First International Workshop on Rewriting Techniques for Program Transformations and Evaluation

WPTE'14, July 13, 2014, Vienna, Austria

Edited by

Manfred Schmidt-Schauß
Masahiko Sakai
David Sabel
Yuki Chiba

Editors

Manfred Schmidt-Schauß
Institute for Informatics
Computer Science and Mathematics Department
Goethe-University Frankfurt am Main
schauss@ki.cs.uni-frankfurt.de

Masahiko Sakai
Graduate School of Information Science
Department of Computer Science
and Mathematical Informatics
Nagoya University
sakai@is.nagoya-u.ac.jp

David Sabel
Institute for Informatics
Computer Science and Mathematics Department
Goethe-University Frankfurt am Main
sabel@ki.cs.uni-frankfurt.de

Yuki Chiba
School of Information Science
Japan Advanced Institute of Science
and Technology
chiba@jaist.ac.jp

ACM Classification 1998
A.0 Conference proceedings, D.3.1 Formal Definitions and Theory, D.3.4 Translator writing systems and compiler generators, F.3.1 Specifying and Verifying and Reasoning about Programs, F.3.2 Semantics of Programming Languages, F.4.1 Mathematical Logic, F.4.2 Grammars and Other Rewriting Systems, I.2.2 Automatic Programming

ISBN 978-3-939897-70-5

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at http://www.dagstuhl.de/dagpub/978-3-939897-70-5.

Publication date
July, 2014

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

Digital Object Identifier: 10.4230/OASIcs.WPTE.2014.i

ISBN 978-3-939897-70-5 **ISSN 2190-6807** **http://www.dagstuhl.de/oasics**

OASIcs – OpenAccess Series in Informatics

OASIcs aims at a suitable publication venue to publish peer-reviewed collections of papers emerging from a scientific event. OASIcs volumes are published according to the principle of Open Access, i.e., they are available online and free of charge.

ISSN 2190-6807

www.dagstuhl.de/oasics

Contents

1st International Workshop on Rewriting Techniques for Program Transformations and Evaluation (WPTE'14).
Editors: Manfred Schmidt-Schauß, Masahiko Sakai, David Sabel, and Yuki Chiba
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Workshop Organization

FLoC 2014 Workshop Chair

Stefan Szeider	Vienna University of Technology, Austria

RTA/TLCA 2014 Workshop Chair

Aleksy Schubert	Warsaw University, Poland

WPTE 2014 Organizers

Manfred Schmidt-Schauß	Goethe-University Frankfurt am Main, Germany
Masahiko Sakai	Nagoya University, Japan
David Sabel	Goethe-University Frankfurt am Main, Germany
Yuki Chiba	Japan Advanced Institute of Science and Technology, Japan

WPTE 2014 Sponsor

Vereinigung von Freunden und Förderern
der Johann Wolfgang Goethe-Universität Frankfurt am Main e.V.

Program Chairs

Masahiko Sakai	Nagoya University, Japan
Manfred Schmidt-Schauß	Goethe-University Frankfurt am Main, Germany

Program Committee

Takahito Aoto	RIEC, Tohoku University, Japan
Yuki Chiba	Japan Advanced Institute of Science and Technology, Japan
Fer-Jan de Vries	University of Leicester, United Kingdom
Santiago Escobar	Universitat Politècnica de València, Spain
Maribel Fernández	King's College London, United Kingdom
Johan Jeuring	Open Universiteit Nederland & Universiteit Utrecht, The Netherlands
Delia Kesner	Université Paris-Diderot, France
Sergueï Lenglet	Université de Lorraine, France
Elena Machkasova	University of Minnesota, Morris, United States
Joachim Niehren	INRIA Lille, France
David Sabel	Goethe-University Frankfurt am Main, Germany
Masahiko Sakai	Nagoya University, Japan
Manfred Schmidt-Schauß	Goethe-University Frankfurt am Main, Germany
Eijiro Sumii	Tohoku University, Japan
Janis Voigtländer	University of Bonn, Germany
Harald Zankl	University of Innsbruck, Austria

External Reviewers

Tom Ridge
Haruhiko Sato

1st International Workshop on Rewriting Techniques for Program Transformations and Evaluation (WPTE'14).
Editors: Manfred Schmidt-Schauß, Masahiko Sakai, David Sabel, and Yuki Chiba
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Preface

This volume contains the papers presented at the *First International Workshop on Rewriting Techniques for Program Transformations and Evaluation (WPTE 2014)* which was held on July 13, 2014 in Vienna, Austria during the *Vienna Summer of Logic 2014 (VSL 2014)* as a workshop of the *Sixth Federated Logic Conference (FLoC 2014)*. WPTE 2014 was affiliated with the *25th International Conference on Rewriting Techniques and Applications joined with the 12th International Conference on Typed Lambda Calculi and Applications (RTA/TLCA 2014)*.

Scope of WPTE

Verification and validation of properties of programs, optimizing and compiling programs, and generating programs can benefit from the application of rewriting techniques. Source-level program transformations are used in compilation to simplify and optimize programs, in code refactoring to improve the design of programs; and in software verification and code validation, program transformations are used to translate and/or simplify programs into the forms suitable for specific verification purposes or tests. Those program transformations can be translations from one language into another one, transformations inside a single language, or the change of the evaluation strategy within the same language.

Since rewriting techniques are of great help for studying correctness of program transformations, translations and evaluation, the aim of WPTE is to bring together the researchers working on program transformations, evaluation, and operationally based programming language semantics, using rewriting methods, in order to share the techniques and recent developments and to exchange ideas to encourage further activation of research in this area.

Topics in the scope of WPTE include the correctness of program transformations, optimizations and translations; program transformations for proving termination, confluence and other properties; correctness of evaluation strategies; operational semantics of programs, operationally-based program equivalences such as contextual equivalences and bisimulations; cost-models for arguing about the optimizing power of transformations and the costs of evaluation; program transformations for verification and theorem proving purposes; translation, simulation, equivalence of programs with different formalisms, and evaluation strategies; program transformations for applying rewriting techniques to programs in specific programming languages; program inversions and program synthesis.

WPTE 2014

For WPTE 2014 six regular research papers were accepted out of the submissions. Additionally the program of WPTE contained the following talks on work in progress

- Yuki Chiba: *Verifying the Correctness of Tupling Transformations based on Conditional Rewriting*
- Guillaume Madelaine, Cédric Lhoussaine and Joachim Niehren: *Attractor Equivalence: An Observational Semantics for Reaction Networks*
- Georg Moser and Michael Schaper: *A Complexity Preserving Transformation from Jinja Bytecode to Rewrite Systems*

WPTE 2014 had two Program Chairs to allow submissions from the program committee and also from the chairs. Each submission was reviewed by at least three members of the

1st International Workshop on Rewriting Techniques for Program Transformations and Evaluation (WPTE'14).
Editors: Manfred Schmidt-Schauß, Masahiko Sakai, David Sabel, and Yuki Chiba
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Program Committee, with the help of two external reviewers. Reviewing of submissions with a program chair as coauthor was handled by the respective other program chair. This politics also permitted to cope with other conflicts of interest where a chair was involved.

Paper submission, reviewing, and the electronic meeting of the program committee used the great EasyChair system of Andrei Voronkov, which was also indispensable for preparing the WPTE program and collecting the papers for these proceedings. However, in its current state, the conflicts of interest between a chair and a submission cannot be properly dealt and are solved without EasyChair.

In addition to the contributed papers, the WPTE program contained an invited talk by Andrew Gill with title "*HERMIT: An Equational Reasoning Model to Implementation Rewrite System for Haskell*".

Acknowledgment

We thank the "Vereinigung von Freunden und Förderern der Johann Wolfgang Goethe Universität Frankfurt am Main e.V." for its financial support of WPTE 2014. Particularly, we thank Dustin Ortlepp for his advice during the application and the Dean of the Computer Science and Mathematics Department of the Goethe-University Thorsten Theobald for his support on our application.

Many people helped to make WPTE a successful event. We thank the organizers of VSL 2014: Florian Aigner, Matthias Baaz, Katinka Böhm, Agata Ciabattoni, Barbara Dolezal-Rainer, Thomas Eiter, Chris Fermüller, Sy David Friedman, Ursula Gerber, Georg Gottlob, Bernhard Gramlich, Franziska Gusel, Thomas Henzinger, Elisabeth Hofmann, Katarina Jurik, Jakob Kellner, Konstantin Korovin, Laura Kovács, Oliver Lehmann, Alexander Leitsch, Nysret Musliu, Thomas Pani, Anna Petukhova, Markus Pichlmair, Toni Pisjak, Norbert Preining, Vesna Sabljakovic-Fritz, Gernot Salzer, Matthias Schlögel, Martina Seidl, Stefan Szeider, Helmut Veith, Daniel Weller and the organizers of FLoC 2014 Matthias Baaz, Azadeh Farzan, Thomas Krennwallner, Moshe Y. Vardi, Helmut Veith, and especially the FLoC Workshop Chair Stefan Szeider and the FLoC Workshop Co-Chair Stefan Rümmele.

We thank the organizers of RTA/TLCA 2014 for hosting our workshop, we are particularly indebted to the RTA/TLCA Workshop Chair Aleksy Schubert for his help in applying for this workshop.

We thank our publisher Schloss Dagstuhl – Leibniz-Zentrum für Informatik for publishing our proceedings in the OpenAccess Series in Informatics (OASIcs). In particular we would like to thank Marc Herbstritt for his very helpful and always prompt support during production of the OASIcs proceedings.

Finally we thank the members of the program committee for their careful reviewing of all submissions and we thank the participants for their valuable contributions.

June 2014

Manfred Schmidt-Schauß
Masahiko Sakai
David Sabel
Yuki Chiba

The Collection of all Abstracts of the Talks at WPTE 2014

The aim of this chapter is to document all talks of the "First International Workshop on Rewriting Techniques for Program Transformations and Evaluation" (WPTE 2014). Hence, this collection contains all abstracts of talks held at WPTE 2014. The abstracts are ordered alphabetically by author names. Further information and e.g. extended abstracts on the talks on work in progress, can also be found on the handouts of the Vienna Summer of Logic 2014 (VSL 2014) which were distributed on USB flash drives to all participants of VSL 2014. For a majority of the contributions the full versions of the papers are available in these proceedings.

Verifying the Correctness of Tupling Transformations based on Conditional Rewriting

Author: Yuki Chiba

Abstract:

Chiba et al. (2010) proposed a framework of program transformation by templates based on term rewriting. Their framework can deal with tupling, which improves efficiency of programs. Outputs of their framework, however, may not always be more efficient than inputs. In this paper, we propose a technique to show the correctness of tupling based on conditional term rewriting. We give an extended equational logic in order to add conditional rules.

HERMIT: An Equational Reasoning Model to Implementation Rewrite System for Haskell

Author: Andrew Gill

Abstract:

HERMIT is a rewrite system for Haskell. Haskell, a pure functional programming language, is an ideal candidate for performing equational reasoning. Equational reasoning, replacing equals with equals, is a tunneling mechanism between different, but equivalent, programs. The ability to be agile in representation and implementation, but retain equivalence, brings many benefits. Post-hoc optimization is one obvious application of representation agility.

1st International Workshop on Rewriting Techniques for Program Transformations and Evaluation (WPTE'14).
Editors: Manfred Schmidt-Schauß, Masahiko Sakai, David Sabel, and Yuki Chiba
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Notes on Structure-Preserving Transformations of Conditional Term Rewrite Systems

Authors: Karl Gmeiner and Naoki Nishida

Abstract:

Transforming conditional term rewrite systems (CTRSs) into unconditional systems (TRSs) is a common approach to analyze properties of CTRSs via the simpler framework of unconditional rewriting. In the past many different transformations have been introduced for this purpose. One class of transformations, so-called unravelings, have been analyzed extensively in the past.

In this paper we provide an overview on another class of transformations that we call structure-preserving transformations. In these transformations the structure of the conditional rule, in particular their left-hand side is preserved in contrast to unravelings. We provide an overview of transformations of this type and define a new transformation that improves previous approaches.

Attractor Equivalence: An Observational Semantics for Reaction Networks

Authors: Guillaume Madelaine, Cédric Lhoussaine and Joachim Niehren

Abstract:

We study observational semantics for networks of chemical reactions as used in systems biology. Reaction networks without kinetic information, as we consider, can be identified with Petri nets. We present a new observational semantics for reaction networks that we call the attractor equivalence. The main idea of the attractor equivalence is to observe reachable attractors and reachability of an attractor divergence in all possible contexts. The attractor equivalence can support powerful simplifications for reaction networks as we illustrate at the example of the *Tet-On* system. Alternative semantics based on bisimulations or traces, in contrast, do not support all needed simplifications.

Verifying Optimizations for Concurrent Programs

Authors: William Mansky and Elsa L. Gunter

Abstract:

While program correctness for compiled languages depends fundamentally on compiler correctness, compiler optimizations are not usually formally verified due to the effort involved, particularly in the presence of concurrency. In this paper, we present a framework for stating and reasoning about compiler optimizations and transformations on programs in the presence of relaxed memory models. The core of the framework is the PTRANS specification language, in which program transformations are expressed as rewrites on control flow graphs with temporal logic side conditions. We demonstrate our technique by verifying the correctness of a redundant store elimination optimization in a simple LLVM-like intermediate language, relying on a theorem that allows us to lift single-thread simulation relations to simulations on multithreaded programs.

A Complexity Preserving Transformation from Jinja Bytecode to Rewrite Systems

Authors: Georg Moser and Michael Schaper

Abstract:

We revisit known transformations from object-oriented bytecode programs to rewrite systems from the viewpoint of runtime complexity. Suitably generalising the constructions proposed in the literature, we define an alternative representation of Jinja bytecode (JBC) executions as *computation graphs* from which we obtain a representation of JBC executions as *constrained rewrite systems*. We show that the transformation is *complexity preserving*. We restrict to non-recursive methods and make use of heap shape pre-analyses.

Inverse Unfold Problem and Its Heuristic Solving

Authors: Masanori Nagashima, Tomofumi Kato, Masahiko Sakai, and Naoki Nishida

Abstract:

Unfold/fold transformations have been widely studied in various programming paradigms and are used in program transformations, theorem proving, and so on. This paper, by using an example, show that restoring an one-step unfolding is not easy, i.e., a challenging task, since some rules used by unfolding may be lost. We formalize this problem by regarding one-step program transformation as a relation. Next we discuss some issues on a specific framework, called pure-constructor systems, which constitute a subclass of conditional term rewriting systems. We show that the inverse of T preserves rewrite relations if T preserves rewrite relations and the signature. We propose a heuristic procedure to solve the problem, and show its successful examples. We improve the procedure, and show examples for which the improvement takes effect.

On Proving Soundness of the Computationally Equivalent Transformation for Normal Conditional Term Rewriting Systems by Using Unravelings

Authors: Naoki Nishida, Makishi Yanagisawa, and Karl Gmeiner

Abstract:

In this paper, we show that the SR transformation, a computationally equivalent transformation proposed by Şerbănuţă and Roşu, is sound for weakly left-linear normal conditional term rewriting systems (CTRS). Here, soundness for a CTRS means that reduction of the transformed unconditional term rewriting system (TRS) creates no undesired reduction for the CTRS. We first show that every reduction sequence of the transformed TRS starting with a term corresponding to the one considered on the CTRS is simulated by the reduction of the TRS obtained by the simultaneous unraveling. Then, we use the fact that the unraveling is sound for weakly left-linear normal CTRSs.

Structural Rewriting in the π-Calculus

Author: David Sabel

Abstract:

We consider reduction in the synchronous π-calculus with replication, without sums. Usual definitions of reduction in the π-calculus use a closure w.r.t. structural congruence of processes. In this paper we operationalize structural congruence by providing a reduction relation for pi-processes which also performs necessary structural conversions explicitly by rewrite rules. As we show, a subset of structural congruence axioms is sufficient. We show that our rewrite strategy is equivalent to the usual strategy including structural congruence w.r.t. the observation of barbs and thus w.r.t. may- and should-testing equivalence in the pi-calculus.

Contextual Equivalences in Call-by-Need and Call-By-Name Polymorphically Typed Calculi (Preliminary Report)

Authors: Manfred Schmidt-Schauß and David Sabel

Abstract:

This paper presents a call-by-need polymorphically typed lambda-calculus with letrec, case, constructors and seq. The typing of the calculus is modelled in a system-F style. Contextual equivalence is used as semantics of expressions. We also define a call-by-name variant without letrec. We adapt several tools and criteria for recognizing correct program transformations to polymorphic typing, in particular an inductive applicative simulation.

List of Authors

Yuki Chiba
School of Information Science
Japan Advanced Institute of Science and Technology, Japan

Andrew Gill
Department of Electrical Engineering and Computer Science
The University of Kansas, USA

Karl Gmeiner
Institute of Computer Science
UAS Technikum Wien, Austria

Elsa Gunter
Department of Computer Science
University of Illinois at Urbana-Champaign, USA

Tomofumi Kato
Graduate School of Information Science
Nagoya University, Japan

Cédric Lhoussaine
BioComputing, Lifl /
University of Lille 1, France

Guillaume Madelaine
BioComputing, Lifl /
University of Lille 1, France

William Mansky
Department of Computer Science
University of Illinois at Urbana-Champaign, USA

Georg Moser
Institute of Computer Science
University of Innsbruck, Austria

Masanori Nagashima
Graduate School of Information Science
Nagoya University, Japan

Joachim Niehren
BioComputing, Lifl /
Inria Lille, France

Naoki Nishida
Graduate School of Information Science
Nagoya University, Japan

David Sabel
Computer Science and Mathematics Department
Goethe-University Frankfurt am Main, Germany

Masahiko Sakai
Graduate School of Information Science
Nagoya University, Japan

Michael Schaper
Institute of Computer Science
University of Innsbruck, Austria

Manfred Schmidt-Schauß
Computer Science and Mathematics Department
Goethe-University Frankfurt am Main, Germany

Makishi Yanagisawa
Graduate School of Information Science
Nagoya University, Japan

1st International Workshop on Rewriting Techniques for Program Transformations and Evaluation (WPTE'14).
Editors: Manfred Schmidt-Schauß, Masahiko Sakai, David Sabel, and Yuki Chiba
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

HERMIT: An Equational Reasoning Model to Implementation Rewrite System for Haskell*

Andrew Gill

Information Technology and Telecommunication Center
Department of Electrical Engineering and Computer Science
The University of Kansas, USA
andygill@ku.edu

Abstract

HERMIT is a rewrite system for Haskell. Haskell, a pure functional programming language, is an ideal candidate for performing equational reasoning. Equational reasoning, replacing equals with equals, is a tunneling mechanism between different, but equivalent, programs. The ability to be agile in representation and implementation, but retain equivalence, brings many benefits. Post-hoc optimization is one obvious application of representation agility.

What we want to explore is the mechanization of rewriting, inside real Haskell programs, enabling the prototyping of new optimizations, the explicit use of types to direct transformations, and perform larger data refinement tasks than are currently undertaken. Paper and pencil program transformations have been published that improve performance in a principled way; indeed some have turned the act of program transformation into an art form. But there is only so far a sheet of paper and a pencil can take you. There are also source code development environments that provide support for refactoring, such as HaRe. These work at the syntactical level, and Haskell is a large and complex language. What we want is mechanization, for examples that are currently undertaken by hand, and for examples that are challenging to perform using current development environments.

In this talk, we overview HERMIT, the Haskell equational reasoning model to implementation tunnel. HERMIT operates at the Glasgow Haskell compiler's Core level, deep inside GHC, where type information is easy to obtain, and the language being rewritten is smaller. HERMIT provides three levels of support for transformation and prototyping: a strategic programming base with many typed rewrite primitives, a simple shell that can used to interactively request rewrites and explore transformation possibilities, and a batch language that can mechanize focused, and optionally program specific, optimizations. We will demonstrate all three of these levels, and show how they cooperate.

The explicit aim of the HERMIT project is to explore the worker/wrapper transformation as a specific way of mechanizing data refinement. HERMIT has been successfully used on small examples, efficient `reverse`, tupling-`fib`, and many other examples from the literature. We will show two larger and more interesting examples of program transformation using HERMIT. Specifically, we will show the mechanization of the making a century program refinement pearl, originally by Richard Bird, and the exploration of datatype alternatives in Graham Hutton's implementation of John Conway's Game of Life.

1998 ACM Subject Classification D.3.4 Translator writing systems and compiler generators

Keywords and phrases Program Transformation, Equational Reasoning, Optimization

Digital Object Identifier 10.4230/OASIcs.WPTE.2014.1

Category Invited Talk

* This material is based upon work supported by the National Science Foundation under Grant No. 1117569.

1st International Workshop on Rewriting Techniques for Program Transformations and Evaluation (WPTE'14).
Editors: Manfred Schmidt-Schauß, Masahiko Sakai, David Sabel, and Yuki Chiba; pp. 1–1

OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Notes on Structure-Preserving Transformations of Conditional Term Rewrite Systems*

Karl Gmeiner[1] and Naoki Nishida[2]

1 **Institute of Computer Science, UAS Technikum Wien**
gmeiner@technikum-wien.at
2 **Graduate School of Information Science, Nagoya University**
nishida@is.nagoya-u.ac.jp

Abstract

Transforming conditional term rewrite systems (CTRSs) into unconditional systems (TRSs) is a common approach to analyze properties of CTRSs via the simpler framework of unconditional rewriting. In the past many different transformations have been introduced for this purpose. One class of transformations, so-called unravelings, have been analyzed extensively in the past.

In this paper we provide an overview on another class of transformations that we call structure-preserving transformations. In these transformations the structure of the conditional rule, in particular their left-hand side is preserved in contrast to unravelings. We provide an overview of transformations of this type and define a new transformation that improves previous approaches.

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

Keywords and phrases conditional term rewriting, unraveling, condition elimination

Digital Object Identifier 10.4230/OASIcs.WPTE.2014.3

Dedicated to the memory of Bernhard Gramlich

1 Introduction

Term rewriting is a widely accepted framework in computer science and has many applications. Conditional rewriting is an intuitive extension of term rewriting that appears naturally in applications like functional programming.

Conditional term rewrite systems (CTRSs) resemble unconditional term rewrite systems (TRSs), yet adding conditions to term rewriting has several drawbacks. From a theoretical point of view, many criteria that hold for unconditional rewriting do not hold for CTRSs, and many properties change their intuitive meaning. From a practical point of view conditional rewriting is complex to implement.

Hence, many transformations have been defined that eliminate the conditions of CTRSs and return unconditional TRSs (e.g. [2, 3, 8]). This way the well-understood framework of unconditional rewriting can be adapted for conditional rewriting, hence giving a better understanding on conditional rewriting and also from a practical point of view allowing us to simulate conditional rewrite sequences.

* The research in this paper is partly supported by the Austrian Science Fund (FWF) international project I963 and the Japan Society for the Promotion of Science (JSPS).

1st International Workshop on Rewriting Techniques for Program Transformations and Evaluation (WPTE'14).
Editors: Manfred Schmidt-Schauß, Masahiko Sakai, David Sabel, and Yuki Chiba; pp. 3–14

OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

We here provide an overview on a class of transformations that we refer to as structure-preserving and explain similarities and differences to a well-analyzed class of transformations, so-called unravelings. Definitions of structure-preserving derivations are usually complex compared to the ones of unravelings and they are usually only defined for CTRSs without extra variables. Therefore, we here also provide a definition of the transformation of [1] for CTRSs with (deterministic) extra variables. Since the transformation of [1] returns good results only for constructor CTRSs we also formally define a transformation of non-constructor CTRSs into constructor CTRSs. We will show that the combination of both transformation has better properties than other structure-preserving transformations. Proving further properties of this transformation will be part of our future work.

2 Preliminaries, Notions and Notations

We assume basic knowledge of conditional term rewriting and follow the basic notions and notations as they are defined in [13].

A conditional term rewrite system (CTRS) is a rewrite system that consists of conditional rules $l \to r \Leftarrow c$. The condition c is usually a conjunction of equations $s_1 = t_1, \ldots, s_k = t_k$.

There are different possible interpretations of equality in the conditions. CTRSs in which equality is interpreted as joinability $\downarrow$ are join CTRSs. Here we mainly consider oriented CTRSs, in which the conditions are interpreted as reducibility $\to^*$.

In contrast to unconditional TRSs, CTRSs may contain extra variables. We will consider CTRSs with extra variables that can be determined by rewrite steps (deterministic extra variables). CTRSs with only deterministic extra variables are called deterministic CTRSs (DCTRSs). Deterministic conditional rewrite rules $l \to r \Leftarrow s_1 \to^* t_1, \ldots, s_k \to^* t_k$ satisfy the condition $\mathcal{V}ar(s_i) \subseteq \mathcal{V}ar(l, t_1, t_{i-1})$ and $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l, t_1, \ldots, t_k)$.

A symbol $f \in \mathcal{F}$ in the signature of a CTRS $(R, \mathcal{F})$ is a defined symbol ($f \in \mathcal{D}$) if it is the root symbol of the left-hand side of a rule in R. All non-defined symbols are constructor symbols $\mathcal{C}$. A term is a constructor term if it only contains function symbols of $\mathcal{C}$ and variables. A CTRS is a constructor CTRS if the left-hand sides of all rules are of the shape $f(u_1, \ldots, u_n)$ where the arguments $u_1, \ldots, u_n$ are constructor terms.

There are several classes of CTRSs depending on the distribution of extra variables. A CTRS without extra variables is a 1-CTRS. If additionally the right-hand sides of the conditions are irreducible ground terms it is a normal 1-CTRS.

In some cases we will use the notation $\overrightarrow{X}$ where X is a set of terms. $\overrightarrow{X}$ represents the stream of variables in X in an unspecified but fixed order. Furthermore we will refer to rules with f as the root symbol on the left-hand side as f-rules.

3 Transformations of CTRSs

3.1 Overview

In [9] a class of transformations is introduced, so-called *unravelings*, and several properties are proved or disproved. There are some unravelings defined for some CTRSs (so-called normal 1-CTRSs and join 1-CTRSs). In [10] and [13] an unraveling is presented for deterministic CTRSs, a class of CTRSs that allows extra variables to a certain extend. This and similar unravelings have been analyzed extensively in the past (e.g. [11, 5, 12, 6]).

In [16] another transformation is presented that does not match the class of unravelings. This type of transformation is extended in [1, 14, 4].

We refer to these transformations as structure-preserving transformations because in contrast to unravelings these transformations do not encode the conditions in new function symbols but instead they encode them in the left-hand side of conditional rules. Hence, the original structure of terms is much better preserved. Such structure-preserving transformations have not been analyzed as much or consistently as unravelings.

In [15] a structure-preserving transformation is introduced for which "computational equivalence" is proven. In [4] a framework is introduced that allows the description of properties of unravelings and structure-preserving derivations consistently. Furthermore, some theoretical and practical drawbacks of the transformation of [15] are pointed out and another transformation is introduced that does not show these drawbacks, yet it is only applicable for a smaller class of CTRSs.

3.2 Transformations for DCTRSs

In transformations for DCTRSs conditions are eliminated by splitting a conditional rule into multiple unconditional rewrite rules in which the conditions are wrapped. The rule that introduces the first conditional argument is the *introduction rule*. After a condition has successfully been evaluated we switch to the next condition using a switch rule, or we eliminate the conditional argument using an elimination rule.

$$\mathbb{T}(l \to r \Leftarrow s_1 \to^* t_1, \ldots, s_k \to^* t_k) = \left\{ \begin{array}{rl} l \to C_1[s_1] & \text{introduction rule} \\ C_1[t_1] \to C_2[s_2] & \text{switch rules} \\ \vdots \quad \vdots & \text{switch rules} \\ C_k[t_k] \to r & \text{elimination rule} \end{array} \right\}$$

3.3 Unravelings

The class of unravelings that was introduced in [9] contains transformations that keep the original signature of the transformed system but add some new function symbols in which the conditions are wrapped while being evaluated. For every conditional rule a new function symbol is used.

▶ **Example 1.** Consider the following simple CTRS

$$\mathcal{R} = \left\{ \alpha : or(x,y) \to true \Leftarrow x \to^* true \qquad \beta : or(x,y) \to true \Leftarrow y \to^* true \right\}$$

Using the unraveling of [13] we obtain the following TRS

$$\mathbb{U}(\mathcal{R}) = \left\{ \begin{array}{ll} or(x,y) \to U_1^\alpha(x,x,y) & or(x,y) \to U_1^\beta(y,x,y) \\ U_1^\alpha(true,x,y) \to true & U_1^\beta(true,x,y) \to true \end{array} \right\}$$

In order to simulate the conditional rewrite sequence $or(true, false) \to_{\mathcal{R}}^* true$ we first apply the introduction rule of α and then the elimination rule:

$$or(true, false) \to_{\mathbb{U}(\mathcal{R})} U_1^\alpha(true, true, false) \to_{\mathbb{U}(\mathcal{R})} true$$

If we apply the introduction rule of β we obtain $or(true, false) \to_{\mathbb{U}(\mathcal{R})} U_1^\beta(false, true, false)$ where the latter term cannot be reduced any further.

Observe that in the previous example the unraveled TRS is not confluent although the original CTRS is confluent. If multiple conditional rules are applicable we must choose one introduction rule that should be applied. The other conditional rule cannot be applied then anymore. In [7] we introduced an unraveling that preserves confluence for some confluent CTRSs (including the CTRS of the previous example), yet for overlapping CTRSs, unravelings usually do not preserve confluence.

3.4 Structure-Preserving Transformations

Structure-preserving transformations stem from [16]. Further transformations of this class were introduced in [1], [15] and [4].

In structure-preserving transformations no new defined symbols are added by the transformation but instead additional conditional arguments are added to defined symbols but increases their arity in order to wrap the conditions. Hence, we need to replace function symbols in terms by the new function symbol. In order to translate terms from the original CTRS into the transformed TRS we will use an initialization mapping ϕ. Such a mapping is not needed in unravelings.

The additional argument in the defined function symbols contains the conditional argument. If the left-hand sides of multiple conditional rules are rooted by the same function symbol, the root symbol contains one conditional argument for each condition. An uninitialized conditional argument is marked by the constant $\bot$.

▶ **Example 2.** Consider the CTRS of Example 1:

$$\mathcal{R} = \left\{ \alpha : or(x,y) \to true \Leftarrow x \to^* true \qquad \beta : or(x,y) \to true \Leftarrow y \to^* true \right\}$$

The transformation of [1] ($\mathbb{S}$ in the following) increases the arity of *or* by two because we need one conditional argument for each conditional rule. If a term matches the left-hand side of a rule and the conditional argument is not initialized we can introduce the conditional argument. The other conditional argument is preserved so that both conditions can be evaluated in parallel.

$$\mathbb{S}(\mathcal{R}) = \left\{ \begin{array}{ll} or'(x,y,\bot,z) \to or'(x,y,x,z) & or'(x,y,z,\bot) \to or'(x,y,z,y) \\ or'(x,y,true,z) \to true & or'(x,y,z,true) \to true \end{array} \right\}$$

In order to simulate the conditional rewrite sequence $or(true, false) \to^*_{\mathcal{R}} true$ we obtain the correct normalform even if we apply the introduction rule of β first:

$$\begin{aligned} or'(true, false, \bot, \bot) &\to_{\mathbb{T}(\mathcal{R})} or'(true, false, \bot, false) \\ &\to_{\mathbb{T}(\mathcal{R})} or'(true, false, true, false) \to_{\mathbb{T}(\mathcal{R})} true \end{aligned}$$

The advantage of structure-preserving transformations compared to unravelings is that the conditions are encoded as decorators of the original terms. Hence, other rules remain applicable even if we evaluate conditions.

Another benefit of this approach is that we can exploit parallelism in reductions. Failed conditions do not block other derivations, in particular we can introduce other conditional rules. While in the unravelings of [9] and [13] we need to make an assumption which condition is satisfied already in the introduction step, we can postpone this decision in structure-preserving transformations until we evaluated all possible conditions.

One drawback of this type of derivations is their more complex definition. While unravelings have been defined for deterministic CTRSs, such a definition is only hinted in [15] and [4] for structure-preserving transformations.

4 Properties of Transformations

4.1 Soundness and Completeness

In order to prove properties of CTRSs by transforming them into unconditional TRSs we need to show that the rewrite relation of the original CTRS is properly approximated by the transformed TRS.

There are two main properties that we are interested in. *Completeness* means that a rewrite sequence in the original CTRS corresponds to a rewrite sequence in the transformed TRS. This property is usually satisfied and easy to prove.

The other direction, soundness, means that a rewrite sequence in the transformed system corresponds to a rewrite sequence in the original system. This property is more difficult to prove and usually not satisfied which has first been shown in [9]. In the past, unravelings have been proven to be sound for many syntactic properties and strategies ([5][12][6]).

4.2 Unsoundness of Structure-Preserving Transformations

In order to preserve confluence in transformations structure-preserving transformations encode conditions in parallel if multiple rules share the same root symbol on their left-hand side. Parallel evaluations of conditions of different rules allow us to postpone the decision which conditional rule will ultimately be applied in the transformed TRS. Yet, this also allows us to interleave multiple conditional rules by applying non-linear rules before applying an elimination step. This in fact causes unsoundness, even for constructor normal 1-CTRSs for which unravelings are known to be sound.

▶ **Example 3.** Consider the following overlay normal 1-CTRS

$$\mathcal{R} = \left\{ \begin{array}{cc} a \to c & f(x) \to C \Leftarrow x \to^* c \\ a \to d & f(x) \to D \Leftarrow x \to^* d \\ g(x,x) \to h(x,x) & \end{array} \right\}$$

f is the root symbol of the left-hand side of two conditional rules, hence we append to conditional arguments to f-terms in the rewrite system and insert the conditional arguments:

$$\mathbb{S}(\mathcal{R}) = \left\{ \begin{array}{cc} a \to c & f'(x,\bot,z) \to f'(x,x,z) \\ a \to d & f'(x,c,z) \to C \\ g(x,x) \to h(x,x) & f'(x,z,\bot) \to f'(x,z,x) \\ & f'(x,z,d) \to D \end{array} \right\}$$

In the original CTRS the term $g(f(a), f(b))$ rewrites to $h(C,C)$ and $h(D,D)$ but not to $h(C,D)$ because $f(a)$ and $f(b)$ do not have a common reduct that rewrites to both $f(a)$ and $f(b)$.

The term $g(f(a), f(b))$ corresponds to the term $g(f'(a,\bot,\bot), f'(b,\bot,\bot))$ in $\mathbb{T}(\mathcal{R})$. Observe the following derivation in the transformed TRS:

$$\begin{aligned} g(f'(a,\bot,\bot), f'(b,\bot,\bot)) &\to^* g(f'(a,a,a), f'(b,b,b)) \to^* g(f'(c,c,d), f'(c,c,d)) \\ &\to^* h(f'(c,c,d), f'(c,c,d)) \to h(C, f'(c,c,d)) \to h(C,D) \end{aligned}$$

Since this derivation is not possible in the original CTRS the transformation is unsound.

In the previous example the term $f'(c, c, d)$ contains two conditional arguments and both of them are satisfied. Therefore we can apply two elimination rules to this term. If we only encoded one conditional argument this would not be possible. In fact, the unravelings of [13] and also [11] are sound for this concrete example.

Therefore, soundness of unravelings do not imply soundness for structure-preserving derivations.

5 Structure-Preserving Transformations for Non-Constructor CTRSs

The structure-preserving transformation of [1] $\mathbb{S}$ is unsound for many non-constructor CTRS:

▶ **Example 4.** Consider the following CTRS from [1]:

$$\mathcal{R} = \{\ f(g(x)) \to x \Leftarrow x \to^* s(0) \qquad g(s(x)) \to g(x)\ \}$$

The CTRS is transformed into the following unconditional TRS:

$$\mathbb{S}(\mathcal{R}) = \left\{ \begin{array}{ll} f'(g(x), \bot) \to f'(g(x), x) & g(s(x)) \to g(x) \\ f'(g(x), s(0)) \to x & \end{array} \right\}$$

In $\mathcal{R}$, $f(g(s(0)))$ rewrites to $s(0)$ because the condition is satisfied. It also rewrites to $f(g(0))$ using the g-rule. The latter term is in normalform because the condition $0 \to^* s(0)$ is not satisfied.

