x_{64},\ x_{23} \to x_{64},$$
$$x_{24} \to x_{64},\ x_{25} \to x_{64},\ x_{26} \to x_{64},\ x_{27} \to x_{64},\ x_{28} \to x_{64},\ x_{29} \to x_{64},$$
$$x_{30} \to x_{64},\ x_{31} \to x_{64},\ x_{32} \to x_{64},\ x_{34} \to x_9,\ x_{35} \to x_9,\ x_{36} \to x_9,\ x_{37} \to x_9,$$
$$x_{38} \to x_9,\ x_{39} \to x_9,\ x_{40} \to x_9,\ x_{41} \to x_9,\ x_{42} \to x_9,\ x_{43} \to x_9,\ x_{44} \to x_9,$$
$$x_{45} \to x_9,\ x_{46} \to x_9,\ x_{47} \to x_9,\ x_{48} \to x_9,\ x_{50} \to x_{64},\ x_{51} \to x_{64},\ x_{52} \to x_{64},$$

$x_{53} \to x_{64},\ x_{54} \to x_{64},\ x_{55} \to x_{64},\ x_{56} \to x_{64},\ x_{57} \to x_{64},\ x_{58} \to x_{64},$
$x_{59} \to x_{64},\ x_{60} \to x_{64},\ x_{61} \to x_{64},\ x_{62} \to x_{64},\ x_{63} \to x_{64}\}$

We remark that as expected the elements of the **ACEB**(Γ) for the unification problems Γ that originate from `CDOD16` and `CDODGB16`, are the same. This also applies for `CDOD32` and `CDODGB32`.

7 New Cayley-Dickson Orthogonal Designs via Equational Unification

In this section, we translate back from unifiers to solutions of the polynomial systems that give rise to Cayley-Dickson orthogonal designs and list their types. As noted before the elements of **ACEB**(Γ) correspond to base orthogonal designs from Corollary 8, which implies that the designs we list below are sufficient to give all Cayley-Dickson orthogonal designs for orders 16, 32 and 64. Therefore, we provide a complete solution to the `CDOD` problem for these orders.

1. For order 16 we obtain the following *two* base Cayley-Dickson orthogonal designs:
   - From σ_1^{16}: $OD(16; 1, 1, 2, 2, 2, 2, 2, 2, 2)$.
   - From σ_2^{16}: $OD(16; 1, 1, 7, 7)$.
2. For order 32 we obtain the following *three* base Cayley-Dickson orthogonal designs:
   - From σ_1^{32}: $OD(32; 1, 1, 2, 14, 14)$.
   - From σ_2^{32}: $OD(32; 1, 1, 2, 4, 4, 4, 4, 4, 4, 4)$.
   - From σ_3^{32}: $OD(32; 1, 1, 15, 15)$.
3. For order 64 we obtain the following *four* base Cayley-Dickson orthogonal designs:
   - From σ_1^{64}: $OD(64; 1, 1, 31, 31)$.
   - From σ_2^{64}: $OD(64; 1, 1, 2, 4, 28, 28)$.
   - From σ_3^{64}: $OD(64; 1, 1, 2, 4, 8, 8, 8, 8, 8, 8, 8)$.
   - From σ_4^{64}: $OD(64; 1, 1, 2, 30, 30)$.

It is important to note here that from the previous list of orthogonal designs, some Cayley-Dickson orthogonal designs appear here for the *first time*. In particular, the $OD(32; 1, 1, 2, 14, 14)$ and $OD(64; 1, 1, 2, 30, 30)$, $OD(64; 1, 1, 2, 4, 28, 28)$ have not been reported in [7] and [8], respectively.

However, these types of orthogonal designs are not new in the literature of orthogonal designs, as they can be obtained by other methods. In particular, the existence of $OD(32; 1, 1, 2, 14, 14)$ is attributed to a result of Robinson (p. 358, Corollary D.2., [6]) which states that all orthogonal designs of type $(1, 1, a, b, c)$, $a+b+c = 2^t - 2$ exist in order $2^t, t \geq 3$, for $a = 2$, $b = 14$, $c = 14$ and $t = 5$. Again from Robinson's result the $OD(64; 1, 1, 2, 30, 30)$ is known for $a = 2$, $b = 30$, $c = 30$ and $t = 6$. Finally by applying the Doubling Lemma (c.f. Lemma 2) to $OD(32; 1, 1, 2, 14, 14)$ we can get $OD(64; 1, 1, 2, 4, 28, 28)$.

From the previous discussion three patterns for the orthogonal designs that are modelled by Cayley-Dickson algebras and obtained via equational unification are visible.

- The four variable designs are of the form $OD(2^n; 1, 1, 2^{n-1} - 1, 2^{n-1} - 1)$, for orders 2^n where $n = 4, 5, 6$. These types of orthogonal designs can also be obtained via simple Paley matrices [6].
- The five variable designs are of the form $OD(2^n; 1, 1, a, b, c)$ where $a = 2$, $b = 2^n - 2$, $c = 2^n - 2$ for $n = 5, 6$. As we already noted these types of orthogonal designs can be obtained from Robinson's results.

- It is clear that there is an analogy between the Cayley-Dickson process and the Doubling Lemma. In particular, by applying the doubling lemma to the nine variable base orthogonal design in order 16 we obtain the ten variable base orthogonal design in order 32. Repeating the process to the latter design, we obtain the eleven variable base orthogonal design in order 64. In addition, as we have shown earlier the six variable design in order 64 can also obtained by doubling of the five variable design in order 32.

Moreover, we would like to explicitly state that these designs are new with respect to the algebraic modelling of Cayley-Dickson algebras (in the class of Cayley-Dickson orthogonal designs), however the corresponding types of ODs have been reported in the literature also with other techniques. To make our contribution in this section more precise, we can say the following:

1. It was not known before that most of the ODs we found belong also to the class of Cayley-Dickson orthogonal designs.
2. Our approach not only reports the ODs, but also constructs the corresponding design matrices. In this way, we *always* give a *constructive* solution to the problem. It is not always the case with the other approaches. In some cases, there are semi-constructive techniques (doubling method), but in some other, there is only the existential, non-constructive method (Robinson's Lemma). (The doubling method is semi-constructive in the sense that one needs to know the design matrix of the initial OD in order to build design matrices of the ODs the doubling method gives.)
3. The design matrices are of interest for the applications of ODs, since in that case it is not enough to know that the design type exists. For example, in weighing experiments you need the design matrix to perform the actual experiment.
4. The fact that the class of Cayley-Dickson orthogonal designs contains the previous types of orthogonal designs is of interest also to the asymptotic existence of orthogonal designs [6] and will be studied further in future work.

8 Conclusion

In this paper, we presented a algebraic framework for modelling orthogonal designs in order of powers of two via Cayley-Dickson algebras of same orders. This framework gives rise to a polynomial system of equations that is unfeasible to be tackled with traditional search algorithms, as the order increases. We exhibited that the structural properties of this algebraic framework can be written in terms of unification theory by establishing important connections between orthogonal designs and unifiers. These connections enabled the development of unification algorithms that can solve the problems arising from the algebraic modelling of orthogonal designs and find solutions that were not known before with this algebraic modelling of Cayley-Dickson algebras.

Acknowledgements. The first author is supported by an NSERC Discovery grant. The second author has been supported by the Austrian Science Fund (FWF) under the project SToUT (P 24087-N18). The third author has been funded in part by the Austrian COMET Program from the Austrian Research Promotion Agency (FFG).

References

1 F. Baader and W. Snyder. Unification theory. In J. A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning (in 2 volumes)*, pages 445–532. Elsevier and MIT Press, 2001.

2 L. D. Baumert and J. M. Hall. Hadamard matrices of the Williamson type. *Math. Comput.*, 19:442–447, 1965.

3 A. Boudet, E. Contejean, and H. Devie. A new AC unification algorithm with an algorithm for solving systems of Diophantine equations. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science (LICS '90), Philadelphia, Pennsylvania, USA, June 4-7, 1990*, pages 289–299. IEEE Computer Society, 1990.

4 G. M. Dixon. *Division Algebras: Octonions, Quaternions, Complex Numbers and the Algebraic Design of Physics*, volume 290 of *Mathematics and its Applications.* Kluwer Academic Publishers Group, Dordrecht, 1994.

5 A. V. Geramita, J. M. Geramita, and J. S. Wallis. Orthogonal designs. *Linear and Multilinear Algebra*, 3:281–306, 1976.

6 A. V. Geramita and J. Seberry. *Orthogonal Designs. Quadratic Forms and Hadamard Matrices*, volume 45 of *Lecture Notes in Pure and Applied Mathematics.* Marcel Dekker, Inc., New York, NY, 1979.

7 I. S. Kotsireas and C. Koukouvinos. Orthogonal designs via computational algebra. *J. Combin. Designs*, 14:351–362, 2006.

8 I. S. Kotsireas and C. Koukouvinos. Orthogonal designs of order 32 and 64 via computational algebra. *Australasian Journal of Combinatorics*, 39:39–48, 2007.

9 T. Kutsia. *Solving and Proving in Equational Theories with Sequence Variables and Flexible Arity Symbols.* PhD thesis, Johannes Kepler University, Linz, Austria, 2002.

10 F. MacWilliams and N. Sloane. *The Theory of Error-Correcting Codes.* North-Holland, The Netherlands, Amsterdam, 1997.

11 R. L. Plackett and J. Burman. The design of optimum multifactorial experiments. *Biometrika*, 33:305–325, 1946.

12 E. Posner. Combinatorial structures in planetary reconnaissance. In H. Mann, editor, *Proceedings of the 1968 Symposium on Error Correcting Codes*, pages 15–46. Wiley, New York, 1968.

13 J. Seberry and R. Craigen. Orthogonal designs. In C. Colbourn and J. Dinitz, editors, *The CRC Handbook of Combinatorial Designs*, pages 400–406. CRC Press, Boca Raton, Fla., 1996.

14 M. E. Stickel. A unification algorithm for associative-commutative functions. *J. ACM*, 28(3):423–434, 1981.

15 V. Taroch, H. Jafarkhani, and A. R. Calderbank. Space-time block codes from orthogonal designs. *IEEE Trans. Inf. Theory*, 45:1456–1467, 1999.

16 J. Van Lint. Coding, decoding and combinatorics. In R. Wilson, editor, *Applications of Combinatorics*. Shiva, Cheshire, 1982.

17 R. Yarlagadda and J. Hershey. *Hadamard Matrix Analysis and Synthesis: With Applications to Communications and Signal/Image Processing.* Kluwer Acad. Pub., Boston, 1997.

18 Y. Zhao, Y. Wang, and J. Seberry. On amicable orthogonal designs of order 8. *Australasian Journal of Combinatorics*, 34:321–329, 2006.

Improving Automatic Confluence Analysis of Rewrite Systems by Redundant Rules*

Julian Nagele, Bertram Felgenhauer, and Aart Middeldorp

Institute of Computer Science, University of Innsbruck, Austria
{julian.nagele|bertram.felgenhauer|aart.middeldorp}@uibk.ac.at

Abstract

We describe how to utilize redundant rewrite rules, i.e., rules that can be simulated by other rules, when (dis)proving confluence of term rewrite systems. We demonstrate how automatic confluence provers benefit from the addition as well as the removal of redundant rules. Due to their simplicity, our transformations were easy to formalize in a proof assistant and are thus amenable to certification. Experimental results show the surprising gain in power.

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

Keywords and phrases term rewriting, confluence, automation, transformation

Digital Object Identifier 10.4230/LIPIcs.RTA.2015.257

1 Introduction

Confluence of first-order term rewrite systems (TRSs) is an important property which is intimately connected to uniqueness of normal forms, and hence to determinism of programs. In recent years there has been tremendous progress in establishing confluence or non-confluence of TRSs automatically, with a number of tools being developed, like ACP [3], Saigawa [11,14], CoLL[1] and our own tool, CSI [29]. There is an annual confluence competition[2] where these tools compete. To increase the trust in the proofs produced by these tools, a certifier like CeTA [27] can be used to verify the proofs. (CeTA is a certifier for termination and confluence proofs for TRSs. Other certifiers already exist for termination proofs, notably Rainbow [5] and CiME3 [6].) The approach taken by CeTA is to formalize various termination and confluence criteria in an interactive theorem prover, together with executable functions that can be used to verify that the criteria are applied correctly. From this formalization, the certifier is automatically extracted, which produces highly trustworthy code. An alternative approach is to convert certificates produced by automated termination (or confluence) tools into proofs that can be replayed in a theorem prover, thereby formally proving the property for the original TRS.

In this paper we present a remarkably simple technique based on the removal and addition of redundant rules, which can significantly enhance the power of automatic confluence provers. The technique is also straightforward to formalize, making it amenable to certification.

▶ **Example 1.** Consider the TRS $\mathcal{R}$ consisting of the two rewrite rules

$$\mathsf{f}(\mathsf{f}(x)) \to x \qquad\qquad \mathsf{f}(x) \to \mathsf{f}(\mathsf{f}(x))$$

* This work is supported by FWF (Austrian Science Fund) project P27528.
[1] http://www.jaist.ac.jp/~s1310032/coll/
[2] http://coco.nue.riec.tohoku.ac.jp/

© Julian Nagele, Bertram Felgenhauer, and Aart Middeldorp;
licensed under Creative Commons License CC-BY
26th International Conference on Rewriting Techniques and Applications (RTA'15).
Editor: Maribel Fernández; pp. 257–268
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The two non-trivial critical pairs

$$\mathsf{f}(\mathsf{f}(\mathsf{f}(x))) \leftarrow\!\rtimes\!\rightarrow x \qquad\qquad x \leftarrow\!\rtimes\!\rightarrow \mathsf{f}(\mathsf{f}(\mathsf{f}(x)))$$

are obviously joinable

$$\mathsf{f}(\mathsf{f}(\mathsf{f}(x))) \rightarrow_{\mathcal{R}} \mathsf{f}(x) \rightarrow_{\mathcal{R}} \mathsf{f}(\mathsf{f}(x)) \rightarrow_{\mathcal{R}} x$$

but not by a multistep (cf. Definition 3). Consequently, the result of van Oostrom [19] on development-closed critical pairs does not apply. After adding the rewrite rule $\mathsf{f}(x) \rightarrow x$ to $\mathcal{R}$, we obtain four new critical pairs

$$\mathsf{f}(x) \leftarrow\!\rtimes\!\rightarrow x \qquad x \leftarrow\!\rtimes\!\rightarrow \mathsf{f}(x) \qquad \mathsf{f}(\mathsf{f}(x)) \leftarrow\!\rtimes\!\rightarrow x \qquad x \leftarrow\!\rtimes\!\rightarrow \mathsf{f}(\mathsf{f}(x))$$

The new rule ensures that $\mathsf{f}^n(x) \twoheadrightarrow\!\!\!\!\!\!\circ\; x$ for all $n \geqslant 0$ and thus confluence of the extension follows from the main result of [19], cf. Example 4 in Section 2. Since the new rule can be simulated by the original rules (i.e., $\mathsf{f}(x) \rightarrow_{\mathcal{R}} \mathsf{f}(\mathsf{f}(x)) \rightarrow_{\mathcal{R}} x$), also $\mathcal{R}$ is confluent.

None of the aforementioned tools can prove confluence of the TRS $\mathcal{R}$ of Example 1, but every tool can prove confluence of the extended TRS. Below we explain how such extensions can be found automatically.

The next example shows that also proving non-confluence may become easier after adding rules.

▶ **Example 2.** Consider the TRS $\mathcal{R}$ consisting of the eight rewrite rules

$$\begin{array}{llll} \mathsf{f}(\mathsf{g}(\mathsf{a}),\mathsf{g}(y)) \rightarrow \mathsf{b} & \mathsf{f}(x,y) \rightarrow \mathsf{f}(x,\mathsf{g}(y)) & \mathsf{g}(x) \rightarrow x & \mathsf{a} \rightarrow \mathsf{g}(\mathsf{a}) \\ \mathsf{f}(\mathsf{h}(x),\mathsf{h}(\mathsf{a})) \rightarrow \mathsf{c} & \mathsf{f}(x,y) \rightarrow \mathsf{f}(\mathsf{h}(x),y) & \mathsf{h}(x) \rightarrow x & \mathsf{a} \rightarrow \mathsf{h}(\mathsf{a}) \end{array}$$

All critical pairs are *deeply*[3] joinable but $\mathcal{R}$ is not confluent [7]. Two of the critical pairs are

$$\mathsf{b} \leftarrow\!\rtimes\!\rightarrow \mathsf{f}(\mathsf{h}(\mathsf{g}(\mathsf{a})),\mathsf{g}(x)) \qquad\qquad \mathsf{c} \leftarrow\!\rtimes\!\rightarrow \mathsf{f}(\mathsf{h}(x),\mathsf{g}(\mathsf{h}(\mathsf{a})))$$

After adding them as rules

$$\mathsf{f}(\mathsf{h}(\mathsf{g}(\mathsf{a})),\mathsf{g}(x)) \rightarrow \mathsf{b} \qquad\qquad \mathsf{f}(\mathsf{h}(x),\mathsf{g}(\mathsf{h}(\mathsf{a}))) \rightarrow \mathsf{c}$$

new critical pairs are obtained, one of which is

$$\mathsf{b} \leftarrow\!\rtimes\!\rightarrow \mathsf{c}$$

Since b and c are different normal forms, the extension is obviously non-confluent. Since the new rules can be simulated by the original rules, also $\mathcal{R}$ is non-confluent. Of the tools mentioned, ACP shows confluence by first deriving the rule $\mathsf{g}(\mathsf{a}) \rightarrow \mathsf{a}$ (which can be simulated by existing rules), which then gives rise to a critical pair that extends to a non-joinable peak:

$$\mathsf{b} \leftarrow \mathsf{f}(\mathsf{g}(\mathsf{a}),\mathsf{g}(\mathsf{a})) \rightarrow \mathsf{f}(\mathsf{g}(\mathsf{a}),\mathsf{a}) \rightarrow^* \mathsf{c}$$

Saigawa also shows non-confluence but exceeded the 60s time limit in our experiments; it considers critical pairs of the (extended) TRS $\mathcal{R} \cup \mathcal{R}^{-1}$, which includes the rule $\mathsf{g}(\mathsf{a}) \rightarrow \mathsf{a}$. Hence Saigawa finds the same non-joinable peak as ACP. However, CoLL and CSI (without the techniques from this paper) fail.

The remainder of the paper is structured as follows. In Section 3, we describe the theory underlying the addition and removal of rules, and Section 4 is devoted to its integration into CeTA. In Section 5 we briefly sketch our implementation in CSI and present experimental results. Related work is presented in Section 6 before we conclude in Section 7.

[3] A critical pair $s \leftarrow\!\rtimes\!\rightarrow t$ is deeply joinable if $u \downarrow v$ for any two reducts u of s and v of t. The example defeats any non-confluence check based on proving non-joinability of peaks starting from critical peaks.

2 Preliminaries

Throughout the paper we assume familiarity with term rewriting; for an introduction to this topic see [4, 25]. We use s and t to denote terms. Given a position p in a term t, $t|_p$ is the subterm at position p of t, and $t[s]_p$ is the result of replacing the subterm $t|_p$ by the term s in t. The letter σ represents a substitution (mapping variables to terms) and $t\sigma$ is the result of applying σ to the term t. For a binary relation R on terms we write $\sigma\ R\ \sigma'$ if $\sigma(x)\ R\ \sigma'(x)$ for all variables x. Term rewrite systems $\mathcal{R}$, $\mathcal{S}$ consist of rewrite rules $\ell \to r$, and induce rewrite relations (e.g., $\to_{\mathcal{R}}$). The relations $\leftarrow$, $\leftrightarrow$, $\to^*$ denote the inverse, the symmetric closure, and the reflexive, transitive closure of $\to$, respectively. Joinability $\downarrow$ and meetability $\uparrow$ are defined by $\downarrow = \to^* \cdot {}^*\!\leftarrow$ and $\uparrow = {}^*\!\leftarrow \cdot \to^*$. Consider two rules $\ell \to r$ and $\ell' \to r' \in \mathcal{R}$ that have been renamed such that ℓ and ℓ' have no variables in common, and a non-variable position p of ℓ. If p is the root position, then we demand that $\ell \to r$ and $\ell' \to r'$ are not variants of each other. If ℓ' and $\ell|_p$ unify with most general unifier σ, then we obtain a critical peak $\ell[r']_p\sigma \leftarrow \ell\sigma \to r\sigma$. We write $\ell[r']_p\sigma \leftarrow\!\rtimes\!\to r\sigma$ for the corresponding critical pair. We also write $s \leftarrow\!\bowtie\!\to t$ to denote overlays, i.e., critical pairs that stem from overlaps at the root position, and $s \leftarrow\!\cdot\!\!\rtimes\!\to t$ for the other critical pairs.

▶ **Definition 3.** For a TRS $\mathcal{R}$, *multisteps* $\mathrel{\circ\!\!\!\!\!\rightarrow}_{\mathcal{R}}$ are defined inductively by

- $x \mathrel{\circ\!\!\!\!\!\rightarrow}_{\mathcal{R}} x$ if x is a variable,
- $\ell\sigma \mathrel{\circ\!\!\!\!\!\rightarrow}_{\mathcal{R}} r\sigma'$ if $\ell \to r \in \mathcal{R}$ and σ, σ' are substitutions with $\sigma \mathrel{\circ\!\!\!\!\!\rightarrow}_{\mathcal{R}} \sigma'$, and
- $f(s_1, \ldots, s_n) \mathrel{\circ\!\!\!\!\!\rightarrow}_{\mathcal{R}} f(t_1, \ldots, t_n)$ if f is a function symbol of arity n and $s_i \mathrel{\circ\!\!\!\!\!\rightarrow}_{\mathcal{R}} t_i$ for $1 \leqslant i \leqslant n$.

The TRS $\mathcal{R}$ is *development-closed* if every critical pair $s \leftarrow\!\rtimes\!\to t$ satisfies $s \mathrel{\circ\!\!\!\!\!\rightarrow} t$. It is *almost development-closed* if $s \mathrel{\circ\!\!\!\!\!\rightarrow} \cdot {}^*\!\leftarrow t$ for all overlays $s \leftarrow\!\bowtie\!\to t$ and $s \mathrel{\circ\!\!\!\!\!\rightarrow} t$ for all other critical pairs $s \leftarrow\!\cdot\!\!\rtimes\!\to t$.

Van Oostrom [19] has shown that (almost) development closed TRSs are confluent, extending results by Huet [13] and Toyama [28].

▶ **Example 4.** We revisit Example 1 and show confluence of $\mathcal{R} \cup \{\mathsf{f}(x) \to x\}$. First we establish that $\mathsf{f}^n(x) \mathrel{\circ\!\!\!\!\!\rightarrow} x$ by induction on n. The claim is trivially true for $n = 0$. Given $\mathsf{f}^{n-1}(x) \mathrel{\circ\!\!\!\!\!\rightarrow} x$, we can take substitutions σ and σ' that map x to $\mathsf{f}^{n-1}(x)$ and x, respectively, and obtain $\mathsf{f}(x)\sigma \mathrel{\circ\!\!\!\!\!\rightarrow} x\sigma'$, i.e., $\mathsf{f}^n(x) \mathrel{\circ\!\!\!\!\!\rightarrow} x$. For each of the critical pairs $s \leftarrow\!\rtimes\!\to t$, we have either $s \mathrel{\circ\!\!\!\!\!\rightarrow} t$ or $s \leftarrow\!\bowtie\!\to t$ and $s \mathrel{\leftarrow\!\!\!\!\!\circ} t$ (which implies $s \mathrel{\circ\!\!\!\!\!\rightarrow} \cdot {}^*\!\leftarrow t$). Therefore the TRS is almost development closed and thus confluent.

3 Theory

In this section we present the easy theory behind the use of redundant rules for proving confluence. For adding such rules we use the following folklore result.

▶ **Lemma 5.** *If $\ell \to_{\mathcal{R}}^* r$ for every rule $\ell \to r$ from $\mathcal{S}$ then $\to_{\mathcal{R}}^* = \to_{\mathcal{R}\cup\mathcal{S}}^*$.*

Proof. The inclusion $\to_{\mathcal{R}}^* \subseteq \to_{\mathcal{R}\cup\mathcal{S}}^*$ is obvious. For the reverse direction it suffices to show that $s \to_{\mathcal{R}}^* t$ whenever $s \to_{\mathcal{S}} t$. The latter ensures the existence of a position p in s, a rewrite rule $\ell \to r$ in $\mathcal{S}$, and a substitution σ such that $s|_p = \ell\sigma$ and $t = s[r\sigma]_p$. We obtain $\ell \to_{\mathcal{R}}^* r$ from the assumption of the lemma. Closure (of $\to_{\mathcal{R}}^*$) under contexts and substitutions yields the desired $s \to_{\mathcal{R}}^* t$. ◀

▶ **Corollary 6.** *If $\ell \to_{\mathcal{R}}^* r$ for every rule $\ell \to r$ from $\mathcal{S}$ then $\mathcal{R}$ is confluent if and only if $\mathcal{R} \cup \mathcal{S}$ is confluent.*

Proof. We obtain $\to_{\mathcal{R}}^* = \to_{\mathcal{R}\cup\mathcal{S}}^*$ from the preceding lemma. Hence also $\downarrow_{\mathcal{R}} = \downarrow_{\mathcal{R}\cup\mathcal{S}}$ and $\uparrow_{\mathcal{R}} = \uparrow_{\mathcal{R}\cup\mathcal{S}}$. Therefore

$$\uparrow_{\mathcal{R}} \subseteq \downarrow_{\mathcal{R}} \quad\iff\quad \uparrow_{\mathcal{R}\cup\mathcal{S}} \subseteq \downarrow_{\mathcal{R}\cup\mathcal{S}}$$ ◀

▶ **Definition 7.** A rule $\ell \to r \in \mathcal{R}$ is *redundant* if $\ell \to_{\mathcal{R}\setminus\{\ell\to r\}}^* r$.

By Corollary 6, if $\ell \to r \in \mathcal{R}$ is redundant, then $\mathcal{R}$ is confluent if and only if $\mathcal{R} \setminus \{\ell \to r\}$ is confluent. In other words, removing a redundant rule does not affect confluence of a TRS. For removing rules while reflecting[4] confluence (or adding rules while reflecting non-confluence) it suffices that the left- and right-hand side are convertible with respect to the remaining rules.

▶ **Lemma 8.** *If $\ell \leftrightarrow_{\mathcal{R}}^* r$ for every rule $\ell \to r$ from $\mathcal{S}$ then $\leftrightarrow_{\mathcal{R}\cup\mathcal{S}}^* = \leftrightarrow_{\mathcal{R}}^*$.*

Proof. The inclusion $\leftrightarrow_{\mathcal{R}}^* \subseteq \leftrightarrow_{\mathcal{R}\cup\mathcal{S}}^*$ is obvious. For the reverse direction it suffices to show that $s \leftrightarrow_{\mathcal{R}}^* t$ whenever $s \to_{\mathcal{S}} t$. The latter ensures the existence of a position p in s, a rewrite rule $\ell \to r$ in $\mathcal{S}$, and a substitution σ such that $s|_p = \ell\sigma$ and $t = s[r\sigma]_p$. We obtain $\ell \leftrightarrow_{\mathcal{R}}^* r$ from the assumption of the lemma. Closure (of $\leftrightarrow_{\mathcal{R}}^*$) under contexts and substitutions yields the desired $s \leftrightarrow_{\mathcal{R}}^* t$. ◀

▶ **Corollary 9.** *If $\mathcal{R}$ is confluent and $\ell \leftrightarrow_{\mathcal{R}}^* r$ for every rule $\ell \to r$ from $\mathcal{S}$ then $\mathcal{R} \cup \mathcal{S}$ is confluent.*

Proof. From the preceding lemma and the confluence of $\mathcal{R}$ we obtain

$$\leftrightarrow_{\mathcal{R}\cup\mathcal{S}}^* \;=\; \leftrightarrow_{\mathcal{R}}^* \;\subseteq\; \downarrow_{\mathcal{R}} \;\subseteq\; \downarrow_{\mathcal{R}\cup\mathcal{S}}$$

Hence $\mathcal{R} \cup \mathcal{S}$ is confluent. ◀

▶ **Example 10.** Consider the TRS from [10, Example 2] consisting of the five rewrite rules

$$\begin{array}{lll} \mathsf{hd}(x : y) \to x & \mathsf{inc}(x : y) \to \mathsf{s}(x) : \mathsf{inc}(y) & \mathsf{nats} \to 0 : \mathsf{inc}(\mathsf{nats}) \\ \mathsf{tl}(x : y) \to y & \mathsf{inc}(\mathsf{tl}(\mathsf{nats})) \to \mathsf{tl}(\mathsf{inc}(\mathsf{nats})) & \end{array}$$

While this system can be shown to be confluent using decreasing diagrams, simply removing the last rule would make confluence obvious, since the remaining four rules constitute an orthogonal TRS. And indeed, because of the following joining sequences, the last rule is superfluous and can be dropped:

$$\mathsf{inc}(\mathsf{tl}(\mathsf{nats})) \to \mathsf{inc}(\mathsf{tl}(0 : \mathsf{inc}(\mathsf{nats}))) \to \mathsf{inc}(\mathsf{inc}(\mathsf{nats}))$$
$$\mathsf{tl}(\mathsf{inc}(\mathsf{nats})) \to \mathsf{tl}(\mathsf{inc}(0 : \mathsf{inc}(\mathsf{nats}))) \to \mathsf{tl}(\mathsf{s}(0) : \mathsf{inc}(\mathsf{inc}(\mathsf{nats}))) \to \mathsf{inc}(\mathsf{inc}(\mathsf{nats}))$$

▶ Remark. Some other examples from [10] can be dealt with in a similar fashion: In [10, Example 1] the first rule is joinable using the other rules, and the remaining system is orthogonal. The same argument (with a different joining conversion) applies to [10, Example 5].

Corollary 9 can also be beneficial when dealing with non-left-linear systems, as demonstrated by the following example.

[4] We are interested in transformations that reflect (rather than preserve) confluence, because our goal is automation, and it is natural to work from the conclusion for finding (non-)confluence proofs.

▶ **Example 11.** Consider the TRS from [24] consisting of the four rewrite rules

$$\mathsf{f}(x,x) \to \mathsf{f}(\mathsf{g}(x),\mathsf{g}(x)) \qquad \mathsf{f}(x,y) \to \mathsf{f}(\mathsf{h}(x),\mathsf{h}(y))$$
$$\mathsf{g}(x) \to \mathsf{p}(x) \qquad \mathsf{h}(x) \to \mathsf{p}(x)$$

Because of the conversion

$$\mathsf{f}(x,x) \to \mathsf{f}(\mathsf{h}(x),\mathsf{h}(x)) \to \mathsf{f}(\mathsf{p}(x),\mathsf{g}(x)) \to \mathsf{f}(\mathsf{p}(x),\mathsf{p}(x)) \leftarrow \mathsf{f}(\mathsf{g}(x),\mathsf{p}(x)) \leftarrow \mathsf{f}(\mathsf{g}(x),\mathsf{g}(x))$$

we can remove the first rule. Since the resulting TRS is orthogonal and the removed rule is convertible using the other rules, also the original TRS is confluent.

It can also be beneficial to both add and remove rules. In particular adding a redundant rule can help with removing other, problematic rules, as shown in the following example.

▶ **Example 12.** Consider the TRS consisting of the three rewrite rules[5]

$$\mathsf{f}(x,y) \to \mathsf{f}(\mathsf{g}(x),\mathsf{g}(x)) \qquad \mathsf{f}(x,x) \to \mathsf{a} \qquad \mathsf{g}(x) \to x$$

After adding the rule $\mathsf{f}(x,y) \to \mathsf{a}$, which is justified since $\mathsf{f}(x,y) \to \mathsf{f}(\mathsf{g}(x),\mathsf{g}(x)) \to \mathsf{a}$, we can remove the first two original rules, due to the following conversions:

$$\mathsf{f}(x,y) \to \mathsf{a} \leftarrow \mathsf{f}(\mathsf{g}(x),\mathsf{g}(x)) \qquad \mathsf{f}(x,x) \to \mathsf{a}$$

The resulting TRS is orthogonal and hence confluent. Since the added rule can be simulated by the original rules, and the removed rules are convertible using the new rule, also the original TRS is confluent.

While adding (or removing) rules using Corollary 6 is always safe in the sense that we cannot lose confluence, it is easy to see that the reverse direction of Corollary 9 does not hold in general. That is, removing convertible rules can make a confluent TRS non-confluent as for example witnessed by the two TRSs $\mathcal{R} = \{\mathsf{a} \to \mathsf{b},\, \mathsf{a} \to \mathsf{c}\}$ and $\mathcal{S} = \{\mathsf{b} \to \mathsf{a}\}$. Clearly $\mathcal{R}$ is not confluent, $\mathcal{S} \cup \mathcal{R}$ is confluent, and $\mathsf{b} \leftrightarrow^*_{\mathcal{R}} \mathsf{a}$.

We give one more example, showing that using removal of redundant rules can considerably speed up finding a confluence proof.

▶ **Example 13.** Consider the TRS consisting of the following two rules:

$$\mathsf{f}(x) \to \mathsf{g}(x,\mathsf{f}(x)) \qquad \mathsf{f}(\mathsf{f}(\mathsf{f}(\mathsf{f}(x)))) \to \mathsf{f}(\mathsf{f}(\mathsf{f}(\mathsf{g}(x,\mathsf{f}(x)))))$$

This TRS is confluent by the simultaneous critical pair criterion of Okui [18][6] which is implemented by ACP. Alas, there are 58 simultaneous critical pairs and indeed ACP, which implements Okui's criterion, does not terminate in five minutes. While 58 looks small, the simultaneous critical pairs become quite big. For example, with $t = \mathsf{g}(\mathsf{f}^3(\mathsf{g}(x,\mathsf{f}(x))),\mathsf{f}^4(\mathsf{g}(x,\mathsf{f}(x))))$, one of the simultaneous critical pairs is

$$\mathsf{f}^3(\mathsf{g}(\mathsf{f}(\mathsf{g}(\mathsf{f}(t),\mathsf{f}(\mathsf{f}(t))),\mathsf{f}(\mathsf{f}(\mathsf{g}(\mathsf{f}(t),\mathsf{f}(\mathsf{f}(t)))))))) \nleftarrow\!\!\rtimes\!\!\to \mathsf{f}^5(\mathsf{g}(\mathsf{f}^3(x),\mathsf{f}^4(x)))$$

and testing joinability using development steps is very expensive. In general, if one takes the rules $\mathsf{f}(x) \to \mathsf{g}(x,\mathsf{f}(x))$ and $\mathsf{f}^n(\mathsf{f}(x)) \to \mathsf{f}^n(\mathsf{g}(x,\mathsf{f}(x)))$, then the number and size of the simultaneous critical pair will grow exponentially in n. However, Corollary 9 is applicable—the second rule can be simulated by the first rule in one step—and showing confluence of the first rule is trivial.

[5] http://www.nue.riec.tohoku.ac.jp/tools/acp/experiments/rtatlca14/examples/u1.trs

[6] Note that the given TRS is feebly orthogonal [21]. The key observation here is that any simultaneous critical pair arises from a peak of a development step and a plain rewrite step. By the orthogonalization procedure from [21], we can obtain an equivalent peak of two orthogonal development steps, which is joinable by two development steps, thus satisfying Okui's criterion.

4 Formalization and Certification

Due to the ever increasing interest in automatic analysis of term rewrite systems in the recent years, it is of great importance whether a proof, automatically generated by some tool, is indeed correct. The complexity of the generated proofs makes checking correctness, i.e., certification, impractical for humans. Thus there is a strong interest in automated certification of proofs generated by e.g. confluence or termination tools. This led to the common approach of using proof assistants for certification.

Our technique is particularly well suited for certification for the following reasons. First, since the theory we use is elementary, formalizing it in a proof assistant is entirely straightforward. Moreover the generated proofs, while simple in nature, can become very large, which makes checking them infeasible by hand, but easy for a machine. Finally, as demonstrated in Section 5, the existing certifiable confluence techniques heavily benefit from our transformations.

As certifier we use CeTA [27], originally developed as a tool for certifying termination proofs which have to be provided as certificates in CPF (certification problem format) [23]. Given a certificate CeTA will either answer `CERTIFIED`, or return a detailed error message why the proof was `REJECTED`. Its correctness is formally proven as part of IsaFoR, the **Isa**belle **F**ormalization **o**f **R**ewriting. IsaFoR contains executable "check"-functions for each formalized proof technique together with formal proofs that whenever such a check succeeds, the technique was indeed applied correctly. Isabelle's code-generation facility is used to obtain a trusted Haskell program from these check functions: the certifier CeTA.[7]

Since 2012 CeTA supports checking (non-)confluence certificates [16, 26]. Checkable criteria that ensure confluence are:

- Knuth and Bendix' criterion [15],
- (weak) orthogonality [22],
- Huet's result on strongly closed critical pairs [13], and
- the rule labeling heuristic for decreasing diagrams [17, 20].

For non-confluence CeTA can check that, given derivations $s \to^* t_1$ and $s \to^* t_2$, t_1 and t_2 cannot be joined. Here the supported justifications are:

- testing that t_1 and t_2 are distinct normal forms,
- testing that $\mathsf{tcap}(t_1\sigma)$ and $\mathsf{tcap}(t_2\sigma)$ are not unifiable [29],
- usable rules, discrimination pairs, argument filters, and interpretations [1], and
- reachability analysis using tree automata [8].

To add support for our transformations to CeTA we formalized the results from Section 3 in Isabelle and integrated them into IsaFoR. The theory `Redundant_Rules.thy` contains the theoretical results, whose formalization, directly following the paper proof, requires a mere 100 lines of Isabelle, stressing the simplicity of the transformations.

We extended CPF for representing proofs using addition and removal of redundant rules and implemented dedicated check functions in the theory `Redundant_Rules_Impl.thy`, enabling CeTA to inspect, i.e., certify such (non-)confluence proofs. A certificate for (non-)confluence of a TRS $\mathcal{R}$ by an application of the redundant rules transformation consists of three parts:

- the modified TRS $\mathcal{R}'$,
- a certificate for the (non-)confluence of $\mathcal{R}'$, and

[7] IsaFoR/CeTA and CPF are available at `http://cl-informatik.uibk.ac.at/software/ceta/`.

- a justification for redundancy of the added and removed rules. Here for the rules that were added, i.e., all $\ell \to r$ in $\mathcal{S} = \mathcal{R}' \setminus \mathcal{R}$, we simply require a bound on the length of the derivations showing $\ell \to_{\mathcal{R}}^* r$.[8] For the deleted rules in a non-confluence certificate, i.e., all $\ell \to r$ in $\mathcal{S} = \mathcal{R} \setminus \mathcal{R}'$, the same bound is used for $\ell \to_{\mathcal{R}'}^* r$. For a confluence proof one can either give explicit conversions $\ell \leftrightarrow_{\mathcal{R}'}^* r$ or rely on the bound again, which then has to ensure $\ell \downarrow_{\mathcal{R}'} r$.

Implementing check functions for such a certificate is then straightforward. We simply compute $\mathcal{S} \setminus \mathcal{R}$ and $\mathcal{R} \setminus \mathcal{S}$ and use the given bound and conversions to ensure redundancy.

Whereas for certification we only need to check that the modified rules really are redundant, the question of how to automatically find suitable rules for addition and deletion is more intricate. In the next section we discuss and evaluate our implementation of three possible approaches in the confluence prover CSI.

5 Implementation and Experiments

CSI features a powerful strategy language [29], which allows to combine confluence techniques in a modular and flexible manner, making it easy to test different strategies that exploit redundant rules.

We tested our implementation on the Cops database[9] using the following three strategies to add and remove rules.

(js) Our first strategy is to add (minimal) joining sequences of critical pairs as rules, i.e., in Corollary 6 choose $\mathcal{S} \subseteq \{s \to u, t \to u \mid s \leftarrow\!\rtimes\!\to t \text{ with } s \to_{\mathcal{R}}^* u \text{ and } t \to_{\mathcal{R}}^* u\}$. The underlying idea here is that critical peaks become joinable in a single step, which is advantageous for other confluence criteria, for example rule labeling [20].

(rhs) The second strategy for obtaining redundant rules to add, is to rewrite right-hand sides of rules, i.e., in Corollary 6 set $\mathcal{S} = \{\ell \to t \mid \ell \to r \in \mathcal{R} \text{ and } r \to_{\mathcal{R}} t\}$. (This idea has already been used for termination by Zantema [30].) Again the motivation is to produce shorter joining sequences for critical pairs, facilitating the use of other confluence criteria.

(del) For removing rules we search for rules whose left- and right-hand sides are joinable, i.e., in Corollary 9 set $\mathcal{S} = \{\ell \to r \mid \ell \downarrow_{\mathcal{R}} r\}$. This decision is motivated by simplicity of implementation and the fact that for confluent TRSs, joinability and convertibility coincide. Removing rules can benefit confluence proofs by eliminating critical pairs. Since our strategy here is a simple greedy one that removes as many rules as possible, we also lose confluence in some cases.

In the case of adding rules we also discard rules that can be simulated by other rules in a single step. Without this refinement, the gain in power would become smaller, and even disappear for CSI's full strategy. We also implemented and tested three other strategies, which did not yield any additional proofs.

- Inspired by Example 1 we tried to add rules specifically for making rewrite systems development closed. That is, we used $\mathcal{S} = \{s \to t \mid s \leftarrow\!\rtimes\!\to t \text{ with } s \to_{\mathcal{R}}^* t \text{ and } s \not\twoheadrightarrow_{\mathcal{R}} t\}$ in Corollary 6. All examples gained by this strategy can also be handled by (js) or (rhs).
- To help with systems containing AC-like rules we tried to add inverted reversible rules, by setting $\mathcal{S} = \{r \to \ell \mid \ell \to r \in \mathcal{R} \text{ with } r \to_{\mathcal{R}}^* \ell\}$ in Corollary 6. Again we gained no additional proofs compared to (js) and (rhs).

[8] This bound is necessary, because in Isabelle all functions have to be total and an unbounded search might not terminate.

[9] All TRS problems (276 at the time of writing) from `http://coco.nue.riec.tohoku.ac.jp/cops/`.

Table 1 Experimental Results (✓ = certified).

	CSI	$\mathsf{CSI}_{\mathsf{js}}$	$\mathsf{CSI}_{\mathsf{rhs}}$	$\mathsf{CSI}_{\mathsf{del}}$	$\mathsf{CSI}_{\mathsf{all}}$
yes	155	156	159	163	166
no	47	48	47	47	48
maybe	74	72	70	66	62
	✓CSI	✓$\mathsf{CSI}_{\mathsf{js}}$	✓$\mathsf{CSI}_{\mathsf{rhs}}$	✓$\mathsf{CSI}_{\mathsf{del}}$	✓$\mathsf{CSI}_{\mathsf{all}}$
yes	71	86	73	78	104
no	47	48	47	47	48
maybe	158	142	156	151	124

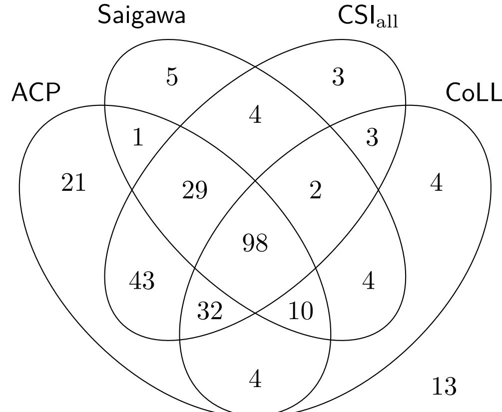


Figure 1 Overlap Between Solved Examples.

- When removing rules we also tried to search for conversions that are not valleys, by using rules in the reverse direction when searching for a join. More precisely, we tried $\mathcal{S} = \{\ell \to r \mid \ell \downarrow_{\mathcal{R} \cup \mathcal{R}^{-1}} r\}$ in Corollary 9. However, this variation only lost examples compared to (del).

The results are shown in Table 1.[10] The experiments were performed on a 48 core 2.2 GHz Opteron 6174 server with 256 GB RAM. We performed two sets of benchmarks, based on CSI's full and certifiable strategies, respectively. For the full strategy, adding joining sequences of critical pairs (js) or rewriting right-hand sides (rhs) show very limited effect, gaining 2 and 4 proofs, respectively. Removing rules (del) is the most effective technique and gains 10 systems while losing 2 other ones. With all techniques combined, 12 new systems can be shown (non-)confluent. Interestingly, the picture for the certifiable strategy is a bit different. Here, (rhs) gains 2 proofs, (js) gains 16 systems and (del) gains 17 proofs while losing 10. Remarkably, in all of those 17 proofs the TRS becomes orthogonal after removing redundant rules, which emphasizes that our transformations can considerably simplify confluence proofs. In total, 34 new systems are shown (non-)confluent.

In Table 2, we compare CSI 0.5,to ACP 0.50, CoLL 1.1, and Saigawa 1.7. Figure 1 shows the examples solved by the four provers in relation to each other.

[10] Detailed results are available at `http://cl-informatik.uibk.ac.at/software/csi/rr-rta2015/`.

Table 2 Comparison of Confluence Provers.

	ACP	CoLL	$\mathsf{CSI}_{\text{all}}$	Saigawa
yes	186	141	166	128
no	52	16	48	25
maybe	38	119	62	123

6 Related Work

Our work draws a lot of inspiration from existing literature. One starting point is [21], where van Oostrom introduces the notion of *feeble orthogonality*. A TRS is feebly orthogonal if the critical peaks arising from its non-redundant* rules are trivial or contain a trivial step (that rewrites a term to itself); a rule is redundant* if it can be simulated by another rule in a single step. Clearly our notion of redundancy generalizes redundancy*.

The most important prior work is [2]. In this paper, Aoto and Toyama describe an automated confluence criterion (which has been implemented in ACP) based on decomposing TRSs into a *reversible* part $\mathcal{P}$ and a terminating part $\mathcal{S}$. In order to help applicability of their criterion, they introduce a procedure based on the inference rules

$$\text{replace} \quad \frac{\langle \mathcal{S} \cup \{\ell \to r\}, \mathcal{P}\rangle}{\langle \mathcal{S} \cup \{\ell \to r'\}, \mathcal{P}\rangle} \quad r \leftrightarrow^*_{\mathcal{P}} r' \qquad\qquad \text{add} \quad \frac{\langle \mathcal{S}, \mathcal{P}\rangle}{\langle \mathcal{S} \cup \{\ell \to r\}, \mathcal{P}\rangle} \quad \ell \leftrightarrow^*_{\mathcal{P}} \cdot \to^*_{\mathcal{S}} r$$

The key is that because $\mathcal{P}$ is reversible, $\leftrightarrow^*_{\mathcal{P}}$ and $\to^*_{\mathcal{P}}$ coincide, and therefore confluence of $\mathcal{S} \cup \mathcal{P}$ is not affected by applying these inference rules. This very same idea underlies Lemma 5, which establishes *reduction equivalence*, and thus Corollary 6. Note that no rule removal is performed in [2].

There is a second connection between our work and [2] that seems noteworthy. Given a reversible $\mathcal{P}$, every rule from $\mathcal{P}^{-1}$ can be simulated by a sequence of $\mathcal{P}$-steps. Therefore, confluence of $\mathcal{S} \cup \mathcal{P}$ and $\mathcal{S} \cup \mathcal{P} \cup \mathcal{P}^{-1}$ coincide by Corollary 6. Using this observation, one could decompose the confluence criteria of [2] into two steps, one that replaces $\mathcal{P}$ by $\mathcal{P} \cup \mathcal{P}^{-1}$, and a respective underlying confluence criterion that does not make use of reversibility, but instead demands that $\mathcal{P}$ is symmetric, i.e., $\mathcal{P}^{-1} \subseteq \mathcal{P}$.

The idea of showing confluence by removing rules whose sides are convertible has already been used in the literature, e.g. [12, Example 11], which is a variation of Example 10.

Other works of interest are [9, 30], where Gramlich and Zantema apply a similar idea to Corollary 6 to termination: If some additional requirements are met, then termination of $\mathcal{R} \cup \{\ell \to r\}$ is equivalent to termination of $\mathcal{R} \cup \{\ell \to r'\}$ where $r \to_{\mathcal{R}} r'$ by a non-erasing rule. This is true for non-overlapping TRSs [9, Theorem 4], or when the rule used in the $r \to_{\mathcal{R}} r'$ step is locally confluent by itself, left-linear, and furthermore it doesn't overlap with any rules from $\mathcal{R} \cup \{\ell \to r\}$ except itself [30, Theorem 4].

7 Conclusion

In this work we demonstrated how a very simple technique, namely adding and removing redundant rules, can boost the power of automated confluence provers. It is easy to implement and we believe that also confluence tools other than CSI could benefit from such transformations, not only increasing their power, but also simplifying the generated proofs. Moreover the technique is well-suited for certification, resulting in more trustworthy proofs. In particular we could significantly increase the number of certifiable confluence proofs in our

experiments—by almost 50%. Interestingly we observed that all 17 of the systems gained by ✓(del) become orthogonal by removing redundant rules. This might be due to the fact that when designing example TRSs for new techniques, one often works by systematically making existing criteria non-applicable and removing rules can undo this effort.

As future work we plan to investigate more elaborate strategies for finding useful redundant rules, both for addition and removal (where the candidates are limited, but performing the transformation might lose confluence). Here one direction to explore might be the use of machine learning techniques to devise such strategies automatically.

Acknowledgments. The comments by the anonymous reviewers helped to improve the presentation.

References

1. T. Aoto. Disproving confluence of term rewriting systems by interpretation and ordering. In P. Fontaine, editor, *Proc. 9th International Workshop on Frontiers of Combining Systems*, volume 8152 of *Lecture Notes in Artificial Intelligence*, pages 311–326, 2013. doi: 10.1007/978-3-642-40885-4_22.
2. T. Aoto and Y. Toyama. A reduction-preserving completion for proving confluence of non-terminating term rewriting systems. *Logical Methods in Computer Science*, 8(1:31):1–29, 2012. doi: 10.2168/LMCS-8(1:31)2012.
3. T. Aoto, J. Yoshida, and Y. Toyama. Proving confluence of term rewriting systems automatically. In R. Treinen, editor, *Proc. 20th International Conference on Rewriting Techniques and Applications*, volume 5595 of *Lecture Notes in Computer Science*, pages 93–102, 2009. doi: 10.1007/978-3-642-02348-4_7.
4. F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.
5. F. Blanqui and A. Koprowski. CoLoR, a Coq library on well-founded rewrite relations and its application to the automated verification of termination certificates. *Mathematical Structures in Computer Science*, 21(4):827–859, 2011. doi: 10.1017/S0960129511000120.
6. E. Contejean, P. Courtieu, J. Forest, O. Pons, and Xavier Urbain. Automated certified proofs with CiME3. In M. Schmidt-Schauß, editor, *Proc. 22nd International Conference on Rewriting Techniques and Applications*, pages 21–30, 2011. doi: 10.4230/LIPIcs.RTA.2011.21.
7. B. Felgenhauer. A proof order for decreasing diagrams. In N. Hirokawa and A. Middeldorp, editors, *Proc. 1st International Workshop on Confluence*, pages 7–14, 2012. `http://cl-informatik.uibk.ac.at/events/iwc-2012/`.
8. B. Felgenhauer and R. Thiemann. Reachability analysis with state-compatible automata. In A.-H. Dediu, editor, *Proc. 8th International Conference on Language and Automata Theory and Applications*, volume 8370 of *Lecture Notes in Computer Science*, pages 347–359, 2013. doi: 10.1007/978-3-319-04921-2_28.
9. B. Gramlich. Simplifying termination proofs for rewrite systems by preprocessing. In F. Pfenning, editor, *Proc. 2nd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 139–150, 2000. doi: 10.1145/351268.351286.
10. B. Gramlich and S. Lucas. Generalizing Newman's lemma for left-linear rewrite systems. In F. Pfenning, editor, *Proc. 17th International Conference on Rewriting Techniques and Applications*, volume 4098 of *Lecture Notes in Computer Science*, pages 66–80, 2006. doi: 10.1007/11805618_6.

11 N. Hirokawa and D. Klein. Saigawa: A confluence tool. In N. Hirokawa and A. Middeldorp, editors, *Proc. 1st International Workshop on Confluence*, page 49, 2012. http://cl-informatik.uibk.ac.at/events/iwc-2012/.

12 N. Hirokawa and A. Middeldorp. Decreasing diagrams and relative termination. *Journal of Automated Reasoning*, 47(4):481–501, 2011. doi: 10.1007/s10817-011-9238-x.

13 G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of the ACM*, 27(4):797–821, 1980. doi: 10.1145/322217.322230.

14 D. Klein and N. Hirokawa. Confluence of non-left-linear TRSs via relative termination. In N. Bjørner and A. Voronkov, editors, *Proc. 18th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, volume 7180 of *Lecture Notes in Computer Science (Advanced Research in Computing and Software Science)*, pages 258–273, 2012.

15 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.

16 J. Nagele and R. Thiemann. Certification of confluence proofs using CeTA. In T. Aoto and D. Kesner, editors, *Proc. 3rd International Workshop on Confluence*, pages 19–23, 2014. http://www.nue.riec.tohoku.ac.jp/iwc2014/.

17 J. Nagele and H. Zankl. Certified rule labeling. In M. Fernández, editor, *Proc. 26th International Conference on Rewriting Techniques and Applications*, volume 36 of *Leibniz International Proceedings in Informatics*, 2015. This volume.

18 S. Okui. Simultaneous critical pairs and Church-Rosser property. In T. Nipkow, editor, *Proc. 9th International Conference on Rewriting Techniques and Applications*, volume 1379 of *Lecture Notes in Computer Science*, pages 2–16, 1998. doi: 10.1007/BFb0052357.

19 V. van Oostrom. Developing developments. *Theoretical Computer Science*, 175(1):159–181, 1997. doi: 10.1016/S0304-3975(96)00173-9.

20 V. van Oostrom. Confluence by decreasing diagrams – converted. In A. Voronkov, editor, *Proc. 19th International Conference on Rewriting Techniques and Applications*, volume 5117 of *Lecture Notes in Computer Science*, pages 306–320, 2008. doi: 10.1007/978-3-540-70590-1_21.

21 V. van Oostrom. Feebly not weakly. In *Proc. 7th International Workshop on Higher-Order Rewriting*, Vienna Summer of Logic flash drive, 2014.

22 B.K. Rosen. Tree-manipulating systems and Church-Rosser theorems. *Journal of the ACM*, 20(1):160–187, 1973. doi: 10.1145/321738.321750.

23 C. Sternagel and R. Thiemann. The certification problem format. In G. Klein and R. Gamboa, editors, *Proc. 11th International Workshop on User Interfaces for Theorem Provers*, volume 167 of *Electronic Proceedings in Theoretical Computer Science*, pages 61–72, 2014. doi: 10.4204/EPTCS.167.8.

24 T. Suzuki, T. Aoto, and Y. Toyama. Confluence proofs of term rewriting systems based on persistency. *Computer Software*, 30(3):148–162, 2013. In Japanese, doi: 10.11309/jssst.30.3_148.

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

26 R. Thiemann. Certification of confluence proofs using CeTA. In N. Hirokawa and A. Middeldorp, editors, *Proc. 1st International Workshop on Confluence*, page 45, 2012. http://cl-informatik.uibk.ac.at/events/iwc-2012/.

27 R. Thiemann and C. Sternagel. Certification of termination proofs using CeTA. In S. Berghofer, editor, *Proc. 22nd International Conference on Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 452–468, 2009. doi: 10.1007/978-3-642-03359-9_31.

28 Y. Toyama. Commutativity of term rewriting systems. In K. Fuchi and L. Kott, editors, *Programming of Future Generation Computers II*, pages 393–407. North-Holland, 1988.

29 H. Zankl, B. Felgenhauer, and A. Middeldorp. CSI – A confluence tool. In N. Bjørner and V. Sofronie-Stokkermans, editors, *Proc. 23rd International Conference on Automated Deduction*, volume 6803 of *Lecture Notes in Artificial Intelligence*, pages 499–505, 2011. doi: 10.1007/978-3-642-22438-6_38.

30 H. Zantema. Reducing right-hand sides for termination. In V. van Oostrom A. Middeldorp and and F. van Raamsdonk, editors, *Processes, Terms and Cycles: Steps on the Road to Infinity, Essays Dedicated to Jan Willem Klop, on the Occasion of his 60th Birthday*, volume 3838 of *Lecture Notes in Computer Science*, pages 173–197, 2005. doi: 10.1007/11601548_12.

Certified Rule Labeling*

Julian Nagele and Harald Zankl

Institute of Computer Science, University of Innsbruck, Austria
`{julian.nagele|harald.zankl}@uibk.ac.at`

Abstract

The rule labeling heuristic aims to establish confluence of (left-)linear term rewrite systems via decreasing diagrams. We present a formalization of a confluence criterion based on the interplay of relative termination and the rule labeling in the theorem prover Isabelle. Moreover, we report on the integration of this result into the certifier CeTA, facilitating the checking of confluence certificates based on decreasing diagrams for the first time. The power of the method is illustrated by an experimental evaluation on a (standard) collection of confluence problems.

1998 ACM Subject Classification F.2 Analysis of Algorithms and Problem Complexity, F.4 Mathematical Logic and Formal Languages

Keywords and phrases term rewriting, confluence, decreasing diagrams, certification

Digital Object Identifier 10.4230/LIPIcs.RTA.2015.269

1 Introduction

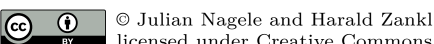
Confluence is an important property of rewrite systems as it ensures unique normal forms. The recent achievements in confluence research have enabled a competition[1] where automated tools try to establish/refute confluence. As the proofs produced by these tools are often complicated and large, there is interest in checking them within a trustable certifier.

Decreasing diagrams [15] provide a complete characterization of confluence for abstract rewrite systems whose convertibility classes are countable. As a criterion for abstract rewrite systems, they can be applied to first- and higher-order rewriting, including term rewriting and the λ-calculus. In this paper we build upon the recent Isabelle formalization of decreasing diagrams (see [28, 29]) and specialize it from abstract rewriting to term rewriting. Moreover, we formalize the rule labeling and present a mechanized proof of the following result (see [31, Corollary 16]):

▶ **Theorem 1.** *A left-linear term rewrite system is confluent if its duplicating rules terminate relative to its other rules and all its critical peaks are decreasing for the rule labeling.*

This result is an adequate candidate for a formalization because of the following reasons. On the one hand, regarding the aspect of automation, it is easily implementable as the relative termination requirement can be outsourced to external (relative) termination provers and the rule labeling heuristic has already been implemented successfully [1, 7]. Furthermore, it is a powerful criterion as demonstrated by an experimental evaluation in Section 6. On the other hand, regarding the aspect of formalization, it is challenging because it involves the combination of different labeling functions (in the sense of [31]). Hence, in our formalization Theorem 1 is not established directly, but obtained as a corollary of more general results.

* This research is supported by FWF (Austrian Science Fund) project P27528.
[1] `http://coco.nue.riec.tohoku.ac.jp/2014`


© Julian Nagele and Harald Zankl;
licensed under Creative Commons License CC-BY
26th International Conference on Rewriting Techniques and Applications (RTA'15).
Editor: Maribel Fernández; pp. 269–284
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

This paves the way for reusing the formalization described here when tackling the remaining criteria in [31].

We based our formalization on the **Isa**belle **F**ormalization **o**f **R**ewriting (IsaFoR) [27] and extended it by the theories `Decreasing_Diagrams2.thy` and `Rule_Labeling_Impl.thy`, which amount to approximately 3500 lines of Isabelle in Isar style. IsaFoR contains executable check functions for each formalized proof technique together with formal proofs that whenever such a check is accepted, the technique is applied correctly. Then Isabelle's code-generation facility is used to obtain a trusted Haskell program, i.e., the certifier CeTA, which is capable of checking proof certificates in CPF [22] (certification problem format).[2] We suitably extended CPF to represent proofs according to Theorem 1 and implemented dedicated check functions in our formalization, enabling CeTA to inspect, i.e., certify such confluence proofs. Typically, these proofs are generated by automated confluence tools. (See Footnote 1 for details.)

A preliminary result of our formalization has already been proved useful in the latest edition of the confluence competition (CoCo 2014), where CeTA certified confluence proofs for *linear* rewrite systems based on the rule labeling (among others). The main challenge in lifting the result from linear to left-linear rewrite systems has not been the relative termination requirement per se, which vacuously holds in the linear case, but the interplay of the relative termination condition with the rule labeling, which is crucial in the the proof of Theorem 1, albeit in the statement of the result these concepts are clearly separated. Besides, to establish decreasingness of variable peaks (involving non-right-linear rules) more details about the joining sequences were needed than the existing theories in IsaFoR provided.

The remainder of this paper is organized as follows. Preliminaries are introduced in the next section. The interplay of several labeling functions favors the notion of *extended local decreasingness* [7], which is proved to imply *local decreasingness* in Section 3, where also the connection to the existing formalization of decreasing diagrams for abstract rewrite systems [28, 29] is established. Afterwards, Section 4 lifts extended local decreasingness from abstract rewriting to results for term rewriting that are parametrized by a labeling. Section 5 instantiates these results with concrete labeling functions to obtain corollaries that ensure confluence. Section 6 presents an experimental evaluation, before we conclude in Section 7.

The full formalization is available from the URL in Footnote 2.

2 Preliminaries

We assume familiarity with rewriting [25] and decreasing diagrams [15]. Basic knowledge of Isabelle [14] is not essential but experience with an interactive theorem prover might be helpful.

Let $\mathcal{F}$ be a signature and $\mathcal{V}$ 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 $\mathcal{V}$. Positions are strings of positive natural numbers, i.e., elements of $\mathbb{N}_+^*$. We write $q \leqslant p$ if $qq' = p$ for some position q', in which case $p\backslash q$ is defined to be q'. Furthermore $q < p$ if $q \leqslant p$ and $q \neq p$. Finally, $q \parallel p$ if neither $q \leqslant p$ nor $p < q$. Positions are used to address subterm occurrences. The set of positions of a term t is defined as $\mathcal{P}\mathsf{os}(t) = \{\epsilon\}$ if t is a variable and as $\mathcal{P}\mathsf{os}(t) = \{\epsilon\} \cup \{iq \mid 1 \leqslant i \leqslant n \text{ and } q \in \mathcal{P}\mathsf{os}(t_i)\}$ if $t = f(t_1,\ldots,t_n)$. 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 $p = iq$ and $t = f(t_1,\ldots,t_n)$. We write $s[t]_p$ for the result of replacing the occurrence of $s|_p$ 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 $\mathcal{P}\mathsf{os}_\mathcal{V}(t) = \mathcal{P}\mathsf{os}(t) \setminus \mathcal{P}\mathsf{os}_\mathcal{F}(t)$.

[2] IsaFoR/CeTA and CPF are available at `http://cl-informatik.uibk.ac.at/software/ceta/`.

A rewrite rule is a pair of terms (l, r), written $l \to r$.[3] A rewrite rule $l \to r$ is duplicating if $|l|_x < |r|_x$ for some $x \in \mathcal{V}$. Here the expression $|t|_x$ indicates the number of occurrences of the variable x in term t. A term rewrite system (TRS) is a signature together with a set of rewrite rules over this signature. In the sequel, signatures are left 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 $\to_{\mathcal{R}}$ (often written as $\to$) to be the smallest rewrite relation that contains $\mathcal{R}$. As usual $\to^=$, $\to^+$, and $\to^*$ denote the reflexive, transitive, and reflexive and transitive closure of $\to$, respectively, while $\to^n$ denotes the n-fold composition of $\to$.

A relative TRS $\mathcal{R}/\mathcal{S}$ is a pair of TRSs $\mathcal{R}$ and $\mathcal{S}$ with the induced rewrite relation $\to_{\mathcal{R}/\mathcal{S}} = \to_{\mathcal{S}}^* \cdot \to_{\mathcal{R}} \cdot \to_{\mathcal{S}}^*$. Sometimes we identify a TRS $\mathcal{R}$ with the relative TRS $\mathcal{R}/\varnothing$ and vice versa. A TRS $\mathcal{R}$ is terminating (relative to a TRS $\mathcal{S}$) if $\to_{\mathcal{R}}$ ($\to_{\mathcal{R}/\mathcal{S}}$) is well-founded.

A critical 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 in $\mathcal{R}$ without common variables, a position $p \in \mathcal{P}\mathsf{os}_{\mathcal{F}}(l_2)$, and a most general unifier μ of l_1 and $l_2|_p$. From a critical 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 $l_2\mu[r_1\mu]_p \leftarrow\!\rtimes\!\to r_2\mu$.

If $l \to r \in \mathcal{R}$, p is a position, and σ is a substitution we call the triple $\pi = \langle p, l \to r, \sigma \rangle$ a redex pattern, and write p_π, l_π, r_π, σ_π for its position, left-hand side, right-hand side, and substitution, respectively. We write $\to^\pi$ (or $\to^{p_\pi, l_\pi \to r_\pi, \sigma_\pi}$) for a rewrite step at position p_π using the rule $l_\pi \to r_\pi$ and the substitution σ_π. A redex pattern π matches a term t if $t|_{p_\pi} = l_\pi \sigma_\pi$, which is then called a redex.