In $\mathbb{S}(\mathcal{R})$, $f(g(s(0)))$ corresponds to the term $f'(g(s(0)), \bot)$. We obtain the following unsound derivation:

$$f'(g(s(0)), \bot) \to f'(g(s(0)), s(0)) \to f'(g(0), s(0)) \to 0$$

In the previous example we obtain unsoundness because both the redex and the reduct of the rewrite step $f(g(s(0))) \to f(g(0))$ match the left-hand side of the conditional rule, yet the variable bindings cannot be reduced to each other. In unravelings this does not cause soundness because the introduction step destroys the structure of the left-hand side of the conditional rule and only keeps the variable bindings. In structure-preserving transformations the structure is preserved and hence it can be modified.

5.1 Transformation $\mathbb{S}_{sr}$

In [15] a transformation is presented that extends the transformation $\mathbb{S}$ so that also overlapping CTRSs can be transformed appropriately. The transformation adds a complex unary operator that is propagated to outer positions and resets conditional arguments.

▶ **Example 5** (Transformation of [15])**.** The transformation of [15] extends the transformation of [1] by a unary function symbol $\{\ .\ \}$ that creates a layer around contracted redexes. The transformed TRS of the CTRS of Example 4 therefore contains the following rules:

$$\mathcal{R}'_1 = \left\{ \begin{array}{ll} f'(g(x), \bot) \to f'(g(x), \{x\}) & g(s(x)) \to \{g(x)\} \\ f'(g(x), \{s(0)\}) \to \{x\} & \end{array} \right\}$$

Now overlapping rewrite steps are blocked because of the new function symbol:

$$f'(g(s(0)), \bot) \to f'(g(s(0)), \{s(0)\}) \to f'(\{g(0)\}, \{s(0)\})$$

The new unary symbol is propagated to outer positions by adding one new rule for each argument of each function symbol. Such propagation steps reset conditional arguments and thereby avoid that outdated conditional arguments are used in elimination steps. Furthermore the new function symbol must be idempotent:

$$\mathcal{R}_2' = \left\{ \begin{array}{cc} f'(\{x\},z) \to \{f'(x,\bot)\} & g(\{x\}) \to \{g(x)\} \\ s(\{x\}) \to \{s(x)\} & \{\{x\}\} \to \{x\} \end{array} \right\}$$

The transformed TRS then is $\mathbb{S}_{sr}(\mathcal{R}) = \mathcal{R}_1 \cup \mathcal{R}_2$.

The term $f'(\{g(0)\},\{s(0)\})$ now can only be reduced by propagating the unary function symbol to the root position which resets the conditional argument:

$$f'(\{g(0)\},\{s(0)\}) \to \{f'(g(0),\bot)\} \to \{f'(g(0),\{0\})\}$$

The last term is irreducible.

5.2 Transformation $\mathbb{S}_{gg}$

In [4] it is pointed out that the transformation of [15] has some disadvantages. Apart from the complex definition of the new function symbol $\{\ .\ \}$ it is also non-preserving for many important syntactic properties like being non-overlapping, being a constructor system or being an overlay system.

From a practical point of view the transformation resets conditional arguments too often. The transformation of [4] tries to resolve these problems. Since the transformation of [4] is very complex in its definition we here provide a simpler refinement.

The main idea of the transformation of [4] is to add information to subterms of redexes to see whether an overlapping rewrite step was applied and the conditional argument should be reset. For this purpose we increase the arity of all defined function symbols (instead of just the root symbol) on the left-hand side of a conditional rule. While the root symbol encodes the condition, defined symbols strictly below the root contain a check argument. If they are uninitialized they contain $\bot$. After the introduction step these additional check arguments are marked with $\top$ to indicate that they were used in a conditional argument. In a rewrite step, all these check arguments are reset to $\bot$ to indicate that a conditional argument might be outdated. An elimination step is only allowed if all check arguments contain $\top$. For the introduction step it is sufficient if the conditional argument or one check argument is uninitialized. Therefore, one conditional rule might give rise to multiple introduction rules.

▶ **Example 6.** The left-hand side of the conditional rule of Example 4 contains the defined symbol g that therefore is replaces by a new binary symbol g' where the second argument is a check argument. If the conditional argument or the check argument is uninitialized we introduce the conditional argument.

$$\mathbb{S}_{\mathrm{gg}}(\mathcal{R}) = \left\{ \begin{array}{cc} f'(g'(x,z),\bot) \to f'(g'((x,\top)),\langle x\rangle) & f'(g'(x,\top),\langle s(0)\rangle) \to x \\ f'(g'(x,\bot),z) \to f'(g(\mathrm{'}(x,\top)),\langle x\rangle) & g'(s(x),z) \to g(x,\bot) \end{array} \right\}$$

Now, $f(g(s(0)))$ gives rise to the following derivation:

$$f'(g'(s(0),\bot),\bot) \to f'(g'(s(0),\top),\langle s(0)\rangle) \to f'(g'(0,\bot),\langle s(0)\rangle) \to f'(g'(0,\top),\langle 0\rangle)$$

It is not possible to reproduce the unsound derivation of Example 4.

In contrast to the transformation of [15] the transformation of [4] and also the refinement that is sketched here preserves many properties like being a constructor system (for normal 1-CTRSs), yet does not return satisfying results in all cases. For non-left-linear confluent CTRSs we might obtain non-confluence even if [15] returns a confluent CTRS. Furthermore, in collapsing CTRSs we still might obtain unsoundness (see [4, Example 8]) even though other transformations are sound.

6 New Transformation

The transformation of [1] is only applicable for constructor normal 1-CTRSs. The transformation of [15] allows the transformation also of non-constructor normal 1-CTRSs, yet it is syntactically complex. The transformation of [4] conservatively extends the transformation of [1], but its definition is complex and furthermore it is less powerful than the transformation of [15].

In our new approach we therefore modularize the transformational approach and use a transformation from non-constructor CTRSs into constructor CTRS before eliminating the conditions. This way we only need to consider constructor CTRSs in our new transformation.

First we define the transformation from non-constructor CTRSs into constructor CTRSs.

▶ **Definition 7** (transformation for non-constructor CTRSs)**.** Let $\alpha : l \to r \Leftarrow s_1 \to^* t_1, \ldots, s_k \to^* t_k$ be a conditional rule, then $cons$ is defined recursively as follows:

$$cons(\alpha) = \begin{cases} cons(\alpha') & \text{where } \alpha' = l[z]_p \to r \Leftarrow z \to^* l|_p, s_1 \to^* t_1, \ldots, s_k \to^* t_k, \\ & z \text{ is a fresh new variable } (z \notin \mathcal{V}ar(\alpha)) \text{ and} \\ & l|_p \ (p \in \mathcal{P}os(l) \setminus \{\epsilon\}) \text{ is not a constructor term} \\ \alpha & \text{if } l|_p \text{ is a constructor terms for all } p \in \mathcal{P}os(l) \setminus \{\epsilon\} \end{cases}$$

The mapping $cons$ is extended to CTRSs as follows: $cons(R) = \bigcup_{\alpha \in R} cons(\alpha)$.

▶ **Example 8.** Consider the CTRS of Example 4. The conditional rule $\alpha : f(g(x)) \to x \Leftarrow x \to^* s(0)$ is not a constructor rule because g is a defined symbol.

Using $cons$, the g-subterm is replaced by the fresh new variable z and a condition $z \to^* g(x)$ is added:

$$cons(\alpha) \quad = \quad f(z) \to x \Leftarrow z \to^* g(x), x \to^* s(0)$$

Observe that the rule α does not contain extra variables while $cons(\alpha)$ contains the deterministic extra variable z.

If the left-hand side of a conditional rule contains multiple non-constructor terms as subterms $cons$ does not imply any order of subterms. Our theoretical results do not depend on a specific order of positions, yet from a practical point of view choosing outer positions over inner positions first leads to less conditions.

▶ **Lemma 9.** *Let $\mathcal{R}$ be a deterministic CTRS. Then $cons(\mathcal{R})$ is a constructor DCTRS.*

Proof. Straightforward from the definition. ◀

▶ **Lemma 10** (Completeness)**.** *Let $\mathcal{R}$ be a DCTRS and $s, t \in \mathcal{T}$ be two terms. Then $s \to_{\mathcal{R}} t$ implies $s \to_{cons(\mathcal{R})} t$.*

Proof. In this case, the variable binding of the left-hand sides of the new conditions immediately matches the right-hand sides. ◀

▶ **Lemma 11** (Soundness). *Let $\mathcal{R}$ be a DCTRS and $s, t \in \mathcal{T}$ be two terms. Then if $s \to_{cons(\mathcal{R})} t$ then also $s \to^+_{\mathcal{R}} t$.*

Proof. We can extract the rewrite sequences in the new conditions and insert them in the replaced subterms. This is possible because all new variables are only used once. ◀

Next, we define the transformation for DCTRSs that conservatively extends the transformation of [1].

In order to transform a CTRS we group conditional rules by the root symbol of their left-hand side. We then transform these groups. In order to encode CTRSs with extra variables we sequentially encode all conditions. If a conditional argument matches the right-hand side of a condition, then we can apply a switch rule to evaluate the next condition. In this switch rule we do not keep the evaluated conditional argument to avoid derivations similar to the unsound derivation in Example 4 but instead encode variables that are needed on the right-hand side of the conditional rule or in one of the following conditions. This set of variables resembles the variables that are encoded in the optimized unraveling of [11], but in our case we only need to encode extra variables because we preserve the left-hand side of the conditional rule.

In order to further distinguish which condition is currently evaluated we also label the tuples that contain the conditional argument and the bindings of extra variables.

▶ **Definition 12** (structure-preserving transformation for sets of conditional rules). Let R_f be a set of conditional rules such that the left-hand sides of all rules are rooted by the same function symbol f with arity n.

Then, the mappings $\phi^{R_f}_X : \mathcal{T} \mapsto \mathcal{T}'$ and $\phi^{R_f}_\bot : \mathcal{T} \mapsto \mathcal{T}'$ are defined as follows:

$$\phi^{R_f}_\bot(u) = \begin{cases} f'(\phi^{R_f}_\bot(u_1), \ldots, \phi^{R_f}_\bot(u_n), \overbrace{\bot, \ldots, \bot}^{|R_f| \text{ times}}) & \text{if } u = f(u_1, \ldots, u_n) \\ g(\phi^{R_f}_\bot(u_1), \ldots, \phi^{R_f}_\bot(u_m)) & \text{if } u = g(u_1, \ldots, u_m) \\ u & \text{if } u \text{ is a variable} \end{cases}$$

$$\phi^{R_f}_X(u) = \begin{cases} f'(\phi^{R_f}_{X_1}(u_1), \ldots, \phi^{R_f}_{X_n}(u_n), z_1, \ldots, z_{|R_f|}) & \text{if } u = f(u_1, \ldots, u_n) \\ g(\phi^{R_f}_{Y_1}(u_1), \ldots, \phi^{R_f}_{Y_m}(u_m)) & \text{if } u = g(u_1, \ldots, u_m) \\ u & \text{if } u \text{ is a variable} \end{cases}$$

where $\{z_1, \ldots, z_{|R_f|}\} \subset X$, $X_1, \ldots, X_n$ are pairwise distinct subsets of $X \setminus \{z_1, \ldots, z_{|R_f|}\}$ and $Y_1, \ldots, Y_m$ are pairwise distinct subsets of X.

Let $i_\alpha \in \{1, \ldots, |R_f|\}$ be a unique index of the rule α in R_f. Then the transformed rules $\mathbb{S}_{\text{new}}(\alpha)$ of the rule $\alpha \in R_f$ are defined as follows:

$$\mathbb{S}_{\text{new}}(\alpha) = \left\{ \begin{array}{r} l'[\bot]_{n+i_\alpha} \to l'[\left\langle \phi^{R_f}_\bot(s_1), \overrightarrow{\mathcal{Z}_1} \right\rangle_1]_{n+i_\alpha} \\ l'[\left\langle \phi^{R_f}_{X_1}(t_1), \overrightarrow{\mathcal{Z}_1} \right\rangle_1]_{n+i_\alpha} \to l'[\left\langle \phi^{R_f}_\bot(s_2), \overrightarrow{\mathcal{Z}_2} \right\rangle_2]_{n+i_\alpha} \\ \vdots \quad \vdots \\ l'[\left\langle \phi^{R_f}_{X_k}(t_k), \overrightarrow{\mathcal{Z}_k} \right\rangle_k]_{n+i_\alpha} \to \phi^{R_f}_\bot(r) \end{array} \right\}$$

where $l' = \phi_X(l)$, $\mathcal{Z}_i = \bigcup \mathcal{V}ar(t_1, \ldots, t_{i-1}) \cap \mathcal{V}ar(t_i, s_{i+1}, \ldots, s_k, t_k, r) \setminus \mathcal{V}ar(l)$ is the set of extra variables that are still required for the rule application, and $X \cap \mathcal{V}ar(\alpha) = \emptyset$ is an infinite set of fresh variables.

The previous definition shows how to transform conditional rules and how to obtain mappings to map the signature of terms to the transformed system. Since we will obtain groups of unconditional rules with a different signature this way we need to provide a mapping to adjust the signature of other rules.

▶ **Definition 13** (adjusting signature)**.** Let R_f be a set of f-rooted conditional rules. Let $\beta : l \to r \Leftarrow s_1 \to^* t_1, \ldots, s_k \to^* t_k$ be a rule that is not f-rooted such that all subterms of l are constructor terms.

Then, $\phi_{R_f}(\beta)$ is defined as follows

$$\phi_{R_f}(\beta) = \phi^{R_f}_{X_0}(l) \to \phi^{R_f}_{\perp}(r) \Leftarrow \phi^{R_f}_{\perp}(s_1) \to^* \phi^{R_f}_{X_1}(t_1), \ldots \phi^{R_f}_{\perp}(s_1) \to^* \phi^{R_f}_{X_k}(t_k)$$

where $X_0, \ldots, X_k$ are infinite pairwise distinct sets of new variables ($\mathcal{V}ar(\beta) \cap \bigcup_{i=0}^{k} X_i = \emptyset$).

The final transformation itself groups rules by their root symbols and applies the transformation of Definition 12 to them. Then, the signature is adjusted for all rules according to the mappings ϕ_{R_f}.

▶ **Definition 14** (structure-preserving transformation for DCTRSs)**.** Let $\mathcal{R} = (R, \mathcal{F})$ be a constructor DCTRS such that $f_1, \ldots, f_n$ are all defined symbols and R_{f_i} contains all conditional f_i-rooted rules ($i \in \{1, \ldots, n\}$). Let furthermore R_{uc} be all unconditional rules in $\mathcal{R}$. Then the transformation $\mathbb{S}_{\text{new}}$ is defined as follows:

$$\mathbb{S}_{\text{new}}(\mathcal{R}) = \phi_{R_{f_1}}(\cdots \phi_{R_{f_n}}(R_{uc}) \cdots) \cup \bigcup_{i=1}^{n} \phi_{R_{f_1}}(\cdots \phi_{R_{f_{i-1}}}(\phi_{R_{f_{i+1}}}(\cdots \phi_{R_{f_n}}(\mathbb{S}_{\text{new}}(R_{f_i}) \cdots)$$

▶ **Example 15.** Consider the following CTRS of Example 4:

$$\mathcal{R} = \{\ f(g(x)) \to x \Leftarrow x \to^* 0 \qquad g(s(x)) \to g(x) \quad \}$$

In order to apply the transformation $\mathbb{T}_{\text{new}}$ we first must apply *cons* to transform $\mathcal{R}$ into a constructor CTRS.

$$\mathit{cons}(\mathcal{R}) = \{\ f(z) \to x \Leftarrow z \to^* g(x), x \to^* s(0) \qquad g(s(x)) \to g(x)\}$$

Next, we transform the conditional f-rule and the unconditional g-rule:

$$\mathbb{S}_{\text{new}}(R_f) = \{\ f'(z, \perp) \to f'(z, \langle z \rangle_1) \quad f'(z, \langle g(x) \rangle_1) \to f'(z, \langle x, x \rangle_2) \quad f'(z, \langle 0, x \rangle_2) \to x \quad \}$$

$$\phi_{R_f}(R_{uc}) = \{\ g(s(x)) \to g(x) \quad \}$$

Finally, we obtain $\mathbb{S}_{\text{new}}(\mathcal{R})$ by adjusting the signature in the transformed rules. Since the symbol g is preserved this is equivalent to the union of both subsystems.

$$\mathbb{S}_{\text{new}}(\mathcal{R}) = \left\{ \begin{array}{l} f'(z, \perp) \to f'(z, \langle z \rangle_1) \quad f'(z, \langle g(x) \rangle_1) \to f'(z, \langle x \rangle_2) \quad f'(z, \langle 0 \rangle_2) \to x \\ g(s(x)) \to g(x) \end{array} \right\}$$

This transformation does not require a resetting-mechanism like the transformation of [15]. Furthermore it preserves the property of being a constructor system if the rhs's of the conditions are constructor terms, and being non-overlapping if the rhs's of the conditions are non-overlapping with the lhs's of the rules. Compared to the transformation of [4] $\mathbb{S}_{\text{new}}$ does not cause non-confluence in connection with non-left-linear rules or unsoundness in connection with collapsing rules. Finally, our definition of $\mathbb{S}_{\text{new}}$ is the only formal definition known to us of a structure-preserving transformation for deterministic CTRSs.

Our next goals will be to prove soundness properties, in particular to compare our novel approach with recent properties of unravelings. Although it is known that structure-preserving transformations are unsound for certain non-erasing CTRSs while some unravelings are sound we hope to present some results in the near future that show a connection in soundness results.

7 Conclusion

We have presented an overview of transformations that preserve the term structure of left-hand sides of conditional rule. This class of transformations stems from [16]. We refer to these transformations as structure preserving transformations.

The transformation of [1] ($\mathbb{S}$) has nice properties for constructor normal 1-CTRSs. Yet for non-constructor CTRSs we obtain undesirable properties. Therefore, some other transformations were defined in the past that are based on this transformation but also return appropriate transformed TRSs for non-constructor CTRSs.

The extension of [15] ($\mathbb{S}_{\text{sr}}$) adds a complex resetting mechanism to $\mathbb{S}$ that has complex syntactic properties. The drawbacks of this transformation are discussed in [4] where also another transformation based on [1] is defined ($\mathbb{S}_{\text{gg}}$) that has better syntactic properties than $\mathbb{S}_{\text{sr}}$. Since the original definition is very complex we here sketched a simpler refined version.

The transformation $\mathbb{S}_{\text{gg}}$ also has undesirable properties for some collapsing CTRSs. Furthermore none of these transformations is formally defined for deterministic CTRSs. Therefore we here define a transformation for constructor DCTRSs that is similar to the one of [1] but it uses a sequential encoding of conditions similar to unravelings for DCTRSs.

In order to also transform non-constructor CTRSs using this CTRS we also define a transformation of non-constructor CTRSs into constructor DCTRSs. Combining these two transformations we obtain a new approach that shows promising first results compared to other structure preserving derivations.

In our future work we hope to provide more formal results and prove the usefulness of our approach in automated confluence tests of CTRSs.

Acknowledgements. We are deeply grateful to the anonymous referees for their useful comments.

References

1 Sergio Antoy, Bernd Brassel, and Michael Hanus. Conditional narrowing without conditions. In *Proc. 5th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, 27–29 August 2003, Uppsala, Sweden*, pages 20–31. ACM Press, 2003.

2 Jan A. Bergstra and Jan Willem Klop. Conditional rewrite rules: Confluence and termination. *Journal of Computer and System Sciences*, 32(3):323–362, 1986.

3 Elio Giovanetti and Corrado Moiso. Notes on the elimination of conditions. In Stéphane Kaplan and Jean-Pierre Jouannaud, editors, *Proc. 1st Int. Workshop on Conditional Rewriting Systems (CTRS'87), Orsay, France, 1987*, volume 308 of *Lecture Notes in Computer Science*, pages 91–97, Orsay, France, 1988. Springer. ISBN 3-540-19242-5.

4 Karl Gmeiner and Bernhard Gramlich. Transformations of conditional rewrite systems revisited. In Andrea Corradini and Ugo Montanari, editors, *Recent Trends in Algebraic Development Techniques (WADT 2008) – Selected Papers*, volume 5486 of *Lecture Notes in Computer Science*, pages 166–186. Springer, 2009.

5 Karl Gmeiner, Bernhard Gramlich, and Felix Schernhammer. On (un)soundness of unravelings. In Christopher Lynch, editor, *Proc. 21st International Conference on Rewriting Techniques and Applications (RTA 2010), July 11–13, 2010, Edinburgh, Scotland, UK*, LIPIcs (Leibniz International Proceedings in Informatics), July 2010.

6 Karl Gmeiner, Bernhard Gramlich, and Felix Schernhammer. On soundness conditions for unraveling deterministic conditional rewrite systems. In Ashish Tiwari, editor, *Proc. 23rd International Conference on Rewriting Techniques and Applications (RTA 2012), May 30 to June 2, 2012, Nagoya, Japan*, LIPIcs (Leibniz International Proceedings in Informatics), May/June 2012.

7 Karl Gmeiner, Naoki Nishida, and Bernhard Gramlich. Proving confluence of conditional term rewriting systems via unravelings. In Nao Hirokawa and Vincent van Oostrom, editors, *Proceedings of the 2nd International Workshop on Confluence*, pages 35–39, 2013.

8 Claus Hintermeier. How to transform canonical decreasing ctrss into equivalent canonical trss. In *Conditional and Typed Rewriting Systems, 4th International Workshop, CTRS-94, Jerusalem, Israel, July 13–15, 1994, Proceedings*, volume 968 of *Lecture Notes in Computer Science*, pages 186–205, 1995.

9 Massimo Marchiori. Unravelings and ultra-properties. In Michael Hanus and Mario Mario Rodríguez-Artalejo, editors, *Proc. 5th Int. Conf. on Algebraic and Logic Programming, Aachen*, volume 1139 of *Lecture Notes in Computer Science*, pages 107–121. Springer, September 1996.

10 Massimo Marchiori. On deterministic conditional rewriting. Technical Report MIT LCS CSG Memo n.405, MIT, Cambridge, MA, USA, October 1997.

11 Naoki Nishida, Masahiko Sakai, and Toshiki Sakabe. Partial inversion of constructor term rewriting systems. In Jürgen Giesl, editor, *Proc. 16th International Conference on Rewriting Techniques and Applications (RTA'05), Nara, Japan, April 19–21, 2005*, volume 3467 of *Lecture Notes in Computer Science*, pages 264–278. Springer, April 2005.

12 Naoki Nishida, Masahiko Sakai, and Toshiki Sakabe. Soundness of unravelings for deterministic conditional term rewriting systems via ultra-properties related to linearity. In Manfred Schmidt-Schauss, editor, *Proc. 22nd International Conference on Rewriting Techniques and Applications (RTA 2011), May 30 to June 1, 2011, Novi Sad, Serbia*, LIPIcs (Leibniz International Proceedings in Informatics), 2011. pages 267–282.

13 Enno Ohlebusch. *Advanced Topics in Term Rewriting*. Springer, 2002.

14 Grigore Rosu. From conditional to unconditional rewriting. In José Luiz Fiadeiro, Peter D. Mosses, and Fernando Orejas, editors, *WADT*, volume 3423 of *Lecture Notes in Computer Science*, pages 218–233. Springer, 2004.

15 Traian-Florin Şerbănuţă and Grigore Roşu. Computationally equivalent elimination of conditions. In Frank Pfenning, editor, *Proc. 17th International Conference on Rewriting Techniques and Applications, Seattle, WA, USA, August 12–14, 2006*, volume 4098 of *Lecture Notes in Computer Science*, pages 19–34. Springer, 2006.

16 Patrick Viry. Elimination of conditions. *J. Symb. Comput.*, 28(3):381–401, 1999.

Verifying Optimizations for Concurrent Programs*

William Mansky and Elsa L. Gunter

Department of Computer Science, University of Illinois at Urbana-Champaign, Thomas M. Siebel Center, 201 N. Goodwin, Urbana, IL 61801-2302, USA
{mansky1,egunter}@illinois.edu

Abstract

While program correctness for compiled languages depends fundamentally on compiler correctness, compiler optimizations are not usually formally verified due to the effort involved, particularly in the presence of concurrency. In this paper, we present a framework for stating and reasoning about compiler optimizations and transformations on programs in the presence of relaxed memory models. The core of the framework is the PTRANS specification language, in which program transformations are expressed as rewrites on control flow graphs with temporal logic side conditions. We demonstrate our technique by verifying the correctness of a redundant store elimination optimization in a simple LLVM-like intermediate language, relying on a theorem that allows us to lift single-thread simulation relations to simulations on multithreaded programs.

1998 ACM Subject Classification F.3.1 Specifying and Verifying and Reasoning about Programs

Keywords and phrases optimizing compilers, interactive theorem proving, program transformations, temporal logic, relaxed memory models

Digital Object Identifier 10.4230/OASIcs.WPTE.2014.15

1 Introduction

Program verification relies fundamentally on compiler correctness. Static analyses for safety or correctness in compiled languages depend implicitly on the fidelity of the compiler to some abstract semantics for the language, but real-world compilers rarely reflect these theoretical semantics [17]. The optimization phase of compilation is particularly error-prone: optimizations are often stated as complex algorithms on program code, with only informal justifications of correctness based on an intuitive understanding of program semantics. Formal methods researchers have devoted considerable effort to verifying these optimizations, either on a program-by-program basis (the translation validation approach [13]), or by general proof of correctness for all possible inputs (the approach taken, for instance, in the CompCert verified C compiler [7]). The problem is only aggravated in the presence of concurrency. Insufficiently analyzed optimizations may result in unreliable execution of concurrent code; compiler writers may even end up having to limit the scope and complexity of the optimizations they develop, in the absence of a method to demonstrate the safety of their optimizations.

In this paper, we present a new methodology for stating and verifying the correctness of compiler optimizations and transformations in the presence of concurrency, centered around a domain-specific language for specifying optimizations as transformations on program graphs

* This material is based upon work supported in part by NSF Grant CCF 13-18191. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

1st International Workshop on Rewriting Techniques for Program Transformations and Evaluation (WPTE'14).
Editors: Manfred Schmidt-Schauß, Masahiko Sakai, David Sabel, and Yuki Chiba; pp. 15–26

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

with temporal logic side conditions. This language, PTRANS, has been formalized in the Isabelle proof assistant [12], so that optimizations expressed in PTRANS can be proved correct with the assistance of state-of-the-art theorem-proving tools, as well as an executable semantics allowing specifications to serve as optimization prototypes. As a proof of concept, we use PTRANS to express and verify an optimization under several different concurrent memory models. Ultimately, we hope that the approach outlined in this paper will assist both formal verifiers and compiler writers in creating complex, concurrency-safe optimizations.

2 The PTRANS Specification Language

2.1 PTRANS: Adapting TRANS to Parallel Programs

The basic approach of the PTRANS specification language is that set out by Kalvala et al. in TRANS [4]: optimizations are specified as rewrites on program code in the form of control flow graphs (CFGs), with side conditions given in temporal logic. The syntax of PTRANS is given by the following grammar:

$$
\begin{array}{rcl}
A & ::= & \mathsf{add\_edge}(n, m, \ell) \mid \mathsf{remove\_edge}(n, m, \ell) \mid \mathsf{split\_edge}(n, m, \ell, p) \\
 & \mid & \mathsf{replace}\ n\ \mathsf{with}\ p_1, ..., p_m \\
\varphi & ::= & \mathsf{true} \mid p \mid \varphi \wedge \varphi \mid \neg\varphi \mid A\ \varphi\ \mathcal{U}\ \varphi \mid E\ \varphi\ \mathcal{U}\ \varphi \mid A\ \varphi\ \mathcal{B}\ \varphi \mid E\ \varphi\ \mathcal{B}\ \varphi \mid \exists x.\ \varphi \\
T & ::= & A_1, ..., A_m \text{ if } \varphi \mid \text{MATCH } \varphi \text{ IN } T \mid T \text{ THEN } T \mid T \,\square\, T \mid \text{APPLY\_ALL } T
\end{array}
$$

The atomic actions A include add_edge and remove_edge, which add and remove (ℓ-labeled) edges between the specified nodes; split_edge, which splits an edge between two nodes, inserting a new node between them; and replace, which replaces the instruction at a given node with a sequence of instructions, adding new nodes to contain the instructions if necessary. Kalvala et al. have shown that a wide range of common program transformations can be expressed using these basic rewrites. The arguments to the atomic actions represent nodes and instructions in the program graph, but may contain *metavariables* that are instantiated to program objects when the rewrites are applied.

At the top level, a transformation T is built out of conditional rewrites combined with *strategies*. The term $A_1, ..., A_m$ if φ is the basic pairing of one or more rewrites with a first-order CTL side condition, which may include the forward until-operator $\mathcal{U}$, its backward counterpart $\mathcal{B}$, and quantifiers over the metavariables appearing in its atomic predicates p. The expression MATCH φ IN T provides an additional side condition for a set of transformations, and also allows metavariables to be bound across multiple rewrites. The THEN and $\square$ operators provide sequencing and (nondeterministic) choice respectively, and APPLY_ALL T recursively applies T wherever possible until it is no longer applicable to the graph under consideration.

2.2 Concurrent Control Flow Graphs

The TRANS-style approach depends fundamentally on a notion of control flow graph (CFG). The atomic rewrites are rewrites on CFGs, and the CTL side conditions are evaluated on paths through CFGs. Thus, we require a concurrent analogue to the CFG in order to extend the approach to the concurrent setting. The particular model used here, adapted from the work of Krinke [5], is the threaded control flow graph (tCFG). In our framework, a tCFG is simply a collection of non-intersecting CFGs, one for each thread in a program. Formally:

▶ **Definition 1.** A *CFG* is a labeled directed graph $(N, E, \mathit{Start}, \mathit{Exit}, L)$ where N is a set of nodes, $E \subseteq N \times T \times N$ is a set of T-labeled edges (where T is given by the target language,

but must contain the label seq), $Start, Exit \in N$ are the distinguished Start and Exit nodes of the graph, and $L : N \to I$ assigns a program instruction to each node, such that: Start has no incoming edges, Exit has no outgoing edges, and the outgoing edges of each node except Exit correspond properly to the instruction label at that node, where the required correspondence is determined by the target language. A *tCFG* is a collection of disjoint CFGs, one for each thread in the program being represented. If $\mathcal{G}$ is a tCFG and t is a thread, we write $\mathcal{G}_t$ for the CFG of t in $\mathcal{G}$.

Paths through a tCFG can then be defined as sequences of vectors of program points, one per thread, and we can use CTL to state properties over tCFGs such as "no load occurs before the following store". The set of atomic predicates used in side conditions may depend on the target language under consideration; here we present some of the fairly general predicates used for our case study. These predicates break down into two types: those that depend on the state (i.e., map from threads to program points) in which they are evaluated, and those that do not (i.e., those that check some global property of the tCFG under consideration). State-based predicates include:

- $\mathsf{node}_t(n)$, which is true of a state q when $q_t = n$.
- $\mathsf{stmt}_t(i)$, which is true of a state q when the instruction at q is i in $\mathcal{G}_t$.
- $\mathsf{out}_t(ty, n')$, which is true of a state q when q_t has an outgoing edge to n' with label ty in $\mathcal{G}_t$.

State-independent predicates include:

- $\mathsf{conlit}(e)$, which is true when e represents a program constant.
- $\mathsf{varlit}(e)$, which is true when e represents a program variable (in our case study, we further distinguish between local (lvarlit) and global (gvarlit)).

Note that all of these predicates are purely syntactic static properties of tCFGs. This is not a coincidence: PTRANS optimizations can be stated and executed independently of the semantics of the target language, so that PTRANS may serve as a design tool even in the absence of formal semantics for the target language. Although we may quantify over paths in our side condition, these are paths through the *syntax* of a program as expressed in a tCFG, rather than dynamic executions of the program. Of course, when reasoning about the correctness of a transformation, we will need to relate these static properties to dynamic properties of program executions.

We also provide several extended predicates that allow the integration of outside analyses into CTL conditions. These predicates include:

- $\mathsf{cannot\_alias}_t(e, e')$, which is true of a state q when alias analysis can show that e and e' are not pointers to the same location in t at q.
- $\mathsf{in\_critical}_t(e, x)$, which is true of a state q when mutex analysis can show that q_t is part of a critical section for e protecting the value of x.
- $\mathsf{protected}_t(e, x)$, which is true when mutex analysis can show that the value of x is only changed in critical sections for e.

We incorporate these analyses by providing an axiomatization of the properties of a correct analysis (e.g., that if $\mathsf{cannot\_alias}_t(e, e')$ holds then e and e' do not point to the same location in any execution), and use these axioms to construct proofs of correctness for an optimization independently of the particular implementation of the analysis used to execute the optimization. The semantics of PTRANS actions and strategies can then be taken directly from our previous formalization of the TRANS system [11] to PTRANS. We have

also developed an execution engine for PTRANS in F#, using the Z3 SMT solver to find solutions to the side conditions, so that we can test optimizations on actual CFGs before engaging in the heavy-duty work of verification.

3 Concurrent Memory Models

In order to verify optimizations on a target language, we must first provide semantics for that language – but before that, we must define our notion of concurrency. Our approach is to give operational semantics to target languages over CFGs, and to parameterize those definitions by a concurrent memory model. A concurrent memory model provides an answer to the question, "what are the values that a memory *read* operation can read?" Almost every processor architecture has its own answer to this question, and many have more than one. Adding to the confusion, many of these models, including the one specified for LLVM [9], are not *operational*; they are phrased as conditions on total executions, rather than as properties that can be checked in individual steps of an operational semantics. As part of the development of PTRANS, we have developed a general approach to specifying operational concurrent memory models. Our memory models must support four functions:

- can_read, the workhorse of the memory model, which returns the set of values that a thread can see at a given memory location
- free_set, which returns the set of locations that are free in the memory
- start_mem, which gives a default initial memory
- update_mem, which updates a memory with a set of memory operations performed by various threads

We define three instances of this axiomatization for use in our example: sequential consistency (SC), total store ordering (TSO), and partial store ordering (PSO). Sequential consistency, the most intuitive memory model, requires that every execution observed could have been produced by some total order on the memory operations in the execution. Operationally, this can be modeled by requiring each read of a location to see the most recent write to that location. We implement SC with a map from memory locations to values and a straightforward implementation of the four required functions. The function can_read looks up its target in the memory map; free_set returns the set of locations with no values in the map; start_mem is the empty map; and update_mem applies the given memory operations to the map, storing a new value on a write or arw, initializing the location with a starting value on an alloc, and clearing the location on a free.

The TSO and PSO models are slightly more complex: they allow writes to be delayed past other instructions (reads of other locations in TSO; reads and writes to other locations in PSO), resulting in executions such as the one shown (in pseudocode) in Figure 1. Under SC, if one of the read instructions returned 0 in an execution, then we would be forced to conclude that the write instruction in the same thread executed before it, and so the other read could only read a value of 1. Under TSO, however, the writes may be delayed past the

Start: $\ell_1 \mapsto 0$ and $\ell_2 \mapsto 0$

write ℓ_1 1	write ℓ_2 1
$x :=$ read ℓ_2	$y :=$ read ℓ_1

Result: $x = 0 \wedge y = 0$

Figure 1 Behavior forbidden by SC but allowed in TSO.

reads, allowing both reads to return 0. As shown by Sindhu et al. [14], this behavior can be modeled by associating a FIFO *write buffer* with each thread (or, for PSO, a write buffer per memory location for each thread). When a write operation is performed, it is inserted into the executing thread's write buffer; at any point, the oldest write in any thread's write buffer may be written to the shared memory. A read operation first looks for the most recent write to the location in the thread's write buffer, and if none exists reads from the location in the shared memory. In this model, atomic arw operations serve as memory fences: they are not executed until the write buffer of the executing thread is cleared.