Let π_1 and π_2 be redex patterns that match a common term. They are called parallel, written $\pi_1 \parallel \pi_2$, if $p_{\pi_1} \parallel p_{\pi_2}$. If $P = \{\pi_1, \dots, \pi_n\}$ is a set of pairwise parallel redex patterns matching a term t, we denote by $t \mathrel{+\!\!\!\!\twoheadrightarrow}^P t'$ the parallel rewrite step from t to t' by P, i.e., $t \to^{\pi_1} \cdot \ldots \cdot \to^{\pi_n} t'$.

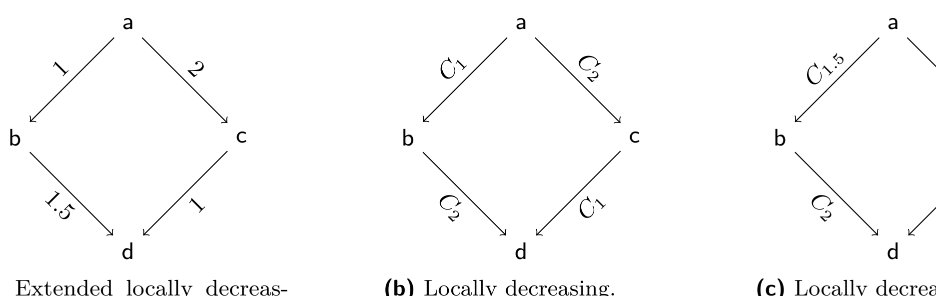
In IsaFoR, an abstract rewrite system (ARS) is a binary relation $\to$ where the domain is left implicit in the type. Let I be an index set. We write $\{\to_\alpha\}_{\alpha \in I}$ to denote the ARS $\to$ where $\to$ is the union of $\to_\alpha$ for all $\alpha \in I$. Let $\{\to_\alpha\}_{\alpha \in I}$ be an ARS and let $>$ and $\geqslant$ be relations on I. Two relations $>$ and $\geqslant$ are called compatible if $\geqslant \cdot > \cdot \geqslant \subseteq >$. Given a relation $\succcurlyeq$ we write $\to_{\curlyvee \alpha_1 \cdots \alpha_n}$ for the union of $\to_\beta$ where $\alpha_i \succcurlyeq \beta$ for some $1 \leqslant i \leqslant n$. Similarly, $\curlyvee S$ is the set of all β such that $\alpha \succcurlyeq \beta$ for some $\alpha \in S$. We call α and β *extended locally decreasing* (for $>$ and $\geqslant$) if ${}_\alpha\!\leftarrow \cdot \to_\beta \subseteq \to_{\vee\alpha}^* \cdot \to_{\vee\!\!\vee\beta}^= \cdot \to_{\vee\alpha\beta}^* \cdot {}_{\vee\alpha\beta}^{\;\;*}\!\!\leftarrow \cdot {}_{\vee\!\!\vee\alpha}^{\;\;=}\!\!\leftarrow \cdot {}_{\vee\beta}^{\;\;*}\!\!\leftarrow$. If there exist a well-founded order $>$ and a preorder $\geqslant$, such that $>$ and $\geqslant$ are compatible, and α and β are extended locally decreasing for all $\alpha, \beta \in I$ then the ARS $\{\to_\alpha\}_{\alpha \in I}$ is *extended locally decreasing* (for $>$ and $\geqslant$). We call an ARS *locally decreasing* (for $>$) if it is extended locally decreasing for $>$ and $=$, where the latter is the identity relation. In the sequel, we often refer to extended locally decreasing as well as to locally decreasing just by decreasing, whenever the context clarifies which concept is meant or the exact meaning is irrelevant.

3 Abstract Rewriting

This section is concerned with the formalization of the following result from [7, Theorem 2]:

▶ **Lemma 2.** *Every extended locally decreasing ARS is confluent.* ◀

[3] We do not require the common *variable conditions*, i.e., the restriction that l is not a variable and all variables in r are contained in l.



(a) Extended locally decreasing. **(b)** Locally decreasing. **(c)** Locally decreasing.

Figure 1 (Extended) locally decreasing peaks.

The results for decreasing diagrams formalized in [28] differ from the above lemma for two reasons. Firstly, [28] establishes results for local decreasingness instead of extended local decreasingness. Secondly, in contrast to the formulation of the lemma above it does not represent the ARS as a family of rewrite relations (i.e., $\{\to_\alpha\}_{\alpha \in I}$) but considers a single labeled relation where a triple (a, α, b) expresses that $(a, b) \in \to_\alpha$.

Given an ARS that is extended locally decreasing for $>$ and $\geqslant$, the proof in [7] constructs a single order $\succ$ on sets of labels and establishes local decreasingness of the ARS for $\succ$. Our formalization goes along the lines with the proposed proof (see below). It turned out that the representation of the ARS as a family of relations is essential to follow the proof in [7]. Hence establishing equivalence of a single labeled ARS with a family of rewrite relations is needed to employ the formalization of [28] in the proof of Lemma 2. This equivalence looks trivial at first sight, but as each representation comes with a different formalization of rewrite steps, also related concepts such as local peaks, joining sequences, and local decreasingness, must be mapped. We refer the interested reader to the formalization and do not present the technical details here.

The remainder of this section sketches the formalization of the next lemma (following [7]).

▶ **Lemma 3.** *Every extended locally decreasing ARS is locally decreasing.*

To prepare for its proof we consider sets of labels.

▶ **Definition 4.** Let C_α denote the set $\{\alpha' \mid \alpha \geqslant \alpha' \text{ and } \alpha \not> \alpha'\}$ and let $\mathcal{C}$ be the set of all C_α. For $C, D \in \mathcal{C}$ let $C \succ D$ if there exist α and β with $C = C_\alpha$, $D = C_\beta$, and $\alpha > \beta$. By $\to_C$, we denote the union of $\to_\alpha$ for all $\alpha \in C$.

The idea is to establish $\{\to_\alpha\}_{\alpha \in I} = \{\to_C\}_{C \in \mathcal{C}}$ and conclude local decreasingness of the ARS $\{\to_C\}_{C \in \mathcal{C}}$ based on extended local decreasingness of the ARS $\{\to_\alpha\}_{\alpha \in I}$. The next example demonstrates some peculiarities of this approach.

▶ **Example 5.** Consider the ARS $\{\to_\alpha\}_{\alpha \in \{1,1.5,2\}}$ with $\to_1 = \{(\mathsf{a},\mathsf{b}),(\mathsf{c},\mathsf{d})\}$, $\to_{1.5} = \{(\mathsf{b},\mathsf{d})\}$, and $\to_2 = \{(\mathsf{a},\mathsf{c})\}$. This ARS is extended locally decreasing for $>_\mathbb{N}$ and $\geqslant_\mathbb{Q}$, as depicted in Figure 1(a). We have $\mathcal{C} = \{C_2, C_{1.5}, C_1\}$ with $C_2 = \{2, 1.5\}$, $C_{1.5} = \{1.5, 1\}$, and $C_1 = \{1\}$. E.g. $1.5 \in C_2$ since $2 \geqslant_\mathbb{Q} 1.5$ but $2 \not>_\mathbb{N} 1.5$. Consequently, $\to_{C_2} = \{(\mathsf{a},\mathsf{c}),(\mathsf{b},\mathsf{d})\}$, $\to_{C_{1.5}} = \{(\mathsf{b},\mathsf{d}),(\mathsf{a},\mathsf{b}),(\mathsf{c},\mathsf{d})\}$, and $\to_{C_1} = \{(\mathsf{a},\mathsf{b}),(\mathsf{c},\mathsf{d})\}$. To establish local decreasingness of the related ARS $\{\to_C\}_{C \in \mathcal{C}}$ the peak $\mathsf{b} \;{}_{C_1}\!\leftarrow \mathsf{a} \to_{C_2} \mathsf{c}$ (emerging from $\mathsf{b} \;{}_1\!\leftarrow \mathsf{a} \to_2 \mathsf{c}$) must be considered, which can be closed in a locally decreasing fashion via $\mathsf{b} \to_{C_2} \mathsf{d} \;{}_{C_1}\!\leftarrow \mathsf{c}$ (based on $\mathsf{b} \to_{1.5} \mathsf{d} \;{}_1\!\leftarrow \mathsf{c}$), as in Figure 1(b). However, the construction also admits the peak

$\mathsf{b}\ {}_{C_{1.5}}\!\!\leftarrow \mathsf{a} \rightarrow_{C_2} \mathsf{c}$, for which there is no peak $\mathsf{b}\ {}_{1.5}\!\!\leftarrow \mathsf{a} \rightarrow_2 \mathsf{c}$ in the original ARS, as it does not contain the step $\mathsf{b}\ {}_{1.5}\!\!\leftarrow \mathsf{a}$. Still, this peak can be closed locally decreasing, cf. Figure 1(c).

The following properties are crucial:

▶ **Lemma 6.** *Let $>$ be a well-founded order and $\geqslant$ a preorder compatible with $>$.*

1. *Then $\succ$ is a well-founded order.*
2. *If $\gamma \geqslant \gamma'$, $\delta \geqslant \delta'$, and $x \rightarrow^*_{\vee\gamma'\delta'} y$ then $x \rightarrow^*_{\vee\gamma\delta} y$.*
3. *If $\gamma \geqslant \gamma'$ and $x \rightarrow^=_{\vee\!\!\vee\gamma'} y$ then $x \rightarrow^=_{\vee\!\!\vee\gamma} y$.*
4. *If $x \rightarrow^*_{\vee\gamma\delta} y$ then $x \rightarrow^*_{\curlyvee C_\gamma C_\delta} y$.*
5. *If $x \rightarrow^=_{\vee\!\!\vee\gamma} y$ then $x \rightarrow^=_{C_\gamma} y$ or $x \rightarrow_{\curlyvee C_\gamma} y$.*

Proof. Items (1–3) follow from the properties of the orders. Items (4) and (5) are established as in [7]. ◀

Item (1) of Lemma 6, i.e., well-foundedness of $\succ$ is not proved explicitly in [7]. Moreover items (2) and (3) are missing in [7]. Their need becomes apparent in the following proof. In [8] (the journal version of [7]) extended local decreasingness is avoided by employing the *predecessor labeling*. Then a rewrite step comes with a set of labels, which is typically not computable and hence inappropriate for certification.

Proof of Lemma 3. We assume the ARS $\{\rightarrow_\alpha\}_{\alpha\in I}$ is extended locally decreasing for $>$ and $\geqslant$ and establish local decreasingness of the ARS $\{\rightarrow_C\}_{C\in\mathcal{C}}$ for $\succ$ by showing

$$\xleftarrow[C]{} \cdot \xrightarrow[D]{} \subseteq \xrightarrow[\curlyvee C]{*} \cdot \xrightarrow[D]{=} \cdot \xrightarrow[\curlyvee CD]{*} \cdot \xleftarrow[\curlyvee CD]{*} \cdot \xleftarrow[C]{=} \cdot \xleftarrow[\curlyvee D]{*} \tag{1}$$

for $C, D \in \mathcal{C}$.[4] By definition of C and D, there exist α and β with $C = C_\alpha$ and $D = C_\beta$, i.e., ${}_C\!\!\leftarrow \cdot \rightarrow_D = {}_{C_\alpha}\!\!\leftarrow \cdot \rightarrow_{C_\beta} = \bigcup_{\alpha'\in C_\alpha,\beta'\in C_\beta} {}_{\alpha'}\!\!\leftarrow \cdot \rightarrow_{\beta'}$. We note that from $y\ {}_{C_\alpha}\!\!\leftarrow x$ in general we may not infer $y\ {}_\alpha\!\!\leftarrow x$, but rather $y\ {}_{\alpha'}\!\!\leftarrow x$ for some $\alpha' \in C_\alpha$ (cf. Example 5). Similarly $x \rightarrow_{C_\beta} z$ implies $x \rightarrow_{\beta'} z$ for some $\beta' \in C_\beta$. Consequently, the extended local decreasingness assumption cannot be applied to α and β (as conveyed in [7]) but must be applied to α' and β' (as sketched in Example 5), i.e.,

$$\xleftarrow[\alpha']{} \cdot \xrightarrow[\beta']{} \subseteq \xrightarrow[\vee\alpha']{*} \cdot \xrightarrow[\vee\!\!\vee\beta']{=} \cdot \xrightarrow[\vee\alpha'\beta']{*} \cdot \xleftarrow[\vee\alpha'\beta']{*} \cdot \xleftarrow[\vee\!\!\vee\alpha']{=} \cdot \xleftarrow[\vee\beta']{*}$$

Then we establish

$$\xrightarrow[\vee\alpha']{*} \cdot \xrightarrow[\vee\!\!\vee\beta']{=} \cdot \xrightarrow[\vee\alpha'\beta']{*} \cdot \xleftarrow[\vee\alpha'\beta']{*} \cdot \xleftarrow[\vee\!\!\vee\alpha']{=} \cdot \xleftarrow[\vee\beta']{*} \subseteq \xrightarrow[\vee\alpha]{*} \cdot \xrightarrow[\vee\!\!\vee\beta]{=} \cdot \xrightarrow[\vee\alpha\beta]{*} \cdot \xleftarrow[\vee\alpha\beta]{*} \cdot \xleftarrow[\vee\!\!\vee\alpha]{=} \cdot \xleftarrow[\vee\beta]{*}$$

using Lemma 6(2-3), from which the desired

$$\xleftarrow[C_\alpha]{} \cdot \xrightarrow[C_\beta]{} \subseteq \xrightarrow[\curlyvee C_\alpha]{*} \cdot \xrightarrow[C_\beta]{=} \cdot \xrightarrow[\curlyvee C_\alpha C_\beta]{*} \cdot \xleftarrow[\curlyvee C_\alpha C_\beta]{*} \cdot \xleftarrow[C_\alpha]{=} \cdot \xleftarrow[\curlyvee C_\beta]{*}$$

is obtained using Lemma 6(4–5). Depending on the case of Lemma 6(5) that applies, the reflexive step either stays, if e.g. $\rightarrow^=_{\vee\!\!\vee\beta}$ becomes $\rightarrow^=_{C_\beta}$, or is merged with the subsequent sequence having smaller labels, if e.g. $\rightarrow^=_{\vee\!\!\vee\beta}$ becomes $\rightarrow_{\curlyvee C_\beta}$, establishing the property (1). The proof concludes by the equivalence $\{\rightarrow_\alpha\}_{\alpha\in I} = \{\rightarrow_C\}_{C\in\mathcal{C}}$, as in [7]. ◀

[4] In [7] the (stronger) property ${}_C\!\!\leftarrow \cdot \rightarrow_D \subseteq \rightarrow^*_{\curlyvee C} \cdot \rightarrow^=_{\curlyvee D} \cdot \rightarrow^*_{\curlyvee CD} \cdot {}_{\curlyvee CD}^{\ \ *}\!\!\leftarrow \cdot {}_{\curlyvee C}^{\ =}\!\!\leftarrow \cdot {}_{\curlyvee D}^{\ *}\!\!\leftarrow$ is claimed, but as this is obviously impossible we anticipate a typo there.

4 Term Rewriting

This section builds upon the result for ARSs from the previous section to prepare for confluence criteria for TRSs, such as Theorem 1. To support confluence results besides Theorem 1, in the formalization we did not follow the easiest way, i.e., suit the definitions and lemmas directly towards Theorem 1. Rather, we adopted the approach from [31], where all results are established via *labeling functions* (satisfying some abstract properties). Apart from avoiding a monolithic proof, this has the advantage that similar proofs need not be repeated for different labeling functions but it suffices to establish that the concrete labeling functions satisfy some abstract conditions. Then decreasingness is established in three steps. The first step comprises joinability results for local peaks (Section 4.1). The second step (Section 4.2) formulates abstract conditions with the help of *labeling functions* that admit a finite characterization of decreasingness of local peaks. Finally, based on the previous two steps, the third step (Section 5) then obtains confluence results by instantiating the abstract labeling functions with concrete ones, e.g. the rule labeling. So only the third step needs to be adapted when formalizing new labeling functions, as steps one and two are unaffected.

4.1 Local Peaks

As IsaFoR already supported Knuth-Bendix' criterion (see [21]), it contained results for joinability of local peaks and the critical pair theorem (the terms obtained by a local peak in a left-linear TRS are joinable or an instance of a critical pair). However, large parts of the existing formalization could not be reused directly as the established results lacked information required for ensuring decreasingness. For instance, to obtain decreasingness for the rule labeling (cf. Section 5) in case of a variable peak, the rewrite rules employed in the joining sequences are crucial, but the existing formalization only states that such a local peak is joinable. On the other hand, the existing notion of critical pairs from IsaFoR could be reused as the foundation for critical peaks. Since the computation of critical pairs requires a formalized unification algorithm, extending IsaFoR admitted focusing on the tasks related to decreasingness.

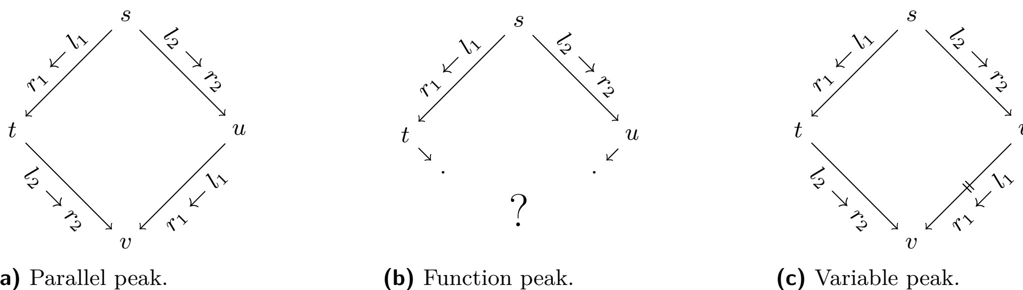
Local peaks can be characterized based on the positions of the diverging rewrite steps. Either the positions are parallel, called a parallel peak, or one position is above the other. In the latter situation we further distinguish whether the lower position is at a function position, called a function peak, or at/below a variable position of the other rule's left-hand side, called a variable peak. More precisely, for a local peak

$$t = s[r_1\sigma_1]_p \leftarrow s[l_1\sigma_1]_p = s = s[l_2\sigma_2]_q \rightarrow s[r_2\sigma_2]_q = u \tag{2}$$

there are three possibilities (modulo symmetry):

(a) $p \parallel q$ (parallel peak),
(b) $q \leqslant p$ and $p \backslash q \in \mathcal{P}\mathsf{os}_{\mathcal{F}}(l_2)$ (function peak),
(c) $q \leqslant p$ and $p \backslash q \notin \mathcal{P}\mathsf{os}_{\mathcal{F}}(l_2)$ (variable peak).

For the situation of a left-linear TRS these cases are visualized in Figure 2. It is easy to characterize parallel, function, and variable peaks in Isabelle (cf. Listing 1) but it requires tedious notation. The information of a rewrite step $s \rightarrow_{\mathcal{R}}^{p,l\rightarrow r,\sigma} t$ is represented in IsaFoR as `(s,t)` $\in$ `rstep_r_p_s` $\mathcal{R}$ `(l,r) p` σ. As the definition of function and variable peaks is asymmetric the five cases of local peaks can be reduced to the above three by mirroring those peaks. Then local peaks can be characterized as in Listing 2. Next we elaborate on the three cases.



(a) Parallel peak. **(b)** Function peak. **(c)** Variable peak.

Figure 2 Three kinds of local peaks.

```
definition local_peaks where "local_peaks R =
 {((s,rl1,p,σ1,t),(s,rl2,q,σ2,u)) | s t u rl1 rl2 p q σ1 σ2.
 ((s,t) ∈ rstep_r_p_s R rl1 p σ1 ∧ (s,u) ∈ rstep_r_p_s R rl2 q σ2)}"

definition parallel_peak where "parallel_peak R pk = (
 pk ∈ local_peaks R ∧ (let ((s,rl1,p,σ1,t),(s,rl2,q,σ2,u)) = pk in
 p ⊥ q))"

definition function_peak where "function_peak R pk = (
 pk ∈ local_peaks R ∧ (let ((s,rl1,p,σ1,t),(s,rl2,q,σ2,u)) = pk in
 ∃r.((p<#>r = q) ∧ r ∈ poss (fst rl1) ∧ is_Fun ((fst rl1) |_ r))))"

definition variable_peak where "variable_peak R pk = (
 pk ∈ local_peaks R ∧ (let ((s,rl1,p,σ1,t),(s,rl2,q,σ2,u)) = pk in
 ∃r.((p<#>r = q) ∧ ¬(r ∈ poss (fst rl1) ∧ is_Fun ((fst rl1) |_ r)))))"
```

Listing 1 Characterization of local peaks.

```
lemma local_peaks_cases:
  assumes "pk ∈ local_peaks R"
  shows "parallel_peak R pk ∨ variable_peak R pk ∨ function_peak R pk
   ∨ variable_peak R (snd pk, fst pk) ∨ function_peak R (snd pk, fst pk)"
```

Listing 2 Cases of local peaks.

Case 1: Parallel Peaks

Figure 2(a) shows the shape of a local peak where the steps take place at parallel positions. For a peak $t \;{}^{\pi_1}\!\!\leftarrow s \rightarrow^{\pi_2} u$ with $\pi_1 \parallel \pi_2$ we established that $t \rightarrow^{\pi_2} v \;{}^{\pi_1}\!\!\leftarrow u$, i.e., the steps drawn at opposing sides in the diagram are corresponding, that is, they apply the same rule/substitution at the same position. The proof is straightforward and based on a decomposition of the terms into a context and the redex.

Case 2: Function Peaks

In general joining function peaks may involve rules not present in the divergence (as indicated by the question mark in Figure 2(b)). To reduce the duty of joining (infinitely many) function

peaks to joining the (in case of a finite TRS finitely many) critical peaks, we established that every function peak is an instance of a critical peak.

▶ **Lemma 7.** *Let $t \;{}^{p,l_1\to r_1,\sigma_1}\!\!\leftarrow s \to^{q,l_2\to r_2,\sigma_2} u$ with $qq' = p$, and $q' \in \mathcal{P}\mathsf{os}_{\mathcal{F}}(l_2)$. Then there are a context C, a substitution τ, and a critical peak $l_2\mu[r_1\mu]_{q'} \leftarrow l_2\mu \to r_2\mu$ such that $s = C[l_2\mu\tau]$, $t = C[(l_2\mu[r_1\mu]_{q'})\tau]$, and $u = C[r_2\mu\tau]$.* ◀

We remark that this fact was already present (multiple times) in IsaFoR, but concealed in larger proofs, e.g. the formalization of orthogonality [12], and never stated explicitly.

As IsaFoR does not enforce that the variables of a rewrite rule's right-hand side are contained in its left-hand side, such rules are just also included in the critical peak computation.

Case 3: Variable Peaks

Variable overlaps (Figure 2(c)) can again be joined by the rules involved in the diverging step.[5] We only consider the case if $l_2 \to r_2$ is left-linear, as our main result assumes left-linearity. More precisely, if q' is the unique position in $\mathcal{P}\mathsf{os}_{\mathcal{V}}(l_2)$ such that $qq' \leqslant p$, $x = l_2|_{q'}$, and $|r_2|_x = n$ then we have $t \to_{l_2\to r_2} v$, which is similar to the case for parallel peaks, as the redex $l_2\sigma$ becomes $l_2\tau$ but is not destroyed, and $u \to^n_{l_1\to r_1} v$. To obtain this result we reason via parallel rewriting. The notion of parallel rewriting already supported by IsaFoR (employed to prove that orthogonal systems are confluent) does not keep track of e.g. the applied rules. Thus we augmented IsaFoR by a new version of parallel steps, which record the information (position, rewrite rule, substitution) of each rewrite step, i.e., the rewrite relation is decorated with the contracted redex patterns:

$$\frac{}{x \overset{\varnothing}{\twoheadrightarrow} x} \qquad \frac{l \to r \in \mathcal{R}}{l\sigma \xrightarrow{\{\langle \epsilon, l\to r, \sigma\rangle\}} r\sigma} \qquad \frac{s_1 \overset{P_1}{\twoheadrightarrow} t_1 \quad \cdots \quad s_n \overset{P_n}{\twoheadrightarrow} t_n}{f(s_1,\dots,s_n) \xrightarrow{(1P_1)\cup\cdots\cup(nP_n)} f(t_1,\dots,t_n)}$$

Here for a set of redex patterns $P = \{\pi_1,\dots,\pi_m\}$ by iP we denote $\{i\pi_1,\dots,i\pi_m\}$ with $i\pi = \langle ip, l \to r, \sigma\rangle$ for $\pi = \langle p, l \to r, \sigma\rangle$. To use this parallel rewrite relation for closing variable peaks we established the following auxiliary results.

▶ **Lemma 8.** *The following properties of the parallel rewrite relation hold:*
1. *For all s we have $s \twoheadrightarrow^{\varnothing} s$.*
2. *If $s \twoheadrightarrow^{\varnothing} t$ then $s = t$.*
3. *If $s \twoheadrightarrow^{P} t$ and $q \in \mathcal{P}\mathsf{os}(u)$ then $u[s]_q \twoheadrightarrow^{qP} u[t]_q$.*
4. *We have $s \to^{\pi} t$ if and only if $s \twoheadrightarrow^{\{\pi\}} t$.*
5. *If $\sigma(x) \to^{\pi} \tau(x)$ and $\sigma(y) = \tau(y)$ for all $y \in \mathcal{V}$ with $y \neq x$ then $t\sigma \twoheadrightarrow^{P} t\tau$ with $l_{\pi'} \to r_{\pi'} = l_\pi \to r_\pi$ for all $\pi' \in P$.*
6. *If $s \twoheadrightarrow^{\{\pi\}\cup P} t$ then there is a u with $s \twoheadrightarrow^{\{\pi\}} u \twoheadrightarrow^{P} t$.*
7. *If $s \twoheadrightarrow^{\{\pi_1,\dots,\pi_n\}} t$ then $s \to^{\pi_1} \cdots \to^{\pi_n} t$.*

Proof. In principle the results follow from the definitions using straightforward induction proofs. However, the additional bookkeeping, required to correctly propagate the information attached to the rewrite relation, makes them considerably more involved than for the existing, agnostic notion of parallel rewriting. ◀

Now for reasoning about variable peaks as above we decompose $u = s[r_2\sigma]_q$ and $v = s[r_2\tau]_q$ where $\sigma(y) = \tau(y)$ for all $y \in \mathcal{V} \setminus \{x\}$ and $\sigma(x) \to_{p\backslash qq', l_1\to r_1} \tau(x)$. From the latter by item (5)

[5] This includes rules having a variable as left-hand side.

```
inductive_set seq for R where
 "(s,[]) ∈ seq R" |
 "(s,t) ∈ rstep_r_p_s R rl p σ
   ⟹ (t,ts) ∈ seq R ⟹ (s,(s,rl,p,σ,t)#ts) ∈ seq R"
```

Listing 3 Rewrite sequences.

we obtain $r_2\sigma \twoheadrightarrow\!\!\!\!\!\!\not\;\;\,^P r_2\tau$, where all redex patterns in P use $l_1 \to r_1$. Then by item (3) we get $s[r_2\sigma]_q \twoheadrightarrow\!\!\!\!\!\!\not\;\;\,^{qP} s[r_2\tau]_q$ and finally $s[r_2\sigma]_q \to^n_{l_1\to r_1} s[r_2\tau]_q$ with $n = |qP| = |P|$ by item (7).

4.2 Local Decreasingness

The aim of this section is a confluence result (cf. Corollary 14) based on decreasingness of the critical peaks. Abstract conditions, via the key notion of a labeling, will ensure that parallel peaks and variable peaks are decreasing. Furthermore these conditions imply that decreasingness of the critical peaks implies decreasingness of the function peaks.

For establishing (extended) local decreasingness, a label must be attached to rewrite steps. To facilitate checking, the formalization makes the rewrite sequences (cf. Listing 3) explicit, i.e., they involve the intermediate terms, applied rules, etc. based on `rstep_r_p_s`. Furthermore, labels are computed by a *labeling (function)*, having (local) information about the rewrite step (such as source and target term, applied rewrite rule, position, and substitution) it is expected to label. For reasons of readability in this presentation we employ the mathematical notation (e.g., $\to^*$, etc.) with all information implicit but remark that the formalization works on rewrite sequences with explicit information (as in Listing 3).

▶ **Definition 9.** A *labeling* is a function ℓ from rewrite steps to a set of labels such that for all contexts C and substitutions σ the following properties are satisfied:

- If $\ell(s \to^{\pi_1} t) > \ell(u \to^{\pi_2} v)$ then $\ell(C[s\sigma] \to^{C[\pi_1\sigma]} C[t\sigma]) > \ell(C[u\sigma] \to^{C[\pi_2\sigma]} C[v\sigma])$
- If $\ell(s \to^{\pi_1} t) \geqslant \ell(u \to^{\pi_2} v)$ then $\ell(C[s\sigma] \to^{C[\pi_1\sigma]} C[t\sigma]) \geqslant \ell(C[u\sigma] \to^{C[\pi_2\sigma]} C[v\sigma])$

Here $C[\pi\sigma]$ denotes $\langle qp, l \to r, \tau\sigma\rangle$ for $\pi = \langle p, l \to r, \tau\rangle$ and $C|_q = \square$.

In presence of a labeling, rewrite sequences can be labeled at any time. This avoids lifting many notions (such as rewrite steps, local peaks, rewrite sequences, etc.) and results from rewriting to labeled rewriting.

Following [28], we separate (local) diagrams (where rewriting is involved) from decreasingness (where only the labels are involved). In the next definition a labeling is extended to peaks and rewrite sequences via the equations: $\ell(t\; {}^{\pi}\!\!\leftarrow s) = \ell(s \to^{\pi} t)$, $\ell(t \to^0 t) = \varnothing$, and $\ell(s \to^{\pi} t \to^* u) = \{\ell(s \to^{\pi} t)\} \cup \ell(t \to^* u)$.

▶ **Definition 10.** A local peak $t\; {}^{\pi_1}\!\!\leftarrow s \to^{\pi_2} u$ is *extended locally decreasing* (for ℓ) if it can be completed into a local diagram $t \to^* t' \to^= t'' \to^* v\; {}^*\!\!\leftarrow u''\; {}^=\!\!\leftarrow u'\; {}^*\!\!\leftarrow u$ such that its labels are extended locally decreasing, i.e.,
$\ell(t \to^* t') \subseteq \curlyvee\ell(t\; {}^{\pi_1}\!\!\leftarrow s)$, $\ell(t' \to^= t'') \subseteq \curlyvee\!\!\!\!/\,\ell(s \to^{\pi_2} u)$, $\ell(t'' \to^* v) \subseteq \curlyvee\ell(t\; {}^{\pi_1}\!\!\leftarrow s \to^{\pi_2} u)$ and $\ell(u'\; {}^*\!\!\leftarrow u) \subseteq \curlyvee\ell(s \to^{\pi_2} u)$, $\ell(u''\; {}^=\!\!\leftarrow u') \subseteq \curlyvee\!\!\!\!/\,\ell(t\; {}^{\pi_1}\!\!\leftarrow s)$, $\ell(v\; {}^*\!\!\leftarrow u'') \subseteq \curlyvee\ell(t\; {}^{\pi_1}\!\!\leftarrow s \to^{\pi_2} u)$.

The corresponding predicate in IsaFoR is given in Listing 4 where extended local decreasingness (`eld`) of a local peak `pk` is expressed via the existence of rewrite sequences `jl` and `jr` that join the divergence caused by the local peak `pk` in the shape of a local diagram (`ld_trs`) and the labels of the underlying rewrite sequences are extended locally decreasing (`eld_seq`). Here `r` is the pair of relations $(>, \geqslant)$.

```
definition eld where "eld R ℓ r pk =
 (∃ jl jr. (ld_trs R pk jl jr ∧ eld_seq ℓ r pk jl jr))"
```

Listing 4 Extended local decreasingness.

Then a function peak is extended locally decreasing if the critical peaks are.

▶ **Lemma 11.** *Let ℓ be a labeling and let all critical peaks of a TRS $\mathcal{R}$ be extended locally decreasing for ℓ. Then every function peak of $\mathcal{R}$ is extended locally decreasing for ℓ.*

Proof. As every function peak is an instance of a critical peak (see Lemma 7), the result follows from ℓ being a labeling (Definition 9). ◀

The notion of compatibility (between a TRS and a labeling) admits a finite characterization of extended local decreasingness.

▶ **Definition 12.** Let ℓ be a labeling. We call ℓ *compatible* with a TRS $\mathcal{R}$ if all parallel peaks and all variable peaks of $\mathcal{R}$ are extended locally decreasing for ℓ.

The key lemma then establishes that if ℓ is compatible with a TRS, then all local peaks are extended locally decreasing.

▶ **Lemma 13.** *Let ℓ be a labeling which is compatible with a TRS $\mathcal{R}$. If the critical peaks of $\mathcal{R}$ are extended locally decreasing for ℓ, then all local peaks of $\mathcal{R}$ are extended locally decreasing for ℓ.*

Proof. The cases of variable and parallel peaks are taken care of by compatibility. The case of function peaks follows from the assumption in connection with Lemma 11. The symmetric cases for function and variable peaks are resolved by mirroring the local diagrams. ◀

Representing a TRS $\mathcal{R}$ over the signature $\mathcal{F}$ and variables $\mathcal{V}$ as the ARS over objects $\mathcal{T}(\mathcal{F},\mathcal{V})$ and relations $\bigcup_\alpha \{(s,t) \mid s \to^\pi t \text{ and } \ell(s \to^\pi t) = \alpha\}$, Lemma 2 immediately applies to TRSs. To this end extended local decreasingness formulated via explicit rewrite sequences (with labeling functions) has to be mapped to extended local decreasingness on families of (abstract rewrite) relations; we omit the technical details here.

Finally, we obtain the following result.

▶ **Corollary 14.** *Let ℓ be a labeling compatible with a TRS $\mathcal{R}$. If the critical peaks of $\mathcal{R}$ are extended locally decreasing for ℓ then $\mathcal{R}$ is confluent.* ◀

Concrete confluence criteria are then obtained as instances of the above result. In the case of Theorem 1 by showing that the relative termination assumption in combination with the rule labeling implies the desired preconditions.

5 Applications

In this section we instantiate Corollary 14 to obtain concrete confluence results. Afterwards we discuss the design of the certificates, checkable by CeTA.

5.1 Rule Labeling

The rule labeling [16] is parametrized by an index mapping $i\colon \mathcal{R} \to \mathbb{N}$, which associates to every rewrite rule a natural number.

▶ **Definition 15.** The function $\ell^i(s \to^\pi t) = i(l_\pi \to r_\pi)$ is called *rule labeling*. Labels due to the rule labeling are compared by $>_\mathbb{N}$ and $\geqslant_\mathbb{N}$.

The rule labeling admits a confluence criterion based on the results established so far.

▶ **Lemma 16.**
1. *The rule labeling is a labeling.*
2. *Parallel peaks are extended locally decreasing for the rule labeling.*
3. *Variable peaks of a linear TRS are extended locally decreasing for the rule labeling.*
4. *The rule labeling is compatible with a linear TRS.*
5. *A linear TRS is confluent if its critical peaks are extended locally decreasing for the rule labeling.*

Proof. Item (1) follows from Definition 9. For (2) and (3) we employ the analysis of parallel and variable peaks from Section 4.1, respectively. Item (4) is then a consequence of (2) and (3). Finally, (5) amounts to an application of Corollary 14. ◀

Eventually, we remark that for the rule labeling extended local decreasingness implies local decreasingness, as $\geqslant_\mathbb{N}$ is the reflexive closure of $>_\mathbb{N}$.

5.2 Relative Termination

That a locally confluent terminating left-linear TRS is confluent can be established in the flavor of Lemma 16. The restriction to left-linearity arises from the lack of considering non-left-linear variable peaks in Section 4.1. As the analysis of such a peak would not give further insights we pursue another aim in this section, i.e., the mechanized proof of Theorem 1.

It is well known that the rule labeling ℓ^i is in general not compatible with left-linear TRSs, cf. [8]. Thus, to obtain extended local decreasingness for variable peaks the additional relative termination assumption is exploited. To this end we use the *source labeling*, which labels each rewrite step by its source, i.e., $\ell^{\mathsf{src}}(s \to^\pi t) = s$. Here, labels due to the source labeling are compared by the orders $\to^+_{\mathcal{R}_\mathsf{d}/\mathcal{R}_\mathsf{nd}}$ and $\to^*_\mathcal{R}$. The relative termination assumption of Theorem 1 makes all variable peaks of a left-linear TRS extended locally decreasing for the source labeling.

Following [31], the aim is to establish that the lexicographic combination $\ell^{\mathsf{src}} \times \ell^i$ is compatible with a left-linear TRS. To employ the rule labeling we have to introduce a weaker version of compatibility.

▶ **Definition 17.** A diagram of the shape $t\ {}_\alpha\!\leftarrow s \to^{l_2 \to r_2}_\beta u$, $t \to_{\vee\!\!/\beta} v\ {}^{\,n}_{\vee\!\!/\alpha}\!\leftarrow u$ is called *weakly extended locally decreasing* if $n \leqslant 1$ whenever r_2 is linear. We call a labeling ℓ *weakly compatible* with a TRS $\mathcal{R}$ if parallel and variable peaks are weakly extended locally decreasing for ℓ.

While weak extended local decreasingness could also be defined in the spirit of extended local decreasingness (with a more complicated join sequence) the chosen formulation eases the definition and simplifies proofs.

Based on the peak analysis of Section 4.1 the following results are established (Such properties must be proved for each labeling function.):

▶ **Lemma 18.** *Let $\mathcal{R}$ be a left-linear TRS.*
1. *Parallel peaks are weakly extended locally decreasing for the rule labeling.*
2. *Variable peaks of $\mathcal{R}$ are weakly extended locally decreasing for the rule labeling.*
3. *The rule labeling is weakly compatible with $\mathcal{R}$.* ◀

Similar results are established for the source labeling.

▶ **Lemma 19.** *Let $\mathcal{R}$ be a left-linear TRS whose duplicating rules terminate relative to the other rules.*
1. *The source labeling is a labeling.*
2. *Parallel peaks are extended locally decreasing for the source labeling.*
3. *Variable peaks of $\mathcal{R}$ are extended locally decreasing for the source labeling.*
4. *The source labeling is compatible with $\mathcal{R}$.* ◀

Using this lemma, we proved the following results for the lexicographic combination of the source labeling with another labeling.

▶ **Lemma 20.** *Let $\mathcal{R}$ be a left-linear TRS whose duplicating rules terminate relative to the other rules, and ℓ a labeling weakly compatible with $\mathcal{R}$.*
1. *Then $\ell^{\mathsf{src}} \times \ell$ is a labeling.*
2. *Then $\ell^{\mathsf{src}} \times \ell$ is compatible with $\mathcal{R}$.* ◀

For reasons of readability we have left the orders $>$ and $\geqslant$ that are required for (weak) compatibility implicit and just mention that the lexicographic extension (as detailed in [31]) preserves the required properties. Finally, we prove the main result of this paper.

Proof of Theorem 1. From Lemma 18(3) in combination with Lemma 20 we obtain that $\ell^{\mathsf{src}} \times \ell^i$ is a labeling compatible with a left-linear TRS, provided the relative termination assumption is satisfied. By assumption, the critical peaks are (extended locally) decreasing for the rule labeling ℓ^i. As along a rewrite sequence labels with respect to ℓ^{src} never increase, the critical peaks are extended locally decreasing for $\ell^{\mathsf{src}} \times \ell^i$. We conclude the proof by an application of Corollary 14. ◀

Hence, actually a stronger result than Theorem 1 has been mechanized, as $\ell^{\mathsf{src}} \times \ell^i$ might show more critical peaks decreasing than ℓ^i alone.

5.3 Certificates

Next we discuss the design of the certificates for confluence proofs via the rule labeling, i.e., how they are represented in CPF, and the executable checker to verify them. A minimal certificate could just claim that the considered rewrite system can be shown decreasing via the rule labeling. However, this is undecidable, even for locally confluent systems [8]. Hence in the certificate the index function i as well as (candidates for) the joining sequences for each critical pair have to be provided. Note that the labels in the joining sequences are not required for the certificate, since CeTA has to check, i.e., compute them anyway. The same holds for the critical peaks.

As the confluence tools that generate certificates might use different renamings than CeTA when computing critical pairs, the joining sequences given in the certificate are subject to a variable renaming. Thus, after computing all critical peaks, CeTA has to look for joining sequences in the certificate modulo renaming of variables.

Now, to verify whether the sequences of labels obtained from the joining sequences by applying the given index function fulfill the extended local decreasingness condition, we need

Table 1 Experimental results for 148 TRSs from CoCo 2014.

method	success	CoCo 2013	CoCo 2014	CeTA 2.19
(weak) orthogonality	4	✓	✓	✓
Knuth-Bendix	26	✓	✓	✓
strong closedness	28	✓	✓	✓
Lemma 16(5)	41	✗	✓	✓
Theorem 1	46	✗	✗	✓
$\sum$		45	56	58

to provide means to decide the following: given two natural numbers α and β and a sequence σ of natural numbers, is there a split $\sigma = \sigma_1\sigma_2\sigma_3$ such that $\sigma_1 \subseteq \vee\alpha$, $\sigma_2 \subseteq \vee\!\!/\beta$ with length of σ_2 at most one, and $\sigma_3 \subseteq \vee\alpha\beta$? To this end our checker employs a simple, greedy approach. That is, we pick the maximal prefix of σ with labels smaller α as σ_1. If the next label is less or equal to β we take it as σ_2 and otherwise we take the empty sequence for σ_2. Finally, the remainder of the sequence is σ_3. A straightforward case analysis shows that this approach is complete, i.e., otherwise no such split exists.

To certify applications of Theorem 1, additionally the relative termination condition has to be checked. Luckily, CeTA already supports a wide range of relative termination techniques, so that here we just needed to make use of existing machinery.

6 Experiments

For experiments we considered the 148 TRSs selected for CoCo 2014 and used the confluence tool CSI [30] to obtain certificates in CPF for confluence proofs. Note that ACP [2] can also produce certificates in CPF, but at the moment they are a subset of the ones reported by CSI. All generated certificates have been certified by CeTA. Note that CeTA can also certify various methods for non-confluence [12]. The largest certificate (for Cops #60) has 760 KB and lists 182 candidate joins for showing the 34 critical peaks decreasing. The certificate is checked within 1.1 seconds. We remark that no confluence tool besides CSI has solved Cops #60 so far, stressing the importance of a certified proof.

Next we elaborate on the impact of the new contributions. Experimental results for various criteria supported by CeTA are shown in Table 1.[6] The CeTA version from CoCo 2013 incorporated (weak) orthogonality [18], Knuth-Bendix' criterion [11], and strong closedness [9]. Due to the formalization described in this paper now also Theorem 1 is supported (column CeTA 2.19). As already employed for CoCo 2014, we included the data for Theorem 1 restricted to linear TRSs, i.e., Lemma 16(5). On our testbed Theorem 1 can establish confluence of more systems than all earlier methods together (46 vs. 45) and admits about 25% increase in power (58 vs. 45) when used in combination with the other criteria.

7 Conclusion

Finally we discuss related work, comment on the existing formalization of rewriting in IsaFoR, and conclude with a short summary.

[6] Details are available from `http://cl-informatik.uibk.ac.at/experiments/2015/rta3`.

7.1 Related Work

Formalizing confluence criteria has a long history in λ-calculus. Huet [10] proved a stronger variant of the parallel moves lemma in Coq. Isabelle/HOL was used in [13] to prove the Church-Rosser property of β, η, and $\beta\eta$. For β-reduction the standard Tait/Martin-Löf proof as well as Takahashi's proof [24] were formalized. The first mechanically verified proof of the Church-Rosser property of β-reduction was done using the Boyer-Moore theorem prover [20]. The formalization in Twelf [17] was used to formalize the confluence proof of a specific higher-order rewrite system in [23].

Next we discuss related work for term rewriting. Newman's lemma (for abstract rewrite systems) and Knuth and Bendix' critical pair theorem (for first-order rewrite systems) have been proved in [19] using ACL. An alternative proof of the latter in PVS, following the higher-order structure of Huet's proof, is presented in [6]. PVS is also used in the formalization of the lemmas of Newman and Yokouchi in [5]. Knuth and Bendix' criterion has also been formalized in Coq [4] and Isabelle/HOL [26]. The strong closedness condition of Huet [9] has been formalized by the first author in Isabelle [12] where reasoning similar to the one in Section 4.1 is used to (strongly) close variable and parallel peaks. However, for strong closedness it suffices to construct a common reduct while for our setting every rewrite step has to be made explicit in order to compute the labels and show local decreasingness.

7.2 Assessment

Next we discuss the usefulness of existing formalizations for this work. The existing machinery of IsaFoR admitted invaluable support. We regard our efforts to establish an annotated version of parallel rewriting not as a shortcoming of IsaFoR, but as a useful extension to it. On the contrary, we could employ many results from IsaFoR without further ado, e.g., completeness of the unification algorithm (to compute critical peaks), plain rewriting (to connect parallel steps with single steps), and the support for relative termination. Although [28] does not provide the main result for decreasing diagrams of abstract rewrite systems that are represented via families (as needed for Lemma 2), the amount of work to use this result has been modest, justifying the usability of [28]. That Lemma 7 occurred several times in IsaFoR can be explained as follows. Note that also in textbook proofs (e.g. [3]) this result is not made explicit but established in the scope of a larger proof, probably due to its nasty formulation. Still, in later proofs the result is used as if it would have been established explicitly. In IsaFoR these proofs have been duplicated, but as formalization papers typically come with code refactoring these deficiencies have been fixed. Note that the duplicated proofs have actually never been published.

Next, differences to [31] are addressed. The concepts of an L-labeling and an LL-labeling from [31] have been unified to the notion of a labeling *compatible* with a TRS while weak-LL-labelings are represented via *weakly compatible* labelings here. This admits the formulation of the abstract conditions such that a labeling ensures confluence (cf. Corollary 14) independent from the TRS being (left-)linear. We anticipate that the key result for closing variable peaks for the left-linear case (cf. Section 4.1) does not rely on the annotated version of parallel rewriting, but as [31] also supports labelings based on parallel rewriting the developed machinery might be useful for future certification efforts. The formalization described in this paper covers a significant amount of the results presented in [31]. As explained, additional concepts (e.g., the annotated version of parallel rewriting) were formalized to already prepare for the remaining criteria. However, for some results that are not covered yet (e.g. persistency), we anticipate that already formalizing the preliminaries requires significant effort.

7.3 Summary

In this paper we presented the formalization of a result establishing confluence of left-linear term rewrite systems based on relative termination and the rule labeling. While our formalization admits stronger results (in order to prepare for further results from [31]), we targeted Theorem 1, whose statement (in contrast to its proof) does not require the complex interplay of relative termination and the rule labeling, admitting the use of external termination provers. Our formalization also admits the (original) criterion for the rule labeling (cf. Lemma 16(5)). As this criterion applies to linear systems only, the involved analysis of non-right-linear variable peaks is not needed. The same holds for (the interplay with) the relative termination condition and the notion of extended local decreasingness (the rule labeling does not benefit from a preorder). Hence the proof of Theorem 1 is significantly more involved than the one of Lemma 16(5).

Acknowledgments. We thank Bertram Felgenhauer, Christian Sternagel, and René Thiemann for discussion and the reviewers for helpful comments.

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, 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.
3. F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.
4. E. Contejean, P. Courtieu, J. Forest, O. Pons, and X. Urbain. Automated certified proofs with CiME3. In *Proc. 22nd International Conference on Rewriting Techniques and Applications*, volume 10 of *Leibniz International Proceedings in Informatics*, pages 21–30, 2011.
5. A.L. Galdino and M. Ayala-Rincón. A formalization of Newman's and Yokouchi's lemmas in a higher-order language. *Journal of Formalized Reasoning*, 1(1):39–50, 2008.
6. A.L. Galdino and M. Ayala-Rincón. A formalization of the Knuth-Bendix(-Huet) critical pair theorem. *Journal of Automated Reasoning*, 45(3):301–325, 2010.
7. 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.
8. N. Hirokawa and A. Middeldorp. Decreasing diagrams and relative termination. *Journal of Automated Reasoning*, 47(4):481–501, 2011.
9. G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of the ACM*, 27(4):797–821, 1980.
10. G. Huet. Residual theory in lambda-calculus: A formal development. *Journal of Functional Programming*, 4(3):371–394, 1994.
11. 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.
12. J. Nagele and R. Thiemann. Certification of confluence proofs using CeTA. In *Proc. 3rd International Workshop on Confluence*, pages 19–23, 2014.
13. T. Nipkow. More Church-Rosser proofs. *Journal of Automated Reasoning*, 26(1):51–66, 2001.

14 T. Nipkow, L.C. Paulson, and M. Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

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

16 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.

17 F. Pfenning. A proof of the Church-Rosser theorem and its representation in a logical framework. Technical Report CMU-CS-92-186, School of Computer Science, Carnegie Mellon University, 1992.

18 B.K. Rosen. Tree-manipulating systems and Church-Rosser theorems. *Journal of the ACM*, 20(1):160–187, 1973.

19 J.-L. Ruiz-Reina, J.-A. Alonso, M.-J. Hidalgo, and F.-J. Martín-Mateos. Formal proofs about rewriting using ACL2. *Annals of Mathematics and Artificial Intelligence*, 36(3):239–262, 2002.

20 N. Shankar. A mechanical proof of the Church-Rosser theorem. *Journal of the ACM*, 35(3):475–522, 1988.

21 C. Sternagel and R. Thiemann. Formalizing Knuth-Bendix orders and Knuth-Bendix completion. In *Proc. 24th International Conference on Rewriting Techniques and Applications*, volume 21 of *Leibniz International Proceedings in Informatics*, pages 287–302, 2013.

22 C. Sternagel and R. Thiemann. The certification problem format. In *Proc. 11th International Workshop on User Interfaces for Theorem Provers*, volume 167 of *Electronic Proceedings in Theoretical Computer Science*, pages 61–72, 2014.

23 K. Støvring. Extending the extensional lambda calculus with surjective pairing is conservative. *Logical Methods in Computer Science*, 2(2:1):1–14, 2006.

24 M. Takahashi. Parallel reductions in λ-calculus. *Information and Computation*, 118(1):120–127, 1995.

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

26 R. Thiemann. Certification of confluence proofs using CeTA. In *Proc. 1st International Workshop on Confluence*, page 45, 2012.

27 R. Thiemann and C. Sternagel. Certification of termination proofs using CeTA. In *Proc. 22nd International Conference on Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 452–468, 2009.

28 H. Zankl. Confluence by decreasing diagrams – formalized. In *Proc. 24th International Conference on Rewriting Techniques and Applications*, volume 21 of *Leibniz International Proceedings in Informatics*, pages 352–367, 2013.

29 H. Zankl. Decreasing diagrams. *Archive of Formal Proofs*, November 2013. Formal proof development, `http://afp.sf.net/entries/Decreasing-Diagrams.shtml`.

30 H. Zankl, B. Felgenhauer, and A. Middeldorp. CSI – A confluence tool. In *Proc. 23rd International Conference on Automated Deduction*, volume 6803 of *Lecture Notes in Artificial Intelligence*, pages 499–505, 2011.

31 H. Zankl, B. Felgenhauer, and A. Middeldorp. Labelings for decreasing diagrams. *Journal of Automated Reasoning*, 54(2):101–133, 2015.

Transforming Cycle Rewriting into String Rewriting

David Sabel[1] and Hans Zantema[2,3]

1 Goethe University Frankfurt am Main, Institute for Computer Science
Frankfurt am Main, Germany
sabel@ki.informatik.uni-frankfurt.de

2 TU Eindhoven, Department of Computer Science,
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
h.zantema@tue.nl

3 Radboud University Nijmegen, Institute for Computing and Information Sciences,
P.O. Box 9010, 6500 GL Nijmegen, The Netherlands

Abstract

We present new techniques to prove termination of cycle rewriting, that is, string rewriting on cycles, which are strings in which the start and end are connected. Our main technique is to transform cycle rewriting into string rewriting and then apply state of the art techniques to prove termination of the string rewrite system. We present three such transformations, and prove for all of them that they are sound and complete. Apart from this transformational approach, we extend the use of matrix interpretations as was studied before. We present several experiments showing that often our new techniques succeed where earlier techniques fail.

1998 ACM Subject Classification F.4.2 Grammars and other rewriting systems

Keywords and phrases rewriting systems, string rewriting, termination

Digital Object Identifier 10.4230/LIPIcs.RTA.2015.285

1 Introduction

String rewriting can not only be applied on strings, but also on cycles. Cycles can be seen as strings of which the left end is connected to the right end, by which the string has no left end or right end any more. Applying string rewriting on cycles is briefly called cycle rewriting. Rewriting behavior is strongly influenced by allowing cycles, for instance, in string rewriting the single rule $ab \to ba$ is terminating, but in cycle rewriting it is not, since the string ab represents the same cycle as ba. From the viewpoint of graph transformation, cycle rewriting is very natural. For instance, in [2] it was shown that if all rules of a graph transformation system are string rewrite rules, termination of the transformation system coincides with termination of the cycle rewrite system, and not with termination of the string rewrite system.

In many areas cycle rewriting is more natural than string rewriting. For instance, the problem of 5 dining philosophers can be expressed as a cycle $FTFTFTFTFT$ where F denotes a fork, and T denotes a thinking philosopher. Writing L for a philosopher who has picked up her left fork, but not her right fork, and E for an eating philosopher, a classical (deadlocking) modeling of the dining philosophers problem (for arbitrary many philosophers) can be expressed by the cycle rewrite system consisting of the rules $FT \to L$, $LF \to E$, $E \to FTF$. As a cycle rewrite system this is clearly not terminating.

© David Sabel and Hans Zantema;
licensed under Creative Commons License CC-BY
26th International Conference on Rewriting Techniques and Applications (RTA'15).
Editor: Maribel Fernández; pp. 285–300
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

So both string rewriting and cycle rewriting provide natural semantics for string rewrite systems, also called semi-Thue systems. Historically, string rewriting got a lot of attention as being a particular case of term rewriting, while cycle rewriting hardly got any attention until recently.

In [18] a first investigation of termination of cycle rewriting was made. Some techniques were presented to prove cycle termination, implemented in a tool `torpacyc`. Further a transformation ϕ was given such that for every string rewriting system (SRS) R, string termination of R holds if and only if cycle termination of $\phi(R)$ holds. As a consequence, cycle termination is undecidable. However, for making use of the strong power of current tools for proving termination of string rewriting in order to prove cycle termination, we need a transformation the other way around: transformations ψ such that for every SRS R, cycle termination of R holds if and only if string termination of $\psi(R)$ holds. The 'if' direction in this 'if and only if' is called 'sound', the 'only if' is called complete. A new way to prove cycle termination of an SRS R is to apply a tool for proving termination of string rewriting to $\psi(R)$ for a sound transformation ψ. The main topic of this paper is to investigate such transformations, and to exploit them to prove termination of cycle rewriting.

A similar approach to exploit the power of tools for termination of term rewriting to prove a modified property was used before in [8, 7]. However, there the typical observation was that the complete transformations were complicated, and for non-trivial examples termination of $\psi(R)$ could not be proved by the tools, while for much simpler sound (but incomplete) transformations ψ, termination of $\psi(R)$ could often be proved by the tools. In our current setting this is different: we introduce a transformation split, for which we prove that it is sound and complete, but we show that for several systems R for which all approaches from [18] fail, cycle termination of R can be concluded from an automatic termination proof of split(R) generated by AProVE [6, 1] or $\mathsf{T_TT_2}$ [12, 15].

It can be shown that if strings of size n exist admitting cycle reductions in which for every rule the number of applications of that rule is more than linear in n, then all techniques from [18] fail to prove cycle termination. Nevertheless, in quite simple examples this may occur while cycle termination holds. As an example consider the following.

A number of people are in a circle, and each of them carries a number, represented in binary notation with a bounded number of bits. Each of them may increase his/her number by one, as long as it fits in the bounded number of bits. Apart from that, every person may increase the number of bits of the number of its right neighbor by two. In order to avoid trivial non-termination, the latter is only allowed if the leading bit of the number is 0, and the new leading bit is put to 1, and the other to 0, by which effectively one extra bit is added. We will prove that this process will always terminate by giving an SRS in which all of the above steps can be described by a number of cycle rewrite steps, and prove cycle termination.

In order to do so we write P for person, and 0 and 1 for the bits of the binary number. For carry handling we introduce an extra symbol c of which the meaning is a 1 with a carry. Assume for every person its number is stored left from it. So if the number ends in 0, by adding one this last bit 0 is replaced by 1, expressed by the rule $0P \to 1P$. In case the number ends in 1, a carry should be created, since c represents a 1 with a carry this is expressed by the rule $1P \to cP$. Next the carry should be processed. In case it is preceded by 0, this 0 should be replaced by 1, while the c is replaced by 0; this is expressed by the rule $0c \to 10$. In case it is preceded by 1, a new carry should be created while again the old carry is replaced by 0; this is expressed by the rule $1c \to c0$. In this way adding one to any number in binary notation can be expressed by a number of rewrite steps, as long as no overflow occurs. Finally, we have to add a rule to extend the bit size of the number of the

right neighbor: the leading bit should be 0, while it is replaced by 100: adding two extra bits of which the leading one is 1 and the other is 0. This is expressed by the rule $P0 \to P100$. Summarizing: we have to prove cycle termination of the SRS consisting of the five rules

$$0P \to 1P,\ 1P \to cP,\ 0c \to 10,\ 1c \to c0,\ P0 \to P100.$$

This is fundamentally impossible by the techniques presented in [18]: by one of the techniques the last rule can be removed, but starting in $0^n P$ a reduction can be made in which all of the remaining four rules are applied an exponential number of times, by which the techniques from [18] fail.

In this paper we give two ways to automatically prove that cycle termination holds for the above example R: $\mathsf{T_TT_2}$ succeeds in proving termination of $\mathsf{split}(R)$, and the other is a variant of matrix interpretations for which we show that it proves cycle termination.

The paper is organized as follows. Section 2 recalls the basics of cycle rewriting. The main section Section 3 presents three transformations split, rotate, and shift, and proves soundness and completeness of all of them. In Section 4 the matrix approach is revisited and extended. In Section 5 experiments on implementations of our techniques are reported. We conclude in Section 6.

2 Preliminaries

A *signature* Σ is a finite alphabet of symbols. With Σ^* we denote the set of strings over Σ. With ε we denote the empty string and for $u, v \in \Sigma^*$, we write uv for the concatenation of the strings u and v. With $|u|$ we denote the length of string $u \in \Sigma^*$ and for $a \in \Sigma$ and $n \in \mathbb{N}$, a^n denotes n replications of symbol a, i.e. $a^0 = \varepsilon$ and $a^i = aa^{i-1}$ for $i > 0$.

Given a binary relation $\to$, we write $\to^i$ for i steps, $\to^{\leq i}$ for at most i steps, $\to^{<i}$ for at most $i-1$ steps, $\to^*$ for the reflexive-transitive closure of $\to$, and $\to^+$ for the transitive closure of $\to_R$. For binary relations $\to_1$ and $\to_2$, we write $\to_1 . \to_2$ for the composition of $\to_1$ and $\to_2$, i.e. $a \to_1 . \to_2 c$ iff there exists a b s.t. $a \to_1 b$ and $b \to_2 c$.

A *string rewrite system* (SRS) is a finite set R of rules $\ell \to r$ where $\ell, r \in \Sigma^*$. The *rewrite relation* $\to_R \subseteq (\Sigma^* \times \Sigma^*)$ is defined as follows: if $w = u\ell v \in \Sigma^*$ and $(\ell \to r) \in R$, then $w \to_R urv$. The *prefix-rewrite relation* $\hookrightarrow_R$ is defined as: if $w = \ell u \in \Sigma^*$ and $(\ell \to r) \in R$, then $w \hookrightarrow_R ru$. The *suffix-rewrite relation* $\hookleftarrow_R$ is defined as: if $w = u\ell \in \Sigma^*$ and $(\ell \to r) \in R$, then $w \hookleftarrow_R ur$.

A (finite or infinite) sequence of rewrite steps $w_1 \to_R w_2 \to_R \cdots$ is called a *rewrite sequence* (sometimes also a *reduction* or a *derivation*). An SRS R is *non-terminating* if there exists a string $w \in \Sigma^*$ and an infinite rewrite sequence $w \to_R w_1 \to_R w_2 \cdots$. Otherwise, R is *terminating*.

We recall the notion of cycle rewriting from [18]. A string can be viewed as a cycle, i.e. the last symbol of the string is connected to the first symbol. To represent cycles by strings, we define the equivalence relation $\sim$ as follows:

$$u \sim v \text{ iff } u = w_1 w_2 \text{ and } v = w_2 w_1 \text{ for some strings } w_1, w_2 \in \Sigma^*$$

With $[u]$ we denote the equivalence class of string u w.r.t. $\sim$.

The *cycle rewrite relation* $\circ\!\!\to_R \subseteq (\Sigma/\!\sim \times \Sigma/\!\sim)$ of an SRS R is defined as

$$[u] \circ\!\!\to_R [v] \text{ iff } \exists w \in \Sigma^* : u \sim \ell w, (\ell \to r) \in R, \text{ and } v \sim rw$$

The cycle rewrite relation $\circ\!\!\to_R$ is called *non-terminating* iff there exists a string $w \in \Sigma^*$ and an infinite sequence $[w] \circ\!\!\to_R [w_1] \circ\!\!\to_R [w_2] \circ\!\!\to_R \cdots$. Otherwise, $\circ\!\!\to_R$ is called *terminating*.

We recall some known facts about cycle rewriting.

▶ **Proposition 1** (see [18]). *Let Σ be a signature, R be an SRS, and $u, v \in \Sigma^*$.*

1. *If $u \to_R v$ then $[u] \circ\!\!\to_R [v]$.*
2. *If $\circ\!\!\to_R$ is terminating, then $\to_R$ is terminating.*
3. *Termination of $\to_R$ does not necessarily imply termination of $\circ\!\!\to_R$.*
4. *Termination of $\circ\!\!\to_R$ is undecidable.*
5. *For every SRS R there exists a transformed SRS $\phi(R)$ s.t. the following three properties are equivalent:*
   - *$\to_R$ is terminating.*
   - *$\to_{\phi(R)}$ is terminating.*
   - *$\circ\!\!\to_{\phi(R)}$ is terminating.*

For an SRS R, the last property implies that termination of $\to_R$ can be proved by proving termination of the translated cycle rewrite relation $\circ\!\!\to_{\phi(R)}$. In [18] it was used to show that termination of cycle rewriting is undecidable and for further results on derivational complexity for cycle rewriting.

3 Transforming Cycle Termination into String Termination

Proposition 1 and the involved transformation ϕ, which transforms string rewriting into cycle rewriting, provide a method to prove string termination by proving cycle termination. However, it does not provide a method to prove termination of the cycle rewrite relation $\circ\!\!\to_R$ by proving termination of the string rewrite relations $\to_R$ or $\to_{\phi(R)}$. Hence, in this section we develop transformations ψ s.t. termination of $\to_{\psi(R)}$ implies termination of $\circ\!\!\to_R$. We call such a transformation ψ *sound*. However, there are "useless" sound transformations, for instance, transformations where $\psi(R)$ is always non-terminating. So at least one wants to find sound transformations which permit to prove termination of non-trivial cycle rewrite relations. However, a better transformation should fulfill the stronger property that $\to_{\psi(R)}$ is terminating if and only if $\circ\!\!\to_R$ is terminating. If termination of $\circ\!\!\to_R$ implies termination of $\to_{\psi(R)}$, then we say ψ is *complete*. For instance, for a complete transformation, non-termination proofs of $\to_{\psi(R)}$ also imply non-termination of $\circ\!\!\to_R$. Hence, our goal is to find sound and complete transformations ψ.

We will introduce and discuss three transformations split, rotate, and shift where the most important one is the transformation split, since it has the following properties: The transformation is sound and complete, and as our experimental results show, it behaves well in practice when proving termination of cycle rewriting. The other two transformations rotate and shift are also sound and complete, but rather complex and – as our experimental results show – they do not behave as well as the transformation split in practice. We include all three transformations in this paper to document some different approaches to transform cycle rewriting into string rewriting.

While we will analyze the transformation split in detail and prove its soundness and completeness, for the other two transformations we briefly list their properties where the corresponding proofs can be found in the longer version [13] of this paper.

3.1 The Transformation Split

The idea of the transformation split is to perform a single cycle rewrite step $[u] \circ\!\!\to_R [v]$ step which uses rule $(\ell \to r) \in R$, by either applying a string rewrite step $u \to_R v$ or by splitting the rule $(\ell \to r)$ into two rules $(\ell_A \to r_A)$ and $(\ell_B \to r_B)$, where $\ell = \ell_A \ell_B$ and $r = r_A r_B$. Then a cycle rewrite step can be simulated by a prefix and a subsequent suffix rewrite step:

first apply rule $\ell_B \to r_B$ to a prefix of u and then apply rule $\ell_A \to r_A$ to a suffix of the obtained string.