Some optimizations, particularly those that do not involve memory in any way, may be proved correct independently of the memory model. However, one of the purposes of relaxed memory models is to allow a wider range of optimizations, so we expect that most interesting optimizations will depend on the memory model being used. In general, some memory models are strictly more permissive than others – for instance, every execution produced under SC can also be produced under TSO – but depending on our notion of correctness, it may not follow that every valid SC optimization is also a valid TSO optimization, since an SC optimization may rely on the correctness of, e.g., a locking mechanism that only functions properly under SC.

4 MiniLLVM: A Sample Intermediate Language

In this section we present MiniLLVM, a language based on the LLVM intermediate language [9], for use as a target for transformation. The syntax of MiniLLVM is defined as follows:

$$\textit{expr} ::= \%\mathrm{x} \mid @\mathrm{x} \mid \mathrm{c} \qquad \textit{type} ::= \texttt{int} \mid \textit{type}^*$$

$$\begin{aligned}\textit{instr} ::= \;& \%\mathrm{x} = \mathrm{op}\ \textit{type}\ \textit{expr},\ \textit{expr} \mid \%\mathrm{x} = \texttt{icmp}\ \mathrm{cmp}\ \textit{type}\ \textit{expr},\ \textit{expr} \mid \texttt{br}\ \textit{expr} \mid \texttt{br} \mid \\ & \%\mathrm{x} = \texttt{call}\ \textit{type}\ (\textit{expr}, \ldots, \textit{expr}) \mid \texttt{return}\ \textit{expr} \mid \texttt{alloca}\ \%\mathrm{x}\ \textit{type} \mid \\ & \%\mathrm{x} = \texttt{load}\ \textit{type}^*\ \textit{expr} \mid \texttt{store}\ \textit{type}\ \textit{expr},\ \textit{type}^*\ \textit{expr} \mid \\ & \%\mathrm{x} = \texttt{cmpxchg}\ \textit{type}^*\ \textit{expr},\ \textit{type}\ \textit{expr},\ \textit{type}\ \textit{expr} \mid \texttt{is\_pointer}\ \textit{expr}\end{aligned}$$

(Note that the *'s indicate not repetition but pointer types.) Because the targets of control-flow instructions are implicit in the CFG, label arguments to `br` instructions and function names in `call` instructions are omitted. We give semantics to the language by specifying a labeled transition relation on program configurations. The single-thread semantics is given by the transition relation $G, t, m \vdash (p, env, st, al) \xrightarrow{a} (p', env', st', al')$ where G is the CFG representing the thread, t is the thread name, m is the shared memory, p is a program point, env is an environment giving values for thread-local variables, st is the call stack for the thread, al is a record of the memory locations allocated by the thread, and a is the set of memory operations performed by the thread. Memory operations are chosen from:

$$a ::= \quad \mathsf{read}\ t\ loc\ v \mid \mathsf{write}\ t\ loc\ v \mid \mathsf{arw}\ t\ loc\ v\ v \mid \mathsf{alloc}\ t\ loc \mid \mathsf{free}\ t\ loc$$

where arw represents an atomic read-and-write operation (as performed by the `cmpxchg` instruction). Several of the semantic rules for MiniLLVM instructions are shown in Figure 2. In the figure, Label $G\ p$ indicates the instruction label assigned to node p in the CFG G, and next $\ell\ p$ indicates the node reached along an outgoing ℓ-labeled edge from p.

A concurrent configuration is a vector of configurations, one for each thread, paired with a shared memory. The concurrent semantics of MiniLLVM is given by a single rule:

$$\frac{G_t, t, m \vdash states_t \xrightarrow{a} (p', env', st', al') \qquad \mathsf{update\_mem}\ m\ a\ m'}{(states, m) \to (states(t \mapsto (p', env', st', al')), m')}$$

$$\frac{\mathsf{Label}\ G\ p = (\%x = \mathrm{op}\ ty\ e_1, e_2) \qquad (e_1\ op\ e_2, env) \Downarrow v}{G, t, m \vdash (p, env, st, al) \to (\mathsf{next\ seq}\ p, env(x \mapsto v), st, al)}$$

$$\frac{\mathsf{Label}\ G\ p = (\mathtt{br}\ e) \qquad (e, env) \Downarrow v \qquad v \neq 0}{G, t, m \vdash (p, env, st, al) \to (\mathsf{next\ true}\ p, env, st, al)}$$

$$\frac{\mathsf{Label}\ G\ p = (\mathtt{alloca}\ \%x\ ty) \qquad loc \in \mathsf{free\_set}\ m}{G, t, m \vdash (p, env, st, al) \xrightarrow{\mathsf{alloc}\ t\ loc} (\mathsf{next\ seq}\ p, env(x \mapsto loc), st, al \cup \{loc\})}$$

$$\frac{\mathsf{Label}\ G\ p = (\mathtt{store}\ ty_1\ e_1, ty_2{}^*\ e_2) \qquad (e_1, env) \Downarrow v \qquad (e_2, env) \Downarrow loc}{G, t, m \vdash (p, env, st, al) \xrightarrow{\mathsf{write}\ t\ loc\ v} (\mathsf{next\ seq}\ p, env, st, al)}$$

Figure 2 Some single-thread transition rules for MiniLLVM.

In other words, we produce a concurrent step simply by selecting one thread to take a step, and then updating the memory with the memory operations performed by that thread.

5 Verification

5.1 Defining Correctness

Before we can begin verifying an optimization, we must clearly state what it means for an optimization to be correct. The semantics of a compiler transformation can be expressed denotationally in terms of the program graphs that may be produced as a result of the transformation on a given input graph. We can call a transformation T correct if, for any graph G, any graph G' output by applying T to G has some desired property relative to G. We will use *observational refinement* [3] as our sense of correctness; in other words, we will require that any observable behavior of G' is also an observable behavior of G, implying that T does not introduce any new behaviors. We will prove this refinement via *simulation* [2]:

▶ **Definition 2.** A *simulation* is a relation $\preceq$ on two labeled transition systems P and Q such that for any states p, p' of P and q of Q, for any label k, if $p \preceq q$ and $p \xrightarrow{k}_P p'$, then $\exists q'.\ q \xrightarrow{k}_Q q'$ and $p' \preceq q'$. By abuse of notation we write $P \preceq Q$ and say that Q simulates P.

The concurrent step relation of MiniLLVM as presented is unlabeled, but we can add labels to indicate the portion of the program's behavior that should be considered observable, which will generally be some portion of the shared memory. For each optimization to be verified, we will choose the maximum possible subset of shared memory as our observables, and state a simulation relation that relates any transformed graph to its original input. (Note that for more complex optimizations, more flexible relations such as weak (stuttering) simulation may be required, but the overall structure of the proof will remain unchanged.)

While PTRANS is expressive enough to allow optimizations that transform multiple threads simultaneously, many optimizations (especially concurrent retoolings of sequential optimizations) only transform a single thread. The following theorem allows us to extend a correct simulation relation on states in a single-thread CFG to one on entire tCFG states:

▶ **Definition 3.** Let the execution state of a multithreaded program with tCFG $\mathcal{G}$ be a pair $(states, m)$, where *states* is a vector of per-thread execution states and m is a shared memory.

The *lifting* of a simulation relation $\preceq$ on single-threaded CFGs to concurrent execution states relative to a thread t is defined by $(states, m)\ \lceil\preceq\rceil_t\ (states', m') \triangleq (states_t, m) \preceq (states'_t, m') \wedge \forall u \neq t.\ states_u = states'_u$.

▶ **Theorem 4.** *Fix a memory model supporting the functions* free_set, can_read, *and* update_mem. *Let $\mathcal{G}$ be a tCFG, t be a thread in $\mathcal{G}$, and obs be the set of observable memory locations. Suppose that $\preceq$ is a simulation relation such that $\mathcal{G}'_t \preceq \mathcal{G}_t$, $\mathcal{G}'_u = \mathcal{G}_u$ for all $u \neq t$, and for all $(s', m') \preceq (s, m)$ the following hold:*

1. free_set m = free_set m'
2. *For any $u \neq t$, if $u, \mathcal{G}_u, m' \vdash s_1 \xrightarrow{a} s_2$, then* can_read $m\ u\ \ell$ = can_read $m'\ u\ \ell$ *for every location ℓ mentioned in an operation in a*
3. *For any $u \neq t$, if $u, \mathcal{G}_u, m \vdash s_1 \xrightarrow{a} s_2$ and* update_mem $m'\ a\ m'_2$ *holds, then there exists some m_2 such that* update_mem $m\ a\ m_2$ *holds, $m_2|_{obs} = m'_2|_{obs}$, and $(s', m'_2) \preceq (s, m_2)$*

Then $\lceil\preceq\rceil_t$ is a simulation relation such that $\mathcal{G}'\ \lceil\preceq\rceil_t\ \mathcal{G}$.

While the exact conditions of the theorem are complicated, the intuition is straightforward: if $\preceq$ is a simulation relation for $\mathcal{G}_t$ and $\mathcal{G}'_t$ such that $(s', m') \preceq (s, m)$ implies that m and m' look the same to all threads $u \neq t$, and $\preceq$ is preserved by steps of threads other than t, then $\lceil\preceq\rceil_t$ is a simulation relation for $\mathcal{G}$ and $\mathcal{G}'$. This theorem allows us to break proofs of correctness for transformations on multithreaded programs into two parts: correctness of the simulation on the transformed thread, and validity of the relation with respect to the remaining threads. Note that in the case in which the simulation relation requires that $m = m'$, i.e., in which the optimization does not change the effects of $\mathcal{G}_t$ on shared memory, most of these conditions are trivial. In optimizations that affect the shared memory, on the other hand, the proof of the theorem's premises will involve some effort.

5.2 Specifying an Optimization

In the following sections, we will show the use of PTRANS in verifying an optimization. The candidate optimization is Redundant Store Elimination (RSE), which eliminates stores that are always overwritten before they are used, as in Figure 3. Note that s is replaced by an `is_pointer` instruction, rather than being eliminated entirely, to preserve failures: if e_2 is not pointer-valued at s the program will fail immediately, while eliminating s would allow the program to run until reaching s', potentially introducing new behavior.

In sequential code, the optimization is safe if, between the eliminated store s and the following store s', the location referred to by e_2 is not read and the value of e_2 is not changed. In the concurrent case, the correctness condition is more complex, since changes to a memory location can be observed by other threads. We will give a correct version of RSE for each of our three memory models. We begin with the rewrite portion of the transformation, which is the same in all cases, and the common portion of the side condition: the basic pattern that

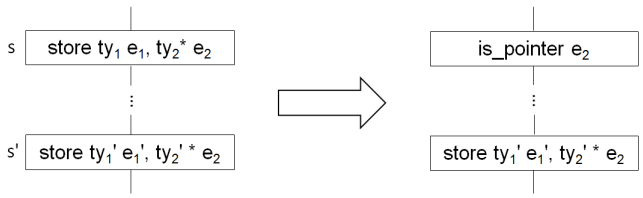


Figure 3 Redundant Store Elimination.

describes the node to be transformed, and a placeholder for the remaining conditions (note that the condition is checked starting at the entry node of the tCFG):

$$\begin{array}{l}\mathsf{replace}\ n\ \mathsf{with}\ \texttt{is\_pointer}\ e_2\ \text{if}\\ EF\ \mathsf{node}_t(n) \wedge \mathsf{stmt}_t(\texttt{store}\ ty_1\ e_1, ty_2^*\ e_2) \wedge \varphi\end{array}$$

Now, for each memory model, we need only provide a condition φ that ensures that the optimization is safe to perform. In general, this will be an "until"-property stating necessary conditions on the nodes between n and the next store to e_2.

Sequential consistency, the most restrictive of our three memory models, naturally has the most restrictive side condition. There are two approaches to securing the optimization: we could require that no memory operations occur between n and the following store, or we could require that e_2 be private to t. In this example we will take the second approach, using an external mutual exclusion analysis to ensure that e_2 is not exposed to other threads while t is in the region between n and the following store to e_2. Using the mutex predicates described in Section 2.2 and a defined $\mathsf{not\_touches}_t$ predicate that checks that a given memory location cannot be read or modified by t, the condition can be written as:

$$\begin{array}{l}\varphi_{SC} \triangleq \mathsf{protected}(x, e_2) \wedge \mathsf{gvarlit}(e_2) \wedge \neg\mathsf{is}(x, e_2) \wedge\\ \quad A\ \mathsf{in\_critical}_t(x, e_2) \wedge (\mathsf{node}_t(n) \vee \mathsf{not\_touches}_t(e_2))\\ \quad \mathcal{U}\ (\mathsf{in\_critical}_t(x, e_2) \wedge \neg\mathsf{node}_t(n)\ \wedge \exists ty_1', e_1', ty_2'.\ \mathsf{stmt}_t(\texttt{store}\ ty_1' e_1', ty_2'\ e_2))\end{array}$$

Next we will consider the appropriate side condition for the TSO memory model. Since TSO allows writes to be delayed past certain other operations, in a program with a redundant store, it is possible that the redundant store may be delayed until immediately before the following store to e_2. If this behavior is possible in the original program, then removing the store will not introduce new behavior. Thus, our side condition need only characterize the circumstances under which the store at n could have been delayed in the original program. In TSO, a write can be delayed past reads to different locations, but not past writes or atomic read-writes. Thus, the necessary side condition is as follows, where $\mathsf{not\_loads}$ checks that no `load` instructions read from a location and $\mathsf{not\_mods}$ ensured that the value of an expression is not changed:

$$\begin{array}{l}\varphi_{TSO} \triangleq AX_t(A\ \mathsf{not\_mods}_t(e_2) \wedge \mathsf{not\_loads}_t(e_2) \wedge\\ \quad \neg(\exists x, ty_1, e_1, ty_2, e_2', ty_3, e_3.\ \mathsf{stmt}_t(\texttt{store}\ ty_1\ e_1, ty_2^*\ e_2') \vee\\ \quad \mathsf{stmt}_t(\%x = \texttt{cmpxchg}\ ty_1^*\ e_1, ty_2\ e_2', ty_3\ e_3))\\ \quad \mathcal{U}\ (\neg\mathsf{node}_t(n)\ \wedge \exists ty_1', e_1', ty_2'.\ \mathsf{stmt}_t(\texttt{store}\ ty_1'\ e_1', ty_2'\ e_2)))\end{array}$$

where AX_t is a derived temporal operator defined such that $AX_t\varphi$ iff φ is true in every state in which the thread t has advanced by one node (regardless of the behavior of other threads). The fragment of the condition inside the AX_t operator provides a useful characterization of the nodes between n and the following store to e_2; we will call it φ'_{TSO}, where $\varphi_{TSO} = AX_t\ \varphi'_{TSO}$. Note that φ_{SC} is also a reasonable side condition under TSO, and we could form a more general optimization by using $\varphi_{SC} \vee \varphi_{TSO}$ as our side condition.

The relaxation of the PSO memory model is a more permissive version of that of TSO, so we can obtain a side condition for it by relaxing the constraints of φ_{TSO}. A write in PSO can be delayed past reads and writes to different locations, but not past operations on the same location or atomic read-writes, so the corresponding side condition is:

$$\begin{array}{l}\varphi_{PSO} \triangleq AX_t(A\ \mathsf{not\_mods}_t(e_2) \wedge \mathsf{not\_touches}_t(e_2) \wedge\\ \quad \neg(\exists x, ty_1, e_1, ty_2, e_2', ty_3, e_3.\ \mathsf{stmt}_t(\%x = \texttt{cmpxchg}\ ty_1^*\ e_1, ty_2\ e_2', ty_3\ e_3))\\ \quad \mathcal{U}\ (\neg\mathsf{node}_t(n)\ \wedge \exists ty_1', e_1', ty_2'.\ \mathsf{stmt}_t(\texttt{store}\ ty_1' e_1', ty_2'\ e_2)))\end{array}$$

This condition is strictly weaker than φ_{TSO}, allowing the optimization to be applied to a wider range of programs. As above, we also define φ'_{PSO} such that $\varphi_{PSO} = AX_t\ \varphi'_{PSO}$ for use in our proofs of correctness.

5.3 Verification of RSE

We are now ready to demonstrate the correctness of MiniLLVM RSE in PTRANS. As laid out in Section 5.1, we prove correctness by showing that for any transformed tCFG $\mathcal{G}'$ produced by applying the optimization to a graph $\mathcal{G}$, there exists a simulation relation $\preceq$ such that $\mathcal{G}'_t \preceq \mathcal{G}_t$, states related by $\preceq$ make the same values visible to threads other than t, and steps by threads other than t preserve $\preceq$. For each version of RSE, we will present such a relation and sketch the proof of its correctness.

▶ **Theorem 5.** *Let $\mathcal{G}'$ be a tCFG in the output of RSE(φ_{SC}) on a tCFG $\mathcal{G}$, and ℓ be the location targeted by the redundant store removed in $\mathcal{G}'$. Let $\preceq_{SC}$ be the relation such that $(s', m') \preceq_{SC} (s, m)$ iff*

- *$s = s'$*
- *either $\ell \in \mathsf{free\_set}\ m$ and $\ell \in \mathsf{free\_set}\ m'$, or $\ell \notin \mathsf{free\_set}\ m$ and $\ell \notin \mathsf{free\_set}\ m'$*
- *either $m = m'$, or else φ_{SC} holds at the program point of s in $\mathcal{G}$ and $m|_{\overline{\ell}} = m'|_{\overline{\ell}}$.*

Then $\lceil \preceq_{SC} \rceil_t$ is a simulation relation such that $\mathcal{G}'\ \lceil \preceq_{SC} \rceil_t\ \mathcal{G}$ with all locations other than ℓ observable.

Proof. Consider two related states (s, m) of $\mathcal{G}_t$ and (s', m') of $\mathcal{G}'_t$. In case (1), the only interesting case is the one in which s is at the transformed node n; in this case, $\mathcal{G}'_t$ executes the `is_pointer` instruction and $\mathcal{G}_t$ executes the `store` instruction. Since the side condition of the RSE transformation is true on $\mathcal{G}$, we know that φ_{SC} holds at n, and so $\preceq_{SC}$ holds on the resulting states. If, on the other hand, we are in case (2), then we know that φ_{SC} holds, so s must be in the region between n and the next store to e_2. If we have not yet reached the next store to e_2, then since $\preceq_{SC}$ holds we know that it does not read or modify the memory at ℓ, and we can conclude that $\mathcal{G}_t$ and $\mathcal{G}'_t$ execute the same instruction and arrive in new configurations (s_2, m_2) and (s'_2, m'_2) such that m_2 and m'_2 differ only at ℓ and φ_{SC} still holds. The guarantees of mutual exclusion ensure the separation of threads required by Theorem 4, and we can conclude that $\lceil \preceq_{SC} \rceil_t$ is a simulation relation showing the correctness of the SC version of RSE. ◀

Recall that, while in SC the memory is simply a map m from locations to values, in TSO and PSO it is a pair (m, b) of a shared memory and per-thread write buffers. Since the correctness of our conditions under these models depends on our ability to delay stores until they become redundant, we must have a notion of one buffer being a "redundant expansion" of another.

▶ **Definition 6.** A *write buffer* is a queue of writes expressed as location-value pairs. A write buffer b' is a *redundant expansion* of b if b' can be constructed from b by adding, in front of each pair (ℓ, v) in b, zero or more writes of other values to ℓ. We will say that a collection of write buffers c' is a redundant expansion of a collection c when each write buffer c'_t is a redundant expansion of the corresponding write buffer c_t.

Because the added writes appear immediately in front of other writes to the same location, they can be immediately overwritten when the buffers are cleared, and are never read when looking for the latest write to a location. This allows a redundant expansion of b to simulate the behavior of b with regard to the memory-model functions.

▶ **Theorem 7.** *Let $\mathcal{G}'$ be a tCFG in the output of RSE(φ_{TSO}) on a tCFG $\mathcal{G}$. Let $\preceq_{TSO}$ be the relation such that $(s', (m', b')) \preceq_{TSO} (s, (m, b))$ iff*

- *$s = s'$, $m = m'$, and $b_u = b'_u$ for all $u \neq t$, and*
- *either (1) b_t is a redundant expansion of b'_t, or else (2) ${\varphi'}_{TSO}$ holds at the program point of s in $\mathcal{G}$, the store eliminated in $\mathcal{G}'$ was to some expression e_2, and there is a location ℓ such that e_2 evaluates to ℓ in s, the last write in b_t is a write to ℓ, and the rest of b_t is a redundant expansion of b'_t.*

Then $\lceil \preceq_{TSO} \rceil_t$ is a simulation relation such that $\mathcal{G}' \lceil \preceq_{TSO} \rceil_t \mathcal{G}$ with all locations observable.

Proof. By Theorem 4. Consider two related states $(s, (m, b))$ of $\mathcal{G}_t$ and $(s', (m', b'))$ of $\mathcal{G}'_t$. If b_t is a redundant expansion of b'_t (case 1), then the only interesting case is the one in which s is at the transformed node n; in this case, $\mathcal{G}'_t$ executes the `is_pointer` instruction, and $\mathcal{G}_t$ executes the `store` instruction, evaluating e_2 to some location ℓ and adding a write to ℓ to its buffer – thus the resulting buffer has the structure described in case (2). Since the side condition of the RSE transformation is true on $\mathcal{G}$, we know that $\varphi_{TSO} = AX_t\ {\varphi'}_{TSO}$ holds at n, and so $\preceq_{TSO}$ holds on the resulting states. If, on the other hand, we are in case (2), s must be in the region between n and the next store to e_2. If s is at a store to e_2 other than n, then both $\mathcal{G}$ and $\mathcal{G}'$ commit a write to ℓ; since b_t was a redundant expansion of b'_t followed by a write to ℓ, this new write makes the last one redundant, and we are now in case (1). If s is somewhere between n and the following store, then since ${\varphi'}_{TSO}$ holds we know that the current instruction does not read the memory at ℓ and is neither a `store` nor a `cmpxchg`, so we can conclude that $\mathcal{G}_t$ and $\mathcal{G}'_t$ execute the same instruction with the same result, that the instruction adds no new writes to t's write buffer, and that the extra write to ℓ in b_t is not forced into main memory (as it would be by a `cmpxchg` instruction). Thus, case (2) of $\preceq_{TSO}$ still holds. Since the only difference in states allowed by $\preceq_{TSO}$ is in the write buffer for t, which is neither visible to nor affected by threads other than t, the separation of threads required by Theorem 4 holds, and we can conclude that $\lceil \preceq_{TSO} \rceil_t$ is a simulation relation showing the correctness of the TSO version of RSE. ◀

▶ **Theorem 8.** *Let $\mathcal{G}'$ be a tCFG in the output of RSE(φ_{PSO}) on a tCFG $\mathcal{G}$. Let $\preceq_{PSO}$ be the relation such that $(s', (m', b')) \preceq_{PSO} (s, (m, b))$ iff*

- *$s = s'$, $m = m'$, $b_{u,\ell} = b'_{u,\ell}$ for all ℓ and all $u \neq t$, and*
- *either (1) $b_{t,\ell}$ is a redundant expansion of $b'_{t,\ell}$ for all ℓ, or else (2) ${\varphi'}_{PSO}$ holds at the program point of s in $\mathcal{G}$, the store eliminated in $\mathcal{G}'$ was to some expression e_2, and there is a location ℓ such that e_2 evaluates to ℓ in s, $b_{t,\ell}$ is a redundant expansion of $b'_{t,\ell}$ followed by a write to ℓ, and $b_{t,\ell'}$ is a redundant expansion of $b'_{t,\ell'}$ for all other locations ℓ'.*

Then $\lceil \preceq_{PSO} \rceil_t$ is a simulation relation such that $\mathcal{G}' \lceil \preceq_{PSO} \rceil_t \mathcal{G}$ with all locations observable.

Proof. By Theorem 4. The proof is nearly identical to that of the TSO case. Since write buffers are per-location, `store` instructions to locations other than ℓ may be executed between the eliminated store and the following write to ℓ without changing the relationship between $b_{t,\ell}$ and $b'_{t,\ell}$, justifying the weaker side condition; otherwise, the proof proceeds entirely analogously. ◀

In this manner, PTRANS allows us to express and verify optimizations under a variety of memory models, sharing information between specifications and proofs of similar transformations. All of the above proofs have been carried out in full formal detail in the Isabelle proof assistant, and can be found online at `http://web.engr.illinois.edu/~mansky1/ptrans`.

6 Conclusions and Related Work

In this paper we present the PTRANS specification language, in which optimizations are expressed as conditional rewrites on program syntax, and show how it can be used to state and verify compiler optimizations. We outline a method for stating and verifying optimizations that transform a single thread in a multithreaded program, with some parts independent of and others dependent on the memory model under consideration. We use this method to verify a redundant store elimination optimization on an LLVM-based language under three memory models, showing that the behaviors of every output program are possible behaviors of the input program. In combination with the executable semantics for PTRANS, which allows PTRANS specifications to serve as prototype optimizations [10], the methodology here presented forms the basis of a new framework for specifying, testing, and verifying compiler optimizations in the presence of concurrency.

Our work builds on the TRANS approach due to Kalvala et al. [4]. Among the tools that build on this approach is the Cobalt specification system [6], which aims to automatically prove the correctness of optimizations. This automation comes at the cost of expressiveness: Cobalt is limited to a much smaller set of CTL side conditions than TRANS (or PTRANS) in general. While interactive proofs require considerably more effort, using a standard framework for proofs across different memory models and target languages can reduce the burden by allowing common facts (about simulation, CTL over CFGs, etc.) to be proved once and for all. To the best of our knowledge, neither Cobalt nor any other TRANS-style work has yet addressed the problem of concurrency.

The most comprehensive compiler correctness effort to date is CompCertTSO [16], the extension of CompCert [7] to the TSO memory model. CompCertTSO includes a range of verified optimizations on intermediate languages at various levels, as well as verified translations between languages, while we have thus far only verified same-language transformations. Our approach has the advantage of language- and memory-model independence; our framework also allows us to separate out the correctness condition for a concurrent optimization into a simulation relation on a single thread and side conditions on the remaining threads, while the one concurrency-aware optimization verified in CompCertTSO involves a whole-program simulation proof. Ševčík [15] has also verified various optimizations, including redundant instruction eliminations, in a language-independent manner under data-race-free sequential consistency, specifying optimizations directly as transformations on the executions traces of programs (which may not directly correspond to modification of program code).

Burckhardt et al. [1] have developed a method of verifying optimizations under relaxed memory models by specifying memory models as sets of rewrite rules on program traces and optimizations as rewrites on local fragments of a program. Their proofs are fully automatic, using the Z3 SMT solver to check that all traces allowed by a transformed program fragment could be produced by applying the rewrite rules allowed by the memory model to the traces of the original program. They rely on a denotational semantics for their target language that gives the set of possible program traces for every program, and thus far have only verified transformations on single instructions or pairs of immediately adjacent instructions (including a simple RSE); their method does not obviously extend to transformations that require analysis over fragments of the program graph of indefinite size (e.g., all the instructions between one instruction and another).

Thus far, we have only verified optimizations that transform one thread at a time, assisted by a theorem that allows us to lift single-thread simulation relations to simulations on multithreaded CFGs. If we expand our scope to optimizations that transform multiple

threads simultaneously (as might be done in some lock-related transformations), we may require both an extended language of side conditions and more general proof techniques, such as the rely-guarantee approach found in RGSim [8]. Similar approaches may help us handle other models of parallel programming, such as fork-join parallelism.

References

1 Sebastian Burckhardt, Madanlal Musuvathi, and Vasu Singh. Verifying local transformations on relaxed memory models. In *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction*, CC'10/ETAPS'10, pages 104–123, Berlin, Heidelberg, 2010. Springer-Verlag.

2 Matthew Hennessy and Robin Milner. On observing nondeterminism and concurrency. In Jaco de Bakker and Jan van Leeuwen, editors, *Automata, Languages and Programming*, volume 85 of *Lecture Notes in Computer Science*, pages 299–309. Springer Berlin / Heidelberg, 1980. doi: 10.1007/3-540-10003-2_79.

3 C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

4 Sara Kalvala, Richard Warburton, and David Lacey. Program transformations using temporal logic side conditions. *ACM Trans. Program. Lang. Syst.*, 31(4):1–48, 2009.

5 Jens Krinke. Context-sensitive slicing of concurrent programs. *SIGSOFT Softw. Eng. Notes*, 28(5):178–187, September 2003.

6 Sorin Lerner, Todd Millstein, and Craig Chambers. Automatically proving the correctness of compiler optimizations. *SIGPLAN Not.*, 38:220–231, May 2003.

7 Xavier Leroy. A formally verified compiler back-end. *J. Autom. Reason.*, 43(4):363–446, December 2009.

8 Hongjin Liang, Xinyu Feng, and Ming Fu. A rely-guarantee-based simulation for verifying concurrent program transformations. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL'12, pages 455–468, New York, NY, USA, 2012. ACM.

9 LLVM Language Reference Manual. http://llvm.org/docs/LangRef.html, April 2014.

10 William Mansky, Dennis Griffith, and Elsa L. Gunter. Specifying and executing optimizations for parallel programs. Accepted for publication by GRAPHITE'14.

11 William Mansky and Elsa Gunter. A framework for formal verification of compiler optimizations. In *Proceedings of the First international conference on Interactive Theorem Proving*, ITP'10, pages 371–386, Berlin, Heidelberg, 2010. Springer-Verlag.

12 Lawrence C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.

13 Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In *TACAS'98: Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 151–166, London, UK, 1998. Springer-Verlag.

14 Pradeep S. Sindhu, Jean-Marc Frailong, and Michel Cekleov. Formal specification of memory models. In Michel Dubois and Shreekant Thakkar, editors, *Scalable Shared Memory Multiprocessors*, pages 25–41. Springer US, 1992.

15 Jaroslav Ševčík. Safe optimisations for shared-memory concurrent programs. *SIGPLAN Not.*, 46(6):306–316, June 2011.

16 Jaroslav Ševčik, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. Relaxed-memory concurrency and verified compilation. *SIGPLAN Not.*, 46(1):43–54, January 2011.

17 Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. *SIGPLAN Not.*, 46(6):283–294, June 2011.

Inverse Unfold Problem and Its Heuristic Solving

Masanori Nagashima, Tomofumi Kato, Masahiko Sakai, and Naoki Nishida

Graduate School of Information Science, Nagoya University
Furo-cho, Chikusa-ku, Nagoya 464–8603 Japan
nagashima@sakabe.i.is.nagoya-u.ac.jp, tomofumi@trs.cm.is.nagoya-u.ac.jp,
sakai@is.nagoya-u.ac.jp, nishida@is.nagoya-u.ac.jp

Abstract

Unfold/fold transformations have been widely studied in various programming paradigms and are used in program transformations, theorem proving, and so on. This paper, by using an example, show that restoring an one-step unfolding is not easy, i.e., a challenging task, since some rules used by unfolding may be lost. We formalize this problem by regarding one-step program transformation as a relation. Next we discuss some issues on a specific framework, called pure-constructor systems, which constitute a subclass of conditional term rewriting systems. We show that the inverse of T preserves rewrite relations if T preserves rewrite relations and the signature. We propose a heuristic procedure to solve the problem, and show its successful examples. We improve the procedure, and show examples for which the improvement takes effect.

1998 ACM Subject Classification I.2.2 Automatic Programming

Keywords and phrases program transformation, unfolding, conditional term rewriting system

Digital Object Identifier 10.4230/OASIcs.WPTE.2014.27

1 Introduction

Unfold/fold transformations have been widely studied on functional[6, 23], logic[11, 24, 25, 21, 22, 20] and constraint logic [12, 7, 4, 8] programs. They are used in program transformations, theorem proving, and so on.

This paper proposes the inverse problem of one-step unfolding. Let's see that the problem is not trivial by an example in terms of term rewriting systems (TRSs). Both TRSs $\mathcal{R}_1$ and $\mathcal{R}_2$, given as follows, define the same function mult that computes the multiplication of two natural numbers:

$$\mathcal{R}_1 = \left\{ \begin{array}{l} \mathsf{mult}(0, y) \to 0, \\ \mathsf{mult}(\mathsf{s}(x), y) \to \mathsf{add}(\mathsf{mult}(x, y), y) \end{array} \right\} \cup \mathcal{R}_{\mathsf{add}}, \text{ and}$$

$$\mathcal{R}_2 = \left\{ \begin{array}{l} \mathsf{mult}(0, y) \to 0, \\ \mathsf{mult}(\mathsf{s}(0), y) \to \mathsf{add}(0, y), \\ \mathsf{mult}(\mathsf{s}^2(x), y) \to \mathsf{add}(\mathsf{add}(\mathsf{mult}(x, y), y), y) \end{array} \right\} \cup \mathcal{R}_{\mathsf{add}},$$

where

$$\mathcal{R}_{\mathsf{add}} = \left\{ \begin{array}{l} \mathsf{add}(0, y) \to y, \\ \mathsf{add}(\mathsf{s}(x), y) \to \mathsf{s}(\mathsf{add}(x, y)) \end{array} \right\}.$$

Here $\mathcal{R}_2$ is derived from $\mathcal{R}_1$ by unfolding $\mathsf{mult}(x, y)$ in the right-hand side of the second rewrite rule in $\mathcal{R}_1$ by using the rules for mult in $\mathcal{R}_1$. On the other hand, however, it is difficult to transform $\mathcal{R}_2$ into $\mathcal{R}_1$ in the reverse direction. One may think a folding operation

1st International Workshop on Rewriting Techniques for Program Transformations and Evaluation (WPTE'14).
Editors: Manfred Schmidt-Schauß, Masahiko Sakai, David Sabel, and Yuki Chiba; pp. 27–38

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

is applicable for this purpose, but it is impossible because the second rewrite rule of $\mathcal{R}_1$ is necessary for the folding, but is missing in $\mathcal{R}_2$.

This paper is organized as follows. First, we define the *inverse problem* of an one-step program transformation in Section 3. In sessions that follow Section 3, we discuss some issues on a specific framework, called pure-constructor system, used in our previous work [14] on a determinization of conditional term rewriting systems. We targeted this framework because of the firmness of the structure of the rules, i.e., every root position of left-hand sides in the body or conditions of rewrite rules is a defined symbol, and all the other position have no defined symbols even in right-hand sides. Remark that a deterministic pure-constructor system is convertible to an equivalent TRS, vice versa. Nested defined symbols in a right-hand side of a rule of a TRS are represented by a sequence of conditions.

In Section 4, we show that the inverse of an one-step transformation T preserves rewrite relations if T preserves rewrite relations and their signatures. In Section 5, we propose a heuristic procedure for this problem and show some examples. Overcoming failure examples, we propose an advanced heuristic solving in Section 6 and demonstrate its effectiveness for those examples. Finally, in Section 7, we show a motivated example induced from the program inversion[10, 15, 16, 17, 18].

2 Preliminaries

In this section, we introduce notations used in this paper. We assume that the reader is familiar with basic concepts of term rewriting [2, 19].

Let $\mathcal{F}$ be a *signature*, a finite set of *function symbols* accompanied with a mapping *arity* which maps each function symbol f to a natural number arity(f). $\mathcal{F}$ is assumed to be partitioned into two disjoint sets $\mathcal{D}$ of *defined symbols* and $\mathcal{C}$ of *constructors*, that is, $\mathcal{F} = \mathcal{D} \uplus \mathcal{C}$. Let $\mathcal{V}$ be a countably infinite set of *variables* such that $\mathcal{F} \cap \mathcal{V} = \emptyset$. A function symbol $g \in \mathcal{F}$ is called a *constant* if arity$(g) = 0$.

The set of *terms* over $\mathcal{F}$ and $\mathcal{V}$ is denoted by $\mathcal{T}(\mathcal{F}, \mathcal{V})$, the set of variables occurring at least one of terms $t_1, \ldots, t_n$ by $\mathcal{V}ar(t_1, \ldots, t_n)$. A term $t \in \mathcal{T}(\mathcal{F}, \emptyset)$ is called *ground*. The set of all ground terms is denoted by $\mathcal{T}(\mathcal{F})$. A term $t \in \mathcal{T}(\mathcal{C}, \mathcal{V})$ is called a *constructor term*. A term of the form $f(t_1, \ldots, t_n)$ is called a *pattern* in $\mathcal{T}(\mathcal{F}, \mathcal{V})$ if $f \in \mathcal{D}$ and $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{C}, \mathcal{V})$. The *root symbol* of a term t is denoted by root(t).

Let $\Box \notin \mathcal{F} \cup \mathcal{V}$ be a special constant, called a *hole*. A *context* is a term $C \in \mathcal{T}(\mathcal{F} \cup \{\Box\}, \mathcal{V})$ with exactly one occurrence of $\Box$. We write $C[t]$ for the term obtained from C by replacing the occurrence of $\Box$ in C with a term t.

A *substitution* is a function $\sigma : \mathcal{V} \to \mathcal{T}(\mathcal{F}, \mathcal{V})$ such that $\{x \mid \sigma(x) \neq x\}$ is finite. The set $\{x \mid \sigma(x) \neq x\}$ is denoted by $\mathcal{D}om(\sigma)$ and called the *domain* of σ. A substitution σ can be extended to $\sigma : \mathcal{T}(\mathcal{F}, \mathcal{V}) \to \mathcal{T}(\mathcal{F}, \mathcal{V})$ in a natural way. We write $t\sigma$ for $\sigma(t)$. A substitution σ is *ground* if $x\sigma \in \mathcal{T}(\mathcal{F})$ for all $x \in \mathcal{D}om(\sigma)$. A substitution σ is a *constructor substitution* if $x\sigma \in \mathcal{T}(\mathcal{C}, \mathcal{V})$ for all $x \in \mathcal{D}om(\sigma)$. The *composition* $\sigma\theta$ of two substitutions σ and θ is defined as $x(\sigma\theta) = (x\sigma)\theta$.

An *equation* is a pair of terms $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$, which is denoted by $s \sim t$. A substitution σ is a *unifier* of a set E of equations if $s\sigma = t\sigma$ for every $s \sim t \in E$. E is said to be *unifiable* if there is a unifier of E. A unifier σ of E is a *most general unifier* of E if for any unifier θ of E, there exists a substitution δ such that $\theta = \sigma\delta$. It is known that most general unifiers of E are unique up to variable renaming. We use mgu(E) for the most general unifier of E.

An *oriented conditional rewrite rule* (*rewrite rule*, for short) is a formula in the form of $l \to r \Leftarrow u_1 \to v_1; \cdots; u_n \to v_n$ $(n \geq 0)$. $l \to r$ and $u_1 \to v_1; \cdots; u_n \to v_n$ are called the

body part and the *conditional part* of the rule, respectively. Terms l and r are called the *left-hand side* and the *right-hand side* of the rule, respectively. Each $u_i \to v_i$ $(1 \leq i \leq n)$ is called a *condition* of the rule. A rewrite rule whose conditional part is empty is called an *unconditional rule*, denoted as $l \to r$ by omitting $\Leftarrow$. The set of variables occurring in an object o (e.g., a rewrite rule) is denoted by $\mathcal{V}ar(o)$.

A condition $u \to v$ is called a *pattern condition* if u is a pattern and v is a constructor term. A rewrite rule $l \to r \Leftarrow c$ is called a *pure-constructor rule* if l is a pattern, r is a constructor term, and the conditions of c are all pattern conditions. Note that pure-constructor rules are also *normal* [5]. A rewrite rule $l \to r \Leftarrow c$ is said to be of *type 3* [13] if $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l) \cup \mathcal{V}ar(c)$. A conditional rewrite rule $l \to r \Leftarrow u_1 \to v_1; \cdots; u_n \to v_n$ of type 3 is called *deterministic* [9] if $\mathcal{V}ar(u_i) \subseteq \mathcal{V}ar(l, v_1, \ldots, v_{i-1})$ for all i $(1 \leq i \leq n)$.

An *oriented conditional term rewriting system* (CTRS, for short) is a finite set $\mathcal{R}$ of oriented conditional rewrite rules. A *pure-constructor system* is a CTRS whose rewrite rules are all pure-constructor rewrite rules.[1] A CTRS $\mathcal{R}$ is called *deterministic* if all rewrite rules of $\mathcal{R}$ are deterministic.

▶ **Example 2.1.** CTRS $\mathcal{R}_{\mathsf{add}}$ in Section 1 is convertible to the following equivalent and deterministic pure-constructor system:

$$\mathcal{R}_{\mathsf{add}} = \left\{ \begin{array}{l} \mathsf{add}(0, y) \to y \\ \mathsf{add}(\mathsf{s}(x), y) \to \mathsf{s}(z) \Leftarrow \mathsf{add}(x, y) \to z \end{array} \right\}.$$

We introduce a relational logic over $\mathcal{T}(\mathcal{F}, \mathcal{V})$. An atom is a pair of terms s and t, denoted by $s \to t$. Formulas are atoms, existentially quantified formulas, conjunction of formulas and implication of formulas. Satisfaction of a formula φ by a pair of a relation $\rightharpoonup$ on $\mathcal{T}(\mathcal{F}, \mathcal{V})$ and a substitution $\sigma : \mathcal{V} \to \mathcal{T}(\mathcal{F}, \mathcal{V})$, denoted by $\langle \rightharpoonup, \sigma \rangle \models \varphi$, is inductively defined as follows:

- $\langle \rightharpoonup, \sigma \rangle \models u \to v$ iff $u\sigma \rightharpoonup v\sigma$,
- $\langle \rightharpoonup, \sigma \rangle \models \varphi \wedge \varphi'$ iff $\langle \rightharpoonup, \sigma \rangle \models \varphi$ and $\langle \rightharpoonup, \sigma \rangle \models \varphi'$, and
- $\langle \rightharpoonup, \sigma \rangle \models \varphi \Rightarrow \varphi'$ iff $\langle \rightharpoonup, \sigma \rangle \models \varphi$ implies $\langle \rightharpoonup, \sigma \rangle \models \varphi'$.

Note that a sequence of conditions $u_1 \to v_1; \cdots; u_n \to v_n$ is regarded as conjunction $u_1 \to v_1 \wedge \cdots \wedge u_n \to v_n$. The *reflexive transitive closure* of a relation $\rightharpoonup$ is denoted by $\overset{*}{\rightharpoonup}$.

The *k-level reduction* $\to_{\mathcal{R}}^{(k)}$ of $\mathcal{R}$ is inductively defined as follows:

- $\to_{\mathcal{R}}^{(0)} = \emptyset$, and
- $\to_{\mathcal{R}}^{(j)} = \to_{\mathcal{R}}^{(j-1)} \cup \{(C[l\sigma], C[r\sigma]) \mid l \to r \Leftarrow c \in \mathcal{R},\ C \in \mathcal{T}(\mathcal{F} \cup \{\Box\}, \mathcal{V}),$ σ is a substitution, $\langle \overset{*}{\to}_{\mathcal{R}}^{(j-1)}, \sigma \rangle \models c\}$ for $j > 0$.

The *reduction* $\to_{\mathcal{R}}$ is defined as $\bigcup_{k \geq 0} \to_{\mathcal{R}}^{(k)}$.

The constructor-based reduction $\underset{c}{\to}_{\mathcal{R}}$ of a CTRS $\mathcal{R}$ [16] can be defined in the same way as the ordinary reduction of a CTRS except that matching substitutions are restricted to constructor substitutions. Note that $\underset{c}{\to}_{\mathcal{R}} \subseteq \to_{\mathcal{R}}$.

3 Inverse Problem of Program Transformation

A program transformation is a procedure to generate another program from a given program and its main purpose is to improve execution efficiency of programs. On the other hand, the inverse of a program transformation is not a function, since program transformations are not one-to-one in general. The notion of the inverse of a program transformation is captured as an inverse image of a program under the transformation.

[1] The class of pure-constructor systems is the same as the class of normalized TRSs in [1].

Unfold:

$$\frac{\mathcal{R}_1 = \mathcal{R} \cup \{\rho :\ l \to r \Leftarrow c; u \to v; d\}}{\mathcal{R}\ \cup \{l\sigma \to r\sigma \Leftarrow c\sigma; c'\sigma; d\sigma \mid \rho' :\ l' \to r' \Leftarrow c' \in \mathcal{R}_1, \mathcal{V}ar(\rho) \cap \mathcal{V}ar(\rho') = \emptyset,\ \sigma = \mathrm{mgu}(\{u \sim l', v \sim r'\})\}}$$

Fold: If $\rho' :\ l' \to r' \Leftarrow c' \in \mathcal{R}$,

$$\frac{\mathcal{R} \cup \{\rho :\ l \to r \Leftarrow c; c'\sigma; d\}}{\mathcal{R} \cup \{l \to r \Leftarrow c; l'\sigma \to r'\sigma; d\}}$$

Note that

1. transformations are performed by applying the transformation rules above in the direction from top to bottom,
2. each substitution σ above is restricted to a constructor substitution,
3. each rewrite rule ρ' above should be renamed so that $\mathcal{V}ar(\rho) \cap \mathcal{V}ar(\rho') = \emptyset$ holds, and
4. we sometimes say "Unfold ρ by ρ'" and "Fold ρ by ρ'" in applying the transformation rules above.

Figure 1 Unfold/Fold Transformation Rules.

In this section, we formalize the inverse problem of an one-step program transformation by regarding it as a relation over programs. For example, an one-step transformation T that converts a program $\mathcal{R}_1$ into $\mathcal{R}_2$ is a relation T with $\mathcal{R}_1\ T\ \mathcal{R}_2$.

The inverse problem of an one-step program transformation is formalized as follows.

▶ **Definition 3.1.** Given a transformation T and a program $\mathcal{R}_1$, the *inverse T problem (T^{-1} problem)* determines whether or not there exists a program $\mathcal{R}_0$ such that $\mathcal{R}_0\ T\ \mathcal{R}_1$ holds. If there exists such $\mathcal{R}_0$, we write $\mathcal{R}_1\ T^{-1}\ \mathcal{R}_0$, which means that transformation T^{-1} is equals to inverse relation of T. $\mathcal{R}_0$ is called a *solution* of T^{-1} problem for $\mathcal{R}_1$.

4 Inverse Unfold Problem

In the rest of the paper, we focus on Unfold^{-1} problem, where we use Unfold/Fold transformation rules [14] on deterministic pure-constructor CTRSs. The definitions of those transformation rules are shown in Figure 1. Remark that a CTRS obtained by applying Unfold is equivalent to the original one, but a CTRS obtained by applying Fold, which is a derived rule of the one in [14], is not equivalent in general. More details on the correctness of Unfold/Fold transformations on pure-constructor systems are discussed in [14].

We revisit the examples in Section 1. $\mathcal{R}_{\mathsf{mult}}$ and $\mathcal{R}_{\mathsf{mult1}}$ in the following example are pure-constructor CTRSs equivalent to $\mathcal{R}_1$ and $\mathcal{R}_2$ in Section 1, respectively.

▶ **Example 4.1.** We obtain a CTRS $\mathcal{R}_{\mathsf{mult1}}$ by applying Unfold rule to the second rewrite rule of $\mathcal{R}_{\mathsf{mult}}$ using its own two rewrite rules.

$$\mathcal{R}_{\mathsf{mult}} = \left\{ \begin{array}{l} \mathsf{mult}(0, y) \to 0 \\ \mathsf{mult}(\mathsf{s}(x), y) \to v \Leftarrow \mathsf{mult}(x, y) \to z; \mathsf{add}(z, y) \to v \end{array} \right\}$$

$$\mathcal{R}_{\mathsf{mult1}} = \left\{ \begin{array}{l} \mathsf{mult}(0, y) \to 0 \\ \mathsf{mult}(\mathsf{s}(0), y) \to v \Leftarrow \mathsf{add}(0, y) \to v \\ \mathsf{mult}(\mathsf{s}^2(x), y) \to v \Leftarrow \mathsf{mult}(x, y) \to z; \mathsf{add}(z, y) \to w; \mathsf{add}(w, y) \to v \end{array} \right\}$$

Unfold^{-1} problem can be regarded as an instance of Definition 3.1 i.e. a decision problem that determines, for a given CTRS $\mathcal{R}_1$, whether or not there exists a program $\mathcal{R}_0$, from which $\mathcal{R}_1$ is obtained by Unfold rule ($\mathcal{R}_0$ Unfold $\mathcal{R}_1$).

▶ **Example 4.2.** A solution of Unfold^{-1} problem for CTRS $\mathcal{R}_{\mathsf{mult1}}$ is CTRS $\mathcal{R}_{\mathsf{mult}}$ because $\mathcal{R}_{\mathsf{mult}}$ Unfold $\mathcal{R}_{\mathsf{mult1}}$ holds in Example 4.1.

Program transformations are usually demanded to preserve the meaning of programs. In case of Example 4.1, the two functions mult defined in $\mathcal{R}_{\mathsf{mult}}$ and $\mathcal{R}_{\mathsf{mult1}}$ are required to derive the same normal forms if the same ground terms are given in the arguments. This preservation property is known as the combination of properties *Simulation soundness* and *simulation completeness*.

These properties for program transformation on pure-constructor systems are defined as follows [14].

▶ **Definition 4.3.** A transformation T over CTRSs is *simulation sound* if and only if $s \xrightarrow[\mathsf{c}]{*}_{\mathcal{R}_2} t$ implies $s \xrightarrow[\mathsf{c}]{*}_{\mathcal{R}_1} t$ for any CTRSs $\mathcal{R}_1$, $\mathcal{R}_2$ such that $\mathcal{R}_1\ T\ \mathcal{R}_2$, and for any $s, t \in \mathcal{T}(\mathcal{D}_{\mathcal{R}_1} \cup \mathcal{C})$.

▶ **Definition 4.4.** A transformation T over CTRSs is *simulation complete* if and only if $s \xrightarrow[\mathsf{c}]{*}_{\mathcal{R}_1} t$ implies $s \xrightarrow[\mathsf{c}]{*}_{\mathcal{R}_2} t$ for any CTRSs $\mathcal{R}_1$, $\mathcal{R}_2$ such that $\mathcal{R}_1\ T\ \mathcal{R}_2$, and for any $s, t \in \mathcal{T}(\mathcal{D}_{\mathcal{R}_1} \cup \mathcal{C})$.

In T^{-1} problem, even if transformation T is simulation sound and complete, T^{-1} is not so in general. A sufficient condition for the simulation soundness and completeness of the inverse problem is easily shown as follows.

▶ **Theorem 4.5.** *If a transformation T is simulation sound and complete, and T introduce no new defined symbol, T^{-1} is also simulation sound and complete.*

Proof. Let $\mathcal{R}_1$, $\mathcal{R}_2$ be CTRSs such that $\mathcal{R}_1\ T^{-1}\ \mathcal{R}_2$. Suppose $s, t \in \mathcal{T}(\mathcal{D}_{\mathcal{R}_1} \cup \mathcal{C})$. By the definition of T^{-1} problem, $\mathcal{R}_2\ T\ \mathcal{R}_1$ holds. Since T introduce no new defined symbol, i.e., $\mathcal{D}_{\mathcal{R}_1} = \mathcal{D}_{\mathcal{R}_2}$, it follows that $s, t \in \mathcal{T}(\mathcal{D}_{\mathcal{R}_2} \cup \mathcal{C})$. Combined this with T's simulation soundness and completeness, ($s \xrightarrow[\mathsf{c}]{*}_{\mathcal{R}_1} t$ implies $s \xrightarrow[\mathsf{c}]{*}_{\mathcal{R}_2} t$) and ($s \xrightarrow[\mathsf{c}]{*}_{\mathcal{R}_2} t$ implies $s \xrightarrow[\mathsf{c}]{*}_{\mathcal{R}_1} t$) hold. ◀

Unfold rule in Figure 1 is simulation sound and complete [14], and introduces no new defined symbol. Thus, Theorem 4.5 derives the following corollary.

▶ **Corollary 4.6.** Unfold^{-1} *is simulation sound and complete.*

This corollary guarantees both CTRSs before and after Unfold^{-1} have the same rewrite relation.

5 Heuristics for Solving Inverse Unfold Problem

We propose a heuristic procedure for solving Unfold^{-1} problems, which is shown in Figure 2. In this procedure, we generate a divergent sequence of rewrite rules by applying Unfold/Fold rules in Figure 1 from a given CTRS $\mathcal{R}_1$ (Step 1–4), generalize rules in the sequence by the difference matching [3] (Step 5) to obtain a solution candidate CTRS $\mathcal{R}_2$ for the Unfold^{-1} problem. In Step 4 and 5, the simulation soundness and completeness are not necessarily preserved. Therefore, Step 6 is necessary in order to confirm that the obtained CTRS $\mathcal{R}_2$ is a solution, where $\mathcal{R}_2$ preserves the behavior of $\mathcal{R}_1$ by Corollary 4.6. Remark that in Step 2, "Unfolding" ($\mathcal{I}$)-labelled rule by (b_0)-, (b_1)- and ($\mathcal{I}$)-labelled rules yields ($\mathcal{I}b_0$)-, ($\mathcal{I}b_1$)- and ($\mathcal{I}^2$)-labelled rules, for example.

Step 1 Give the labels $(b_0), (b_1), \ldots$ to each rule for a base case of the target function in the given CTRS $\mathcal{R}_1$ in order of argument size. Similarly, give the label $(\mathcal{I})$ to the rule for the induction case.
Step 2 Unfold $(\mathcal{I}^n)$-labelled[1)] rule by all rules including itself. Attach each resulted rule the label obtained by concatenating the label of Unfolded rule and that of Unfolding rule in this order.
Step 3 Again, do (Step 2) l times[2)].
Step 4 Fold each $(\mathcal{I}^n b_m)$-labelled rule by $(\mathcal{I}^n b_{m-1})$-labelled rule in lexicographically descending order of (n, m). In an exceptional case, each $(\mathcal{I}^n b_0)$-labelled rule is "Fold"ed by $(\mathcal{I}^{n-1} b_{\mathsf{max}(m)})$-labelled rule. The each label of generated rules is $(\mathcal{I}^n b_m)'$, respectively.
Step 5 Generalize rules in the divergent sequence generated in (Step 4) by the difference matching [3] to obtain a solution candidate CTRS $\mathcal{R}_2$.
Step 6 If $\mathcal{R}_2$ Unfold $\mathcal{R}_1$ is satisfied, $\mathcal{R}_2$ is a solution of Unfold^{-1} problem for $\mathcal{R}_1$.

1) $\mathcal{I}^n$ denotes $\overbrace{\mathcal{I} \cdots \mathcal{I}}^{n}$.
2) l is a given fixed number.

Figure 2 Heuristic procedure for Unfold^{-1} problem.

▶ **Example 5.1.** We solve Unfold^{-1} problem for $\mathcal{R}_{\mathsf{mult1}}$ in Example 4.1 by applying the heuristic procedure in Figure 2.

Step 1: First, we give labels to rewrite rules in $\mathcal{R}_{\mathsf{mult1}}$.

$$\mathcal{R}_{\mathsf{mult1}} = \left\{ \begin{array}{ll} \mathsf{mult}(0, y) \to 0 & (b_0) \\ \mathsf{mult}(\mathsf{s}(0), y) \to v \ \Leftarrow \mathsf{add}(0, y) \to v & (b_1) \\ \mathsf{mult}(\mathsf{s}^2(x), y) \to v \Leftarrow \mathsf{mult}(x, y) \to z; \mathsf{add}(z, y) \to w; \mathsf{add}(w, y) \to v & (\mathcal{I}) \end{array} \right\}$$

Step 2: Next, we "Unfold" $(\mathcal{I})$-labelled rule by (b_0)-, (b_1)- and $(\mathcal{I})$-labelled rules, which yields the following CTRS $\mathcal{R}_{\mathsf{mult2}}$.

$$\mathcal{R}_{\mathsf{mult2}} = \left\{ \begin{array}{ll} \mathsf{mult}(0, y) \to 0 & (b_0) \\ \mathsf{mult}(\mathsf{s}(0), y) \to v \ \Leftarrow \mathsf{add}(0, y) \to v & (b_1) \\ \mathsf{mult}(\mathsf{s}^2(0), y) \to v \Leftarrow \mathsf{add}(0, y) \to w; \mathsf{add}(w, y) \to v & (\mathcal{I}b_0) \\ \mathsf{mult}(\mathsf{s}^3(0), y) \to v \Leftarrow \mathsf{add}(0, y) \to w_2; \mathsf{add}(w_2, y) \to w; \mathsf{add}(w, y) \to v & (\mathcal{I}b_1) \\ \mathsf{mult}(\mathsf{s}^4(x), y) \to v \Leftarrow \mathsf{mult}(x, y) \to z; & \\ \qquad \mathsf{add}(z, y) \to w_3; \mathsf{add}(w_3, y) \to w_2; & \\ \qquad \mathsf{add}(w_2, y) \to w; \mathsf{add}(w, y) \to v & (\mathcal{I}^2) \end{array} \right\}$$

Step 3: Again, we "Unfold" $(\mathcal{I}^2)$-labelled rule in $\mathcal{R}_{\mathsf{mult2}}$ by the rules with the labels from (b_0) through $(\mathcal{I}^2)$, which yields the following CTRS $\mathcal{R}_{\mathsf{mult3}}$.

$$\mathcal{R}_{\mathsf{mult3}} = \left\{ \begin{array}{ll} \mathsf{mult}(0, y) \to 0 & (b_0) \\ \mathsf{mult}(\mathsf{s}(0), y) \to v \ \Leftarrow \mathsf{add}(0, y) \to v & (b_1) \\ \mathsf{mult}(\mathsf{s}^2(0), y) \to v \Leftarrow \mathsf{add}(0, y) \to w; \mathsf{add}(w, y) \to v & (\mathcal{I}b_0) \\ \mathsf{mult}(\mathsf{s}^3(0), y) \to v \Leftarrow \mathsf{add}(0, y) \to w_2; \mathsf{add}(w_2, y) \to w; \mathsf{add}(w, y) \to v & (\mathcal{I}b_1) \\ \mathsf{mult}(\mathsf{s}^4(0), y) \to v \Leftarrow \mathsf{add}(0, y) \to w_3; \cdots; \mathsf{add}(w, y) \to v & (\mathcal{I}^2 b_0) \\ \mathsf{mult}(\mathsf{s}^5(0), y) \to v \Leftarrow \mathsf{add}(0, y) \to w_4; \cdots; \mathsf{add}(w, y) \to v & (\mathcal{I}^2 b_1) \\ \mathsf{mult}(\mathsf{s}^6(0), y) \to v \Leftarrow \mathsf{add}(0, y) \to w_5; \cdots; \mathsf{add}(w, y) \to v & (\mathcal{I}^3 b_0) \\ \mathsf{mult}(\mathsf{s}^7(0), y) \to v \Leftarrow \mathsf{add}(0, y) \to w_6; \cdots; \mathsf{add}(w, y) \to v & (\mathcal{I}^3 b_1) \\ \mathsf{mult}(\mathsf{s}^8(x), y) \to v \Leftarrow \mathsf{mult}(x, y) \to z; & \\ \qquad \mathsf{add}(z, y) \to w_7; \cdots; \mathsf{add}(w, y) \to v & (\mathcal{I}^4) \end{array} \right\}$$

Table 1 Examples of heuristic procedure in Figure 2.

Given CTRS	Solution CTRS
$\left\{\begin{array}{l}\mathsf{add}(0,y)\to y\\ \mathsf{add}(\mathsf{s}(0),y)\to\mathsf{s}(y)\\ \mathsf{add}(\mathsf{s}^2(x),y)\to\mathsf{s}^2(z)\Leftarrow\mathsf{add}(x,y)\to z\end{array}\right\}$	–
$\left\{\begin{array}{l}\mathsf{mult}(0,y)\to 0\\ \mathsf{mult}(\mathsf{s}(0),y)\to v\ \Leftarrow\mathsf{add}(0,y)\to v\\ \mathsf{mult}(\mathsf{s}^2(x),y)\to v\Leftarrow\mathsf{mult}(x,y)\to z;\\ \qquad\qquad\qquad\qquad\mathsf{add}(z,y)\to w;\\ \qquad\qquad\qquad\qquad\mathsf{add}(w,y)\to v\end{array}\right\}$	$\left\{\begin{array}{l}\mathsf{mult}(0,y)\to 0\\ \mathsf{mult}(\mathsf{s}(x),y)\to v\Leftarrow\mathsf{mult}(x,y)\to z;\\ \qquad\qquad\qquad\qquad\mathsf{add}(z,y)\to v\end{array}\right\}$
$\left\{\begin{array}{l}\mathsf{rev}([\,])\to[\,]\\ \mathsf{rev}(x_1:[\,])\to zs\Leftarrow\mathsf{app}([\,],x_1:[\,])\to zs\\ \mathsf{rev}(x_1:x_2:xs)\to zs\\ \qquad\qquad\Leftarrow\mathsf{rev}(xs)\to zs_2;\\ \qquad\qquad\quad\mathsf{app}(zs_2,x_2:[\,])\to zs_1;\\ \qquad\qquad\quad\mathsf{app}(zs_1,x_1:[\,])\to zs\end{array}\right\}$	$\left\{\begin{array}{l}\mathsf{rev}([\,])\to[\,]\\ \mathsf{rev}(x:xs)\to zs\Leftarrow\mathsf{rev}(xs)\to zs_1;\\ \qquad\qquad\qquad\qquad\mathsf{app}(zs_1,x:[\,])\to zs\end{array}\right\}$
$\left\{\begin{array}{l}\mathsf{frev}([\,],ys)\to ys\\ \mathsf{frev}(x:[\,],ys)\to x:ys\\ \mathsf{frev}(x_1:x_2:xs,ys)\to zs\\ \qquad\qquad\Leftarrow\mathsf{frev}(xs,x_2:x_1:ys)\to zs\end{array}\right\}$	–

Step 4: We "Fold" $(\mathcal{I}^3b_1)$-labelled rule by $(\mathcal{I}^3b_0)$-labelled rule, which yields a rule "$\mathsf{mult}(\mathsf{s}^7(0),y)\to v\Leftarrow\mathsf{mult}(\mathsf{s}^6(0),y)\to w,\mathsf{add}(w,y)\to v$". Similarly, the rules with the labels from $(\mathcal{I}^3b_0)$ through $(\mathcal{I}b_0)$ are "Folded" by the rules with the labels from $(\mathcal{I}^2b_1)$ through (b_1), respectively. We get the following CTRS $\mathcal{R}_{\mathsf{mult3}'}$.

$$\mathcal{R}_{\mathsf{mult3}'}=\left\{\begin{array}{lll}\mathsf{mult}(0,y)\to 0 & & (b_0)\\ \mathsf{mult}(\mathsf{s}(0),y)\to v & \Leftarrow\mathsf{add}(0,y)\to v & (b_1)\\ \mathsf{mult}(\mathsf{s}(\mathsf{s}(0)),y)\to v & \Leftarrow\mathsf{mult}(\mathsf{s}(0),y)\to w;\mathsf{add}(w,y)\to v & (\mathcal{I}b_0)'\\ \mathsf{mult}(\mathsf{s}(\mathsf{s}^2(0)),y)\to v & \Leftarrow\mathsf{mult}(\mathsf{s}^2(0),y)\to w;\mathsf{add}(w,y)\to v & (\mathcal{I}b_1)'\\ \mathsf{mult}(\mathsf{s}(\mathsf{s}^3(0)),y)\to v & \Leftarrow\mathsf{mult}(\mathsf{s}^3(0),y)\to w;\mathsf{add}(w,y)\to v & (\mathcal{I}^2b_0)'\\ \mathsf{mult}(\mathsf{s}(\mathsf{s}^4(0)),y)\to v & \Leftarrow\mathsf{mult}(\mathsf{s}^4(0),y)\to w;\mathsf{add}(w,y)\to v & (\mathcal{I}^2b_1)'\\ \mathsf{mult}(\mathsf{s}(\mathsf{s}^5(0)),y)\to v & \Leftarrow\mathsf{mult}(\mathsf{s}^5(0),y)\to w;\mathsf{add}(w,y)\to v & (\mathcal{I}^3b_0)'\\ \mathsf{mult}(\mathsf{s}(\mathsf{s}^6(0)),y)\to v & \Leftarrow\mathsf{mult}(\mathsf{s}^6(0),y)\to w;\mathsf{add}(w,y)\to v & (\mathcal{I}^3b_1)'\\ \mathsf{mult}(\mathsf{s}^8(x),y)\to v & \Leftarrow\mathsf{mult}(x,y)\to z; & \\ & \quad\mathsf{add}(z,y)\to w_7;\cdots;\mathsf{add}(w,y)\to v & (\mathcal{I}^4)\end{array}\right\}$$

Step 5: Generalize the divergent rules in $\mathcal{R}_{\mathsf{mult3}'}$ with the labels from $(\mathcal{I}b_0)'$ through $(\mathcal{I}^3b_1)'$ by the difference matching, which yields a rule "$\mathsf{mult}(\mathsf{s}(x),y)\to v\Leftarrow\mathsf{mult}(x,y)\to w,\mathsf{add}(w,y)\to v$". Now $\mathcal{R}_{\mathsf{mult}}$ is obtained as a solution candidate of the Unfold^{-1} problem for $\mathcal{R}_{\mathsf{mult1}}$.

Step 6: It is confirmed that $\mathcal{R}_{\mathsf{mult}}$ Unfold $\mathcal{R}_{\mathsf{mult1}}$. Thus $\mathcal{R}_{\mathsf{mult}}$ is certainly a solution.

Table 1 shows the results obtained by applying the procedure in Figure 2 by hand to four problems: addition of two natural numbers, multiplication of two natural numbers, reverse of a list and fast reverse of a list. Here, '–' in the table shows that the procedure failed to solve Unfold^{-1} for the problem.

6 Heuristics Introducing Identity Function

Consider the first CTRS $\mathcal{R}_{\mathsf{add1}}$ in Table 1, for which the procedure in Section 5 fails.

$$\mathcal{R}_{\mathsf{add1}} = \left\{ \begin{array}{l} \mathsf{add}(0, y) \to y \\ \mathsf{add}(\mathsf{s}(0), y) \to \mathsf{s}(y) \\ \mathsf{add}(\mathsf{s}^2(x), y) \to \mathsf{s}^2(z) \Leftarrow \mathsf{add}(x, y) \to z \end{array} \right\}.$$

The following CTRS is obtained from $\mathcal{R}_{\mathsf{add1}}$ as an intermediate result by the heuristic procedure in Figure 2; applying Step 1 to Step 3 (Step 3 twice):

$$\mathcal{R}' = \left\{ \begin{array}{ll} \mathsf{add}(0, y) \to y & (b_0) \\ \mathsf{add}(\mathsf{s}(0), y) \to \mathsf{s}(y) & (b_1) \\ \mathsf{add}(\mathsf{s}^2(0), y) \to \mathsf{s}^2(y) & (\mathcal{I}b_0) \\ \mathsf{add}(\mathsf{s}^3(0), y) \to \mathsf{s}^3(y) & (\mathcal{I}b_1) \\ \mathsf{add}(\mathsf{s}^4(0), y) \to \mathsf{s}^4(y) & (\mathcal{I}^2b_0) \\ \mathsf{add}(\mathsf{s}^5(0), y) \to \mathsf{s}^5(y) & (\mathcal{I}^2b_1) \\ \mathsf{add}(\mathsf{s}^6(0), y) \to \mathsf{s}^6(y) & (\mathcal{I}^3b_0) \\ \mathsf{add}(\mathsf{s}^7(0), y) \to \mathsf{s}^7(y) & (\mathcal{I}^3b_1) \\ \mathsf{add}(\mathsf{s}^8(x), y) \to \mathsf{s}^8(z) \Leftarrow \mathsf{add}(x, y) \to z & (\mathcal{I}^4) \end{array} \right\}.$$

Then any applications of Fold rule in Step 4 are impossible, because all rules with the labels from (b_0) through $(\mathcal{I}^3b_1)$ have no conditional part, which are necessary in applying Fold rule. If the second rule of $\mathcal{R}_{\mathsf{add1}}$ were of the form

$$\mathsf{add}(\mathsf{s}(0), y) \to z \Leftarrow \mathsf{add}(0, y) \to z,$$

then the transformation would be successful. Since the former is obtained by simplifying the latter, this means that some necessary information in the conditional part may be lost by a simplification. One possibility to avoid this issue is recovering the information from the simpler rule such as the former. It is, however, difficult to find a clue. Instead of recovering the conditional part directly, we adopt an alternative that inserts a condition with transparent function id, which is identity function defined by

$$\mathsf{id}(x) \to x.$$

We introduce a conditional part with the identity function to make each right-hand side of body parts is a variable. The CTRS $\mathcal{R}_{\mathsf{add1}}$ is transformed into the following CTRS:

$$\mathcal{R}_{\mathsf{add1}'} = \left\{ \begin{array}{l} \mathsf{add}(0, y) \to y \\ \mathsf{add}(\mathsf{s}(0), y) \to w \ \Leftarrow \mathsf{id}(\mathsf{s}(y)) \to w \\ \mathsf{add}(\mathsf{s}^2(x), y) \to w \Leftarrow \mathsf{add}(x, y) \to z, \mathsf{id}(\mathsf{s}(z)) \to w_1, \mathsf{id}(\mathsf{s}(w_1)) \to w \end{array} \right\}.$$

This process is formalized as the following procedure.

▶ **Procedure 6.1** (id attachment). The id-attached rule of a pure-constructor rule $\rho : l \to r \Leftarrow c$ is constructed as follows.

1. If $r \notin \mathcal{V}$, then convert ρ into $\rho' : l \to z \Leftarrow c, \mathsf{id}(r) \to z$. Otherwise, let ρ' be ρ itself.
2. Replace a condition in ρ' of the form $\mathsf{id}(g(t_1, \ldots, t_i, \ldots, t_n)) \to v$ such that $t_i \notin \mathcal{V}$ with $\mathsf{id}(t_i) \to z_i, \mathsf{id}(g(t_1, \ldots, z_i, \ldots, t_n)) \to v$

Step 0	Apply Procedure 6.1 to each rule in the given CTRS $\mathcal{R}_1$.
Step 1	Give labels to each rule in the same way as Figure 2 (Step 1).
Step 2	Unfold rules in the same way as Figure 2 (Step 2). For each "Unfolded" rule, apply Procedure 6.1 to the generated rule.
Step 3–6	Do each step in the same way as Figure 2.

Figure 3 Modified heuristic procedure for Unfold^{-1} problem.

3. Repeat 2 until it can not be applicable.

Note that variables z and z_i above must be fresh.

Actually, this procedure must be applied after each Unfolding application during Step 2–3 in Figure 2 because conditions of the generated rule may disappear by Unfolding. The modified heuristics we propose is summarized in Figure 3.

▶ **Example 6.2.** We solve Unfold^{-1} problem for $\mathcal{R}_{\mathsf{add1}}$ applying the modified heuristic procedure.

Step 0: As described above, we obtain $\mathcal{R}_{\mathsf{add1}'}$ by Step 0.