▶ **Example 2.** Let $R = \{abc \to bbbb\}$ and $[bcdda] \circ\!\!\to_R [bbddbb]$. The rule $abc \to bbbb$ can be split into the rules $a \to bb$ and $bc \to bb$ s.t. $bcdda \hookrightarrow_{\{bc\to bb\}} bbdda \hookleftarrow_{\{a\to bb\}} bbddbb$.

We describe the idea of the transformation split more formally. It uses the following observation of cycle rewriting: if $[u] \circ\!\!\to_R [v]$, then $u \sim \ell w$, $(\ell \to r) \in R$, and $v \sim rw$. From $u \sim \ell w$ follows that $u = u_1 u_2$ and $\ell w = u_2 u_1$ for some u_1, u_2. We consider the cases for u_2:

1. If $u_i = \varepsilon$ (for $i = 1$ or $i = 2$), then $u = \ell w$ and $u \hookrightarrow_R rw$ by a prefix string-rewrite step.
2. If ℓ is a prefix of u_2, i.e. $\ell u_2' = u_2$, then $w = u_2' u_1$, $u = u_1 \ell u_2' \to_R u_1 r u_2'$, and $u_1 r u_2' \sim rw$.
3. If u_2 is a proper prefix of ℓ, then there exist ℓ_A, ℓ_B with $\ell = \ell_A \ell_B$ s.t. $u_2 = \ell_A$ and ℓ_B is a true prefix of u_1, i.e. $u_1 = \ell_B w$ and $u = u_1 u_2 = \ell_B w \ell_A \hookrightarrow_{\{\ell_B \to r_B\}} r_B w \ell_A \hookleftarrow_{\{\ell_A \to r_A\}} r_B w r_A \sim rw$ if $r_A r_B = r$.

The three cases show that a cycle rewrite step $[u] \circ\!\!\to_{\{\ell\to r\}} [v]$ can either be performed by applying a string rewrite step $u \to_{\{\ell\to r\}} v'$ where $v' \sim v$ (cases 1 and 2) or in case 3 by splitting $\ell \to r$ into two rules $\ell_A \to r_A$ and $\ell_B \to r_B$ such that $u \hookrightarrow_{\{\ell_B\to r_B\}} u'$ replaces a prefix of u by r_B and $u' \hookleftarrow_{\{\ell_A\to r_A\}} v'$ replaces a suffix of u' by r_A s.t. $v' \sim v$.

For splitting a rule $(\ell \to r)$ into rules $\ell_A \to r_A$ and $\ell_B \to r_B$, we may choose any decomposition of r for r_A and r_B (s.t. $r = r_A r_B$). We will work with $r_A = r$ and $r_B = \varepsilon$.

The above cases for cycle rewriting show that a *sound* transformation of the cycle rewrite relation $\circ\!\!\to_R$ into a string rewrite relation is the SRS which consists of all rules of R and all pairs of rules $\ell_B \to \varepsilon$ and $\ell_A \to r$ for all $(\ell \to r) \in R$ and all ℓ_A, ℓ_B with $|\ell_A| > 0$, $|\ell_B| > 0$, and $\ell = \ell_A \ell_B$. However, this transformation does not ensure that the rules evolved by splitting are used as prefix and suffix rewrite steps only. Indeed, the transformation in this form is useless for nearly all cases, since whenever the right-hand side r of a rule $(\ell \to r) \in R$ contains a symbol $a \in \Sigma$ which is the first or the last symbol in ℓ, then the transformed SRS is non-terminating. For instance, for $R = \{aa \to aba\}$ the cycle rewrite relation $\circ\!\!\to_R$ is terminating, while the rule $a \to aba$ (which would be generated by splitting the left-hand side of the rule) leads to non-termination of the string rewrite relation. Note that this also holds if we choose any other decomposition of the right-hand side. Hence, in our transformation we introduce additional symbols to ensure:

- $\ell_B \to \varepsilon$ can only be applied to a prefix of the string.
- $\ell_A \to r$ can only be applied to a suffix of the string.
- If $\ell_B \to \varepsilon$ is applied to a prefix, then also $\ell_A \to r$ must be applied, in a synchronized manner (i.e. no other rule $\ell_B' \to \varepsilon$ or $\ell_A' \to r'$ can be applied in between).

In detail, we will prepend the fresh symbol B to the beginning of the string, and append the fresh symbol E to the end of the string. These symbols guarantee, that prefix rewrite steps $\ell u \hookrightarrow_{(\ell\to r)} ru$ can be expressed with usual string rewrite rules by replacing the left hand side ℓ with $\mathsf{B}\ell$ and analogous for suffix rewrite steps $u\ell \hookleftarrow_{(\ell\to r)} ur$ by replacing the left hand side ℓ with $\ell\mathsf{E}$. Let $(\ell_i \to r_i)$ be the ith rule of the SRS which is split into two rules $\ell_B \to \varepsilon$ and $\ell_A \to r_i$, where $\ell_A \ell_B = \ell_i$. After applying the rule $\ell_B \to \varepsilon$ to a prefix of the string, the symbol B will be replaced by the two fresh symbols W (for "wait") and $\mathsf{R}_{i,j}$ where i represents the ith rule and j means that ℓ_i has been split after j symbols (i.e. $|\ell_A| = j$). The fresh symbol L is used to signal that the suffix has been rewritten by rule $\ell_A \to r$. Finally, we use a copy of the alphabet, to ensure completeness of the transformation: for an alphabet Σ,

we denote by $\overline{\Sigma}$ a fresh copy of Σ, i.e. $\overline{\Sigma} = \{\overline{a} \mid a \in \Sigma\}$. For a word $w \in \Sigma^*$ with $\overline{w} \in \overline{\Sigma}^*$ we denote the word w where every symbol a is replaced by $\overline{a}$. Analogously, for a word $w \in \overline{\Sigma}^*$ with $\underline{w} \in \Sigma$ we denote w where every symbol $\overline{a}$ is replaced by the symbol a.

▶ **Definition 3** (The transformation split). Let $R = \{\ell_1 \to r_1, \ldots, \ell_n \to r_n\}$ be an SRS over alphabet Σ. Let $\overline{\Sigma}$ be a fresh copy of Σ and let $\mathsf{B}, \mathsf{E}, \mathsf{W}, \mathsf{R}_{i,j}, \mathsf{L}$ be fresh symbols (fresh for $\Sigma \cup \overline{\Sigma}$). The SRS $\mathsf{split}(R)$ over alphabet $\Sigma \cup \overline{\Sigma} \cup \{\mathsf{B}, \mathsf{E}, \mathbf{L}, \mathsf{W}\} \cup \bigcup_{i=1}^{n} \{\mathsf{R}_{i,j} \mid 1 \leq j < |\ell_i|\}$ consists of the following string rewrite rules:

$$\ell_i \to r_i \quad \text{for every rule } (\ell_i \to r_i) \in R \qquad \text{(splitA)}$$
$$\overline{a}\mathsf{L} \to \mathsf{L}a \quad \text{for all } a \in \Sigma \qquad \text{(splitB)}$$
$$\mathsf{WL} \to \mathsf{B} \qquad \text{(splitC)}$$

and for every rule $(\ell_i \to r_i) \in R$, for all $1 \leq j < |\ell_i|$ and $\ell_A \ell_B = \ell_i$ with $|\ell_A| = j$:

$$\mathsf{B}\ell_B \to \mathsf{W}\mathsf{R}_{i,j} \qquad \text{(splitD)}$$
$$\mathsf{R}_{i,j}\,\ell_A\,\mathsf{E} \to \mathsf{L}r_i\,\mathsf{E} \qquad \text{(splitE)}$$
$$\mathsf{R}_{i,j}\,a \to \overline{a}\mathsf{R}_{i,j} \quad \text{for all } a \in \Sigma \qquad \text{(splitF)}$$

We describe the intended use of the rules and the extra symbols. The symbols B and E mark the start and the end of the string, i.e. for a cycle $[u]$ the SRS $\mathsf{split}(R)$ rewrites $\mathsf{B}u\mathsf{E}$.

Let $[u] \circ\!\!\to_R [w]$. The rule (splitA) covers the case that also $u \to_R w$ holds. Now assume that for $w' \sim w$ we have $u \hookrightarrow_{\{\ell_B \to \varepsilon\}} v \hookleftarrow_{\{\ell_A \to r\}} w'$ (where $(\ell_A \ell_B \to r) \in R$). Rule (splitD) performs the prefix rewrite step and replaces B by W to ensure that no other such a rule can be applied. Additionally, the symbol $\mathsf{R}_{i,j}$ corresponding to the rule and its splitting is added to ensure that only the right suffix rewrite step is applicable. Rule (splitF) moves the symbol $\mathsf{R}_{i,j}$ to right and rule (splitE) performs the suffix rewrite step. Rules (splitB) and (splitC) are used to finish the simulation of the cycle rewrite step by using the symbol L to restore the original alphabet and to finally replace WL by B.

▶ **Example 4.** For $R_1 = \{aa \to aba\}$ the transformed string rewrite system $\mathsf{split}(R_1)$ is:

aa	$\to aba$	(splitA)	$\overline{a}\mathsf{L}$	$\to \mathsf{L}a$	(splitB)	$\overline{b}\mathsf{L}$	$\to \mathsf{L}b$	(splitB)
WL	$\to \mathsf{B}$	(splitC)	$\mathsf{B}a$	$\to \mathsf{W}\mathsf{R}_{1,1}$	(splitD)	$\mathsf{R}_{1,1}a\mathsf{E}$	$\to \mathsf{L}aba\mathsf{E}$	(splitE)
$\mathsf{R}_{1,1}a$	$\to \overline{a}\mathsf{R}_{1,1}$	(splitF)	$\mathsf{R}_{1,1}b$	$\to \overline{b}\mathsf{R}_{1,1}$	(splitF)			

For instance, the cycle rewrite step $[aba] \circ\!\!\to_{R_1} [baba]$ is simulated in the transformed system by $\mathsf{B}aba\mathsf{E} \to \mathsf{W}\mathsf{R}_{1,1}ba\mathsf{E} \to \mathsf{W}\overline{b}\mathsf{R}_{1,1}a\mathsf{E} \to \mathsf{W}\overline{b}\mathsf{L}aba\mathsf{E} \to \mathsf{WL}baba\mathsf{E} \to \mathsf{B}baba\mathsf{E}$. As a further example, for $R_2 = \{abc \to cbacba,\ aa \to a\}$ the transformed SRS $\mathsf{split}(R_2)$ is:

abc	$\to cbacba$	(splitA)	aa	$\to a$	(splitA)	WL	$\to \mathsf{B}$	(splitC)
$\overline{a}\mathsf{L}$	$\to \mathsf{L}a$	(splitB)	$\overline{b}\mathsf{L}$	$\to \mathsf{L}b$	(splitB)	$\overline{c}\mathsf{L}$	$\to \mathsf{L}c$	(splitB)
$\mathsf{B}bc$	$\to \mathsf{W}\mathsf{R}_{1,1}$	(splitD)	$\mathsf{B}c$	$\to \mathsf{W}\mathsf{R}_{1,2}$	(splitD)	$\mathsf{B}a$	$\to \mathsf{W}\mathsf{R}_{2,1}$	(splitD)
$\mathsf{R}_{1,1}a\mathsf{E}$	$\to \mathsf{L}cbacba\mathsf{E}$	(splitE)	$\mathsf{R}_{1,2}ab\mathsf{E}$	$\to \mathsf{L}cbacba\mathsf{E}$	(splitE)	$\mathsf{R}_{2,1}a\mathsf{E}$	$\to \mathsf{L}a\mathsf{E}$	(splitE)
$\mathsf{R}_{1,1}a$	$\to \overline{a}\mathsf{R}_{1,1}$	(splitF)	$\mathsf{R}_{1,2}a$	$\to \overline{a}\mathsf{R}_{1,2}$	(splitF)	$\mathsf{R}_{2,1}a$	$\to \overline{a}\mathsf{R}_{2,1}$	(splitF)
$\mathsf{R}_{1,1}b$	$\to \overline{b}\mathsf{R}_{1,1}$	(splitF)	$\mathsf{R}_{1,2}b$	$\to \overline{b}\mathsf{R}_{1,2}$	(splitF)	$\mathsf{R}_{2,1}b$	$\to \overline{b}\mathsf{R}_{2,1}$	(splitF)
$\mathsf{R}_{1,1}c$	$\to \overline{c}\mathsf{R}_{1,1}$	(splitF)	$\mathsf{R}_{1,2}c$	$\to \overline{c}\mathsf{R}_{1,2}$	(splitF)	$\mathsf{R}_{2,1}c$	$\to \overline{c}\mathsf{R}_{2,1}$	(splitF)

Termination of $\mathsf{split}(R_1)$ and $\mathsf{split}(R_2)$ can be proved by AProVE and $\mathsf{T_TT_2}$.

▶ **Proposition 5** (Soundness of split). *If $\to_{\mathsf{split}(R)}$ is terminating then $\circ\!\!\to_R$ is terminating.*

Proof. By construction of $\mathsf{split}(R)$, it holds that if $[u] \circ\!\!\to_R [v]$, then $\mathsf{B}u'\mathsf{E} \to^+_{\mathsf{split}(R)} \mathsf{B}v'\mathsf{E}$ with $u \sim u'$ and $v \sim v'$. Thus for every infinite sequence $[w_1] \circ\!\!\to_R [w_2] \circ\!\!\to_R \cdots$ there exists an infinite sequence $\mathsf{B}\ w'_1\ \mathsf{E} \to_{\mathsf{split}(R)} \mathsf{B}\ w'_2\ \mathsf{E} \to_{\mathsf{split}(R)} \cdots$ with $w_i \sim w'_i$ for all i. ◀

3.1.1 Completeness of Split

We use type introduction [17] and use the sorts A, $\overline{A}$, C, and T, and type the symbols used by $\mathsf{split}(R)$ (seen as unary function symbols, and seen as a term rewrite system) as follows:

$$\begin{array}{llll} \mathsf{L} :: A \to \overline{A} & \mathsf{W} :: \overline{A} \to T & a :: A \to A & \text{for all } a \in \Sigma \\ \mathsf{B} :: A \to T & \mathsf{E} :: C \to A & \overline{a} :: \overline{A} \to \overline{A} & \text{for all } \overline{a} \in \overline{\Sigma} \\ & & \mathsf{R}_{i,j} :: A \to \overline{A} & \text{for all } \mathsf{R}_{i,j} \end{array}$$

We also add a constant c of sort C to the alphabet, s.t. ground terms for any sort exist. First one can verify that all rewrite rules of $\mathsf{split}(R)$ are well-typed: (splitA) rewrites terms of sort A, (splitB), (splitE), and (splitF) rewrite terms of sort $\overline{A}$, and (splitC) and (splitD) rewrite terms of sort T. Since there are no collapsing and duplicating rules, type introduction can be used (see [17]), i.e. (string) termination of the typed system is equivalent to (string) termination of the untyped system.

We consider ground terms of the sorts A, $\overline{A}$, C, and T. For simplicity we use the representation as strings (instead of terms), and we write E instead of Ec. First we show that our analysis of non-termination can be restricted to terms of sort T:

▶ **Lemma 6.** *If a term w of sort S with $S \in \{A, \overline{A}, C\}$ admits an infinite reduction w.r.t. $\mathsf{split}(R)$, then there exists a term w' of sort T, which admits an infinite reduction.*

Proof. The only term of sort C is the constant c which is a normal form. If a term w of sort A admits an infinite reduction w.r.t. $\to_{\mathsf{split}(R)}$, then also the term $\mathsf{B}w$ of type T admits an infinite reduction w.r.t. $\to_{\mathsf{split}(R)}$. If a term w of sort $\overline{A}$ admits an infinite reduction w.r.t. $\to_{\mathsf{split}(R)}$, then also the term $\mathsf{W}w$ of type T admits an infinite reduction w.r.t. $\to_{\mathsf{split}(R)}$. ◀

Inspecting the typing of the symbols shows:

▶ **Lemma 7.** *Any term of sort T is of one of the following forms:*
- $\mathsf{B}u\mathsf{E}$ *where* $u \in \Sigma^*$
- $\mathsf{W}w\mathsf{L}u\mathsf{E}$ *where* $w \in \overline{\Sigma}^*$ *and* $u \in \Sigma^*$
- $\mathsf{W}w\mathsf{R}_{i,j}u\mathsf{E}$ *where* $w \in \overline{\Sigma}^*$ *and* $u \in \Sigma^*$

We define a mapping from terms of sort T into strings over Σ as follows:

▶ **Definition 8.** For a term $w :: T$, the string $\Phi(w) \in \Sigma^*$ is defined according to the cases of Lemma 7:

$$\begin{array}{lll} \Phi(\mathsf{B}u\mathsf{E}) & := & u \\ \Phi(\mathsf{W}w\mathsf{L}u\mathsf{E}) & := & \underline{w}u \\ \Phi(\mathsf{W}w\mathsf{R}_{i,j}u\mathsf{E}) & := & \ell_B\underline{w}u \quad \text{if } (\mathsf{B}\ell_B \to \mathsf{W}\mathsf{R}_{i,j}) \in \mathsf{split}(R) \end{array}$$

▶ **Lemma 9.** *Let w be of sort T and $w \to_{\mathsf{split}(R)} w'$. Then $\Phi(w) \circ\!\!\to_R^* \Phi(w')$.*

Proof. We inspect the cases of Lemma 7 for w:
- If $w = \mathsf{B}u\mathsf{E}$ where $u \in \Sigma^*$, then the step $w \to_{\mathsf{split}(R)} w'$ can use a rule of type (splitA) or (splitD). If rule (splitA) is applied, then $\Phi(w) \to_R \Phi(w')$ and thus $\Phi(w) \circ\!\!\to_R \Phi(w')$. If rule (splitD) is applied, then $w = \mathsf{B}\ell_2 u' \to_{\mathsf{split}(R)} \mathsf{W}\mathsf{R}_{i,j}u' = w'$ and $\Phi(w) = \ell_2 u' = \Phi(w')$.
- If $w = \mathsf{W}v\mathsf{L}u\mathsf{E}$ where $v \in \overline{\Sigma}^*$ and $u \in \Sigma^*$, then the step $w \to_{\mathsf{split}(R)} w'$ can use rules of type (splitA), (splitB), or (splitC). If rule (splitA) is used, then $\Phi(w) \to_R \Phi(w')$ and thus $\Phi(w) \circ\!\!\to_R \Phi(w')$. If rule (splitB) or (splitC) is used, then $\Phi(w) = \Phi(w')$.

- If $w = \mathsf{W}v\mathsf{R}_{i,j}u\mathsf{E}$ where $v \in \overline{\Sigma}^*$ and $u \in \Sigma^*$, then the step $w \to_{\mathsf{split}(R)} w'$ can use a rule of type (splitA), (splitE), or (splitF). If rule (splitA) is used, then $\Phi(w) \to_R \Phi(w')$ and thus $\Phi(w) \circ\!\!\to_R \Phi(w')$. If rule (splitF) is used, then $\Phi(w) = \Phi(w')$. If rule (splitE) is used, then $w = \mathsf{W}v\mathsf{R}_{i,j}\ell_A\mathsf{E}$ and $w' = \mathsf{W}v\mathsf{L}r_i\mathsf{E}$ and $\Phi(w) = \ell_B\underline{v}\ell_A \sim \ell_A\ell_B\underline{v} \to_R r_i\underline{v} \sim \underline{v}r_i = \Phi(w')$ and thus $\Phi(w) \circ\!\!\to_R \Phi(w')$. ◀

▶ **Theorem 10** (Soundness and completeness of split)**.** *The transformation* split *is sound and complete, i.e.* $\to_{\mathsf{split}(R)}$ *is terminating if, and only if* $\circ\!\!\to_R$ *is terminating.*

Proof. Soundness is proved in Proposition 5. It remains to show completeness. W.l.o.g. we assume that $\to_R$ is terminating, since otherwise $\circ\!\!\to_R$ is obviously non-terminating. Type introduction and Lemma 6 show that it is sufficient to construct a non-terminating cycle rewrite sequence for any term w of sort T where w has an infinite $\to_{\mathsf{split}(R)}$-reduction. For every infinite reduction $w \to_{\mathsf{split}(R)} w_1 \to_{\mathsf{split}(R)} w_2 \cdots$ we use Lemma 9 to construct a cycle rewrite sequence $\Phi(w) \circ\!\!\to_R^* \Phi(w_1) \circ\!\!\to_R^* \Phi(w_2) \cdots$. It remains to show that the constructed sequence is infinite: one can observe that the infinite sequence must have infinitely many applications of rule (splitE) (which is translated by $\Phi(\cdot)$ into exactly one $\circ\!\!\to_R$-step), since every sequence of $\xrightarrow{(\text{splitA})\vee(\text{splitB})\vee(\text{splitC})\vee(\text{splitD})\vee(\text{splitF})}$-steps is terminating (since we assumed that $\to_R$ is terminating). ◀

3.2 Alternative Transformations

We first present the general ideas of the transformations rotate and shift before giving their definitions. We write $\curvearrowright$ for the relation which moves the first element of a string to the end, i.e. $au \curvearrowright ua$ for every $a \in \Sigma$ and $u \in \Sigma^*$. We write $u \curvearrowright^{|\cdot|} v$ if, and only if $u \curvearrowright^{<|u|} v$ holds. Clearly, $u \sim v$ holds if and only if $u \curvearrowright^{<|u|} v$ holds. For a string rewrite system R, we define $\text{len}(R)$ as the size of the largest left-hand side of the rules in R, i.e. $\text{len}(R) = \max_{(\ell_i \to r_i) \in R} |\ell_i|$.

Using these notations we present two approaches to simulate cycle rewriting by string rewriting. One approach is to shift at most $\text{len}(R) - 1$ symbols from the left end to the right end and then to apply a string rewrite step (this is the relation $\curvearrowright^{<\text{len}(R)} . \to_R$). Another approach is to first rotate the string and then to apply a prefix rewrite step (this is the relation $\curvearrowright^{|\cdot|} . \hookrightarrow_R$).

▶ **Example 11.** As in Example 2, let $R = \{abc \to bbbb\}$ and $[bcdda] \circ\!\!\to_R [bbddbb]$.

For the first approach ("shift") we shift symbols from the left end to the right end of the string until abc becomes a substring. Then we apply a string rewrite step, i.e. $bcdda \curvearrowright^{<\text{len}(R)} . \to_R ddbbbb$, since $bcdda \curvearrowright cddab \curvearrowright ddabc \to_R ddbbbb$.

For the second approach ("rotate"), we rotate the string (by iteratively shifting symbols) until abc becomes a prefix. Then we apply a prefix rewrite step, i.e. $bcdda \curvearrowright^{|\cdot|} . \hookrightarrow_R bbbbdd$, since $bcdda \curvearrowright cddab \curvearrowright ddabc \curvearrowright dabcd \curvearrowright abcdd \hookrightarrow_R bbbbdd$.

It is quite easy to verify that the following proposition holds:

▶ **Proposition 12.** *Let R be an SRS. If $\circ\!\!\to_R$ is non-terminating, then*
1. $\curvearrowright^{<\text{len}(R)} . \to_R$ *admits an infinite reduction, and*
2. $\curvearrowright^{|\cdot|} . \hookrightarrow_R$ *admits an infinite reduction.*

For an SRS R the SRS shift(R) encodes the relation $\curvearrowright^{<\text{len}(R)} . \to_R$ and the SRS rotate(R) encodes the relation $\curvearrowright^{|\cdot|} . \hookrightarrow_R$ where extra symbols are used to separate the steps, and copies of the alphabet underlying R are used to ensure completeness of the transformations. We provide the definitions and some explanations of the transformations shift and rotate,

but for space reasons we exclude their proofs of soundness and completeness. They can be found in the longer version [13] of this paper. For the remainder of the section, we fix an SRS R over alphabet $\Sigma_A = \{a_1, \ldots, a_n\}$. Let us write $\Sigma_B, \Sigma_C, \Sigma_D, \Sigma_E$ for fresh copies of the alphabet Σ_A. We use the following notation to switch between the alphabets: for $X, Y \in \{A, B, C, D, E\}$ and $w \in \Sigma_X$ we write $\llparenthesis w \rrparenthesis_Y$ to denote the copy of w in the alphabet Y where every symbol is translated from alphabet X to alphabet Y.

▶ **Definition 13** (The transformation shift). Let R be an SRS over alphabet Σ_A and let $N = \max(0, \mathrm{len}(R) - 1)$. The SRS $\mathsf{shift}(R)$ over the alphabet $\Sigma_A \cup \Sigma_B \cup \Sigma_C \cup \{\mathsf{B}, \mathsf{E}, \mathsf{W}, \mathsf{V}, \mathsf{M}, \mathsf{L}, \mathsf{R}, \mathsf{D}\}$ (where $\mathsf{B}, \mathsf{E}, \mathsf{W}, \mathsf{V}, \mathsf{M}, \mathsf{L}, \mathsf{R}, \mathsf{D}$ are fresh for $\Sigma_A \cup \Sigma_B \cup \Sigma_C$) consists of the following rules:

$$
\begin{array}{rll}
\mathsf{B} \to \mathsf{W}\mathsf{M}^N\mathsf{V} & & \text{(shiftA)} \\
\mathsf{M} \to \varepsilon & & \text{(shiftB)} \\
\mathsf{M}\mathsf{V}a \to \mathsf{V}\llparenthesis a \rrparenthesis_B & \text{for all } a \in \Sigma_A & \text{(shiftC)} \\
ba \to ab & \text{for all } a \in \Sigma_A \text{ and all } b \in \Sigma_B & \text{(shiftD)} \\
b\mathsf{E} \to \llparenthesis b \rrparenthesis_A \mathsf{E} & \text{for all } b \in \Sigma_B & \text{(shiftE)} \\
\mathsf{W}\mathsf{V} \to \mathsf{R}\mathsf{L} & & \text{(shiftF)} \\
\mathsf{L}a \to \llparenthesis a \rrparenthesis_C \mathsf{L} & \text{for all } a \in \Sigma_A & \text{(shiftG)} \\
\mathsf{L}\ell \to \mathsf{D}r & \text{for all } (\ell \to r) \in R & \text{(shiftH)} \\
c\mathsf{D} \to \mathsf{D}\llparenthesis c \rrparenthesis_A & \text{for all } c \in \Sigma_C & \text{(shiftI)} \\
\mathsf{R}\mathsf{D} \to \mathsf{B} & & \text{(shiftJ)}
\end{array}
$$

The rules (shiftA) - (shiftE) encode the relation $\curvearrowright^{<\mathrm{len}(R)}$, i.e. for $uv \in \Sigma_A^*$ with $|u| < \mathrm{len}(R)$, the string $\mathsf{B}uv\mathsf{E}$ is rewritten into $\mathsf{W}\mathsf{V}vu\mathsf{E}$ by these five rules. The sequence of symbols M generated by rule (shiftA) denotes the potential of moving at most $\mathsf{len}(R) - 1$ symbols. The rules (shiftB) and (shiftC) either remove one from the potential or start the moving of one symbol. The rule (shiftD) performs the movement of a single symbol until it reaches the end of the string and rule (shiftE) finishes the movement.

The remaining rewrite rules perform a single string rewrite step, i.e. for a rule $(\ell \to r) \in R$ the string $\mathsf{W}\mathsf{V}w_1\ell w_2\mathsf{E}$ is rewritten to $\mathsf{B}w_1 r w_2\mathsf{E}$ by rules (shiftF) - (shiftJ).

This shows that if $u \curvearrowright^{<\mathrm{len}(R)} v \to_R w$, then $\mathsf{B}u\mathsf{E} \to^+_{\mathsf{shift}(R)} \mathsf{B}w\mathsf{E}$ and thus by Proposition 12 soundness of the transformation shift holds.

▶ **Definition 14** (The transformation rotate). Let R be an SRS over alphabet Σ_A. The SRS $\mathsf{rotate}(R)$ over the alphabet $\Sigma_A \cup \Sigma_B \cup \Sigma_C \cup \Sigma_D \cup \Sigma_E \cup \{$Begin, End, Rewrite, Goright, Guess, Rotate, Cut, Moveleft, Wait, Finish, Finish2$\}$ (where Begin, End, Rewrite, Goright, Guess, Rotate, Cut, Moveleft, Wait, Finish, and Finish2 are fresh for $\Sigma_A \cup \Sigma_B \cup \Sigma_C \cup \Sigma_D \cup \Sigma_E$) is:

$$
\begin{array}{rll}
\mathsf{Begin}\,\mathsf{End} \to \mathsf{Rewrite}\,\mathsf{End} & & \text{(rotateA)} \\
\mathsf{Begin}\,a \to \mathsf{Rotate}\,\mathsf{Cut}\,\llparenthesis a \rrparenthesis_D \mathsf{Guess} & \text{for all } a \in \Sigma_A & \text{(rotateB)} \\
\mathsf{Guess}\,a \to \llparenthesis a \rrparenthesis_D \mathsf{Guess} & \text{for all } a \in \Sigma_A & \text{(rotateC)} \\
\mathsf{Guess}\,a \to \mathsf{Moveleft}\,\llparenthesis a \rrparenthesis_C \mathsf{Wait} & \text{for all } a \in \Sigma_A & \text{(rotateD)} \\
\mathsf{Guess}\,\mathsf{End} \to \mathsf{Finish}\,\mathsf{End} & & \text{(rotateE)} \\
d\,\mathsf{Moveleft}\,c \to \mathsf{Moveleft}\,c\,\llparenthesis d \rrparenthesis_B & \text{for all } c \in \Sigma_c \text{ and all } d \in \Sigma_D & \text{(rotateF)} \\
\mathsf{Cut}\,\mathsf{Moveleft}\,c \to \llparenthesis c \rrparenthesis_E \mathsf{Cut}\,\mathsf{Goright} & \text{for all } c \in \Sigma_C & \text{(rotateG)} \\
\mathsf{Goright}\,b \to \llparenthesis b \rrparenthesis_D \mathsf{Goright} & \text{for all } b \in \Sigma_B & \text{(rotateH)} \\
\mathsf{Goright}\,\mathsf{Wait}\,a \to \mathsf{Moveleft}\,\llparenthesis a \rrparenthesis_C \mathsf{Wait} & \text{for all } a \in \Sigma_A & \text{(rotateI)} \\
\mathsf{Goright}\,\mathsf{Wait}\,\mathsf{End} \to \mathsf{Finish}\,\mathsf{End} & & \text{(rotateJ)} \\
d\,\mathsf{Finish} \to \mathsf{Finish}\,\llparenthesis d \rrparenthesis_A & \text{for all } d \in \Sigma_D & \text{(rotateK)}
\end{array}
$$

$$\begin{aligned}
\mathsf{Cut}\,\mathsf{Finish} &\to \mathsf{Finish2} && && \text{(rotateL)}\\
e\,\mathsf{Finish2} &\to \mathsf{Finish2}\,(\!|e|\!)_A && \text{for all } e \in \Sigma_E && \text{(rotateM)}\\
\mathsf{Rotate}\,\mathsf{Finish2} &\to \mathsf{Rewrite} && && \text{(rotateN)}\\
\mathsf{Rewrite}\,\ell &\to \mathsf{Begin}\,r && \text{for all } (\ell \to r) \in R && \text{(rotateO)}
\end{aligned}$$

We describe the intended use of the rewrite rules, where we ignore the copies of the alphabet in our explanations. The goal is that for any string $w \in \Sigma_A^*$, the string $\mathsf{Begin}\,w\,\mathsf{End}$ is rewritten to $\mathsf{Begin}\,u\,\mathsf{End}$, where $w \curvearrowright^{|\cdot|} . \hookrightarrow_R u$. The prefix rewrite step is performed by the last rule (rotateO). All other rules perform the rotation $\curvearrowright^{|\cdot|}$ s.t. $\mathsf{Begin}\,w\,\mathsf{End}$ is rewritten into $\mathsf{Rewrite}\,v\,\mathsf{End}$ where $w \sim v$. Instead of moving symbols from the front to the end (as $\curvearrowright$ does), the rules move a suffix of the string in front of the string (which has the same effect).

The first rewrite rule (rotateA) covers the case that w is empty. If $w = a_1 \dots a_n$, then first choose a position to cut the string into two parts $w_1 w_2$. The symbol Guess is used for the non-deterministic selection of the position. Rule (rotateB) starts the rotate phase and the guessing, rule (rotateC) shifts the Guess-marker and rule (rotateD) stops the guessing. Rule (rotateE) covers the case that $w_2 = \varepsilon$ and no rotation will be performed. After stopping the guessing, every symbol of w_2 is moved in front of w_1, resulting in $w_2 w_1$. A typical situation is $a_{k+1} \dots a_m a_1 \dots a_k a_{m+1} \dots a_n$ and now the symbol a_{m+1} must be moved in between a_m and a_1. To keep track of the position of a_1, the symbol Cut (inserted in front of a_1) marks the original beginning, and to keep track of the position of a_k, the symbol Wait (inserted after a_k) marks this position. The symbol $\mathsf{Moveleft}$ guards the movement of a_{m+1} (by rule (rotateF)). When arriving at the right place (rule (rotateG)), the symbol $\mathsf{Goright}$ is used to walk along the string (rule (rotateH)) to find the next symbol which has to be moved (rule (rotateI)). If all symbols are moved, rule (rotateJ) is applied to start the clean-up phase. There the symbols Finish and $\mathsf{Finish2}$ are used to remove the markers and to replace the copied symbols of the alphabet with the original ones (rules (rotateK) – (rotateN)).

The construction of $\mathsf{rotate}(R)$ shows that $\mathsf{Begin}\,u\,\mathsf{End} \to^*_{\mathsf{rotate}(R)} \mathsf{Begin}\,w\,\mathsf{End}$ whenever $u \curvearrowright^{|\cdot|} v \hookrightarrow_R w$. Thus Proposition 12 implies soundness of rotate.

In the long version [13] of this paper, we also prove completeness of both transformations shift and rotate and thus the following theorem holds:

▶ **Theorem 15** (see [13]). *The transformations* shift *and* rotate *are sound and complete.*

4 Trace Decreasing Matrix Interpretations

In this section we present a variant of matrix interpretations suitable for proving cycle termination. The basics of matrix interpretations for string and term rewriting were presented in [9, 10, 5, 11]. The special case of tropical and arctic matrix interpretations for cycle rewriting was presented in [18], in the setting of *type graphs*. This section extends matrix interpretations for cycle rewriting along the lines suggested by Johannes Waldmann.

Fix a dimension $d > 0$. Define M_d to be the set of $d \times d$ matrices A over $\mathbb{N}$ for which $A_{11} > 0$. On M_d we define the relations $>$ and $\geq$ by

$$A > B \iff A_{11} > B_{11} \wedge \forall i,j : A_{ij} \geq B_{ij}, \qquad A \geq B \iff \forall i,j : A_{ij} \geq B_{ij}.$$

Write $\times$ for matrix multiplication. Note that $(A \times B) \in M_d$ whenever $A, B \in M_d$. The following lemma is easily checked.

▶ **Lemma 16.** *Let* $A, B, C \in M_d$.

- *If* $A > B$ *then* $A \times C > B \times C$ *and* $C \times A > C \times B$,

- *If $A \geq B$ then $A \times C \geq B \times C$ and $C \times A \geq C \times B$.*

A *matrix interpretation* $\langle\cdot\rangle$ for a signature Σ is defined to be a mapping from Σ to M_d. It is extended to $\langle\cdot\rangle : \Sigma^* \to M_d$ by defining inductively $\langle\varepsilon\rangle = I$ and $\langle ua\rangle = \langle u\rangle \times \langle a\rangle$ for all $u \in \Sigma^*$, $a \in \Sigma$, where I is the identity matrix.

▶ **Theorem 17.** *Let $R' \subseteq R$ be SRSs over Σ and let $\langle\cdot\rangle : \Sigma \to M_d$ such that*

- *$\circ\!\!\to_{R'}$ is terminating,*
- *$\langle\ell\rangle \geq \langle r\rangle$ for all $(\ell \to r) \in R'$, and*
- *$\langle\ell\rangle > \langle r\rangle$ for all $(\ell \to r) \in R \setminus R'$.*

Then $\circ\!\!\to_R$ is terminating.

Proof. For a square matrix A of dimension d, its *trace* $\mathsf{tr}(A)$ is defined to be $\sum_{i=1}^{d} A_{ii}$: the sum of its diagonal. It is well known and easy to check that $\mathsf{tr}(A \times B) = \mathsf{tr}(B \times A)$ for all A, B. As a consequence we obtain $\mathsf{tr}(\langle u\rangle) = \mathsf{tr}(\langle v\rangle)$ for u, v satisfying $u \sim v$. Since $A > B$ implies $\mathsf{tr}(A) > \mathsf{tr}(B)$ and $A \geq B$ implies $\mathsf{tr}(A) \geq \mathsf{tr}(B)$, from Lemma 16 we obtain $\mathsf{tr}(\langle u\rangle) \geq \mathsf{tr}(\langle v\rangle)$ if $u \to_{R'} v$, and $\mathsf{tr}(\langle u\rangle) > \mathsf{tr}(\langle v\rangle)$ if $u \to_{R\setminus R'} v$. Combining these observations yields $\mathsf{tr}(\langle u\rangle) \geq \mathsf{tr}(\langle v\rangle)$ if $[u] \circ\!\!\to_{R'} [v]$, and $\mathsf{tr}(\langle u\rangle) > \mathsf{tr}(\langle v\rangle)$ if $[u] \circ\!\!\to_{R\setminus R'} [v]$.

Assume an infinite $\circ\!\!\to_R$ reduction exists: $[u_1] \circ\!\!\to_R [u_2] \circ\!\!\to_R [u_3] \circ\!\!\to_R [u_4] \circ\!\!\to_R \cdots$. Since $\circ\!\!\to_{R'}$ is terminating, it contains infinitely many steps $[u_i] \circ\!\!\to_{R\setminus R'} [u_{i+1}]$, all giving rise to $\mathsf{tr}(\langle u_i\rangle) > \mathsf{tr}(\langle u_{i+1}\rangle)$, while all other steps give rise to $\mathsf{tr}(\langle u_i\rangle) \geq \mathsf{tr}(\langle u_{i+1}\rangle)$. As $\mathsf{tr}(\langle\cdot\rangle)$ always yields a natural number, this yields an infinite descending sequence of natural numbers, contradiction. ◀

Although this proof is not very hard, it is quite subtle: it is essential to first use the full order on matrices and disallow A_{11} to be 0 in order to obtain Lemma 16, and next apply the trace to get the same interpretations for u and v if $u \sim v$. It is not essential to apply this approach to natural numbers with usual addition and multiplication: other well-founded semirings can be used as well. Using the semiring $\mathbb{N} \cup \{\infty\}$ with minimum as semiring addition and + as semiring multiplication yields *tropical matrix interpretations*. Using the semiring $\mathbb{N} \cup \{-\infty\}$ with maximum as semiring addition and + as semiring multiplication yields *arctic matrix interpretations*. For general theory on matrix interpretations for term rewriting we refer to [10, 5]. Validity of tropical and arctic matrix interpretations for cycle rewriting has been proved in [18], in the setting of *type graphs*.

The original versions of matrix interpretations in [9, 5] fail for proving cycle termination since they succeed in proving termination of $aa \to bc$, $bb \to ac$, $cc \to ab$, for which cycle termination does not hold due to $[ccaa] \circ\!\!\to [abaa] \circ\!\!\to [abbc] \circ\!\!\to [aacc]$. The main difference is that in our setting the interpretation of symbols is multiplication by a matrix, while in [9, 5] it combines such a matrix multiplication by adding a vector.

The search for an application of Theorem 17 has been implemented in our tool `torpacyc`, extending the version presented in [18]. It is done by transforming the requirements to SMT format and calling the external SMT solver `Yices` [3, 16]. As an example consider the system from the introduction:

$$0P \to 1P,\ 1P \to cP,\ 0c \to 10,\ 1c \to c0,\ P0 \to P100.$$

First by finding a tropical matrix interpretation, the last rule is removed by `torpacyc`. Next the following matrices are found:

$$\langle P\rangle = \begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix},\ \langle 0\rangle = \begin{pmatrix} 1 & 2 \\ 0 & 2 \end{pmatrix},\ \langle 1\rangle = \begin{pmatrix} 1 & 1 \\ 0 & 2 \end{pmatrix},\ \langle c\rangle = \begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix}.$$

For these interpretations we obtain $\langle 0P\rangle > \langle 1P\rangle$, $\langle 1P\rangle > \langle cP\rangle$, $\langle 0c\rangle \geq \langle 10\rangle$ and $\langle 1c\rangle \geq \langle c0\rangle$, hence by Theorem 17 it suffices to prove cycle termination of $0c \to 10,\ 1c \to c0$, for which `torpacyc` finds a simple counting argument.

4.1 Limitations of the Method

In this section we give an example where the matrix approach fails and the transformational approach succeeds.

The method for proving cycle termination induced by Theorem 17 has similar limitations as the method of matrix interpretations in [10] for string termination: Since the entries of a product of n matrices are bounded by an exponential function in n, the method cannot prove cycle termination of systems which allow reduction sequences where every rewrite rule is applied more often than exponentially often.

▶ **Example 18.** The rewrite system $R_1 := \{ab \to bca, cb \to bbc\}$ allows for string derivations of a length which is a tower of exponentials (see [10]), i.e. the string $a^k b^k$ has such a long derivation, since the derivation $ab^n \to^*_{R_1} b^{2^n-1}c^n a$ exists and this can be iterated for every a in a^k. Moreover, the number of applications of the first and of the second rule of R_1 is a tower of exponentials. This shows that the matrix interpretations in [10] are unable to prove string termination of R_1. The system $\phi(R_1) := \{RE \to LE, aL \to La', bL \to Lb', cL \to Lc', Ra' \to aR, Rb' \to bR, Rc' \to cR, abL \to bcaR, cbL \to bbcR\}$ uses the transformation ϕ from [18] and transforms the string rewrite system R_1 into a cycle rewrite system s.t. R_1 is string terminating iff $\phi(R_1)$ is cycle terminating. One can verify that $[ab^n LE] \circ\!\!\to^*_{\phi(R_1)} [b^{2^n-1}c^n aLE]$ which can also be iterated s.t. $[a^k b^k LE]$ has a cycle rewriting sequence whose length is a tower of k exponentials. Inspecting all nine rules of $\phi(R_1)$, the number of applications of any of the rules in this rewrite sequence is also a tower of k exponentials and thus it is impossible to prove cycle termination of $\phi(R_1)$ using Theorem 17. Consequently, our tool `torpacyc` does not find a termination proof for $\phi(R_1)$.

▶ **Remark 19.** As expected our tool `torpacyc` does not find a termination proof for $\phi(R_1)$ from Example 18. On the other hand, with our transformational approach a cycle termination can be proved: AProVE proves strings termination of $\mathsf{split}(\phi(R_1))$.

A further question is whether matrix interpretations are limited to cycle rewrite systems with exponential derivation lengths only. The following example shows that this is not true:

▶ **Example 20.** The SRS $R_2 := \{ab \to baa, cb \to bbc\}$ (see [10]) has derivations of doubly exponential length (since $ac^k b \to^*_{R_2} b^{2^k} a^{2^{2^k}} c^k$ and any rewrite step adds one symbol), but its string termination can be proved by relative termination and matrix interpretations by first removing the rule $cb \to bbc$ and then removing the other rule. This is possible, since the second rule is applied only exponentially often. For cycle rewriting the encoding ϕ from [18] is $\phi(R_2) = \{RE \to LE, aL \to La', bL \to Lb', cL \to Lc', Ra' \to aR, Rb' \to bR, Rc' \to cR, abL \to baaR, cbL \to bbcR\}$ and $\phi(R_2)$ is cycle terminating iff R_2 is string terminating. The system $\phi(R_2)$ also has doubly exponential cycle derivations, e.g. $[ac^k bLE] \circ\!\!\to^*_{\phi(R_2)} [b^{2^k} a^{2^{2^k}} c^k LE]$. However, `torpacyc` proves cycle termination of $\phi(R_2)$ by first removing the last rule using the matrix interpretation

$$\langle R\rangle = \begin{pmatrix} 1 & 2 \\ 1 & 0 \end{pmatrix}, \langle E\rangle = \begin{pmatrix} 2 & 0 \\ 0 & 0 \end{pmatrix}, \langle L\rangle = \begin{pmatrix} 1 & 2 \\ 1 & 0 \end{pmatrix}, \langle a\rangle = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \langle a'\rangle = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix},$$
$$\langle b\rangle = \begin{pmatrix} 1 & 2 \\ 0 & 1 \end{pmatrix}, \langle b'\rangle = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}, \langle c\rangle = \begin{pmatrix} 3 & 0 \\ 0 & 1 \end{pmatrix}, \langle c'\rangle = \begin{pmatrix} 1 & 0 \\ 1 & 3 \end{pmatrix}.$$

Thereafter the remaining rules (which now only have derivations of exponential length) are eliminated by matrix interpretations and counting arguments.

Note that the methods in [18] are not able to prove cycle termination of $\phi(R_2)$, since they can only remove rules which are applied polynomially often in any derivation.

Also the transformational approach successfully proves cycle termination of $\phi(R_2)$: $\mathsf{T_TT_2}$ proves string termination of $\mathsf{split}(\phi(R_2))$. Interestingly, we did not find a termination proof of $\mathsf{split}(\phi(R_2))$ using AProVE.

5 Tools and Experimental Results

We implemented the search for matrix interpretations described in Section 4 in `torpacyc`. We also implemented a tool which transforms an SRS by split, rotate, or shift. We also automatized the proof of cycle termination by a command line tool which can either use `torpacyc` to prove cycle termination or by first performing one of the transformations, and then using one of the termination provers AProVE or $\mathsf{T_TT_2}$ to prove string termination of the transformed problem. Also two kinds of combinations of both approaches are implemented:

- variant 1: first `torpacyc` tries to find a termination proof and if no proof is found, the (perhaps simplified) cycle rewrite system is transformed into a string rewrite system by split and then used as input for AProVE or $\mathsf{T_TT_2}$.
- variant 2: first string non-termination of the rewrite system is checked by AProVE or $\mathsf{T_TT_2}$. If a non-termination proof is found, then also the cycle rewrite system is non-terminating. Otherwise, the same procedure as in variant 1 is used.

The automatic transformation and the prover can also be used online via a web interface available via `http://www.ki.informatik.uni-frankfurt.de/research/cycsrs/` where also the tools and experimental data can be found.

5.1 Experiments

A problem for doing experiments is that no real appropriate test set is available. We played around with several examples like the ones in this paper, but we were also interested in larger scale experiments. To that end we chose two problem sets. The first set is the `SRS_Standard`-branch of the Termination Problem Data Base [14], which is a benchmark set for proving termination of string rewrite systems. The second set consists of 50 000 randomly generated cycle rewrite systems of size 12 over an alphabet of size 3. For all problems we tried to prove cycle termination using the following methods (all with a time limit of 60 seconds):

- `torpacyc`: We applied `torpacyc` to the problems, where version 2014 is described in [18] and version 2015 includes the matrix interpretations described in Section 4.
- split, rotate, shift: We transformed the problem by the transformation into a string termination problem and then applied the termination provers AProVE or $\mathsf{T_TT_2}$, respectively.
- combinations: We also tried the combination of the methods as described before (using AProVE and $\mathsf{T_TT_2}$) where for variant 1, `torpacyc` as well as the second termination prover had a time limit of 30 seconds, and for variant 2, both – the non-termination check and `torpacyc` – had a time limit of 15 seconds, and the termination prover applied to the transformed problem (i.e. AProVE or $\mathsf{T_TT_2}$) had a time limit of 30 seconds.

Table 1 Results of proving cycle termination of the problems in SRS_Standard of the TPDB.

	torpacyc		split		rotate		shift		combination variant 1		combination variant 2			string termination	
	2014	2015	AProVE	TTT2	AProVE	TTT2	AProVE	TTT2	AProVE	TTT2	AProVE	TTT2	any	AProVE	TTT2
yes	36	46	40	30	10	6	10	8	55	55	54	54	**63**	652	627
no	0	0	309	168	45	0	65	0	310	161	335	173	**336**	97	38
maybe	1289	1269	966	1117	1260	1309	1240	1307	950	1099	926	1088	**916**	566	650

5.1.1 String Rewrite Systems of the Termination Problem Data Base

We tested all 1315 string rewrite problems of the `SRS_Standard`-branch of the Termination Problem Data Base [14]. Table 1 shows the summary of the performed tests, where in the column titled with“any” the output of all tools are combined (per problem). The last two columns show the results of proving string termination of the original problems in our test environment using AProVE and TTT2. Note that a non-termination proof of string rewriting also implies non-termination of cycle rewriting, which does not hold for termination proofs. The results show that having sound and complete transformations is advantageous (compared to having sound transformations only), since we were able to prove cycle non-termination for 336 problems by the transformational approach (and by using string non-termination). Note that `torpacyc` (in both version) has no technique to prove non-termination.

However, the results also show that most of the problems in the test set are too hard to be proved by the techniques (only 399 out of 1315 problems were shown to be terminating or non-terminating). This is not really surprising since the test set contains already ‘hard’ instances for proving string termination as shown by the results in the last two columns. Moreover, a substantial part of the problems is expected not to be cycle terminating, for instance by containing a renaming of $ab \to ba$. This is confirmed by the result of non-termination by split and AProVE for over 300 of the systems.

Comparing the three transformations, the transformation split leads to much better results than the other two transformations, which holds for termination and for non-termination proofs. Ignoring the non-termination results (since `torpacyc` does not check for non-termination), the following observations are made: the new version of `torpacyc` indeed improves the former version, the power of the transformation split together with AProVE compared to `torpacyc` 2015 is more or less equal, while other transformations and back-ends do not perform as well as these tools. The combination of techniques increases the power, not only since the different tools perform well on different problems, but also, since `torpacyc` passes its output (a perhaps simplified SRS) to the string termination prover.

5.1.2 Randomly Generated String Rewrite Systems

As a further test set which contains much simpler problems than the former test set, we generated 50 000 SRSs of size 12 over an alphabet of size 3, where only SRSs with at least one rule ($\ell \to r$) with $|r| \geq |\ell|$ are considered (to rule out obviously terminating problems), SRSs with rules ($\ell \to u\ell v$) are not considered (to rule out obviously non-terminating problems), only SRSs without collapsing rules are considered (since `torpacyc` 2014 cannot handle such problems), and no alpha-equivalent (i.e. renamed and reordered) SRSs are generated.

The summarized results of applying our methods to this problem set are shown in Table 2.

Table 2 Results of proving cycle termination of 50 000 randomly generated problems of size 12 over an alphabet of size 3.

	torpacyc		split		rotate		shift		combination variant 1		combination variant 2		
	2014	2015	AProVE	T_TT_2	AProVE	T_TT_2	AProVE	T_TT_2	AProVE	T_TT_2	AProVE	T_TT_2	any
yes	46981	46929	47017	47073	36967	36260	37053	37027	47071	46967	47064	47015	**47124**
no	0	0	2331	2201	184	0	771	0	2328	2011	2334	2174	**2339**
maybe	3019	3071	652	726	12849	13740	12176	12973	601	1022	602	811	**537**

Most of the problems are cycle terminating and both versions of `torpacyc` as well as the transformation split together with AProVE or T_TT_2 can show termination of most of the problems. The problems seem to be too simple to separate the power of these successful approaches and their combinations. However, for the two transformations rotate and shift, the results show their lack in power, since no proof is found for roughly a quarter of all problems.

6 Conclusions

We developed new techniques to prove cycle termination. The main approach is to reduce the problem of cycle termination to the problem of string termination by applying a sound and complete transformation from cycle into string rewriting. We presented and analyzed three such transformations. Apart from that we provided a variant of matrix interpretations which improves the approach presented in [18]. Our implementations and the corresponding experimental results show that both techniques are useful in the sense that they apply for several examples for which the earlier techniques failed.

Together with the sound and complete transformation ϕ in the reverse direction from [18], the existence of a sound and complete transformation like split implies that the problems of cycle termination and string termination of SRSs are equivalent in a strong sense. For instance, it implies that they are in the same level of the arithmetic hierarchy, which is Π_2^0-complete along the lines of [4]. Alternatively, Π_2^0-completeness of cycle termination can be concluded from the sound and complete transformation ϕ combined with the observation that cycle termination is in Π_2^0.

Acknowledgments. We thank Johannes Waldmann for fruitful remarks, in particular for his suggestions leading to Section 4 on trace decreasing matrix interpretations. We also thank the anonymous reviewers for their valuable comments.

References

1. Homepage of AProVE, 2015. http://aprove.informatik.rwth-aachen.de.
2. H. J. Sander Bruggink, Barbara König, and Hans Zantema. Termination analysis for graph transformation systems. In Josep Diaz, Ivan Lanese, and Davide Sangiorgi, editors, *Proc. 8th IFIP International Conference on Theoretical Computer Science*, volume 8705 of *Lecture Notes in Comput. Sci.*, pages 179–194. Springer, 2014.
3. Bruno Dutertre. Yices 2.2. In Armin Biere and Roderick Bloem, editors, *Computer-Aided Verification (CAV'2014)*, volume 8559 of *Lecture Notes in Comput. Sci.*, pages 737–744. Springer, 2014.
4. Jörg Endrullis, Herman Geuvers, Jakob Grue Simonsen, and Hans Zantema. Levels of undecidability in rewriting. *Inf. Comput.*, 209(2):227–245, 2011.
5. Jörg Endrullis, Johannes Waldmann, and Hans Zantema. Matrix interpretations for proving termination of term rewriting. *J. Autom. Reasoning*, 40(2-3):195–220, 2008.
6. Jürgen Giesl, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, Thomas Ströder, Stephanie Swiderski, and René Thiemann. Proving termination of programs automatically with AProVE. In Stéphane Demri, Deepak Kapur, and Christoph Weidenbach, editors, *Proc. 7th International Joint Conference on Automated Reasoning (IJCAR'14)*, volume 8562 of *Lecture Notes in Comput. Sci.*, pages 184–191. Springer, 2014.
7. Jürgen Giesl and Aart Middeldorp. Transformation techniques for context-sensitive rewrite systems. *J. Funct. Program.*, 14(4):379–427, 2004.
8. Jürgen Giesl and Hans Zantema. Liveness in rewriting. In Robert Nieuwenhuis, editor, *Proc. 14th Conference on Rewriting Techniques and Applications (RTA)*, volume 2706 of *Lecture Notes in Comput. Sci.*, pages 321–336. Springer, 2003.
9. Dieter Hofbauer and Johannes Waldmann. Termination of $\{aa\text{->}bc, bb\text{->}ac, cc\text{->}ab\}$. *Inf. Process. Lett.*, 98(4):156–158, 2006.
10. Dieter Hofbauer and Johannes Waldmann. Termination of string rewriting with matrix interpretations. In Frank Pfenning, editor, *Proc. 17th Conference on Rewriting Techniques and Applications (RTA)*, volume 4098 of *Lecture Notes in Comput. Sci.*, pages 328–342. Springer, 2006.
11. Adam Koprowski and Johannes Waldmann. Arctic termination ...below zero. In Andrei Voronkov, editor, *Proc. 19th Conference on Rewriting Techniques and Applications (RTA)*, volume 5117 of *Lecture Notes in Comput. Sci.*, pages 202–216. Springer, 2008.
12. Martin Korp, Christian Sternagel, Harald Zankl, and Aart Middeldorp. Tyrolean termination tool 2. In Ralf Treinen, editor, *Proc. 20th Conference on Rewriting Techniques and Applications (RTA)*, volume 5595 of *Lecture Notes in Comput. Sci.*, pages 295–304. Springer, 2009.
13. David Sabel and Hans Zantema. Transforming cycle rewriting into string rewriting (extended version). http://www.ki.informatik.uni-frankfurt.de/research/cycsrs/SZ15.pdf, 2015.
14. The termination problem data base, 2015. http://termination-portal.org/wiki/TPDB.
15. Homepage of $\mathsf{T_TT_2}$, 2015. http://cl-informatik.uibk.ac.at/software/ttt2/.
16. Homepage of Yices, 2015. http://yices.csl.sri.com/.
17. Hans Zantema. Termination of term rewriting: Interpretation and type elimination. *J. Symb. Comput.*, 17(1):23–50, 1994.
18. Hans Zantema, Barbara König, and Harrie Jan Sander Bruggink. Termination of cycle rewriting. In Gilles Dowek, editor, *Proc. Joint 25th Conference on Rewriting Techniques and Applications and 12th Conference on Typed Lambda Calculi and Applications (RTATLCA)*, volume 8560 of *Lecture Notes in Comput. Sci.*, pages 476–490. Springer, 2014.

Confluence of Orthogonal Nominal Rewriting Systems Revisited

Takaki Suzuki, Kentaro Kikuchi, Takahito Aoto, and Yoshihito Toyama

RIEC, Tohoku University,
2-1-1 Katahira, Aoba-ku, Sendai, Miyagi, 980-8577, Japan
{takaki, kentaro, aoto, toyama}@nue.riec.tohoku.ac.jp

Abstract

Nominal rewriting systems (Fernández, Gabbay & Mackie, 2004; Fernández & Gabbay, 2007) have been introduced as a new framework of higher-order rewriting systems based on the nominal approach (Gabbay & Pitts, 2002; Pitts, 2003), which deals with variable binding via permutations and freshness conditions on atoms. Confluence of orthogonal nominal rewriting systems has been shown in (Fernández & Gabbay, 2007). However, their definition of (non-trivial) critical pairs has a serious weakness so that the orthogonality does not actually hold for most of standard nominal rewriting systems in the presence of binders. To overcome this weakness, we divide the notion of overlaps into the self-rooted and proper ones, and introduce a notion of α-stability which guarantees α-equivalence of peaks from the self-rooted overlaps. Moreover, we give a sufficient criterion for uniformity and α-stability. The new definition of orthogonality and the criterion offer a novel confluence condition effectively applicable to many standard nominal rewriting systems. We also report on an implementation of a confluence prover for orthogonal nominal rewriting systems based on our framework.

1998 ACM Subject Classification F.4.1 Mathematical Logic

Keywords and phrases Nominal rewriting, Confluence, Orthogonality, Higher-order rewriting, α-equivalence

Digital Object Identifier 10.4230/LIPIcs.RTA.2015.301

1 Introduction

Expressive formal systems such as systems of predicate logics, λ-calculi, process calculi, etc. need variable binding. *Nominal rewriting* [5][3] is a framework that extends first-order term rewriting by a binding mechanism. Studies of nominal rewriting are preceded by extensive studies of a *nominal approach* to terms and unifications [6][13][17]. A distinctive feature of the nominal approach is that α-conversion and capture-avoiding substitution are not relegated to meta-level—they are explicitly dealt with at object-level. This makes nominal rewriting significantly different from classical frameworks of higher-order rewriting systems such as *Combinatory Reduction Systems* [8] and *Higher-Order Rewriting Systems* [9] based on 'higher-order syntax'.

Confluence is a fundamental property of rewriting systems. As expected, the first results on confluence of *nominal rewriting systems* (*NRSs* for short) are generalisations of two classical results on confluence, namely Rosen's criterion (orthogonal systems are confluent) and Knuth-Bendix's criterion (terminating and locally confluent systems are confluent) [3]. We notice, however, that the confluence criterion in [3] for orthogonal NRSs is not applicable to standard NRSs—as the orthogonality in [3] contains the emptiness of the root overlaps of equivariant rules obtained from the same rule (*self-rooted overlaps*), which does not hold if

© Takaki Suzuki, Kentaro Kikuchi, Takahito Aoto, and Yoshihito Toyama;
licensed under Creative Commons License CC-BY
26th International Conference on Rewriting Techniques and Applications (RTA'15).
Editor: Maribel Fernández; pp. 301–317
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

the system contains a rewrite rule with binders (cf. Remark at the end of Subsection 3.2). Moreover, in contrast to the first-order case, one cannot skip the joinability check of self-rooted overlaps (cf. Example 19)—thus, if one relaxes the definition of orthogonality to omit such overlaps, then confluence is not guaranteed.

The contributions of this paper are summarised as follows:

- Our rewrite relation does not allow α-equivalent terms on the result of rewriting. Accordingly, we come to study confluence properties of rewriting modulo α-equivalence, which enables us to perform fine-grained analysis where confluence modulo and Church-Rosser modulo are different properties. Such an approach was suggested in [19, page 220].
- We overcome the above-mentioned defect of the orthogonality in [3] by introducing a notion of α-stability, which guarantees α-equivalence of peaks from the self-rooted overlaps. We prove Church-Rosser modulo α-equivalence for the class of orthogonal nominal rewriting systems that are uniform and α-stable.
- We introduce a notion of abstract skeleton preserving (ASP) as a sufficient criterion for uniformity and α-stability. To show the α-stability of ASP rewrite rules, we prove some lemmas on the system of α-equivalence in the nominal setting, which seem to be new.
- We report on an implementation of a confluence prover for nominal rewriting systems based on our criterion, that is, orthogonality and ASP. To check the emptiness of the proper overlaps, we use equivariant unification [2] with one permutation variable.

While a rewrite system in [5][3] is defined as an infinite set of rewrite rules that is closed under equivariance, we define a rewriting system as a finite set of rewrite rules. Instead of appealing to the property of equivariance, we specify a permutation as a parameter in each rewrite relation. (The idea of specifying a permutation as a parameter is found also in [4].) This allows us to make a discussion on avoiding capture of a free atom (cf. the latter part of Example 10) without referring to the property of equivariance.

As regards related work, Vestergaard and Brotherston [18][19] study a confluence proof of λ-calculus with variable names, not in the nominal setting, where α-conversion is seriously taken into account. Their definition of confluence is that of the reflexive transitive closure of $\to_\alpha \cup \to_\beta$. Formalisation of a confluence proof of first-order orthogonal term rewriting systems has been studied, e.g. in [11]. Our proof of Church-Rosser modulo α-equivalence can be seen as an extension of an inductive confluence proof of first-order orthogonal term rewriting systems (e.g. [16, Section 4.7] and [7]).

The organisation of the paper is as follows. In Section 2, we explain basic notions and notations of nominal rewriting. In Section 3, we discuss problems on confluence in nominal rewriting, and prove confluence of a class of nominal rewriting systems. In Section 4, we give a sufficient criterion for the class, and conclude in Section 6.

2 Nominal rewriting

Nominal rewriting [5][3] is a framework that extends first-order term rewriting by a binding mechanism. In this section, we redefine nominal rewriting systems as finite sets of rewrite rules, and introduce a notion of rewrite relation that is related to but different from the rewrite relation defined in [5][3]. In the subsequent sections, we will study confluence properties on our notion of rewrite relation.

2.1 Nominal terms

First, we introduce notions and notations concerning nominal terms.

A *nominal signature* Σ is a set of fixed arity *function symbols* ranged over by $f, g, \dots$. We fix a countably infinite set $\mathcal{X}$ of *variables* ranged over by $X, Y, Z, \dots$, and a countably infinite set $\mathcal{A}$ of *atoms* ranged over by $a, b, c, \dots$, and assume that Σ, $\mathcal{X}$, and $\mathcal{A}$ are pairwise disjoint. Unless otherwise stated, different meta-variables for objects in Σ, $\mathcal{X}$, or $\mathcal{A}$ denote different objects. A *swapping* is a pair of atoms, written $(a\ b)$. *Permutations* π are bijections on $\mathcal{A}$ such that the set of atoms for which $a \neq \pi(a)$ is finite. Permutations are represented by lists of swappings applied in the right-to-left order. For example, $((b\ c)(a\ b))(a) = c$, $((b\ c)(a\ b))(b) = a$, $((b\ c)(a\ b))(c) = b$. We write Id for the identity permutation, π^{-1} for the inverse of π, and $\pi \circ \pi'$ for the composition of π' and π.

Nominal terms, or simply *terms*, are generated by the grammar

$$t, s ::= a \mid \pi{\cdot}X \mid [a]t \mid f\ t \mid (t_1, \dots, t_n)$$

and called, respectively, atoms, moderated variables, abstractions, function applications (which must respect the arity of the function symbol) and tuples. We abbreviate $Id{\cdot}X$ as X if there is no ambiguity. We write $f\ ()$ as simply f. An abstraction $[a]t$ is intended to represent t with a bound. The set of *free* atoms occurring in t, denoted by $FA(t)$, is defined as follows: $FA(a) = \{a\}$; $FA(\pi{\cdot}X) = \emptyset$; $FA([a]t) = FA(t) \setminus \{a\}$; $FA(f\ t) = FA(t)$; $FA((t_1, \dots, t_n)) = \bigcup_i FA(t_i)$. We write $V(t) (\subseteq \mathcal{X})$ for the set of variables occurring in t. A *linear* term is a term in which any variable occurs at most once.

▶ **Example 1.** A nominal signature for the λ-calculus has two function symbols $\mathtt{lam}$ and $\mathtt{app}$ with arity 1 and 2, respectively. The nominal term $\mathtt{app}(\mathtt{lam}([a]\mathtt{lam}([b]\mathtt{app}(a, X))), a)$ represents the λ-term $(\lambda a.\lambda b.aX)a$ in the usual notation. Here X is a (meta-level) variable which can be instantiated by another term possibly with free atoms a and b. For this term t, we have $FA(t) = \{a\}$ and $V(t) = \{X\}$. ◀

Positions are finite sequences of positive integers. The empty sequence is denoted by ε. The set of positions in a term t, denoted by $Pos(t)$, is defined as follows: $Pos(a) = Pos(\pi{\cdot}X) = \{\varepsilon\}$; $Pos([a]t) = Pos(f\ t) = \{1p \mid p \in Pos(t)\} \cup \{\varepsilon\}$; $Pos((t_1, \dots, t_n)) = \bigcup_i \{ip \mid p \in Pos(t_i)\} \cup \{\varepsilon\}$. The subterm of t at a position $p \in Pos(t)$ is written as $t|_p$. For each $X \in V(t)$, we define $Pos_X(t) = \{p \in Pos(t) \mid \exists\pi.\ t|_p = \pi{\cdot}X\}$, and the set of *variable positions* in t is defined by $Pos_{\mathcal{X}}(t) = \bigcup_{X \in V(t)} Pos_X(t)$. The set of *atom positions* in t is defined by $Pos_{\mathcal{A}}(t) = \{p \in Pos(t) \mid \exists a \in \mathcal{A}.\ t|_p = a\}$, and we define $Pos_{\mathcal{X}\mathcal{A}}(t) = Pos_{\mathcal{X}}(t) \cup Pos_{\mathcal{A}}(t)$.

A *context* is a term in which a distinguished function symbol $\Box$ with arity 0 occurs. The term obtained from a context $C[\,]$ by replacing each $\Box$ at positions p_i by terms t_i is written as $C[t_1, \dots, t_n]_{p_1, \dots, p_n}$ or simply $C[t_1, \dots, t_n]$.

Next, we define two kinds of permutation actions, which operate on terms extending a permutation on atoms. These actions are used to define substitution, α-equivalence and rewrite relation for nominal rewriting systems. The first permutation action, written $\pi{\cdot}t$, is defined inductively by: $\pi{\cdot}a = \pi(a)$; $\pi{\cdot}(\pi'{\cdot}X) = (\pi \circ \pi'){\cdot}X$; $\pi{\cdot}(t_1, \dots, t_n) = (\pi{\cdot}t_1, \dots, \pi{\cdot}t_n)$; $\pi{\cdot}([a]t) = [\pi{\cdot}a](\pi{\cdot}t)$; $\pi{\cdot}(f\ t) = f\ \pi{\cdot}t$. The second permutation action, written t^π, is defined by: $a^\pi = \pi(a)$; $(\pi'{\cdot}X)^\pi = (\pi \circ \pi' \circ \pi^{-1}){\cdot}X$; $(t_1, \dots, t_n)^\pi = (t_1^\pi, \dots, t_n^\pi)$; $([a]t)^\pi = [a^\pi](t^\pi)$; $(f\ t)^\pi = f\ t^\pi$. The difference consists in the clause for moderated variables. In particular, when $\pi' = Id$, π is suspended before X in the first action as $\pi{\cdot}(Id{\cdot}X) = (\pi \circ Id){\cdot}X = \pi{\cdot}X$, while in the second action π has no effect as $(Id{\cdot}X)^\pi = (\pi \circ Id \circ \pi^{-1}){\cdot}X = Id{\cdot}X$.

A *substitution* is a map σ from variables to terms such that the set $\{X \in \mathcal{X} \mid \sigma(X) \neq X\}$ is finite. Substitutions act on variables, without avoiding capture of atoms. We write $t\sigma$ for

$$\frac{}{\nabla \vdash a\#b} \qquad \frac{\nabla \vdash a\#t}{\nabla \vdash a\#f\ t} \qquad \frac{\nabla \vdash a\#t_1 \quad \cdots \quad \nabla \vdash a\#t_n}{\nabla \vdash a\#(t_1,\ldots,t_n)}$$

$$\frac{}{\nabla \vdash a\#[a]t} \qquad \frac{\nabla \vdash a\#t}{\nabla \vdash a\#[b]t} \qquad \frac{\pi^{-1}{\cdot}a\#X \in \nabla}{\nabla \vdash a\#\pi{\cdot}X}$$

Figure 1 Rules for freshness constraints.

$$\frac{}{\nabla \vdash a \approx_\alpha a} \qquad \frac{\nabla \vdash t \approx_\alpha s}{\nabla \vdash f\ t \approx_\alpha f\ s} \qquad \frac{\nabla \vdash t_1 \approx_\alpha s_1 \quad \cdots \quad \nabla \vdash t_n \approx_\alpha s_n}{\nabla \vdash (t_1,\ldots,t_n) \approx_\alpha (s_1,\ldots,s_n)}$$

$$\frac{\nabla \vdash t \approx_\alpha s}{\nabla \vdash [a]t \approx_\alpha [a]s} \qquad \frac{\nabla \vdash (a\ b){\cdot}t \approx_\alpha s \quad \nabla \vdash b\#t}{\nabla \vdash [a]t \approx_\alpha [b]s} \qquad \frac{\forall a \in ds(\pi,\pi').\ a\#X \in \nabla}{\nabla \vdash \pi{\cdot}X \approx_\alpha \pi'{\cdot}X}$$

Figure 2 Rules for α-equivalence.

the application of σ on t. Note here that by replacing X of a moderated variable $\pi{\cdot}X$ in t by $\sigma(X)$, a permutation action $\pi{\cdot}(\sigma(X))$ occurs. For a permutation π and a substitution σ, we define the substitution $\pi{\cdot}\sigma$ by $(\pi{\cdot}\sigma)(X) = \pi{\cdot}(\sigma(X))$.

2.2 α-equivalence and nominal rewriting systems

The distinctive feature of nominal rewriting is that it is equipped with a mechanism to avoid accidental capture of free atoms on the way of rewriting. This is partly achieved by α-conversion built in the matching process of the LHS of a rule and a redex involving also permutations (cf. Example 10).

In this subsection, we first recall the notion of α-equivalence in the nominal setting. This is different from α-equivalence in the traditional sense in that equivalence between terms is discussed under assumptions on the freshness of atoms in variables.

A pair $a\#t$ of an atom a and a term t is called a *freshness constraint*. Intuitively, this means that a does not occur as a free atom in t, including the cases where the variables in t are instantiated by other terms. A finite set $\nabla \subseteq \{a\#X \mid a \in \mathcal{A}, X \in \mathcal{X}\}$ is called a *freshness context*. For a freshness context ∇, we define $V(\nabla) = \{X \in \mathcal{X} \mid \exists a.\ a\#X \in \nabla\}$, $\nabla^\pi = \{a^\pi\#X \mid a\#X \in \nabla\}$, and $\nabla\sigma = \{a\#\sigma(X) \mid a\#X \in \nabla\}$.

The rules in Figure 1 defines the relation $\nabla \vdash a\#t$, which means that $a\#t$ is satisfied under the freshness context ∇. It can be seen that $a \notin FA(t)$ whenever $\nabla \vdash a\#t$. An example using the last rule is $\{c\#X\} \vdash a\#((a\ b)(b\ c)){\cdot}X$, since $((a\ b)(b\ c))^{-1}{\cdot}a = ((b\ c)(a\ b))(a) = c$.

The rules in Figure 2 defines the relation $\nabla \vdash t \approx_\alpha s$, which means that t is α-equivalent to s under the freshness context ∇. $ds(\pi,\pi')$ in the last rule denotes the set $\{a \in \mathcal{A} \mid \pi{\cdot}a \neq \pi'{\cdot}a\}$. For example, $ds((a\ b), Id) = \{a, b\}$.

▶ **Example 2.** Consider the nominal signature for the λ-calculus in Example 1, and suppose $\nabla = \{a\#X, b\#X\}$. Then we have the following derivation:

$$\cfrac{\cfrac{\cfrac{\cfrac{a\#X \in \nabla}{\nabla \vdash a\#X} \quad \cfrac{b\#X \in \nabla}{\nabla \vdash b\#X}}{\nabla \vdash (a\ b){\cdot}X \approx_\alpha X} \quad \cfrac{b\#X \in \nabla}{\nabla \vdash b\#X}}{\nabla \vdash [a]X \approx_\alpha [b]X}}{\nabla \vdash \mathtt{lam}([a]X) \approx_\alpha \mathtt{lam}([b]X)}$$

◀

The following properties are shown in [3].

▶ **Proposition 3** ([3]).

1. $\nabla \vdash a\#t$ *if and only if* $\nabla \vdash \pi{\cdot}a\#\pi{\cdot}t$.

2. $\nabla \vdash t \approx_\alpha s$ *if and only if* $\nabla \vdash \pi{\cdot}t \approx_\alpha \pi{\cdot}s$.

3. *If* $\nabla \vdash a\#t$ *and* $\nabla \vdash t \approx_\alpha s$ *then* $\nabla \vdash a\#s$.

4. $\forall a \in ds(\pi, \pi').\, \nabla \vdash a\#t$ *if and only if* $\nabla \vdash \pi{\cdot}t \approx_\alpha \pi'{\cdot}t$.

▶ **Proposition 4** ([3]). *For any freshness context* ∇*, the binary relation* $\nabla \vdash - \approx_\alpha -$ *is a congruence (i.e. an equivalence relation that is closed under any context* $C[\,]$*).*

For terms with no variables, this relation coincides with usual α-equivalence (i.e. the relation reached by renamings of bound atoms) [6].

Now we define nominal rewrite rules and nominal rewriting systems.

▶ **Definition 5** (Nominal rewrite rule). A *nominal rewrite rule*, or simply *rewrite rule*, is a triple of a freshness context ∇ and terms l and r such that $V(\nabla) \cup V(r) \subseteq V(l)$. We write $\nabla \vdash l \to r$ for a rewrite rule. A rewrite rule $\nabla \vdash l \to r$ is *left-linear* if l is linear. We define $V(\nabla \vdash l \to r) = V(\nabla) \cup V(l) \cup V(r)$ and $(\nabla \vdash l \to r)^\pi = \nabla^\pi \vdash l^\pi \to r^\pi$.

▶ **Example 6.** Using the nominal signature for the λ-calculus in Example 1, the η-rule can be represented by the following rewrite rule (we omit the braces on the LHS of $\vdash$):

$$a\#X \quad \vdash \quad \mathtt{lam}([a]\mathtt{app}(X, a)) \quad \to \quad X \qquad (\mathsf{Eta})$$

This rule is left-linear. ◀

▶ **Definition 7** (Nominal rewriting system). A *nominal rewriting system*, or simply *rewriting system*, is a finite set of rewrite rules. A rewriting system is *left-linear* if so are all its rewrite rules.

▶ **Example 8.** We extend the signature in Example 1 by a function symbol $\mathtt{sub}$ with arity 2. By $\mathtt{sub}([a]t, s)$, we represent an explicit substitution $t\langle a := s\rangle$. Then, a nominal rewriting system to perform β-reduction is defined by the rule (Beta):

$$\vdash \quad \mathtt{app}(\mathtt{lam}([a]X), Y) \quad \to \quad \mathtt{sub}([a]X, Y) \qquad (\mathsf{Beta})$$

together with a rewriting system $\mathcal{R}_\sigma$ to execute substitution:

$$\mathcal{R}_\sigma = \left\{ \begin{array}{rcrcll} & \vdash & \mathtt{sub}([a]\mathtt{app}(X, Y), Z) & \to & \mathtt{app}(\mathtt{sub}([a]X, Z), \mathtt{sub}([a]Y, Z)) & (\sigma_{\mathtt{app}}) \\ & \vdash & \mathtt{sub}([a]a, X) & \to & X & (\sigma_{\mathtt{var}}) \\ & \vdash & \mathtt{sub}([a]b, X) & \to & b & (\sigma_{\mathtt{var}\epsilon}) \\ b\#Y & \vdash & \mathtt{sub}([a]\mathtt{lam}([b]X), Y) & \to & \mathtt{lam}([b]\mathtt{sub}([a]X, Y)) & (\sigma_{\mathtt{lam}}) \end{array} \right.$$

In a standard notation, the system $\mathcal{R}_\sigma$ is represented as follows:

$$\mathcal{R}_\sigma = \left\{ \begin{array}{rcrcll} & \vdash & (XY)\langle a := Z\rangle & \to & (X\langle a := Z\rangle)(Y\langle a := Z\rangle) & (\sigma_{\mathtt{app}}) \\ & \vdash & a\langle a := X\rangle & \to & X & (\sigma_{\mathtt{var}}) \\ & \vdash & b\langle a := X\rangle & \to & b & (\sigma_{\mathtt{var}\epsilon}) \\ b\#Y & \vdash & (\lambda b.X)\langle a := Y\rangle & \to & \lambda b.(X\langle a := Y\rangle) & (\sigma_{\mathtt{lam}}) \end{array} \right.$$

◀

In [5][3], nominal rewrite systems are defined as infinite sets of rewrite rules that are closed under equivariance, i.e., if R is a rule of a rewrite system $\mathcal{R}$ then so is R^π for any permutation π. In the present paper, we define rewriting systems as finite sets of rewrite rules that may not be closed under equivariance. Accordingly, our rewrite relation is defined with a permutation as a parameter unlike in the definition of rewrite relation in [5][3]. In the following, $\vdash$ is extended to mean to hold for every member of a set or a sequence on the RHS.

▶ **Definition 9** (Rewrite relation). Let $R = \nabla \vdash l \to r$ be a rewrite rule. For a freshness context Δ and terms s and t, the *rewrite relation* is defined by

$$\Delta \vdash s \to_{\langle R,\pi,p,\sigma\rangle} t \quad \stackrel{\text{def}}{\iff} \quad \Delta \vdash \nabla^\pi\sigma,\ s = C[s']_p,\ \Delta \vdash s' \approx_\alpha l^\pi\sigma,\ t = C[r^\pi\sigma]_p$$

where $V(l) \cap (V(\Delta) \cup V(s)) = \emptyset$. We write $\Delta \vdash s \to_{\langle R,\pi\rangle} t$ if there exist p and σ such that $\Delta \vdash s \to_{\langle R,\pi,p,\sigma\rangle} t$. We write $\Delta \vdash s \to_R t$ if there exists π such that $\Delta \vdash s \to_{\langle R,\pi\rangle} t$. For a rewriting system $\mathcal{R}$, we write $\Delta \vdash s \to_{\mathcal{R}} t$ if there exists $R \in \mathcal{R}$ such that $\Delta \vdash s \to_R t$.

▶ **Example 10.** Using the rule (Beta) in Example 8, we see that the term representing $(\lambda a.\lambda b.ba)b$ rewrites to $(\lambda b.ba)\langle a := b\rangle$, that is, we have

$$\vdash \mathtt{app}(\mathtt{lam}([a]\mathtt{lam}([b]\mathtt{app}(b,a))),b) \to_{\langle \mathsf{Beta},Id,\varepsilon,\sigma\rangle} \mathtt{sub}([a]\mathtt{lam}([b]\mathtt{app}(b,a)),b)$$

where σ is the substitution $[X := \mathtt{lam}([b]\mathtt{app}(b,a)), Y := b]$. The resulting term rewrites further to a normal form $\mathtt{lam}([c]\mathtt{app}(c,b))$ in four steps with rules of the system $\mathcal{R}_\sigma$. Here we give a detail of the first step with rule $(\sigma_{\mathtt{lam}})$ to see how capture of a free atom is avoided.

Let $s = \mathtt{sub}([a]\mathtt{lam}([b]\mathtt{app}(b,a)),b)$. Since the rule has a freshness context $\nabla = \{b\#Y\}$, to apply $(\sigma_{\mathtt{lam}})$ to s at the position $p = \varepsilon$, it is necessary to find a permutation π and a substitution σ that satisfy $\vdash \nabla^\pi\sigma$ and $\vdash s \approx_\alpha (\mathtt{sub}([a]\mathtt{lam}([b]X),Y))^\pi\sigma$. Here one cannot simply take $\pi = Id$, because then $\sigma(Y) = b$ from the condition for $\approx_\alpha$, which contradicts $\vdash \nabla^\pi\sigma$. So we take, e.g. $\pi = (b\ c)$ and $\sigma = [X := \mathtt{app}(c,a), Y := b]$ to satisfy the conditions, and get $(\mathtt{lam}([b]\mathtt{sub}([a]X,Y)))^\pi\sigma = \mathtt{lam}([c]\mathtt{sub}([a]\mathtt{app}(c,a),b))$ as the result of rewriting. ◀

In the following, a binary relation $\Delta \vdash - \bowtie -$ ($\bowtie$ is $\to_R$, $\approx_\alpha$, etc.) with a fixed freshness context Δ is called the relation $\bowtie$ under Δ, or simply the relation $\bowtie$ if there is no ambiguity. If a relation $\bowtie$ is written using $\to$ then the inverse is written using $\leftarrow$. Also, we write $\bowtie^=$ for the reflexive closure, and $\bowtie^*$ for the reflexive transitive closure. We use $\circ$ for the composition of relations.

▶ Remark. In [5, page 113][3, page 946], the rewrite relation, which we denote by $\Delta \vdash s \xrightarrow[FGM]{}_R t$, is defined in the following way. For a given rewrite rule $R = \nabla \vdash l \to r$, $\Delta \vdash s \xrightarrow[FGM]{}_R t$ holds if

1. $V(R) \cap (V(\Delta) \cup V(s)) = \emptyset$.
2. $s = C[s']$ for some context $C[\,]$ and term s', such that $\Delta \vdash \nabla\sigma$, $\Delta \vdash s' \approx_\alpha l\sigma$ for some σ.
3. $\Delta \vdash t \approx_\alpha C[r\sigma]$.

Hence, $\xrightarrow[FGM]{}_R$ differs from our $\to_R$ in the following two points. First, the rules of a rewrite system in [5][3] are closed under equivariance, so that the rewrite relation is defined without a permutation as a parameter. Secondly, α-equivalent terms are allowed on the result of rewriting. Consequently, under the same freshness context, we have $\xrightarrow[FGM]{}_R = \to_{\langle R,Id\rangle} \circ \approx_\alpha$.

3 Confluence of nominal rewriting systems

Having defined basic notions on nominal terms and nominal rewriting systems, we now set out to investigate confluence properties on the rewrite relations of nominal rewriting systems.

To be exact, we study confluence properties modulo the equivalence relation $\approx_\alpha$ in terms of abstract reduction systems [10].

▶ **Definition 11.** Let $\mathcal{R}$ be a nominal rewriting system.

1. $\to_{\mathcal{R}}$ is *confluent modulo* $\approx_\alpha$ if $\Delta \vdash s\ ({}_{\mathcal{R}}^{*}\!\leftarrow \circ \to_{\mathcal{R}}^{*})\ t$ implies $\Delta \vdash s\ (\to_{\mathcal{R}}^{*} \circ \approx_\alpha \circ\ {}_{\mathcal{R}}^{*}\!\leftarrow)\ t$.
2. $\to_{\mathcal{R}}$ is *Church-Rosser modulo* $\approx_\alpha$ if $\Delta \vdash s\ (\leftarrow_{\mathcal{R}} \cup \to_{\mathcal{R}} \cup \approx_\alpha)^*\ t$ implies $\Delta \vdash s\ (\to_{\mathcal{R}}^{*} \circ \approx_\alpha \circ\ {}_{\mathcal{R}}^{*}\!\leftarrow)\ t$.
3. $\to_{\mathcal{R}}$ is *strongly compatible with* $\approx_\alpha$ if $\Delta \vdash s\ (\approx_\alpha \circ \to_{\mathcal{R}})\ t$ implies $\Delta \vdash s\ (\to_{\mathcal{R}}^{=} \circ \approx_\alpha)\ t$.

It is known that Church-Rosser modulo an equivalence relation $\sim$ is a stronger property than confluence modulo $\sim$ [10]. So we aim to prove Church-Rosser modulo $\approx_\alpha$ for some class of nominal rewriting systems. The strong compatibility with $\approx_\alpha$ also plays an important role in proving Church-Rosser modulo $\approx_\alpha$ through results in [10].

3.1 Self-rooted and proper overlaps

In the study of confluence, the notion of overlaps is important because they are useful for analysing how peaks $\Delta \vdash s \to_{\mathcal{R}} t$ and $\Delta \vdash s \to_{\mathcal{R}} t'$ occur. In this subsection, we introduce two kinds of overlaps and give some examples.

First, we define unification for nominal terms. Let P be a set of equations and freshness constraints $\{s_1 \approx t_1, \ldots, s_m \approx t_m, a_1\#u_1, \ldots, a_n\#u_n\}$ (where a_i and a_j may denote the same atom). Then, P is *unifiable* if there exist a freshness context Γ and a substitution θ such that $\Gamma \vdash s_1\theta \approx_\alpha t_1\theta, \ldots, s_m\theta \approx_\alpha t_m\theta, a_1\#u_1\theta, \ldots, a_n\#u_n\theta$; the pair $\langle \Gamma, \theta \rangle$ is called a *unifier* of P. It is known that the unification problem for nominal terms is decidable [17].

▶ **Example 12.** Consider the nominal signature for the λ-calculus in Example 1, and let $P = \{\mathtt{lam}([a]\mathtt{app}(X, a)) \approx \mathtt{lam}([a]Y), a\#X\}$. Then, $\langle \{a\#X\}, [Y := \mathtt{app}(X, a)] \rangle$ is a unifier of P. ◀

▶ **Definition 13** (Overlap). Let $R_i = \nabla_i \vdash l_i \to r_i$ $(i = 1, 2)$ be rewrite rules. We assume without loss of generality that $V(R_1) \cap V(R_2) = \emptyset$. If $\nabla_1 \cup \nabla_2^{\pi_2} \cup \{l_1 \approx l_2^{\pi_2}|_p\}$ is unifiable for some permutation π_2 and a non-variable position p, then we say that R_1 *overlaps* on R_2, and the situation is called an *overlap* of R_1 on R_2. If R_1 and R_2 are identical modulo renaming of variables and $p = \varepsilon$, then the overlap is said to be *self-rooted*. An overlap that is not self-rooted is said to be *proper*.

▶ **Example 14.** Let R_1 and R_2 be the rules (Eta) $a\#X \vdash \mathtt{lam}([a]\mathtt{app}(X, a)) \to X$ and (Beta) $\vdash \mathtt{app}(\mathtt{lam}([a]Y), Z) \to \mathtt{sub}([a]Y, Z)$ from Examples 6 and 8, respectively. Then, R_1 overlaps on R_2, since $\{a\#X\} \cup \{\mathtt{lam}([a]\mathtt{app}(X, a)) \approx (\mathtt{app}(\mathtt{lam}([a]Y), Z))^{Id}|_{11} (= \mathtt{lam}([a]Y))\}$ is unifiable as seen in Example 12. This overlap is proper. ◀

▶ **Example 15.** There exist self-rooted overlaps of the rule (Beta) on its renamed variant, since $\{\mathtt{app}(\mathtt{lam}([a]Y), Z) \approx (\mathtt{app}(\mathtt{lam}([a]X), W))^\pi\}$ is unifiable for any permutation π. In the case of $\pi(a) = b$, we take $\langle \{a\#X, b\#X\}, [Y := X, Z := W] \rangle$ as a unifier (cf. Example 2). ◀

In first-order term rewriting, self-rooted overlaps do not matter, and only proper overlaps need to be analysed. However, in the case of nominal rewriting, that is not enough as seen in the next subsection.

3.2 Problems on confluence of nominal rewriting systems

In this subsection, we discuss problems on confluence in nominal rewriting that are not present in first-order term rewriting.

A standard confluence criterion in rewriting theory is the one by orthogonality.

▶ **Definition 16** (Orthogonality). A nominal rewriting system $\mathcal{R}$ is *orthogonal* if it is left-linear and for any rules $R_1, R_2 \in \mathcal{R}$, there exists no proper overlap of R_1 on R_2.

This definition of orthogonality is different from the one in [3] (cf. Remark at the end of this subsection).

Unlike in first-order term rewriting, orthogonality is not enough to guarantee confluence of a nominal rewriting system.

▶ **Example 17.** Consider the nominal rewriting system $\mathcal{R}_{\mathsf{uc}\text{-}\eta}$ with the only rewrite rule:

$$\vdash \quad \mathtt{lam}([a]\mathtt{app}(X,a)) \quad \to \quad X \qquad (\mathsf{Uncond\text{-}eta})$$

The system $\mathcal{R}_{\mathsf{uc}\text{-}\eta}$ is orthogonal, but is not Church-Rosser (even confluent) modulo $\approx_\alpha$, since $\vdash \mathtt{lam}([a]\mathtt{app}(a,a)) \to_{\langle \mathsf{Uncond\text{-}eta}, Id\rangle} a$ and $\vdash \mathtt{lam}([a]\mathtt{app}(a,a)) \to_{\langle \mathsf{Uncond\text{-}eta},(a\ b)\rangle} b$. The latter follows from $\vdash \mathtt{lam}([a]\mathtt{app}(a,a)) \approx_\alpha \mathtt{lam}([b]\mathtt{app}(b,b)) = (\mathtt{lam}([a]\mathtt{app}(X,a)))^{(a\ b)}[X := b]$ (the third condition of rewrite relation in Definition 9). ◀

The above kind of rules can be excluded by the uniformity condition introduced in [3]. Intuitively, uniformity means that if an atom a is not free in s and s rewrites to t then a is not free in t. Here we employ the following definition of uniformity which is equivalent to the one in [3].

▶ **Definition 18** (Uniformity). A rewrite rule $\nabla \vdash l \to r$ is *uniform* if for any atom a and any freshness context Δ, $\Delta \vdash \nabla$ and $\Delta \vdash a\#l$ imply $\Delta \vdash a\#r$. A rewriting system is *uniform* if so are all its rewrite rules.

The rule (Uncond-eta) in Example 17 is not uniform, since $\vdash a\#\mathtt{lam}([a]\mathtt{app}(X,a))$ but not $\vdash a\#X$. Uniform rewriting systems have many good properties, which we use in the proof of confluence in the next section.

Our definition of orthogonality together with uniformity does not guarantee confluence of a nominal rewriting system, as seen in the next example.

▶ **Example 19.** We extend the signature in Example 1 by a function symbol `uc-eta-exp` with arity 1. Consider the nominal rewriting system $\mathcal{R}_{\mathsf{uc}\text{-}\eta\text{-}\mathsf{exp}}$ with the only rewrite rule:

$$\vdash \quad \mathtt{uc\text{-}eta\text{-}exp}(X) \quad \to \quad \mathtt{lam}([a]\mathtt{app}(X,a)) \qquad (\mathsf{Uncond\text{-}eta\text{-}exp})$$

The system $\mathcal{R}_{\mathsf{uc}\text{-}\eta\text{-}\mathsf{exp}}$ is orthogonal and uniform. Uniformity follows from the observation that for any atom a' and any freshness context Δ, if $\Delta \vdash a'\#\mathtt{uc\text{-}eta\text{-}exp}(X)$, which is equivalent to $\Delta \vdash a'\#X$, then $\Delta \vdash a'\#\mathtt{lam}([a]\mathtt{app}(X,a))$. We see, however, that $\mathcal{R}_{\mathsf{uc}\text{-}\eta\text{-}\mathsf{exp}}$ is not Church-Rosser (even confluent) modulo $\approx_\alpha$, since $\vdash \mathtt{uc\text{-}eta\text{-}exp}(a) \to_{\langle \mathsf{Uncond\text{-}eta\text{-}exp}, Id\rangle} \mathtt{lam}([a]\mathtt{app}(a,a))$ and $\vdash \mathtt{uc\text{-}eta\text{-}exp}(a) \to_{\langle \mathsf{Uncond\text{-}eta\text{-}exp},(a\ b)\rangle} \mathtt{lam}([b]\mathtt{app}(a,b))$, where the resulting two terms are normal forms in $\mathcal{R}_{\mathsf{uc}\text{-}\eta\text{-}\mathsf{exp}}$ and not α-equivalent. ◀

So orthogonality together with uniformity is still not enough to guarantee confluence of a nominal rewriting system. In the next section, we consider another condition that excludes rewrite rules like the rule (Uncond-eta-exp) in Example 19.

▶ Remark. It is claimed in [3] that terminating uniform nominal rewrite systems are confluent if all non-trivial[1] critical pairs are joinable, and that orthogonal uniform nominal rewrite systems are confluent. However, the latter criterion is not applicable to standard nominal rewrite systems: In [3], a critical pair is said to be trivial if it is obtained from the root overlap of the same (renamed) rewrite rule or obtained from a variable overlap[2], where overlaps are considered without permutations unlike in our Definition 13, and a nominal rewrite system is said to be orthogonal if it is left-linear and has no non-trivial critical pair. Then, for example, a critical pair obtained from the root overlap of the two rules (Beta) and (Beta)$^\pi$ (cf. Example 15) is non-trivial, and so any rewrite system with the rule (Beta), which also has (Beta)$^\pi$ by equivariance, is not orthogonal in the sense of [3]. The same can be said of many other rewrite rules and systems.

3.3 Confluence proof of orthogonal nominal rewriting systems

In the previous subsection, we discussed problems on confluence that are peculiar to nominal rewriting. In this subsection, we prove confluence (Church-Rosser modulo $\approx_\alpha$) for a class of nominal rewriting systems with one more condition besides uniformity and orthogonality. The proof can be considered as an adaptation of inductive confluence proofs of first-order orthogonal term rewriting systems found, e.g. in [16, Section 4.7] and [7]. We omit some of (the details of) the proofs, which are available at [15].

First, we define a notion of parallel reduction using a particular kind of contexts.

▶ **Definition 20.** The *grammatical contexts*, ranged over by $G[\,]$, are the contexts defined by

$$G[\,] ::= a \mid \pi{\cdot}X \mid [a]\Box \mid f\ \Box \mid (\Box_1, \ldots, \Box_n)$$

Let $\mathcal{R}$ be a nominal rewriting system. For a given freshness context Δ, we define the relation $\Delta \vdash - \twoheadrightarrow\!\!\!\!\mapsto_{\mathcal{R}} -$ inductively by the following rules:

$$\frac{\Delta \vdash s_1 \twoheadrightarrow\!\!\!\!\mapsto_{\mathcal{R}} t_1 \quad \cdots \quad \Delta \vdash s_n \twoheadrightarrow\!\!\!\!\mapsto_{\mathcal{R}} t_n}{\Delta \vdash G[s_1, \ldots, s_n] \twoheadrightarrow\!\!\!\!\mapsto_{\mathcal{R}} G[t_1, \ldots, t_n]}\ (\mathsf{context}) \qquad \frac{\Delta \vdash s \to_{\langle R,\pi,\varepsilon,\sigma\rangle} t \quad R \in \mathcal{R}}{\Delta \vdash s \twoheadrightarrow\!\!\!\!\mapsto_{\mathcal{R}} t}\ (\mathsf{head})$$

where $n\,(\geq 0)$ depends on the form of $G[\,]$. We define $\Delta \vdash \sigma \twoheadrightarrow\!\!\!\!\mapsto_{\mathcal{R}} \delta$ by $\forall X.\, \Delta \vdash X\sigma \twoheadrightarrow\!\!\!\!\mapsto_{\mathcal{R}} X\delta$.

▶ **Lemma 21.** ***1.*** *$\Delta \vdash s \twoheadrightarrow\!\!\!\!\mapsto_{\mathcal{R}} s$.*
2. *If $\Delta \vdash s \twoheadrightarrow\!\!\!\!\mapsto_{\mathcal{R}} t$ then $\Delta \vdash C[s] \twoheadrightarrow\!\!\!\!\mapsto_{\mathcal{R}} C[t]$.*
3. *If $\Delta \vdash s \to_{\langle R,\pi,p,\sigma\rangle} t$ then $\Delta \vdash s \twoheadrightarrow\!\!\!\!\mapsto_{\mathcal{R}} t$.*
4. *If $\Delta \vdash s \twoheadrightarrow\!\!\!\!\mapsto_{\mathcal{R}} t$ then $\Delta \vdash s \to^*_{\mathcal{R}} t$.*

Instead of showing the diamond property of $\twoheadrightarrow\!\!\!\!\mapsto_{\mathcal{R}}$ as in usual confluence proofs, we prove strong local confluence modulo $\approx_\alpha$ (Lemma 27), which together with strong compatibility with $\approx_\alpha$ (Lemma 22) yields Church-Rosser modulo $\approx_\alpha$ of $\twoheadrightarrow\!\!\!\!\mapsto_{\mathcal{R}}$ (and hence of $\to_{\mathcal{R}}$).

▶ **Lemma 22** (Strong compatibility with $\approx_\alpha$)**.** *Let $\mathcal{R}$ be a uniform nominal rewriting system. If $\Delta \vdash s' \approx_\alpha s \twoheadrightarrow\!\!\!\!\mapsto_{\mathcal{R}} t$ then there exists t' such that $\Delta \vdash s' \twoheadrightarrow\!\!\!\!\mapsto_{\mathcal{R}} t' \approx_\alpha t$.*

Proof. By induction on the derivation of $\Delta \vdash s \twoheadrightarrow\!\!\!\!\mapsto_{\mathcal{R}} t$. For the details, see [15]. ◀

[1] As mentioned soon, the definition of non-trivial here is different from the standard one.
[2] This definition is not very standard, but it is not the point here.

The key lemma to strong local confluence modulo $\approx_\alpha$ is Lemma 25, which corresponds to Lemma 4.7.7 of [16, page 122] in the first-order case. To show it, we first prove the following two technical lemmas.

▶ **Lemma 23.** *Let $\mathcal{R}$ be a nominal rewriting system, and let $R = \nabla \vdash l \to r \in \mathcal{R}$ and $\hat{R} = \hat{\nabla} \vdash \hat{l} \to \hat{r} \in \mathcal{R}$ (we assume $V(R) \cap V(\hat{R}) = \emptyset$). Suppose that l' is a proper subterm of l^π for some π where l' is not a moderated variable, and that there exist $\sigma, \hat{\pi}, \hat{\sigma}, \Delta, s$ that satisfy $\Delta \vdash \nabla^\pi \sigma$, $\Delta \vdash s \approx_\alpha l'\sigma$, $\Delta \vdash \hat{\nabla}^{\hat{\pi}} \hat{\sigma}$ and $\Delta \vdash s \approx_\alpha \hat{l}^{\hat{\pi}} \hat{\sigma}$. Then $\mathcal{R}$ is not orthogonal.*

Proof. Let p be the position of the subterm l' of l^π, i.e., $l^\pi|_p = l'$. Since $\hat{\nabla} \cup \nabla^{\hat{\pi}^{-1} \circ \pi} \cup \{\hat{l} \approx l^{\hat{\pi}^{-1} \circ \pi}|_p (= l'^{\hat{\pi}^{-1}})\}$ is unifiable with a unifier $\langle \Delta, \hat{\pi}^{-1} \cdot (\hat{\sigma} \cup \sigma) \rangle$, $\mathcal{R}$ is not orthogonal. ◀

▶ **Lemma 24.** *Let $\mathcal{R}$ be a uniform rewriting system. Then, if $\Delta \vdash \nabla\sigma$, $\Delta \vdash s \approx_\alpha \pi \cdot X\sigma$ and $\Delta \vdash s \twoheadrightarrow\!\!\!\!\!\!\!\!+\;\;\;_\mathcal{R} t$ then there exists δ such that $\Delta \vdash \nabla\delta$, $\Delta \vdash t \approx_\alpha \pi \cdot X\delta$, $\Delta \vdash \sigma \twoheadrightarrow\!\!\!\!\!\!\!\!+\;\;\;_\mathcal{R} \delta$ and for any $Y \neq X$, $Y\sigma = Y\delta$.*

Now we prove the announced lemma.

▶ **Lemma 25.** *Let $\mathcal{R}$ be an orthogonal uniform rewriting system, and let $\nabla \vdash l \to r \in \mathcal{R}$. Suppose that l' is a proper subterm of l^π for some π. Then, if $\Delta \vdash \nabla^\pi \sigma$, $\Delta \vdash s \approx_\alpha l'\sigma$ and $\Delta \vdash s \twoheadrightarrow\!\!\!\!\!\!\!\!+\;\;\;_\mathcal{R} t$ then there exists δ such that $\Delta \vdash \nabla^\pi \delta$, $\Delta \vdash t \approx_\alpha l'\delta$, $\Delta \vdash \sigma \twoheadrightarrow\!\!\!\!\!\!\!\!+\;\;\;_\mathcal{R} \delta$ and for any $X \notin V(l')$, $X\sigma = X\delta$.*

Proof. By induction on l'. The case where l' is a moderated variable $\pi' \cdot X$ follows from Lemma 24. For the other cases, we first show that the last rule used in the derivation of $\Delta \vdash s \twoheadrightarrow\!\!\!\!\!\!\!\!+\;\;\;_\mathcal{R} t$ can not be (head). Suppose otherwise. Then by the definition of rewrite relation, we have $\Delta \vdash \hat{\nabla}^{\hat{\pi}} \hat{\sigma}$ and $\Delta \vdash s \approx_\alpha \hat{l}^{\hat{\pi}} \hat{\sigma}$ for some $\hat{\pi}$, $\hat{\sigma}$ and $\hat{\nabla} \vdash \hat{l} \to \hat{r} \in \mathcal{R}$. However, by Lemma 23, this contradicts the orthogonality of $\mathcal{R}$. Hence, the last rule used in the derivation of $\Delta \vdash s \twoheadrightarrow\!\!\!\!\!\!\!\!+\;\;\;_\mathcal{R} t$ is (context). The rest of the proof is by case analysis according to the form of l'. For the details, see [15]. ◀

Now we introduce a notion of α-stability for proving Lemma 27. This notion as well as uniformity may be introduced independently from the study of confluence as a notion that yields well-behaved rewriting. Here we consider a version in which the redex position is ε.

▶ **Definition 26** (α-stability)**.** A rewrite rule $R = \nabla \vdash l \to r$ is *α-stable* if $\Delta \vdash s \approx_\alpha s'$, $\Delta \vdash s \to_{\langle R,\pi,\varepsilon,\sigma \rangle} t$ and $\Delta \vdash s' \to_{\langle R,\pi',\varepsilon,\sigma' \rangle} t'$ imply $\Delta \vdash t \approx_\alpha t'$. A rewriting system $\mathcal{R}$ is *α-stable* if so is every rewrite rule $R \in \mathcal{R}$.

The rule (Uncond-eta-exp) in Example 19 is not α-stable, since, as we saw,
$\vdash \mathtt{uc\text{-}eta\text{-}exp}(a) \to_{\langle \text{Uncond-eta-exp}, Id, \varepsilon, [] \rangle} \mathtt{lam}([a]\mathtt{app}(a,a))$ and
$\vdash \mathtt{uc\text{-}eta\text{-}exp}(a) \to_{\langle \text{Uncond-eta-exp}, (a\ b), \varepsilon, [] \rangle} \mathtt{lam}([b]\mathtt{app}(a,b))$, but not $\vdash \mathtt{lam}([a]\mathtt{app}(a,a)) \approx_\alpha \mathtt{lam}([b]\mathtt{app}(a,b))$. In the next section, we give a sufficient criterion for α-stability.

Now we show that $\twoheadrightarrow\!\!\!\!\!\!\!\!+\;\;\;_\mathcal{R}$ is strongly locally confluent modulo $\approx_\alpha$ for a class of orthogonal nominal rewriting systems.

▶ **Lemma 27** (Strong local confluence modulo $\approx_\alpha$)**.** *Let $\mathcal{R}$ be an orthogonal rewriting system that is uniform and α-stable. If $\Delta \vdash s \twoheadrightarrow\!\!\!\!\!\!\!\!+\;\;\;_\mathcal{R} t$ and $\Delta \vdash s \twoheadrightarrow\!\!\!\!\!\!\!\!+\;\;\;_\mathcal{R} t'$ then there exist u and u' such that $\Delta \vdash t \twoheadrightarrow\!\!\!\!\!\!\!\!+\;\;\;_\mathcal{R} u$, $\Delta \vdash t' \twoheadrightarrow\!\!\!\!\!\!\!\!+\;\;\;_\mathcal{R} u'$ and $\Delta \vdash u \approx_\alpha u'$.*

Proof. By induction on s. We distinguish cases according to the last rules used in the derivations of $\Delta \vdash s \twoheadrightarrow\!\!\!\!\!\!\!\!+\;\;\;_\mathcal{R} t$ and $\Delta \vdash s \twoheadrightarrow\!\!\!\!\!\!\!\!+\;\;\;_\mathcal{R} t'$.

1. Both rules are (head). If they are by the same rewrite rule $R \in \mathcal{R}$, then we use the α-stability of $\mathcal{R}$. Otherwise, this case contradicts the orthogonality of $\mathcal{R}$.

2. Both rules are (context). The claim follows from the induction hypothesis.
3. One is (context) and the other is (head). It can be shown that the claim holds by using Lemma 24 or Lemma 25. For the details, see [15].

◀

We are now ready to show that $\to_{\mathcal{R}}$ is Church-Rosser modulo $\approx_\alpha$.

▶ **Theorem 28** (Church-Rosser modulo $\approx_\alpha$)**.** *Let $\mathcal{R}$ be an orthogonal nominal rewriting system that is uniform and α-stable. Then, $\to_{\mathcal{R}}$ is Church-Rosser modulo $\approx_\alpha$.*

Proof. By Lemma 22, $\twoheadrightarrow\!\!\!\!\!\!+\;_{\mathcal{R}}$ is strongly compatible with $\approx_\alpha$, and by Lemma 27, $\twoheadrightarrow\!\!\!\!\!\!+\;_{\mathcal{R}}$ is strongly locally confluent modulo $\approx_\alpha$. Hence by the results in [10], $\twoheadrightarrow\!\!\!\!\!\!+\;_{\mathcal{R}}$ is Church-Rosser modulo $\approx_\alpha$. Since $\to_{\mathcal{R}} \subseteq \twoheadrightarrow\!\!\!\!\!\!+\;_{\mathcal{R}} \subseteq \to^*_{\mathcal{R}}$ by Lemma 21, $\to_{\mathcal{R}}$ is Church-Rosser modulo $\approx_\alpha$. ◀

4 Criterion for uniformity and α-stability

In this section, we consider a sufficient criterion for uniform and α-stable nominal rewriting systems. The main reason why a rewrite rule R does not keep α-stability is that some free atom occurring in a term is bounded through a rewrite step by R. However, such irrelevant rewrite steps can be avoided by adding an appropriate constraint to the freshness context of R. We introduce the notion of abstract skeleton preserving for characterising this constraint and show it gives a sufficient criterion for uniformity and α-stability.

Throughout this section, different meta-variables for atoms may denote the same atom.

4.1 Abstract skeleton

The abstract skeleton of a nominal term is defined as a subterm abstracted with the binders occurring on the path from the root position ε to the position of the subterm.

▶ **Definition 29** (Abstract skeleton)**.** For a nominal term t and a position $p \in Pos(t)$, $skel(p,t)$ is defined as follows:

$$\begin{array}{rcl} skel(\varepsilon, s) & = & s \\ skel(1q, [a]s) & = & [a]skel(q,s) \\ skel(1q, f\ s) & = & skel(q,s) \\ skel(iq, (s_1,\dots,s_n)) & = & skel(q, s_i) \end{array}$$

$skel(p,t)$ is called an *abstract skeleton* at p of t. $skel(p,t) = [a_1]\dots[a_n]s$ is *non-duplicating* if $i \neq j$ implies $a_i \neq a_j$. We define $Skel(t) = \{skel(p,t) \mid p \in Pos(t)\}$.

▶ **Example 30.** Figure 3 shows the abstract skeletons at the leaf positions of the left and the right hand sides of the rule $(\sigma_{\mathtt{1am}})$ in Example 8. ◀

For each $X \in V(t)$, we define $Skel_X(t) = \{skel(p,t) \mid p \in Pos_X(t)\}$. We also define $Skel_{\mathcal{X}}(t) = \{skel(p,t) \mid p \in Pos_{\mathcal{X}}(t)\}$, $Skel_{\mathcal{A}}(t) = \{skel(p,t) \mid p \in Pos_{\mathcal{A}}(t)\}$, and $Skel_{\mathcal{XA}}(t) = Skel_{\mathcal{X}}(t) \cup Skel_{\mathcal{A}}(t)$.

The following lemmas are useful for discussing the freshness context and the α-equivalency of the term through the decomposed parts. (For the proofs, see [15].)

▶ **Lemma 31.** $\Delta \vdash a\#t\sigma \iff \forall u \in Skel_{\mathcal{XA}}(t).\ \Delta \vdash a\#u\sigma$

▶ **Lemma 32.** $\Delta \vdash t\sigma \approx_\alpha t^\pi\rho \iff \forall u \in Skel_{\mathcal{XA}}(t).\ \Delta \vdash u\sigma \approx_\alpha u^\pi\rho$

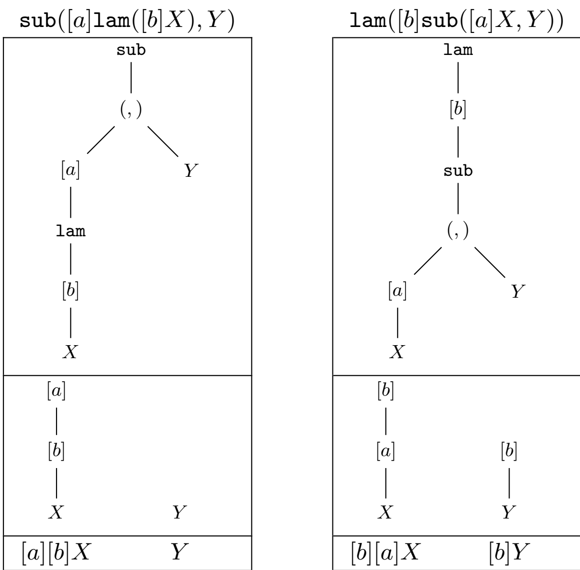


Figure 3 Abstract skeletons of $\mathtt{sub}([a]\mathtt{lam}([b]X), Y)$ and $\mathtt{lam}([b]\mathtt{sub}([a]X, Y))$.

4.2 Abstract skeleton preserving rewrite rules

In this subsection, we introduce the notions of abstract skeleton preserving rules and systems. First, we restrict rewrite rules to the following ones.

▶ **Definition 33** (Standard). A nominal rewrite rule $\nabla \vdash l \to r$ is *standard* when:
(S1) For every moderated variable $\pi{\cdot}X$ appearing in l or r, $\pi = Id$,
(S2) $FA(r) \subseteq FA(l)$,
(S3) Every abstract skeleton $[a_1]\dots[a_n]t \in \mathit{Skel}(l) \cup \mathit{Skel}(r)$ is non-duplicating.
A nominal rewriting system $\mathcal{R}$ is *standard* if so is every rewrite rule $R \in \mathcal{R}$.

All examples of rewrite rules we treated so far are standard.

We now define the abstract skeleton preserving nominal rewriting systems.

▶ **Definition 34** (Abstract skeleton preseving). A nominal rewrite rule $\nabla \vdash l \to r$ is *abstract skeleton preserving* (*ASP* for short) if it is standard and

$$\forall [a_1]\dots[a_m]X \in \mathit{Skel}_X(r).\exists [b_1]\dots[b_n]X \in \mathit{Skel}_X(l).\forall a \in ds(\{a_i\}_i, \{b_j\}_j).\ a\#X \in \nabla$$

where $ds(\{a_i\}_i, \{b_j\}_j)$ is the set of atoms such that $a \in \{a_1,\dots,a_m\}$ and $a \notin \{b_1,\dots,b_n\}$, or $a \notin \{a_1,\dots,a_m\}$ and $a \in \{b_1,\dots,b_n\}$. A nominal rewriting system $\mathcal{R}$ is *abstract skeleton preserving* (*ASP* for short) if so is every rewrite rule $R \in \mathcal{R}$.

It is easy to judge whether a standard rewrite rule is ASP or not. The rule (Uncond-eta) in Example 17 and the rule (Uncond-eta-exp) in Example 19 are not ASP. All the other rewrite rules we treated so far are ASP.

In the rest of this section, we show that the ASP property gives a sufficient criterion for the uniformity and the α-stability of nominal rewriting systems. First we prove the uniformity of ASP rewrite rules.

▶ **Lemma 35.** *An ASP rewrite rule is uniform.*

Proof. We show that for any ASP rewrite rule $R = \nabla \vdash l \to r$, if $\Delta \vdash \nabla$ and $\Delta \vdash a\#l$ then $\Delta \vdash a\#r$. Suppose $\Delta \vdash \nabla$ and $\Delta \vdash a\#l$. From the latter, we have $\forall u \in \mathit{Skel}_{\mathcal{X}\mathcal{A}}(l).\, \Delta \vdash a\#u$ by Lemma 31. Similarly, the conclusion $\Delta \vdash a\#r$ is equivalent to $\forall u \in \mathit{Skel}_{\mathcal{X}\mathcal{A}}(r).\, \Delta \vdash a\#u$. We show $\Delta \vdash a\#u$ for $u \in \mathit{Skel}_{\mathcal{A}}(r)$ and for $u \in \mathit{Skel}_{\mathcal{X}}(r)$, respectively.

1. $u \in \mathit{Skel}_{\mathcal{A}}(r)$. Then u has the form $[a_1] \dots [a_m]b$. If $a \in \{a_1, \dots, a_m\}$ or $b \neq a$, then $\Delta \vdash a\#[a_1] \dots [a_m]b$ holds. Otherwise, we have $a = b \in \mathit{FA}(r)$. Since R is standard, $a \in \mathit{FA}(l)$; contradicting $\Delta \vdash a\#l$.
2. $u \in \mathit{Skel}_{\mathcal{X}}(r)$. Then u has the form $[a_1] \dots [a_m]X$. If $a \in \{a_1, \dots, a_m\}$ then $\Delta \vdash a\#[a_1] \dots [a_m]X$ holds. Otherwise, it is enough to show $\Delta \vdash a\#X$. Since R is ASP, there exists $[b_1] \dots [b_n]X \in \mathit{Skel}_X(l)$ such that $\forall a \in \mathit{ds}(\{a_i\}_i, \{b_j\}_j).\, a\#X \in \nabla$. Since we have $\forall u \in \mathit{Skel}_{\mathcal{X}\mathcal{A}}(l).\, \Delta \vdash a\#u$, it holds that $\Delta \vdash a\#[b_1] \dots [b_n]X$. Thus, if $a \notin \{b_1, \dots, b_n\}$ then $\Delta \vdash a\#X$. If $a \in \{b_1, \dots, b_n\}$, then we have $a \in \mathit{ds}(\{a_i\}_i, \{b_j\}_j)$ because now we discuss the case of $a \notin \{a_1, \dots, a_m\}$. Hence, $a\#X \in \nabla$ holds. Since we suppose $\Delta \vdash \nabla$, $\Delta \vdash a\#X$ is obtained.

◀

4.3 α-stability of abstract skeleton preserving rewrite rules

Next, we prove the α-stability of ASP rewrite rules. For this, we need to derive α-equivalence of respective reducts s' and t' of terms s, t, from α-equivalence of s and t. The idea is to use Lemma 32 and infer α-equivalence via abstract skeletons. Recall abstract skeletons of the rule $(\sigma_{\mathtt{lam}})$ in Example 30. Here, an abstract skeleton $[a][b]X$ in LHS changes to $[b][a]X$ in RHS. Thus, $[a][b]X \approx_\alpha [c][d]X'$ should imply $[b][a]X \approx_\alpha [d][c]X'$. But generally, this is not true; for example, we have $\vdash [a][a]a \approx_\alpha [a][b]b$ but $\not\vdash [a][a]a \approx_\alpha [b][a]b$. This can be guaranteed, however, for non-duplicating skeletons (cf. Lemma 36). Another abstract skeleton Y in LHS changes to $[b]Y$ in RHS in Example 30. Again, $\vdash Y \approx_\alpha Y'$ does not imply $\vdash [b]Y \approx_\alpha [b]Y'$ in general. Fortunately, the freshness constraint of the rule $(\sigma_{\mathtt{lam}})$ contains $b\#Y$. Thus, it suffices to guarantee $b\#Y, b\#Y' \vdash Y \approx_\alpha Y'$ implies $b\#Y, b\#Y' \vdash [b]Y \approx_\alpha [b]Y'$, which is indeed the case (cf. Lemma 37). The proofs of the following lemmas are found in [15].

▶ **Lemma 36.** *Let two terms $[a_1] \dots [a_n]s$ and $[b_1] \dots [b_n]t$ be both non-duplicating. Then,*

$$\begin{aligned} & \Delta \vdash [a_1] \dots [a_i] \dots [a_j] \dots [a_n]s \approx_\alpha [b_1] \dots [b_i] \dots [b_j] \dots [b_n]t \\ \Longrightarrow\ & \Delta \vdash [a_1] \dots [a_j] \dots [a_i] \dots [a_n]s \approx_\alpha [b_1] \dots [b_j] \dots [b_i] \dots [b_n]t \end{aligned}$$

▶ **Lemma 37.** *Let two terms $[a_1] \dots [a_n]s$ and $[b_1] \dots [b_n]t$ be both non-duplicating, and let $\Delta \vdash a_i\#s, b_i\#t$. Then,*

$$\begin{aligned} & \Delta \vdash [a_1] \dots [a_{i-1}][a_i][a_{i+1}] \dots [a_n]s \approx_\alpha [b_1] \dots [b_{i-1}][b_i][b_{i+1}] \dots [b_n]t \\ \Longleftrightarrow\ & \Delta \vdash [a_1] \dots [a_{i-1}][a_{i+1}] \dots [a_n]s \approx_\alpha [b_1] \dots [b_{i-1}][b_{i+1}] \dots [b_n]t \end{aligned}$$

▶ **Lemma 38.** *If $\Delta \vdash t\sigma \approx_\alpha t^\pi \rho$ then $\forall a \in \mathit{FA}(t).\, a = \pi{\cdot}a$.*

▶ **Theorem 39.** *ASP nominal rewriting systems are uniform and α-stable.*

Proof. We show that if $\mathcal{R}$ is ASP then every $R = \nabla \vdash l \to r \in \mathcal{R}$ is α-stable, that is,

$$\Delta \vdash s \approx_\alpha \hat{s} \wedge \Delta \vdash s \to_{\langle R, \pi, \varepsilon, \sigma \rangle} t \wedge \Delta \vdash \hat{s} \to_{\langle R, \hat{\pi}, \varepsilon, \hat{\sigma} \rangle} \hat{t} \implies \Delta \vdash t \approx_\alpha \hat{t}.$$

Considering R^π as R and $\hat{\pi} \circ \pi^{-1}$ as $\hat{\pi}$, we can take $\pi = \mathit{Id}$ without loss of generality. (Note that if R is ASP then so is R^π.) Thus from here on we take Id as $\hat{\pi}$. From the definition of

the rewrite relation,

$$\begin{array}{lcl} \Delta \vdash s \to_{\langle R,\pi,\epsilon,\sigma\rangle} t & \Longleftrightarrow & \Delta \vdash \nabla^\pi\sigma,\ \Delta \vdash s \approx_\alpha l^\pi\sigma,\ t = r^\pi\sigma \\ \Delta \vdash \hat{s} \to_{\langle R,Id,\epsilon,\hat{\sigma}\rangle} \hat{t} & \Longleftrightarrow & \Delta \vdash \nabla\hat{\sigma},\ \Delta \vdash \hat{s} \approx_\alpha l\hat{\sigma},\ \hat{t} = r\hat{\sigma} \end{array}$$

From the assumption and the transitivity, we have $\Delta \vdash \nabla^\pi\sigma$, $\Delta \vdash \nabla\hat{\sigma}$ and $\Delta \vdash l\hat{\sigma} \approx_\alpha l^\pi\sigma$. Now our aim is to show $\Delta \vdash r\hat{\sigma} \approx_\alpha r^\pi\sigma$. Here, we have

$$\begin{array}{lcll} \Delta \vdash l\hat{\sigma} \approx_\alpha l^\pi\sigma & \Longleftrightarrow & \forall u \in \mathit{Skel}_{\mathcal{XA}}(l).\ \Delta \vdash u\hat{\sigma} \approx_\alpha u^\pi\sigma \text{ (from Lemma 32)} & (1) \\ \Delta \vdash r\hat{\sigma} \approx_\alpha r^\pi\sigma & \Longleftrightarrow & \forall v \in \mathit{Skel}_{\mathcal{XA}}(r).\ \Delta \vdash v\hat{\sigma} \approx_\alpha v^\pi\sigma \text{ (from Lemma 32)} & \end{array}$$

We show $\Delta \vdash v\hat{\sigma} \approx_\alpha v^\pi\sigma$ for $v \in \mathit{Skel}_{\mathcal{A}}(r)$ and for $v \in \mathit{Skel}_{\mathcal{X}}(r)$, respectively.

1. $v \in \mathit{Skel}_{\mathcal{A}}(r)$. Then v has the form $[a_1]\ldots[a_m]b$.
   a. $b \in \{a_1,\ldots,a_m\}$. First we show $\Delta \vdash [a_i]a_i \approx_\alpha [\pi{\cdot}a_i]\pi{\cdot}a_i$. It is clear when $a_i = \pi{\cdot}a_i$. When $a_i \neq \pi{\cdot}a_i$, from $\Delta \vdash \pi{\cdot}a_i \# a_i$ and $\Delta \vdash (a_i\ \pi{\cdot}a_i){\cdot}a_i \approx_\alpha \pi{\cdot}a_i$ it follows. Moreover, for $a_j (\neq a_i)$, $\Delta \vdash a_j \# a_i$ and $\Delta \vdash \pi{\cdot}a_j \# \pi{\cdot}a_i$ hold. Since v and $\pi{\cdot}v$ are non-duplicating, applying Lemma 37 to $\Delta \vdash [a_i]a_i \approx_\alpha [\pi{\cdot}a_i]\pi{\cdot}a_i$ repeatedly, we obtain
   $\Delta \vdash [a_1]\ldots[a_{i-1}][a_i][a_{i+1}]\ldots[a_m]a_i \approx_\alpha [\pi{\cdot}a_1]\ldots[\pi{\cdot}a_{i-1}][\pi{\cdot}a_i][\pi{\cdot}a_{i+1}]\ldots[\pi{\cdot}a_m]\pi{\cdot}a_i$
   Thus $\Delta \vdash v\hat{\sigma} \approx_\alpha v^\pi\sigma$ follows.
   b. $b \notin \{a_1,\ldots,a_m\}$. It is clear that $b \in \mathit{FA}(r)$. Since R is standard, $b \in \mathit{FA}(l)$. From $\Delta \vdash l\hat{\sigma} \approx_\alpha l^\pi\sigma$ and Lemma 38 it holds that $b = \pi{\cdot}b$. Thus, $\Delta \vdash b \approx_\alpha \pi{\cdot}b$. Moreover, $\Delta \vdash a_j \# b$ and $\Delta \vdash \pi{\cdot}a_j \# \pi{\cdot}b$ for every a_j. Since v and $\pi{\cdot}v$ are non-duplicating, applying Lemma 37 to $\Delta \vdash b \approx_\alpha \pi{\cdot}b$ repeatedly, we obtain $\Delta \vdash v\hat{\sigma} \approx_\alpha v^\pi\sigma$.
2. $v \in \mathit{Skel}_{\mathcal{X}}(r)$. Then v has the form $[a_1]\ldots[a_m]X$. Since R is ASP, there exists $[b_1]\ldots[b_n]X \in \mathit{Skel}_X(l)$ such that $\forall a \in \mathit{ds}(\{a_i\}_i, \{b_j\}_j).\ a\#X \in \nabla$. By (1), we have $\Delta \vdash ([b_1]\ldots[b_n]X)\hat{\sigma} \approx_\alpha ([b_1]\ldots[b_n]X)^\pi\sigma$, that is, $\Delta \vdash [b_1]\ldots[b_n]X\hat{\sigma} \approx_\alpha [\pi{\cdot}b_1]\ldots[\pi{\cdot}b_n]X\sigma$. Now, let $\{c_1,\ldots,c_k\} = \{a_1,\ldots,a_m\} \cap \{b_1,\ldots,b_n\}$. Then $\forall a \in \mathit{ds}(\{b_j\}_j, \{c_h\}_h).a\#X \in \nabla$ and $\forall a \in \mathit{ds}(\{\pi{\cdot}b_j\}_j, \{\pi{\cdot}c_h\}_h).\ a\#X \in \nabla^\pi$. From $\Delta \vdash \nabla\hat{\sigma}$ and $\Delta \vdash \nabla^\pi\sigma$, we obtain $\forall a \in \mathit{ds}(\{b_j\}_j, \{c_h\}_h).\ \Delta \vdash a\#X\hat{\sigma}$ and $\forall a \in \mathit{ds}(\{\pi{\cdot}b_j\}_j, \{\pi{\cdot}c_h\}_h).\ \Delta \vdash a\#X\sigma$. Similarly, $\forall a \in \mathit{ds}(\{a_i\}_i, \{c_h\}_h).\ \Delta \vdash a\#X\hat{\sigma}$ and $\forall a \in \mathit{ds}(\{\pi{\cdot}a_i\}_i, \{\pi{\cdot}c_h\}_h).\ \Delta \vdash a\#X\sigma$. Therefore,

$$\begin{array}{ll} & \Delta \vdash [b_1]\ldots[b_n]X\hat{\sigma} \approx_\alpha [\pi{\cdot}b_1]\ldots[\pi{\cdot}b_n]X\sigma \\ \Longleftrightarrow & \Delta \vdash [c_1]\ldots[c_k]X\hat{\sigma} \approx_\alpha [\pi{\cdot}c_1]\ldots[\pi{\cdot}c_k]X\sigma \text{ (from Lemmas 36 and 37)} \\ \Longleftrightarrow & \Delta \vdash [a_1]\ldots[a_m]X\hat{\sigma} \approx_\alpha [\pi{\cdot}a_1]\ldots[\pi{\cdot}a_m]X\sigma \text{ (from Lemmas 36 and 37)} \\ \Longleftrightarrow & \Delta \vdash v\hat{\sigma} \approx_\alpha v^\pi\sigma \end{array}$$

◀

By Theorems 28 and 39, we have the following corollary.

▶ **Corollary 40.** *Let $\mathcal{R}$ be an orthogonal nominal rewriting system that is ASP. Then, $\to_{\mathcal{R}}$ is Church-Rosser modulo $\approx_\alpha$.*

▶ **Example 41.** The rewriting system $\mathcal{R}_\sigma$ in Example 8 is left-linear and has no proper overlaps, and hence orthogonal. Moreover, all its rewrite rules are ASP. Hence, $\to_{\mathcal{R}_\sigma}$ is Church-Rosser modulo $\approx_\alpha$ by Corollary 40. ◀

5 Implementation and Experiments

We have implemented a confluence prover for NRSs proving that input NRSs are CR modulo $\approx_\alpha$, based on Corollary 40. We note that recently some confluence provers for TRSs and CTRSs are emerged (e.g. [1, 20, 14]) and the competition of confluence provers have been

Table 1 Summary of experiments.

	NRS	LL	non-PO	ASP	result	time (ms)
1	$\mathcal{R}_\sigma$	✓	✓	✓	CR	3
2	NNF of f.o.-formulas without DNE	✓	✓	✓	CR	<1
3	$\mathcal{R}_{\mathsf{uc}\text{-}\eta\text{-}\mathsf{exp}}$	✓	✓	×	Failure	<1
4	PNF of f.o.-formulas (Example 44 [3])	✓	×	✓	Failure	20
5	NNF of f.o.-formulas	✓	×	✓	Failure	<1
6	$\{\mathsf{Beta}\} \cup \mathcal{R}_\sigma$	✓	×	✓	Failure	8
7	$\{\mathsf{Beta}\} \cup \{\mathsf{Eta}\} \cup \mathcal{R}_\sigma$	✓	×	✓	Failure	3
8	β-reduction (Example 43 [3])	✓	×	✓	Failure	29
9	η-expansion (Introduction [3])	✓	✓	✓	CR	<1
10	structural substitution for $\lambda\mu$-term ([12])	✓	✓	✓	CR	17
11	fragment of ML ([3])	✓	×	✓	Failure	152
12	$\{a\#X \vdash f(X) \to [a]X\}$	✓	✓	✓	CR	<1
13	$\{\vdash f(X) \to [a]X\}$	✓	✓	×	Failure	<1
14	$\{a\#X \vdash X \to [a]X\}$ (Proof of Lemma 56 [3])	✓	✓	✓	CR	<1
15	Non-joinable trivial critical pair (Proof of Lemma 56 [3])	✓	✓	×	Failure	<1
16	PNF of f.o.-formulas with additional rules (Example 44 [3])	✓	×	✓	Failure	33
17	Substitution for λ-term (Example 43 [3])	✓	×	✓	Failure	27
18	$\{\mathsf{Eta}\}$	✓	✓	✓	CR	<1
19	$\mathcal{R}_{\mathsf{uc}\text{-}\eta}$	✓	✓	×	Failure	<1

held annually[3]. In contrast, no confluence provers for NRSs has been known previously, up to our knowledge.