Step 1–3: As a result of Step 1–3, we obtain

$$\mathcal{R}_{\mathsf{add2}} = \left\{ \begin{array}{ll}
\mathsf{add}(0,y) \to y & (b_0) \\
\mathsf{add}(\mathsf{s}(0),y) \to w \Leftarrow \mathsf{id}(\mathsf{s}(y)) \to w & (b_1) \\
\mathsf{add}(\mathsf{s}^2(0),y) \to w \Leftarrow \mathsf{id}(\mathsf{s}(y)) \to w_1; \mathsf{id}(\mathsf{s}(w_1)) \to w & (\mathcal{I}b_0) \\
\mathsf{add}(\mathsf{s}^3(0),y) \to w \Leftarrow \mathsf{id}(\mathsf{s}(y)) \to w_2; \mathsf{id}(\mathsf{s}(w_2)) \to w_1; \mathsf{id}(\mathsf{s}(w_1)) \to w & (\mathcal{I}b_1) \\
\mathsf{add}(\mathsf{s}^4(0),y) \to w \Leftarrow \mathsf{id}(\mathsf{s}(y)) \to w_3; \mathsf{id}(\mathsf{s}(w_3)) \to w_2; \cdots; \mathsf{id}(\mathsf{s}(w_1)) \to w & (\mathcal{I}^2b_0) \\
\mathsf{add}(\mathsf{s}^5(0),y) \to w \Leftarrow \mathsf{id}(\mathsf{s}(y)) \to w_4; \mathsf{id}(\mathsf{s}(w_4)) \to w_3; \cdots; \mathsf{id}(\mathsf{s}(w_1)) \to w & (\mathcal{I}^2b_1) \\
\mathsf{add}(\mathsf{s}^6(0),y) \to w \Leftarrow \mathsf{id}(\mathsf{s}(y)) \to w_5; \mathsf{id}(\mathsf{s}(w_5)) \to w_4; \cdots; \mathsf{id}(\mathsf{s}(w_1)) \to w & (\mathcal{I}^3b_0) \\
\mathsf{add}(\mathsf{s}^7(0),y) \to w \Leftarrow \mathsf{id}(\mathsf{s}(y)) \to w_6; \mathsf{id}(\mathsf{s}(w_6)) \to w_5; \cdots; \mathsf{id}(\mathsf{s}(w_1)) \to w & (\mathcal{I}^3b_1) \\
\mathsf{add}(\mathsf{s}^8(x),y) \to w \Leftarrow \mathsf{add}(x,y) \to z; \mathsf{id}(\mathsf{s}(z)) \to w_7; \cdots; \mathsf{id}(\mathsf{s}(w_1)) \to w; & (\mathcal{I}^4)
\end{array} \right\}.$$

Step 4: Fold transformations in Step 4 create the following rules.

$$\mathcal{R}_{\mathsf{add2}'} = \left\{ \begin{array}{ll}
\mathsf{add}(0,y) \to y & (b_0) \\
\mathsf{add}(\mathsf{s}(0),y) \to w \Leftarrow \mathsf{id}(\mathsf{s}(y)) \to w & (b_1) \\
\mathsf{add}(\mathsf{s}(\mathsf{s}(0)),y) \to w \Leftarrow \mathsf{add}(\mathsf{s}(0),y) \to w_1; \mathsf{id}(\mathsf{s}(w_1)) \to w & (\mathcal{I}b_0)' \\
\mathsf{add}(\mathsf{s}(\mathsf{s}^2(0)),y) \to w \Leftarrow \mathsf{add}(\mathsf{s}^2(0),y) \to w_1; \mathsf{id}(\mathsf{s}(w_1)) \to w; & (\mathcal{I}b_1)' \\
\mathsf{add}(\mathsf{s}(\mathsf{s}^3(0)),y) \to w \Leftarrow \mathsf{add}(\mathsf{s}^3(0),y) \to w_1; \mathsf{id}(\mathsf{s}(w_1)) \to w & (\mathcal{I}^2b_0)' \\
\mathsf{add}(\mathsf{s}(\mathsf{s}^4(0)),y) \to w \Leftarrow \mathsf{add}(\mathsf{s}^4(0),y) \to w_1; \mathsf{id}(\mathsf{s}(w_1)) \to w & (\mathcal{I}^2b_1)' \\
\mathsf{add}(\mathsf{s}(\mathsf{s}^5(0)),y) \to w \Leftarrow \mathsf{add}(\mathsf{s}^5(0),y) \to w_1; \mathsf{id}(\mathsf{s}(w_1)) \to w & (\mathcal{I}^3b_0)' \\
\mathsf{add}(\mathsf{s}(\mathsf{s}^6(0)),y) \to w \Leftarrow \mathsf{add}(\mathsf{s}^6(0),y) \to w_1; \mathsf{id}(\mathsf{s}(w_1)) \to w & (\mathcal{I}^3b_1)' \\
\mathsf{add}(\mathsf{s}^8(x),y) \to w \Leftarrow \mathsf{add}(x,y) \to z; \mathsf{id}(\mathsf{s}(z)) \to w_7; \cdots; & \\
\qquad \mathsf{id}(\mathsf{s}(w_1)) \to w; & (\mathcal{I}^4)
\end{array} \right\}.$$

Step 5: Generalize the divergent rules in $\mathcal{R}_{\mathsf{add2}'}$ with the labels from $(\mathcal{I}b_0)'$ through $(\mathcal{I}^3b_1)'$ by the difference matching, which yields a rule "$\mathsf{add}(\mathsf{s}(x),y) \to w \Leftarrow \mathsf{add}(x,y) \to w_1; \mathsf{id}(\mathsf{s}(w_1)) \to w$", which is equal to "$\mathsf{add}(\mathsf{s}(x),y) \to \mathsf{s}(w_1) \Leftarrow \mathsf{add}(x,y) \to w_1$". So a solution candidate of

Table 2 Examples of the modified heuristic procedure in Figure 3.

Given CTRS	Solution CTRS
$\left\{\begin{array}{l}\mathsf{add}(0,y)\to y\\ \mathsf{add}(\mathsf{s}(0),y)\to \mathsf{s}(y)\\ \mathsf{add}(\mathsf{s}^2(x),y)\to \mathsf{s}^2(z)\Leftarrow \mathsf{add}(x,y)\to z\end{array}\right\}$	$\left\{\begin{array}{l}\mathsf{add}(0,y)\to y\\ \mathsf{add}(\mathsf{s}(x),y)\to \mathsf{s}(z)\Leftarrow \mathsf{add}(x,y)\to z\end{array}\right\}$
$\left\{\begin{array}{l}\mathsf{frev}([\,],ys)\to ys\\ \mathsf{frev}(x:[\,],ys)\to x:ys\\ \mathsf{frev}(x_1:x_2:xs,ys)\to zs\\ \qquad\qquad\Leftarrow \mathsf{frev}(xs,x_2:x_1:ys)\to zs\end{array}\right\}$	$\left\{\begin{array}{l}\mathsf{frev}([\,],ys)\to ys\\ \mathsf{frev}(x:xs,ys)\to zs\\ \qquad\qquad\Leftarrow \mathsf{frev}(xs,x:ys)\to zs\end{array}\right\}$

Unfold^{-1} problem for $\mathcal{R}_{\mathsf{add1}}$ is

$$\mathcal{R}_{\mathsf{add}}=\left\{\begin{array}{l}\mathsf{add}(0,y)\to y\\ \mathsf{add}(\mathsf{s}(x),y)\to \mathsf{s}(z)\Leftarrow \mathsf{add}(x,y)\to z\end{array}\right\}.$$

Step 6: It is confirmed that $\mathcal{R}_{\mathsf{add}}$ Unfold $\mathcal{R}_{\mathsf{add1}}$. Thus $\mathcal{R}_{\mathsf{add}}$ is certainly a solution.

Table 2 shows the results by applying the modified heuristics to the failed problems in Table 1.

7 Application

In this section, we show an example induced from the program inversion[10, 15, 16, 17, 18]. Consider the following TRS $\mathcal{R}_{\mathsf{rev}'}$, which defines a fast reverse function of a list:

$$\mathcal{R}_{\mathsf{rev}'}=\left\{\begin{array}{l}\mathsf{reverse}(xs)\to \mathsf{frev}(xs,[\,])\\ \mathsf{frev}([\,],ys)\to ys\\ \mathsf{frev}(x:[\,],ys)\to x:ys\\ \mathsf{frev}(x_1:x_2:xs,ys)\to \mathsf{frev}(xs,x_2:x_1:ys)\end{array}\right\}.$$

Note that the definition of frev is convertible to an equivalent pure-constructor CTRS in the second column of Table 2. For TRS $\mathcal{R}_{\mathsf{rev}'}$, a program inversion tool *repius* [2] produces the following CTRS:

$$\mathcal{R}_{\mathsf{invrev}'}=\left\{\begin{array}{l}\mathsf{inv\text{-}reverse}(ys)\to \mathsf{tp1}(xs)\Leftarrow \mathsf{tinv\text{-}frev}([\,],ys)\to \mathsf{tp2}(xs,[\,]),\\ \mathsf{inv\text{-}reverse}(x:ys)\to \mathsf{tp1}(xs)\Leftarrow \mathsf{tinv\text{-}frev}(x:[\,],ys)\to \mathsf{tp2}(xs,[\,]),\\ \mathsf{tinv\text{-}frev}(xs,[\,])\to \mathsf{tp2}(xs,[\,]),\\ \mathsf{tinv\text{-}frev}(xs,x_2:x_1:ys)\to \mathsf{tinv\text{-}frev}(x_1:x_2:xs,ys).\end{array}\right\},$$

where $\mathsf{tp1}(\cdot)$ and $\mathsf{tp2}(\cdot,\cdot)$ are constructors introduced by *repius* for representing 1-tuple and 2-tuple, respectively. The function inv-reverse in $\mathcal{R}_{\mathsf{invrev}'}$ works as the inversion of frev, but non-determinacy in computation is necessary to obtain the expected results; the first rule should be applied to an odd-length list and the second rule to even-length list.

Next, we consider the following definition of frev, which is the solution CTRS in Table 2.

$$\mathcal{R}_{\mathsf{rev}}=\left\{\begin{array}{l}\mathsf{reverse}(xs)\to \mathsf{frev}(xs,[\,]),\\ \mathsf{frev}([\,],ys)\to ys,\\ \mathsf{frev}(x:xs,ys)\to \mathsf{frev}(xs,x:ys)\end{array}\right\}.$$

[2] http://www.trs.cm.is.nagoya-u.ac.jp/repius/

For this TRS, *repius* produces the following CTRS:

$$\mathcal{R}_{\mathsf{invrev}} = \left\{ \begin{array}{l} \mathsf{inv\text{-}reverse}(ys) \to \mathsf{tp1}(xs) \Leftarrow \mathsf{tinv\text{-}frev}([\,], ys) \to \mathsf{tp2}(xs, [\,]) \\ \mathsf{tinv\text{-}frev}(xs, [\,]) \to \mathsf{tp2}(xs, [\,]) \\ \mathsf{tinv\text{-}frev}(xs, x : ys) \to \mathsf{tinv\text{-}frev}(x : xs, ys) \end{array} \right\}.$$

The CTRS $\mathcal{R}_{\mathsf{invrev}}$ is left-linear and non-overlapping and hence non-determinacy is not necessary any more.

8 Conclusion

In this paper, we formalized the inverse problem of an one-step program transformation, and focused on inverse Unfold problem, which is simulation sound and complete. For this problem, we proposed a heuristic procedure and its improvement with the identity function. Using these heuristics, we have also shown some successful examples and an application example on program inversion.

As mentioned in Section 1, we used pure-constructor systems as a platform because of the firmness of the structure of the rules. However, the heuristics proposed in this paper may be modified for the general TRSs and unfoldings for them. Moreover, in that framework, id symbol in Section 6 might not be necessary. It is also interesting to consider this issue.

We should address the following future tasks.

Target: So far, the scope of heuristic solvings in this paper is limited to functions whose arguments consist of simple list-like data structures. Tree-like data structures and mutual recursive functions will be considered as targets. We should open the class of CTRSs in which our heuristics succeeds.

Completeness: We will find a subclass for which the heuristic procedure is complete; the procedure can always find a solution if it exists.

Mechanization: In our heuristics, there are multiple choices which rules to be unfolded/folded. Strategies to narrow down the options for automation is promising.

Acknowledgements. We would like to thank the anonymous referees for their corrections and valuable comments.

References

1 Jesús Manuel Almendros-Jiménez and Germán Vidal. Automatic partial inversion of inductively sequential functions. In Zoltán Horváth, Viktória Zsók, and Andrew Butterfield, editors, *Proc. of 18th International Symposium on Implementation and Application of Functional Languages*, volume 4449 of *Lecture Notes in Computer Science*, pages 253–270, 2007.

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

3 David Basin and Toby Walsh. Difference matching. In Deepak Kapur, editor, *11th International Conference on Automated Deduction*, volume 607 of *Lecture Notes in Computer Science*, pages 295–309. Springer, 1992.

4 N. Bensaou and Irène Guessarian. Transforming constraint logic programs. *Theoretical Computer Science*, 206(1-2):81–125, 1998.

5 Jan A. Bergstra and Jan Willem Klop. Conditional rewrite rules: Confluence and termination. *Journal of Computer and System Sciences*, 32(3):323–362, 1986.

6 Rod M. Burstall and John Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, 1977.

7 Sandro Etalle and Maurizio Gabbrielli. Transformations of CLP modules. *Theoretical Computer Science*, 166(1-2):101–146, 1996.

8 Fabio Fioravanti, Alberto Pettorossi, and Maurizio Proietti. Transformation rules for locally stratified constraint logic programs. In Maurice Bruynooghe and Kung-Kiu Lau, editors, *Program Development in Computational Logic*, volume 3049 of *Lecture Notes in Computer Science*, pages 77–89. Springer, 2004.

9 Harald Ganzinger. Order-sorted completion: The many-sorted way. *Theoretical Computer Science*, 89(1):3–32, 1991.

10 Robert Glück and Masahiko Kawabe. A method for automatic program inversion based on LR(0) parsing. *Fundamenta Informmaticae*, 66(4):367–395, 2005.

11 Tadashi Kanamori and Kenji Horiuchi. Construction of logic programs based on generalized unfold/fold rules. In Jean-Louis Lassez, editor, *Proceedings of the 4th International Conference on Logic Programming*, pages 744–768, 1987.

12 Michael J. Maher. A transformation system for deductive database modules with perfect model semantics. *Theoretical Computer Science*, 110(2):377–403, 1993.

13 Aart Middeldorp and Erik Hamoen. Completeness results for basic narrowing. *Applicable Algebra in Engineering, Communication and Computing*, 5:213–253, 1994.

14 Masanori Nagashima, Masahiko Sakai, and Toshiki Sakabe. Determinization of conditional term rewriting systems. *Theoretical Computer Science*, 464:72–89, 2012.

15 Naoki Nishida, Masahiko Sakai, and Toshiki Sakabe. Partial inversion of constructor term rewriting systems. In Jürgen Giesl, editor, *Proc. of the 16th Int'l Conf. on Rewriting Techniques and Applications*, volume 3467 of *LNCS*, pages 264–278. Springer, 2005.

16 Naoki Nishida and Germán Vidal. Program inversion for tail recursive functions. In Manfred Schmidt-Schauß, editor, *Proceedings of the 22nd International Conference on Rewriting Techniques and Applications*, volume 10 of *LIPIcs*, pages 283–298. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2011.

17 Naoki Nishida and Germán Vidal. Computing more specific versions of conditional rewriting systems. In Elvira Albert, editor, *Revised Selected Papers of the 22nd International Symposium on Logic-Based*, volume 7844 of *Lecture Notes in Computer Science*, pages 137–154, 2013.

18 Minami Niwa, Naoki Nishida, and Masahiko Sakai. Extending matching operation in grammar program for program inversion. In Elvira Albert, editor, *Informal Proceedings of the 22nd International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2012)*, pages 130–139, 2012.

19 Enno Ohlebusch. *Advanced Topics in Term Rewriting*. Springer-Verlag, 2002.

20 Alberto Pettorossi, Maurizio Proietti, and Valerio Senni. Constraint-based correctness proofs for logic program transformations. *Formal Aspects of Computing*, 24(4–6):569–594, 2012.

21 Abhik Roychoudhury, K. Narayan Kumar, C. R. Ramakrishnan, and I. V. Ramakrishnan. Beyond Tamaki-Sato style unfold/fold transformations for normal logic programs. *International Journal of Foundations of Computer Science*, 13(3):387–403, 2002.

22 Abhik Roychoudhury, K. Narayan Kumar, C. R. Ramakrishnan, and I. V. Ramakrishnan. An unfold/fold transformation framework for definite logic programs. *ACM Transactions on Programming Languages and Systems*, 26(3):464–509, 2004.

23 David Sands. Total correctness by local improvement in the transformation of functional programs. *ACM Trans. on Programming Languages and Systems*, 18(2):175–234, 1996.

24 Taisuke Sato. Equivalence-preserving first-order unfold/fold transformation systems. *Theoretical Computer Science*, 105(1):57–84, 1992.

25 Hirohisa Seki. Unfold/fold transformation of general logic programs for the well-founded semantics. *Journal of Logic Programming*, 16(1):5–23, 1993.

On Proving Soundness of the Computationally Equivalent Transformation for Normal Conditional Term Rewriting Systems by Using Unravelings*

Naoki Nishida[1], Makishi Yanagisawa[1], and Karl Gmeiner[2]

1 **Graduate School of Information Science, Nagoya University**
Furo-cho, Chikusa-ku, 4648603 Nagoya, Japan
nishida@is.nagoya-u.ac.jp, makishi@trs.cm.is.nagoya-u.ac.jp

2 **Institute of Computer Science, UAS Technikum Wien**
gmeiner@technikum-wien.at

Abstract

In this paper, we show that the SR transformation, a computationally equivalent transformation proposed by Şerbănuţă and Roşu, is sound for weakly left-linear normal conditional term rewriting systems (CTRS). Here, soundness for a CTRS means that reduction of the transformed unconditional term rewriting system (TRS) creates no undesired reduction for the CTRS. We first show that every reduction sequence of the transformed TRS starting with a term corresponding to the one considered on the CTRS is simulated by the reduction of the TRS obtained by the simultaneous unraveling. Then, we use the fact that the unraveling is sound for weakly left-linear normal CTRSs.

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

Keywords and phrases conditional term rewriting, unraveling, condition elimination

Digital Object Identifier 10.4230/OASIcs.WPTE.2014.39

1 Introduction

Conditional term rewriting is known to be much more complicated than unconditional term rewriting in the sense of analyzing properties, e.g., *operational termination* [14], *confluence* [23], *reachability* [5]. A popular approach to the analysis of conditional term rewriting systems (CTRS) is to transform a CTRS into an unconditional term rewriting system (TRS) that is an overapproximation of the CTRS in terms of reduction. This approach enables us to use techniques for the analysis of TRSs, which are well investigated in the literature. For example, if the transformed TRS is terminating, then the CTRS is operationally terminating [4] — to prove termination of the transformed TRS, we can use many termination proving techniques which have been well investigated for TRSs (cf. [19]). Another interesting application of the approach is the analysis of (un)reachability on CTRSs, especially unreachability for TRSs for, e.g., verifying *cryptographic protocol* [7]. Many techniques to construct *tree automata* [3] for accepting all the reachable ground terms for given (a recognizable set of) ground terms have been established (see, e.g., [12, 6, 24]), and thus, by transforming CTRSs into TRSs, we can use such techniques for TRSs to analyze (un)reachablity.

* The research in this paper is partly supported by the Austrian Science Fund (FWF) international project I963 and the Japan Society for the Promotion of Science (JSPS).

1st International Workshop on Rewriting Techniques for Program Transformations and Evaluation (WPTE'14).
Editors: Manfred Schmidt-Schauß, Masahiko Sakai, David Sabel, and Yuki Chiba; pp. 39–50

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

There are two approaches to transformations of CTRSs into TRSs: *unravelings* [15, 16] proposed by Marchiori (see, e.g., [8, 17]), and a transformation [25] proposed by Viry (see, e.g., [21, 8]).

Unravelings are transformations from a CTRS into a TRS over an extension of the original signature for the CTRS. They are *complete* for (reduction of) the CTRS [15], i.e., for every derivation of the CTRS, there exists a corresponding derivation of the unraveled TRS. In this respect, the unraveled TRS is an overapproximation of the CTRS w.r.t. reduction, and is useful for analyzing the properties of the CTRS, such as syntactic properties, modularity, and operational termination, since TRSs are in general much easier to handle than CTRSs.

The latest transformation based on Viry's approach is a *computationally equivalent* transformation proposed by Şerbănuţă and Roşu [21, 22], called *the SR transformation*. This converts a left-linear confluent normal CTRS into a TRS which is computationally equivalent to the CTRS. This means that the converted TRS can be used to exactly simulate reduction sequences of the CTRS to normal forms.

This paper aims at investigating sufficient conditions for *soundness* of the SR transformation w.r.t. reduction. Neither any unraveling nor the SR transformation is sound for (reduction of) all CTRSs. Here, soundness for a CTRS means that reduction of the converted TRS creates no undesired reduction for the CTRS. Since soundness is one of the most important properties for transformations of CTRSs, sufficient conditions for soundness have been well investigated, especially for unravelings (see, e.g., [9, 17, 10]). For example, the *simultaneous unraveling* [15], which is proposed by Marchiori (and then improved by Ohlebusch [18]), is sound for *weakly left-linear*, *confluent*, *non-erasing*, or *ground conditional* normal CTRSs [9].

As for unravelings, soundness of the SR transformation plays a very important role for, e.g., computational equivalence. The main purpose of transformations along the Viry's approach is to use the soundly transformed TRS to simulate the reduction of the original CTRS. The experimental results in [21] indicate that the rewriting engine using the soundly transformed TRS is much more efficient than the one using the original left-linear confluent CTRS. However, unlike unravelings, soundness conditions for the SR transformation have not been investigated well, and the known conditions are left-linearity or confluence of CTRSs [21, 22]. To get an efficient rewriting engine for CTRSs, soundness conditions for the SR transformation are worth investigating.

To clarify the relationship between unravelings and the SR transformation in terms of soundness, it has been shown that if the SR transformation is sound for a CTRS, then so is the corresponding unraveling [17]. This is not so surprising since the SR transformation is more powerful than unravelings in terms of evaluating conditions in parallel. For the same reason, however, it is not so easy to prove the converse of the above claim — as shown later, the converse does not hold for all normal CTRSs.

In this paper, we show that the SR transformation is sound for *weakly left-linear* normal CTRSs. To this end, we first show that every reduction sequence of the transformed TRS starting with a term corresponding to the one considered on the CTRS is simulated by the reduction of the unraveled TRS obtained by the simultaneous unraveling [15, 18]. Then, we use the fact that the unraveling is sound for weakly left-linear normal CTRSs. One of the reasons why we take this approach is to avoid conditional rewriting in proofs for soundness.

As already described, unravelings are nice tools to analyze properties of CTRSs, and the SR transformation is a nice tool to get a computationally equivalent TRS which provides very efficient computation compared to the one on the original CTRS. For this reason, we do not discuss the usefulness of our results for analyzing properties of CTRSs, and we concentrate on soundness conditions of the SR transformation.

This paper is organized as follows. In Section 2, we briefly recall basic notations of term rewriting. In Section 3, we recall the notion of soundness, the simultaneous unraveling, and the SR transformation for normal CTRSs. We will adopt a slightly different formulation of the SR transformation from the original one [21], while the resulting TRSs are the same. In Section 4, we show that the SR transformation is sound for weakly left-linear normal CTRSs. In Section 5, we conclude this paper and describe future work on this research.

2 Preliminaries

In this section, we recall basic notions and notations of term rewriting [2, 19].

Throughout the paper, we use $\mathcal{V}$ as a countably infinite set of *variables*. Let $\mathcal{F}$ be a *signature*, a finite set of *function symbols* each of which has its own fixed arity, and $\mathtt{arity}_{\mathcal{F}}(\mathsf{f})$ be the arity of function symbol f. We often write $\mathsf{f}/n \in \mathcal{F}$ instead of "$\mathsf{f} \in \mathcal{F}$ and $\mathtt{arity}_{\mathcal{F}}(\mathsf{f}) = n$". The set of *terms* over F ($\subseteq \mathcal{F}$) and V ($\subseteq \mathcal{V}$) is denoted by $T(F, V)$, and the set of variables appearing in any of the terms $t_1, \ldots, t_n$ is denoted by $\mathcal{V}ar(t_1, \ldots, t_n)$. A term t is called *ground* if $\mathcal{V}ar(t) = \emptyset$. A term is called *linear* if any variable occurs in the term at most once, and called *linear w.r.t. a variable* if the variable appears at most once in t. The function symbol at the *root* position ε of term t is denoted by $\mathtt{root}(t)$. Given an n-hole *context* $C[\ ]$ with parallel positions $p_1, \ldots, p_n$, the notation $C[t_1, \ldots, t_n]_{p_1,\ldots,p_n}$ represents the term obtained by replacing hole $\Box$ at position p_i with term t_i for all $1 \leq i \leq n$. We may omit the subscript "$p_1, \ldots, p_n$" from $C[\ldots]_{p_1,\ldots,p_n}$. For positions p and p' of a term, we write $p' \geq p$ if p is a prefix of p' (i.e., there exists a sequence q such that $pq = p'$). Moreover, we write $p' > p$ if p is a proper prefix of p'.

The *domain* and *range* of a *substitution* σ are denoted by $\mathcal{D}om(\sigma)$ and $\mathcal{R}an(\sigma)$, respectively. We may denote σ by $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ if $\mathcal{D}om(\sigma) = \{x_1, \ldots, x_n\}$ and $\sigma(x_i) = t_i$ for all $1 \leq i \leq n$. For F ($\subseteq \mathcal{F}$) and V ($\subseteq \mathcal{V}$), the set of *substitutions* that range over F and V is denoted by $\mathcal{S}ub(F, V)$: $\mathcal{S}ub(F, V) = \{\sigma \mid \mathcal{R}an(\sigma) \subseteq T(F, V)\}$. For a substitution σ and a term t, the application $\sigma(t)$ of σ to t is abbreviated to $t\sigma$, and $t\sigma$ is called an *instance* of t. Given a set X of variables, $\sigma|_X$ denotes the *restricted* substitution of σ w.r.t. X: $\sigma|_X = \{x \mapsto x\sigma \mid x \in \mathcal{D}om(\sigma) \cap X\}$.

An (oriented) *conditional rewrite rule* over a signature $\mathcal{F}$ is a triple (l, r, c), denoted by $l \to r \Leftarrow c$, such that the *left-hand side* l is a non-variable term in $\mathcal{T}(\mathcal{F}, \mathcal{V})$, the *right-hand side* r is a term in $\mathcal{T}(\mathcal{F}, \mathcal{V})$, and the *conditional part* c is a sequence $s_1 \twoheadrightarrow t_1; \ldots; s_k \twoheadrightarrow t_k$ of term pairs ($k \geq 0$) where all of $s_1, t_1, \ldots, s_k, t_k$ are terms in $T(\mathcal{F}, \mathcal{V})$. In particular, a conditional rewrite rule is called *unconditional* if the conditional part is the empty sequence (i.e., $k = 0$), and we may abbreviate it to $l \to r$. We sometimes attach a unique label ρ to the conditional rewrite rule $l \to r \Leftarrow c$ by denoting $\rho : l \to r \Leftarrow c$, and we use the label to refer to the rewrite rule.

An *(oriented) conditional term rewriting system* (CTRS) over a signature $\mathcal{F}$ is a set of conditional rules over $\mathcal{F}$. A CTRS is called an (unconditional) *term rewriting system* (TRS) if every rule $l \to r \Leftarrow c$ in the CTRS is unconditional and satisfies $\mathcal{V}ar(l) \supseteq \mathcal{V}ar(r)$. The *reduction relation* of a CTRS $\mathcal{R}$ is defined as $\to_{\mathcal{R}} = \bigcup_{n \geq 0} \to_{(n),\mathcal{R}}$ where $\to_{(0),\mathcal{R}} = \emptyset$, and $\to_{(i+1),\mathcal{R}} = \{(C[l\sigma]_p, C[r\sigma]_p) \mid \rho : l \to r \Leftarrow s_1 \twoheadrightarrow t_1; \ldots; s_k \twoheadrightarrow t_k \in \mathcal{R},\ s_1\sigma \to^*_{(i),\mathcal{R}} t_1\sigma,\ \ldots,\ s_k\sigma \to^*_{(i),\mathcal{R}} t_k\sigma\}$ for $i \geq 0$. To specify the applied rule ρ and the position p where ρ is applied, we may write $\to_{p,\rho}$ or $\to_{p,\mathcal{R}}$ instead of $\to_{\mathcal{R}}$. Moreover, we may write $\to_{>\varepsilon,\mathcal{R}}$ instead of $\to_{p,\mathcal{R}}$ if $p > \varepsilon$. The *parallel reduction* $\rightrightarrows_{\mathcal{R}}$ is defined as follows: $\rightrightarrows_{\mathcal{R}} = \{(C[s_1, \ldots, s_n]_{p_1,\ldots,p_n}, C[t_1, \ldots, t_n]_{p_1,\ldots,p_n}) \mid s_1 \to_{\mathcal{R}} t_1,\ \ldots,\ s_n \to_{\mathcal{R}} t_n\}$. To specify the positions $p_1, \ldots, p_n$ in the definition, we may write $\rightrightarrows_{\{p_1,\ldots,p_n\},\mathcal{R}}$ instead of $\rightrightarrows_{\mathcal{R}}$, and we may

write $\rightrightarrows_{>\varepsilon,\mathcal{R}}$ instead of $\rightrightarrows_{\mathcal{R}}$ if $p_i > \varepsilon$ for all $1 \leq i \leq n$. We denote n-step parallel reduction by $\rightrightarrows^n_{\mathcal{R}}$, and for $m \geq n$, we may write $\rightrightarrows^{\leq m}_{\mathcal{R}}$ instead of $\rightrightarrows^n_{\mathcal{R}}$.

A conditional rewrite rule $l \to r \Leftarrow c$ is called *left-linear* if l is linear, *right-linear* if r is linear, *non-erasing* if $\mathcal{V}ar(l) \subseteq \mathcal{V}ar(r)$, and *ground conditional* if c contains no variable. For a syntactic property P of conditional rewrite rules, we say that a CTRS has the property P if all of its rules have the property P, e.g., a CTRS is called *left-linear* if all of its rules are left-linear.

A conditional rewrite rule $\rho : l \to r \Leftarrow s_1 \twoheadrightarrow t_1; \ldots; s_k \twoheadrightarrow t_k$ is called *normal* if $\mathcal{V}ar(s_1, \ldots, s_k) \subseteq \mathcal{V}ar(l)$ and $t_1, \ldots, t_n$ are normal forms w.r.t. the *underlying unconditional system* $\mathcal{R}_u = \{l \to r \mid l \to r \Leftarrow c \in \mathcal{R}\}$. A CTRS is called *normal* (or a *normal CTRS*) if every rewrite rule of the CTRS is normal. Note that we consider *1-CTRSs* (i.e., $\mathcal{V}ar(l) \supseteq \mathcal{V}ar(r)$ for $l \to r \Leftarrow c$). A normal CTRS $\mathcal{R}$ is called *weakly left-linear* [9] if every conditional rewrite rule having at least one condition is left-linear, and for every unconditional rule, any non-linear variable in the left-hand side does not occur in the right-hand side.

Let $\mathcal{R}$ be a CTRS over a signature $\mathcal{F}$. The sets of *defined symbols* and *constructors* of $\mathcal{R}$ are denoted by $\mathcal{D}_{\mathcal{R}}$ and $\mathcal{C}_{\mathcal{R}}$, respectively: $\mathcal{D}_{\mathcal{R}} = \{\mathtt{root}(l) \mid l \to r \Leftarrow c \in \mathcal{R}\}$ and $\mathcal{C}_{\mathcal{R}} = \mathcal{F} \setminus \mathcal{D}_{\mathcal{R}}$. Terms in $T(\mathcal{C}_{\mathcal{R}}, \mathcal{V})$ are *constructor terms of* $\mathcal{R}$. $\mathcal{R}$ is called a *constructor system* if all proper subterms of the left-hand sides in $\mathcal{R}$ are constructor terms of $\mathcal{R}$.

3 Transformations from Normal CTRSs into TRSs

In this section, we recall the notion of *soundness*, the *simultaneous unraveling* [19], the SR transformation [21] for normal CTRSs.

We first show a general notion of soundness of completeness between two (C)TRSs (see [8, 17]). Let $\mathcal{R}_1$ and $\mathcal{R}_2$ be (C)TRSs over signature $\mathcal{F}_1$ and $\mathcal{F}_2$, respectively, ϕ be an *initialization* (total) mapping from $T(\mathcal{F}_1, \mathcal{V})$ to $T(\mathcal{F}_2, \mathcal{V})$, and ψ be a partial inverse of ϕ, a *backtranslation* mapping from $T(\mathcal{F}_2, \mathcal{V})$ to $T(\mathcal{F}_1, \mathcal{V})$ such that $\psi(\phi(t_1)) = t_1$ for any term $t_1 \in T(\mathcal{F}_1, \mathcal{V})$. We say that

- $\mathcal{R}_2$ is *sound for* (*reduction of*) $\mathcal{R}_1$ *w.r.t.* (ϕ, ψ) if, for any term $t_1 \in T(\mathcal{F}_1, \mathcal{V})$ and for any term $t_2 \in T(\mathcal{F}_2, \mathcal{V})$, $\phi(t_1) \to^*_{\mathcal{R}_2} t_2$ implies $t_1 \to^*_{\mathcal{R}_1} \psi(t_2)$ whenever $\psi(t_2)$ is defined, and
- $\mathcal{R}_2$ is *complete for* (*reduction of*) $\mathcal{R}_1$ *w.r.t.* (ϕ, ψ) if, for all terms $t_1, t_1' \in T(\mathcal{F}_1, \mathcal{V})$, $t_1 \to^*_{\mathcal{R}_1} t_1'$ implies $\phi(t_1) \to^*_{\mathcal{R}_2} \phi(t_1)$.

For the sake of readability, we restrict our interest to CTRSs, any rule of which has at most one condition. Note that this is not a restriction on the results in this paper (see [21]). We often denote a sequence $t_i, t_{i+1}, \ldots, t_j$ of terms by $\boldsymbol{t}_{i..j}$. Moreover, for the application of a mapping f to $\boldsymbol{t}_{i..j}$, we denote $f(t_i), \ldots, f(t_j)$ by $f(\boldsymbol{t}_{i..j})$, e.g., for a substitution θ, we denote $t_i\theta, \ldots, t_j\theta$ by $\theta(\boldsymbol{t}_{i..j})$.

3.1 Simultaneous Unraveling

The *simultaneous unraveling* for normal CTRSs, which is reformulated by Ohlebusch, is defined as follows.

▶ **Definition 1** ($\mathbb{U}$ [19])**.** Let $\mathcal{R}$ be a normal CTRS over a signature $\mathcal{F}$. Then,

$$\mathbb{U}(\rho : l \to r \Leftarrow s \twoheadrightarrow t) = \{\ \ l \to U_\rho(s, \boldsymbol{x}_{1..n}),\ \ U_\rho(t, \boldsymbol{x}_{1..n}) \to r\ \ \}$$

where $\{x_1, \ldots, x_n\} = \mathcal{V}ar(l)$ and U_ρ is a fresh $n+1$-ary function symbol, called a *U symbol*. Note that for every unconditional rule $l \to r \in \mathcal{R}$, $\mathbb{U}(l \to r) = \{l \to r\}$. $\mathbb{U}$ is straightforwardly

extended to normal CTRSs: $\mathbb{U}(\mathcal{R}) = \bigcup_{\rho\in\mathcal{R}} \mathbb{U}(\rho)$. We abuse $\mathbb{U}$ to represent the extended signature of $\mathcal{F}$: $\mathbb{U}_{\mathcal{R}}(\mathcal{F}) = \mathcal{F} \cup \{U_\rho \mid \rho : l \to r \Leftarrow s \twoheadrightarrow t \in \mathcal{R}\}$. We omit $\mathcal{R}$ from $\mathbb{U}_{\mathcal{R}}(\mathcal{F})$. Note that $\mathbb{U}(\mathcal{R})$ is a TRS over $\mathbb{U}(\mathcal{F})$. We say that $\mathbb{U}$ (and also $\mathbb{U}(\mathcal{R})$) is *sound* (*complete*) *for* $\mathcal{R}$ if $\mathbb{U}(\mathcal{R})$ is sound (complete) for $\mathcal{R}$ w.r.t. $(\mathtt{id}, \mathtt{id})$, where $\mathtt{id}$ is the identity mapping for $T(\mathcal{F}, \mathcal{V})$.

Note that $\mathbb{U}$ is complete for all normal CTRSs [15, 18].

▶ **Example 2.** Consider the following normal CTRS, a simplified variant of the one in [21]:

$$\mathcal{R}_1 = \{\ \mathsf{e}(\mathsf{0}) \to \mathsf{true},\quad \mathsf{e}(\mathsf{s}(x)) \to \mathsf{true} \Leftarrow \mathsf{e}(x) \twoheadrightarrow \mathsf{false},\quad \mathsf{e}(\mathsf{s}(x)) \to \mathsf{false} \Leftarrow \mathsf{e}(x) \twoheadrightarrow \mathsf{true}\ \}$$

$\mathcal{R}_1$ is unraveled to the following TRS:

$$\mathbb{U}(\mathcal{R}_1) = \left\{ \begin{array}{lll} \mathsf{e}(\mathsf{0}) \to \mathsf{true}, & \mathsf{e}(\mathsf{s}(x)) \to \mathsf{u}_1(\mathsf{e}(x), x), & \mathsf{u}_1(\mathsf{false}, x) \to \mathsf{true}, \\ & \mathsf{e}(\mathsf{s}(x)) \to \mathsf{u}_2(\mathsf{e}(x), x), & \mathsf{u}_2(\mathsf{true}, x) \to \mathsf{false} \end{array} \right\}$$

$\mathbb{U}(\mathcal{R}_1)$ is not confluent, while $\mathcal{R}_1$ is confluent. This means that $\mathbb{U}$ does not always preserve confluence of CTRSs. The reason why $\mathbb{U}$ loses confluence is that once we start evaluating a condition, the expected goal for the condition is fixed and then we cannot cancel the evaluation. This is illustrated in the derivation $\mathsf{e}(\mathsf{s}(\mathsf{0})) \to_{\mathbb{U}(\mathcal{R}_1)} \mathsf{u}_1(\mathsf{e}(\mathsf{0}), \mathsf{0}) \to_{\mathbb{U}(\mathcal{R}_1)} \mathsf{u}_1(\mathsf{true}, \mathsf{0})$. To reduce $\mathsf{e}(\mathsf{s}(\mathsf{0}))$ to false, we should have applied $\mathsf{e}(\mathsf{s}(x)) \to \mathsf{u}_2(\mathsf{e}(x), x) \in \mathbb{U}(\mathcal{R}_1)$ to the initial term. However, we applied another wrong rule, u_1 expects $\mathsf{e}(\mathsf{0})$ to be reduced to true (the expected goal for u_1 at this point), and we cannot redo applying the desired rule.

To simplify the discussion, we do not consider any optimization of unravelings (see e.g. [11]).

As shown in [15], $\mathbb{U}$ is not sound for every normal CTRS (see also [19, Example 7.2.14]). For some classes of normal CTRSs, $\mathbb{U}$ is sound (cf. [9, 17]).

▶ **Theorem 3** ([9])**.** *$\mathbb{U}$ is sound for a normal CTRS satisfying at least one of the following: weak left-linearity, confluence, non-erasingness, or ground conditional.*

3.2 The SR Transformation

Next, we introduce the SR transformation and its properties. In the following, the word "conditional rule" is used for representing rules having exactly one condition.

Before transforming a CTRS $\mathcal{R}$, we first extend the signature of $\mathcal{R}$ as follows:

- we leave constructors of $\mathcal{R}$ without any change,
- the arity n of defined symbol f is extended to $n + m$ where f has m conditional rules in $\mathcal{R}$, and we replace f by $\overline{\mathsf{f}}$ with the arity $n + m$, and
- a fresh constant $\bot$ and a fresh unary symbol $\langle\cdot\rangle$ are introduced.

We denote the extended signature by $\overline{\mathcal{F}}$: $\overline{\mathcal{F}} = \{\mathsf{c} \mid \mathsf{c} \in \mathcal{C}_{\mathcal{R}}\} \cup \{\overline{\mathsf{f}} \mid \mathsf{f} \in \mathcal{D}_{\mathcal{R}}\} \cup \{\bot, \langle\cdot\rangle\}$. We introduce a mapping $\mathtt{ext}(\cdot)$ to extend the arguments of defined symbols in a term as follows: $\mathtt{ext}(x) = x$ for $x \in \mathcal{V}$; $\mathtt{ext}(\mathsf{c}(\boldsymbol{t}_{1..n})) = \mathsf{c}(\mathtt{ext}(\boldsymbol{t}_{1..n}))$ for $\mathsf{c}/n \in \mathcal{C}_{\mathcal{R}}$; $\mathtt{ext}(\mathsf{f}(\boldsymbol{t}_{1..n})) = \overline{\mathsf{f}}(\mathtt{ext}(\boldsymbol{t}_{1..n}), \boldsymbol{z}_{1..m})$ for $\mathsf{f}/n \in \mathcal{D}_{\mathcal{R}}$, where f has m conditional rules in $\mathcal{R}$, $\mathtt{arity}_{\overline{\mathcal{F}}}(\overline{\mathsf{f}}) = n + m$, and $z_1, \ldots, z_m$ are fresh variables. The extended arguments of $\overline{\mathsf{f}}$ are used for evaluating the corresponding conditions, and the fresh constant $\bot$ is introduced to the extended arguments of defined symbols, which does not store any evaluation. To put $\bot$ into the extended arguments, we define a mapping $(\cdot)^\bot$ which puts $\bot$ to all the extended arguments of defined symbols, as follows: $(x)^\bot = x$ for $x \in \mathcal{V}$; $(\mathsf{c}(\boldsymbol{t}_{1..n}))^\bot = \mathsf{c}((\boldsymbol{t}_{1..n})^\bot)$ for $\mathsf{c} \in \mathcal{C}_{\mathcal{R}}$; $(\overline{\mathsf{f}}(\boldsymbol{t}_{1..n}, \boldsymbol{u}_{1..m}))^\bot = \overline{\mathsf{f}}((\boldsymbol{t}_{1..n})^\bot, \bot, \ldots, \bot)$ for $\mathsf{f} \in \mathcal{D}_{\mathcal{R}}$; $(\langle t\rangle)^\bot = \langle (t)^\bot\rangle$; $(\bot)^\bot = \bot$. Note that in applying $(\cdot)^\bot$ to *reachable terms* defined later, the case of applying $(\cdot)^\bot$ to $\bot$ never happens. Now we define a mapping $\overline{\cdot}$ from $T(\mathcal{F}, \mathcal{V})$ to $T(\overline{\mathcal{F}}, \mathcal{V})$ as $\overline{t} = (\mathtt{ext}(t))^\bot$.

The SR transformation [21] is defined as follows.

▶ **Definition 4** ($\mathbb{SR}$)**.** Let $\mathsf{f}/n \in \mathcal{D}_{\mathcal{R}}$ that has m conditional rules in $\mathcal{R}$ (i.e., $\overline{\mathsf{f}}/(n+m) \in \overline{\mathcal{F}}$). Then, $\mathbb{SR}(\mathsf{f}(\boldsymbol{w}_{1..n}) \to r) = \{\, \overline{\mathsf{f}}(\mathtt{ext}(\boldsymbol{w}_{1..n}), \boldsymbol{z}_{1..m}) \to \langle \overline{r} \rangle \,\}$ and, for the i-th conditional rule of f,

$$\mathbb{SR}(\mathsf{f}(\boldsymbol{w}_{1..n}) \to r_i \Leftarrow s_i \twoheadrightarrow t_i) = \left\{ \begin{array}{l} \overline{\mathsf{f}}(\boldsymbol{w'}_{1..n}, \boldsymbol{z}_{1..i-1},\ \bot,\ \boldsymbol{z}_{i+1..m}) \to \overline{\mathsf{f}}(\boldsymbol{w'}_{1..n}, \boldsymbol{z}_{1..i-1}, \langle s_i \rangle, \boldsymbol{z}_{i+1..m}), \\ \overline{\mathsf{f}}(\boldsymbol{w'}_{1..n}, \boldsymbol{z}_{1..i-1}, \langle t_i \rangle, \boldsymbol{z}_{i+1..m}) \to \langle \overline{r_i} \rangle \end{array} \right\}$$

where $\boldsymbol{w'}_{1..n} = \mathtt{ext}(\boldsymbol{w}_{1..n})$ and $z_1, \ldots, z_m$ are fresh variables. The set of auxiliary rules is defined as follows:

$$\begin{array}{rl} \mathcal{R}_{aux} = & \{\ \langle\langle x \rangle\rangle \to \langle x \rangle\ \} \cup \{\ \mathsf{c}(\boldsymbol{x}_{1..i-1}, \langle x_i \rangle, \boldsymbol{x}_{i+1..n}) \to \langle \mathsf{c}(\boldsymbol{x}_{1..n}) \rangle \mid \mathsf{c}/n \in \mathcal{C}_{\mathcal{R}},\ 1 \le i \le n\ \} \\ & \cup \{\ \overline{\mathsf{f}}(\boldsymbol{x}_{1..i-1}, \langle x_i \rangle, \boldsymbol{x}_{i+1..n}, \boldsymbol{z}_{1..m}) \to \langle \overline{\mathsf{f}}(\boldsymbol{x}_{1..n}, \bot, \ldots, \bot) \rangle \mid \mathsf{f}/n \in \mathcal{D}_{\mathcal{R}},\ 1 \le i \le n\ \} \end{array}$$

where $z_1, \ldots, z_m$ are fresh variables. The transformation $\mathbb{SR}$ is defined as follows: $\mathbb{SR}(\mathcal{R}) = \bigcup_{\rho \in \mathcal{R}} \mathbb{SR}(\rho) \cup \mathcal{R}_{aux}$. Note that $\mathbb{SR}(\mathcal{R})$ is a TRS over $\overline{\mathcal{F}}$. Note also that $\mathcal{R}_{aux}$ is linear. The backtranslation mapping $\widehat{\cdot}$ for $\overline{\cdot}$ is defined as follows: $\widehat{x} = x$ for $x \in \mathcal{V}$; $\widehat{\mathsf{c}(\boldsymbol{t}_{1..n})} = \mathsf{c}(\widehat{\boldsymbol{t}_{1..n}})$ for $\mathsf{c}/n \in \mathcal{C}_{\mathcal{R}}$; $\widehat{\overline{\mathsf{f}}(\boldsymbol{t}_{1..n}, \boldsymbol{u}_{1..m})} = \mathsf{f}(\widehat{\boldsymbol{t}_{1..n}})$ for $\mathsf{f}/n \in \mathcal{D}_{\mathcal{R}}$; $\widehat{\langle t \rangle} = \widehat{t}$; $\widehat{\bot} = \bot$. Note that $\widehat{\cdot}$ is a total function. A term t in $T(\overline{\mathcal{F}}, \mathcal{V})$ is called *reachable* if there exists a term $s \in T(\mathcal{F}, \mathcal{V})$ such that $\langle \overline{s} \rangle \to^*_{\mathbb{SR}(\mathcal{R})} t$. We say that $\mathbb{SR}$ (and also $\mathbb{SR}(\mathcal{R})$) is *sound* (*complete*) *for* $\mathcal{R}$ if $\mathbb{SR}(\mathcal{R})$ is sound (complete) for $\mathcal{R}$ w.r.t. $(\overline{\cdot}, \widehat{\cdot})$.

Note that $\mathbb{SR}$ is complete for all CTRSs [21]. Note also that $\mathbb{SR}$ is not sound for all normal CTRSs since for any normal CTRS $\mathcal{R}$, $\mathbb{SR}(\mathcal{R})$ can simulate any reduction of $\mathbb{U}(\mathcal{R})$ [17] — roughly speaking, any undesired derivation on $\mathbb{U}(\mathcal{R})$ holds on $\mathbb{SR}(\mathcal{R})$. It is clear that for any reachable term $t \in T(\overline{\mathcal{F}}, \mathcal{V})$, any term $t' \in T(\overline{\mathcal{F}}, \mathcal{V})$ with $t \to^*_{\mathbb{SR}(\mathcal{R})} t'$ is reachable.

To evaluate the condition of the i-th conditional rule $\mathsf{f}(\boldsymbol{w}_{1..n}) \to r_i \Leftarrow s_i \twoheadrightarrow t_i$, the i-th conditional rule is transformed into the two unconditional rules: the first one starts to evaluate the condition (an instance of s_i), and the second examines whether the condition holds. The first rule $\langle\langle x \rangle\rangle \to \langle x \rangle$ in $\mathcal{R}_{aux}$ removes the nest of $\langle\cdot\rangle$, the second rule $\mathsf{c}(\boldsymbol{x}_{1..i-1}, \langle x_i \rangle, \boldsymbol{x}_{i+1..n}) \to \langle \mathsf{c}(\boldsymbol{x}_{1..n}) \rangle$ is used for shifting $\langle\cdot\rangle$ upward, and the third rule $\overline{\mathsf{f}}(\boldsymbol{x}_{1..i-1}, \langle x_i \rangle, \boldsymbol{x}_{i+1..n}, \boldsymbol{z}_{1..m}) \to \langle \overline{\mathsf{f}}(\boldsymbol{x}_{1..n}, \bot, \ldots, \bot) \rangle$ is used for both shifting $\langle\cdot\rangle$ upward and resetting the evaluation of conditions at the extended arguments of $\overline{\mathsf{f}}$. The unary symbol $\langle\cdot\rangle$ and its rules in $\mathcal{R}_{aux}$ are introduced to preserve confluence of normal CTRSs on reachable terms (see [21] for the detail of the role of $\langle\cdot\rangle$ and its rules).

▶ **Example 5.** Consider $\mathcal{R}_1$ in Example 2 again. $\mathcal{R}_1$ is transformed by $\mathbb{SR}$ as follows:

$$\mathbb{SR}(\mathcal{R}_1) = \left\{ \begin{array}{l} \overline{\mathsf{e}}(\mathsf{0}, z_1, z_2) \to \langle \mathsf{true} \rangle, \\ \overline{\mathsf{e}}(\mathsf{s}(x), \bot, z_2) \to \overline{\mathsf{e}}(\mathsf{s}(x), \langle \overline{\mathsf{e}}(x, \bot, \bot) \rangle, z_2), \quad \overline{\mathsf{e}}(\mathsf{s}(x), \langle \mathsf{false} \rangle, z_2) \to \langle \mathsf{true} \rangle, \\ \overline{\mathsf{e}}(\mathsf{s}(x), z_1, \bot) \to \overline{\mathsf{e}}(\mathsf{s}(x), z_1, \langle \overline{\mathsf{e}}(x, \bot, \bot) \rangle), \quad \overline{\mathsf{e}}(\mathsf{s}(x), z_1, \langle \mathsf{true} \rangle) \to \langle \mathsf{false} \rangle, \\ \langle\langle x \rangle\rangle \to \langle x \rangle, \qquad \mathsf{s}(\langle x \rangle) \to \langle \mathsf{s}(x) \rangle, \qquad \overline{\mathsf{e}}(\langle x \rangle, z_1, z_2) \to \langle \overline{\mathsf{e}}(x, \bot, \bot) \rangle \end{array} \right\}$$

In contrast to $\mathbb{U}$, $\mathbb{SR}$ preserves confluence of CTRSs as confluence on reachable terms, e.g., $\mathbb{SR}(\mathcal{R}_1)$ is confluent on reachable terms, while $\mathbb{U}(\mathcal{R}_1)$ is not. Note that $\mathbb{SR}(\mathcal{R}_1)$ is not confluent. Let us consider the derivation starting from $\mathsf{e}(\mathsf{s}(\mathsf{0}))$ in Example 2 again. The corresponding derivation on $\mathbb{SR}(\mathcal{R}_1)$ is illustrated as follows:

$$\overline{\mathsf{e}}(\mathsf{s}(\mathsf{0}), \bot, \bot) \ \to_{\mathbb{SR}(\mathcal{R}_1)}\ \overline{\mathsf{e}}(\mathsf{s}(\mathsf{0}), \langle \overline{\mathsf{e}}(\mathsf{0}, \bot, \bot) \rangle, \bot) \ \to^*_{\mathbb{SR}(\mathcal{R}_1)}\ \overline{\mathsf{e}}(\mathsf{s}(\mathsf{0}), \langle \mathsf{true} \rangle, \bot)$$

Unlike the case of $\mathbb{U}(\mathcal{R}_1)$, we can apply the desired rule to the last term above:

$$\cdots \to_{\mathbb{SR}(\mathcal{R}_1)}\ \overline{\mathsf{e}}(\mathsf{s}(\mathsf{0}), \langle \mathsf{true} \rangle, \langle \overline{\mathsf{e}}(\mathsf{0}, \bot, \bot) \rangle) \ \to^*_{\mathbb{SR}(\mathcal{R}_1)}\ \overline{\mathsf{e}}(\mathsf{s}(\mathsf{0}), \langle \mathsf{true} \rangle, \langle \mathsf{true} \rangle) \ \to_{\mathbb{SR}(\mathcal{R}_1)}\ \langle \mathsf{false} \rangle$$

In the case of $\mathbb{U}$, to reach false, we need to backtrack from the undesired normal form $\mathsf{u}_1(\mathsf{true}, 0)$ (see Example 2), but in the case of $\mathbb{SR}$, we do not have to backtrack — choosing an arbitrary redex from reducible terms is sufficient to reach a desired normal form since $\mathbb{SR}(\mathcal{R}_1)$ is confluent on reachable terms.

Finally, we recall some important properties of $\mathbb{SR}$.

▶ **Theorem 6** ([21]). *$\mathbb{SR}$ is sound for left-linear or confluent[1] normal CTRSs.*

▶ **Theorem 7** ([17]). *If $\mathbb{SR}$ is sound for a normal CTRS, then so is $\mathbb{U}$.*

4 Soundness of the SR Transformation for Weakly Left-linear CTRSs

In this section, by using $\mathbb{U}$, we show that $\mathbb{SR}$ is sound for a weakly left-linear normal CTRS.

Before the discussion, we consider the role of $(\cdot)^\perp$ again. The mapping $(\cdot)^\perp$ puts $\perp$ into the extended arguments of defined symbols. We straightforwardly extend $(\cdot)^\perp$ to substitutions: $(\theta)^\perp = \{x \mapsto (x\theta)^\perp \mid x \in \mathcal{D}om(\theta)\}$ for a substitution θ such that $\mathcal{R}an(\theta) \subseteq T(\overline{\mathcal{F}}, \mathcal{V})$. The mapping $(\cdot)^\perp$ has the following properties which are trivial by definition.

▶ **Proposition 8.** *Let $\mathcal{R}$ be a normal CTRS. Then, all of the following hold:*
- *For any term $s \in T(\mathcal{F}, \mathcal{V})$, $\overline{s} = (\overline{s})^\perp$.*
- *For any term $t \in T(\overline{\mathcal{F}}, \mathcal{V})$, $(t\theta)^\perp = (t)^\perp(\theta)^\perp$ for any substitution $\theta \in \mathcal{S}ub(\overline{\mathcal{F}}, \mathcal{V})$ such that $t\theta$ is reachable.*

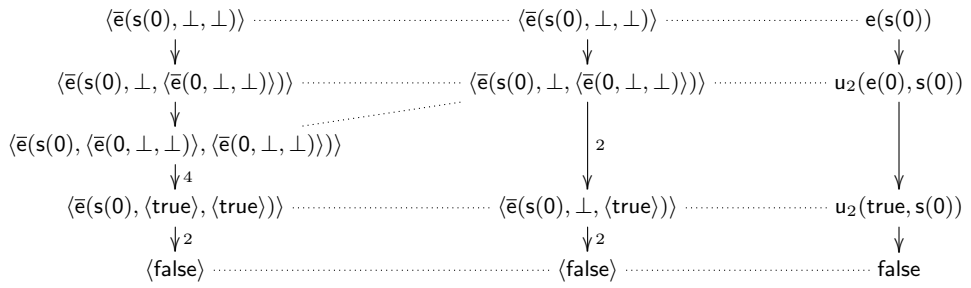
We may use Proposition 8 without notice.

To prove key claims (e.g., Lemma 13 shown later) related to the derivation $\langle\overline{s}\rangle \to^*_{\mathbb{SR}(\mathcal{R})} t$, the mappings $\bar{\cdot}$ and $\widehat{\cdot}$ often prevent us from using induction because $\widehat{\cdot}$ removes all occurrences of $\langle\cdot\rangle$ from terms. For this reason, using the mapping $(\cdot)^\perp$ instead of $\langle\bar{\cdot}\rangle$ is a breakthrough to prove our main theorem.

Next, we observe reduction sequences $\langle\overline{s}\rangle \to^*_{\mathbb{SR}(\mathcal{R})} t$ with $s \in T(\mathcal{F}, \mathcal{V})$ and $t \in T(\overline{\mathcal{F}}, \mathcal{V})$. The main feature of $\mathbb{SR}$ is to evaluate two or more conditions in parallel. However, to get $\widehat{t}$, it suffices to evaluate successfully at most one condition in each parallel evaluation of conditions. This means that every term appearing in $\langle\overline{s}\rangle \to^*_{\mathbb{SR}(\mathcal{R})} t$, which is rooted by a defined symbol $\bar{\mathsf{f}}$, is of the form $\bar{\mathsf{f}}(\boldsymbol{t}_{1..n}, \boldsymbol{u}_{1..m})$ where $\mathtt{arity}_{\mathcal{F}}(\mathsf{f}) = n$ and at most one of $u_1, \ldots, u_m$ is rooted by $\langle\cdot\rangle$ (i.e., others are $\perp$). Such a term is the key idea of this paper, and we say that the term *has no parallel evaluation of conditions*. For a term having no parallel evaluation of conditions, we can uniquely determine the corresponding term over $\mathbb{U}(\mathcal{F})$: for a term $\bar{\mathsf{f}}(\boldsymbol{t}_{1..n}, \boldsymbol{u}_{1..m})$, if all $u_1, \ldots, u_m$ are $\perp$, then the root is f, and otherwise, assuming that u_i is not $\perp$ and the others are $\perp$, then the root is $U_{\mathsf{f},i}$ which is introduced for the i-th conditional rule of f. This correspondence is illustrated in Figure 1. We first show how to convert a reachable term in $T(\overline{\mathcal{F}}, \mathcal{V})$ to a term in $T(\mathbb{U}(\mathcal{F}), \mathcal{V})$.

▶ **Definition 9.** Let $\mathcal{R}$ be a normal CTRS. Then, we define a mapping Φ from reachable terms in $T(\overline{\mathcal{F}}, \mathcal{V})$ to $T(\mathbb{U}(\mathcal{F}), \mathcal{V})$ as follows:
- $\Phi(x) = x$ for $x \in \mathcal{V}$,
- $\Phi(\mathsf{c}(\boldsymbol{t}_{1..n})) = \mathsf{c}(\Phi(\boldsymbol{t}_{1..n}))$ for $\mathsf{c}/n \in \mathcal{C}_{\mathcal{R}}$,
- $\Phi(\bar{\mathsf{f}}(\boldsymbol{t}_{1..n}, \perp, \ldots, \perp)) = \mathsf{f}(\Phi(\boldsymbol{t}_{1..n}))$ for $\mathsf{f}/n \in \mathcal{D}_{\mathcal{R}}$,

[1] In [21], "ground confluence" is used instead of "confluence" since reduction sequences on ground terms are considered. From the proofs in [21], we can consider "confluence" for the case that arbitrary reduction sequences are considered.



(a) on $\mathbb{SR}(\mathcal{R}_1)$ (b) on $\mathbb{SR}(\mathcal{R}_1)$ with single evaluation (c) on $\mathbb{U}(\mathcal{R}_1)$

Figure 1 The correspondence of derivations between $\mathbb{SR}(\mathcal{R}_1)$ and $\mathbb{U}(\mathcal{R}_1)$.

- $\Phi(\overline{\mathsf{f}}(\theta(\mathtt{ext}(\boldsymbol{w}_{1..n})), \overbrace{\bot, \ldots, \bot}^{i-1}, \langle u_i\rangle, \bot, \ldots, \bot)) = U_{\mathsf{f},i}(\Phi(u_i), \Phi(\theta(\boldsymbol{x}_{1..j})))$ for $\mathsf{f}/n \in \mathcal{D}_{\mathcal{R}}$, where the i-th conditional rule of f is $\rho : \mathsf{f}(\boldsymbol{w}_{1..n}) \to r_i \Leftarrow s_i \twoheadrightarrow t_i \in \mathcal{R}$, $U_{\mathsf{f},i}(t_i, \boldsymbol{x}_{1..j}) \to r_i \in \mathbb{U}(\rho)$ with $\mathcal{V}ar(\boldsymbol{w}_{1..n}) = \{\boldsymbol{x}_{1..j}\}$, $\Phi(\theta(\mathtt{ext}(w_j)))$ is defined for all $1 \le j \le n$, $\Phi(x\theta)$ is defined for any variable $x \in \mathcal{D}om(\theta)$, and $\Phi((\langle\overline{s_i}\rangle\theta)^{\bot}) \to^*_{\mathbb{U}(\mathcal{R})} \Phi((\langle u_i\rangle)^{\bot})$, and
- $\Phi(\langle t\rangle) = \Phi(t)$.

We say that $\Phi(\theta)$ is *defined* if $\Phi(x\theta)$ is defined for any variable $x \in \mathcal{D}om(\theta)$, and we denote by $\Phi(\theta)$ the substitution $\{x \mapsto \Phi(x\theta) \mid x \in \mathcal{D}om(\theta)\}$.

Note that Φ is a partial mapping, and linearity is not important for Φ while $w_1, \ldots, w_n$ in the proof of Lemma 13 are linear without any shared variable. By definition, it is clear that $\Phi((t)^{\bot}) = \widehat{t}$ for a reachable term $t \in T(\overline{\mathcal{F}}, \mathcal{V})$. We will use this property without notice.

▶ **Example 10.** Consider $\mathbb{U}(\mathcal{R}_1)$ and $\mathbb{SR}(\mathcal{R}_1)$ in Examples 2 and 5, respectively, again. We have that $\Phi(\langle\overline{\mathsf{e}}(\mathsf{s}(0), \bot, \bot)\rangle) = \mathsf{e}(\mathsf{s}(0))$, $\Phi(\langle\overline{\mathsf{e}}(\mathsf{s}(0), \bot, \langle\overline{\mathsf{e}}(0, \bot, \bot)\rangle)\rangle) = \mathsf{u}_2(\mathsf{e}(0), \mathsf{s}(0))$, and $\Phi(\langle\overline{\mathsf{e}}(\mathsf{s}(0), \bot, \langle\mathsf{false}\rangle)\rangle) = \mathsf{u}_2(\mathsf{false}, \mathsf{s}(0))$. On the other hand, Φ is not defined for the term $\langle\overline{\mathsf{e}}(\mathsf{s}(0), \langle\overline{\mathsf{e}}(0, \bot, \bot)\rangle, \langle\overline{\mathsf{e}}(0, \bot, \bot)\rangle)\rangle$ which contains two parallel evaluations of conditions.

Unfortunately, the above idea for the proof does not hold for all normal CTRSs.

▶ **Example 11.** Consider the following normal CTRS:

$$\mathcal{R}_2 = \left\{ \begin{array}{llll} \mathsf{f}(x) \to x \Leftarrow x \twoheadrightarrow \mathsf{c}, & \mathsf{g}(x,x) \to \mathsf{h}(x,x), & \mathsf{h}(\mathsf{f}(\mathsf{d}), x) \to x, & \\ \mathsf{a} \to \mathsf{c}, & \mathsf{a} \to \mathsf{d}, & \mathsf{b} \to \mathsf{c}, & \mathsf{b} \to \mathsf{d} \end{array} \right\}$$

$\mathcal{R}_2$ is transformed by $\mathbb{U}$ and $\mathbb{SR}$, respectively, as follows:

$$\mathbb{U}(\mathcal{R}_2) = \{\ \mathsf{f}(x) \to \mathsf{u}_3(x,x),\quad \mathsf{u}_3(\mathsf{c}, x) \to x,\quad \mathsf{g}(x,x) \to \mathsf{h}(x,x),\quad \mathsf{h}(\mathsf{f}(\mathsf{d}), x) \to x,\quad \ldots\ \}$$

$$\mathbb{SR}(\mathcal{R}_2) = \left\{ \begin{array}{llll} \overline{\mathsf{f}}(x, \bot) \to \overline{\mathsf{f}}(x, \langle x\rangle), & \overline{\mathsf{f}}(x, \langle\mathsf{c}\rangle) \to \langle x\rangle, & \overline{\mathsf{a}} \to \langle\mathsf{c}\rangle, & \overline{\mathsf{a}} \to \langle\mathsf{d}\rangle, \\ \overline{\mathsf{g}}(x,x) \to \langle\overline{\mathsf{h}}(x,x)\rangle, & \overline{\mathsf{h}}(\overline{\mathsf{f}}(\mathsf{d}, z_1), x) \to \langle x\rangle, & \overline{\mathsf{b}} \to \langle\mathsf{c}\rangle, & \overline{\mathsf{b}} \to \langle\mathsf{d}\rangle, \\ \langle\langle x\rangle\rangle \to \langle x\rangle, & \overline{\mathsf{g}}(\langle x\rangle, y) \to \langle\overline{\mathsf{g}}(x,y)\rangle, & \overline{\mathsf{h}}(\langle x\rangle, y) \to \langle\overline{\mathsf{h}}(x,y)\rangle, & \\ \overline{\mathsf{f}}(\langle x\rangle, z_1) \to \langle\overline{\mathsf{f}}(x, \bot)\rangle, & \overline{\mathsf{g}}(x, \langle y\rangle) \to \langle\overline{\mathsf{g}}(x,y)\rangle, & \overline{\mathsf{h}}(x, \langle y\rangle) \to \langle\overline{\mathsf{h}}(x,y)\rangle & \end{array} \right\}$$

We have the following derivations:

$$\begin{array}{ll} \langle\overline{\mathsf{g}}(\overline{\mathsf{f}}(\overline{\mathsf{a}}, \bot), \overline{\mathsf{f}}(\overline{\mathsf{b}}, \bot))\rangle & \mathsf{g}(\mathsf{f}(\mathsf{a}), \mathsf{f}(\mathsf{b})) \\ \quad \to^*_{\mathbb{SR}(\mathcal{R}_2)} \langle\overline{\mathsf{g}}(\overline{\mathsf{f}}(\langle\mathsf{d}\rangle, \langle\mathsf{c}\rangle), \overline{\mathsf{f}}(\langle\mathsf{d}\rangle, \langle\mathsf{c}\rangle))\rangle & \quad \to^*_{\mathbb{U}(\mathcal{R}_2)} \mathsf{g}(\mathsf{u}_3(\mathsf{c},\mathsf{d}), \mathsf{u}_3(\mathsf{c},\mathsf{d})) \\ \quad \to^*_{\mathbb{SR}(\mathcal{R}_2)} \langle\overline{\mathsf{f}}(\langle\mathsf{d}\rangle, \langle\mathsf{c}\rangle)\rangle \to^*_{\mathbb{SR}(\mathcal{R}_2)} \langle\mathsf{d}\rangle & \quad \not\to_{\mathbb{U}(\mathcal{R}_2)} \mathsf{u}_3(\mathsf{c},\mathsf{d}) \to_{\mathbb{U}(\mathcal{R}_2)} \mathsf{d} \end{array}$$

Neither $\mathbb{U}(\mathcal{R}_2)$ nor $\mathcal{R}_2$ can simulate the derivation $\langle\overline{\mathsf{g}}(\overline{\mathsf{f}}(\overline{\mathsf{a}}, \bot), \overline{\mathsf{f}}(\overline{\mathsf{b}}, \bot))\rangle \to^*_{\mathbb{SR}(\mathcal{R}_2)} \langle\mathsf{d}\rangle$, and thus, $\mathbb{SR}$ is not sound for $\mathcal{R}_2$.

As shown below, we only succeed in proving that the idea works for weakly left-linear normal CTRSs. Weakly left-linear normal CTRSs have the following syntactic properties with respect to $\mathbb{SR}$, which are trivial by definition.

▶ **Proposition 12.** *If $\mathcal{R}$ is weakly left-linear, then $\mathbb{SR}(\mathcal{R})$ is weakly left-linear, especially, for transformed rewrite rules $\overline{\mathsf{f}}(\boldsymbol{w'}_{1..n}, \boldsymbol{z}_{1..i-1}, \bot, \boldsymbol{z}_{i+1..m}) \to \overline{\mathsf{f}}(\boldsymbol{w'}_{1..n}, \boldsymbol{z}_{1..i-1}, \langle s_i\rangle, \boldsymbol{z}_{i+1..m})$ and $\overline{\mathsf{f}}(\boldsymbol{w'}_{1..n}, \boldsymbol{z}_{1..i-1}, \langle t_i\rangle, \boldsymbol{z}_{i+1..m}) \to \langle \overline{r_i}\rangle$ in $\mathbb{SR}(\mathsf{f}(\boldsymbol{w}_{1..n}) \to r_i \Leftarrow s_i \twoheadrightarrow t_i)$,*

- *$w'_1, \ldots, w'_n$ are linear without any shared variable, and*
- *$\overline{\mathsf{f}}(\boldsymbol{w'}_{1..n}, \boldsymbol{z}_{1..i-1}, \langle t_i\rangle, \boldsymbol{z}_{i+1..m}) \to \langle \overline{r_i}\rangle$ is left-linear.*

The following claim is an auxiliary lemma to show that $\mathbb{U}(\mathcal{R})$ can simulate every reduction sequence of the form $\langle \overline{s}\rangle \to^*_{\mathbb{SR}(\mathcal{R})} t$ with $s \in T(\mathcal{F}, \mathcal{V})$ and $t \in T(\overline{\mathcal{F}}, \mathcal{V})$.

▶ **Lemma 13.** *Let $\mathcal{R}$ be a weakly left-linear normal CTRS, s be a term in $T(\overline{\mathcal{F}}, \mathcal{V})$, t be a term in $T(\mathcal{C}_\mathcal{R} \cup \{\langle\cdot\rangle\}, \mathcal{V})$, and $\theta \in \mathcal{S}ub(\overline{\mathcal{F}}, \mathcal{V})$ with $\mathcal{D}om(\theta) \subseteq \mathcal{V}ar((t)^\bot)$. If $s \rightrightarrows^k_{\mathbb{SR}(\mathcal{R})} t\theta$ ($k \geq 0$) and $\Phi(s)$ is defined, then there exists a substitution $\theta' \in \mathcal{S}ub(\overline{\mathcal{F}}, \mathcal{V})$ such that*

- *$\mathcal{D}om(\theta') = \mathcal{D}om(\theta)$,*
- *$\Phi(\theta')$ is defined,*
- *for any variable $x \in \mathcal{D}om(\theta')$, if t is linear w.r.t. x, then $x\theta' \rightrightarrows^{\leq k}_{\mathbb{SR}(\mathcal{R})} x\theta$ and $\Phi((x\theta')^\bot) \to^*_{\mathbb{U}(\mathcal{R})} \Phi((x\theta)^\bot)$, and otherwise, $x\theta' = (x\theta)^\bot$, and*
- *$\Phi((s)^\bot) \to^*_{\mathbb{U}(\mathcal{R})} \Phi((t\theta')^\bot)$.*

Proof. This lemma can be proved by induction on the lexicographic product $(k, |s|)$ where $|s|$ denotes the size of s (see the appendix in a full version of this paper [2]). ◀

Weak left-linearity is used skillfully in the proof of Lemma 13.