In order to prove confluence of an NRS $\mathcal{R}$ based on Corollary 40, we have to show that (1) $\mathcal{R}$ is orthogonal and (2) $\mathcal{R}$ is abstract skeleton preserving (ASP). It is straightforward to check (2), as the standardness is just a syntactical restriction and $\nabla \vdash a\#X$ is easily checked for any freshness constraint ∇, $a \in \mathcal{A}$ and $X \in \mathcal{X}$. For (1), one has to check (1-a) left-linearity and that (1-b) there's no proper overlaps. The checking of (1-a) is easy. For (1-b), we have to check whether $\nabla_1 \cup \nabla_2^{\pi_2} \cup \{l_1 \approx l_2^{\pi_2}|_p\}$ is unifiable for some permutation π_2, for given $\nabla_1, \nabla_2, l_1, l_2|_p$—this problem is different from nominal unification problems as π_2 is not fixed. Fortunately, the problem can be directly reduced to a problem of *equivariant unification* [2], which has been known to be decidable. From the equivariant unification algorithm in [2], we obtain a constraint of π_2 for unifiability, if the problem is equivariantly unifiable. Our system reports concrete critical pairs generated from this constraint, if there is a proper overlap.

We have tested our confluence prover with 19 NRSs, collected from the literature, and constructed during our study. The summary of our experiments is shown in Table 1. The columns below 'NRS', 'LL', 'non-PO', 'ASP' , 'result' 'time (ms)' show the input NRS, left-linearity, non-existence of proper overlaps, ASP, the result of the confluence prover and

[3] Confluence Competition (CoCo) `http://coco.nue.riec.tohoku.ac.jp/`

execution time in millisecond, respectively. Here, PNF (NNF) denotes rules for computing prenex normal forms (resp. negation normal forms), and DNE denotes double negation elimination (`not (not X)` $\to$ `X`). The symbol '✓' denotes that the property holds, and the symbol '×' denotes that the property does not hold, which have been checked by the prover. Among 19 examples, our prover succeeded in proving confluence of 7 examples. All tests have been performed in a PC equipped with Intel Core i7-4600U processors of 2.1GHz and a memory of 8GB.

All details of the experiments are available on the webpage `http://www.nue.riec.tohoku.ac.jp/tools/experiments/rta15nrs/`.

6 Conclusion

Using our notion of rewrite relation with a permutation as a parameter, we have presented a proof of Church-Rosser modulo $\approx_\alpha$ for the class of orthogonal nominal rewriting systems that are uniform and α-stable. Moreover, we have introduced a notion of abstract skeleton preserving as a sufficient criterion for uniformity and α-stability. We have also implemented a confluence prover based on our result on Church-Rosser modulo $\approx_\alpha$ for abstract skeleton preserving rewriting systems.

As continuations of this work, we are going to study confluence of nominal rewriting systems with proper overlaps in both terminating and non-terminating cases. In such studies, it will be necessary to investigate joinability check of critical pairs with permutation variables. This is left as future work.

Acknowledgements. We would like to thank the anonymous referees for useful comments. This research was supported by JSPS KAKENHI Grant Numbers 25330004, 25280025 and 15K00003.

References

1. T. Aoto, Y. Yoshida, and Y. Toyama. Proving confluence of term rewriting systems automatically. In *Proceedings of RTA'09*, LNCS 5595, pages 93–102. Springer-Verlag, 2009.
2. J. Cheney. Equivariant unification. *Journal of Automated Reasoning*, 45:267–300, 2010.
3. M. Fernández and M. J. Gabbay. Nominal rewriting. *Information and Computation*, 205:917–965, 2007.
4. M. Fernández and M. J. Gabbay. Closed nominal rewriting and efficiently computable nominal algebra equality. In *Proceedings of LFMTP'10*, EPTCS 34, pages 37–51, 2010.
5. M. Fernández, M. J. Gabbay, and I. Mackie. Nominal rewriting systems. In *Proceedings of PPDP'04*, pages 108–119. ACM Press, 2004.
6. M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13:341–363, 2002.
7. D. Kesner. Confluence. Course material, `http://www.pps.univ-paris-diderot.fr/~kesner/enseignement/master1/semantique/ConfluenceSP-4.pdf`.
8. J. W. Klop, V. van Oostrom, and F. van Raamsdonk. Combinatory reduction systems: introduction and survey. *Theoretical Computer Science*, 121:279–308, 1993.
9. R. Mayr and T. Nipkow. Higher-order rewrite systems and their confluence. *Theoretical Computer Science*, 192:3–29, 1998.
10. E. Ohlebusch. Church-Rosser theorems for abstract reduction modulo an equivalence relation. In *Proceedings of RTA'98*, LNCS 1379, pages 17–31. Springer-Verlag, 1998.
11. A. C. R. Oliveira and M. Ayala-Rincón. Formalizing the confluence of orthogonal rewriting systems. In *Proceedings of LSFA'12*, EPTCS 113, pages 145–152, 2012.

12 M. Parigot. $\lambda\mu$-calculus: an algorithmic interpretation of classical natural deduction. In *Proceedings of LPAR'92*, LNAI 624, pages 190–201. Springer-Verlag, 1992.

13 A. M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 186:165–193, 2003.

14 T. Sternagel and A. Middeldorp. Conditional confluence (system description). In *Proceedings of Joint RTA and TLCA'14*, LNCS 8560, pages 456–465. Springer-Verlag, 2014.

15 T. Suzuki, K. Kikuchi, T. Aoto, and Y. Toyama. Confluence of orthogonal nominal rewriting systems revisited. `http://www.nue.riec.tohoku.ac.jp/user/kentaro/cr-nominal/`.

16 Terese. *Term Rewriting Systems*. Cambridge University Press, 2003.

17 C. Urban, A. M. Pitts, and M. J. Gabbay. Nominal unification. *Theoretical Computer Science*, 323:473–497, 2004.

18 R. Vestergaard and J. Brotherston. A formalised first-order confluence proof for the λ-calculus using one-sorted variable names. In *Proceedings of RTA'01*, LNCS 2051, pages 306–321. Springer-Verlag, 2001.

19 R. Vestergaard and J. Brotherston. A formalised first-order confluence proof for the λ-calculus using one-sorted variable names. *Information and Computation*, 183:212–244, 2003.

20 H. Zankl, B. Felgenhauer, and A. Middeldorp. CSI – A confluence tool. In *Proceedings of CADE'11*, LNAI 6803, pages 499–505. Springer-Verlag, 2011.

Matrix Interpretations on Polyhedral Domains

Johannes Waldmann

F-IMN, HTWK Leipzig, Germany
`johannes.waldmann@htwk-leipzig.de`

Abstract

We refine matrix interpretations for proving termination and complexity bounds of term rewrite systems we restricting them to domains that satisfy a system of linear inequalities. Admissibility of such a restriction is shown by certificates whose validity can be expressed as a constraint program. This refinement is orthogonal to other features of matrix interpretations (complexity bounds, dependency pairs), but can be used to improve complexity bounds, and we discuss its relation with the usable rules criterion. We present an implementation and experiments.

1998 ACM Subject Classification F.3.1 Specifying and Verifying and Reasoning about Programs

Keywords and phrases Termination of term rewriting, matrix interpretations, constraint programming, linear inequalities

Digital Object Identifier 10.4230/LIPIcs.RTA.2015.318

1 Introduction

To prove termination of a rewrite system, we can give an interpretation of function symbols that defines a well-founded monotone algebra that is compatible with the rules [30]. By restricting the class of interpretations, we get termination proof methods that can be automated, in the sense that the interpretation is determined by a finite set of parameters that can be computed by a program. An early instance is polynomial interpretations [7], where the parameters are the coefficients of polynomials. We are concerned here with vector-valued interpretations, where the parameters are coefficients (for the matrix representations) of multi-linear functions [13, 11]. The domain for these interpretations is $\mathbb{N}^d$, ordered by $x > y \quad \text{iff} \quad x_1 > y_1 \wedge x_2 \geq y_2 \wedge \ldots \wedge x_d \geq y_d$. This order is non-total, and the method can prove non-simple termination.

Several variants and modifications have been investigated, and we list some that are relevant for the present investigation:

- By restricting the shape of matrices, we can prove not just termination, but polynomial derivational complexity [18, 28].
- Vector-valued interpretations can be used to define reduction pairs for termination proofs in the dependency pair framework [2]. The main point here is that monotonicity constraints can be relaxed.
- Instead of vectors and linear functions over $(\mathbb{N}, +, \cdot)$, we can take vectors and linear functions over other semirings, e.g., the arctic semiring $(\mathbb{N} \cup \{-\infty\}, \max, +)$ [14]. In this semiring, monotonicity of operations is different (from $\mathbb{N}$), in a way that this is well-suited to the dependency pair framework.
- Returning to $\mathbb{N}^d$, we may consider different orders on that domain [19].

In the present paper, we modify matrix interpretations in yet another way: we keep the semiring ($\mathbb{N}$) and order $>$ on $\mathbb{N}^d$, but restrict the domain of interpretations by additional linear inequalities. We obtain a convex polyhedral domain $D \subseteq \mathbb{N}^d$. Using such domains is a

© Johannes Waldmann;
licensed under Creative Commons License CC-BY
26th International Conference on Rewriting Techniques and Applications (RTA'15).
Editor: Maribel Fernández; pp. 318–333
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

standard approach in static analysis of imperative programs that is routinely used in optimizing compilers (cf. the Parma Polyhedral Library [3] for the GNU Compiler Collection). In the context of automated termination analysis of rewrite systems, polyhedral domains were first suggested by Lucas and Meseguer [17]. In the present paper, we substantially extend their idea.

Let us illustrate the method by an example.

▶ **Example 1.** The goal is to prove termination and polynomial derivational complexity of the string rewriting system

$$R = \{fg \to ff, gf \to gg\},$$

where the string fg is an abbreviation for the term $f(g(x))$, etc. We define a domain $D = \{(x_1, x_2, x_3) \in \mathbb{N}^3 \mid x_3 \geq x_2 + 1\}$. This set is non-empty, e.g., $(0, 0, 1) \in D$. Then both

$$\begin{aligned} [f](x_1, x_2, x_3) &= (x_1 + 2x_2 + 1, 0, x_3 + 1) \\ [g](x_1, x_2, x_3) &= (x_1 \qquad\qquad , x_3, x_3 + 1) \end{aligned}$$

map D into D. Now we combine interpretations:

$$\begin{aligned} [fg](x) &= (x_1 + \qquad 2x_3 + 1, 0, x_3 + 2), \\ [ff](x) &= (x_1 + 2x_2 + \qquad 2, 0, x_3 + 2). \end{aligned}$$

The point is now that $\forall x \in D : [fg](x) > [ff](x)$ even though we don't have a point-wise inequality between corresponding coefficients in the first component: the coefficient of x_2 in $[fg](x)_1$ is zero, and the coefficient of x_2 in $[ff](x)_1$ is two.

By the condition that defines D, we have

$$[fg](x)_1 \geq x_1 + 2x_3 + 1 \geq x_1 + (2x_2 + 2) + 1 > x_1 + 2x_2 + 2 = [ff](x)_1.$$

Additionally, we verify (without using D conditions)

$$[gf](x) = (x_1 + 2x_2 + 1, x_3 + 1, x_3 + 2) > (x_1, x_3 + 1, x_3 + 2) = [gg](x).$$

We also note that $[f]$ and $[g]$ are strictly monotone w.r.t. $>$, since all coefficients are non-negative, and the coefficient of x_1 in the first component is positive.

This proves that $[\cdot]$ is a strictly monotone D-valued interpretation that is strictly compatible with the rewrite system R. So, the interpretation certifies termination of R [11].

Moreover, the coefficient matrices of $[\cdot]$ are upper triangular, so we actually proved polynomial derivational complexity [18]. By closer inspection (there are just two occurrences of 1 on the main diagonals) the complexity is quadratic. This property of R was known before, e.g., CaT [15] proves it via root labelling [22]. Ours seems to be the first "direct" proof.

In the remainder of the paper, we formally define and justify the method (Sections 3 and 4), discuss modifications with respect to derivational complexity (Section 6) and the dependency pair method (Sections 7) with the usable-rules criterion (Section 8) and finally (Section 9) describe an implementation and experiments.

2 Notation and Preliminaries

A ranked signature maps function symbols to arities, e.g., $\Sigma = \{(a,2),(f,1),(g,1)\}$. The size $\|\Sigma\|$ of a signature is $\sum_{(f,k)\in\Sigma)} k$, e.g., $\|\Sigma\| = 4$. We consider terms in $\mathsf{Term}(\Sigma, V)$ with symbols from Σ and variables from V. We denote by $\mathsf{Var}(t)$ the set of variables appearing in a term t, and by $|t|$ the size of the term (the number of its positions). A rewrite rule is a pair $(l,r) \in \mathsf{Term}(\Sigma, V)^2$, written $l \to r$, and a set R of rewrite rules defines a relation $\to_R$ on $\mathsf{Term}(\Sigma)$ in the usual way. We write $\to_1 \circ \to_2$ for the product of relations $\to_1$ and $\to_2$. For a relation $\to$, we denote by $\to^k$ its k-fold product, by $\to^+$ its transitive closure, and by $\to^*$ its transitive reflexive closure. We write $\to_1 / \to_2$ for the relation $\to_2^* \circ \to_1 \circ \to_2^*$. We say a relation $\to$ is well-founded if there is no infinite $\to$-chain. A rewrite system R is called terminating if $\to_R$ is well-founded. A rewrite system R is called terminating relative to a rewrite system S if $\to_R / \to_S$ is well-founded. The derivational complexity $\mathsf{dc}_\to$ of a relation $\to$ on terms describes the length of $\to$-chains as a function of the size of the start term. Formally, $\mathsf{dc}_\to : \mathbb{N} \to \mathbb{N} \cup \{\infty\}$ is the function $s \mapsto \sup\{k \mid \exists t_1, t_2 \in \mathsf{Term}(\Sigma) : |t_1| \leq s \wedge t_1 \to^k t_2\}$. This is in fact a function $\mathbb{N} \to \mathbb{N}$ in case $\to$ is terminating and finitely branching. We write dc_R for $\mathsf{dc}_{\to_R}$ and $\mathsf{dc}_{R/S}$ for $\mathsf{dc}_{\to_R/\to_S}$.

An algebra A for signature Σ is given by a domain D_A, and for each k-ary symbol f from Σ, a k-ary function $[f]_A : D_A^k \to D_A$. The algebra then maps each $t \in \mathsf{Term}(\Sigma)$ to an element of D_A, also denoted $[t]_A$, and by extension, each $t \in \mathsf{Term}(\Sigma, V)$ with $|Var(t)| = k$, to a k-ary function $[t]_A : D_A^k \to D_A$. An algebra is monotone w.r.t. an order $>_A$ on its domain if each $[f]_A$ is monotone in each argument: $x_i >_A x_i'$ implies $[f]_A(x_1, \ldots, x_i, \ldots, x_k) >_A [f]_A(x_1, \ldots, x_i', \ldots, x_k)$. An algebra with order $>_A$ is well-founded if $>_A$ is well-founded. A Σ-algebra A is compatible with relation $\to$ if $x \to y$ implies $[x]_A > [y]_A$. If A is clear from the context, we write $[t]$ for $[t]_A$, and $>$ for $>_A$, etc.

A d-dimensional matrix interpretation defines an algebra with domain $\mathbb{N}^d$, the order is given by $x > y$ iff $x_1 > y_1 \wedge x_2 \geq y_2 \wedge \ldots \wedge x_d \geq y_d$, and the interpretation of a k-ary symbol f is given by a multi-linear function of shape $[f](x_1, \ldots, x_k) = F_0 + \sum_i F_i x_i$, where F_0 is a vector (the absolute part), and $F_1, \ldots, F_k$ are matrices (the coefficients for the linear part). Because of this presentation, we think of vectors $x_1, \ldots, x_k, F_0$ as column vectors. All coefficients in $F_0, F_1, \ldots, F_k$ are nonnegative (because $[f]$ must map into $\mathbb{N}^d$). A matrix interpretation is monotone w.r.t. $>$ if each top left entry of $F_1, \ldots, F_k$ is positive. A matrix interpretation is strictly (weakly) compatible with a rule (l,r) with $|\mathsf{Var}(l) \cup \mathsf{Var}(r)| = k$ if the interpretations $[l]$ and $[r]$, which can be written as $[l] = F_0 + \sum_i F_i x_i$, $[r] = G_0 + \sum_i G_i x_i$, verify $F_0 > G_0$ ($F_0 \geq G_0$) and for all $1 \leq i \leq k$, $F_i \geq G_i$ (here, $\geq$ on matrices is the point-wise extension of $\geq$ on $\mathbb{N}$).

▶ **Example 2.** For $\{f(g(x)) \to f(f(x)), g(f(x)) \to g(g(x))\}$, the interpretation

$$[f](x_1) = \begin{pmatrix}0\\0\\1\end{pmatrix} + \begin{pmatrix}1&1&0\\0&0&0\\0&1&1\end{pmatrix} \cdot x_1, \qquad [g](x_1) = \begin{pmatrix}1\\1\\0\end{pmatrix} + \begin{pmatrix}1&0&2\\0&1&1\\0&0&0\end{pmatrix} \cdot x_1$$

is monotone and compatible with the rules, since

$$[fg](x_1) = \begin{pmatrix}2\\0\\2\end{pmatrix} + \begin{pmatrix}1&1&3\\0&0&0\\0&1&1\end{pmatrix} \cdot x_1 \;>\; [ff](x_1) = \begin{pmatrix}0\\0\\2\end{pmatrix} + \begin{pmatrix}1&1&0\\0&0&0\\0&1&1\end{pmatrix} \cdot x_1$$

$$[gf](x_1) = \begin{pmatrix}3\\2\\0\end{pmatrix} + \begin{pmatrix}1&3&2\\0&1&1\\0&0&0\end{pmatrix} \cdot x_1 \;>\; [gg](x_1) = \begin{pmatrix}2\\2\\0\end{pmatrix} + \begin{pmatrix}1&0&2\\0&1&1\\0&0&0\end{pmatrix} \cdot x_1$$

We note that it is decidable whether a given d-dimensional matrix interpretation is monotone, and compatible with a given R. The decision procedure is a straight-line program (the control flow does not depend on the data), so we can derive a constraint system from it, and use it to compute a suitable interpretation, once R is given. The constraint language contains inequalities between polynomials (called `QFNIA` in [4]). Because of its intractability, one often restricts unknown numbers to finite ranges, and represents them as bit vectors (using `QFBV` in [4]), in binary, or even unary [8].

3 Interpretations on Polyhedral Domains

We now define the concepts, and illustrate them by formalizing Example 1.

▶ **Definition 3.** A *polyhedral interpretation* A with domain dimension $d \in \mathbb{N}$ and constraint dimension $c \in \mathbb{N}$ for a ranked signature Σ consists of

- (the polyhedral domain) a matrix $C_A \in \mathbb{Q}^{c\times d}$ and a vector $B_A \in \mathbb{Q}^{c\times 1}$, describing the set $D_A = \{x \mid x \in \mathbb{Q}^d, x \geq 0 \wedge C_A x + B_A \geq 0\}$
- (the underlying interpretation) for each $(f,k) \in \Sigma$, a (column) vector $F_0 \in \mathbb{N}^{d\times 1}$, and a list of k square matrices $F_1,\dots,F_k \in \mathbb{N}^{d\times d}$, describing a function $[f]_A : (\mathbb{N}^d)^k \to \mathbb{N}^d : (x_1,\dots,x_k) \mapsto F_0 + \sum_i F_i x_i$

Subscript A is omitted when it can be inferred from the context.

Note that we use rational numbers ($\mathbb{Q}$) for describing the constraints (this fits with the theory of linear algebra that we will need) but natural numbers for the interpretation (this fits with well-foundedness of the domain). In examples, and in our implementation (see Section 9), we will substitute $\mathbb{Z}$ for $\mathbb{Q}$.

▶ **Example 4** (Example 1 continued)**.** The signature is $\Sigma = \{(f,1),(g,1)\}$, the domain dimension is $d = 3$, the constraint dimension is $c = 1$, the domain is described by $C = \begin{pmatrix}0 & -1 & 1\end{pmatrix}, B = \begin{pmatrix}-1\end{pmatrix}$, and the underlying interpretation is

$$[f](x) = \begin{pmatrix}1 & 2 & 0\\ 0 & 0 & 0\\ 0 & 0 & 1\end{pmatrix} x + \begin{pmatrix}1\\0\\1\end{pmatrix}, [g](x) = \begin{pmatrix}1 & 0 & 0\\ 0 & 0 & 1\\ 0 & 0 & 1\end{pmatrix} x + \begin{pmatrix}0\\0\\1\end{pmatrix}.$$

Since we will later determine polyhedral interpretations by solving constraint systems, we collect information that helps to determine their size. In particular, we are interested in how many extra constraints we need, compared to the standard matrix method.

▶ **Observation 5.** *A domain description (in Def. 3) contains $c \cdot (d+1)$ unknowns.*

Properties of a polyhedral interpretation will be derived from the existence of a valid certificate, which contains a part that refers to the domain, and a part that refers to the rewrite system. These certificates are derived from a general principle

▶ **Lemma 6** (Inhomogenous Farkas' Lemma)**.** *([25, 26]) A linear inequality $a^T x \leq p$ is a consequence of a solvable system of inequalities $Ax \leq b$ iff there is some $y \geq 0$ with $a = A^T y$ and $y^T b \leq p$.*

In other words, the conclusion is implied by a nonnegative linear combination of the premises. Our certificates are in fact representations of the coefficients in that linear combination.

Given a polyhedral interpretation A for signature Σ, we ask for $(f,k) \in \Sigma$ whether $[f]_A : D^k \to D$. We have the representation $[f](x_1,\dots,x_k) = F_0 + \sum_i F_i x_i$, with $F_0 \geq$

$0, \dots, F_i \geq 0$, and we know that the arguments are from the domain: $\forall i : x_i \in D$, that is, $\forall i : x_i \geq 0 \wedge Cx_i + B \geq 0$. We combine all these assumptions, and collect them in a matrix (with $k \cdot (d+c)$ rows, $k \cdot d + 1$ columns)

$$\begin{pmatrix} I & 0 & \dots & 0 & 0 \\ 0 & I & & 0 & 0 \\ \vdots & & \ddots & & \\ 0 & 0 & \dots & I & 0 \\ C & 0 & \dots & 0 & B \\ 0 & C & & 0 & B \\ \vdots & & \ddots & & \\ 0 & 0 & \dots & C & B \end{pmatrix} \tag{1}$$

Do we have $[f](x_1, \dots, x_k) \in D$? In other words, do these inequalities imply $C(F_0 + \sum_i F_i x_i) + B \geq 0$? (Note that $[f](x_1, \dots, x_k) \geq 0$ is already implied by $F_i \geq 0$ and $x_i \geq 0$.) In our matrix notation, the conclusion is

$$\begin{pmatrix} CF_1 & CF_2 & \dots & CF_k & CF_0 + B \end{pmatrix}$$

with c rows, $k \cdot d + 1$ columns. For each of the c inequalities (rows) from the conclusion, Lemma 6 gives one coefficient per each of the $k \cdot (c+d)$ assumptions. In total, we have $kc(c+d)$ coefficients, and we can arrange them as matrices $V_1, \dots, V_k \in \mathbb{Q}_+^{c \times d}, W_1, \dots, W_k \in \mathbb{Q}_+^{c \times c}$ and get

$$V_1 + W_1 C = CF_1 \wedge \dots \wedge V_k + W_k C = CF_k \wedge \sum_i W_i B \leq CF_0 + B.$$

Since $V_i \geq 0$ we can simplify the equations to inequalities, obtaining

▶ **Lemma 7.** *The function $(x_1, \dots, x_k) \mapsto F_0 + \sum_i F_i x_i$ with $F_i \geq 0$ maps $D^k \to D$ if and only if there exist $W_i \in \mathbb{Q}_+^{c \times c}$ with*

$$W_1 C \leq CF_1 \wedge \dots \wedge W_k C \leq CF_k \wedge \sum_i W_i B \leq CF_0 + B.$$

Now we consider compatibility of the interpretation A with a rewrite rule (l, r) with $|\mathsf{Var}(l) \cup \mathsf{Var}(r)| = k$. Then the difference of interpretations $[l]_A - [r]_A$ is a linear function $\Delta : (x_1, \dots, x_k) \mapsto \Delta_0 + \sum_i \Delta_i x_i$. When $x_i \in D$, we want $\Delta(x_1, \dots, x_k) > 0$ or ≥ 0 (strict or weak compatibility). So, the conclusion (d rows) is

$$\begin{pmatrix} \Delta_1 & \Delta_2 & \dots & \Delta_k & \Delta_0 \end{pmatrix},$$

resp. Δ_0' in the last component, where Δ_0' is obtained from Δ_0 by decreasing the first component by 1. Again by Lemma 6, there are coefficients $T_i, U_i \in \mathbb{Q}_+^{d \times c}$ with

$$T_1 + U_1 C = \Delta_1 \wedge \dots \wedge T_k + U_k C = \Delta_k \wedge \sum_i U_i B \leq \Delta_0,$$

and we simplify (since $T_i \geq 0$) all equations to inequalities, and obtain

▶ **Lemma 8.** *A polyhedral interpretation A is strictly (weakly, respectively) compatible with a rewrite rule (l, r) with $|\mathsf{Var}(l) \cup \mathsf{Var}(r)| = k$ and $([l] - [r])(x_1, \dots, x_k) = \Delta_0 + \sum_i \Delta_i x_i$ if and only if there exist matrices $U_i \in \mathbb{Q}_+^{d \times c}$ such that*

$$U_1 C \leq \Delta_1 \wedge \dots \wedge U_k C < \Delta_k \wedge \sum_i U_i B \leq \Delta_0 \quad (\leq \Delta_0, \textit{resp.})$$

4 Certificates for Polyhedral Interpretations

We use one direction of Lemmata 7 and 8 to define certificates for properties of polyhedral interpretations.

▶ **Definition 9.** A *domain certificate* for a polyhedral interpretation consists of

- a vector $n \in \mathbb{Q}_+^d$ which is called valid if $Cn + B \geq 0$.
- for each $(f, k) \in \Sigma$ with $[f](x_1, \dots, x_k) = F_0 + \sum_i F_i x_i$, matrices $W_1, \dots W_k \in \mathbb{Q}_+^{c \times c}$ which are called valid if
  - $\forall 1 \leq i \leq k : CF_i \geq W_i C$ and $CF_0 + B \geq (\sum_i W_i)B$

▶ **Observation 10.** *The domain certificate contains $d + \|\Sigma\| \cdot c^2$ unknowns. Validity of the domain certificate can be checked with $1 + |\Sigma|$ matrix multiplications in $(c \times d) \cdot (d \times 1)$, $|\Sigma|$ matrix multiplications in $(c \times c) \cdot (c \times 1)$, $\|\Sigma\|$ matrix multiplications in $(c \times d) \cdot (d \times d)$, $|\Sigma|$ matrix multiplications in $(c \times c) \cdot (c \times d)$.*

There are also additions and comparisons, but their cost is dominated by multiplication.

▶ **Example 11** (Example 4 continued)**.** We take $n = \begin{pmatrix} 0 & 0 & 1 \end{pmatrix}^T$ which is valid since $Cn+B = \begin{pmatrix} 0 & -1 & 1 \end{pmatrix} \begin{pmatrix} 0 & 0 & 1 \end{pmatrix}^T - 1 = 0$, and for both f and g, the choice $W_1 = (0)$ is valid since $CF_1 \geq 0$ and $CG_1 \geq 0$ and $CF_0 + B = CG_0 + B = 0$.

▶ **Lemma 12.** *The following statements are equivalent:*

- *polyhedral interpretation A has a valid domain certificate,*
- *D_A is non-empty, and for each $(f, k) \in \Sigma$, the function $[f]_A$ maps D_A^k into D_A.*

Proof. This follows from Lemma 7. Additionally, we give an explicit computation that shows one direction of the equivalence: For $y = [f](x_1, \dots, x_k) = F_0 + \sum_i F_i x_i$, we have $y \in D$ by the chain of inequalities $Cy + B = C(F_0 + \sum F_i x_i) + B = CF_0 + \sum_i CF_i x_i + B \geq CF_0 + \sum_i W_i C x_i + B \geq CF_0 - \sum W_i B + B \geq 0$. ◀

▶ **Definition 13.** A *compatibility certificate* for polyhedral interpretation A w.r.t. rewriting system R contains, for each rule $(l, r) \in R$ with $|\mathsf{Var}(l) \cup \mathsf{Var}(r)| = k$ and $([l]_A - [r]_A)(x_1, \dots, x_k) = \Delta_0 + \sum_i \Delta_i x_i$, matrices $U_1, \dots, U_k \in \mathbb{Q}_+^{d \times c}$, which are called valid if

- $\forall i : \Delta_i \geq U_i C$ and
-
  - $\Delta_0 \geq \sum_i U_i B$ (then the certificate is called *weak* for that rule)
  - or $\Delta_0 > \sum_i U_i B$ (then the certificate is called *strict* for that rule)

For the following, we need notation $\|R\| = \sum\{|\mathsf{Var}(l) \cup \mathsf{Var}(r)| \mid (l, r) \in R\}$.

▶ **Observation 14.** *The compatibility certificate contains $\|R\| \cdot d \cdot c$ unknowns. Validity of the compatibility certificate can be checked with $\|R\|$ matrix multiplications in $(d \times c) \cdot (c \times d)$, and $|R|$ matrix multiplications in $(d \times c) \cdot (c \times 1)$, assuming Δ_i are already given.*

▶ **Example 15** (Example 4 continued)**.** For rule (fg, ff), we compute

$$[fg](x) = \begin{pmatrix} 1 & 0 & 2 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} x + \begin{pmatrix} 1 \\ 0 \\ 2 \end{pmatrix}, \quad [ff](x) = \begin{pmatrix} 1 & 2 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} x + \begin{pmatrix} 2 \\ 0 \\ 2 \end{pmatrix}, \quad \Delta^{(fg,ff)} = \begin{pmatrix} 0 & -2 & 2 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} x + \begin{pmatrix} -1 \\ 0 \\ 0 \end{pmatrix}.$$

A valid strict certificate for this rule is $U_1^{(fg,ff)} = \begin{pmatrix} 2 & 0 & 0 \end{pmatrix}^T$, since

$$U_1 C = \begin{pmatrix} 2 \\ 0 \\ 0 \end{pmatrix} \begin{pmatrix} 0 & -1 & 1 \end{pmatrix} = \begin{pmatrix} 0 & -2 & 2 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \leq \Delta_1, \; U_1 B = \begin{pmatrix} 2 \\ 0 \\ 0 \end{pmatrix} (-1) = \begin{pmatrix} -2 \\ 0 \\ 0 \end{pmatrix} < \Delta_0$$

For rule (gf, gg), we compute

$$[gf](x) = \begin{pmatrix}1 & 2 & 0\\0 & 0 & 1\\0 & 0 & 1\end{pmatrix} x + \begin{pmatrix}1\\1\\2\end{pmatrix}, \qquad [gg](x) = \begin{pmatrix}1 & 0 & 0\\0 & 0 & 1\\0 & 0 & 1\end{pmatrix} x + \begin{pmatrix}0\\1\\2\end{pmatrix}, \qquad \Delta^{(gf,gg)} = \begin{pmatrix}0 & 2 & 0\\0 & 0 & 0\\0 & 0 & 0\end{pmatrix} x + \begin{pmatrix}1\\0\\0\end{pmatrix}.$$

and a valid strict certificate is $U_1^{(gf,gg)} = \begin{pmatrix}0 & 0 & 0\end{pmatrix}^T$.

▶ **Lemma 16.** *These statements are equivalent:*

- *polyhedral interpretation A has a valid compatibility certificate w.r.t. rewrite system R*
- *A is strictly compatible with the rules of R for which the certificate is strict, and weakly compatible with the rules for which the certificate is weak.*

Proof. This follows from Lemma 8. Additionally, we give an explicit computation for one direction. For $(l, r) \in R$, we have $([l]_A - [r]_A)(x_1, \ldots, x_k) = \Delta_0 + \sum_i \Delta_i x_i \geq \Delta_0 + \sum_i U_i C x_i \geq \Delta_0 - \sum_i U_i B$ which is ≥ 0 or > 0. ◀

From previous observations, and assuming that matrix multiplication in $(a \times b) \cdot (b \times c)$ can be done with $O(a \cdot b \cdot c)$ elementary operations, we obtain the following, which is the basis for our implementation, see Section 9.

▶ **Theorem 17.** *The validity of the certificate of a polyhedral interpretation with domain dimension d and constraint dimension c for a rewrite system R over signature Σ is decidable. The certificate can be represented by $d + \|\Sigma\|c^2 + \|R\|cd$ unknowns and $O(\|\Sigma\|cd^2 + |\Sigma|c^2 d + \|R\|cd^2 + |R|cd)$ elementary constraints.*

5 Polyhedral Interpretations for Termination and Complexity

Polyhedral interpretations can be used for proofs of termination:

▶ **Theorem 18.** *If a polyhedral interpretation has a valid domain certificate, and a strict compatibility certificate for a rewrite system R, and a weakly compatibility certificate for a rewrite system S, and the underlying interpretation is monotone, then R is terminating relative to S.*

Proof. The polyhedral interpretation defines a well-founded monotone algebra on a subset of $(\mathbb{N}^d, >)$ that is compatible with $\to_R / \to_S$. ◀

Compared to the standard matrix method, we kept the order, restricted the domain, and changed the test for compatibility: we can now use properties of the polyhedral domain, and thus ease the requirement of comparing coefficients of $[l]_A > [r]_A$ point-wise.

▶ **Example 19.** [1] We apply the method to problem `Ex16_Luc06_C` from the TPDB [21].

[1] `http://nfa.imn.htwk-leipzig.de/termcomp/show_job_pair/41269410`

$$\begin{aligned}
active(f(X,X)) \to mark(f(a,b)), \quad active(b) \to mark(a),\\
active(f(X1,X2)) \to f(active(X1),X2), \quad f(mark(X1),X2) \to mark(f(X1,X2)),\\
proper(f(X1,X2)) \to f(proper(X1),proper(X2)),\\
proper(a) \to ok(a), \quad proper(b) \to ok(b),\\
f(ok(X1),ok(X2)) \to ok(f(X1,X2)),\\
top(mark(X)) \to top(proper(X)), \quad top(ok(X)) \to top(active(X))
\end{aligned}$$

we remove $active(b) \to mark(a)$ and then use interpretation

$$\begin{aligned}
mark \mapsto \begin{pmatrix}0\\0\end{pmatrix} + \begin{pmatrix}1 & 0\\0 & 1\end{pmatrix} \cdot x_1, f \mapsto \begin{pmatrix}1\\2\end{pmatrix} + \begin{pmatrix}1 & 3\\0 & 0\end{pmatrix} \cdot x_1 + \begin{pmatrix}1 & 0\\0 & 0\end{pmatrix} \cdot x_2,\\
a \mapsto \begin{pmatrix}1\\0\end{pmatrix}, b \mapsto \begin{pmatrix}0\\1\end{pmatrix}, ok \mapsto \begin{pmatrix}0\\0\end{pmatrix} + \begin{pmatrix}1 & 0\\0 & 1\end{pmatrix} \cdot x_1, top \mapsto \begin{pmatrix}0\\1\end{pmatrix} + \begin{pmatrix}1 & 2\\1 & 1\end{pmatrix} \cdot x_1,\\
active \mapsto \begin{pmatrix}0\\0\end{pmatrix} + \begin{pmatrix}1 & 0\\0 & 1\end{pmatrix} \cdot x_1, proper \mapsto \begin{pmatrix}0\\0\end{pmatrix} + \begin{pmatrix}1 & 0\\0 & 1\end{pmatrix} \cdot x_1
\end{aligned}$$

on a domain restricted by $\begin{pmatrix}-1\end{pmatrix} + \begin{pmatrix}1 & 1\end{pmatrix} \cdot x \geq 0$. Equivalently, $x_1 + x_2 \geq 1$, so just $(0,0)^T$ is excluded from the domain. Rule $active(f(X,X)) \to mark(f(a,b))$ is interpreted by

$$[\text{lhs}] = \begin{pmatrix}1\\2\end{pmatrix} + \begin{pmatrix}2 & 3\\0 & 0\end{pmatrix} \cdot x_1, \quad [\text{rhs}] = \begin{pmatrix}2\\2\end{pmatrix} + \begin{pmatrix}0 & 0\\0 & 0\end{pmatrix} \cdot x_1,$$

Note the absolute parts are not decreasing: $(1\ 2)^T \not\geq (2\ 2)^T$. This rule has a strong compatibility certificate $(2\ 0)^T$. In effect, we add twice the domain constraint, to prove the decrease in the first component. Starting from the right-hand side: $2 \leq 2+2(-1+x_{11}+x_{12}) = x_{11}+x_{12} < 1+2x_{11}+3x_{12}$. Then other rules can be removed by the standard matrix method.

This termination problem was solved in the 2014 competition only by AProVE [12], using back-transformation to CSR `QTRSToCSRProof`. [2]

The previous example suggests that polyhedral interpretations are strictly more powerful than standard matrix interpretations (even when these are combined with other methods), but we currently have no proof. At least we can be sure that they are not less powerful:

▶ **Observation 20.**
- *For any domain and constraint dimension: A polyhedral interpretation with domain constraint $B = 0, C = 0$ is a standard matrix interpretation.*
- *A polyhedral interpretation with constraint dimension 0 is a standard matrix interpretation.*

Proof. First part: take $n = 0, W_i = 0, U_i = 0$ and verify that the compatibility condition reduces to $\Delta_0 \geq 0 (> 0, \text{resp.})$ and $\Delta_i \geq 0$. Second part: The matrices in the domain certificate have extension 0×0, the matrices in the compatibility certificate have extension $d \times 0$, so they are zero matrices, and the first part applies. ◀

Even if we do have constraints, we can ignore them, to obtain a statement on derivational complexity. Recall that the height function dc_A of a well-founded monotone Σ-algebra A on $(D_A, >_A)$ is the function $s \mapsto \sup\{\mathsf{dc}_{>_A}([t]_A) \mid t \in \mathsf{Term}(\Sigma), |t| \leq s\}$.

[2] `http://nfa.imn.htwk-leipzig.de/termcomp/display_proof/26921465`

▶ **Lemma 21.** *If a polyhedral interpretation A is monotone and strictly compatible with a rewrite system R, then dc_R is bounded by the height dc_A of the matrix interpretation that underlies A.*

Proof. Each $\to_R$-chain is mapped to a $>$-chain in the polyhedral domain D, which is also a $>$-chain in $\mathbb{N}^d$. ◀

We mention two consequences.

The original matrix method is limited because matrix products grow at most exponentially: with matrix interpretations, it is impossible to reduce a termination problem by "removing a rule" that is used more than exponentially often. E.g., no matrix interpretation can remove a rule from $\{ab \to bca, cb \to bbc\}$ [13], and polyhedral constraints will not change that.

If we have a polyhedral interpretation A that is compatible with R and where the underlying matrix interpretation grows polynomially only, then we have a proof that dc_R is polynomially bounded. The introductory Example 1 already applies this. We will show in the next section that we can do better in some cases, by not ignoring the information in the constraints.

6 Improving Polynomial Growth Bounds

We show that polyhedral constraints can serve to lower a bound for polynomial growth of a matrix interpretation.

Recall that for each component of a vector valued interpretation we can assign a degree of growth, and the degree of the interpretation is the degree of the first component.

If the interpretation uses upper triangular matrices (of dimension d), the degree of the i-th component is at most $d+1-i$, and there is a refinement where the degree can be reduced further if all matrices have zero diagonal entries at index (i,i).

Now polyhedral constraints for triangular matrices in some cases bound the i-th component from above by some positive linear combination of components with higher indices, that is, of lower degree.

As a special case, the very last component could be bounded by a constant, as in the following example.

▶ **Example 22.** `TRS/secret06/jambox/5` [3] The rewrite system

$$\{a(a(y,0),0) \to y, c(c(y)) \to y, c(a(c(c(y)),x)) \to a(c(c(c(a(x,0)))),y)\}$$

has a compatible polyhedral interpretation that uses upper triangular matrices

$$0 \mapsto \begin{pmatrix}0\\0\\0\\0\end{pmatrix}, c \mapsto \begin{pmatrix}0\\0\\0\\2\end{pmatrix} + \begin{pmatrix}1&1&0&1\\0&1&2&0\\0&0&1&1\\0&0&0&0\end{pmatrix} \cdot x_1, a \mapsto \begin{pmatrix}1\\0\\0\\0\end{pmatrix} + \begin{pmatrix}1&0&0&0\\0&1&0&1\\0&0&1&0\\0&0&0&1\end{pmatrix} \cdot x_1 + \begin{pmatrix}1&2&2&2\\0&1&4&4\\0&0&1&1\\0&0&0&0\end{pmatrix} \cdot x_2$$

with domain restriction $\begin{pmatrix}2\end{pmatrix} + \begin{pmatrix}0&0&0&-1\end{pmatrix} \cdot x \geq 0$. The restriction means that $x_4 \leq 2$. (By inspecting the interpretation, the domain for x_4 is found to be $\{0,2\}$.) There are 4 non-zero positions on the main diagonal, so the naive degree bound is 4. The domain constraint says

[3] `http://nfa.imn.htwk-leipzig.de/termcomp/show_job_pair/40307443`

that x_4 is bounded by 2, so the degree of x_3 is at most linear, x_2 at most quadratic, and x_1 at most cubic.

The states $3, 4$ of the underlying automaton are in fact an unambiguous component, so degree 3 would have been detected by the method from [28], which is however more expensive to implement.

This example was not solved in the 2014 complexity competition.

▶ **Example 23.** For system `ExProp7_Luc06_GM` (not solved in 2014 complexity competition), we find a compatible polyhedral upper triangular interpretation [4] with constraint $\begin{pmatrix} 0 & 0 & -1 & 1 \end{pmatrix} x + 2 \geq 0$. That is $x_4 + 2 \geq x_3$, so the degree of x_3 is linear (not quadratic), and this reduces the degree estimate for the interpretation as a whole.

7 Polyhedral Constraints and the Dependency Pair Method

The dependency pair (DP) method [2] can use reduction pairs that come from matrix interpretations [11]. We briefly recap the notation. Given Σ, the marked signature $\Sigma^\#$ is $\{(f^\#, k) \mid (f, k) \in \Sigma\}$. We use sort symbols $\{O, \#\}$ and say that $(f, k) \in \Sigma$ has type $O^k \to O$, while $f^\#$ has type $O^k \to \#$. For $t = f(t_1, \ldots, t_k) \in \mathsf{Term}(\Sigma, V)$, we write $t^\#$ for $f^\#(t_1, \ldots, t_k) \in \mathsf{Term}(\Sigma \cup \Sigma^\#, V)$. The root symbol of a term t is $\mathsf{root}(t)$. The set of defined symbols of a rewrite system R over Σ is $\mathsf{Def}(R) = \{\mathsf{root}(l) \mid (l, r) \in R\}$. The dependency pairs of R are $\mathsf{DP}(R) = \{(l^\#, s^\#) \mid (l, r) \in R, \mathsf{root}(s) \in \mathsf{Def}(R), s \trianglelefteq r, s \not\trianglelefteq l\}$. Termination of R is then proved by a reduction pair $(>, \geq)$ where $\mathsf{DP}(R) \subseteq >$ and $R \subseteq \geq$.

In this context, a d-dimensional matrix interpretation defines an extended monotone algebra by interpreting sort O by $(\mathbb{N}^d, \geq)$, sort $\#$ by $(\mathbb{N}^1, >)$, and function symbols by multilinear functions (respecting the sorts) as before.

For this basic version of the DP method with matrix interpretations, we can apply polyhedral constraints with the following (inessential) modifications:

- We do not need strict monotonicity, so the top-left entries of matrices are unconstrained.
- We restrict the domain for sort O only (not $\#$). That is, we need domain certificates only for symbols from Σ (not $\Sigma^\#$).
- Compatibility certificates are needed for all rules. For rules from R, this is done as before, and for rules from $\mathsf{DP}(R)$, the target domain is $\mathbb{N}^1$, so the matrices U_i in the compatibility certificates have extension $(1 \times c)$.

The DP method allows for many enhancements, which we do not discuss here, but use in examples.

▶ **Example 24.** We consider the termination problem `TRS_Standard/Various_04/11`

$$\{f(0, 1, x) \to f(h(x), h(x), x), h(0) \to 0, h(g(x, y)) \to y\}$$

[4] `http://nfa.imn.htwk-leipzig.de/termcomp/show_job_pair/40858473`

After DP transformation, we look at the SCC that contains $\{f^{\#}(0,1,x) \to f^{\#}(h(x),h(x),x)\}$, and apply the matrix interpretation [5]

$$0 \mapsto \begin{pmatrix}1\\1\end{pmatrix}, 1 \mapsto \begin{pmatrix}0\\2\end{pmatrix}, h \mapsto \begin{pmatrix}0\\0\end{pmatrix} + \begin{pmatrix}0 & 1\\1 & 0\end{pmatrix} \cdot x_1, g \mapsto \begin{pmatrix}0\\0\end{pmatrix} + \begin{pmatrix}0 & 0\\0 & 0\end{pmatrix} \cdot x_1 + \begin{pmatrix}0 & 1\\1 & 0\end{pmatrix} \cdot x_2,$$

$$f \mapsto \begin{pmatrix}0\\0\end{pmatrix} + \begin{pmatrix}0 & 0\\0 & 0\end{pmatrix} \cdot x_1 + \begin{pmatrix}0 & 0\\0 & 0\end{pmatrix} \cdot x_2 + \begin{pmatrix}0 & 0\\0 & 0\end{pmatrix} \cdot x_3,$$

$$f^{\#} \mapsto \begin{pmatrix}0\end{pmatrix} + \begin{pmatrix}1 & 0\end{pmatrix} \cdot x_1 + \begin{pmatrix}0 & 1\end{pmatrix} \cdot x_2 + \begin{pmatrix}1 & 1\end{pmatrix} \cdot x_3$$

with constraint $\begin{pmatrix}2\end{pmatrix} + \begin{pmatrix}-1 & -1\end{pmatrix} \cdot x \geq 0$. The interpretation of $f^{\#}(0,1,x) \to f^{\#}(h(x),h(x),x)$ is

$$[\text{lhs}] = \begin{pmatrix}3\end{pmatrix} + \begin{pmatrix}1 & 1\end{pmatrix} \cdot x_1, \quad [\text{rhs}] = \begin{pmatrix}0\end{pmatrix} + \begin{pmatrix}2 & 2\end{pmatrix} \cdot x_1.$$

We can verify that adding the domain constraint to the value of the right-hand side gives $(2) + (1\ 1) \cdot x_1$ which is in the proper point-wise relation to the left-hand side. This problem was solved in the 2014 termination competition only by Mu-Term [16] and AProVE [12], using innermost rewriting and narrowing. [6]

▶ **Example 25.** For `TRS_Standard/Endrullis_06/pair2simple2`

$$\{p(a(x_0), p(a(a(a(x_1))), x_2)) \to p(a(x_2), p(a(a(b(x_0))), x_2))\}$$

we find an interpretation [7]

$$b \mapsto \begin{pmatrix}0\\0\\0\end{pmatrix} + \begin{pmatrix}0 & 0 & 1\\0 & 0 & 0\\0 & 0 & 0\end{pmatrix} \cdot x_1, a \mapsto \begin{pmatrix}0\\0\\1\end{pmatrix} + \begin{pmatrix}0 & 1 & 0\\0 & 0 & 3\\0 & 0 & 0\end{pmatrix} \cdot x_1,$$

$$p \mapsto \begin{pmatrix}0\\0\\0\end{pmatrix} + \begin{pmatrix}1 & 0 & 0\\0 & 0 & 0\\0 & 0 & 0\end{pmatrix} \cdot x_1 + \begin{pmatrix}1 & 1 & 0\\0 & 0 & 0\\0 & 0 & 0\end{pmatrix} \cdot x_2,$$

$$p^{\#} \mapsto \begin{pmatrix}0\end{pmatrix} + \begin{pmatrix}0 & 0 & 0\end{pmatrix} \cdot x_1 + \begin{pmatrix}1 & 0 & 1\end{pmatrix} \cdot x_2$$

with polyhedral constraint dimension two: $\begin{pmatrix}2\\1\end{pmatrix} + \begin{pmatrix}0 & -1 & 1\\0 & 0 & -1\end{pmatrix} \cdot x \geq 0.$

Rule $p(a(x_0), p(a(a(a(x_1))), x_2)) \to p(a(x_2), p(a(a(b(x_0))), x_2))$ is interpreted by

$$[\text{lhs}] = \begin{pmatrix}3\\0\\0\end{pmatrix} + \begin{pmatrix}0 & 1 & 0\\0 & 0 & 0\\0 & 0 & 0\end{pmatrix} \cdot x_1 + \begin{pmatrix}0 & 0 & 0\\0 & 0 & 0\\0 & 0 & 0\end{pmatrix} \cdot x_2 + \begin{pmatrix}1 & 1 & 0\\0 & 0 & 0\\0 & 0 & 0\end{pmatrix} \cdot x_3$$

$$[\text{rhs}] = \begin{pmatrix}0\\0\\0\end{pmatrix} + \begin{pmatrix}0 & 0 & 0\\0 & 0 & 0\\0 & 0 & 0\end{pmatrix} \cdot x_1 + \begin{pmatrix}0 & 0 & 0\\0 & 0 & 0\\0 & 0 & 0\end{pmatrix} \cdot x_2 + \begin{pmatrix}1 & 2 & 0\\0 & 0 & 0\\0 & 0 & 0\end{pmatrix} \cdot x_3,$$

and the (weak) compatibility certificate is $\begin{pmatrix}0 & 0\\0 & 0\\0 & 0\end{pmatrix}, \begin{pmatrix}0 & 0\\0 & 0\\0 & 0\end{pmatrix}, \begin{pmatrix}1 & 1\\0 & 0\\0 & 0\end{pmatrix}$

This problem was solved in the 2014 competition with recursive path order [8] and it seems there is no compatible standard matrix interpretation.

---

[5] `http://nfa.imn.htwk-leipzig.de/termcomp/show_job_pair/40515447`
[6] `http://nfa.imn.htwk-leipzig.de/termcomp/show_job_pair/26924062`
[7] `http://nfa.imn.htwk-leipzig.de/termcomp/show_job_pair/40849733`
[8] `http://nfa.imn.htwk-leipzig.de/termcomp/show_job_pair/26919296`

8 Polyhedral Constraints and the DP Method with Usable Rules

The DP method with the "usable rules" extension [2] requires the following change. It is required that the reduction pair is C_ϵ-compatible (for a fresh function symbol C, we need $[C](x, y) \geq x \wedge [C](x, y) \geq y$.) For a reduction pair that comes from an interpretation, this means that its domain must allow to construct least upper bounds.

Let us make explicit how this works for the standard matrix method: the domain is $\mathbb{N}^d$, the (weak) order is component-wise $\geq$, the least upper bound $\sup(x, y)$ of $x = (x_1, \ldots, x_d), y = (y_1, \ldots, y_d)$ is $(\max(x_1, y_1), \ldots, \max(x_n, y_n))$, so $\sup(x, y) \in \mathbb{N}^d$ trivially.

Now consider some polyhedral domain $D \subseteq \mathbb{N}^d$. The order we use on D is exactly the order on $\mathbb{N}^d$ as before, so the least upper bound is the same as well. But it is not always true that $x, y \in D$ implies $\sup(x, y) \in D$.

▶ **Example 26.** In dimension 2, consider the constraint $x_1 + x_2 \leq 1$. This means $D = \{(0, 0), (0, 1), (1, 0)\}$. Then $\sup((0, 1), (1, 0)) = (1, 1) \notin D$.

Indeed we would obtain erroneous termination statements when using the "usable rules" extension with polyhedral constraints on domains that are not sup-closed. [9]

We now give a sufficient condition for a polyhedral domain to allow sup.

▶ **Theorem 27.** *Let $C \in \mathbb{Q}^{c \times d}, B \in \mathbb{Q}^{c \times 1}$ describe a domain $D = \{x \mid x \geq 0, Cx + B \geq 0\} \subseteq \mathbb{Q}^d$. If each row of C contains at most one negative entry, then $x, y \in D$ implies $\sup(x, y) \in D$.*

Proof. We analyze the i-th constraint, given by row $c = C_i$, and entry $b = B_i$. We need to show $c \cdot \sup(x, y) + b \geq 0$. We can write $c = c^+ + c^-$ where $c^+ = \sup(c, 0)$ and $c^- = \inf(c, 0)$. (all entries in c^+ are ≥ 0, all entries in c^- are ≤ 0). Multiplication by c^+ is monotone: if $x \leq z$, then $c^+ x \leq c^+ z$. If c has no negative entry, then $c = c^+$, so multiplication by c is monotone, and we have $c \cdot \sup(x, y) + b \geq cx + b \geq 0$. Assume c has one negative entry $c_k < 0$. Without loss of generality, we have $x_k \geq y_k$ (if not, swap x with y). Then $c^- \cdot \sup(x, y) = c^- x$. We have $c \cdot \sup(x, y) + b = c^+ \cdot \sup(x, y) + c^- \cdot \sup(x, y) + b \geq c^+ x + c^- x + b = cx + b \geq 0$. ◀

The condition in Theorem 27 is easily implemented in a constraint program.

9 Implementation and Experiments

We extended to the implementation of matrix interpretations in `matchbox` [29] by adding polyhedral constraints. The constraint system already has unknowns for the interpretation, and we added unknowns for the domain constraint and certificate, and for the compatibility certificates. The constraint program already computes the interpretation of rules (with compression) [5], and we added the validity constraints for domain and compatibility.

To prove termination of a rewrite system, we have (as usual) one parameter $d \in \mathbb{N}$, for the dimension of the interpretation domain, and now an extra parameter $c \in \mathbb{N}$: the number of inequalities (the height of the C matrix). We found that already $c = 1$ is often helpful, see most examples in this paper.

According to Sections 3 and 4, unknowns should be from $\mathbb{Q}$ for the domain constraint, and $\mathbb{Q}_+$ for certificates. Since we don't know of a competitive constraint solver over $\mathbb{Q}$, we restrict to $\mathbb{Z}$ for the domain constraint, and $\mathbb{N}$ for certificates. Experiments suggest that

[9] `http://nfa.imn.htwk-leipzig.de/termcomp/show_job_pair/40422582`

most domain constraints use small numbers, so we further restrict to $\{-1, 0, 1\}$ for C (not for B).

The constraint system consists of (in)equalities between polynomials, so it is expressible in the QFNRA (QFNIA, resp.) logic [4]. With a bit-blasting approach in mind, we can also use QFBV (bit vectors). Our implementation allows us to choose between Boolector [6] as a QFBV-solver, or built-in bit-blasting, and then MiniSat [10] as a SAT-solver.

The following data is typical of how polyhedral constraints increase the size of the constraint systems:

▶ **Example 28** (Example 22 continued)**.** For `TRS/secret06/jambox/5`, with domain dimension 4, and bit width 4, matchbox' built-in bit-blaster was applied. The number of variables/clauses is: for constraint dimension 0 (the original matrix method): 31614/40256, for constraint dimension 1: 39867/67064.

We have two remarks on BV-solving/bit-blasting:

Overflow is forbidden in our context, but allowed in the QFBV standard. So each arithmetical operation (`add`, `mul`) is immediately followed by computing the overflow and asserting that it is false. We use functions `saddo`, `smulo` provided in Boolector's API.

We need signed numbers. Among the unknowns, just C and B may contain negative numbers, but signed numbers will propagate into the validity constraints. We note that while we have additions of signed numbers, all multiplications have at most one signed factor.

In both cases (no overflow, some signs are known), a constraint solver could exploit this information statically. We especially think that "non-overflowing arithmetic" would be a useful addition to the QFBV-standard.

We were running our implementation on the termination and complexity problems of the 2014 termination competition (TPDB version 8). The main purpose was to extract interesting examples, used in this paper. These examples show that there are several cases where polyhedral constraints allow a matrix termination proof where none was given in the last competition, or only proofs that use other methods.

We checked the effect that polyhedral constraints have when added to a base version of matchbox with arctic and natural matrix interpretations. We observe different behaviour in less than 10 % of the benchmarks.

We also compared Boolector and bit-blasting/MiniSat back-ends. Our conclusion is that Boolector wins by a small margin.

A web page with that presents our experimental data is provided. [10]

10 Discussion

Related work. Polyhedral constraints for interpretations in termination proofs where first suggested by Lucas and Meseguer [17]. Our contribution is to provide an actual implementation that handles the case where interpretation and domain constraints are unknown, and extensions (to complexity analysis, dependency pairs, and usable rules).

We mentioned that polyhedral analysis of imperative programs, especially loops, is a standard method. The difference to analysis of rewriting systems is: the semantics of the imperative program is literally given by the numerical values and operations appearing in the program text (e.g., in `int y = 2*x+1`, the symbol `2` denotes the number 2, and the symbol `+` denotes addition). For rewrite systems, a priori there is no semantics, so it has

[10] `http://www.imn.htwk-leipzig.de/~waldmann/etc/polyhedral/`

to be determined during the analysis (e.g., it will be defined via an interpretation). This means that for polyhedral domains for rewrite systems, we cannot use directly the methods developed for imperative programs.

Certification. It seems straightforward (in principle) to integrate polyhedral domains for matrix interpretations into the CeTA certification framework [23]. To check validity of domain and compatibility certificates, we just need to verify the calculations from Section 4, while a certified proof of Farkas' Lemma (Section 3) is not needed.

Challenges in Rewriting. There are two long-standing challenges: find a matrix interpretation that gives a tight complexity bound for $\{a^2b^2 \to b^3a^3\}$ (z001), and for $\{a^2 \to bc, b^2 \to ac, c^2 \to ab\}$ (z086). For both rewrite systems, matrix interpretations with exponential growth are known, while the growth of rewrite sequences is known to be polynomial (z001 by matchbounds, z086 by a manual proof [1]). Can we prove polynomial derivational complexity via matrix interpretations on a polyhedral domain? So far, we did not succeed—using several days of CPU time.

Extensions: Order. It may be interesting to analyze different orders on polyhedral domains, as Neurauter et al. [20] did for the full standard domain $\mathbb{N}^d$. We might get more termination proofs, or better complexity bounds. It is to be expected that monotonicity, which is now easy (top left coefficient ≥ 1), needs to be replaced with something more elaborate, that requires a certificate.

Extensions: Negative Coefficients. Can we allow negative coefficients in interpretations (in $F_0, F_1, \dots, F_k$)? We then need to make sure that $x \geq 0$ is respected (by extra domain certificates), and require additional "monotonicity certificates".

Extensions: Domain. We can perhaps even drop the $x \geq 0$ restriction, This implies changes in other certificates, and requires an extra "well-foundedness certificate" that shows that values for the first component of interpretations are bounded from below.

Extensions: Semiring. Another direction for extension is to choose a different underlying semiring, e.g., arctic or tropical, and apply results from tropical linear algebra. At least in principle, it is clear what to do: a polyhedral domain [9] is described as $\{x \mid A_1 \cdot x + b_1 \leq x \leq A_2 \cdot x + b_2\}$ (since addition is not invertible, we cannot move the right-hand-side x to the left) and certificates must be constructed accordingly.

Acknowledgments. I am grateful to Alfons Geser, Dieter Hofbauer, and anonymous reviewers for discussions and comments, and to René Thiemann and Harald Zankl for help with their software. The Starexec platform [24] and Starexec-Presenter [27] software proved to be very helpful for running experiments and analyzing data. Armin Biere made available Boolector [6] for use as an alternative solver back-end.

References

1 Sergei I. Adian. Upper bound on the derivational complexity in some word rewriting system. *Doklady Mathematics*, 80(2):679–683, 2009.

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

3 Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. The Parma Polyhedra Library: Toward a Complete Set of Numerical Abstractions for the Analysis and Verification of Hardware and Software Systems. *CoRR*, abs/cs/0612085, 2006.

4 Clark Barrett, Aaron Stump, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `http://smt-lib.org/`, 2010–.

5 Alexander Bau, Markus Lohrey, Eric Nöth, and Johannes Waldmann. Compression of rewriting systems for termination analysis. In Femke van Raamsdonk, editor, *24th International Conference on Rewriting Techniques and Applications, RTA 2013, June 24-26, 2013, Eindhoven, The Netherlands*, volume 21 of *LIPIcs*, pages 97–112. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2013.

6 Armin Biere and Robert Brummayer. Boolector. `http://fmv.jku.at/boolector/`, 2008–.

7 Ahlem Ben Cherifa and Pierre Lescanne. Termination of rewriting systems by polynomial interpretations and its implementation. *Sci. Comput. Program.*, 9(2):137–159, 1987.

8 Michael Codish, Yoav Fekete, Carsten Fuhs, Jürgen Giesl, and Johannes Waldmann. Exotic semi-ring constraints. In Pascal Fontaine and Amit Goel, editors, *10th International Workshop on Satisfiability Modulo Theories, SMT 2012, Manchester, UK, June 30 - July 1, 2012*, volume 20 of *EPiC Series*, pages 88–97. EasyChair, 2012.

9 Mike Develin and Josephine Yu. Tropical polytopes and cellular resolutions. *Experimental Mathematics*, 16(3):277–291, 2007.

10 Niklas Een and Niklas Sörensson. Minisat. http://minisat.se/, 2003–.

11 Jörg Endrullis, Johannes Waldmann, and Hans Zantema. Matrix interpretations for proving termination of term rewriting. *J. Autom. Reasoning*, 40(2-3):195–220, 2008.

12 Jürgen Giesl, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, Thomas Ströder, Stephanie Swiderski, and René Thiemann. Proving Termination of Programs Automatically with AProVE. In Stéphane Demri, Deepak Kapur, and Christoph Weidenbach, editors, *Automated Reasoning - 7th International Joint Conference, IJCAR 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 19-22, 2014. Proceedings*, volume 8562 of *Lecture Notes in Computer Science*, pages 184–191. Springer, 2014.

13 Dieter Hofbauer and Johannes Waldmann. Termination of string rewriting with matrix interpretations. In Frank Pfenning, editor, *Term Rewriting and Applications, 17th International Conference, RTA 2006, Seattle, WA, USA, August 12-14, 2006, Proceedings*, volume 4098 of *Lecture Notes in Computer Science*, pages 328–342. Springer, 2006.

14 Adam Koprowski and Johannes Waldmann. Max/plus tree automata for termination of term rewriting. *Acta Cybern.*, 19(2):357–392, 2009.

15 Martin Korp and Harald Zankl. Cat (complexity and termination). `http://cl-informatik.uibk.ac.at/software/cat/`, 2008–.

16 Salvador Lucas. mu-term: A tool for proving termination of context-sensitive rewriting. In Vincent van Oostrom, editor, *Rewriting Techniques and Applications, 15th International Conference, RTA 2004, Aachen, Germany, June 3-5, 2004, Proceedings*, volume 3091 of *Lecture Notes in Computer Science*, pages 200–209. Springer, 2004.

17 Salvador Lucas and José Meseguer. Models for logics and conditional constraints in automated proofs of termination. In Gonzalo A. Aranda-Corral, Jacques Calmet, and Francisco J. Martín-Mateos, editors, *Artificial Intelligence and Symbolic Computation - 12th International Conference, AISC 2014, Seville, Spain, December 11-13, 2014. Proceedings*, volume 8884 of *Lecture Notes in Computer Science*, pages 9–20. Springer, 2014.

18 Georg Moser, Andreas Schnabl, and Johannes Waldmann. Complexity analysis of term rewriting based on matrix and context dependent interpretations. In Ramesh Hariharan, Madhavan Mukund, and V. Vinay, editors, *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2008, December 9-11,*

2008, Bangalore, India, volume 2 of *LIPIcs*, pages 304–315. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2008.

19 Friedrich Neurauter and Aart Middeldorp. Revisiting matrix interpretations for proving termination of term rewriting. In Manfred Schmidt-Schauß, editor, *Proceedings of the 22nd International Conference on Rewriting Techniques and Applications, RTA 2011, May 30 - June 1, 2011, Novi Sad, Serbia*, volume 10 of *LIPIcs*, pages 251–266. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011.

20 Friedrich Neurauter, Harald Zankl, and Aart Middeldorp. Revisiting matrix interpretations for polynomial derivational complexity of term rewriting. In Christian G. Fermüller and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 17th International Conference, LPAR-17, Yogyakarta, Indonesia, October 10-15, 2010. Proceedings*, volume 6397 of *Lecture Notes in Computer Science*, pages 550–564. Springer, 2010.

21 Albert Rubio, Claude Marche, Hans Zantema, and René Thiemann. Termination problems data base (tpdb). `http://termination-portal.org/wiki/TPDB`, 2003–.