Next, we show that $\langle \overline{s}\rangle \to^*_{\mathbb{SR}(\mathcal{R})} t$ can be simulated by $\mathbb{U}(\mathcal{R})$.

▶ **Theorem 14.** *Let $\mathcal{R}$ be a weakly left-linear normal CTRS, s be a term in $T(\mathcal{F}, \mathcal{V})$, and t be a term in $T(\overline{\mathcal{F}}, \mathcal{V})$. If $\langle \overline{s}\rangle \to^*_{\mathbb{SR}(\mathcal{R})} t$, then $s \to^*_{\mathbb{U}(\mathcal{R})} \widehat{t}$.*

Proof. By definition, $\Phi((\langle \overline{s}\rangle)^\bot)$ $(= s)$ is defined, and thus, it follows from Lemma 13 that $\Phi((\langle \overline{s}\rangle)^\bot) \to^*_{\mathbb{U}(\mathcal{R})} \Phi((t)^\bot)$. By definition, $\Phi((t)^\bot) = \widehat{t}$. Therefore, $s \to^*_{\mathbb{U}(\mathcal{R})} \widehat{t}$. ◀

Theorem 14 does not hold for all normal CTRSs.

▶ **Example 15.** Consider the following normal CTRS which is not weakly left-linear:

$$\mathcal{R}_3 = \left\{ \begin{array}{llll} \mathsf{f}(x) \to \mathsf{c} \Leftarrow x \twoheadrightarrow \mathsf{c}, & \mathsf{f}(x) \to \mathsf{d} \Leftarrow x \twoheadrightarrow \mathsf{d}, & \mathsf{g}(x,x) \to \mathsf{h}(x,x), & \\ \mathsf{a} \to \mathsf{c}, & \mathsf{a} \to \mathsf{d}, & \mathsf{b} \to \mathsf{c}, & \mathsf{b} \to \mathsf{d} \end{array} \right\}$$

$\mathcal{R}_3$ is transformed by $\mathbb{U}$ and $\mathbb{SR}$, respectively, as follows:

$$\mathbb{U}(\mathcal{R}_3) = \left\{ \begin{array}{lll} \mathsf{f}(x) \to \mathsf{u}_5(x,x), & \mathsf{f}(x) \to \mathsf{u}_6(x,x), & \mathsf{g}(x,x) \to \mathsf{h}(x,x), \\ \mathsf{u}_5(\mathsf{c},x) \to \mathsf{c}, & \mathsf{u}_6(\mathsf{d},x) \to \mathsf{d}, & \ldots \end{array} \right\}$$

$$\mathbb{SR}(\mathcal{R}_3) = \left\{ \begin{array}{lll} \overline{\mathsf{f}}(x,\bot,z_2) \to \overline{\mathsf{f}}(x,\langle x\rangle,z_2), & \overline{\mathsf{f}}(x,z_1,\bot) \to \overline{\mathsf{f}}(x,z_1,\langle x\rangle), & \overline{\mathsf{g}}(x,x) \to \langle \mathsf{h}(x,x)\rangle, \\ \overline{\mathsf{f}}(x,\langle \mathsf{c}\rangle,z_2) \to \langle \mathsf{c}\rangle, & \overline{\mathsf{f}}(x,z_1,\langle \mathsf{d}\rangle) \to \langle \mathsf{d}\rangle, & \overline{\mathsf{a}} \to \langle \mathsf{c}\rangle, \\ \overline{\mathsf{a}} \to \langle \mathsf{d}\rangle, & \overline{\mathsf{b}} \to \langle \mathsf{c}\rangle, & \overline{\mathsf{b}} \to \langle \mathsf{d}\rangle, \\ \langle\langle x\rangle\rangle \to \langle x\rangle, & \mathsf{h}(\langle x\rangle, y) \to \langle \mathsf{h}(x,y)\rangle, & \mathsf{h}(x,\langle y\rangle) \to \langle \mathsf{h}(x,y)\rangle, \\ \overline{\mathsf{f}}(\langle x\rangle, z_1, z_2) \to \langle \overline{\mathsf{f}}(x,\bot,\bot)\rangle, & \overline{\mathsf{g}}(\langle x\rangle, y) \to \langle \overline{\mathsf{g}}(x,y)\rangle, & \overline{\mathsf{g}}(x,\langle y\rangle) \to \langle \overline{\mathsf{g}}(x,y)\rangle \end{array} \right\}$$

[2] Available from `http://www.trs.cm.is.nagoya-u.ac.jp/~nishida/wpte14/`.

We have the following derivations:

$$\begin{array}{ll}\langle \overline{\mathsf{g}}(\overline{\mathsf{f}}(\mathsf{a},\bot,\bot),\overline{\mathsf{f}}(\overline{\mathsf{b}},\bot,\bot))\rangle & \mathsf{g}(\mathsf{f}(\mathsf{a}),\mathsf{f}(\mathsf{b}))\\ \quad \to^*_{\mathbb{SR}(\mathcal{R}_3)} \ \langle \mathsf{h}(\overline{\mathsf{f}}(\langle \mathsf{d}\rangle,\langle \mathsf{c}\rangle,\langle \mathsf{d}\rangle),\overline{\mathsf{f}}(\langle \mathsf{d}\rangle,\langle \mathsf{c}\rangle,\langle \mathsf{d}\rangle))\rangle & \quad \to^*_{\mathbb{U}(\mathcal{R}_3)} \ \mathsf{h}(\mathsf{u}_5(\mathsf{c},\mathsf{d}),\mathsf{u}_5(\mathsf{c},\mathsf{d}))\\ \quad \to_{\mathbb{SR}(\mathcal{R}_3)} \ \langle \mathsf{h}(\langle \mathsf{c}\rangle,\langle \mathsf{d}\rangle)\rangle \ \to^*_{\mathbb{SR}(\mathcal{R}_3)} \ \langle \mathsf{h}(\mathsf{c},\mathsf{d})\rangle & \quad \not\to_{\mathbb{U}(\mathcal{R}_3)} \ \mathsf{h}(\mathsf{c},\mathsf{d})\end{array}$$

The other normal forms of $\mathsf{g}(\mathsf{f}(\mathsf{a}),\mathsf{f}(\mathsf{b}))$ on $\mathbb{U}(\mathcal{R}_3)$ are $\mathsf{h}(\mathsf{u}_5(\mathsf{c},\mathsf{c}),\mathsf{u}_5(\mathsf{c},\mathsf{c}))$, $\mathsf{h}(\mathsf{u}_5(\mathsf{d},\mathsf{c}),\mathsf{u}_5(\mathsf{d},\mathsf{c}))$, and $\mathsf{h}(\mathsf{u}_5(\mathsf{d},\mathsf{d}),\mathsf{u}_5(\mathsf{d},\mathsf{d}))$, but none of them corresponds to $\mathsf{h}(\mathsf{c},\mathsf{d})$. For this reason, the derivation on $\mathbb{SR}(\mathcal{R}_3)$ cannot be simulated by $\mathbb{U}(\mathcal{R}_3)$. The derivation $\mathsf{g}(\mathsf{f}(\mathsf{a}),\mathsf{f}(\mathsf{b})) \to^* \mathsf{h}(\mathsf{c},\mathsf{d})$ does not hold on $\mathcal{R}_3$, either, and thus, $\mathbb{SR}$ is not sound for $\mathcal{R}_3$. In addition, being a constructor system is not sufficient for soundness of $\mathbb{SR}$ since $\mathcal{R}_3$ is a constructor system.

We show the main result obtained by Theorem 14.

▶ **Theorem 16.** *$\mathbb{SR}$ is sound for weakly left-linear normal CTRS.*

Proof. Let $\mathcal{R}$ be a weakly left-linear CTRS, $s \in T(\mathcal{F},\mathcal{V})$ and $t \in T(\overline{\mathcal{F}},\mathcal{V})$. Suppose that $\langle s\rangle \to^*_{\mathbb{SR}(\mathcal{R})} t$. Then, it follows from Theorem 14 that $s \to^*_{\mathbb{U}(\mathcal{R})} \widehat{t}$. Since $\widehat{t} \in T(\mathcal{F},\mathcal{V})$ and $\mathbb{U}$ is sound for $\mathcal{R}$, we have that $s \to^*_{\mathcal{R}} \widehat{t}$. Therefore, $\mathbb{SR}$ is sound for $\mathcal{R}$. ◀

Theorem 16 does not hold for all normal CTRSs (see Examples 11 and 15).

Finally, we discuss the remaining soundness conditions of $\mathbb{U}$: non-erasingness and groundness of conditions. Non-erasingness of normal CTRSs is not sufficient for soundness since $\mathcal{R}_2$ is non-erasing but $\mathbb{SR}$ is not sound for $\mathcal{R}_2$. Groundness of conditions is not sufficient for soundness, either.

▶ **Example 17.** Consider the following ground-conditional normal CTRS, a variant of $\mathcal{R}_3$:

$$\mathcal{R}_4 = \left\{ \begin{array}{llll} \mathsf{f}(\mathsf{a}) \to \mathsf{c} \Leftarrow \mathsf{a} \twoheadrightarrow \mathsf{c}, & \mathsf{f}(\mathsf{b}) \to \mathsf{d} \Leftarrow \mathsf{b} \twoheadrightarrow \mathsf{d}, & \mathsf{g}(x,x) \to \mathsf{h}(x,x), & \\ \mathsf{a} \to \mathsf{c}, & \mathsf{a} \to \mathsf{d}, & \mathsf{b} \to \mathsf{c}, & \mathsf{b} \to \mathsf{d} \end{array} \right\}$$

We have that $\overline{\mathsf{g}}(\overline{\mathsf{f}}(\mathsf{a},\bot,\bot),\overline{\mathsf{f}}(\overline{\mathsf{b}},\bot,\bot)) \to^*_{\mathbb{SR}(\mathcal{R}_4)} \langle \mathsf{h}(\mathsf{c},\mathsf{d})\rangle$, but $\mathsf{g}(\mathsf{f}(\mathsf{a}),\mathsf{f}(\mathsf{b})) \not\to^*_{\mathcal{R}_4} \mathsf{h}(\mathsf{c},\mathsf{d})$. Therefore, $\mathbb{SR}$ is not sound for $\mathcal{R}_4$.

5 Conclusion

In this paper, by using the soundness of $\mathbb{U}$ for weakly left-linear normal CTRSs, we showed that the SR transformation is sound for weakly left-linear normal CTRSs. As far as we know, this paper is the second work on comparing soundness of unravelings and the SR transformation. The first one is a previous work [17] of the first author, in which the converse of Theorem 7 was left as a conjecture. As a negative result, we showed that the converse of Theorem 7 does not hold in general.

One may think that as the first step, we should have started with the transformation proposed by Antoy et al [1], which is a variant of Viry's transformation. As described in [21], for constructor systems, the unary symbol introduced in the SR transformation to wrap terms evaluating conditions is not necessary and then the SR transformation is the same as the one in [1]. The transformation in [1] is sound for left-linear constructor normal CTRSs, and is extended to the SR transformation in order to adapt it to arbitrary normal CTRSs. This means that any result for the SR transformation can be adapted to the transformation in [1]. Moreover, the SR transformation has been extended to *syntactically or strongly deterministic* CTRSs [22], which we would like to deal with at the next step of this research. For these reasons, we started with the SR transformation.

Schernhammer and Gramlich showed in [20] that a particular *context-sensitive condition* [13] is sufficient for soundness of Ohlebusch's unraveling [18], which is an improved variant of Marchiori's one. However, the context-sensitive condition is not sufficient for preserving confluence of CTRSs. For this reason, not all the unraveled TRSs with the context-sensitive condition are computationally equivalent to the original CTRSs, and in this sense, the SR transformation is more useful than unravelings with the context-sensitive condition. Moreover, the context-sensitive condition restricts the reduction to the context-sensitive one. Due to this restriction, we did not use the context-sensitive condition in proving the main result in this paper.

As future work, we will extend Theorem 14 to a pair of Ohlebusch's unraveling [18] and the SR transformation for syntactically or strongly deterministic CTRSs in order to extend Theorem 16 to the SR transformation for the CTRSs. We did not discuss confluence of normal CTRSs as a soundness condition since the SR transformation is known to be sound for confluent normal CTRSs. However, for the extension to deterministic CTRSs, we will adapt the proof technique in this paper to confluent normal CTRSs. Moreover, we will investigate other soundness conditions of unravelings in order to make the SR transformation applicable to more classes of CTRSs as a computationally equivalent transformation.

Acknowledgement. We are deeply grateful to the anonymous referees for their useful comments to improve this paper. We would like to dedicate this paper to the memory of Bernhard Gramlich who encouraged us to further research the soundness of unravelings.

References

1 Sergio Antoy, Bernd Brassel, and Michael Hanus. Conditional narrowing without conditions. In *Proceedings of the 5th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pp. 20–31, ACM, 2003.

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

3 Hubert Comon, Max Dauchet Reémi Gilleron, Florent Jacquemard, Denis Lugiez Christof, Löding, Sophie Tison, and Marc Tommasi. Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 2007.

4 Francisco Durán, Salvador Lucas, José Meseguer, Claude Marché, and Xavier Urbain. Proving termination of membership equational programs. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*, pp. 147–158, ACM, 2004.

5 Guillaume Feuillade and Thomas Genet. Reachability in conditional term rewriting systems. *Electronic Notes in Theoretical Computer Science*, 86(1):133–146, 2003.

6 Thomas Genet. Decidable approximations of sets of descendants and sets of normal forms. In *Proceedings of the 9th International Conference on Rewriting Techniques and Applications*, volume 1379 of *Lecture Notes in Computer Science*, pp. 151–165, Springer, 1998.

7 Thomas Genet and Francis Klay. Rewriting for cryptographic protocol verification. In *Proceedings of the 17th International Conference on Automated Deduction*, volume 1831 of *Lecture Notes in Computer Science*, pp. 271–290, Springer, 2000.

8 Karl Gmeiner and Bernhard Gramlich. Transformations of conditional rewrite systems revisited. In *Proceedings of the 19th International Workshop on Recent Trends in Algebraic Development Techniques*, volume 5486 of *Lecture Notes in Computer Science*, pp. 166–186, Springer, 2009.

9 Karl Gmeiner, Bernhard Gramlich, and Felix Schernhammer. On (un)soundness of unravelings. In *Proceedings of the 21st International Conference on Rewriting Techniques and*

Applications, volume 6 of *LIPIcs*, pp. 119–134, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2010.

10 Karl Gmeiner, Bernhard Gramlich, and Felix Schernhammer. On soundness conditions for unraveling deterministic conditional rewrite systems. In *Proceedings of the 23rd International Conference on Rewriting Techniques and Applications*, volume 15 of *LIPIcs*, pp. 193–208, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2012.

11 Karl Gmeiner, Naoki Nishida, and Bernhard Gramlich. Proving confluence of conditional term rewriting systems via unravelings. In *Proceedings of the 2nd International Workshop on Confluence*, pp. 35–39, 2013.

12 Florent Jacquemard. Decidable approximations of term rewriting systems. In *Proceedings of the 7th International Conference on Rewriting Techniques and Applications*, volume 1103 of *Lecture Notes in Computer Science*, pp. 362–376, Springer, 1996.

13 Salvador Lucas. Context-sensitive computations in functional and functional logic programs. *Journal of Functional and Logic Programming*, 1998(1), 1998.

14 Salvador Lucas, Claude Marché, and José Meseguer. Operational termination of conditional term rewriting systems. *Information Processing Letters*, 95(4):446–453, 2005.

15 Massimo Marchiori. Unravelings and ultra-properties. In *Proceedings of the 5th International Conference on Algebraic and Logic Programming*, volume 1139 of *Lecture Notes in Computer Science*, pp. 107–121, Springer, 1996.

16 Massimo Marchiori. On deterministic conditional rewriting. Computation Structures Group, Memo 405, MIT Laboratory for Computer Science, 1997.

17 Naoki Nishida, Masahiko Sakai, and Toshiki Sakabe. Soundness of unravelings for conditional term rewriting systems via ultra-properties related to linearity. *Logical Methods in Computer Science*, 8(3):1–49, 2012.

18 Enno Ohlebusch. Termination of logic programs: Transformational methods revisited. *Applicable Algebra in Engineering, Communication and Computing*, 12(1/2):73–116, 2001.

19 Enno Ohlebusch. *Advanced Topics in Term Rewriting*. Springer, 2002.

20 Felix Schernhammer and Bernhard Gramlich. Characterizing and proving operational termination of deterministic conditional term rewriting systems. *Journal of Logic and Algebraic Programming*, 79(7):659–688, 2010.

21 Traian-Florin Şerbănuţă and Grigore Roşu. Computationally equivalent elimination of conditions. In *Proceedings of the 17th International Conference on Rewriting Techniques and Applications*, volume 4098 of *Lecture Notes in Computer Science*, pp. 19–34, Springer, 2006.

22 Traian-Florin Şerbănuţă and Grigore Roşu. Computationally equivalent elimination of conditions. Technical Report UIUCDCS-R-2006-2693, Department of Computer Science, University of Illinois at Urbana-Champaign, 2006.

23 Taro Suzuki, Aart Middeldorp, and Tetsuo Ida. Level-confluence of conditional rewrite systems with extra variables in right-hand sides. In *Proceedings of the 6th International Conference on Rewriting Techniques and Applications*, volume 914 of *Lecture Notes in Computer Science*, pp. 179–193, Springer, 1995.

24 Toshinori Takai, Yuichi Kaji, and Hiroyuki Seki. Right-linear finite path overlapping term rewriting systems effectively preserve recognizability. In *Proceedings of the 11th International Conference on Rewriting Techniques and Applications*, volume 1833 of *Lecture Notes in Computer Science*, pp. 246–260, Springer, 2000.

25 Patrick Viry. Elimination of conditions. *Journal of Symbolic Computation*, 28(3):381–401, 1999.

Structural Rewriting in the π-Calculus

David Sabel

Goethe-University, Frankfurt am Main
sabel@ki.cs.uni-frankfurt.de

Abstract

We consider reduction in the synchronous π-calculus with replication, without sums. Usual definitions of reduction in the π-calculus use a closure w.r.t. structural congruence of processes. In this paper we operationalize structural congruence by providing a reduction relation for pi-processes which also performs necessary structural conversions explicitly by rewrite rules. As we show, a subset of structural congruence axioms is sufficient. We show that our rewrite strategy is equivalent to the usual strategy including structural congruence w.r.t. the observation of barbs and thus w.r.t. may- and should-testing equivalence in the pi-calculus.

1998 ACM Subject Classification F.4.2 Grammars and Other Rewriting Systems, F.3.2 Semantics of Programming Languages, D.3.1 Formal Definitions and Theory

Keywords and phrases Process calculi, Rewriting, Semantics

Digital Object Identifier 10.4230/OASIcs.WPTE.2014.51

1 Introduction

The π-calculus [9, 8, 21] is a well-known core model for concurrent and mobile processes with message passing. Its syntax includes parallel process-composition, named channels, and input/output-capabilities on the channels. The data flow in programs of the π-calculus is represented by communication between processes. Since links between processes can be sent as messages, the induced network of processes behaves dynamically.

Even though the π-calculus has been investigated for several decades, its analysis is still an active research topic. One reason is that variants of the π-calculus have a lot of applications even outside the field of computer science. Examples are the Spi-calculus [1] to reason about cryptographic protocols, the stochastic π-calculus [14] with applications in molecular biology, and the use of the π-calculus to model business processes.

The operational behavior of π-processes is defined in terms of a reduction semantics (and often extended by an action semantics, for semantic reasoning), which is built by a single reduction rule for exchanging a message, and applying this reduction rule in so-called reduction contexts. Additionally, a notion of structural congruence is used, which can be applied to processes before and after the reduction step. Structural congruence allows conversions like commuting the order of parallel processes, moving the scope of binders, etc. Unfortunately the complexity of structural congruence is very high. Indeed in its original formulation by Milner, it is even unknown whether structural congruence is decidable. A recent result [22] shows that it is at least EXPSPACE-hard. For reasoning on reductions and semantics of processes the *implicit* use of structural congruence is difficult and also error-prone. This becomes even more difficult, when such proofs are automated. Hence, in this paper we make the conversion w.r.t. structural congruence *explicit* by including the congruence axioms as separate reduction rules in the reduction relation. Moreover, we also simplify the conversion by dropping some rules, which are not necessary for defining the reduction relation (but they may be used as program optimizations).

1st International Workshop on Rewriting Techniques for Program Transformations and Evaluation (WPTE'14).
Editors: Manfred Schmidt-Schauß, Masahiko Sakai, David Sabel, and Yuki Chiba; pp. 51–62

OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Our main result is that the semantics of π-processes remains unchanged if our new reduction relation replaces the original one. As semantics we use barbed may- and should-testing equivalence [4] which holds for two processes if their input and output capabilities coincide if the processes are plugged into any program context. This notion of process equivalence is directly based on the reduction semantics and – unlike other process equivalences – like bisimulation or barbed congruence (see e.g. [21]) – it is insensitive for reduction traces. Hence barbed may- and should-testing equivalence can be viewed as the coarsest notion of process equivalence which distinguishes obviously different processes (see [4] for a deep analysis of the different notions of process equivalences and their relation).

Outline In Sect. 2 we recall the synchronous π-calculus and its reduction semantics which includes structural congruence, and we define may- and should-testing equivalence. In Sect. 3 we introduce the simplified reduction semantics – called $\mathcal{D}$-standard reduction – which makes conversions w.r.t. structural congruence explicit. In Sect. 4 we show that may- and should-testing equivalence remains unchanged if the $\mathcal{D}$-standard reduction is used. We conclude in Sect. 5.

2 The Synchronous π-Calculus with May- and Should-Testing

We consider the synchronous π-calculus with replication but without sums (and recursion).

▶ **Definition 2.1** (Syntax of the π-Calculus)**.** Let $\mathcal{N}$ be a countably infinite set of *names*. *Processes* P and *action prefixes* π are defined by the following grammar, where $x, y \in \mathcal{N}$:

$$\begin{array}{lcl} P & ::= & \pi.P \mid P_1 \mid P_2 \mid \mathbf{!}\,P \mid \mathbf{0} \mid \nu x.P \\ \pi & ::= & x(y) \mid \overline{x}\langle y\rangle \end{array}$$

The prefix $x(y)$ is called an *input-prefix*, and $\overline{x}\langle y\rangle$ is called an *output-prefix*.

Names are *bound* by the ν-binder (in $\nu x.P$ name x is bound with scope P) and by the input-prefix (in $x(y).P$ name y is bound with scope P). This induces a notion of α-renaming and α-equivalence as usual. We treat α-equivalent processes as equal. If necessary, we make α-renaming explicit and denote α-equivalence by $=_\alpha$. We use $\mathsf{fn}(P)$ ($\mathsf{fn}(\pi)$, resp.) for the set of *free names* of process P (prefix π, resp.) and $\mathsf{bn}(P)$ ($\mathsf{bn}(\pi)$, resp.) for the set of *bound names* of process P (prefix π, resp.). Note that $\mathsf{fn}(x(y)) = \{x\}$, $\mathsf{fn}(\overline{x}\langle y\rangle) = \{x, y\}$, $\mathsf{bn}(x(y)) = \{y\}$, and $\mathsf{bn}(\overline{x}\langle y\rangle) = \emptyset$. We assume the distinct name convention, i.e. free names are distinct from bound names and bound names are pairwise distinct.

A process $x(y).P$ has the capability to *receive* some name z along the channel named x and then behaves like $P[z/y]$ where $[z/y]$ is the capture free substitution of name y by name z. A process $\overline{x}\langle y\rangle.P$ has the capability to *send* a name y along the channel named x. The *silent process* $\mathbf{0}$ has no capabilities to communicate. $P_1 \mid P_2$ is the *parallel composition* of processes P_1 and P_2. $\nu z.P$ *restricts* the scope of the name z to process P. As a notation we use $\nu\mathcal{X}.P$ abbreviating $\nu x_1.\ldots.\nu x_n.P$. We also use set-notation for $\mathcal{X}$ and e.g. write $x \in \mathcal{X}$ with its obvious meaning. $\mathbf{!}\,P$ is the *replication* of process P, i.e. it can be interpreted as infinitely many copies of P running in parallel.

▶ **Definition 2.2** (Contexts)**.** A *process context* $C \in \mathcal{C}$ is a process with a *hole* $[\cdot]$ at process-position, i.e. $C \in \mathcal{C} ::= [\cdot] \mid \pi.C \mid C \mid P \mid P \mid C \mid \mathbf{!}\,C \mid \nu x.C$. For a context C and a process P the construct $C[P]$ denotes the process where the hole of C is replaced by process P. For contexts C_1, C_2 we say C_1, C_2 *are prefix disjoint* iff there does not exist a context C_3 with

$C_1 = C_2[C_3]$ or $C_2 = C_1[C_3]$. For $P = C[Q]$ the *replication depth* of Q in P w.r.t. C is the number m of replications above Q which is exactly the number of contexts C_i of the form $!\,[\cdot]$ if $C = C_1 \dots C_n$ where C_i are contexts of hole-depth 1.

▶ **Definition 2.3** (Structural congruence $\equiv$)**.** Structural congruence $\equiv$ is the smallest congruence on processes satisfying the equations:

$$\begin{array}{rcl@{\qquad}rcl} P & \equiv & Q, \text{ if } P =_\alpha Q & \nu z.\nu w.P & \equiv & \nu w.\nu z.P \\ P_1 \mid (P_2 \mid P_3) & \equiv & (P_1 \mid P_2) \mid P_3 & \nu z.\mathbf{0} & \equiv & \mathbf{0} \\ P_1 \mid P_2 & \equiv & P_2 \mid P_1 & \nu z.(P_1 \mid P_2) & \equiv & P_1 \mid \nu z.P_2, \text{ if } z \notin \mathsf{fn}(P_1) \\ P \mid \mathbf{0} & \equiv & P & !\,P & \equiv & P \mid !\,P \end{array}$$

It is unknown whether structural congruence of processes is decidable (see also [2, 3, 6] for more discussion and further results), at least it is a very hard problem:

▶ **Proposition 2.4** ([22, Corollary 4.4])**.** *The decision problem whether for two π-processes $P \equiv Q$ holds is EXPSPACE-hard.*

We define the usual reduction semantics of π-processes (see e.g. [21]) as a small-step reduction relation. The only reduction rule is the rule (ia) for communication (interaction):

$$x(y).P \mid \overline{x}\langle v\rangle.Q \xrightarrow{ia} P[v/y] \mid Q$$

Reduction contexts $\mathcal{D}$ are process contexts where the hole is not below a replication or a π-prefix, i.e. $D \in \mathcal{D} ::= [\cdot] \mid D \mid P \mid P \mid D \mid \nu x.D$ where $x \in \mathcal{N}$ and where P is a process.

▶ **Definition 2.5** (Standard Reduction, $\xrightarrow{sr}$)**.** With $\xrightarrow{\mathcal{D},ia}$ we denote the closure of $\xrightarrow{ia}$ w.r.t. reduction contexts, and a *standard reduction* $\xrightarrow{sr}$ consists of applying a $\xrightarrow{\mathcal{D},ia}$-reduction modulo structural congruence, i.e.

$$\frac{P \xrightarrow{ia} Q}{D[P] \xrightarrow{\mathcal{D},ia} D[Q]} \text{ where } D \in \mathcal{D} \qquad\qquad \frac{P \equiv P' \ \wedge \ P' \xrightarrow{\mathcal{D},ia} Q' \ \wedge \ Q' \equiv Q}{P \xrightarrow{sr} Q}$$

We use the following notation for unions of transformations, where a transformation is some binary relation on processes (e.g. $\xrightarrow{sr}$):

▶ **Definition 2.6.** For a transformation $\xrightarrow{a}$ we define $\xrightarrow{a,0} := \{(P,P) \mid P \text{ is a process}\}$ and $\xrightarrow{a,i} := \{(P,Q) \mid P \xrightarrow{a} S \wedge S \xrightarrow{a,i-1} Q\}$ for $i > 0$. Transitive and reflexive-transitive closure are defined as $\xrightarrow{a,+} := \bigcup_{i>0} \xrightarrow{a,i}$, and $\xrightarrow{a,*} := \bigcup_{i\geq 0} \xrightarrow{a,i}$. For $\xrightarrow{a}$ and $\xrightarrow{b}$ let $\xrightarrow{a\vee b} := \xrightarrow{a} \cup \xrightarrow{b}$.

▶ **Example 2.7.** An example for a sequence of standard reductions is

$$\nu x.(\overline{x}\langle w\rangle.\mathbf{0} \mid x(y).\overline{z}\langle y\rangle.\mathbf{0}) \mid !\,z(u).\mathbf{0} \xrightarrow{sr} \nu x.(\overline{z}\langle w\rangle.\mathbf{0}) \mid !\,z(u).\mathbf{0} \xrightarrow{sr} !\,z(u).\mathbf{0}$$

Making the conversions w.r.t. $\equiv$ more explicit we can write this as:

$$\begin{array}{rl} & \nu x.(\overline{x}\langle w\rangle.\mathbf{0} \mid x(y).\overline{z}\langle y\rangle.\mathbf{0}) \mid !\,z(u).\mathbf{0} \equiv \nu x.(x(y).\overline{z}\langle y\rangle.\mathbf{0} \mid \overline{x}\langle w\rangle.\mathbf{0}) \mid !\,z(u).\mathbf{0} \\ \xrightarrow{\mathcal{D},ia} & \nu x.(\overline{z}\langle w\rangle.\mathbf{0} \mid \mathbf{0}) \mid !\,z(u).\mathbf{0} \equiv \nu x.(\overline{z}\langle w\rangle.\mathbf{0}) \mid !\,z(u).\mathbf{0} \equiv \nu x.(\overline{z}\langle w\rangle.\mathbf{0} \mid !\,z(u).\mathbf{0}) \\ \equiv & \nu x.(\overline{z}\langle w\rangle.\mathbf{0} \mid (z(u).\mathbf{0} \mid !\,z(u).\mathbf{0})) \equiv \nu x.((z(u).\mathbf{0} \mid \overline{z}\langle w\rangle.\mathbf{0}) \mid !\,z(u).\mathbf{0}) \\ \xrightarrow{\mathcal{D},ia} & \nu x.((\mathbf{0} \mid \mathbf{0}) \mid !\,z(u).\mathbf{0}) \equiv \nu x.(\mathbf{0} \mid !\,z(u).\mathbf{0}) \equiv \nu x.\mathbf{0} \mid !\,z(u).\mathbf{0} \equiv \mathbf{0} \mid !\,z(u).\mathbf{0} \equiv !\,z(u).\mathbf{0} \end{array}$$

The high complexity of deciding structural congruence justifies making structural conversion visible during reduction. I.e., instead of using $\equiv$ implicitly during reduction, it makes sense to introduce the conversions as separate reduction rules. Moreover, not all axioms of structural congruence are required to apply $\xrightarrow{\mathcal{D},ia}$-steps, e.g. the axiom $\mathbf{0} \mid P \equiv P$ is not necessary, and also it is not necessary to apply the axioms below a replication or a π-prefix. Before defining a modified reduction for the π-calculus (in Sect. 3) we define the semantics of π-processes by a process equivalence. In the π-calculus several definitions and notions for process equivalences exist. We choose the approach of testing the input and output capabilities of processes in all contexts and also test whether the capability may or should occur. This notion of may- and should-testing (sometimes also called must- or fair must testing, see e.g. [11, 7, 4, 16]) is close to contextual equivalence [10, 13] in other program calculi like the lambda calculus, but adapted to the concurrent setting (see e.g. [12, 17, 18, 19, 23] for contextual equivalence with may- and should-semantics in extended concurrent lambda calculi). A strong connection between may- and should-testing equivalence and a classic notion of contextual equivalence (using a notion of success) for the π-calculus was shown in [20]. May- and should-testing equivalence is a coarse notion of program equivalence equating as much processes as possible, but discriminating obviously different processes.

Input and output capabilities are formalized by the notion of a barb:

▶ **Definition 2.8** (Barb)**.** A process P *has a barb on input* x (written as $P\!\Rsh^{x}$) iff P can receive a name on channel x, i.e. $P = \nu\mathcal{X}.(x(y).P' \mid P'')$ where $x \notin \mathcal{X}$, and P *has a barb on output* x (written as $P\!\Rsh^{\overline{x}}$) iff P can emit a name on channel x, i.e. $P = \nu\mathcal{X}.(\overline{x}\langle y\rangle.P' \mid P'')$ where $x \notin \mathcal{X}$. We write $P \equiv\!\Rsh^{x}$ ($P \equiv\!\Rsh^{\overline{x}}$, resp.) iff $P \equiv P'$ and $P'\!\Rsh^{x}$ ($P'\!\Rsh^{\overline{x}}$, resp.).

As observations for process equivalence we will use on the one hand whether a process may reduce to a process that has a barb, and on the other hand whether a process has the ability to have a barb on every reduction path:

▶ **Definition 2.9** (May-barb and Should-barb)**.** For $\mu \in \{x, \overline{x}\}$, P *may have a barb on* μ (written as $P{\downarrow_\mu}$) iff $P \xrightarrow{sr,*} Q \wedge Q \equiv\!\Rsh^{\mu}$, and P *should have a barb on* μ (written as $P{\Downarrow_\mu}$) iff $P \xrightarrow{sr,*} P' \implies P'{\downarrow_\mu}$. We write $P{\uparrow_\mu}$ iff $P{\Downarrow_\mu}$ does not hold, and $P{\Uparrow_\mu}$ iff $P{\downarrow_\mu}$ does not hold.

Note that $P{\uparrow_\mu}$ equivalently means that P can reduce to a process that has no input (or output, resp.) capabilities on the channel μ, i.e. $P{\uparrow_\mu}$ holds iff $P \xrightarrow{sr,*} P'$ and $P'{\Uparrow_\mu}$.

▶ **Definition 2.10** (Barbed May- and Should-Testing Equivalence)**.** Processes P and Q are *barbed testing equivalent* (written $P \sim Q$) iff $P \precsim Q \wedge Q \precsim P$, where $\precsim := \precsim_{\text{may}} \cap \precsim_{\text{should}}$ and

$$P \precsim_{\text{may}} Q \quad \text{iff } \forall x \in \mathcal{N},\ \mu \in \{x, \overline{x}\}, C \in \mathcal{C}\colon\ C[P]{\downarrow_\mu} \implies C[Q]{\downarrow_\mu}$$
$$P \precsim_{\text{should}} Q \text{ iff } \forall x \in \mathcal{N},\ \mu \in \{x, \overline{x}\}, C \in \mathcal{C}\colon\ C[P]{\Downarrow_\mu} \implies C[Q]{\Downarrow_\mu}$$

Our definition of barbed testing equivalence is given in a general form, since the may- and should-behavior, all channel names, the input and output barbs, and also all channels are separately considered. However, in the π-calculus the definition is equivalent to simpler definitions. I.e., it is sufficient to observe the should-behavior only, and to observe input (or output) channels exclusively, and to either observe a single channel name, or existentially observing the barb capabilities (see [4] for the asynchronous π-calculus and [20] for the synchronous variant used in this paper). However, since our results also apply for the general definition, we refrain from working with the simpler definition.

As an easy result, we show that structural congruence preserves the semantics of processes:

▶ **Proposition 2.11.** *If* $P \equiv Q$ *then* $P \sim Q$.