22 Christian Sternagel and Aart Middeldorp. Root-labeling. In Andrei Voronkov, editor, *Rewriting Techniques and Applications, 19th International Conference, RTA 2008, Hagenberg, Austria, July 15-17, 2008, Proceedings*, volume 5117 of *Lecture Notes in Computer Science*, pages 336–350. Springer, 2008.

23 Christian Sternagel, René Thiemann, Sarah Winkler, and Harald Zankl. CeTA - A Tool for Certified Termination Analysis. *CoRR*, abs/1208.1591, 2012.

24 Aaron Stump, Tyler Jensen, and Cesare Tinelli. Starexec. `https://www.starexec.org/`, 2013–.

25 Sergej N. Tschernikow. *Lineare Ungleichungen*. Berlin, 1971.

26 Aard Varks. Farkas' lemma. `http://aardvarks.bandcamp.com/album/farkas-lemma`, 1996.

27 Stefan von der Krone. Stareexec-presenter. `https://github.com/stefanvonderkrone/star-exec-presenter`, 2014–.

28 Johannes Waldmann. Polynomially bounded matrix interpretations. In Christopher Lynch, editor, *Proceedings of the 21st International Conference on Rewriting Techniques and Applications, RTA 2010, July 11-13, 2010, Edinburgh, Scottland, UK*, volume 6 of *LIPIcs*, pages 357–372. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010.

29 Johannes Waldmann, Alexander Bau, and Eric Noeth. Matchbox termination prover. http://github.com/jwaldmann/matchbox/, 2014–.

30 Hans Zantema. Termination of term rewriting by interpretation. In Michaël Rusinowitch and Jean-Luc Remy, editors, *Conditional Term Rewriting Systems, Third International Workshop, CTRS-92, Pont-à-Mousson, France, July 8-10, 1992, Proceedings*, volume 656 of *Lecture Notes in Computer Science*, pages 155–167. Springer, 1992.

Inferring Lower Bounds for Runtime Complexity*

Florian Frohn, Jürgen Giesl, Jera Hensel, Cornelius Aschermann, and Thomas Ströder

LuFG Informatik 2, RWTH Aachen University, Germany
{florian.frohn,giesl,hensel,cornelius,stroeder}@informatik.rwth-aachen.de

Abstract

We present the first approach to deduce lower bounds for innermost runtime complexity of term rewrite systems (TRSs) automatically. Inferring lower runtime bounds is useful to detect bugs and to complement existing techniques that compute upper complexity bounds. The key idea of our approach is to generate suitable families of rewrite sequences of a TRS and to find a relation between the length of such a rewrite sequence and the size of the first term in the sequence. We implemented our approach in the tool AProVE and evaluated it by extensive experiments.

1998 ACM Subject Classification F.1.3 - Complexity Measures and Classes, F.4.2 - Grammars and Other Rewriting Systems, I.2.3 Deduction and Theorem Proving

Keywords and phrases Term Rewriting, Runtime Complexity, Lower Bounds, Induction

Digital Object Identifier 10.4230/LIPIcs.RTA.2015.334

1 Introduction

There exist numerous methods to infer *upper bounds* for the runtime complexity of TRSs [3, 12, 14, 17, 21]. We present the first automatic technique to infer *lower bounds* for the innermost[1] runtime complexity of TRSs. *Runtime complexity* [12] refers to the "worst" cases in terms of evaluation length and our goal is to find lower bounds for these cases. While upper complexity bounds help to prove the absence of bugs that worsen the performance of programs, lower bounds can be used to *find* such bugs. Moreover, in combination with methods to deduce upper bounds, our approach can prove *tight* complexity results. In addition to *asymptotic* lower bounds, in many cases our technique can even compute *concrete* bounds.

As an example, consider the following TRS $\mathcal{R}_{\mathsf{qs}}$ for *quicksort*. The auxiliary function $\mathsf{low}(x, xs)$ returns those elements from the list xs that are smaller than x (and high works analogously). To ease readability, we use infix notation for the function symbols $\leq$ and $\mathsf{++}$.

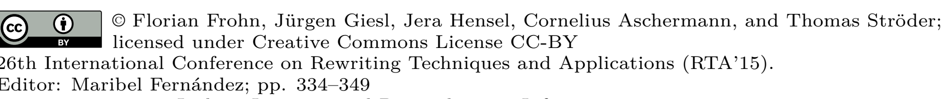
▶ **Example 1** (TRS $\mathcal{R}_{\mathsf{qs}}$ for Quicksort).

$$\begin{array}{rcll}
\mathsf{qs}(\mathsf{nil}) & \to & \mathsf{nil} & (1)\\
\mathsf{qs}(\mathsf{cons}(x, xs)) & \to & \mathsf{qs}(\mathsf{low}(x, xs)) \mathbin{\mathsf{++}} \mathsf{cons}(x, \mathsf{qs}(\mathsf{high}(x, xs))) & (2)\\
\mathsf{low}(x, \mathsf{nil}) & \to & \mathsf{nil} & \\
\mathsf{low}(x, \mathsf{cons}(y, ys)) & \to & \mathsf{ifLow}(x \leq y, x, \mathsf{cons}(y, ys)) & \\
\mathsf{ifLow}(\mathsf{true}, x, \mathsf{cons}(y, ys)) & \to & \mathsf{low}(x, ys) & \\
\mathsf{ifLow}(\mathsf{false}, x, \mathsf{cons}(y, ys)) & \to & \mathsf{cons}(y, \mathsf{low}(x, ys)) & \\
\mathsf{high}(x, \mathsf{nil}) & \to & \mathsf{nil} & \\
\mathsf{high}(x, \mathsf{cons}(y, ys)) & \to & \mathsf{ifHigh}(x \leq y, x, \mathsf{cons}(y, ys)) &
\end{array}$$

$$\begin{array}{rcl}
\mathsf{zero} \leq x & \to & \mathsf{true}\\
\mathsf{succ}(x) \leq \mathsf{zero} & \to & \mathsf{false}\\
\mathsf{succ}(x) \leq \mathsf{succ}(y) & \to & x \leq y
\end{array}$$

* Supported by the DFG grant GI 274/6-1.

[1] We consider *innermost* rewriting, since TRSs resulting from the translation of programs usually have to be evaluated with an innermost strategy (e.g., [10, 18]). Obviously, lower bounds for innermost reductions are also lower bounds for full reductions (i.e., our approach can also be used for full rewriting).


© Florian Frohn, Jürgen Giesl, Jera Hensel, Cornelius Aschermann, and Thomas Ströder; licensed under Creative Commons License CC-BY

$$\mathsf{ifHigh}(\mathsf{true}, x, \mathsf{cons}(y, ys)) \to \mathsf{cons}(y, \mathsf{high}(x, ys)) \qquad \mathsf{nil} \mathbin{++} ys \to ys \qquad (3)$$
$$\mathsf{ifHigh}(\mathsf{false}, x, \mathsf{cons}(y, ys)) \to \mathsf{high}(x, ys) \qquad \mathsf{cons}(x, xs) \mathbin{++} ys \to \mathsf{cons}(x, xs \mathbin{++} ys)$$

For any $n \in \mathbb{N}$, let $\gamma_{\mathbf{List}}(n)$ be the term $\overbrace{\mathsf{cons}(\mathsf{zero}, \ldots, \mathsf{cons}(\mathsf{zero}, \mathsf{nil})}^{n \text{ times}} \ldots)$, i.e., the list of length n where all elements have the value zero (we also use the notation "$\mathsf{cons}^n(\mathsf{zero}, \mathsf{nil})$"). To find lower bounds, we automatically generate *rewrite lemmas* that describe families of rewrite sequences. For example, our technique infers the following rewrite lemma automatically.

$$\mathsf{qs}(\gamma_{\mathbf{List}}(n)) \quad \xrightarrow{\mathrm{i}}{}^{3n^2+2n+1} \quad \gamma_{\mathbf{List}}(n) \qquad (4)$$

This rewrite lemma means that for each $n \in \mathbb{N}$, there is an innermost rewrite sequence of length $3n^2 + 2n + 1$ that reduces $\mathsf{qs}(\mathsf{cons}^n(\mathsf{zero}, \mathsf{nil}))$ to $\mathsf{cons}^n(\mathsf{zero}, \mathsf{nil})$. From this rewrite lemma, our technique then concludes that the innermost runtime of $\mathcal{R}_{\mathsf{qs}}$ is at least quadratic.

While most methods to infer upper bounds are adaptions of termination techniques, the approach in this paper is related to our technique to prove non-termination of TRSs [7]. Both techniques generate "meta-rules" representing infinitely many rewrite sequences. However, the *rewrite lemmas* in the current paper are more general than the meta-rules in [7], as they can be parameterized by *several* variables $n_1, \ldots, n_m$ of type $\mathbb{N}$.

In Sect. 2 we show how to automatically speculate conjectures that may result in suitable rewrite lemmas. Sect. 3 explains how these conjectures can be verified automatically by induction. From these induction proofs, one can deduce information on the lengths of the rewrite sequences represented by a rewrite lemma, cf. Sect. 4. Thus, the use of induction to infer lower runtime bounds represents a novel application for automated inductive theorem proving. This complements our earlier work on using inductive theorem proving for termination analysis [9]. Finally, Sect. 5 shows how rewrite lemmas can be used to infer lower bounds for the innermost runtime complexity of a TRS.

Sect. 6 discusses an improvement of our approach by pre-processing the TRS before the analysis and Sect. 7 extends our approach to handle rewrite lemmas with arbitrary unknown right-hand sides. We implemented our technique in the tool AProVE [11] and demonstrate its power by an extensive experimental evaluation in Sect. 8. All proofs can be found in [8].

2 Speculating Conjectures

We now show how to speculate conjectures (whose validity must be proved afterwards in Sect. 3). See, e.g., [5] for the basics of rewriting, where we only consider finite TRSs. $\mathcal{T}(\Sigma, \mathcal{V})$ is the set of all terms over a (finite) signature Σ and a set of variables $\mathcal{V}$ and $\mathcal{T}(\Sigma) = \mathcal{T}(\Sigma, \varnothing)$ is the set of ground terms. The *arity* of a symbol $f \in \Sigma$ is denoted by $\mathrm{ar}_\Sigma(f)$. As usual, the *defined symbols* of a TRS $\mathcal{R}$ are $\Sigma_{def}(\mathcal{R}) = \{\mathrm{root}(\ell) \mid \ell \to r \in \mathcal{R}\}$ and the *constructors* $\Sigma_{con}(\mathcal{R})$ are all other function symbols in $\mathcal{R}$. Thus, $\Sigma_{def}(\mathcal{R}_{\mathsf{qs}}) = \{\mathsf{qs}, \mathsf{low}, \mathsf{ifLow}, \mathsf{high}, \mathsf{ifHigh}, \mathsf{++}, \leq\}$ and $\Sigma_{con}(\mathcal{R}_{\mathsf{qs}}) = \{\mathsf{nil}, \mathsf{cons}, \mathsf{zero}, \mathsf{succ}, \mathsf{true}, \mathsf{false}\}$.

Our approach is based on rewrite lemmas containing *generator functions* such as $\gamma_{\mathbf{List}}$ for types like **List**. Hence, in the first step of our approach we compute suitable types for the TRS $\mathcal{R}$ to be analyzed. While ordinary TRSs are defined over untyped signatures Σ, Def. 2 shows how to extend such signatures by (monomorphic) types (see, e.g., [9, 14, 22]).

▶ **Definition 2** (Typing)**.** Let Σ be an (untyped) signature. A many-sorted signature Σ' is a *typed variant* of Σ if it contains the same function symbols as Σ, with the same arities. So $f \in \Sigma$ with $\mathrm{ar}_\Sigma(f) = k$ iff $f \in \Sigma'$ where f's type has the form $\tau_1 \times \ldots \times \tau_k \to \tau$. Similarly, a typed variant $\mathcal{V}'$ of the set of variables $\mathcal{V}$ contains the same variables as $\mathcal{V}$, but now every

variable has a type τ. We always assume that for every type τ, $\mathcal{V}'$ contains infinitely many variables of type τ. Given Σ' and $\mathcal{V}'$, $t \in \mathcal{T}(\Sigma, \mathcal{V})$ is a *well-typed* term of type τ iff

- $t \in \mathcal{V}'$ is a variable of type τ or
- $t = f(t_1, \ldots, t_k)$ with $k \geq 0$, where each t_i is a well-typed term of type τ_i, and where $f \in \Sigma'$ has the type $\tau_1 \times \ldots \times \tau_k \to \tau$.

We only permit typed variants Σ' where there exist well-typed ground terms of types $\tau_1, \ldots, \tau_k$ over Σ', whenever some $f \in \Sigma'$ has type $\tau_1 \times \ldots \times \tau_k \to \tau$.[2]

A TRS $\mathcal{R}$ over Σ and $\mathcal{V}$ is *well typed* w.r.t. Σ' and $\mathcal{V}'$ iff for all $\ell \to r \in \mathcal{R}$, we have that ℓ and r are well typed and that they have the same type.[3]

For any TRS $\mathcal{R}$, one can use a standard type inference algorithm to compute a typed variant Σ' such that $\mathcal{R}$ is well typed. Of course, a trivial solution is to use a many-sorted signature with just one sort (then every term and every TRS are trivially well typed). But to make our approach more powerful, it is advantageous to use the most general typed variant where $\mathcal{R}$ is well typed. Here, the set of terms is decomposed into as many types as possible. Then fewer terms are well typed and more useful rewrite lemmas can be generated.

To make $\mathcal{R}_{\mathsf{qs}}$ from Ex. 1 well typed, we obtain a typed variant of its signature with the types **Nats**, **Bool**, and **List**. Here, the function symbols have the following types:

$$
\begin{aligned}
\mathsf{nil} &: \mathbf{List} & \mathsf{qs} &: \mathbf{List} \to \mathbf{List}\\
\mathsf{cons} &: \mathbf{Nats} \times \mathbf{List} \to \mathbf{List} & \mathsf{++} &: \mathbf{List} \times \mathbf{List} \to \mathbf{List}\\
\mathsf{zero} &: \mathbf{Nats} & \leq &: \mathbf{Nats} \times \mathbf{Nats} \to \mathbf{Bool}\\
\mathsf{succ} &: \mathbf{Nats} \to \mathbf{Nats} & \mathsf{low}, \mathsf{high} &: \mathbf{Nats} \times \mathbf{List} \to \mathbf{List}\\
\mathsf{true}, \mathsf{false} &: \mathbf{Bool} & \mathsf{ifLow}, \mathsf{ifHigh} &: \mathbf{Bool} \times \mathbf{Nats} \times \mathbf{List} \to \mathbf{List}
\end{aligned}
$$

A type τ *depends* on a type τ' (denoted $\tau \sqsupseteq_{dep} \tau'$) iff $\tau = \tau'$ or if there is a $c \in \Sigma'_{con}(\mathcal{R})$ of type $\tau_1 \times \ldots \times \tau_k \to \tau$ where $\tau_i \sqsupseteq_{dep} \tau'$ for some $1 \leq i \leq k$. To ease the presentation, we do not allow mutually recursive types (i.e., if $\tau \sqsupseteq_{dep} \tau'$ and $\tau' \sqsupseteq_{dep} \tau$, then $\tau' = \tau$). To speculate conjectures, we now introduce generator functions γ_τ. For any $n \in \mathbb{N}$, $\gamma_\tau(n)$ is a term from $\mathcal{T}(\Sigma'_{con}(\mathcal{R}))$ where a recursive constructor of type τ is nested n times. A constructor $c : \tau_1 \times \ldots \times \tau_k \to \tau$ is *recursive* iff $\tau_i = \tau$ for some $1 \leq i \leq k$. So for the type **Nats** above, we have $\gamma_{\mathbf{Nats}}(0) = \mathsf{zero}$ and $\gamma_{\mathbf{Nats}}(n+1) = \mathsf{succ}(\gamma_{\mathbf{Nats}}(n))$. If a constructor has a non-recursive argument of type τ', then γ_τ instantiates this argument by $\gamma_{\tau'}(0)$. So for **List**, we get $\gamma_{\mathbf{List}}(0) = \mathsf{nil}$ and $\gamma_{\mathbf{List}}(n+1) = \mathsf{cons}(\mathsf{zero}, \gamma_{\mathbf{List}}(n))$. If a constructor has several recursive arguments, then several generator functions are possible. So for a type **Tree** with the constructors $\mathsf{leaf} : \mathbf{Tree}$ and $\mathsf{node} : \mathbf{Tree} \times \mathbf{Tree} \to \mathbf{Tree}$, we have $\gamma_{\mathbf{Tree}}(0) = \mathsf{leaf}$, but either $\gamma_{\mathbf{Tree}}(n+1) = \mathsf{node}(\gamma_{\mathbf{Tree}}(n), \mathsf{leaf})$ or $\gamma_{\mathbf{Tree}}(n+1) = \mathsf{node}(\mathsf{leaf}, \gamma_{\mathbf{Tree}}(n))$. Similarly, if a type has several non-recursive or recursive constructors, then several different generator functions can be constructed by considering all combinations of non-recursive and recursive constructors.

To ease the presentation, we only consider generator functions for *simply structured* types τ. Such types have exactly two constructors $c, d \in \Sigma'_{con}(\mathcal{R})$, where c is not recursive, d has exactly one argument of type τ, and each argument type $\tau' \neq \tau$ of c or d is simply structured, too. The presented approach can easily be extended to more complex types by applying suitable heuristics to choose one of the possible generator functions.

---

[2] This is not a restriction, as one can simply add new constants to Σ and Σ'.

[3] W.l.o.g., here one may rename the variables in every rule. Then it is not a problem if the variable x is used with type τ_1 in one rule and with type τ_2 in another rule.

▶ **Definition 3** (Generator Functions and Equations)**.** Let $\mathcal{R}$ be a TRS that is well typed w.r.t. Σ' and $\mathcal{V}'$. We extend the set of types by a fresh type $\mathbb{N}$. For every type $\tau \neq \mathbb{N}$, let γ_τ be a fresh *generator function symbol* of type $\mathbb{N} \to \tau$. The set $\mathcal{G}_\mathcal{R}$ consists of the following *generator equations* for every simply structured type τ with the constructors $c : \tau_1 \times \ldots \times \tau_k \to \tau$ and $d : \rho_1 \times \ldots \times \rho_b \to \tau$, where $\rho_j = \tau$. We write $\mathcal{G}$ instead of $\mathcal{G}_\mathcal{R}$ if $\mathcal{R}$ is clear from the context.

$$\begin{aligned}\gamma_\tau(0) &= c(\gamma_{\tau_1}(0), \ldots, \gamma_{\tau_k}(0))\\ \gamma_\tau(n+1) &= d(\gamma_{\rho_1}(0), \ldots, \gamma_{\rho_{j-1}}(0), \gamma_\tau(n), \gamma_{\rho_{j+1}}(0), \ldots, \gamma_{\rho_b}(0))\end{aligned}$$

We extend $\sqsupseteq_{dep}$ to $\Sigma_{def}(\mathcal{R})$ by defining $f \sqsupseteq_{dep} h$ iff $f = h$ or if there is a rule $f(\ldots) \to r$ and a symbol g in t with $g \sqsupseteq_{dep} h$. When speculating conjectures, we take the dependencies between defined symbols into account. If $f \sqsupseteq_{dep} g$ and $g \not\sqsupseteq_{dep} f$, then we first generate a rewrite lemma for g. This lemma can be used when generating a lemma for f afterwards.

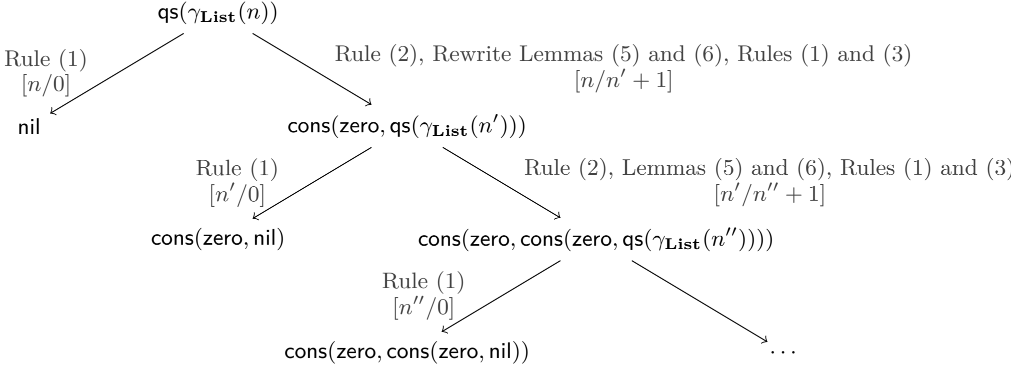
For $f \in \Sigma'_{def}(\mathcal{R})$ of type $\tau_1 \times \ldots \times \tau_k \to \tau$ with simply structured types $\tau_1, \ldots, \tau_k$, our goal is to speculate a conjecture of the form $f(\gamma_{\tau_1}(s_1), \ldots, \gamma_{\tau_k}(s_k)) \xrightarrow{\text{i}}^* t$, where the $s_1, \ldots, s_k$ are polynomials over variables $n_1, \ldots, n_m$ of type $\mathbb{N}$. Moreover, t is a term built from Σ, arithmetic expressions, generator functions, and $n_1, \ldots, n_m$. As usual, a rewrite step is *innermost* (denoted $s \xrightarrow{\text{i}}_\mathcal{R} t$ where we omit the index $\mathcal{R}$ if it is clear from the context) if the reduced subterm of s does not have redexes as proper subterms. From the speculated conjecture, we afterwards infer a rewrite lemma $f(\gamma_{\tau_1}(s_1), \ldots, \gamma_{\tau_k}(s_k)) \xrightarrow{\text{i}}^{\mathit{rt}(n_1,\ldots,n_m)} t$, where $\mathit{rt} : \mathbb{N}^m \to \mathbb{N}$ describes the runtime of the lemma. To speculate a conjecture, we first generate *sample conjectures* that describe the effect of applying f to specific arguments. To this end, we narrow $f(\gamma_{\tau_1}(n_1), \ldots, \gamma_{\tau_k}(n_k))$ where $n_1, \ldots, n_k \in \mathcal{V}$ using the rules of the TRS and the lemmas we have proven so far, taking also the generator equations and integer arithmetic into account.

For any proven rewrite lemma $s \xrightarrow{\text{i}}^{\mathit{rt}(\ldots)} t$, let the set $\mathcal{L}$ contain the rule $s \to t$. Moreover, let $\mathcal{A}$ be the infinite set of all valid equalities in the theory of $\mathbb{N}$ with addition and multiplication. Then s *narrows* to t ("$s \rightsquigarrow_{(\mathcal{R}\cup\mathcal{L})/(\mathcal{G}\cup\mathcal{A})} t$" or just "$s \rightsquigarrow t$" if $\mathcal{R}$, $\mathcal{L}$, $\mathcal{G}$ are clear from the context) iff there exist a term s', a substitution σ that maps variables of type $\mathbb{N}$ to arithmetic expressions, a position π, and a variable-renamed rule $\ell \to r \in \mathcal{R} \cup \mathcal{L}$ such that $s\sigma \equiv_{\mathcal{G}\cup\mathcal{A}} s'\sigma$, $s'|_\pi\sigma = \ell\sigma$, and $s'[r]_\pi\sigma = t$. Although checking $s\sigma \equiv_{\mathcal{G}\cup\mathcal{A}} s'\sigma$ (i.e., $\mathcal{G} \cup \mathcal{A} \models s\sigma = s'\sigma$) is undecidable in general, the required narrowing can usually be performed automatically using SMT solvers.

▶ **Example 4** (Narrowing)**.** In Ex. 1 we have $\mathsf{qs} \sqsupseteq_{dep} \mathsf{low}$ and $\mathsf{qs} \sqsupseteq_{dep} \mathsf{high}$. If the lemmas

$$\mathsf{low}(\gamma_{\mathbf{Nats}}(0), \gamma_{\mathbf{List}}(n)) \xrightarrow{\text{i}}^{3n+1} \gamma_{\mathbf{List}}(0) \quad (5) \qquad \mathsf{high}(\gamma_{\mathbf{Nats}}(0), \gamma_{\mathbf{List}}(n)) \xrightarrow{\text{i}}^{3n+1} \gamma_{\mathbf{List}}(n) \quad (6)$$

were already proved, then the following narrowing tree can be generated to find sample conjectures for qs. The arrows are annotated with the rules and the substitutions used for variables of type $\mathbb{N}$. To save space, some arrows correspond to *several* narrowing steps.



The goal is to get representative rewrite sequences, but not to cover all reductions. So we stop constructing the tree after some steps and choose suitable narrowings heuristically.

After constructing a narrowing tree for f, we collect *sample points* (t, σ, d). Here, t results from a $\rightsquigarrow$-normal form q reached in a path of the tree by normalizing q w.r.t. the generator equations $\mathcal{G}$ applied from right to left. So terms from $\mathcal{T}(\Sigma, \mathcal{V})$ are rewritten to generator symbols with arithmetic expressions as arguments. Moreover, σ is the substitution for variables of type $\mathbb{N}$, and d is the number of applications of recursive f-rules on the path (the *recursion depth*). A rule $f(\dots) \to r$ is *recursive* iff r contains a symbol g with $g \sqsupseteq_{dep} f$.

▶ **Example 5** (Sample Points)**.** In Ex. 4, we obtain the following set of sample points:[4]

$$S \;=\; \{\, (\gamma_{\mathbf{List}}(0), [n/0], 0), \quad (\gamma_{\mathbf{List}}(1), [n/1], 1), \quad (\gamma_{\mathbf{List}}(2), [n/2], 2)\,\} \tag{7}$$

The sequence from $\mathsf{qs}(\gamma_{\mathbf{List}}(n))$ to nil does not use recursive qs-rules. So its recursion depth is 0 and the $\rightsquigarrow$-normal form nil rewrites to $\gamma_{\mathbf{List}}(0)$ when applying $\mathcal{G}$ from right to left. The sequence from $\mathsf{qs}(\gamma_{\mathbf{List}}(n))$ to $\mathsf{cons}(\mathsf{zero}, \mathsf{nil})$ (resp. $\mathsf{cons}(\mathsf{zero}, \mathsf{cons}(\mathsf{zero}, \mathsf{nil}))$) uses the recursive qs-rule (2) once (resp. twice), i.e., it has recursion depth 1 (resp. 2). Moreover, these $\rightsquigarrow$-normal forms rewrite to $\gamma_{\mathbf{List}}(1)$ (resp. $\gamma_{\mathbf{List}}(2)$) when using $\mathcal{G}$ from right to left.

A sample point (t, σ, d) for a narrowing tree with the root $s = f(\dots)$ represents the *sample conjecture* $s\sigma \xrightarrow{\mathrm{i}}^* t$, which stands for a reduction with d applications of recursive f-rules. So for $s = \mathsf{qs}(\gamma_{\mathbf{List}}(n))$, the sample points in (7) represent the sample conjectures $\mathsf{qs}(\gamma_{\mathbf{List}}(0)) \xrightarrow{\mathrm{i}}^* \gamma_{\mathbf{List}}(0)$, $\mathsf{qs}(\gamma_{\mathbf{List}}(1)) \xrightarrow{\mathrm{i}}^* \gamma_{\mathbf{List}}(1)$, $\mathsf{qs}(\gamma_{\mathbf{List}}(2)) \xrightarrow{\mathrm{i}}^* \gamma_{\mathbf{List}}(2)$. Now the goal is to speculate a general conjecture from these sample conjectures (whose validity must be proved afterwards).

In general, we search for a maximal subset of sample conjectures that are suitable for generalization. More precisely, if s is the root of the narrowing tree, then we take a maximal subset S_{max} of sample points such that for all $(t, \sigma, d), (t', \sigma', d') \in S_{max}$, the sample conjectures $s\sigma \xrightarrow{\mathrm{i}}^* t$ and $s\sigma' \xrightarrow{\mathrm{i}}^* t'$ are identical up to the occurring natural numbers and the variable names. For instance, $\mathsf{qs}(\gamma_{\mathbf{List}}(0)) \xrightarrow{\mathrm{i}}^* \gamma_{\mathbf{List}}(0)$, $\mathsf{qs}(\gamma_{\mathbf{List}}(1)) \xrightarrow{\mathrm{i}}^* \gamma_{\mathbf{List}}(1)$, and $\mathsf{qs}(\gamma_{\mathbf{List}}(2)) \xrightarrow{\mathrm{i}}^* \gamma_{\mathbf{List}}(2)$ are indeed identical up to the numbers in these sample conjectures. To obtain a general conjecture, we replace all numbers in the sample conjectures by polynomials. So in our example, we want to speculate a conjecture of the form $\mathsf{qs}(\gamma_{\mathbf{List}}(pol^{left})) \xrightarrow{\mathrm{i}}^* \gamma_{\mathbf{List}}(pol^{right})$. Here, pol^{left} and pol^{right} are polynomials in one variable n (the *induction variable* of the conjecture) that stands for the recursion depth. This facilitates a proof of the resulting conjecture by induction on n.

[4] We always simplify arithmetic expressions in terms and substitutions, e.g., the substitution $[n/0+1]$ in the second sample point is simplified to $[n/1]$.

So in general, in any sample conjecture $s\sigma \xrightarrow{\text{i}}{}^* t$ that correspond to a sample point $(t,\sigma,d) \in S_{max}$, we replace the natural numbers in $s\sigma$ and t by polynomials. For any term q, let $\mathrm{pos}(q)$ be the set of its positions and $\Pi_{\mathbb{N}}^q = \{\pi \in \mathrm{pos}(q) \mid q|_\pi \in \mathbb{N}\}$. Then for each $\pi \in \Pi_{\mathbb{N}}^{s\sigma}$ (resp. $\pi \in \Pi_{\mathbb{N}}^{t}$) with $(t,\sigma,d) \in S_{max}$, we search for a polynomial pol_π^{left} (resp. pol_π^{right}). To this end, for every sample point $(t,\sigma,d) \in S_{max}$, we generate the constraints

$$\text{“}pol_\pi^{left}(d) = s\sigma|_\pi\text{” for every } \pi \in \Pi_{\mathbb{N}}^{s\sigma} \quad \text{and} \quad \text{“}pol_\pi^{right}(d) = t|_\pi\text{” for every } \pi \in \Pi_{\mathbb{N}}^{t}. \tag{8}$$

Here, pol_π^{left} and pol_π^{right} are polynomials with abstract coefficients. So if one searches for polynomials of degree e, then the polynomials have the form $c_0 + c_1 \cdot n + c_2 \cdot n^2 + \ldots + c_e \cdot n^e$ and the constraints in (8) are linear diophantine equations over the unknown coefficients $c_i \in \mathbb{N}$.[5] These equations can easily be solved automatically. Finally, the desired generalized speculated conjecture is obtained from $s\sigma \xrightarrow{\text{i}}{}^* t$ by replacing $s\sigma|_\pi$ with pol_π^{left} for every $\pi \in \Pi_{\mathbb{N}}^{s\sigma}$ and by replacing $t|_\pi$ with pol_π^{right} for every $\pi \in \Pi_{\mathbb{N}}^{t}$.

▶ **Example 6** (Speculating Conjectures)**.** In Ex. 4, we narrowed $s=\mathsf{qs}(\gamma_{\mathbf{List}}(n))$ and S_{max} is the set S in (7). For each $(t,\sigma,d) \in S_{max}$, we have $\Pi_{\mathbb{N}}^{s\sigma}=\{1.1\}$ and $\Pi_{\mathbb{N}}^{t}=\{1\}$. So from the sample conjecture $\mathsf{qs}(\gamma_{\mathbf{List}}(0)) \xrightarrow{\text{i}}{}^* \gamma_{\mathbf{List}}(0)$, where the recursion depth is $d{=}0$, we obtain the constraints $pol_{1.1}^{left}(d)=pol_{1.1}^{left}(0)=\mathsf{qs}(\gamma_{\mathbf{List}}(0))|_{1.1}=0$ and $pol_1^{right}(d)=pol_1^{right}(0)=\gamma_{\mathbf{List}}(0)|_1=0$. Similarly, from the two other sample conjectures we get $pol_{1.1}^{left}(1)=pol_1^{right}(1)=1$ and $pol_{1.1}^{left}(2)=pol_1^{right}(2)=2$. When using $pol_{1.1}^{left}=c_0+c_1\cdot n+c_2\cdot n^2$ and $pol_1^{right}=d_0+d_1\cdot n+d_2\cdot n^2$ with the abstract coefficients $c_0,\ldots,c_2,d_0,\ldots,d_2$, the solution $c_0=c_2=d_0=d_2=0$, $c_1=d_1=1$ (i.e., $pol_{1.1}^{left}=n$ and $pol_1^{right}=n$) is easily found automatically. So the resulting conjecture is $\mathsf{qs}(\gamma_{\mathbf{List}}(pol_{1.1}^{left})) \xrightarrow{\text{i}}{}^* \gamma_{\mathbf{List}}(pol_1^{right})$, i.e., $\mathsf{qs}(\gamma_{\mathbf{List}}(n)) \xrightarrow{\text{i}}{}^* \gamma_{\mathbf{List}}(n)$.

If S_{max} contains sample points with e different recursion depths, then we generate polynomials of at most degree $e-1$ satisfying the constraints (8) (these polynomials are determined uniquely). Ex. 7 shows how to speculate conjectures with *several* variables.

▶ **Example 7** (Conjecture With Several Variables)**.** The following TRS combines half and plus.

$$\mathsf{hp}(\mathsf{zero},y) \to y \qquad \mathsf{hp}(\mathsf{succ}(\mathsf{succ}(x)),y) \to \mathsf{succ}(\mathsf{hp}(x,y))$$

Narrowing $s = \mathsf{hp}(\gamma_{\mathbf{Nats}}(n_1),\gamma_{\mathbf{Nats}}(n_2))$ yields the sample points $(\gamma_{\mathbf{Nats}}(n_2),[n_1/0],0)$, $(\gamma_{\mathbf{Nats}}(n_2+1),[n_1/2],1)$, $(\gamma_{\mathbf{Nats}}(n_2+2),[n_1/4],2)$, and $(\gamma_{\mathbf{Nats}}(n_2+3),[n_1/6],3)$. For the last three sample points (t,σ,d), the only number in $s\sigma$ is at position 1.1 and the polynomial $pol_{1.1}^{left} = 2\cdot n$ satisfies the constraint $pol_{1.1}^{left}(d) = s\sigma|_{1.1}$. Moreover, the only number in t is at position 1.2 and the polynomial $pol_{1.2}^{right} = n$ satisfies $pol_{1.2}^{right} = t|_{1.2}$. Thus, we speculate the conjecture $\mathsf{hp}(\gamma_{\mathbf{Nats}}(2\cdot n),\gamma_{\mathbf{Nats}}(n_2)) \xrightarrow{\text{i}}{}^* \gamma_{\mathbf{Nats}}(n_2+n)$ with the induction variable n.

3 Proving Rewrite Lemmas

If the proof of a speculated conjecture succeeds, then we have found a *rewrite lemma.*

▶ **Definition 8** (Rewrite Lemmas)**.** Let $\mathcal{R}$ be a TRS that is well typed w.r.t. Σ' and $\mathcal{V}'$. For any term q, let $q{\downarrow_{\mathcal{G}/\mathcal{A}}}$ be q's normal form w.r.t. $\mathcal{G}_\mathcal{R}$, where the generator equations are applied from left to right and $\mathcal{A}$-equivalent (sub)terms are considered to be equal. Moreover, let $s \xrightarrow{\text{i}}{}^* t$ be a conjecture with $\mathcal{V}(s) = \{n_1,\ldots,n_m\} \neq \varnothing$, where $\overline{n} = (n_1,\ldots,n_m)$ are pairwise

[5] Note that in the constraints (8), n is instantiated by an actual number d. Thus, if $pol_\pi^{left} = c_0 + c_1\cdot n + c_2\cdot n^2 + \ldots + c_e\cdot n^e$, then $pol_\pi^{left}(d)$ is a *linear* polynomial over the unknowns $c_0,\ldots,c_e$.

different variables of type $\mathbb{N}$, s is well typed, $\mathrm{root}(s) \in \Sigma_{def}(\mathcal{R})$, and s has no defined symbol from $\Sigma_{def}(\mathcal{R})$ below the root. Let $rt : \mathbb{N}^m \to \mathbb{N}$. Then $s \to_{\mathcal{R}}^{rt(\overline{n})} t$ is a *rewrite lemma* for $\mathcal{R}$ iff $s\sigma\!\downarrow_{\mathcal{G}/\mathcal{A}} \xrightarrow{\mathrm{i}}_{\mathcal{R}}^{rt(\overline{n}\sigma)} t\sigma\!\downarrow_{\mathcal{G}/\mathcal{A}}$ for all $\sigma : \mathcal{V}(s) \to \mathbb{N}$, i.e., $s\sigma\!\downarrow_{\mathcal{G}/\mathcal{A}}$ can be reduced to $t\sigma\!\downarrow_{\mathcal{G}/\mathcal{A}}$ in exactly $rt(n_1\sigma, \dots, n_m\sigma)$ innermost $\mathcal{R}$-steps. We omit the index $\mathcal{R}$ if it is clear from the context.

So the conjecture $\mathsf{qs}(\gamma_{\mathbf{List}}(n)) \xrightarrow{\mathrm{i}}^* \gamma_{\mathbf{List}}(n)$ gives rise to a rewrite lemma, since $\sigma(n) = b \in \mathbb{N}$ implies $\mathsf{qs}(\gamma_{\mathbf{List}}(b))\!\downarrow_{\mathcal{G}/\mathcal{A}} = \mathsf{qs}(\mathsf{cons}^b(\mathsf{zero}, \mathsf{nil})) \xrightarrow{\mathrm{i}}^{3b^2+2b+1} \mathsf{cons}^b(\mathsf{zero}, \mathsf{nil}) = \gamma_{\mathbf{List}}(b)\!\downarrow_{\mathcal{G}/\mathcal{A}}$.

To prove rewrite lemmas, essentially we use rewriting with $\xrightarrow{\mathrm{i}}_{(\mathcal{R}\cup\mathcal{L})/(\mathcal{G}\cup\mathcal{A})}$.[6] However, this would allow us to prove lemmas that do not correspond to *innermost* rewriting with $\mathcal{R}$, if $\mathcal{R}$ contains rules with overlapping left-hand sides. Consider $\mathcal{R} = \{\mathsf{g}(\mathsf{zero}) \to \mathsf{zero}, \mathsf{f}(\mathsf{g}(x)) \to \mathsf{zero}\}$. We have $\mathsf{f}(\mathsf{g}(\gamma_{\mathbf{Nats}}(n))) \xrightarrow{\mathrm{i}}_{(\mathcal{R}\cup\mathcal{L})/(\mathcal{G}\cup\mathcal{A})} \mathsf{zero}$, but for the instantiation $[n/0]$, this would not be an innermost reduction. To avoid this, we use the following relation $\xrightarrow{\mathrm{i}}_{\mathcal{R}} \subseteq \xrightarrow{\mathrm{i}}_{(\mathcal{R}\cup\mathcal{L})/(\mathcal{G}\cup\mathcal{A})}$: We have $s \xrightarrow{\mathrm{i}}_{\mathcal{R}} t$ iff there exist a term s', a substitution σ, a position π, and a rule $\ell \to r \in \mathcal{R} \cup \mathcal{L}$ such that $s \equiv_{\mathcal{G}\cup\mathcal{A}} s'$, $s'|_\pi = \ell\sigma$ and $s'[r\sigma]_\pi \equiv_{\mathcal{G}\cup\mathcal{A}} t$. Moreover, if $\ell \to r \in \mathcal{R}$, then there must not be any proper non-variable subterm q of $\ell\sigma$, a (variable-renamed) rule $\ell' \to r' \in \mathcal{R}$, and a substitution σ' such that $\ell'\sigma' \equiv_{\mathcal{G}\cup\mathcal{A}} q\sigma'$. Now $\mathsf{f}(\mathsf{g}(\gamma_{\mathbf{Nats}}(n))) \not\xrightarrow{\mathrm{i}}_{\mathcal{R}} \mathsf{zero}$, because the subterm $\mathsf{g}(\gamma_{\mathbf{Nats}}(n))$ unifies with the left-hand side $\mathsf{g}(\mathsf{zero})$ modulo $\mathcal{G}\cup\mathcal{A}$.

When proving a conjecture $s \xrightarrow{\mathrm{i}}^* t$ by induction, in the step case we try to reduce $s[n/n+1]$ to $t[n/n+1]$, where one may use the rule IH: $s \to t$ as induction hypothesis. Here, the variables in IH may not be instantiated. The reason for not allowing instantiations of the non-induction variables from $\mathcal{V}(s) \setminus \{n\}$ is that such induction proofs are particularly suitable for inferring runtimes of rewrite lemmas, cf. Sect. 4.

Thus, for any rule IH: $\ell \to r$, let $s \mapsto_{\mathrm{IH}} t$ iff there exist a term s' and a position π such that $s \equiv_{\mathcal{G}\cup\mathcal{A}} s'$, $s'|_\pi = \ell$ and $s'[r]_\pi \equiv_{\mathcal{G}\cup\mathcal{A}} t$. Let $\xrightarrow{\mathrm{i}}_{(\mathcal{R},\mathrm{IH})} = \xrightarrow{\mathrm{i}}_{\mathcal{R}} \cup \mapsto_{\mathrm{IH}}$. Moreover, $\xrightarrow{\mathrm{i}}_{\mathcal{R}}^*$ (resp. $\xrightarrow{\mathrm{i}}_{(\mathcal{R},\mathrm{IH})}^*$) denotes the transitive-reflexive closure of $\xrightarrow{\mathrm{i}}_{\mathcal{R}}$ (resp. $\xrightarrow{\mathrm{i}}_{(\mathcal{R},\mathrm{IH})}^*$), where in addition $s \xrightarrow{\mathrm{i}}_{\mathcal{R}}^* s'$ and $s \xrightarrow{\mathrm{i}}_{(\mathcal{R},\mathrm{IH})}^* s'$ also hold if $s \equiv_{\mathcal{G}\cup\mathcal{A}} s'$. Thm. 9 shows which rewrite sequences are needed to prove a conjecture $s \xrightarrow{\mathrm{i}}^* t$ by induction on its induction variable n.

▶ **Theorem 9** (Proving Rewrite Lemmas). *Let $\mathcal{R}$, s, t be as in Def. 8, $n \in \mathcal{V}(s) = \{n_1, \dots, n_m\}$, and $\overline{n} = (n_1, \dots, n_m)$. If $s[n/0] \xrightarrow{\mathrm{i}}_{\mathcal{R}}^* t[n/0]$ and $s[n/n+1] \xrightarrow{\mathrm{i}}_{(\mathcal{R},\mathrm{IH})}^* t[n/n+1]$, where* IH *is the rule $s \to t$, then there is an $rt : \mathbb{N}^m \to \mathbb{N}$ such that $s \xrightarrow{\mathrm{i}}^{rt(\overline{n})} t$ is a rewrite lemma for $\mathcal{R}$.*

▶ **Example 10** (Proof of Rewrite Lemma). Assume that we have already proved the rewrite lemmas (5) and (6). To prove the conjecture $\mathsf{qs}(\gamma_{\mathbf{List}}(n)) \xrightarrow{\mathrm{i}}^* \gamma_{\mathbf{List}}(n)$, in the induction base we show $\mathsf{qs}(\gamma_{\mathbf{List}}(0)) \xrightarrow{\mathrm{i}}_{\mathcal{R}} \gamma_{\mathbf{List}}(0)$ and in the induction step, we obtain $\mathsf{qs}(\gamma_{\mathbf{List}}(n+1)) \xrightarrow{\mathrm{i}}_{\mathcal{R}}^* \mathsf{nil} \mathbin{++} \mathsf{cons}(\mathsf{zero}, \mathsf{qs}(\gamma_{\mathbf{List}}(n))) \mapsto_{\mathrm{IH}} \mathsf{nil} \mathbin{++} \mathsf{cons}(\mathsf{zero}, \gamma_{\mathbf{List}}(n)) \xrightarrow{\mathrm{i}}_{\mathcal{R}} \gamma_{\mathbf{List}}(n+1)$. Thus, there is a rewrite lemma $\mathsf{qs}(\gamma_{\mathbf{List}}(n)) \xrightarrow{\mathrm{i}}^{rt(n)} \gamma_{\mathbf{List}}(n)$. Sect. 4 will clarify how to find the function rt.

4 Inferring Bounds for Rewrite Lemmas

Now we show how to infer the function rt for a rewrite lemma $s \xrightarrow{\mathrm{i}}^{rt(\overline{n})} t$ from its proof. If $n \in \overline{n}$ was the induction variable and the induction hypothesis was applied ih times in the induction step, then we get the following recurrence equations for rt where $\widetilde{n}$ is $\overline{n}$ without the variable n:

$$rt(\overline{n}[n/0]) = ib(\widetilde{n}) \qquad \text{and} \qquad rt(\overline{n}[n/n+1]) = ih \cdot rt(\overline{n}) + is(\overline{n}) \tag{9}$$

[6] Here, we define $\xrightarrow{\mathrm{i}}_{(\mathcal{R}\cup\mathcal{L})/(\mathcal{G}\cup\mathcal{A})}$ to be the relation $\equiv_{\mathcal{G}\cup\mathcal{A}} \circ (\xrightarrow{\mathrm{i}}_{\mathcal{R}} \cup \to_{\mathcal{L}}) \circ \equiv_{\mathcal{G}\cup\mathcal{A}}$. An adaption of our approach to runtime complexity of full rewriting is obtained by considering $\to_{(\mathcal{R}\cup\mathcal{L})/(\mathcal{G}\cup\mathcal{A})}$ instead.

Here, $\mathit{ib}(\widetilde{n})$ is the length of the reduction $s[n/0]{\downarrow_{\mathcal{G}/\mathcal{A}}} \xrightarrow{\text{i}}{}^*_{\mathcal{R}} t[n/0]{\downarrow_{\mathcal{G}/\mathcal{A}}}$, which must exist due to the induction base. The addend $\mathit{is}(\overline{n})$ is the length of $s[n/n+1]{\downarrow_{\mathcal{G}/\mathcal{A}}} \xrightarrow{\text{i}}{}^*_{\mathcal{R}} t[n/n+1]{\downarrow_{\mathcal{G}/\mathcal{A}}}$, but without those subsequences that are covered by the induction hypothesis IH. Since the non-induction variables were not instantiated in IH, $\mathit{rt}(\overline{n})$ is the runtime for each application of IH. To compute ib and is, for each previous rewrite lemma $s' \xrightarrow{\text{i}}{}^{\mathit{rt}'(\overline{n}')} t'$ that was used in the proof of $s \xrightarrow{\text{i}}{}^{\mathit{rt}(\overline{n})} t$, we assume that rt' is known. Thus, rt' can be used to infer the number of rewrite steps represented by that previous lemma. To avoid treating rules and rewrite lemmas separately, in Def. 11 we regard each rule $s \to t \in \mathcal{R}$ as a rewrite lemma $s \xrightarrow{\text{i}}{}^1 t$.

▶ **Definition 11** ($\mathit{ih}, \mathit{ib}, \mathit{is}$)**.** Let $s \xrightarrow{\text{i}}{}^{\mathit{rt}(\overline{n})} t$ be a rewrite lemma with an induction proof as in Thm. 9. More precisely, let $u_1 \xrightarrow{\text{i}}_{\mathcal{R}} \ldots \xrightarrow{\text{i}}_{\mathcal{R}} u_{b+1}$ be the rewrite sequence $s[n/0] \xrightarrow{\text{i}}{}^*_{\mathcal{R}} t[n/0]$ for the induction base and let $v_1 \xrightarrow{\text{i}}_{(\mathcal{R},\text{IH})} \ldots \xrightarrow{\text{i}}_{(\mathcal{R},\text{IH})} v_{k+1}$ be the rewrite sequence $s[n/n+1] \xrightarrow{\text{i}}{}^*_{(\mathcal{R},\text{IH})} t[n/n+1]$ for the induction step, where IH: $s \to t$ is applied ih times. For $j \in \{1,\ldots,b\}$, let $\ell_j \xrightarrow{\text{i}}{}^{\mathit{rt}_j(\overline{y}_j)} r_j$ and σ_j be the rewrite lemma and substitution used to reduce u_j to u_{j+1}. Similarly for $j \in \{1,\ldots,k\}$, let $p_j \to^{\mathit{rt}'_j(\overline{z}_j)} q_j$ and θ_j be the lemma and substitution used to reduce v_j to v_{j+1}. Then we define:

$$\mathit{ib}(\widetilde{n}) \;=\; \sum_{j\in\{1,\ldots,b\}} \mathit{rt}_j(\overline{y}_j\sigma_j) \qquad \text{and} \qquad \mathit{is}(\overline{n}) \;=\; \sum_{j\in\{1,\ldots,k\},\; p_j\to q_j\neq \text{IH}} \mathit{rt}'_j(\overline{z}_j\theta_j)$$

By solving the recurrence equations (9), we can now compute rt explicitly.

▶ **Theorem 12** (Explicit Runtime of Rewrite Lemmas)**.** *Let $s \xrightarrow{\text{i}}{}^{\mathit{rt}(\overline{n})} t$ be a rewrite lemma, where ih, ib, and is are as in Def. 11. Then we obtain $\mathit{rt}(\overline{n}) = \mathit{ih}^n \cdot \mathit{ib}(\widetilde{n}) + \sum_{i=0}^{n-1} \mathit{ih}^{n-1-i} \cdot \mathit{is}(\overline{n}[n/i])$.*

▶ **Example 13** (Computing rt)**.** Reconsider $\mathsf{qs}(\gamma_{\mathbf{List}}(n)) \xrightarrow{\text{i}}{}^{\mathit{rt}(n)} \gamma_{\mathbf{List}}(n)$ from Ex. 10. The proof of the induction base is $\mathsf{qs}(\gamma_{\mathbf{List}}(0)) \equiv_{\mathcal{G}} \mathsf{qs}(\mathsf{nil}) \xrightarrow{\text{i}}_{\mathcal{R}_{\mathsf{qs}}} \mathsf{nil} \equiv_{\mathcal{G}} \gamma_{\mathbf{List}}(0)$. Hence, $\mathit{ib} = \mathit{rt}_1 = 1$. The proof of the induction step is as follows. Here, we use that the runtime of both previously proved lemmas (5) and (6) is $3n+1$. Note that the non-overlap condition required by the relation $\xrightarrow{\text{i}}_{\mathcal{R}_{\mathsf{qs}}}$ is clearly satisfied in all steps with $\xrightarrow{\text{i}}_{\mathcal{R}_{\mathsf{qs}}}$ in the proof.

$$\begin{array}{rcll|l}
\mathsf{qs}(\gamma_{\mathbf{List}}(n+1)) & \equiv_{\mathcal{G}} & \mathsf{qs}(\mathsf{cons}(\gamma_{\mathbf{Nats}}(0), \gamma_{\mathbf{List}}(n))) & \xrightarrow{\text{i}}_{\mathcal{R}_{\mathsf{qs}}} & \mathit{rt}'_1 = 1 \\
\multicolumn{3}{r}{\mathsf{qs}(\mathsf{low}(\gamma_{\mathbf{Nats}}(0), \gamma_{\mathbf{List}}(n)))\ {+}{+}\ \mathsf{cons}(\gamma_{\mathbf{Nats}}(0), \mathsf{qs}(\mathsf{high}(\ldots)))} & \to_{\mathcal{L}} & \mathit{rt}'_2(n) = 3n+1 \\
\multicolumn{3}{r}{\mathsf{qs}(\gamma_{\mathbf{List}}(0))\ {+}{+}\ \mathsf{cons}(\gamma_{\mathbf{Nats}}(0), \mathsf{qs}(\mathsf{high}(\gamma_{\mathbf{Nats}}(0), \gamma_{\mathbf{List}}(n))))} & \to_{\mathcal{L}} & \mathit{rt}'_3(n) = 3n+1 \\
\multicolumn{3}{r}{\mathsf{qs}(\gamma_{\mathbf{List}}(0))\ {+}{+}\ \mathsf{cons}(\gamma_{\mathbf{Nats}}(0), \mathsf{qs}(\gamma_{\mathbf{List}}(n)))} & \equiv_{\mathcal{G}} & \\
\multicolumn{3}{r}{\mathsf{qs}(\mathsf{nil})\ {+}{+}\ \mathsf{cons}(\mathsf{zero}, \mathsf{qs}(\gamma_{\mathbf{List}}(n)))} & \xrightarrow{\text{i}}_{\mathcal{R}_{\mathsf{qs}}} & \mathit{rt}'_4 = 1 \\
\multicolumn{3}{r}{\mathsf{nil}\ {+}{+}\ \mathsf{cons}(\mathsf{zero}, \mathsf{qs}(\gamma_{\mathbf{List}}(n)))} & \mapsto_{\text{IH}} & \mathit{rt}'_5(n) = \mathit{rt}(n) \\
\multicolumn{3}{r}{\mathsf{nil}\ {+}{+}\ \mathsf{cons}(\mathsf{zero}, \gamma_{\mathbf{List}}(n))} & \xrightarrow{\text{i}}_{\mathcal{R}_{\mathsf{qs}}} & \mathit{rt}'_6 = 1 \\
\multicolumn{3}{r}{\mathsf{cons}(\mathsf{zero}, \gamma_{\mathbf{List}}(n)) \;\equiv_{\mathcal{G}}\; \gamma_{\mathbf{List}}(n+1)} & &
\end{array}$$

$$\begin{array}{lcccl}
\text{Hence, } \mathit{is}(n) & = & \sum_{j\in\{1,\ldots,6\},\; p_j\to q_j\neq \text{IH}} \mathit{rt}'_j(\overline{z}_j\theta_j) & = & \mathit{rt}'_1 + \mathit{rt}'_2(n) + \mathit{rt}'_3(n) + \mathit{rt}'_4 + \mathit{rt}'_6 \\
 & = & 1 + (3n+1) + (3n+1) + 1 + 1 & = & 6n+5.
\end{array}$$

In our example, we have $\mathit{ih} = 1$. So Thm. 12 implies $\mathit{rt}(n) = \mathit{ib} + \sum_{i=0}^{n-1} \mathit{is}(i) = 1 + \sum_{i=0}^{n-1}(6i+5)$ $= 3n^2 + 2n + 1$. Thus, we get the rewrite lemma (4): $\mathsf{qs}(\gamma_{\mathbf{List}}(n)) \xrightarrow{\text{i}}{}^{3n^2+2n+1} \gamma_{\mathbf{List}}(n)$.

To compute asymptotic bounds for the complexity of a TRS afterwards, we have to infer asymptotic bounds for the runtime of rewrite lemmas. Based on Thm. 12, such bounds can be automatically obtained from the induction proofs of the lemmas. To ease the formulation of bounds for $\mathit{rt} : \mathbb{N}^m \to \mathbb{N}$, we define the unary function $\mathit{rt}_{\mathbb{N}} : \mathbb{N} \to \mathbb{N}$ as $\mathit{rt}_{\mathbb{N}}(n) = \mathit{rt}(n,\ldots,n)$.

If the induction hypothesis was not used in the proof of a rewrite lemma (i.e., $\mathit{ih} = 0$), then we have $\mathit{rt}(\overline{n}[n/0]) = \mathit{ib}(\widetilde{n})$ and $\mathit{rt}(\overline{n}[n/n+1]) = \mathit{is}(\overline{n})$. Thus, if ib and is are polynomials of degree d_{ib} and d_{is}, respectively, then we obtain $\mathit{rt}_{\mathbb{N}}(n) \in \Omega(n^{\max\{d_{\mathit{ib}},d_{\mathit{is}}\}})$.

If $\mathit{ih} = 1$, then Thm. 12 implies $\mathit{rt}(\overline{n}) = \mathit{ib}(\widetilde{n}) + \sum_{i=0}^{n-1} \mathit{is}(\overline{n}[n/i])$. Again, let ib and is be polynomials of degree d_{ib} and d_{is}, respectively. Then $\mathit{is}(\overline{n}) = t_0+t_1 n+t_2 n^2+\ldots+t_{d_{\mathit{is}}} n^{d_{\mathit{is}}}$, where the t_j are polynomials of degree at most $d_{\mathit{is}} - j$ containing variables from $\widetilde{n}$. Hence, $\mathit{rt}(\overline{n}) =$

$$\mathit{ib}(\widetilde{n}) + \sum_{i=0}^{n-1}(t_0 + t_1 i + t_2 i^2 + \ldots + t_{d_{\mathit{is}}} i^{d_{\mathit{is}}}) = \mathit{ib}(\widetilde{n}) + t_0 \cdot \sum_{i=0}^{n-1} i^0 + t_1 \cdot \sum_{i=0}^{n-1} i^1 + t_2 \cdot \sum_{i=0}^{n-1} i^2 + \ldots + t_{d_{\mathit{is}}} \cdot \sum_{i=0}^{n-1} i^{d_{\mathit{is}}}.$$

By Faulhaber's formula [15], for any $e \in \mathbb{N}$, $\sum_{i=0}^{n-1} i^e$ is a polynomial over the variable n of degree $e+1$. For example if $e = 1$, then $\sum_{i=0}^{n-1} i^1 = \frac{n\cdot(n-1)}{2}$ has degree 2. By taking also the degree d_{ib} of ib into account, rt has degree $\max\{d_{\mathit{ib}}, d_{\mathit{is}} + 1\}$, i.e., $\mathit{rt}_{\mathbb{N}}(n) \in \Omega(n^{\max\{d_{\mathit{ib}},d_{\mathit{is}}+1\}})$.

Finally we consider the case where the induction hypothesis was used several times, i.e., $\mathit{ih} > 1$. By construction we always have $\mathit{is}(\overline{n}) \geq 1$ (since the induction step cannot only consist of applying the induction hypothesis). Thus, Thm. 12 implies $\mathit{rt}(\overline{n}) \geq \sum_{i=0}^{n-1} \mathit{ih}^{n-1-i} = \sum_{j=0}^{n-1} \mathit{ih}^j = \frac{\mathit{ih}^n-1}{\mathit{ih}-1}$. So $\mathit{rt}_{\mathbb{N}}(n) \in \Omega(\mathit{ih}^n)$, i.e., the runtime of the rewrite lemma is exponential.

▶ **Theorem 14** (Asymptotic Runtime of Rewrite Lemmas). *Let $s \xrightarrow{\mathsf{i}}{}^{\mathit{rt}(\overline{n})} t$ be a rewrite lemma with ih, ib, and is as in Def. 11. Moreover, let ib and is be polynomials of degree d_{ib} and d_{is}.*
- *If $\mathit{ih} = 0$, then $\mathit{rt}_{\mathbb{N}}(n) \in \Omega(n^{\max\{d_{\mathit{ib}},d_{\mathit{is}}\}})$.*
- *If $\mathit{ih} = 1$, then $\mathit{rt}_{\mathbb{N}}(n) \in \Omega(n^{\max\{d_{\mathit{ib}},d_{\mathit{is}}+1\}})$.*
- *If $\mathit{ih} > 1$, then $\mathit{rt}_{\mathbb{N}}(n) \in \Omega(\mathit{ih}^n)$.*

▶ **Example 15** (Exponential Runtime). Consider the TRS $\mathcal{R}_{exp}$ with the rules $\mathsf{f}(\mathsf{succ}(x), \mathsf{succ}(x)) \to \mathsf{f}(\mathsf{f}(x, x), \mathsf{f}(x, x))$ and $\mathsf{f}(\mathsf{zero}, \mathsf{zero}) \to \mathsf{zero}$. Our approach speculates and proves the rewrite lemma $\mathsf{f}(\gamma_{\mathbf{Nats}}(n), \gamma_{\mathbf{Nats}}(n)) \xrightarrow{\mathsf{i}}{}^{\mathit{rt}(n)} \mathsf{zero}$. For the induction base, we have $\mathsf{f}(\gamma_{\mathbf{Nats}}(0), \gamma_{\mathbf{Nats}}(0)) \equiv_{\mathcal{G}} \mathsf{f}(\mathsf{zero}, \mathsf{zero}) \xrightarrow{\mathsf{i}}_{\mathcal{R}_{exp}} \mathsf{zero}$ and thus $\mathit{ib} = 1$. The induction step is proved as follows:

$$\begin{array}{rcrl|r}
\mathsf{f}(\gamma_{\mathbf{Nats}}(n+1), \gamma_{\mathbf{Nats}}(n+1)) & \equiv_{\mathcal{G}} & \mathsf{f}(\mathsf{succ}(\gamma_{\mathbf{Nats}}(n)), \mathsf{succ}(\gamma_{\mathbf{Nats}}(n))) & \xrightarrow{\mathsf{i}}_{\mathcal{R}_{exp}} & \mathit{rt}'_1 = 1 \\
 & & \mathsf{f}(\mathsf{f}(\gamma_{\mathbf{Nats}}(n), \gamma_{\mathbf{Nats}}(n)), \mathsf{f}(\gamma_{\mathbf{Nats}}(n), \gamma_{\mathbf{Nats}}(n))) & \mapsto^2_{\mathrm{IH}} & \\
 & & \mathsf{f}(\mathsf{zero}, \mathsf{zero}) & \xrightarrow{\mathsf{i}}_{\mathcal{R}_{exp}} & \mathit{rt}'_4 = 1 \\
 & & \mathsf{zero} & &
\end{array}$$

Thus, $\mathit{ih} = 2$ and $\mathit{is}(n)$ is the constant 2 for all $n \in \mathbb{N}$. Hence, by Thm. 14 we have $\mathit{rt}(n) \in \Omega(2^n)$. Indeed, Thm. 12 implies $\mathit{rt}(n) = 2^n + \sum_{i=0}^{n-1} 2^{n-1-i} \cdot 2 = 2^{n+1} + 2^n - 2$.

5 Inferring Bounds for TRSs

We now use rewrite lemmas to infer lower bounds for the *innermost runtime complexity* $\mathrm{irc}_{\mathcal{R}}$ of a TRS $\mathcal{R}$. To define $\mathrm{irc}_{\mathcal{R}}$, the *derivation height* of a term t w.r.t. a relation $\to$ is the length of the longest $\to$-sequence starting with t, i.e., $\mathrm{dh}(t, \to) = \sup\{ m \mid \exists t' \in \mathcal{T}(\Sigma, \mathcal{V}),\ t \to^m t' \}$, cf. [13]. Here, for any $M \subseteq \mathbb{N} \cup \{\omega\}$, $\sup M$ is the least upper bound of M and $\sup \varnothing = 0$. Since we only regard finite TRSs, $\mathrm{dh}(t, \xrightarrow{\mathsf{i}}_{\mathcal{R}}) = \omega$ iff t starts an infinite sequence of $\xrightarrow{\mathsf{i}}_{\mathcal{R}}$-steps. So as in [17], dh treats terminating and non-terminating terms in a uniform way.

When analyzing the complexity of *programs*, one is interested in evaluations of *basic terms* $f(t_1, \ldots, t_k)$ where a defined symbol $f \in \Sigma_{def}(\mathcal{R})$ is applied to data objects $t_1, \ldots, t_k \in \mathcal{T}(\Sigma_{con}(\mathcal{R}), \mathcal{V})$. The *innermost runtime complexity function* $\mathrm{irc}_{\mathcal{R}}$ corresponds to the usual notion of "complexity" for programs. It maps any $n \in \mathbb{N}$ to the length of the longest sequence of $\xrightarrow{\mathsf{i}}_{\mathcal{R}}$-steps starting with a basic term t with $|t| \leq n$. Here, the *size* of a term is $|x| = 1$ for $x \in \mathcal{V}$ and $|f(t_1, \ldots, t_k)| = 1 + |t_1| + \ldots + |t_k|$, and $\mathcal{T}_B$ is the set of all basic terms.

▶ **Definition 16** (Innermost Runtime Complexity $\mathrm{irc}_\mathcal{R}$ [12]). For a TRS $\mathcal{R}$, its *innermost runtime complexity function* $\mathrm{irc}_\mathcal{R} : \mathbb{N} \to \mathbb{N} \cup \{\omega\}$ is $\mathrm{irc}_\mathcal{R}(n) = \sup\{ \mathrm{dh}(t, \xrightarrow{\mathrm{i}}_\mathcal{R}) \mid t \in \mathcal{T}_B, |t| \leq n \}$.

In Sect. 4 we computed the length $rt(\overline{n})$ of the rewrite sequences represented by a rewrite lemma $s \xrightarrow{\mathrm{i}}^{rt(\overline{n})} t$, where $\mathcal{V}(s) = \overline{n}$. However, $\mathrm{irc}_\mathcal{R}$ is defined w.r.t. the size of the start term of a rewrite sequence. Thus, to obtain a lower bound for $\mathrm{irc}_\mathcal{R}$ from $rt(\overline{n})$, for any $\sigma : \mathcal{V}(s) \to \mathbb{N}$ one has to take the relation between $\overline{n}\sigma$ and the size of the start term $s\sigma\!\downarrow_{\mathcal{G}/\mathcal{A}}$ into account. Note that our approach in Sect. 2 only speculates lemmas where s has the form $f(\gamma_{\tau_1}(s_1), \ldots, \gamma_{\tau_k}(s_k))$. Here, $f \in \Sigma_{def}(\mathcal{R})$, $s_1, \ldots, s_k$ are polynomials over $\overline{n}$, and $\tau_1, \ldots, \tau_k$ are simply structured types. For any τ_i, let $d_{\tau_i} : \rho_1 \times \cdots \times \rho_b \to \tau$ be τ_i's recursive constructor. Then for any $n \in \mathbb{N}$, Def. 3 implies $|\gamma_{\tau_i}(n)\!\downarrow_{\mathcal{G}/\mathcal{A}}| = sz_{\tau_i}(n)$ for $sz_{\tau_i} : \mathbb{N} \to \mathbb{N}$ with

$$sz_{\tau_i}(n) = |\gamma_{\tau_i}(0)\!\downarrow_{\mathcal{G}/\mathcal{A}}| + n \cdot \left(1 + |\gamma_{\rho_1}(0)\!\downarrow_{\mathcal{G}/\mathcal{A}}| + \cdots + |\gamma_{\rho_b}(0)\!\downarrow_{\mathcal{G}/\mathcal{A}}| - |\gamma_{\tau_i}(0)\!\downarrow_{\mathcal{G}/\mathcal{A}}|\right).$$

The reason is that $\gamma_{\tau_i}(n)\!\downarrow_{\mathcal{G}/\mathcal{A}}$ contains n occurrences of d_{τ_i} and of each $\gamma_{\rho_1}(0)\!\downarrow_{\mathcal{G}/\mathcal{A}}, \ldots, \gamma_{\rho_b}(0)\!\downarrow_{\mathcal{G}/\mathcal{A}}$ except $\gamma_{\tau_i}(0)\!\downarrow_{\mathcal{G}/\mathcal{A}}$, and just one occurrence of $\gamma_{\tau_i}(0)\!\downarrow_{\mathcal{G}/\mathcal{A}}$. For instance, $|\gamma_{\mathbf{Nats}}(n)\!\downarrow_{\mathcal{G}/\mathcal{A}}|$ is $sz_{\mathbf{Nats}}(n) = |\gamma_{\mathbf{Nats}}(0)\!\downarrow_{\mathcal{G}/\mathcal{A}}| + n \cdot (1 + |\gamma_{\mathbf{Nats}}(0)\!\downarrow_{\mathcal{G}/\mathcal{A}}| - |\gamma_{\mathbf{Nats}}(0)\!\downarrow_{\mathcal{G}/\mathcal{A}}|) = |\mathsf{zero}| + n = 1 + n$ and $|\gamma_{\mathbf{List}}(n)\!\downarrow_{\mathcal{G}/\mathcal{A}}|$ is $sz_{\mathbf{List}}(n) = |\gamma_{\mathbf{List}}(0)\!\downarrow_{\mathcal{G}/\mathcal{A}}| + n \cdot (1 + |\gamma_{\mathbf{Nats}}(0)\!\downarrow_{\mathcal{G}/\mathcal{A}}|) = |\mathsf{nil}| + n \cdot (1 + |\mathsf{zero}|) = 1 + n \cdot 2$. Consequently, the size of $s\!\downarrow_{\mathcal{G}/\mathcal{A}} = f(\gamma_{\tau_1}(s_1), \ldots, \gamma_{\tau_k}(s_k))\!\downarrow_{\mathcal{G}/\mathcal{A}}$ with $\mathcal{V}(s) = \overline{n}$ is given by the following function $sz : \mathbb{N}^m \to \mathbb{N}$:

$$sz(\overline{n}) = 1 + sz_{\tau_1}(s_1) + \cdots + sz_{\tau_k}(s_k)$$

For instance, the term $\mathsf{qs}(\gamma_{\mathbf{List}}(n))\!\downarrow_{\mathcal{G}/\mathcal{A}} = \mathsf{qs}(\mathsf{cons}^n(\mathsf{zero}, \mathsf{nil}))$ has the size $sz(n) = 1 + sz_{\mathbf{List}}(n) = 2n + 2$. Since $|\gamma_\tau(0)\!\downarrow_{\mathcal{G}/\mathcal{A}}|$ is a constant for each type τ, sz is a polynomial whose degree is given by the maximal degree of the polynomials $s_1, \ldots, s_k$.

So the rewrite lemma (4) for qs states that there are terms of size $sz(n) = 2n + 2$ with reductions of length $rt(n) = 3n^2 + 2n + 1$. To determine a lower bound for $\mathrm{irc}_{\mathcal{R}_{\mathsf{qs}}}$, we construct an inverse function sz^{-1} with $(sz \circ sz^{-1})(n) = n$. In our example where $sz(n) = 2n+2$, we have $sz^{-1}(n) = \frac{n-2}{2}$ if n is even. So there are terms of size $sz(sz^{-1}(n)) = n$ with reductions of length $rt(sz^{-1}(n)) = rt(\frac{n-2}{2}) = \frac{3}{4}n^2 - 2n + 2$. Since multivariate polynomials $sz(n_1, \ldots, n_m)$ cannot be inverted, we invert the unary function $sz_\mathbb{N} : \mathbb{N} \to \mathbb{N}$ with $sz_\mathbb{N}(n) = sz(n, \ldots, n)$ instead.

Of course, inverting $sz_\mathbb{N}$ fails if $sz_\mathbb{N}$ is not injective. However, the conjectures speculated in Sect. 2 only contain polynomials with natural coefficients. Then, $sz_\mathbb{N}$ is always strictly monotonically increasing. So we only proceed if there is a $sz_\mathbb{N}^{-1} : \mathrm{img}(sz_\mathbb{N}) \to \mathbb{N}$ where $(sz_\mathbb{N} \circ sz_\mathbb{N}^{-1})(n) = n$ holds for all $n \in \mathrm{img}(sz_\mathbb{N}) = \{n \in \mathbb{N} \mid \exists v \in \mathbb{N}.\ sz_\mathbb{N}(v) = n\}$. To extend $sz_\mathbb{N}^{-1}$ to a function on $\mathbb{N}$, for any (total) function $h : M \to \mathbb{N}$ with $M \subseteq \mathbb{N}$, we define $\lfloor h \rfloor(n) : \mathbb{N} \to \mathbb{N}$ by:

$$\lfloor h \rfloor(n) = h(\max\{n' \mid n' \in M, n' \leq n\}), \text{ if } n \geq \min(M) \quad \text{and} \quad \lfloor h \rfloor(n) = 0, \text{ otherwise}$$

Using this notation, the following theorem states how we can derive lower bounds for $\mathrm{irc}_\mathcal{R}$.

▶ **Theorem 17** (Explicit Lower Bounds for $\mathrm{irc}_\mathcal{R}$). *Let $s \xrightarrow{\mathrm{i}}^{rt(n_1,\ldots,n_m)} t$ be a rewrite lemma for $\mathcal{R}$, let $sz : \mathbb{N}^m \to \mathbb{N}$ be a function such that $sz(b_1, \ldots, b_m)$ is the size of $s[n_1/b_1, \ldots, n_m/b_m]\!\downarrow_{\mathcal{G}/\mathcal{A}}$ for all $b_1, \ldots, b_m \in \mathbb{N}$, and let $sz_\mathbb{N}$'s inverse function $sz_\mathbb{N}^{-1}$ exist. Then $rt_\mathbb{N} \circ \lfloor sz_\mathbb{N}^{-1} \rfloor$ is a lower bound for $\mathrm{irc}_\mathcal{R}$, i.e., $(rt_\mathbb{N} \circ \lfloor sz_\mathbb{N}^{-1} \rfloor)(n) \leq \mathrm{irc}_\mathcal{R}(n)$ holds for all $n \in \mathbb{N}$ with $n \geq \min(\mathrm{img}(sz_\mathbb{N}))$.*

So for the rewrite lemma (4) for qs where $sz_\mathbb{N}(n) = 2n + 2$, we have $\lfloor sz_\mathbb{N}^{-1} \rfloor(n) = \lfloor \frac{n-2}{2} \rfloor \geq \frac{n-3}{2}$ and $\mathrm{irc}_{\mathcal{R}_{\mathsf{qs}}}(n) \geq rt(\lfloor sz_\mathbb{N}^{-1} \rfloor(n)) \geq rt(\frac{n-3}{2}) = \frac{3}{4}n^2 - \frac{7}{2}n + \frac{19}{4}$ for all $n \geq 2$.

However, even if $sz_\mathbb{N}^{-1}$ exists, finding resp. approximating $sz_\mathbb{N}^{-1}$ automatically can be non-trivial in general. Therefore, we now show how to obtain an asymptotic lower bound

for $\mathrm{irc}_{\mathcal{R}}$ directly from a rewrite lemma $f(\gamma_{\tau_1}(s_1), \dots, \gamma_{\tau_k}(s_k)) \xrightarrow{\mathrm{i}}{}^{rt(\overline{n})} t$ without constructing $sz_{\mathbb{N}}^{-1}$. As mentioned, if e is the maximal degree of the polynomials $s_1, \dots, s_k$, then sz is also a polynomial of degree e and thus, $sz_{\mathbb{N}}(n) \in \mathcal{O}(n^e)$. Moreover, from the induction proof of the rewrite lemma we obtain an asymptotic lower bound for $rt_{\mathbb{N}}$, cf. Thm. 14. Using these bounds, Lemma 18 can be used to infer an asymptotic lower bound for $\mathrm{irc}_{\mathcal{R}}$ directly.

▶ **Lemma 18** (Asymptotic Bounds for Function Composition). *Let $rt_{\mathbb{N}}, sz_{\mathbb{N}} : \mathbb{N} \to \mathbb{N}$ where $sz_{\mathbb{N}} \in \mathcal{O}(n^e)$ for some $e \geq 1$ and where $sz_{\mathbb{N}}$ is strictly monotonically increasing.*

- *If $rt_{\mathbb{N}}(n) \in \Omega(n^d)$ with $d \geq 0$, then $(rt_{\mathbb{N}} \circ \lfloor sz_{\mathbb{N}}^{-1} \rfloor)(n) \in \Omega(n^{\frac{d}{e}})$.*
- *If $rt_{\mathbb{N}}(n) \in \Omega(b^n)$ with $b \geq 1$, then $(rt_{\mathbb{N}} \circ \lfloor sz_{\mathbb{N}}^{-1} \rfloor)(n) \in \Omega(b^{\sqrt[e]{n}})$.*

So for the rewrite lemma $\mathsf{qs}(\gamma_{\mathbf{List}}(n)) \xrightarrow{\mathrm{i}}{}^{rt(n)} \gamma_{\mathbf{List}}(n)$ where $rt_{\mathbb{N}} = rt$ and $sz_{\mathbb{N}} = sz$, we only need the asymptotic bounds $sz(n) \in \mathcal{O}(n)$ and $rt(n) \in \Omega(n^2)$, to conclude $\mathrm{irc}_{\mathcal{R}_{\mathsf{qs}}}(n) \in \Omega(n^{\frac{2}{1}}) = \Omega(n^2)$, i.e., to prove that the quicksort TRS has at least quadratic complexity.

So while Thm. 17 explains how to find concrete lower bounds for $\mathrm{irc}_{\mathcal{R}}$ (if $sz_{\mathbb{N}}$ can be inverted), the following theorem summarizes our results on asymptotic lower bounds for $\mathrm{irc}_{\mathcal{R}}$. To this end, we combine Thm. 14 on the inference of asymptotic bounds for rt with Lemma 18.

▶ **Theorem 19** (Asymptotic Lower Bounds for $\mathrm{irc}_{\mathcal{R}}$). *Let $s \xrightarrow{\mathrm{i}}{}^{rt(\overline{n})} t$ be a rewrite lemma for $\mathcal{R}$ and let $sz : \mathbb{N}^m \to \mathbb{N}$ be a function such that $sz(b_1, \dots, b_m)$ is the size of $s[n_1/b_1, \dots, n_m/b_m]\downarrow_{\mathcal{G}/\mathcal{A}}$ for all $b_1, \dots, b_m \in \mathbb{N}$, where $sz_{\mathbb{N}}(n) \in \mathcal{O}(n^e)$ for some $e \geq 1$ and $sz_{\mathbb{N}}$ is strictly monotonically increasing. Furthermore, let ih, ib, and is be defined as in Def. 11.*

1. *If $\mathit{ih} = 0$ and ib and is are polynomials of degree d_{ib} and d_{is}, then $\mathrm{irc}_{\mathcal{R}}(n) \in \Omega(n^{\frac{\max\{d_{\mathit{ib}}, d_{\mathit{is}}\}}{e}})$.*
2. *If $\mathit{ih} = 1$ and ib and is are polynomials of degree d_{ib} and d_{is}, then $\mathrm{irc}_{\mathcal{R}}(n) \in \Omega(n^{\frac{\max\{d_{\mathit{ib}}, d_{\mathit{is}}+1\}}{e}})$.*
3. *If $\mathit{ih} > 1$, then $\mathrm{irc}_{\mathcal{R}}(n) \in \Omega(\mathit{ih}^{\sqrt[e]{n}})$.*

6 Preprocessing TRSs by Argument Filtering

A drawback of our approach is that generator functions only represent homogeneous data objects (e.g., lists or trees where all elements have the same value zero). To prove lower complexity bounds also in cases where one needs other forms of rewrite lemmas, we use *argument filtering* [2] to remove certain argument positions of function symbols.

▶ **Example 20** (Argument Filtering). Consider the following TRS $\mathcal{R}_{\mathsf{intlist}}$:

$$\mathsf{intlist}(\mathsf{zero}) \to \mathsf{nil} \qquad \mathsf{intlist}(\mathsf{succ}(x)) \to \mathsf{cons}(x, \mathsf{intlist}(x))$$

We have $\mathsf{intlist}(\mathsf{succ}^n(\mathsf{zero})) \xrightarrow{\mathrm{i}}{}^{n+1} \mathsf{cons}(\mathsf{succ}^{n-1}(\mathsf{zero}), \dots \mathsf{cons}(\mathsf{succ}(\mathsf{zero}), \mathsf{cons}(\mathsf{zero}, \mathsf{nil})))$ for all $n \in \mathbb{N}$. However, the inhomogeneous list on the right-hand side cannot be expressed in a rewrite lemma. Filtering away the first argument of cons yields the TRS $(\mathcal{R}_{\mathsf{intlist}})_{\setminus(\mathsf{cons},1)}$:

$$\mathsf{intlist}(\mathsf{zero}) \to \mathsf{nil} \qquad \mathsf{intlist}(\mathsf{succ}(x)) \to \mathsf{cons}(\mathsf{intlist}(x))$$

For this TRS, our approach can speculate and prove the rewrite lemma $\mathsf{intlist}(\gamma_{\mathbf{Nats}}(n)) \xrightarrow{\mathrm{i}}{}^{n+1} \gamma_{\mathbf{List}}(n)$, i.e., $\mathsf{intlist}(\mathsf{succ}^n(\mathsf{zero})) \xrightarrow{\mathrm{i}}{}^{n+1} \mathsf{cons}^n(\mathsf{nil})$. From this rewrite lemma, one can infer $n - 1 \leq \mathrm{irc}_{(\mathcal{R}_{\mathsf{intlist}})_{\setminus(\mathsf{cons},1)}}(n)$ for all $n \geq 2$ resp. $\mathrm{irc}_{(\mathcal{R}_{\mathsf{intlist}})_{\setminus(\mathsf{cons},1)}}(n) \in \Omega(n)$.

Def. 21 introduces the concept of argument filtering for terms and TRSs formally.

▶ **Definition 21** (Argument Filtering). Let Σ be a signature with $f \in \Sigma$, $\mathrm{ar}_\Sigma(f) = k$, and let $i \in \{1, \ldots, k\}$. Let $\Sigma_{\setminus(f,i)}$ be like Σ, but with $\mathrm{ar}_{\Sigma_{\setminus(f,i)}}(f) = k - 1$. For any term $t \in \mathcal{T}(\Sigma, \mathcal{V})$, we define the term $t_{\setminus(f,i)} \in \mathcal{T}(\Sigma_{\setminus(f,i)}, \mathcal{V})$ resulting from *filtering* away the i-th argument of f:

$$t_{\setminus(f,i)} = \begin{cases} t, & \text{if } t \text{ is a variable} \\ f(\,(t_1)_{\setminus(f,i)}, \ldots, (t_{i-1})_{\setminus(f,i)}, (t_{i+1})_{\setminus(f,i)}, \ldots, (t_k)_{\setminus(f,i)}\,), & \text{if } t = f(t_1, \ldots, t_k) \\ g(\,(t_1)_{\setminus(f,i)}, \ldots, (t_b)_{\setminus(f,i)}\,), & \text{if } t = g(t_1, \ldots, t_b) \text{ for } g \neq f \end{cases}$$

Let $\mathcal{R}$ be a TRS over Σ. Then we define $\mathcal{R}_{\setminus(f,i)} = \{\ell_{\setminus(f,i)} \to r_{\setminus(f,i)} \mid \ell \to r \in \mathcal{R}\}$.

However, a lower bound for the runtime of $\mathcal{R}_{\setminus(f,i)}$ does not imply a lower bound for $\mathcal{R}$ if the argument that is filtered away influences the control flow of the evaluation. Thus, several conditions have to be imposed to ensure that argument filtering is sound for lower bounds:

(a) Argument filtering must not remove function symbols on left-hand sides of rules.
An argument may not be filtered away if it is used for non-trivial pattern matching (i.e., if there is a left-hand side of a rule where the i-th argument of f is not a variable). As an example, consider $\mathcal{R} = \{\mathsf{f}(\mathsf{cons}(\mathsf{true}, xs)) \to \mathsf{f}(\mathsf{cons}(\mathsf{false}, xs))\}$ where $\mathrm{irc}_\mathcal{R}(n) \leq 1$ for all n. But if one filters away the first argument of cons, then one obtains the non-terminating rule $\mathsf{f}(\mathsf{cons}(xs)) \to \mathsf{f}(\mathsf{cons}(xs))$, i.e., $\mathrm{irc}_{\mathcal{R}_{\setminus(\mathsf{cons},1)}}(n) = \omega$ for $n \geq 3$.

(b) The TRS must be left-linear.
To illustrate this, consider $\mathcal{R} = \{\mathsf{f}(xs, xs) \to \mathsf{f}(\mathsf{cons}(\mathsf{true}, xs), \mathsf{cons}(\mathsf{false}, xs))\}$, where again $\mathrm{irc}_\mathcal{R}(n) \leq 1$. But filtering away the first argument of cons yields the non-terminating rule $\mathsf{f}(xs, xs) \to \mathsf{f}(\mathsf{cons}(xs), \mathsf{cons}(xs))$, i.e., $\mathrm{irc}_{\mathcal{R}_{\setminus(\mathsf{cons},1)}}(n) = \omega$ for $n \geq 3$.

(c) Argument filtering must not result in free variables on right-hand sides of rules.
The reason is that otherwise, argument filtering might again turn terminating TRSs into non-terminating ones. For instance, consider $\mathcal{R} = \{\mathsf{f}(\mathsf{cons}(x, xs)) \to \mathsf{f}(xs)\}$ where $\mathrm{irc}_\mathcal{R}(n) = \lfloor \frac{n}{2} \rfloor - 1$. But if one filters away the second argument of cons, then one gets the rule $\mathsf{f}(\mathsf{cons}(x)) \to \mathsf{f}(xs)$ whose runtime is unbounded, i.e., $\mathrm{irc}_{\mathcal{R}_{\setminus(\mathsf{cons},2)}}(n) = \omega$ for $n \geq 3$.

Thm. 22 states that (a) - (c) are indeed sufficient for the soundness of argument filtering. To infer a lower bound for $\mathrm{irc}_\mathcal{R}$ from a bound for $\mathrm{irc}_{\mathcal{R}_{\setminus(f,i)}}$, we have to take into account that filtering changes the size of terms. As an example, consider $\mathcal{R} = \{\mathsf{f}(x) \to \mathsf{a}\}$. Here, we have $\mathrm{irc}_{\mathcal{R}_{\setminus(\mathsf{f},1)}}(1) = 1$ due to the rewrite sequence $\mathsf{f} \xrightarrow{\mathsf{i}}_{\mathcal{R}_{\setminus(\mathsf{f},1)}} \mathsf{a}$. The corresponding rewrite sequence in the original TRS $\mathcal{R}$ is $\mathsf{f}(x) \xrightarrow{\mathsf{i}}_\mathcal{R} \mathsf{a}$. Thus, $\mathrm{irc}_\mathcal{R}(2) = 1$, but all terms of size 1 are normal forms of $\mathcal{R}$, i.e., $\mathrm{irc}_\mathcal{R}(1) = 0$. So $\mathrm{irc}_{\mathcal{R}_{\setminus(f,i)}}(n) \leq \mathrm{irc}_\mathcal{R}(n)$ does not hold in general. Nevertheless, for any rewrite sequence of $\mathcal{R}_{\setminus(f,i)}$ starting with a term t, there is a corresponding rewrite sequence of $\mathcal{R}$ starting with a term[7] s where $|s| \leq 2 \cdot |t|$. Thus, if we have derived a lower bound $p(n)$ for $\mathrm{irc}_{\mathcal{R}_{\setminus(f,i)}}(n)$, we can use $p(\frac{n}{2})$ as a lower bound for $\mathrm{irc}_\mathcal{R}(n)$. Hence, in Ex. 20, we obtain $\frac{n}{2} - 1 \leq \mathrm{irc}_{\mathcal{R}_{\mathsf{intlist}}}(n)$ for all $n \geq 4$ resp. $\mathrm{irc}_{\mathcal{R}_{\mathsf{intlist}}}(n) \in \Omega(n)$.

▶ **Theorem 22** (Soundness of Argument Filtering). *Let $f \in \Sigma$, $\mathrm{ar}_\Sigma(f) = k$, and $i \in \{1, \ldots, k\}$. Moreover, let $\mathcal{R}$ be a TRS over Σ where the following conditions hold for all rules $\ell \to r \in \mathcal{R}$:*

(a) *If $f(t_1, \ldots, t_k)$ is a subterm of ℓ, then $t_i \in \mathcal{V}$.*

[7] The term s can be obtained from t by adding a variable as the i-th argument for any f occurring in t.

(b) *For any* $x \in \mathcal{V}$*, there is at most one position* $\pi \in \text{pos}(\ell)$ *such that* $\ell|_\pi = x$*.*
(c) $\mathcal{V}(r_{\setminus(f,i)}) \subseteq \mathcal{V}(\ell_{\setminus(f,i)})$*.*
Then for all $n \in \mathbb{N}$*, we have* $\text{irc}_{\mathcal{R}_{\setminus(f,i)}}(\frac{n}{2}) \leq \text{irc}_{\mathcal{R}}(n)$*.*

In our implementation, as a heuristic we always perform argument filtering if it is permitted by Thm. 22, except for cases where filtering removes defined function symbols on right-hand sides of rules. As an example, consider $\mathcal{R} = \{\mathsf{a} \to \mathsf{f}(\mathsf{a},\mathsf{b})\}$ where $\text{irc}_{\mathcal{R}}(n) = \omega$ for $n \geq 1$. If one filters away f's first argument, then one obtains $\mathsf{a} \to \mathsf{f}(\mathsf{b})$ and thus, $\text{irc}_{\mathcal{R}_{\setminus(\mathsf{f},1)}}(n) = 1$ for $n \geq 1$. So here, argument filtering is sound, but it results in significantly worse lower bounds.

7 Indefinite Rewrite Lemmas

Our technique often fails if the analyzed TRS is not completely defined, i.e., if there are normal forms containing defined symbols. As an example, the runtime complexity of $\mathcal{R}_{in} = \{\mathsf{f}(\mathsf{succ}(x)) \to \mathsf{succ}(\mathsf{f}(x))\}$ is linear due to the rewrite sequences $\mathsf{f}(\mathsf{succ}^n(\mathsf{zero})) \xrightarrow{\mathrm{i}}{}^n \mathsf{succ}^n(\mathsf{f}(\mathsf{zero}))$. However, the term $\mathsf{succ}^n(\mathsf{f}(\mathsf{zero}))$ on the right-hand side contains f and thus, it cannot be represented in a rewrite lemma. Therefore, we now also allow *indefinite* conjectures and rewrite lemmas with unknown right-hand sides. Then for our example, we could speculate the indefinite conjecture $\mathsf{f}(\gamma_{\mathbb{N}}(n)) \xrightarrow{\mathrm{i}}{}^* \star$, which gives rise to the indefinite rewrite lemma $\mathsf{f}(\gamma_{\mathbb{N}}(n)) \xrightarrow{\mathrm{i}}{}^n \star$, where $\star$ represents an arbitrary unknown term. To distinguish indefinite conjectures and rewrite lemmas from ordinary ones, we call the latter *definite.*

Recall that when speculating conjectures in Sect. 2, we built a narrowing tree for a term $s = f(\ldots)$ and obtained a sample point (t, σ, d) whenever we reached a normal form t. When speculating indefinite conjectures, we do not narrow in order to reach normal forms, but we create a sample point (σ, d) after each application of a recursive f-rule. Here, σ is again the substitution and d is the recursion depth of the path. Note that while previously proven lemmas $\mathcal{L}$ may be used during narrowing, we do not use previous indefinite rewrite lemmas, since they do not yield any information on the terms resulting from rewriting.

▶ **Example 23** (Speculating Indefinite Conjectures)**.** For $\mathcal{R}_{in}$, we narrow the term $s = \mathsf{f}(\gamma_{\mathbf{Nats}}(x))$. We get $\mathsf{f}(\gamma_{\mathbf{Nats}}(x)) \leadsto \mathsf{succ}(\mathsf{f}(\gamma_{\mathbf{Nats}}(x')))$ with the substitution $\sigma_1 = [x/x'+1]$. Since we applied a recursive f-rule once, we construct the sample point $(\sigma_1, 1)$. We continue narrowing and get $\mathsf{succ}(\mathsf{f}(\gamma_{\mathbf{Nats}}(x'))) \leadsto \mathsf{succ}(\mathsf{succ}(\mathsf{f}(\gamma_{\mathbf{Nats}}(x''))))$ with the substitution $\sigma_2 = [x'/x''+1]$ and recursion depth 2. Since $\sigma_2 \circ \sigma_1$ corresponds to $[x/x''+2]$, this yields the sample point $([x/x''+2], 2)$. Another narrowing step results in the sample point $([x/x'''+3], 3)$.

These sample points represent the sample conjectures $\mathsf{f}(\gamma_{\mathbf{Nats}}(x'+1)) \xrightarrow{\mathrm{i}}{}^* \star$, $\mathsf{f}(\gamma_{\mathbf{Nats}}(x''+2)) \xrightarrow{\mathrm{i}}{}^* \star$, $\mathsf{f}(\gamma_{\mathbf{Nats}}(x'''+3)) \xrightarrow{\mathrm{i}}{}^* \star$ that are identical up to the occurring numbers and variable names. Thus, they are suitable for generalization. As in Sect. 2, we replace the numbers in the sample conjectures by a polynomial *pol* in one variable n that stands for the recursion depth. This leads to $\mathsf{f}(\gamma_{\mathbf{Nats}}(x + \mathit{pol})) \xrightarrow{\mathrm{i}}{}^* \star$ and the constraints $\mathit{pol}(1) = 1$, $\mathit{pol}(2) = 2$, $\mathit{pol}(3) = 3$. A solution is $\mathit{pol} = n$ and thus, we speculate the indefinite conjecture $\mathsf{f}(\gamma_{\mathbf{Nats}}(x + n)) \xrightarrow{\mathrm{i}}{}^* \star$.

Every indefinite conjecture gives rise to an indefinite rewrite lemma.

▶ **Definition 24** (Indefinite Rewrite Lemmas)**.** Let $\mathcal{R}$, s, rt be as in Def. 8. Then $s \xrightarrow{\mathrm{i}}{}^{\mathit{rt}(\overline{n})} \star$ is an *indefinite rewrite lemma* for $\mathcal{R}$ iff for all $\sigma : \mathcal{V}(s) \to \mathbb{N}$ there is a term t such that $s\sigma{\downarrow_{\mathcal{G}/\mathcal{A}}} \xrightarrow{\mathrm{i}}{}_{\mathcal{R}}^{\mathit{rt}(\overline{n}\sigma)} t$, i.e., $s\sigma{\downarrow_{\mathcal{G}/\mathcal{A}}}$ starts an innermost $\mathcal{R}$-reduction of at least $\mathit{rt}(n_1\sigma, \ldots, n_m\sigma)$ steps.

In principle, proving indefinite conjectures $s \xrightarrow{\mathrm{i}}{}^* \star$ is not necessary, since $s \xrightarrow{\mathrm{i}}{}^0 \star$ is always a valid indefinite rewrite lemma. However, to derive useful lower complexity bounds, we need

rewrite lemmas $s \xrightarrow{\mathrm{i}}{}^{rt(\overline{n})} \star$ for non-trivial functions rt. Thm. 25 shows that the approaches for proving lemmas from Sect. 3 and for deriving bounds from these proofs in Sect. 4 can also be used for indefinite rewrite lemmas. The only adaption needed is that the relation $\xrightarrow{\mathrm{i}}_{\mathcal{R}}$ may not reduce redexes that contain the symbol $\star$. This restriction is needed due to the innermost evaluation strategy, because $\star$ represents arbitrary terms that are not necessarily in normal form. In this way, all previously proven (definite or indefinite) rewrite lemmas $\mathcal{L}$ can be used in the proof of new (definite or indefinite) rewrite lemmas.

▶ **Theorem 25** (Bounds for Indefinite Rewrite Lemmas). *Let $\xrightarrow{\mathrm{i}}_{\mathcal{R}}$ and $\xrightarrow{\mathrm{i}}_{(\mathcal{R},\mathrm{IH})}$ be restricted such that redexes may not contain the symbol $\star$ and let ih, ib, and is be defined as in Def. 11. Here, for an indefinite rewrite lemma $s \xrightarrow{\mathrm{i}}{}^{rt(\overline{n})} \star$ with $n \in \mathcal{V}(s)$, we say that any rewrite sequence $s[n/0] = u_1 \xrightarrow{\mathrm{i}}_{\mathcal{R}} u_2 \xrightarrow{\mathrm{i}}_{\mathcal{R}} \dots \xrightarrow{\mathrm{i}}_{\mathcal{R}} u_{b+1}$ "proves" the induction base and any rewrite sequence $s[n/n+1] = v_1 \xrightarrow{\mathrm{i}}_{(\mathcal{R},\mathrm{IH})} v_2 \xrightarrow{\mathrm{i}}_{(\mathcal{R},\mathrm{IH})} \dots \xrightarrow{\mathrm{i}}_{(\mathcal{R},\mathrm{IH})} v_{k+1}$ "proves" the induction step, where* IH *is the rule $s \to \star$. Then Thm. 12 and Thm. 14 on explicit and asymptotic runtimes hold for any definite or indefinite rewrite lemma.*

▶ **Example 26** (Complexity of Indefinite Rewrite Lemmas). To continue with Ex. 23, we now infer the runtime for the rewrite lemma $\mathsf{f}(\gamma_{\mathbf{Nats}}(x+n)) \xrightarrow{\mathrm{i}}{}^{rt(x,n)} \star$. Since $\mathsf{f}(\gamma_{\mathbf{Nats}}(x+0))$ is already in normal form w.r.t. $\xrightarrow{\mathrm{i}}_{\mathcal{R}}$, the length of the rewrite sequence in the induction base is $ib(x) = 0$. In the induction step, we obtain $\mathsf{f}(\gamma_{\mathbf{Nats}}(x+n+1)) \xrightarrow{\mathrm{i}}_{\mathcal{R}} \mathsf{succ}(\mathsf{f}(\gamma_{\mathbf{Nats}}(x+n))) \mapsto_{\mathrm{IH}} \mathsf{succ}(\star)$. Thus, the induction hypothesis is applied $ih = 1$ time and the number of remaining rewrite steps is $is(x,n) = 1$. According to Thm. 12, we have $rt(x,n) = ih^n \cdot ib(x) + \sum_{i=0}^{n-1} ih^{n-1-i} \cdot is(x,i) = 1^n \cdot 0 + \sum_{i=0}^{n-1} 1^{n-1-i} \cdot 1 = n$. Similarly, since $ih = 1$ and both $ib(x) = 0$ and $is(x,n) = 1$ are polynomials of degree 0, Thm. 14 implies $rt_{\mathbb{N}}(n) \in \Omega(n^{\max\{0,0+1\}}) = \Omega(n)$.

8 Experiments and Conclusion

We presented the first approach to infer *lower* bounds for the innermost runtime complexity of TRSs automatically. It is based on speculating rewrite lemmas by narrowing, proving them by induction, and determining the length of the corresponding rewrite sequences from this proof. By taking the size of the start term of the rewrite lemma into account, this yields a lower bound for $\mathrm{irc}_{\mathcal{R}}$. Our approach can be improved by argument filtering and by allowing rewrite lemmas with unknown right-hand sides. In this way the rewrite lemmas do not have to represent rewrite sequences of the original TRS precisely. Future work will be concerned with considering more general forms of induction proofs and rewrite lemmas.

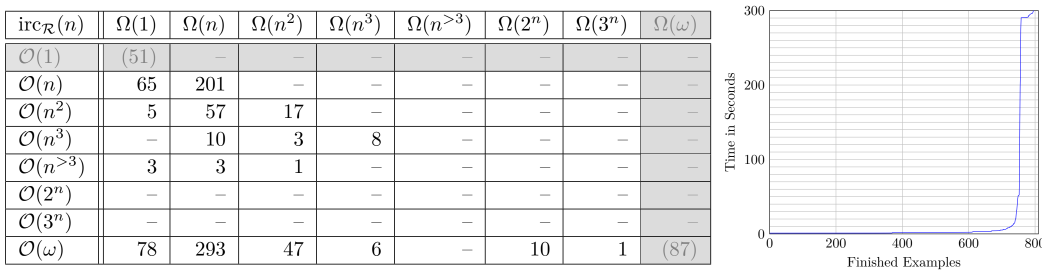
We implemented our approach in AProVE [11], which uses Z3 [6] to solve arithmetic constraints. While our technique can also infer concrete bounds, currently AProVE only computes asymptotic bounds and provides the lemma that leads to the reported runtime as a witness.

There exist a few results on lower bounds for *derivational complexity* (e.g., [16, 20]) and in the *Termination Competitions*[8] 2009 - 2011, Matchbox [19] proved lower bounds for full derivational complexity where arbitrary rewrite sequences are considered.[9] However, there are no other tools that infer lower bounds for innermost runtime complexity. Hence, we compared our results with the asymptotic *upper* bounds computed by AProVE and TcT [4], the winners

[8] See `http://termination-portal.org/wiki/Termination_Competition`.

[9] For derivational complexity, every non-empty TRS has a trivial linear lower bound. In contrast, proving linear lower bounds for runtime complexity is not trivial. Thus, lower bounds for derivational complexity are in general unsound for runtime complexity. Therefore, an experimental comparison with tools for derivational complexity is not meaningful.

in the category *"Runtime Complexity – Innermost Rewriting"* at the *Termination Competition* 2014. We tested with 808 TRSs from this category of the *Termination Problem Data Base* (TPDB 9.0.2) which was used for the Termination Competition 2014. We omitted 118 non-standard TRSs with extra variables on right-hand sides or relative rules. We also disregarded 51 TRSs where AProVE or TcT proved $\mathrm{irc}_{\mathcal{R}}(n) \in \mathcal{O}(1)$ and 87 examples with $\mathrm{irc}_{\mathcal{R}}(n) \in \Omega(\omega)$ (gray cells in the table below). To identify the latter, we adapted existing innermost non-termination techniques to only consider sequences starting with basic terms. Each tool had a time limit of 300 s for each example. The following table compares the lower bound found by our implementation with the minimum upper bound computed by AProVE or TcT.



$\mathrm{irc}_{\mathcal{R}}(n)$	$\Omega(1)$	$\Omega(n)$	$\Omega(n^2)$	$\Omega(n^3)$	$\Omega(n^{>3})$	$\Omega(2^n)$	$\Omega(3^n)$	$\Omega(\omega)$
$\mathcal{O}(1)$	(51)	–	–	–	–	–	–	–
$\mathcal{O}(n)$	65	201	–	–	–	–	–	–
$\mathcal{O}(n^2)$	5	57	17	–	–	–	–	–
$\mathcal{O}(n^3)$	–	10	3	8	–	–	–	–
$\mathcal{O}(n^{>3})$	3	3	1	–	–	–	–	–
$\mathcal{O}(2^n)$	–	–	–	–	–	–	–	–
$\mathcal{O}(3^n)$	–	–	–	–	–	–	–	–
$\mathcal{O}(\omega)$	78	293	47	6	–	10	1	(87)

The average runtime of our implementation was 22.5 s, but according to the chart above, it was usually much faster. In 694 cases, the analysis finished in 5 seconds. AProVE inferred lower bounds for 657 (81%) of the 808 TRSs. Upper bounds were only obtained for 373 (46%) TRSs, although such bounds exist for at least all 670 TRSs where AProVE shows innermost termination. So although this is the first technique for lower $\mathrm{irc}_{\mathcal{R}}$-bounds, its applicability exceeds the applicability of the techniques for upper bounds which were developed for years. Tight bounds (where the lower and upper bounds are equal) were proven for the 226 TRSs on the diagonal of the table. There are just 74 TRSs where different non-trivial lower and upper bounds were inferred and for 60 of these cases, they just differ by the factor n.

Our approach is particularly powerful for TRSs that implement realistic algorithms, e.g., it shows $\mathrm{irc}_{\mathcal{R}}(n) \in \Omega(n^2)$ for many implementations of classical sorting algorithms like *quicksort*, *maxsort*, *minsort*, and *selection-sort* from the TPDB where neither AProVE nor TcT prove $\mathrm{irc}_{\mathcal{R}}(n) \in \mathcal{O}(n^2)$. Detailed experimental results and a web interface for our implementation are available at [1].

Acknowledgments. We thank Fabian Emmes for important initial ideas for this paper.

References

1 AProVE: `http://aprove.informatik.rwth-aachen.de/eval/lowerbounds/`.

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

3 M. Avanzini and G. Moser. A combination framework for complexity. In *Proc. RTA '13*, LIPIcs 21, pages 55–70, 2013.

4 M. Avanzini and G. Moser. Tyrolean Complexity Tool: Features and usage. In *Proc. RTA '13*, LIPIcs 21, pages 71–80, 2013.

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

6 L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. TACAS '08*, LNCS 4963, pages 337–340, 2008.

7 F. Emmes, T. Enger, and J. Giesl. Proving non-looping non-termination automatically. In *Proc. IJCAR '12*, LNAI 7364, pages 225–240, 2012.
8 F. Frohn, J. Giesl, F. Emmes, T. Ströder, C. Aschermann, and J. Hensel. Inferring lower bounds for runtime complexity. Technical Report AIB 2015-15, RWTH Aachen, 2015. Available from [1] and from `http://aib.informatik.rwth-aachen.de`.
9 C. Fuhs, J. Giesl, M. Parting, P. Schneider-Kamp, S. Swiderski. Proving termination by dependency pairs and inductive theorem proving. *Journal of Automated Reasoning*, 47(2):133–160, 2011.
10 J. Giesl, M. Raffelsieper, P. Schneider-Kamp, S. Swiderski, and R. Thiemann. Automated termination proofs for Haskell by term rewriting. *ACM Transactions on Programming Languages and Systems*, 33(2), 2011.
11 J. Giesl, M. Brockschmidt, F. Emmes, F. Frohn, C. Fuhs, C. Otto, M. Plücker, P. Schneider-Kamp, T. Ströder, S. Swiderski, and R. Thiemann. Proving termination of programs automatically with AProVE. In *Proc. IJCAR '14*, LNAI 8562, pages 184–191, 2014.
12 N. Hirokawa and G. Moser. Automated complexity analysis based on the dependency pair method. In *Proc. IJCAR '08*, LNAI 5195, pages 364–379, 2008.
13 D. Hofbauer and C. Lautemann. Termination proofs and the length of derivations. In *Proc. RTA '89*, LNCS 355, pages 167–177, 1989.
14 M. Hofmann and G. Moser. Amortised resource analysis and typed polynomial interpretations. In *Proc. RTA-TLCA '14*, LNCS 8560, pages 272–286, 2014.
15 D. Knuth. Johann Faulhaber and sums of powers. *Mathematics of Computation*, 61(203):277–294, 1993.
16 G. Moser and A. Schnabl. The derivational complexity induced by the dependency pair method. *Logical Methods in Computer Science*, 7(3), 2011.
17 L. Noschinski, F. Emmes, and J. Giesl. Analyzing innermost runtime complexity of term rewriting by dependency pairs. *Journal of Automated Reasoning*, 51(1):27–56, 2013.
18 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.
19 J. Waldmann. Matchbox: A tool for match-bounded string rewriting. In *Proc. RTA '04*, LNCS 3091, pages 85–94, 2004.
20 J. Waldmann. Automatic termination. In *Proc. RTA '09*, LNCS 5595, pages 1–16, 2009.
21 H. Zankl and M. Korp. Modular complexity analysis for term rewriting. *Logical Methods in Computer Science*, 10(1), 2014.
22 H. Zantema. Termination of term rewriting: Interpretation and type elimination. *Journal of Symbolic Computation*, 17(1):23–50, 1994.

A Simple and Efficient Step Towards Type-Correct XSLT Transformations

Markus Lepper[1] and Baltasar Trancón y Widemann[2]

1 <semantics/> GmbH, Berlin, DE
2 Ilmenau University of Technology, Ilmenau, DE

Abstract

XSLT 1.0 is a standardized functional programming language and widely used for defining transformations on XML models and documents, in many areas of industry and publishing. The problem of *XSLT type checking* is to verify that a given transformation, when applied to an input which conforms to a given structure definition, e.g. an XML DTD, will always produce an output which adheres to a second structure definition. This problem is known to be undecidable for the full range of XSLT and document structure definition languages. Either one or both of them must be significantly restricted, or only approximations can be calculated. The algorithm presented here takes a different approach towards type correct XSLT transformations. It does not consider the type of the input document at all. Instead it parses the fragments of the result document contained *verbatim* in the transformation code and verifies that these can potentially appear in the result language, as defined by a given DTD. This is a kind of *abstract interpretation*, which can be executed on the fly and in linear time when parsing the XSLT program. Generated error messages are located accurately to a child subsequence of a single result element node. Apparently the method eliminates a considerable share of XSLT programming errors, on the same order of magnitude as a full fledged global control-flow analysis.

1998 ACM Subject Classification D.1.1 Functional Programming, D.3.2 Functional Language I.7.2 Scripting languages

Keywords and phrases XSLT, type checking, abstract interpretation

Digital Object Identifier 10.4230/LIPIcs.RTA.2015.350

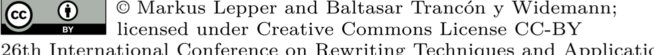
1 Introduction

1.1 XSLT Transformations and Document Types

XSLT, in its different versions, is a standard transformation language for processing XML documents. There are different implementations of XSLT processors, employing various technologies. The contribution in this article is about XSLT 1.0. All versions of the language are Turing complete and fully functional programming languages. “Functional” in the most obvious sense means that there are no variables which can change state, but instead functions which can be applied to constant parameters and thereby yield a certain result. These function calls can be recursive. Functions are called “templates” in the context of the language specification.

An XSLT program is in most cases used to convert an XML document, serving as the *input*, into a second XML document serving as its *result*.[1] The templates which are applied to

[1] The conversion into unstructured, plain text, or into text structured by other means than XML is also possible but not covered by this article.

© Markus Lepper and Baltasar Trancón y Widemann;
licensed under Creative Commons License CC-BY
26th International Conference on Rewriting Techniques and Applications (RTA’15).
Editor: Maribel Fernández; pp. 350–364

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

certain elements of the input document are selected by a simple mechanism of pattern matching — either in a fully automated way by pre-defined standard rules, or semi-automatically after some explicit pre-selection by the author of the program. For writing meaningful programs, it is therefore a prerequisite that all input documents to a given transformation have certain invariant structural properties. This is *not* a technical necessity imposed by the language: the semantics of XSLT are rather "robust" and as long as no errors are raised *explicitly* by the programmer, arbitrary input will be transformed into some output. But in practice, the document type of the input document will be defined in some precise formalism anyhow, e.g. as RELAX NG grammar, W3C Schema, or W3C DTD.

In many cases it is required that the output of an XSLT transformation adheres also to such a precise document type. Of course this can always be checked a posteriori by *validating* the result of every transformation explicitly against this result document type. All cases where the result document violates the intended result document type are caused by programming errors in the transformation.

Obviously, it is highly desirable to find these programming errors earlier, when constructing the XSLT program. This is not only relevant for performance issues, since validating always implies total parsing, but also for increased reliability of services based on transformations: XML and XSLT processing are more and more applied to critical data, like business objects, physical real-time data, medical files, etc.

The general problem is that of *type checking*: an XSLT program is type correct, if and only if every input which adheres to a given input type will produce an output which adheres to a given output type. This problem has been proven to be intractable in the general case [9]. Furthermore, restrictions of the involved two languages (XSLT and document type definition) have been defined for which it is solvable, and the complexity of these problems has been thoroughly analyzed in the last two decades. For surveys see e.g. [11] and [12].

1.2 Fragmented Validation

In contrast to these theoretically advanced studies, this paper presents a totally different approach, a very simple and pragmatic idea which turns out to be rather effective for concrete programming, called *fragmented validation* (FV). It does not consider type of the input document at all, but *only the consecutive sequences of nodes from the result language* which occur in the contents of an element somewhere in the transformation program's XML representation, and which serve as constant data for the transformation process. The transformation result will be produced by combining these fragments;[2] therefore they must match the result document type in at least *some* context. In other words, there must exist at least one rule in the result document type definition which is able to produce contents that contains the fragment.

For example, when producing XHTML output, any XSLT code like

```
<xsl:template ...>
  <xsl:.../><xsl:.../><tr>...</tr><xsl:.../><td>...</td>
</xsl:template>
```

will never produce a valid result: there is no "content model" (i.e. regular expression, see section 2.1) in the type definition of XHTML which allows the elements `<tr>` (table row)

[2] There are other ways of producing output, e.g. by copying nodes of the input document, or explicit element construction, but in many cases their role is marginal.

and `<td>` (table data cell) to appear on the same level of nesting. This violation can be found independently from the contents of the other embedded XSLT commands in this example. These can produce anything from the empty sequence to arbitrary sequences of *complete* elements. So their outcome cannot change the levels of nesting, and cannot heal the violation.[3]

That all appearing constant result fragments match the result document type is neither a necessary nor a sufficient condition for the correctness of the transformation in a strict mathematical sense: the combination of correct fragments can still violate the result type, and incorrect fragments appearing in the source may belong to "dead code" and may never be used. Debugging XSLT transformations is a rather tedious task, in spite of the intended readability of the "graphical" text format, see next section. We found that not only at about thirty percent of programming errors are detected a priori by fragmented validation, but also that the remaining errors of illegal combinations are much easier diagnosed, because the strategy for debugging applied by the programmers change fundamentally and becomes much more focused, as soon it is guaranteed that the constant fragments themselves cannot be the source of typing errors in some generated output.

The algorithm presented here performs simple and comprehensible validation of *all* result fragments contained in an XSLT program. This is done by a kind of *abstract interpretation*, which operates on sets of states, and pre-figures a possible later parsing process of the result document. It does so in linear time, when the XSLT program is parsed and its data model is constructed, "on the fly", by tracking the corresponding SAX events.

1.3 XSLT Program as a Two-Coloured Tree

The front-end representation chosen by the designers of XSLT integrates the XSLT language constructs and the constants of the result language seamlessly: both are represented as well-formed XML structures, which can be interspered in a rather free fashion. The intention is to make the formatted program easily readable from two different viewpoints, in the style of a visual ambiguity: the XSLT language constructs can contain fragments of the result language, as they are to appear in the output, as their operands, and these fragments in turn can contain XSLT elements which will be replaced by their evaluation result when executing the transformation.

Therefore an XSLT program comes as a *two-coloured* XML tree, in which elements from XSLT and elements from the result language are mixed. The rules for either embedding are defined atop a categorization of the XSLT elements:

- The top element is always a `stylesheet` element from XSLT.[4]
- This element contains elements from the **T** or "top" category of XSLT elements.
- Some of those and of their children belong to the **R** or "result element-containing" category, which can contain result-language elements directly in their contents.
- Vice versa, there is the **I** or "instruction" category[5], which can be inserted anywhere

---

[3] This principle is the core of the type safety of XSLT compared to string manipulation languages like PHP, which produce "tag soup", and can indeed be a severe obstacle for programmers used to these, as the long lasting discussion in `https://bugzilla.mozilla.org/show_bug.cgi?id=98168` about "`disable-output-escaping`" shows.

[4] This element can also be called `transform`, and all definitions must be doubled accordingly. The following text ignores this synonym for better readability. The nomenclature in the XSLT standard [14] is a little bit peculiar anyhow, e.g. functions are called "templates", etc.

[5] This strange wording is again taken from the standard; the one character abbreviations of the categories are ours.

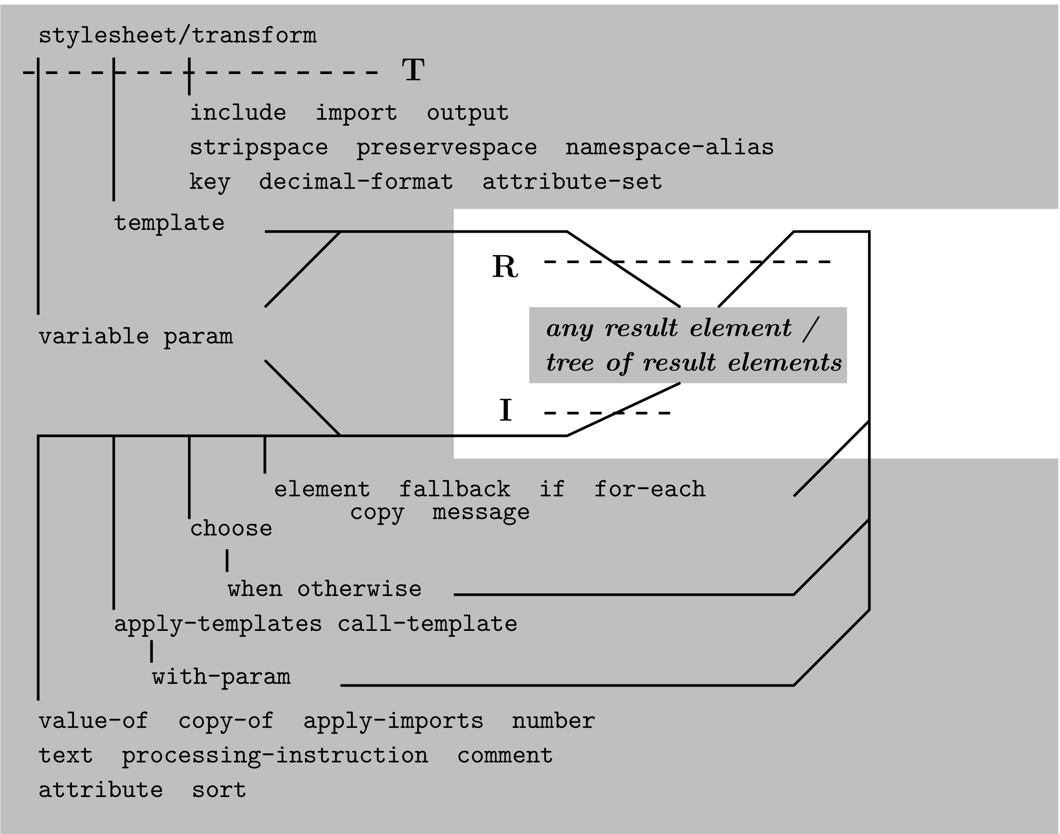


Figure 1 XSLT and result language nesting, with categories.

in such contents, or arbitrarily deeply nested in a result language structure. Later, when the whole program will be executed, these XSLT elements will evaluate to some result language elements ("produce" them), which will be inserted into the surrounding constant result structure.

Figure 1 shows the resulting "sandwich" structure and the possible transitions between both element sets: each arrow indicates a possible parent-child relationship in an XSLT tree. Figure 2 shows a typical XSLT source, a bi-coloured tree with pure and mixed sub-trees. The positions where result language elements may occur in an XSLT structure are limited, namely the complete contents of the elements of category **R**; there arbitrary sequences of result language elements may appear, mixed again with XSLT elements from category **I**.

By defining an artificial XSLT element e_{R}, which wraps these sequences as its contents, and which appears (like a normal XSLT element) in the content models of the **R** elements as a placeholder for this of embedding, we can integrate those mixed sequences smoothly into the parsing process of XSLT.

The other way round, starting with an element of the result language, this easy procedure is not applicable, because the **I** category elements from XSLT can appear *ubiquitously* in the result language structure. Therefore the theoretically possible approach of factoring out all possible interleavings and treat them by constructing the joint deterministic automaton is not feasible in practice, due to combinatorial explosion. Instead, we construct two independent conventional deterministic automata for both language definitions in the form of a *transition relation* and some auxiliary mappings. Simple algebraic operations on this

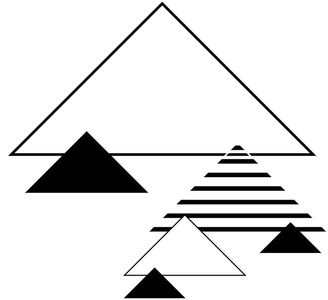


Figure 2 XSLT (white), result (black) and unparsed, two-coloured subtrees of an XSLT program.

Table 1 Content models and calculation of transition relation.

$$C ::= (E, S) \mid (C\,\texttt{,}\,\ldots\,\texttt{,}\,C) \mid (C\,\texttt{|}\,\ldots\,\texttt{|}\,C) \mid C\texttt{?} \mid C\texttt{*} \mid C\texttt{+}$$

$$\mathsf{startState} : (E \setminus \{e_{\mathrm{C}}, e_{\mathrm{R}}\}) \to S$$

$$\mathsf{refTo} : S \nrightarrow E$$

$$\mathsf{collect} : C \times \mathbb{P}S \times (S \leftrightarrow S) \quad \to \quad \mathbb{P}S \times (S \leftrightarrow S)$$

$$\frac{\mathsf{collect}(c_1, T, U) = (V, W) \qquad \mathsf{collect}(c_2, T, U) = (X, Y)}{\mathsf{collect}\big((c_1\,\texttt{|}\,c_2), T, U\big) = (V \cup X, W \cup Y)}$$

$$\frac{\mathsf{collect}(c_1, T, U) = (V, W) \qquad \mathsf{collect}(c_2, V, W) = (X, Y)}{\mathsf{collect}\big((c_1\,\texttt{,}\,c_2), T, U\big) = (X, W \cup Y)}$$

$$\frac{\mathsf{collect}(c, T, U) = (V, W)}{\mathsf{collect}(c\texttt{?}, T, U) = (T \cup V, W)}$$

$$\mathsf{collect}(c\texttt{+}, T, U) = \mathsf{collect}\big((c\,\texttt{,}\,c\texttt{?}), T, U\big)$$

$$\mathsf{collect}(c\texttt{*}, T, U) = \mathsf{collect}\big((c\texttt{+})\texttt{?}, T, U\big)$$

$$\mathsf{collect}\big((e, s), T, U\big) = \big(\{s\}, U \cup (T \times \{s\})\big)$$

$$S_{\mathrm{acc}} = \bigcup_{e \in E} \pi_1\big(\mathsf{collect}(c_e, \{\mathsf{startState}(e)\}, \{\})\big)$$

$$\mathsf{goesTo} = \bigcup_{e \in E} \pi_2\big(\mathsf{collect}(c_e, \{\mathsf{startState}(e)\}, \{\})\big)$$

relation allow us to switch between deterministic and non-deterministic modes of operation, whenever the special parsing situations arise.

By these means the algorithm constitutes a transition relation, which can be evaluated on the fly when parsing an XSLT source. As soon as for the given input no further transition is possible, either a validation of genuine XSLT syntax is found, or a fragment of the result language which is not part of any valid result document.

2 Parsing XSLT Programs With Fragmented Validation

2.1 Validation of the XSLT Program

The algorithm presented here is based on DTDs as the means for defining the structure of the involved documents. This formalism is specified in the core XML standard [3]. A DTD

defines (1) a set of string values as *element tags*, and for every such element (2) a list of *attributes* with names, types and default values[6], and for every element (3) a *content model*, which is an *extended regular expression* over the set of element names. As ususal, a regular expression defines an accepted language. In this case this is a set of sequences of element names, which defines the legal contents of the element under definition.

While the XSLT language is not specified in terms of a DTD, this can easily be constructed, and is called D_{X} in the following.[7] The corresponding sets of element tags is E_{X}. Beside the element tags defined in the standard, E_{X} contains an element e_{D} which represents the top-most level of the document tree (in XML this called "document" and is an additional level one above the top-most element), and an additional element e_{R} which wraps all embedded sequences of result elements, as described above.

The structure of the result language must be given by a second DTD, called D_{R} and a set of element names E_{R}. References to character data in the content models (`#PCDATA`) are realized as a reference to some additional, reserved element e_{C}. This is the only element contained in both sets E_{X} and E_{R}. Their union is E.

Each DTD contains a mapping from its subset of E to content models C, see the definitions in Table 1. Let $c_e \in C$ be the content model for a given $e \in E$. A content model is an extended regular expression: the atoms are references to elements; the unary constructors are option, star and plus; the n-ary constructors are sequence and alternative, which can be realized by associative binary operators.

In our approach the references to elements in a content model are realized by a pair of the element's name e and a state number $s \in S$. These states are *unique* over all content models of both DTDs and identify the symbolic state of the accepting automaton *after* a complete element with the name e has been consumed in that position. The content model of the document level is additionally defined to $c_{e_{\mathrm{D}}} = (\mathtt{stylesheet}, s_{\mathrm{D}})$.

That a state appears in a pair (e, s) in a content model is reflected by (s, e) appearing in the mapping $\mathsf{refTo} : S \nrightarrow E$. Additionally, there is an initial state for every content model, given by $\mathsf{startState} : E \nrightarrow S$, having consumed nothing and thus not in the domain of refTo.

All automata of all content models of both DTDs are realized by the one global relation $\mathsf{goesTo} : S \leftrightarrow S$, together with the set of accepting states $S_{\mathrm{acc}} \subset S$. These are constructed by the function $\mathsf{collect}(c, T, U)$, which performs an abstract parsing process of the content model c, with T being the set of final states of all preceding parsing steps, i.e. the "incoming states", and U being the accumulated transition relation so far.

For each element, this function is started with its whole content model, its start state, and an emtpy "goesTo" relation, see the last two lines in Table 1. The rules for the different kinds of content models are written as logical inference rules and can be executed deterministically by function evaluation. The rule for alternatives $c_1 \,\mathtt{|}\, c_2$ simply unifies both sets; the rule for sequences $c_1 \,\mathtt{|}\, c_2$ starts parsing of c_2 with the results of c_1 and unifies the transition relation; the rule for $c\mathtt{?}$ simply adds the incoming states to the set of final states, thus reflecting the epsilon case of its parsing. The other combinators are defined by equivalence; finally the parsing of a reference adds transitions from all incoming state to its own state, which also becomes the single final state.

So far, goesTo, S_{acc} and refTo are nothing more than a decomposed representation of a standard labeled transition graph. Since all epsilon transitions are eliminated on the

6 Attributes are not treated in this article.

7 The DTD in the appendix of [14] is non-normative. We took it as a starting point, but defined slightly different abstractions.

fly, and since XML is basically LL(1), the resulting relation corresponds to a *deterministic* finite automaton.[8] This decomposition makes parsing look more complicated in the simple, deterministic case. But it allows to switch into non-deterministic mode by two simple set-based operations, which are the heart of our algorithm, see the boxed expressions in Table 2.[9]

This table shows the operational semantics of the complete parsing process, omitting tactics for error recovery, which can easily be integrated according to [6]. An XML source can be seen as a stream of elements from J, which are open tags, close tags and character data. The nesting of the tags constitute the borders of the encoded element contents. Parsing means to translate such a stream into a tree-like structure N, which is an algebraic data type, built from an element tag and the sequence of the element's children. This is realized by the function translate from Table 2, which starts the process with e_{D}, and succeeds if both the accepting state s_D of this content model and the end of the input are reached.

A stack frame is a pair of the node currently under construction, and the set of active states of the accepting (deterministic or non-deterministic) automaton. The parsing function $\longmapsto$ operates on a pair of such a stack and an input stream and is written as $\kappa \triangleleft f \;/\; b \triangleright \beta \longmapsto \kappa' \triangleleft f' \;/\; b' \triangleright \beta'$. There $\kappa \triangleleft f$ stands for a stack or list with last (top) element f with the predecessors κ, whereas $b \triangleright \beta$ stands for a stream with the first (head) element b and tail β.

Corresponding to the tree nature of XML, the overall parsing is realized by a recursive call of parsers for content models, and the first three rules of table 2 are very similar to those known from standard tree parsers.

The parsing process can take the following steps:

(open) — Whenever an open tag is found, and at least one of the current states goes to a state which consumes this element, then a new stack frame is opened and the parsing of this element is started. This new frame contains the start state of this element as its only member, so the parsing process begins in a traditional deterministic way. This rule is the same for XSLT and result elements and can only be taken if the current and the new element are from the same set; otherwise there is no transition in goesTo.[10]

(chars) — Character data in the input stream is simply appended to the contents of the currently parsed element. If it is not totally made of white space characters (indicated by *WS* in the formula) then the set of states is modified and reflects the consumption of the pseudo element e_{C}. This rule is the same for XSLT and result elements.

(charsWs) — Character data which is totally made of white space characters does not change the set of states. All states which represent "mixed" contents do accept this input without effect on the further transitions, and states from the other content models will treat

[8] XML allows non-deterministic parsing only of empty input sequences, e.g. content models like "`(a*|b*)`", as long as "`(a+|b+)`" would be LL(1).

[9] By employing a library for the manipulation of relations, this algorithm directly describes a practical implementation.

[10] Technical detail: the set of states *after* this step is needed already here to check the legality of the input open tag (the set must be non-empty). For optimization, the calculated result is stored in the stack frame, which from now on reflects the *future* situation after the successful acceptance of the whole new element.

Table 2 Unified parsing process, deterministic and non-deterministic.

$$J ::= \mathsf{open}(E \setminus \{e_\mathrm{C}, e_\mathrm{R}, e_\mathrm{D}\}) \mid \mathsf{close}(E \setminus \{e_\mathrm{C}, e_\mathrm{R}, e_\mathrm{D}\}) \mid \mathit{Chars}$$

$$N ::= \mathsf{node}(E \setminus \{e_\mathrm{C}\} \times \mathsf{SEQ}(N \cup \mathit{Chars}))$$

$$\mathsf{frame} = N \times \mathbb{P}S$$

$$_ \,/\, _ \longmapsto _ \,/\, _ : (\mathsf{SEQ\ frame}) \times (\mathsf{SEQ}\ J) \to (\mathsf{SEQ\ frame}) \times (\mathsf{SEQ}\ J)$$

$$\mathsf{startFrame}(b) = \big(\{\mathsf{startState}(b)\}, \mathsf{node}(b, \langle\rangle)\big)$$

$$\frac{\mathsf{refTo}^{-1}(b) \cap \mathsf{goesTo}(\![s]\!) = s' \neq \{\}}{\kappa \lhd (s, n) \;/\; \mathsf{open}(b) \rhd \beta \longmapsto \kappa \lhd (s', n) \lhd \mathsf{startFrame}(b) \;/\; \beta} \quad \textbf{(open)}$$

$$\frac{\mathsf{refTo}^{-1}(e_\mathrm{C}) \cap \mathsf{goesTo}(\![s]\!) = s' \neq \{\} \qquad \mathit{chars} \in \mathit{Chars} \setminus \mathit{WS}}{\kappa \lhd \big(s, \mathsf{node}(a, \alpha)\big) \;/\; \mathit{chars} \rhd \beta \longmapsto \kappa \lhd \big(s', \mathsf{node}(a, \alpha \lhd \mathit{chars})\big) \;/\; \beta} \quad \textbf{(chars)}$$

$$\frac{\mathit{chars} \in \mathit{WS}}{\kappa \lhd \big(s, \mathsf{node}(a, \alpha)\big) \;/\; \mathit{chars} \rhd \beta \longmapsto \kappa \lhd \big(s, \mathsf{node}(a, \alpha \lhd \mathit{chars})\big) \;/\; \beta} \quad \textbf{(charsWs)}$$

$$\frac{S_\mathrm{acc} \cap s \neq \{\}}{\begin{array}{c}\kappa \lhd \big(s', \mathsf{node}(a, \alpha)\big) \lhd \big(s, \mathsf{node}(b, \beta)\big) \;/\; \mathsf{close}(b) \rhd \gamma \longmapsto \\ \kappa \lhd \big(s', \mathsf{node}(a, \alpha \lhd \mathsf{node}(b, \beta))\big) \;/\; \gamma\end{array}} \quad \textbf{(close)}$$

$$\frac{\mathsf{refTo}^{-1}(e_\mathrm{R}) \cap \mathsf{goesTo}(\![s]\!) = s' \neq \{\} \qquad b \in E_\mathrm{R}}{\begin{array}{r}\kappa \lhd (s, n) \;/\; \mathsf{open}(b) \rhd \beta \longmapsto \\ \kappa \lhd (s', n) \lhd \big(\boxed{\mathsf{refTo}^{-1}(\mathrm{b})}, \mathsf{node}(e_\mathrm{R}, \langle\rangle)\big) \lhd \mathsf{startFrame}(b) \;/\; \beta\end{array}} \quad \textbf{(x2r)}$$

$$\frac{}{\begin{array}{c}\kappa \lhd \big(s, \mathsf{node}(x, \alpha)\big) \lhd \big(s', \mathsf{node}(e_\mathrm{R}, \alpha')\big) \;/\; \mathsf{close}(x) \rhd \beta \longmapsto \\ \kappa \lhd \big(s, \mathsf{node}(x, \alpha \frown \alpha')\big) \;/\; \mathsf{close}(x) \rhd \beta\end{array}} \quad \textbf{(x2r')}$$

$$\frac{a \in (E_\mathrm{R} \cup \{e_\mathrm{R}\}) \qquad b \in E_\mathrm{XI}}{\begin{array}{c}\kappa \lhd \big(s, \mathsf{node}(a, \alpha)\big) \;/\; \mathsf{open}(b) \rhd \beta \longmapsto \\ \kappa \lhd \big(\boxed{\mathsf{goesTo}^*(\![s]\!)} \;/\; \mathsf{node}(a, \alpha)\big) \lhd \mathsf{startFrame}(b), \beta\end{array}} \quad \textbf{(r2x)}$$

$$\mathsf{translate} : \mathsf{SEQ}\ J \twoheadrightarrow N$$

$$\mathsf{translate}(j) = n \iff \mathsf{startFrame}(e_\mathrm{D}), j \longmapsto^* \langle(\{s_\mathrm{D}\}, n)\rangle, \langle\rangle$$

the input as "ignorable whitespace" and are not affected either.[11]

(close) — Whenever the expected close tag is found, and at least one of the active states is accepting, then the top-most stack frame is dropped, the currently parsed element is

[11] Note that the rule "A validating XML processor MUST also inform the application which of these characters constitute white space appearing in element content" from the XML specification [3, sect. 2.10], is somehow ill-defined in the context of XSLT sources. Let "_" symbolize some white space in the input text, like in

```
<xsl:with-param> ___<b/>__<c/></xsl:with-param>
```

Then, together with the result language DTD definitions

```
<!ELEMENT x (a,b,c)>
<!ELEMENT y (#PCDATA|a|b|c)*>
```

this whitespace will be ignorable or not, depending on the *dynamic* context of the later expansion.

considered complete and it is appended to the contents of the element parsed one level above. This rule is the same for any combination of XSLT and result elements on both positions.

So far the rules are only a complicated implementation of well-known standard parsing. But the following rules are specific for fragmented validation and represent the topic and main contribution of this article. They manage the transition between the two sets of D_{X} and D_{R}, and introduce non-determinism:

(x2r) — Whenever an open tag of the result language element appears while an XSLT element is parsed, *two* stack frames are added: the upper one has the artificial element e_{R} as its growing node, and represents the embedding of a sequence of result elements in the XSLT elements' contents. This integration is in a smooth way: no further ad-hoc action or adjustment are necessary, as the comparison between **(open)** and the upper and first part of **(x2r)** shows.

The second frame represents the result element b and its further contents, in the same way as in rule **(open)**. The framed part of the formula makes the difference: since we do not know statically in which context the result elements will be inserted later, when executing the XSLT program, *all states* which refer to the result element (i.e. which are reached by consuming it) are put into the state set of the e_{R}-frame. Here a *first source of non-determinism* comes into play.

Then parsing continues normally, according to **(close)** and **(open)** (and possibly (r2x), see below): the contents of b will be completed, and after its close tag all those sequences of result elements may follow, which may follow in *any* content model from C_R after *any* occurence of b. This is achieved because *all* corresponding states have been entered into the frame by refTo^{-1}.

(x2r') — This process continues until the close tag of x is parsed. Now the accumulated contents of e_{R} are appended to the contents of x, and the stack frame for e_{R} is discarded.[12] After this, the rule **(close)** will apply normally.

(r2x) — Let $E_{\mathrm{X}\mathbf{I}} \subset E_{\mathrm{X}}$ be the XSLT elements from the **I** category, which can appear ubiquitously in result elements. Whenever a corresponding open tag appears in a result element's contents, this is *always* legal and starts the XSLT parsing process. This is very similar to the simple rule **(open)**, as the comparison of the formulas shows.

Additionally, the state set of the result element is upgraded by applying the reflexive-transitive closure of the transition function; see again the framed expression. This reflects the fact that we do not know *how many* of the still required/allowed children from the result content model will later be delivered by the expansion of the XSLT term. This may be anything from the empty list, up to all the missing rest.

Both kinds of non-determinism combine nicely. The underlying relational operations can be implemented efficiently as table lookups; the relations depend only on the two DTDs and their setup time and space complexity grow polynomially with $|S|$ only, hence they can feasibly be precomputed and cached ahead of time. Time consumption of FV is linear with

---

[12] In this version of XSLT, the upper stack frame created by rule (x2r) could be spared, since the information s' is not required after completion of e_R: it stands always at a terminal and accepting position of an XSLT content model. But this does not hold for the general case in which there could be more than one appearance of e_R in a content model.

the size of the XSLT source, and it can be executed in parallel with parsing. As mentioned above, as soon as no transition is possible, an error has been detected. Nevertheless, parsing and FV can be resumed with the next top-level XSLT `template` element. FV is incremental, it can be applied to incomplete programs in each phase of programming, and to general purpose libraries, since it is independent of the input format.

2.2 Generating Diagnostic Information

Whenever the set of active states becomes empty due to the value of the head of the input stream, either a violation of the XSLT syntax or an invalid fragment of the result language is detected. In the latter case error information like

```
Error in xslt file (file id / line number):
The sequence of elements
      [tr][xsl:if][td]
cannot produce valid content w.r.t. "xhtml-1-0-strict.dtd"
```

can easily be derived. A concrete tool can give further hints to the programmer, e.g. print out all content models which ever contributed to the state set in this stack frame.

2.3 Example

Table 3 illustrates the operation of the algorithm. The top shows content models which are indeed a small fragment of a typical situation in practice. The informal notation shows the initial states from startState(e) and the states from (e, s), as defined in Table 1, as exponents. Below the transition system generated by the algorithm from that table.

The operation of the transition function $\longmapsto$ shows on the left side only the element names and state sets of the stack frames, ie. omits the nodes' contents, and on the right only the heads of the input stream. All possibly intervening states and inputs for parsing of the contents of the nodes are also omitted.

The trace ends with the detection of an error: There is no content model which allows more than one `title` element on the same level of nesting.

2.4 Tests

Table 4 shows the results of applying our local analysis FV in comparison to the global *control-flow based* analysis from Møller et al. [10] to their test data. Their approach is referred to as “XSLV” in the following, for a description see the following section on related work. We used version 0.9 of their tool. The rather tedious mining and preparation of this real-world test data is described in detail in [10]. We applied the XSLV tool and our implementation of FV against the XSLT sources of ten of their test cases and the XHTML 1.0 DTD.[13]

The figures in Table 4 are in no way meant competitive. They only can give an impression of the possible impact of both tools on programming and debugging practice.

The first numeric column gives the lines of code of the xslt source. The second column gives the number of errors signalled by XSLV, and the third, labeled “(cf)”, the subset of those which really can only found by control flow analysis. These are the errors which can

[13] Four of the fifteen cases have been excluded because they do not use HTML as their result language; one test case was not re-producable.

Table 3 Example run of the algorithm.

```
html = ¹ head² , body³
head = ⁴ ( script⁵ | style⁶ )*,
    ( title⁷ , ( script⁸ | style⁹ )*, ( base¹⁰ , ( script¹¹ | style¹² )*)?)
    | base¹³ ( script¹⁴ | style¹⁵ )*, title¹⁶ , ( script¹⁷ | style¹⁸ )*
    )
...
h1 = ⁷¹ ( #chars⁷² | a⁷³ | b⁷⁴ | script⁷⁵ )*,
```

$$
\begin{array}{lll}
\mathsf{goesTo} & = \{ & 1 \mapsto 2, 2 \mapsto 3, \\
 & & 4 \mapsto 5, 4 \mapsto 6, 4 \mapsto 7, 4 \mapsto 13 \\
 & & 5 \mapsto 5, 5 \mapsto 6, 5 \mapsto 7, 5 \mapsto 13, \\
 & & 6 \mapsto 5, 6 \mapsto 6, 6 \mapsto 7, 6 \mapsto 13, \\
 & & 7 \mapsto 8, 7 \mapsto 9, 7 \mapsto 10, \\
 & & 8 \mapsto 8, 8 \mapsto 9, 8 \mapsto 10, \\
 & & 9 \mapsto 8, 9 \mapsto 9, 9 \mapsto 10, \\
 & & 10 \mapsto 11, 10 \mapsto 12, 11 \mapsto 11, 11 \mapsto 12, 12 \mapsto 11, 12 \mapsto 12, \\
 & & 13 \mapsto 14, 13 \mapsto 15, 13 \mapsto 16, \\
 & & 14 \mapsto 14, 14 \mapsto 15, 14 \mapsto 16, 15 \mapsto 14, 15 \mapsto 15, 15 \mapsto 16, \\
 & & 16 \mapsto 17, 16 \mapsto 18, 17 \mapsto 17, 17 \mapsto 18, 18 \mapsto 17, 18 \mapsto 18, \\
 & & \ldots \\
 & & 71 \mapsto 72, 71 \mapsto 73, 71 \mapsto 74, 71 \mapsto 75, \\
 & & 72 \mapsto 72, 72 \mapsto 73, 72 \mapsto 74, 72 \mapsto 75, \\
 & & 73 \mapsto 72, 73 \mapsto 73, 73 \mapsto 74, 73 \mapsto 75, \\
 & & 74 \mapsto 72, 74 \mapsto 73, 74 \mapsto 74, 74 \mapsto 75, \\
 & & 75 \mapsto 72, 75 \mapsto 73, 75 \mapsto 74, 75 \mapsto 75 \\
 & \} & \\
S_{\mathrm{acc}} & = & \{3, 7, 8, 9, 10, 11, 12, 16, 17, 18, 71, 72, 73, 74, 75\}
\end{array}
$$

$$
\begin{array}{lrcl}
 & \kappa_1 = \langle \ldots, (\mathtt{xsl:template}\langle\rangle\{\ldots\})\rangle & / & \mathsf{open}(\mathtt{script}) \triangleright \ldots \\
\longmapsto & \kappa_1 \triangleleft (\mathtt{e_R}\langle\rangle\{5, 8, 11, 14, 17, 75\}) \triangleleft (\mathtt{script}\langle\rangle\{\ldots\}) & / & \mathsf{close}(\mathtt{script}) \triangleright \ldots \\
\longmapsto & \kappa_1 \triangleleft (\mathtt{e_R}\langle\rangle\{5, 8, 11, 14, 17, 75\}) & / & \mathsf{open}(\mathtt{style}) \triangleright \ldots \\
\longmapsto & \kappa_1 \triangleleft (\mathtt{e_R}\langle\rangle\{6, 9, 12, 15, 18\}) \triangleleft (\mathtt{style}\langle\rangle\{\ldots\}) & / & \mathsf{close}(\mathtt{style}) \triangleright \ldots \\
\longmapsto & \kappa_1 \triangleleft (\mathtt{e_R}\langle\rangle\{6, 9, 12, 15, 18\}) & / & \mathsf{open}(\mathtt{title}) \triangleright \ldots \\
\longmapsto & \kappa_1 \triangleleft (\mathtt{e_R}\langle\rangle\{7, 16\}) \triangleleft (\mathtt{title}\langle\rangle\{\ldots\}) & / & \mathsf{close}(\mathtt{title}) \triangleright \ldots \\
\longmapsto & \kappa_1 \triangleleft (\mathtt{e_R}\langle\rangle\{7, 16\}) & / & \mathsf{open}(\mathtt{xsl:if}) \triangleright \ldots \\
\longmapsto & \kappa_1 \triangleleft (\mathtt{e_R}\langle\rangle\{7, 8, 9, 10, 11, 12, 16, 17, 18\}) \triangleleft (\mathtt{xsl:if}\langle\rangle\{\ldots\}) & / & \mathsf{close}(\mathtt{xsl:if}) \triangleright \ldots \\
\longmapsto & \kappa_1 \triangleleft (\mathtt{e_R}\langle\rangle\{7, 8, 9, 10, 11, 12, 16, 17, 18\}) & / & \mathsf{open}(\mathtt{title}) \triangleright \ldots \\
\longmapsto & \kappa_1 \triangleleft (\mathtt{e_R}\langle\rangle\{\}) & &
\end{array}
$$

never be found by the purely local method of FV. But if the test data is fairly representative, as claimed in the discussion in [10], then these low figures support our approach.

The last column shows the number of errors detected by our tool, most of them by FV, enhanced by checks for missing and wrongly used *attribute values*, based on a similar, but simpler local strategy. Not considering the (cf) errors, both tools always found the same errors, and one tool some additionals. The cases when FV "won" are due to implementation flaws: Theoretically XSLV finds all errors FV can find.

2.5 Execution of the XSLT program

In our implementation fragmented validation is performed when constructing an internal data model of an XSLT program. In the context of our metatools framework [16], XML models are driven by DTDs, and rely on the fundamental property that DTD content models

Table 4 Test results: control flow-based XSLV vs. local-only FV.

Nr	Testcase Nickname	loc	XSLV tool	(cf)	FV tool
1	poem	36	3		3
2	AffordableSupplies	42	8		8
3	agenda	43	2		1
6	adressebog	76	2	1	1
8	slideshow	119	2		0
9	psicode-links	128	8	2	12
11	proc-def	258	7	1	10
12	email_list	243	2		3
13	tip	265	7	2	3
14	window	701	4		1

are specific for element names and do not depend on the context. In this framework, the tdom tool is a program which translates a DTD into a collection of JAVA sources which can realize exclusively all well-typed text corpora w.r.t. this DTD: classes are generated for every element declaration, and for every sub-expression of a nested content model. On all levels, all constructor methods ensure and all setter methods preserve validity.

In the context of XSLT, this technology is applied throughout, except for the “two-coloured” lists which combine elements from both sets. They are realized by lists of the supertype of both tdom models. Since no tdom element instance can be constructed with such a sequence as contents, the two-coloured nature propagates up the document tree, until it is absorbed by the synthetic element e_R, which combines a two-coloured downside with a well-typed upside (see Fig. 2).

With this data model the evaluation of an XSLT program becomes a very simple transformation: all subtrees from XSLT must be replaced by their evaluation results; the two-coloured lists turn into homogeneous ones of result type, and only these must finally be parsed incrementally into a tdom subtree. All other contents have already been checked statically when creating the program’s model, by fragmented validation.

3 Outlook

We have presented an enhancement of a standard tree parser algorithm applied to XSLT sources. Simple algebraic operations on the transition relation reflect the non-determinism which is induced by XSLT elements serving as parents or siblings of result elements. This abstract interpretation allows to detect a substantial share of typing errors in sources of XSLT transformations by an easy to implement “on-the-fly” algorithm.

3.1 Future Work

This kind of abstract interpretation, which operates on sequences of sets of states, seems promising for further research and possible natural extensions.

First, the treatment of XSLT instructions embedded into output can be differentiated further, according to their known meaning: for instance, a `<comment>` element can not affect the parsing state, and an `<element name=...>` instruction can be treated as a *verbatim* result element, if the name attribute’s value can be determined statically.

Also, in a general sense, extensions are possible: assume a sequence mixed from result and XSLT elements r_1, x, r_2 is parsed, and the set of states after parsing r_1 is s_1. Then when parsing x the rule **(r2x)** widens the set of states to the set s_2 by applying the reflexive-transitive closure of the transition function, which represents the uncertainty of the future execution, and for r_2 the rule **(open)** narrows the set of states again to the set s_3. From these sets immediately follows a certain *type* of the XSLT function x, which in any case must produce a sequence of elements which lie on a path from some state in s_1 to some state in s_3.

In many cases this type information can be exploited. E.g. an XSLT `<if>` expression with constant contents can only expand to that contents or to the empty sequence. Similarly, a `<choose>` represents the disjunction of its contents. When x is the call of a "named template", then all types demanded by all places of such calls can be intersected for inference.

It seems worth exploring how far such a notion of "type" will lead. Possibly not very far w.r.t. absolutely preventing typing errors, simply due to the proven un decidability of this problem in general. E.g. as soon as dynamically selected processing is initiated by an `<apply-templates>` statement, our approach reaches its limits. But in any case a regular expression can be synthesized and delivered to the programmer as a *hint* which sequence of elements the code of such a "framed" sub-expression, like x in the example above, must deliver.

In the field of XSLT processing some benchmarks and standard conformance test suites had been developed, but most of them already have disappeared again. A recent benchmark framework and test case collection has been released by Saxonica company as open source project [5]. How far this can be adopted to our "DTD aware" approach is currently under research. The same holds for the only still available conformance test suite by OASIS [1]. The results of applying them both would be very valuable, but since for our tool the result document DTD must always be provided, e.g. re-constructed, we expect this to become rather expensive soon, esp. in the second case with its nearly four thousand test cases.

3.2 Related Work

The problem of type checking XML transformations in a general sense has been studied thoroughly during the last decade in dozens of papers in the context of data base queries and of dedicated functional XML transformation languages, but seldom w.r.t. XSLT. For a survey see [12] and [11].

Very early proposals and influential suggestions can be found in [2]. This paper is purely theoretical. It translates a very simple subset of XSLT into a collection of formal constraints, but excludes the implications of XPath navigation and the `<apply-templates>` matching mechanism completely.

Tozawa [13] also restricts the analyzed transformation language to a non-Turing-complete subset of XSLT, excluding XPath horizontal and upwards navigation, and again the implications in control flow induced by pattern matching. The chosen technique is *backward type inference*, which infers the type of all input documents w.r.t. a given result type. It seems that this interesting approach could not be extended to full-scale XSLT.

A variant called "exact type checking" by its authors restricts types and transformations until type checking becomes decidable. (It is of course highly desirable that analyses of this kind would be carried out *before* the corresponding industrial standardization decisions are met.) For most advanced results and a survey on this line, see Maneth, Perst and Seidel [8].

The opposite approach is to look at the full functional range of XSLT and execute analysis as far as possible. This approach is more related to programming practice and thus to our

FV. The current standard in this field is the work of Møller, Olesen and Schwartzbach [10]. Their approach covers two very different problems: first from the collection of all statements of the form <apply-templates select="α"> and <template match="β"> in the program, an upper limit of the control flow graph is derived. (I.e. flow of execution from template to template, enriched with the change of the “current input focus” from one element tag to a set of possible element tags.)

All templates are translated into data graphs, which each can produce a certain language of result trees, and then these graphs are plugged together according to the flow graph from step one. In a second step it is checked that the resulting overall data graph produces only output which is in the language of the required result type. For most violations detected, detailed diagnostic information can be derived. Their solution has been implemented, is freely available, and has been tested with considerable amounts of real-world test data.

This valuable work plays of course in a different league than our approach. Tellingly, their paper takes nearly thirty pages to describe the algorithm, excluding the second step which is cited from an earlier work. Nevertheless, what they can calculate is (naturally) still an approximation, albeit a very good one. A comparison between of practical test results with both tools is given in section 2.4 above.

Currently, the most widely used XSLT processors (Xalan, Saxon, XT, libxslt) do not include any type checking, not even automated validation. The 2.0 version of the XSLT standard [15] defines a feature called “schema awareness” for XSLT processors. This includes the ability to read and define schema information in the sense of the W3C Schema language [4]. The standard foresees this information for *explicit validation* of subtrees of the generated output, controlled selectively by the programmer. Currently only the latest versions of some XSLT engines support this still-evolving standard. While it is arguable from the compiler construction point of view whether this is the right direction of development (these explicit excursions to the meta level are called “pragmas” in general-purpose programming languages, and generally deprecated for portability), the same information could be used in future for feeding type checking algorithms.

Acknowledgments. A two page extended abstract of this work has been published in the ICMT 2013 [7]. Many thanks to Anders Møller for giving us access to the XSLV tool and test cases. And to the anonymous reviewers for valuable hints.

References

1. *OASIS XSLT Conformance TC Public Documents.* OASIS, 2005. `http://www.oasis-open.org/commitees/tc_home.php/?wg_abbrev=xslt`.
2. Philippe Audebaud and Kristoffer Rose. Stylesheet validation. Technical report, Laboratoire de l’Informatique du Parallélisme, 2000.
3. Tim Bray, Jean Paoli, C.M. Sperberg-McQueen, Eve Maler, Fran cois Yergeau, and John Cowan. *Extensible Markup Language (XML) 1.1 (Second Edition).* W3C, `http://www.w3.org/TR/2006/REC-xml11-20060816/`, 2006.
4. David C. Fallside and Priscilla Walmsley. *XML Schema Part 0: Primer Second Edition.* W3C, `http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/`, 2004.
5. Michael Kay and Debbie Lockett. Benchmarking xslt performance. In *XML London 2014 – Conference Proceedings*, volume 10, pages 23–38. XML London, 2014.
6. Markus Lepper and Baltasar Trancón y Widemann. d2d — a robust front-end for prototyping, authoring and maintaining XML encoded documents by domain experts. In Joaquim Filipe and J.G.Dietz, editors, *Proceedings of the International Conference on Knowledge*

Engineering and Ontology Delelopgment, KEOD 2011, pages 449–456, Lisboa, 2011. SciTePress.

7 Markus Lepper and Baltasar Trancón y Widemann. Fragmented validation — a simple and efficient contribution to xslt checking (extended abstract). In *Proc. ICMT 2013, Int. Conference on Theory and Practice of Model Transformations*, volume 7909 of *LNCS*. Springer, 2013.

8 Sebastian Maneth, Thomas Perst, and Helmut Seidl. Exact xml type checking in polynomial time. In *In ICDT*, pages 254–268, 2007.

9 Wim Martens and Frank Neven. On the complexity of typechecking top-down XML transformations. *Theoretical Computer Science*, 336:153–180, 2005.

10 Anders Møller, Mads Østerby Olesen, and Michael I. Schwartzbach. Static validation of XSL Transformations. *ACM Transactions on Programming Languages and Systems*, 29(4), July 2007.

11 Anders Møller and Michael I. Schwartzbach. The design space of type checkers for XML transformation languages, 2004.

12 Dan Suciu. The XML typechecking problem. *SIGMOD Rec.*, 31(1):89–96, March 2002.

13 Akihiko Tozawa. Towards static type checking for XSLT. In *Proceedings of the 2001 ACM Symposium on Document engineering*, DocEng '01, pages 18–27, New York, NY, USA, 2001. ACM.

14 W3C, `http://www.w3.org/TR/1999/REC-xslt-19991116`. *XSL Transformations (XSLT) Version 1.0*, 1999.

15 W3C, `http://www.w3.org/TR/2007/REC-xslt20-20070123/`. *XSL Transformations (XSLT) Version 2.0*, 2007.

16 Baltasar Trancon y Widemann, Markus Lepper, and Jacob Wieland. Automatic construction of XML-based tools seen as meta-programming. *Automated Software Engineering*, 10(1):23–38, 2003.

A Mathematical Notation

The employed mathematical notation borrows from the Z formalism. For convenience, the following table shows the details which are beyond basic set theory.

$A \rightarrow B$	The type of *total* functions from A to B
$A \nrightarrow B$	The type of *partial* functions from A to B
$A \leftrightarrow B$	The type of relations from A to B
$f \,(\!\mid s \mid\!)$	The image of set s under function or relation f
$f^{-1}(y)$	The preimage of value y under function f
r^*	The reflexive-transitive closure of relation r
$A \times B$	The product type of two sets A and B, i.e. all pairs $\{c = (a, b) \vert a \in A \wedge b \in B\}$.
π_n	The nth component of a tuple.
$\mathbb{P}(A)$	Power set, the type of all subsets of the set A.
SEQ A	The type of finite sequences from elements of A
$\langle\rangle$	The empty sequence
$\alpha \frown \alpha'$	Concatenation of sequences α and α'
$a \triangleright \alpha$	A stream(/list/sequence) with element a followed by rest α
$\alpha \triangleleft a$	A list(/stack) with element a preceded by rest α

DynSem: A DSL for Dynamic Semantics Specification

Vlad Vergu, Pierre Neron, and Eelco Visser

Delft University of Technology
Delft, The Netherlands
{v.a.vergu|p.j.m.neron|visser}@tudelft.nl

Abstract

The formal semantics of a programming language and its implementation are typically separately defined, with the risk of divergence such that properties of the formal semantics are not properties of the implementation. In this paper, we present DynSem, a domain-specific language for the specification of the dynamic semantics of programming languages that aims at supporting both formal reasoning and efficient interpretation. DynSem supports the specification of the *operational semantics* of a language by means of *statically typed conditional term reduction rules*. DynSem supports *concise* specification of reduction rules by providing *implicit build and match coercions* based on reduction arrows and implicit term constructors. DynSem supports *modular* specification by adopting implicit propagation of semantic components from I-MSOS, which allows omitting propagation of components such as environments and stores from rules that do not affect those. DynSem supports the declaration of *native operators* for delegation of aspects of the semantics to an external definition or implementation. DynSem supports the definition of auxiliary *meta-functions*, which can be expressed using regular reduction rules and are subject to semantic component propagation. DynSem specifications are *executable* through automatic generation of a Java-based AST interpreter.

1998 ACM Subject Classification F.3.2. Semantics of Programming Languages (D.3.1.)

Keywords and phrases programming languages, dynamic semantics, reduction semantics, semantics engineering, IDE, interpreters, modularity

Digital Object Identifier 10.4230/LIPIcs.RTA.2015.365

1 Introduction

The specification of the dynamic semantics is the core of a programming language design as it describes the runtime behavior of programs. In practice, the implementation of a compiler or an interpreter for the language often stands as the *only* definition of the semantics of a language. Such implementations, in a traditional programming language, often lack the clarity and the conciseness that a specification in a formal semantics framework provides. Therefore, they are a poor source of *documentation* about the semantics. On the other hand, formal definitions are not executable to the point that they can be used as implementations to run programs. Even when both a formal specification and an implementation co-exist, they typically diverge. As a result, important properties of a language as established based on its formal semantics may not hold for its implementation. Our goal is to unify the semantics engineering and language engineering of programming language designs [22] by providing a notation for the specification of the dynamic semantics that can serve both as a readable formalization as well as the source of an execution engine.

In this paper, we present DynSem, a DSL for the concise, modular, statically typed, and executable specification of the dynamic semantics of programming languages. DynSem

© Vlad Vergu, Pierre Neron, and Eelco Visser;
licensed under Creative Commons License CC-BY
26th International Conference on Rewriting Techniques and Applications (RTA'15).
Editor: Maribel Fernández; pp. 365–378
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

supports the specification of the *operational semantics* of a language by means of *conditional term reduction rules*.

- DynSem rules are *statically typed* with respect to the declaration of the signature of constructors of abstract syntax nodes and values, and the declaration of typed (and optionally named) reduction arrows.
- DynSem supports *concise* specification of reduction rules by providing *implicit build and match coercions* based on reduction arrows and implicit term constructors.
- DynSem supports *modular* specification by adopting the implicit propagation of semantic components from the I-MSOS (Implicitly Modular Structural Operational Semantics) formalism [16], which allows omitting propagation of components such as environments and stores from rules that do not affect those. The implicit semantic components are explicated through a source-to-source transformation, to produce a complete and unambigous specification. This formalizes informal conventions that are often adopted in language specifications such as *The Definition of Standard ML* [14]. The declaration of variable schemes for semantic components allows their concise identification in rules, following typical naming conventions in typeset renderings of semantic rules.
- DynSem supports the declaration of *native operators* for delegation of aspects of the semantics to an external definition or implementation. For example, the details of the semantics of numeric operations may be abstracted over by delegating to a library of native numeric operators.
- DynSem supports the definition of auxiliary *meta-functions*, which can be expressed using regular reduction rules and are subject to semantic component propagation. Thus, one can provide abstractions over run-time operations that one *does* want to define explicitly in the semantics, without having to inline their definition in each rule that uses them.
- DynSem specifications are *executable* through automatic generation of a Java-based AST interpreter. DynSem specifications can be written either in big-step style or in small-step style, but our interpreter generator is geared towards big-step style rules. While small-step style rules are compiled as well, the interpreter generator does not (yet) provide any optimizations (such as refocusing [18]) to reduce the excessive traversals inherent in small-step specifications. The interpreter generated from big-step style specifications has reasonable performance, allowing one to run and experiment with the design of a language on concrete programs.

The DynSem language is integrated into the Spoofax language workbench, a tool for the definition of (domain-specific) programming languages [11]. From a language definition, Spoofax generates a complete programming environment including parser, type checker, and language-aware editor (IDE). The DynSem integration of Spoofax extends these programming environments with generated interpreters. DynSem itself is built using Spoofax and comes with a full-fledged IDE.

The rest of the paper is organized as follows. In the next section we introduce the definition of term reduction rules and the definition of signatures that declare sorts, constructors, and arrows. In Section 3 we discuss the features that DynSem provides for supporting concise definitions of rules. In Section 4 we describe the generation of interpreters from DynSem specifications. Finally, in Section 5 and Section 6 we discuss related and future work.

2 DynSem Reduction Rules

The specification of a language semantics in DynSem is expressed using term reduction rules such as the one presented in Figure 1. This rule is the fully explicit version of the definition

```
rules
  Env E ⊢ Plus(e1, e2) :: Sto s1 ⟶ NumV(PlusI(n1, n2)) :: Sto s3
  where
    Env E ⊢ e1 :: Sto s1 ⟶ NumV(n1) :: Sto s2,
    Env E ⊢ e2 :: Sto s2 ⟶ NumV(n2) :: Sto s3
```

Figure 1 Fully explicit addition reduction rule.

of the reduction of an addition constructor `Plus` in a language whose semantics uses both an environment and a store. In general, a reduction rule in DynSem has the form

```
    Rs ⊢ t1 :: Ws -a-> t2 :: Ws′
    where ps
```

and defines the reduction `t1` $\xrightarrow{a}$ `t2` from a term matching term pattern `t1` to a term obtained by instantiating term pattern `t2`, provided that the premises `ps` succeed. Reduction arrows are identified by their name and the type of their input (left) argument term. Arrows can be, and typically are, unnamed (i.e. use the default name) as is the case in Figure 1.

The *premises* of a reduction rule are of one of the following forms:

- A *reduction premise* is the (recursive) invocation `Rs ⊢ t :: Ws` $\xrightarrow{b}$ `t′ :: Ws′` of a reduction on the term `t`, matching the result against the term pattern `t′`. The premises in Figure 1 are of this form.
- An *equality or inequality premise* tests the equality or inequality of two terms. For example, in Figure 6, the `Ifz` constructor has two corresponding reduction rules that are disambiguated depending on the integer the first sub-term (the condition) reduces to.

`Ws′` are used to describe the context in which a particular reduction takes place. For example, the environment `Env E` and the store `Sto S` are semantics components in the rule in Figure 1.

In the conclusion of a rule, the left-hand side terms (`Rs`, `t1` and `Ws`) are patterns binding the meta-variables they contain, while the terms on the right-hand side (`t2` and `Ws′`) are instantiated to construct the result of reduction. This is reversed in reduction premises, where the left-hand side terms are term instantiations, while the right-hand side terms are binding patterns. A variable can only be bound once per rule.

DynSem rules are *statically typed* with respect to the declaration of the signature of term constructors, the declaration of typed (and optionally named) reduction arrows, and the declaration of semantic components.

Term Signature. DynSem rules define reductions on *terms* that represent the *abstract syntax trees* of a programming language. The abstract syntax tree constructors of a language are declared as an algebraic signature, as illustrated in Figure 2. Semantic rules typically use additional intermediate representations and final values, which are also represented using terms and should also be declared in the algebraic signature. For example, in a big-step semantics the values produced by reductions are typically not a subset of the abstract syntax, but are defined as a separate set of terms.

The base components of an algebraic signature are *sorts* and *constructors*. The sorts of the object language such as the sorts `Expr` and `Bind` in Figure 2 and the sort of the intermediate or output values such as the sort of values `V` in Figure 3 are declared in the

```
signature
  sorts Expr Bind
  constructors
    Num    : String → Expr
    Plus   : Expr * Expr → Expr
    Var    : String → Expr
    Bind   : String * Expr → Bind
    Let    : List<Bind> * Expr → Expr
    Fun    : String * Expr → Expr
    App    : Expr * Expr → Expr
    Box    : Expr → Expr
    Unbox  : Expr → Expr
    SetBox : Expr * Expr → Expr
    Seq    : Expr * Expr → Expr
    Ifz    : Expr * Expr * Expr → Expr
```

Figure 2 Signatures of the object language.

```
signature
  sorts V
  semantic-components
    Env = Map<String, V>
    Sto = Map<Int, V>
  constructors
    NumV  : Int → V  {implicit}
    ClosV : Expr * Env → V
    BoxV  : Int → V
  arrows
    Expr  ⟶ V
    List<Bind> -b-> Env
  native operators
    plusI   : Int * Int → Int
    str2int : String → Int
```

Figure 3 Auxiliary components.

```
signature
  constructors
    allocate : V  ⟶ Int
    write    : Int * V  ⟶ V
    read     : Int  ⟶ V
```

Figure 4 Meta-functions.

```
signature
  variables
    E : Env
    S : Sto
```

Figure 5 Variable Schemes.

corresponding `sort` section in the **`signature`**. Constructor declarations consist of a name, the sorts of the argument terms, and the result sort.

In addition to the declared sorts, the parametric sorts, **`List`** and **`Map`** can be used in signatures. Given a sort `X`, **`List`**`<X>` is the sort of lists of terms of sort `X`. The corresponding terms are `[]`, which denotes the empty list and `[hd|tl]`, which denotes the list with head element `hd` and tail `tl` (following Prolog list notation). Given two sorts `X` and `Y`, **`Map`**`<X,Y>` denotes the sort of finite maps from terms of sort `X` to terms of sort `Y`. The corresponding terms are `{}`, which denotes the empty mapping, `{x ↦ v, S}`, which extends the mapping `S` by associating `v` to `x`, and `S[l]`, which denotes the term associated with `l` in mapping `S`. These sorts can be used directly in constructor signatures such as the `Let` constructor in Figure 2 that takes a list of binders as first parameter.

Semantic Components. The specification of operational semantics using big-step rules defines reduction relations that involve not only abstract syntax terms but also other entities, known as *semantic components*, representing the state of a program under reduction, such as an environment or a store. DynSem distinguishes two types of semantic components based on how they are passed through the reduction rules: *read-only* and *read-write* semantic components. Terms of any sort may be used as semantic component after being declared as such. The **`semantic-components`** section in Figure 3 declares `Env` as a semantic component of sort **`Map`**`<String,V>`, mapping identifiers (strings) to values, and `Sto` as a semantic component of sort **`Map`**`<Int,V>`, mapping locations (integers) to values.

Semantic components appearing left of $\vdash$ (in `Rs`) are *read-only*. These components propagate downwards, i.e. are passed into premises, but do not propagate horizontally, i.e. changes introduced by one premise are not visible in the next premise. A typical example of a read-only component is an environment, mapping variables to values. Downward propagation

corresponds to the lexical scope of variables. Components that appear right of :: are *read-write* semantic components. Since the value of such components is threaded through the computation, they appear on both the right and the left side of a reduction. A typical example of a read-write component is a store mapping locations to values. Threading a store is expressed as `e :: Sto st` $\longrightarrow$ `v :: Sto st'` when reduction of expression `e` into value `v` changes the corresponding store form `st` to `st'`. Store threading corresponds to the dynamic extent of store mappings.

Arrows. A reduction arrow defines a relation between terms, possibly involving semantics components. However, only the sorts of the source and target terms of an arrow have to be declared in its signature. The set of semantic components an arrow uses is inferred, as described in Section 3. The signature in Figure 3 declares two arrows, the main evaluation arrow $\longrightarrow$ from expressions `Expr` to values `V`, and the binder reduction $\xrightarrow{b}$ from lists of binders to environments. Arrow declarations can be overloaded on the input sort. That is, one can declare several arrows with the same name, as long as they have different input types. For example, in Figure 3 we could also use $\longrightarrow$ instead of $\xrightarrow{b}$ to denote the reduction from lists of binders to environments.

Native Operators and Meta Functions. The core rules of a semantics define reductions on the abstract syntax terms of the language under consideration. However, some aspects of the semantics we would like to delegate, i.e. not define directly. DynSem provides two mechanism for such delegation.

A *native operator* is a function that is defined externally to DynSem. The specification just records its signature. A typical use of native operators is the API for numeric operations. For example, Figure 3 declares the native operators `plusI` for adding integers and `str2int` for parsing strings as integer literals. From the perspective of the type system, native operators are term constructors. From the perspective of reduction, a native operator comes with an implementation that replaces its invocation with a term of the appropriate sort.

Sometimes we do want to define the semantics of an operation in DynSem, but we want to abstract over a frequently occurring pattern. A *meta-function*, declared as a constructor `C: ts` $\longrightarrow$ `t` (with a long arrow), introduces an auxiliary term constructor in combination with an arrow evaluating it to its result type. For example, Figure 4 declares the signature of meta-functions `allocate`, `write`, and `read`, which abstract over operations on the store, simplifying the definition of rules using the store.

3 Conciseness and Modularity

The previous section introduced the general form of DynSem reduction rules and the signatures that declare the structure of terms and arrows. However, as one can observe in Figure 1, full-fledged reduction rules can be rather verbose, requiring quite a bit of boilerplate code for the reduction of subterms and the propagation of semantic components. Such verbosity is bad since it hides the essence of a rule; which parts of the rule in Figure 1 are truly related to the addition? Worse, the explicit propagation of semantic components is harmful to modularity. When introducing a new feature to a language that requires a new semantic component, all existing rules need to be adjusted to propagate the new semantic component. It is common practice to reduce the verbosity of rules by means of informal conventions [14]. DynSem provides several features formalizing such conventions so that rules are unambiguous, yet have can be defined concisely and modularly. These features can be separated in two

```
Num(n) ⟶ str2int(n)

Ifz(0, e1, _) ⟶ e1

Ifz(n: Int, _, e2) ⟶ e2
where
  n ≢ 0

E ⊢ Var(x) ⟶ E[x]

[] : List(Bind) -b-> {}

[Bind(x, v: V) | lb] -b-> {x ↦ v, E}
where
  lb -b-> E

E ⊢ Let(E_b, e) ⟶ v'
where
  Env {E_b, E} ⊢ e ⟶ v'

Seq(v1 : V, e2) ⟶ e2
```

```
Plus(NumV(i1), NumV(i2)) ⟶ plusI(i1, i2)

E ⊢ f@Fun(_, _) ⟶ ClosV(f, E)

App(ClosV(Fun(x, e), E), v: V) ⟶ v'
where
  Env {x ↦ v, E} ⊢ e ⟶ v'

Box(e) ⟶ BoxV(allocate(e))

Unbox(BoxV(loc)) ⟶ read(loc)

SetBox(BoxV(loc), e) ⟶ write(loc, e)

allocate(v) ⟶ loc
where
  fresh ⇒ loc, write(loc, v) ⟶ _

write(loc, v) :: S ⟶ v :: Sto {loc ↦ v, S}

read(loc) :: S ⟶ S[loc] :: S
```

Figure 6 DynSem reduction rules for a functional language with references (boxes). The meta-functions allocate, write, and read define reusable abstractions over store operations that are used in the definition of the rules for boxes.

categories: (1) *coercions* that allow implicit transformation from one sort into another, and (2) *implicit propagation of semantics components* that allows the omission of those semantic components that are not affected by a rule.

3.1 Implicit Coercions

Implicit Constructors. Including the terms of one sort in another sort requires the introduction of an explicit constructor. For example, in order to use integers (sort Int) as values (sort V), a constructor NumV: Int → V is required, which should be applied to wrap each use of an integer as a value. In order to avoid such unnecessary constructor applications we allow unary constructors to be declared as implicit. An implicit constructor C with signature S1 →S2 defines a coercion between sort S1 and sort S2. That is, one can use a term of sort S1 in a context where a term of sort S2 is expected. The constructor C will then be automatically inserted to match the context. Therefore, since the NumV constructor is declared as implicit in the signature presented in Figure 3, we can omit it when the expected sort is known, such as in the first rule in Figure 6 for Num(n) where the right-hand side is an Int while a V is expected according to the ⟶ arrow declaration. Coercion expansion will introduce this implicit constructor, transforming the rule into

```
Num(n) ⟶ NumV(str2int(n))
```

Arrows as Coercions. In big-step operational semantics, most of the reductions rules have a similar scheme. That is, for a given constructor, reduce its subterms to values using a (recursive) invocation of the arrow and then combine these values to produce the final value of the term. If the reductions of the sub-terms behave uniformly with respect to semantic components, we can infer the sub-reduction by treating an arrow as declaration of an implicit coercion. Implicit arrow coercions are made explicit by the introduction of reduction premises.

If a term t of sort A appears in a position where sort B is expected, and if there is an arrow from A to B, an additional premise reducing t to a variable of sort B will be introduced. Conversely, if a pattern t of sort B is used in a position where a pattern of sort A is expected, and there is an arrow from A to B, the pattern is replaced with a fresh meta-variable, which is subsequently reduced in a new reduction premise to pattern t. For example making coercions explicit in the Ifz rules of Figure 6 leads to the following rules:

```
rules
  Ifz(e0, e1, _) ⟶ v1
  where
    e0 ⟶ NumV(0),
    e1 ⟶ v1
```

```
  Ifz(e0, _, e2) ⟶ v1
  where
    e0 ⟶ NumV(n),
    n ≢ 0,
    e2 ⟶ v1
```

In order to force coercion applications, one can also annotate variables in a pattern with their required type (e.g. the n : Int pattern in the second Ifz rule) enforcing a coercion from this type to the expected one to be introduced.

Variable Schemes. Big-step rules for operational semantics often require the use of several semantics components. In such a case, the use of meta-variables to represent the different semantics components may lead to an ambiguity about which semantics component is represented by a particular variable. Therefore, the name of a semantic component is required to appear before the term that represents its value, as in Figure 1. DynSem supports writing semantic components using just a variable, through a declaration of variable schemes. Given a semantics components SC, a variable scheme declaration such as S : SC reserves the name S and several extensions (e.g. Sn where n is an integer or S_x where x is any identifier extension) to be used as meta-variables for the semantic component SC. These reserved variables do not require the explicit semantic component name annotation as shown in the variable rule in Figure 6 where E is inferred to be the semantic component Env given the variable declaration in Figure 5.

3.2 Implicit Semantic Components

While the rule for addition in Figure 1 explicitly passes the environment and store to its premises, the rules in Figure 6 do not mention semantic components or only selectively. DynSem uses implicit propagation of semantic components, as introduced in I-MSOS [16], to support the ommission of semantic components in rules that are not affected them. For example, the rule in Figure 1 can be rewritten to the Plus rule in Figure 6. Explication of coercions and semantic components automatically transforms this rule to the one in Figure 1. In addition to being more concise, this rule is also more modular, since it can be used in combination with language constructs that require different semantic components.

Semantic Component Dependency. In order to ensure consistency of the reduction relations, all reduction rules for a given arrow have to be extended with the same set of read-only and read-write semantic components. Moreover a particular semantic component can not appear both as a read-only and read-write component. Therefore, before explicating the semantic components in all rules, we first infer which semantic components are implicitly used by each arrow in the specification.

The semantic components used by an arrow do not only depend on the rules defining the arrow but also on the semantic components used by other arrows on which it depends. Let $\xrightarrow{r}$ denote a particular reduction arrow, the set of semantic components that have to appear in the expansion of the reduction rules defining $\xrightarrow{r}$ has to satisfy the following properties:

```
Env E_0 ⊢ [Bind(x, e_0) | lb] :: Sto S_0 -b-> {x ↦ v, E} :: Sto S_2
 where
   Env E_0 ⊢ e_0 :: Sto S_0 ⟶ v :: Sto S_1,
   Env E_0 ⊢ lb :: Sto S_1 -b-> E :: Sto S_2

E ⊢ Let(b_0, e) :: Sto S_0 ⟶ v' :: Sto S_2
 where
   Env E_0 ⊢ b_0 :: Sto S_0 -b-> E_b :: Sto S_1,
   Env {E_b, E} ⊢ e :: Sto S_1 ⟶ v' :: Sto S_2
```

Figure 7 Explicated rules for Bind and Let.

- All the semantic components appearing explicitly in one of the occurrences of arrow $\xrightarrow{r}$, either in the premise or conclusion of a rule, should appear in its expansion.
- A semantic component appearing in the expansion of an arrow $\xrightarrow{r}$, which is used in the premise of a rule defining arrow $\xrightarrow{r}$, should appear in the expansion of $\xrightarrow{r}$.

Explication of Semantic Components. Given these dependencies, we define an *explication* transformation on reduction rules, which make the use of semantic components explicit. Assume that for every reduction relation $\xrightarrow{r}$ we know its corresponding set of read-only $R_{\mathtt{r}}$ and read-write $W_{\mathtt{r}}$ semantic components according to the dependencies introduced above. Then, for each reduction rule, we apply the following transformations:

- Extend the set of read-only and read-write semantic components of the left-hand side of the conclusion with free variables for each of the components in $R_{\mathtt{r}}$ and $W_{\mathtt{r}}$
- Extend the set of read-only semantic components of each reduction premise using the read-only semantic components of the conclusion (the second dependency rule ensures that there is one) as required by its dependencies.
- Thread the read-write semantic components through the premises, as follows:
  - Initialize the context of read-write semantic components Sc with the read-write semantic components of the left-hand side of the conclusion
  - For each reduction premise:
    - Extend the semantic components on the left-hand side as required by the dependencies of the arrow of the premise
    - Use fresh variables to extend the semantic components on the right-hand side
    - Update the context Sc by replacing the semantic components used in the left-hand side with the corresponding fresh variables used in the right-hand side.
  - Extend the semantic components on the right-hand side of the conclusion using the context returned by premises processing.

Coercion explication followed by the semantic components explication turns the `Plus` rule from Figure 6 in the rule of Figure 1. It also introduces the `Env` and `Sto` semantic components in the binder reduction rules $\xrightarrow{b}$ (through the implicit use of the $\longrightarrow$ arrow in `v : V`) leading to the explicated rules for binder lists and `Let` in Figure 7. Note that, unlike with usual inductive rules, due to the threading of semantics components, the order of the premises matters when using implicit semantics components.

4 Interpreter generation

Our objective is to compile DynSem specifications to efficient interpreters. Their performance should be acceptable and they must lend themselves to later optimizations. To achieve this we translate a DynSem specification into an interpreter in Java. ASTs of object language programs are mirrored to instantiations of the generated classes which yields executable ASTs. Programs in the object language are then directly executable after parsing. Executable ASTs are simpler to reason about than bytecode interpreters, since they preserve the program structure and since they lend themselves to local program optimizations. The runtime behavior of the generated executable ASTs resembles hand-written Java code which allows the JVM's JIT to recognize common patterns and optimize the running program.

In a nutshell, the generator maps signatures of the embedded language to classes in Java, arrow declarations to method stubs and general premisses to method bodies. Classes are generated into a class hierarchy dictated by the relation between sorts and between constructors and sorts. Reduction rule alternatives become callable in ancestor classes.

Signatures. A sort definition translates to an abstract Java class. Sorts related by a subtype relationship translate to Java classes in a subtype relation. A sort definition also derives a specialized Java implementation for a list of that specific sort.

$$\textbf{sort } S \quad \Longrightarrow \begin{cases} \textbf{abstract class } A_S \textbf{ extends } \mathit{AbstractNode} \texttt{ \{ ... \}} \\ \textbf{class } \mathit{List}_S \textbf{ implements List}\langle A_S\rangle \texttt{ \{ ... \}} \end{cases}$$

All generated list classes implement the $\textbf{List}\langle T\rangle$ interface and generated sort classes extend the framework-provided *AbstractNode* class.

A constructor C of arity n and sort S derives a Java class C_n which extends the class A_S. Classes derived from non-nullary constructors form roots for program subtrees. The generation scheme is

$$C : S_1 * ... * S_n \to S_t \quad \Longrightarrow \textbf{class } C_n \textbf{ extends } A_{S_t}\{A_{S_1}\ _1; ...; A_{S_n}\ _n;\}$$

An explicated arrow declaration for $\xrightarrow{rel}$ produces a method declaration in the class corresponding to the arrow's source sort. The return type R_{A_T} of the method is a record of the target type and the read-write semantic component types:

$$R_1 \cdot ... \cdot R_{n_r} \vdash (S :: S_1 \cdot ... \cdot S_{n_s}) \xrightarrow{rel} (T :: S_1 \cdot ... \cdot S_{n_s})$$
$$\Longrightarrow \begin{cases} \textbf{public } R_{A_T}\ rel(A_{R_1}, ..., A_{R_{n_r}}, A_{S_1}, ..., A_{S_{n_r}})\{\ def(S, rel)\ \} \text{ in } target(S) \\ \textbf{class } R_{A_T}\ \{A_T\ _0; A_{S_1}\ _1; ...; A_{S_{n_r}}\ _n;\} \end{cases}$$

The class containing the method is given by the conclusion's source pattern:

$$target(t) = \begin{cases} \textbf{class } C_n & \text{if } t = C(x_1, ..., x_n) \\ \textbf{class } \mathit{List}_S & \text{if } t : \textbf{List}\langle S\rangle \\ \textbf{abstract class } A_S & \text{if } t : S \text{, otherwise} \end{cases}$$

The generated method calls its ancestor or raises an exception if the containing class does not have an ancestor:

$$def(s, rel) = \begin{cases} \textbf{super}.rel(...) & \text{if } s = C(x_0, ..., x_n) \\ \textbf{super}.rel(...) & \text{if } s : T, T \neq \textbf{List}\langle T'\rangle \text{ and } \exists T'', T \leq T'' \\ \text{raise exception} & \text{otherwise} \end{cases}$$

term construction	$C_n(t_1, ..., t_n)$	$\Longrightarrow$ **new** $C_n(e_1, ..., e_n)$
list construction	$[t_1 \vert t_2]$ (: **List**$\langle T \rangle$)	$\Longrightarrow$ **new** $List_T(e_1, e_2)$
empty map	$\{\}$	$\Longrightarrow$ **new** $PersistentTreeMap()$
map extension	$\{t_1 \mapsto t_2, t_3\}$	$\Longrightarrow e_3.plus(e_1, e_2)$
map access	$t_1[t_2]$	$\Longrightarrow e_1.get(e_2)$
assignment	$t \Rightarrow v$ (: T)	$\Longrightarrow A_T\ v = e$
pattern match	$t \Rightarrow C(v_1, ..., v_n)$	$\Longrightarrow$ **if** (e **instanceof** C_n) { ... }
equality check	$t_1 = t_2$	$\Longrightarrow$ **if** ($e_1.equals(e_2)$) { ... }
relation	$t^*_{rs} \vdash t_s :: t_{sc} \xrightarrow{rel} v :: v_{tc}$ (: T)	$\Longrightarrow R_{A_T}\ _v = e_s.rel(e^*_{rs}, e^*_{sc})$

Figure 8 Correspondence of DynSem premises in Java.

Reduction rules. A pre-processing step merges all rules defining reductions for the same constructor (e.g. the two rules for `Ifz` in Figure 6), common premises are factorized and an *or* combinator ($\vee$) is introduced to combine the different alternatives. The merging algorithm is similar to left-factoring as described by Pettersson [19]. A difference is that premise equivalence in DynSem is decided modulo alpha-equivalence. Non-trivial pattern matching on the right-hand side of reduction premises is factored out using a pattern matching premise `t⇒p`. When `p` is not a variable, we apply the following transformation on the premise:

```
Rs ⊢ t :: Ws ⟶ p :: Ws'   ⟹   Rs ⊢ t :: Ws ⟶ x :: Ws', x ⇒ p
```

with `x` a fresh variable. We use similar rules for non-trivial patterns in `Ws'`. A merged reduction rule has its conclusion in normal form and a single general premise. It derives a method into the class described above. The signature of the generated method respects the signature derived from the arrow declaration. The general premise derives the method body. Figure 8 gives the correspondence of DynSem premises to Java statements and expressions. The individual premises are combined either as a sequence or as alternatives. Premises that can fail, such as pattern matches, are always guarded. If at evaluation a guard fails, its alternative premise is evaluated. A successful general premise returns from the method. For each reduction rule $rs \vdash s :: sc \xrightarrow{rel} t :: tc$ **where** gp the weaving of Java statements is given by the function $gen(gp \cdot\ sup(s, rel), t)$:

$$
\begin{aligned}
gen((p_1 \cdot p_2) \vee p_3, t) &:= \quad \textbf{if } gen_g(p_1) \textbf{ then } gen_s(p_1) \textbf{ ; } gen(p_2, t) \\
&\qquad\qquad\qquad\qquad\ \textbf{else } gen(p_3, t) \\
gen(p_1 \cdot p_2, t) &:= \quad gen(p_1, t) \textbf{ ; } gen(p_2, t) \\
gen(\epsilon, t) &:= \quad \textbf{return } t\textbf{;} \\
gen(sup(s, rel), t) &:= \quad def(s, rel)
\end{aligned}
$$

where $gen_g(p)$ and $gen_s(p)$ generate Java code according to Figure 8 for the guard and the successful case of premise p, respectively.

5 Related work

Definition of Standard ML. The implicit propagation of semantic components in DynSem is an implementation of the notation used in the *The Definition of Standard ML* [14] to define the dynamic semantics for a core of Standard ML. This core of Standard ML is defined, just like definitions in DynSem, in the style of natural semantics [10]. Although inspired by

I-MSOS [16], explication in DynSem is closer to the definition used by Milner et al. which defines an implicit order of premises and is applied to big-step style semantics. Following I-MSOS, rules in DynSem may also omit the unused read-only semantic components from rules, which are left implicit by Milner et al. *The Definition of Standard ML* uses a single reduction arrow symbol.

I-MSOS, MSOS & interpreter generation. Modular SOS (MSOS) [15] introduces relation arrows that have record values as arrow labels. This record value carries auxiliary entities to be propagated together in the label of the arrow. Labels can be composed but have to be explicitly mentioned on all uses of the relation arrow, regardless of whether the rule accesses the contents of the label or not.

I-MSOS [16], which is the inspiration for DynSem, improves on the modularity and conciseness of MSOS. It introduces the distinction between auxiliary entities (semantic components) that implicitly propagate either only downwards or are threaded through the premises of rules. Each I-MSOS specification has a translation to an MSOS specification by aggregating the semantic components on the arrow label. I-MSOS also supports multiple relation arrows, but in contrast to DynSem, using multiple arrows in the same rule propagates all of the semantic components to all of the arrows used together. As a special case of this difference DynSem also prevents propagation of auxiliary entities for ground terms.

I-MSOS and MSOS can derive specialized interpreters [1] in Prolog. In the naive generation strategy each reduction rule translates to a Prolog clause which calls a stepping predicate which searches for a next reduction in the program. The number of step inferences harms performance the interpreter. A refocusing strategy, following Danvy et al. [18], rewrites an MSOS specification to a specification in which each rule attempts to transitively completely evaluate intermediate values. This heuristic, speculating on locality of evaluation, significantly reduces the number of inferences and improves execution performance. Special rules are introduced to propagate abruptly terminated computations. Specialization of interpreters from MSOS involves left-factoring the rules as described by Pettersson [19]. Left-factoring is similar to the merging of rules in DynSem. While left-factoring eliminates only identical premises, merging in DynSem also eliminates mutually exclusive premises and premises which can be unified modulo alpha-renaming. The motivation for left-factoring is to reduce backtracking, merging additionally has the goal of grouping premises per constructor.

K Semantic Framework [20] is a mature language and toolchain for specification of dynamic semantics of programming languages. K has been applied to production-size languages such as C [7] and Java [4]. Semantics in K are given as rewrite rules. The homologue of semantic components in K are configurations. These consist of (nested) cells used to store the interpreted program and additional data structures. Configurations are automatically inherited into rules and rewriting takes place directly inside the cells of the configuration. Rules that do not mention the configuration automatically perform the rewrite inside the K (program) cell and resemble rewrite rules. Rules that explicitly mention the configuration resemble context-sensitive reduction rules. Cells in the configuration that are not accessed are left unchanged and implicitly propagated. Cells are only propagated from input to output since K rules do not have premises.

K requires that the syntax definition of the object language be embedded in the K specification. Annotations on the syntax definition can define the evaluation order or strictness requirements. For example a *strict*(1) annotation on the conjunction expression states that the left subexpression has to be evaluated first and completely. Once this syntax

is defined in K one can use a mix of object language concrete syntax and K syntax to match and build terms. Mixing of concrete object language syntax and DynSem can be obtained by assimilation following [21]. K specifications derive textual representations of the rules and graphical layouts for the configurations and cells to aid in documenting the object language. K specifications are compiled to rewriting logic in Maude [6]. Our previous micro-benchmarks [22] revealed that on big-step style specifications of the same language in DynSem and K, interpreters derived from DynSem are much faster than those derived from K.

PLT Redex [8] is an executable functional domain-specific language for semantic models. In Redex the semantics of a language is defined using context-sensitive reduction relations and meta-functions. Redex comes with a rich standard library of functions that can be used by semantic models. Its toolchain has facilities targeted at semantics prototyping such as randomized testing in the style of QuickCheck [5] and step-wise visualization of reductions [12]. Semantics can use either explicit substitution or environments and states. If environments and stores are used they have to be explicitly mentioned in every reduction relation. Redex supports ellipsis pattern placement for list matching and redex matching in program trees using the *in-hole* pattern, features that are not supported in DynSem. Readability of reduction relations in Redex is reduced by Racket's syntax [9] and by the need to explicitly mention semantic components. Programs are run by interpreting their reduction semantics in Redex. Racket's JIT compiler improves the runtime of interpreted programs but performance is still hampered by the redex lookup overhead and by non-deterministic rules. Performance of the interpreter is not an explicit goal of PLT Redex according to the published papers.

Implicit parameters in Scala exhibit similarities to semantic components in DynSem. Function parameters that are declared as implicit in the function definition may be omitted from the function application. Propagation of the implicit parameters is resolved by the compiler. The compiler prioritizes the inherited implicit parameters over the locally defined instances of the same type. Implicit parameters can be used to emulate the implicit read-only semantic components of DynSem but special care has to be taken to the mutability of the objects used. Persistent semantic components cannot be emulated with implicit parameters. While implicit parameters can be omitted from function applications they cannot be omitted from function definitions.

Monad transformers [13] allow definition of aggregated data structures with implicit packing and unpacking of components. The monad abstraction allows interpreters to be modularly composed. Examples are the state and IO monads. Monad transformers deliver implicit propagation similar to that of read-write semantic components in DynSem. The infectious propagation of the IO monad in Haskell resembles the upwards propagation of the semantic components in DynSem. In Haskell, like in the Scala case, the type signature of the function has to explicitly declare the monad as a function parameter. Both Haskell and Scala programs are less suitable for verification than more formal specification languages.

XSemantics [3] (the successor of Xtypes [2]) is a DSL for the specification of type systems for languages written in Xtext [23]. XSemantics is also applicable to implementation of interpreters. Dynamic semantics are specified in a syntax similar to deduction rules but with the rule conclusion preceding the rule premises. Multiple relation symbols can be declared and relation symbols can be overloaded. Only a single, mutable environment, can be used in a rule and there is no implicit propagation of the environment into the premises.

6 Future work

DynSem provides a good starting point for the further exploration of the integration of semantics engineering and language engineering. We plan to further develop the DynSem language and its accompanying toolset, and to investigate opportunities for abstraction in specifications. In particular, we plan to investigate the relation between static and dynamic binding of names in programs building on our foundational work on name resolution [17].

A goal for the generated interpreters is to obtain high performance execution engines. Techniques such as meta-tracing and dynamic compilation are proving successful for the optimization of custom built interpreters. We want to investigate whether such techniques can be generically applicable at the level of interpreter generation. Two other research avenues are to grow DynSem to cover a wide range of semantic styles and to automatically check that DynSem rules are type preserving.

Acknowledgments. We thank Peter Mosses and Martin Churchill for hosting the productive visit of Eelco Visser to Swansea that led to the DynSem language. We thank the participants of the Summer School on Language Frameworks in Sinaia, Romania in July 2012 — in particular Grigore Rosu, Robby Findler, and Peter Mosses — for inspiring us to address dynamic semantics in language workbenches. This research was partially funded by the NWO VICI *Language Designer's Workbench* project (639.023.206) and by a gift from the Oracle Corporation.

References

1. Casper Bach Poulsen and Peter D. Mosses. Generating specialized interpreters for modular structural operational semantics. In *Proceedings of the 23rd international symposium on Logic Based Program Synthesis and Transformation*, LOPSTR, 2013.
2. Lorenzo Bettini. A dsl for writing type systems for xtext languages. In Christian W. Probst and Christian Wimmer, editors, *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java, PPPJ 2011, Kongens Lyngby, Denmark, August 24-26, 2011*, pages 31–40. ACM, 2011.
3. Lorenzo Bettini. Implementing java-like languages in xtext with xsemantics. In Sung Y. Shin and José Carlos Maldonado, editors, *Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '13, Coimbra, Portugal, March 18-22, 2013*, pages 1559–1564. ACM, 2013.
4. Denis Bogdanas and Grigore Rosu. K-java: A complete semantics of java. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 445–456. ACM, 2015.
5. Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *ICFP*, pages 268–279, 2000.
6. Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott, editors. *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. Chucky Ellison and Grigore Rosu. An executable formal semantics of C with applications. In John Field and Michael Hicks, editors, *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 533–544. ACM, 2012.

8. Matthias Felleisen, Robby Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 2009.
9. Matthew Flatt. Plt. reference: Racket. Technical report, Technical Report PLT-TR-2010-1, PLT Inc., 2010.
10. Gilles Kahn. Natural semantics. In Franz-Josef Brandenburg, Guy Vidal-Naquet, and Martin Wirsing, editors, *STACS 87, 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany, February 19-21, 1987, Proceedings*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer, 1987.
11. Lennart C. L. Kats and Eelco Visser. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In William R. Cook, Siobhán Clarke, and Martin C. Rinard, editors, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*, pages 444–463, Reno/Tahoe, Nevada, 2010. ACM.
12. Casey Klein, John Clements, Christos Dimoulas, Carl Eastlund, Matthias Felleisen, Matthew Flatt, Jay A. McCarthy, Jon Rafkind, Sam Tobin-Hochstadt, and Robert Bruce Findler. Run your research: on the effectiveness of lightweight mechanization. In John Field and Michael Hicks, editors, *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 285–296. ACM, 2012.
13. Sheng Liang, Paul Hudak, and Mark P. Jones. Monad transformers and modular interpreters. In *POPL*, pages 333–343, 1995.
14. Robin Milner, Mads Tofte, and Robert Harper. *Definition of standard ML*. MIT Press, 1990.
15. Peter D. Mosses. Modular structural operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:195–228, 2004.
16. Peter D. Mosses and Mark J. New. Implicit propagation in structural operational semantics. *Electronic Notes in Theoretical Computer Science*, 229(4):49–66, 2009.
17. Pierre Neron, Andrew Tolmach, Eelco Visser, and Guido Wachsmuth. A theory of name resolution. In Jan Vitek, editor, *Programming Languages and Systems*, volume 9032 of *Lecture Notes in Computer Science*, pages 205–231. Springer Berlin Heidelberg, 2015.
18. Danvy Olivier, Nielsen, and Lasse R. *Refocusing in reduction semantics*. 2004.
19. Mikael Pettersson. *Compiling Natural Semantics*, volume 1549 of *Lecture Notes in Computer Science*. Springer, 1999.
20. Grigore Rosu and Traian-Florin Serbanuta. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.
21. Eelco Visser. Meta-programming with concrete object syntax. In Don S. Batory, Charles Consel, and Walid Taha, editors, *Generative Programming and Component Engineering, ACM SIGPLAN/SIGSOFT Conference, GPCE 2002, Pittsburgh, PA, USA, October 6-8, 2002, Proceedings*, volume 2487 of *Lecture Notes in Computer Science*, pages 299–315. Springer, 2002.
22. Eelco Visser, Guido Wachsmuth, Andrew P. Tolmach, Pierre Neron, Vlad A. Vergu, Augusto Passalaqua, and Gabriël D. P. Konat. A language designer's workbench: A one-stop-shop for implementation and verification of language designs. In Andrew P. Black, Shriram Krishnamurthi, Bernd Bruegge, and Joseph N. Ruskiewicz, editors, *Onward! 2014, Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, part of SLASH '14, Portland, OR, USA, October 20-24, 2014*, pages 95–111. ACM, 2014.
23. Xtext documentation. `http://www.eclipse.org/Xtext/documentation/2.6.0/Xtext`, 2014.