Proof. Let $P \equiv Q$, C be a context s.t. $P{\downarrow_\mu}$ ($P{\uparrow_\mu}$, resp.), i.e. $C[P] \xrightarrow{sr,*} P'$ and $P' \equiv\!\!\Uparrow^\mu$ (or $P' \Uparrow_\mu$, resp.). Since $\equiv$ is a congruence and included in $\xrightarrow{sr}$, we have $C[Q] \equiv C[P] \xrightarrow{sr,*} P'$ and $C[Q] \xrightarrow{sr,*} P'$ which shows $C[Q]{\downarrow_\mu}$ ($C[Q]{\uparrow_\mu}$, resp.). Also $C[Q]{\downarrow_\mu} \implies C[P]{\downarrow_\mu}$ and $C[Q]{\uparrow_\mu} \implies C[P]{\uparrow_\mu}$ hold by symmetry of $\equiv$. Since $S{\uparrow_\mu} \iff \neg S{\Downarrow_\mu}$, this implies $P \sim Q$. ◀

3 Structural Congruence as Rewriting

In this section we make the conversion w.r.t. structural equivalence explicit and also restrict this conversion to reduction contexts. In the following definition the relation $\xrightarrow{sca}$ is the reduction relation corresponding to structural congruence axioms applied in both directions. However, the rewrite rules are a little bit more general than the congruence axioms, but not more general than the structural congruence relation. The relation $\xrightarrow{sc}$ is a restriction of $\xrightarrow{sca}$, the replication axiom is only permitted in the expanding direction, and rules adding, removing and moving down ν-binders as well as adding or removing the silent process are not included. The removed rules can be seen as optimizations, i.e. removing "garbage", but – as we show – they are dispensable for reasoning about may- and should-testing equivalence.

▶ **Definition 3.1** (Structural Reduction)**.** The relation $\xrightarrow{sc}$ is defined by the following rules:

$$
\begin{array}{llll}
(assocl) & P_1 \mid (P_2 \mid P_3) \xrightarrow{sc} (P_1 \mid P_2) \mid P_3 & (replunfold) & !\,P \xrightarrow{sc} P \mid !\,P \\
(assocr) & (P_1 \mid P_2) \mid P_3 \xrightarrow{sc} P_1 \mid (P_2 \mid P_3) & (nuup) & D[\nu z.P] \xrightarrow{sc} \nu z.D[P], \text{ if } z \notin \mathsf{fn}(D), \\
(commute) & P_1 \mid P_2 \xrightarrow{sc} P_2 \mid P_1 & & [\cdot] \neq D \in \mathcal{D}
\end{array}
$$

The relation $\xrightarrow{sca}$ is defined by the rules:

$$
\begin{array}{llll}
 & P \xrightarrow{sca} Q \text{ if } P \xrightarrow{sc} Q & (replfold) & P \mid !\,P \xrightarrow{sca} !\,P \\
(nuintro) & P \xrightarrow{sca} \nu z.P \text{ if } z \notin \mathsf{fn}(P) & (intro0l) & P \xrightarrow{sca} \mathbf{0} \mid P \\
(nurem) & \nu z.P \xrightarrow{sca} P \text{ if } z \notin \mathsf{fn}(P) & (intro0r) & P \xrightarrow{sca} P \mid \mathbf{0} \\
(nudown) & \nu z.D[P] \xrightarrow{sca} D[\nu z.P], \text{ if } z \notin \mathsf{fn}(D), [\cdot] \neq D \in \mathcal{D} & (rem0r) & P \mid \mathbf{0} \xrightarrow{sca} P
\end{array}
$$

The relations $\xrightarrow{\mathcal{D},sc}$ and $\xrightarrow{\mathcal{C},sca}$ are defined as:

$$
\frac{P \xrightarrow{sc} Q}{D[P] \xrightarrow{\mathcal{D},sc} D[Q]} \text{ where } D \in \mathcal{D} \qquad \frac{P \xrightarrow{sca} Q}{C[P] \xrightarrow{\mathcal{C},sca} C[Q]} \text{ where } C \in \mathcal{C}
$$

We sometimes add more information on the reduction arrow, and e.g. write $\xrightarrow{\mathcal{D},sc,nuup,*}$ for a (maybe empty) finite sequence of $\xrightarrow{\mathcal{D},sc}$-transformations which all apply the rule $(nuup)$.

▶ **Lemma 3.2.** *The relation* $\xrightarrow{\mathcal{C},sca,*}$ *coincides with structural congruence, i.e.* $\xrightarrow{\mathcal{C},sca,*} = \equiv$.

Since $\xrightarrow{\mathcal{C},sca,*}$ and $\equiv$ coincide, we can replace $\equiv$ in the definition of standard reduction:

▶ **Corollary 3.3.** $P \xrightarrow{sr} Q$ *iff* $P \xrightarrow{\mathcal{C},sca,*} P' \;\wedge\; P' \xrightarrow{\mathcal{D},ia} Q' \;\wedge\; Q' \xrightarrow{\mathcal{C},sca,*} Q$.

We define our new variant of standard reduction, called $\mathcal{D}$-standard reduction, which restricts the conversion w.r.t. structural congruence to $\xrightarrow{\mathcal{D},sc}$-transformations:

▶ **Definition 3.4** ($\mathcal{D}$-Standard Reduction, $\xrightarrow{dsr}$)**.** A *$\mathcal{D}$-standard reduction* $\xrightarrow{dsr}$ applies the rule (ia) in a reduction context $D \in \mathcal{D}$ modulo $\xrightarrow{\mathcal{D},sc,*}$, i.e. :

$$
\frac{P \xrightarrow{\mathcal{D},sc,*} P' \;\wedge\; P' \xrightarrow{\mathcal{D},ia} Q' \;\wedge\; Q' \xrightarrow{\mathcal{D},sc,*} Q}{P \xrightarrow{dsr} Q}
$$

▶ **Example 3.5.** We consider the same process as in Example 2.7.

$$\begin{aligned} & \nu x.(\overline{x}\langle w\rangle.\mathbf{0} \mid x(y).\overline{z}\langle y\rangle.\mathbf{0}) \mid !\, z(u).\mathbf{0} \xrightarrow{\mathcal{D},sc} \nu x.(x(y).\overline{z}\langle y\rangle.\mathbf{0} \mid \overline{x}\langle w\rangle.\mathbf{0}) \mid !\, z(u).\mathbf{0} \\ \xrightarrow{\mathcal{D},ia} \; & \nu x.(\overline{z}\langle w\rangle.\mathbf{0} \mid \mathbf{0}) \mid !\, z(u).\mathbf{0} \xrightarrow{\mathcal{D},sc,5} \nu x.(\mathbf{0} \mid ((\overline{z}\langle w\rangle.\mathbf{0} \mid z(u).\mathbf{0}) \mid !\, z(u).\mathbf{0})) \\ \xrightarrow{\mathcal{D},ia} \; & \nu x.(\mathbf{0} \mid ((\mathbf{0} \mid \mathbf{0}) \mid !\, z(u).\mathbf{0})) \end{aligned}$$

Note that $\mathcal{D}$-standard reduction cannot remove the $\mathbf{0}$-components and the ν-binder.

Replacing standard reduction by $\mathcal{D}$-standard reduction results in modified observation predicates and a modified definition of may- and should-testing equivalence. However, we will show that the modified equivalence coincides with the original one in Theorem 4.13.

▶ **Definition 3.6.** For $\mu \in \{x, \overline{x}\}$, P *may have a barb w.r.t.* $\xrightarrow{dsr}$ *on* x (written as $P{\downarrow_{\mathcal{D},\mu}}$) iff $P \xrightarrow{dsr,*} Q \xrightarrow{\mathcal{D},sc,*} Q'$ and $Q' \not\mapsto^{\mu}$, and P *should have a barb w.r.t.* $\xrightarrow{dsr}$ *on* x (written as $P{\Downarrow_{\mathcal{D},\mu}}$) iff for all processes P' such that $P \xrightarrow{dsr,*} P'$, also $P'{\downarrow_{\mathcal{D},\mu}}$ holds. We write $P{\uparrow_{\mathcal{D},\mu}}$ iff $\neg(P{\Downarrow_{\mathcal{D},\mu}})$ holds and $P \Uparrow_{\mathcal{D},\mu}$ iff $\neg(P \downarrow_{\mathcal{D},\mu})$ holds. Processes P and Q are *barbed testing equivalent w.r.t.* $\xrightarrow{dsr}$ (written $P \sim_{\mathcal{D}} Q$) iff $P \precsim_{\mathcal{D}} Q \wedge Q \precsim_{\mathcal{D}} P$, where $\precsim_{\mathcal{D}} := \precsim_{\mathcal{D},\text{may}} \cap \precsim_{\mathcal{D},\text{should}}$ and

$$\begin{aligned} P \precsim_{\mathcal{D},\text{may}} Q \quad & \text{iff } \forall x \in \mathcal{N},\ \mu \in \{x, \overline{x}, C \in \mathcal{C}\}\colon\ C[P]{\downarrow_{\mathcal{D},\mu}} \implies C[Q]{\downarrow_{\mathcal{D},\mu}} \\ P \precsim_{\mathcal{D},\text{should}} Q & \text{ iff } \forall x \in \mathcal{N},\ \mu \in \{x, \overline{x}\}, C \in \mathcal{C}\}\colon\ C[P]{\Downarrow_{\mathcal{D},\mu}} \implies C[Q]{\Downarrow_{\mathcal{D},\mu}} \end{aligned}$$

Note that $P{\uparrow_{\mathcal{D},\mu}}$ is equivalent to: $\exists Q : P \xrightarrow{dsr,*} Q \wedge Q \Uparrow_{\mathcal{D},\mu}$. Our main result will be that $\sim$ and $\sim_{\mathcal{D}}$ coincide and thus $\mathcal{D}$-standard reduction can be used for reasoning about process equivalence. For showing $\sim = \sim_{\mathcal{D}}$ it is sufficient to prove that ${\downarrow_{\mu}} = {\downarrow_{\mathcal{D},\mu}}$ and ${\Downarrow_{\mu}} = {\Downarrow_{\mathcal{D},\mu}}$.

4 Relating Reduction Strategies

As a required notation we define conversions w.r.t. $\equiv$ that are not included in $\xrightarrow{\mathcal{D},sc}$:

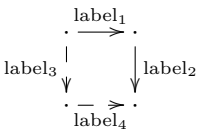
▶ **Definition 4.1** (Internal $\xrightarrow{\mathcal{C},sca}$-Transformations, $\xrightarrow{isca}$)**.** With $\xrightarrow{isca}$ we denote a $\xrightarrow{\mathcal{C},sca}$ transformation that is not a $\xrightarrow{\mathcal{D},sc}$ transformation, i.e. $\xrightarrow{isca} = \xrightarrow{\mathcal{C},sca} \setminus \xrightarrow{\mathcal{D},sc}$. With $\xrightarrow{isca\langle k\rangle}$ we denote a $\xrightarrow{isca}$-transformation at replication depth k, i.e. k is the number of replications above the redex of the $\xrightarrow{isca}$-transformation.

In the remainder of this section we establish our main result by showing that $\sim$ and $\sim_{\mathcal{D}}$ coincide. The proof is structured into three parts: in Sect. 4.1 we show that for a sequence of standard reductions $P \xrightarrow{sr,*} Q$ the internal conversions $\xrightarrow{isca}$ can be shifted to the right resulting in a reduction $P \xrightarrow{dsr,*} Q' \xrightarrow{isca,*} Q$. In Sect. 4.2 we show that if a process has a barb w.r.t. $\equiv$, i.e. $P \equiv \not\mapsto^{\mu}$, then we can remove internal conversions $\xrightarrow{isca}$ and thus $P \xrightarrow{\mathcal{D},sc,*} P'$ s.t. $P' \not\mapsto^{\mu}$. In Sect. 4.3 we use the results of both previous sections to show ${\downarrow_{\mu}} = {\downarrow_{\mathcal{D},\mu}}$ and ${\Downarrow_{\mu}} = {\Downarrow_{\mathcal{D},\mu}}$ which implies the coincidence of $\sim$ and $\sim_{\mathcal{D}}$.

4.1 Shifting Internal Conversions to the End

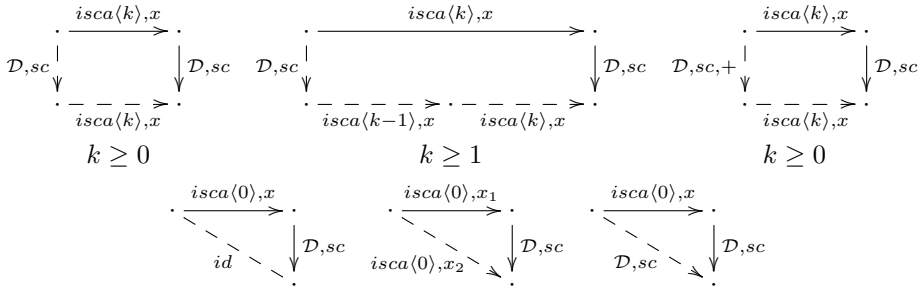
In this section we show that for every standard reduction sequence $P \xrightarrow{sr,*} Q$, also a $\mathcal{D}$-standard reduction sequence followed by internal structural conversion steps exists, i.e. $P \xrightarrow{dsr,*} Q' \xrightarrow{isca,*} Q$. The result is established by inspecting overlappings of the forms $P_1 \xrightarrow{isca} P_2 \xrightarrow{\mathcal{D},sc} P_3$ and $P_1 \xrightarrow{isca} P_2 \xrightarrow{\mathcal{D},ia} P_3$, where in both cases the $\xrightarrow{isca}$-transformation must be commuted with the other reduction or transformation.

We will use so-called commuting diagrams as a notation, which are diagrams of the form as shown on the right. Every arrow represents a transformation step or a sequence of steps denoted by $*$ (0 or more steps) or $+$ (1 or more steps) as part of the label. Labels denote the used reduction rule, solid arrows mean given transformations, dashed arrows are existentially quantified transformations. If the label is *id*, then this means that processes are identical. Labels may include variables x, x_i for the name of a used rule. These variables are meant universally quantified for the diagram, i.e. all occurrences of x in one diagram can only be instantiated with the same rule name x, and for diagrams with different variables (e.g. x_1 and x_2) any variable can be instantiated with different (or equal) rule names. If not otherwise stated x, x_1, x_2 may be instantiated with (*assocl*), (*assocr*), (*commute*), (*nucomm*), (*nudownr*), (*nuup*), (*replunfold*), (*replfold*), (*nuintro*), (*intro0l*), (*intro0r*), (*rem0r*), or (*nurem*).



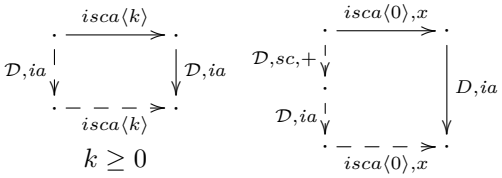
Inspecting all overlappings between a $\xrightarrow{\mathcal{D},sc}$- and an $\xrightarrow{isca}$-step shows:

▶ **Lemma 4.2.** *Given a sequence $P_1 \xrightarrow{isca} P_2 \xrightarrow{\mathcal{D},sc} P_3$, then the sequence can always be commuted by one of the following diagrams:*



Inspecting all overlappings between a $\xrightarrow{\mathcal{D},ia}$- and an $\xrightarrow{isca}$-step

▶ **Lemma 4.3.** *Given a sequence $P_1 \xrightarrow{isca} P_2 \xrightarrow{\mathcal{D},ia} P_3$, then the sequence can always be commuted by one of the following diagrams:*



Using the diagrams of the two previous lemmas we are able to show that in a $\xrightarrow{sr}$-reduction sequence all $\xrightarrow{isca}$-steps can always be shifted to the right end.

▶ **Proposition 4.4.** *Let $P_1 \xrightarrow{a_1} P_2 \xrightarrow{a_2} \ldots \xrightarrow{a_{n-1}} P_n$ where every $\xrightarrow{a_i}$ is either an $\xrightarrow{\mathcal{C},sca}$ or a $\xrightarrow{\mathcal{D},ia}$ transformation. Then there exists $P_1 \xrightarrow{b_1} Q_1 \xrightarrow{b_2} \ldots \xrightarrow{b_m} Q_m \xrightarrow{isca,*} P_n$ where every $\xrightarrow{b_i}$ is either a $\xrightarrow{\mathcal{D},ia}$ or a $\xrightarrow{\mathcal{D},sc}$ transformation.*

Proof. In the input sequence (and also intermediate sequences in the proof) we can switch to the representation where $\xrightarrow{a_i}$ is either an $\xrightarrow{isca\langle k\rangle}$ (for some $k \geq 0$) or a $\xrightarrow{\mathcal{D},ia}$, or a $\xrightarrow{\mathcal{D},sc}$ transformation. We represent the two latter cases uniformly by $\xrightarrow{\mathcal{D},sc\vee ia}$ and thus only have to deal with sequences consisting of $\xrightarrow{isca\langle k\rangle}$- and $\xrightarrow{\mathcal{D},sc\vee ia}$-steps. The diagrams of Lemmas 4.2 and 4.3 can be seen as rewriting rules on such sequences, where we also simplify

the representation resulting in the rules:

$$\xrightarrow{isca\langle k\rangle} . \xrightarrow{\mathcal{D},sc\vee ia} \rightsquigarrow \xrightarrow{\mathcal{D},sc\vee ia} . \xrightarrow{isca\langle k-1\rangle} . \xrightarrow{isca\langle k\rangle} \text{ for } k \geq 1 \quad (1)$$

$$\xrightarrow{isca\langle k\rangle} . \xrightarrow{\mathcal{D},sc\vee ia} \rightsquigarrow \xrightarrow{\mathcal{D},sc\vee ia,n} . \xrightarrow{isca\langle k\rangle} \text{ for } k \geq 0 \text{ and any } n \geq 1 \quad (2)$$

$$\xrightarrow{isca\langle 0\rangle} . \xrightarrow{\mathcal{D},sc\vee ia} \rightsquigarrow \varepsilon \text{ (where } \varepsilon \text{ represents the empty string)} \quad (3)$$

$$\xrightarrow{isca\langle 0\rangle} . \xrightarrow{\mathcal{D},sc\vee ia} \rightsquigarrow \xrightarrow{isca\langle 0\rangle} \quad (4)$$

$$\xrightarrow{isca\langle 0\rangle} . \xrightarrow{\mathcal{D},sc\vee ia} \rightsquigarrow \xrightarrow{\mathcal{D},sc\vee ia} \quad (5)$$

Let $\mathfrak{R} = P_1 \xrightarrow{a_1} P_2 \xrightarrow{a_2} \ldots \xrightarrow{a_{n-1}} P_n$. We define a measure $\mu(\mathfrak{R})$ for those reduction sequences: let $\mathsf{P}(\mathfrak{R})$ be the longest prefix of $\mathfrak{R}$ that has an $\xrightarrow{\mathcal{D},sc\vee ia}$-step as its last step, or the empty sequence, if there is no $\xrightarrow{\mathcal{D},sc\vee ia}$-step in $\mathfrak{R}$. W.l.o.g. $\mathsf{P}(\mathfrak{R}) = P_1 \xrightarrow{a_1} P_2 \xrightarrow{a_2} \ldots \xrightarrow{a_{j-1}} P_j$. Now the multiset $\mu(\mathfrak{R})$ is constructed by inserting the pair $\#(P_i \xrightarrow{isca\langle k\rangle} P_{i+1})$ for every reduction step $P_i \xrightarrow{isca\langle k\rangle} P_{i+1}$ in $\mathsf{P}(\mathfrak{R})$ (and inserting no pairs for steps $P_i \xrightarrow{\mathcal{D},sc\vee ia} P_{i+1}$), where

$$\#(P_i \xrightarrow{isca\langle k\rangle} P_{i+1}) = \begin{cases} (k,\infty), \text{ if } \xrightarrow{isca\langle k\rangle} \text{ is not the last } \xrightarrow{isca} \text{ reduction in } \mathsf{P}(\mathfrak{R}) \\ (k,d), \quad \text{otherwise, where } d = j-i-1 \text{ (i.e. } d \text{ is the number of} \\ \qquad\quad \text{reductions in } \mathsf{P}(\mathfrak{R}) \text{ that follow after } P_i \xrightarrow{isca\langle k\rangle} P_{i+1}) \end{cases}$$

We use the multiset ordering, where pairs are ordered lexicographically. The ordering is well-founded and if the multiset $\mu(\mathfrak{R})$ is empty, then $\mathsf{P}(\mathfrak{R})$ does not contain $\xrightarrow{isca}$-transformations. We show the claim by induction on $\mu(\mathfrak{R})$: If $\mu(\mathfrak{R})$ is empty then $\mathsf{P}(\mathfrak{R})$ only contains $\xrightarrow{\mathcal{D},sc\vee ia}$-steps (or is empty). Hence $\mathfrak{R}$ is of the right form and the claim holds. For the induction step assume that $\mu(\mathfrak{R})$ contains at least one pair, hence $\mathsf{P}(\mathfrak{R})$ has at least a subsequence $\xrightarrow{isca}\xrightarrow{\mathcal{D},sc\vee ia}$. Let $\mathsf{P}(\mathfrak{R}) = P_1 \xrightarrow{a_1} \ldots \xrightarrow{a_i} P_i \xrightarrow{isca\langle k\rangle} P_{i+1} \xrightarrow{\mathcal{D},sc\vee ia} P_{i+2} \xrightarrow{\mathcal{D},sc\vee ia,m} P_{i+2+m}$ and $\mathfrak{R} = \mathsf{P}(\mathfrak{R}) \xrightarrow{isca,*} P_n$. We apply one of the rewriting rules derived from the diagrams to the subsequence $P_i \xrightarrow{isca\langle k\rangle} P_{i+1} \xrightarrow{\mathcal{D},sc\vee ia} P_{i+2}$, Let $\mathfrak{R}'$ be the sequence $\mathfrak{R}$ after applying the rewrite rule. We inspect the cases for all five rules (1)–(5):

(1) $\mathfrak{R}' = P_1 \xrightarrow{a_1} \ldots \xrightarrow{a_i} P_i \xrightarrow{\mathcal{D},sc\vee ia} P' \xrightarrow{isca\langle k-1\rangle} P'' \xrightarrow{isca\langle k\rangle} P_{i+2} \xrightarrow{\mathcal{D},sc\vee ia,m} P_{i+2+m} \xrightarrow{isca,*} P_n$. For $\mu(\mathfrak{R}')$ compared to $\mu(\mathfrak{R})$ there are two possibilities:

- If $m = 0$, then $\mathsf{P}(\mathfrak{R}') = P_1 \xrightarrow{a_1} \ldots \xrightarrow{a_i} P_i \xrightarrow{\mathcal{D},sc\vee ia} P'$ and a pair $(k,1)$ is removed and perhaps some other pair (k,∞) is replaced by (k,d) for some $d < \infty$.
- If $m > 0$, then $\mathsf{P}(\mathfrak{R}') = P_1 \xrightarrow{a_1} \ldots \xrightarrow{a_i} P_i \xrightarrow{\mathcal{D},sc\vee ia} P' \xrightarrow{isca\langle k-1\rangle} P'' \xrightarrow{isca\langle k\rangle} P_{i+2} \xrightarrow{\mathcal{D},sc\vee ia,m} P_{i+2+m}$ and a pair $(k,m+1)$ is replaced by two pairs (k,m) and $(k-1,\infty)$.

In both cases $\mu(\mathfrak{R}') < \mu(\mathfrak{R})$ and thus the induction hypothesis can be applied to $\mathfrak{R}'$.

(2) $\mathfrak{R}' = P_1 \xrightarrow{a_1} \ldots \xrightarrow{a_i} P_i \xrightarrow{\mathcal{D},sc\vee ia,m'} P' \xrightarrow{isca\langle k\rangle} P_{i+2} \xrightarrow{\mathcal{D},sc\vee ia,m} P_{i+2+m} \xrightarrow{isca,*} P_n$ where $m' \geq 1$. For $\mu(\mathfrak{R}')$ compared to $\mu(\mathfrak{R})$ there are two possibilities:

- If $m = 0$, then $\mathsf{P}(\mathfrak{R}') = P_1 \xrightarrow{a_1} \ldots \xrightarrow{a_i} P_i \xrightarrow{\mathcal{D},sc\vee ia,m'} P'$ and a pair $(k,1)$ is removed and perhaps some other pair (k,∞) is replaced by (k,d) where $d < \infty$.
- If $m > 0$, then $\mathsf{P}(\mathfrak{R}') = P_1 \xrightarrow{a_1} \ldots \xrightarrow{a_i} P_i \xrightarrow{\mathcal{D},sc\vee ia,m'} P' \xrightarrow{isca\langle k\rangle} P_{i+2} \xrightarrow{\mathcal{D},sc\vee ia,m} P_{i+2+m}$ and a pair $(k,m+1)$ is replaced by (k,m).

In both cases $\mu(\mathfrak{R}') < \mu(\mathfrak{R})$ and thus the induction hypothesis can be applied to $\mathfrak{R}'$.

(3) $\mathfrak{R}' = P_1 \xrightarrow{a_1} \ldots \xrightarrow{a_i} P_i = P_{i+2} \xrightarrow{\mathcal{D},sc\vee ia,m} P_{i+2+m} \xrightarrow{isca,*} P_n$. For $\mu(\mathfrak{R}')$ compared to $\mu(\mathfrak{R})$ there are two possibilities:

- If $m = 0$, then $\mathsf{P}(\mathfrak{R}')$ is a prefix of $P_1 \xrightarrow{a_1} \ldots \xrightarrow{a_i} P_i$ and a pair $(k, 1)$ is removed and perhaps a pair (k, ∞) is replaced by (k, d) where $d < \infty$.
- If $m > 0$, then $\mathsf{P}(\mathfrak{R}') = P_1 \xrightarrow{a_1} \ldots \xrightarrow{a_i} P_i = P_{i+2} \xrightarrow{\mathcal{D},sc\vee ia,m} P_{i+2+m}$ and a pair $(k, m+1)$ is removed, and perhaps a pair (k, ∞) is replaced by (k, d) for some $d < \infty$.

In both cases $\mu(\mathfrak{R}') < \mu(\mathfrak{R})$ and thus the induction hypothesis can be applied to $\mathfrak{R}'$.

(4) $\mathfrak{R}' = P_1 \xrightarrow{a_1} \ldots \xrightarrow{a_i} P_i \xrightarrow{isca\langle k\rangle} P_{i+2} \xrightarrow{\mathcal{D},sc\vee ia,m} P_{i+2+m} \xrightarrow{isca,*} P_n$. For $\mu(\mathfrak{R}')$ compared to $\mu(\mathfrak{R})$ there are two possibilities:

- If $m = 0$, then $\mathsf{P}(\mathfrak{R}')$ is a prefix of $P_1 \xrightarrow{a_1} \ldots \xrightarrow{a_i} P_i$ and a pair $(k, 1)$ is removed and perhaps some other pair (k, ∞) is replaced by (k, d) where $d < \infty$.
- If $m > 0$, then $\mathsf{P}(\mathfrak{R}') = P_1 \xrightarrow{a_1} \ldots \xrightarrow{a_i} P_i \xrightarrow{isca\langle k\rangle} P_{i+2} \xrightarrow{\mathcal{D},sc\vee ia,m} P_{i+2+m}$ and a pair $(k, m+1)$ is replaced by (k, m).

(5) $\mathfrak{R}' = P_1 \xrightarrow{a_1} \ldots \xrightarrow{a_i} P_i \xrightarrow{\mathcal{D},sc\vee ia} P_{i+2} \xrightarrow{\mathcal{D},sc\vee ia,m} P_{i+2+m} \xrightarrow{isca,*} P_n$. For $\mu(\mathfrak{R}')$ compared to $\mu(\mathfrak{R})$ there are two possibilities:

- $\mathsf{P}(\mathfrak{R}')$ is empty, since all $\xrightarrow{a_j}$-steps are $\xrightarrow{\mathcal{D},sc\vee ia}$-steps. Then $\mathfrak{R}'$ is of the right form.
- $\mathsf{P}(\mathfrak{R}') = P_1 \xrightarrow{a_1} \ldots \xrightarrow{a_i} P_i \xrightarrow{\mathcal{D},sc\vee ia} P_{i+2} \xrightarrow{\mathcal{D},sc\vee ia,m} P_{i+2+m}$. Then a pair (k, m) is removed and another pair (k, ∞) is replaced by a pair (k, d) where $d < \infty$ (and $d > m$). In this case $\mu(\mathfrak{R}') < \mu(\mathfrak{R})$ and thus we can apply the induction hypothesis to $\mathfrak{R}'$. ◀

▶ **Remark 4.5.** Termination of the rewriting in the previous proof can also be shown automatically by encoding the rules as an (extended) term rewriting system (TRS) and then using a termination prover. Let K,N, and X be variables, S, Z be constructor symbols encoding Peano-numbers, and *isca*, *dscdia*, and *gen* be defined function symbols. The encoding of the rules (1)–(5) is straight-forward, except for rule (2) due to the constraint $n \geq 1$. It is encoded by three TRS-rules (2), (2'), (2"), where the first rule "guesses" the number n, and the rules (2') and (2") generate the corresponding number of $\xrightarrow{\mathcal{D},sc\vee ia}$-steps:

$$\begin{aligned}
isca(S(k), dscdia(\mathsf{X})) &\to dscdia(isca(\mathsf{K}, isca(S(\mathsf{K}), \mathsf{X}))) && (1)\\
isca(\mathsf{K}, dscdia(\mathsf{X})) &\to gen(S(\mathsf{N}), isca(\mathsf{K}, \mathsf{X})) && (2)\\
gen(S(\mathsf{N}), \mathsf{X}) &\to dscdia(gen(\mathsf{N}, \mathsf{X})) && (2')\\
gen(Z, \mathsf{X}) &\to \mathsf{X} && (2'')\\
isca(Z, dscdia(\mathsf{X})) &\to \mathsf{X} && (3)\\
isca(Z, dscdia(\mathsf{X})) &\to isca(Z, \mathsf{X}) && (4)\\
isca(Z, dscdia(\mathsf{X})) &\to dscdia(Z, \mathsf{X}) && (5)
\end{aligned}$$

Thus the TRS has rules with free variables on the right hand side. However, the termination prover AProVE [5] has shown innermost-termination of the TRS (which is sufficient), and the certifier CeTA [24] has certified the termination proof[1]. A similar encoding approach was already used for automating correctness proofs for program transformations in [15].

We conclude this subsection by showing that also the number of (ia)-reductions is preserved by shifting internal transformations:

▶ **Theorem 4.6.** ***1.*** *If $P \xrightarrow{dsr,n} Q$ then $P \xrightarrow{sr,n} Q$.* ***2.*** *If $P \xrightarrow{sr,n} Q$ then $P \xrightarrow{dsr,n} Q' \xrightarrow{isca,*} Q$.*

Proof. Part (1) holds, since every $\xrightarrow{dsr}$-step is also an $\xrightarrow{sr}$-step. For part (2), let $P = P_0 \xrightarrow{sr} P_1 \ldots \xrightarrow{sr} P_n = Q$. Corollary 3.3 shows that for $\leq i \leq n-1$ $P_i \xrightarrow{sr} P_{i+1}$ can be written as $P_i \xrightarrow{\mathcal{C},sca,*} P_i' \xrightarrow{\mathcal{D},ia} P_i'' \xrightarrow{\mathcal{C},sca,*} P_{i+1}$ for some processes P_i', P_i'' and thus $P = P_0 \xrightarrow{\mathcal{C},sca,*} P_0' \xrightarrow{\mathcal{D},ia} P_0'' \xrightarrow{\mathcal{C},sca,*} P_1 \ldots P_{n-1} \xrightarrow{\mathcal{C},sca,*} P_{n-1}' \xrightarrow{\mathcal{D},ia} P_{n-1}'' \xrightarrow{\mathcal{C},sca,*} P_n = Q$. Proposition 4.4 shows that there exists a process Q' s.t. $P \xrightarrow{\mathcal{D},ia\vee sc,*} Q' \xrightarrow{isca,*} Q$. The

[1] the termination proof is available at http://www.ki.cs.uni-frankfurt.de/persons/sabel/picalc.html

construction in the proof of Proposition 4.4 together with the diagrams in Lemmas 4.2 and 4.3 imply that no $\xrightarrow{\mathcal{D},ia}$ transformation is eliminated or introduced. Thus $P \xrightarrow{\mathcal{D},ia\vee sc,*} Q'$ must contain exactly n $\xrightarrow{\mathcal{D},ia}$-transformations and thus it is also a sequence $P \xrightarrow{dsr,n} Q'$. ◀

4.2 Removing Internal Conversions from Barbs

In this section we show that $P \equiv \Gamma^{\mu}$ also implies that $P \xrightarrow{\mathcal{D},sc,*} P'$ s.t. $P' \Gamma^{\mu}$. Let $\mathcal{F}$-contexts be the class of contexts that do not have a hole below an input- or output prefix, i.e.

$$F \in \mathcal{F} ::= [\cdot] \mid !\,F \mid F \mid P \mid P \mid F \mid \nu x.F \quad \text{where } x \in \mathcal{N} \text{ and } P \text{ is a process}$$

We say that $F \in \mathcal{F}$ *captures* the name $x \in \mathcal{N}$ iff the hole of F is in scope of a restriction νx.

▶ **Lemma 4.7.** *If $P \equiv \nu\mathcal{X}.(\pi.Q \mid R)$, then there exists $F \in \mathcal{F}$, a prefix π', and process Q', s.t. $P = F[\pi'.Q']$ and $\mathsf{fn}(\pi) \cap \mathcal{X} = \mathsf{fn}(\pi') \cap \mathsf{fn}(P)$.*

Proof. Let $P_0 \xrightarrow{\mathcal{C},sca,n} P_n = \nu\mathcal{X}.(\pi.Q \mid R)$. We use induction on n. If $n = 0$ then the claim holds, since $\nu\mathcal{X}.([\cdot] \mid R)$ is an $\mathcal{F}$-context. If $n > 0$ let $P_0 \xrightarrow{\mathcal{C},sca} P_1 \xrightarrow{\mathcal{C},sca,n-1} \nu\mathcal{X}.(\pi.Q \mid R)$. The induction hypothesis shows that $P_1 = F_1[\pi_1.Q_1]$ s.t. $\mathsf{fn}(\pi) \cap \mathcal{X} = \mathsf{fn}(\pi_1) \cap \mathsf{fn}(P_1)$. Inspecting all possibilities for the step $P_0 \xrightarrow{\mathcal{C},sca} P_1$, shows that $P_0 = F_0[\pi_0.Q_0]$ for some $\mathcal{F}$-context F_0, and where π_0 is π_1 but perhaps with a renaming of variables due to α-renaming. However, $\xrightarrow{\mathcal{C},sca}$-transformation can neither capture free names nor move bound names out of their scope, and thus the claim of the lemma holds. ◀

▶ **Lemma 4.8.** *If $P = F[\pi.Q]$ for some $F \in \mathcal{F}$, prefix π, and process Q, then $P \xrightarrow{\mathcal{D},sc,*} \nu\mathcal{X}.(\pi'.Q' \mid R)$ s.t. $\mathsf{fn}(\pi) \cap \mathsf{fn}(F[\pi.Q]) = \mathsf{fn}(\pi') \cap \mathcal{X}$.*

Proof. We use structural induction on F. If F is empty, then P is of the required form.

- If $F = F' \mid R$ then by the induction hypothesis (and since $[\cdot] \mid R$ is a $\mathcal{D}$-context): $F'[\pi.Q] \mid R \xrightarrow{\mathcal{D},sc,*} (\nu\mathcal{X}.(\pi'.Q' \mid R')) \mid R$ s.t. $\mathsf{fn}(\pi) \cap \mathsf{fn}(F[\pi.Q]) = \mathsf{fn}(\pi') \cap \mathcal{X}$. Suppose $\mathcal{X} = \{x_1, \ldots, x_m\}$ and let $\mathcal{Y} := \{y_1, \ldots, y_m\}$ be fresh names, and $[\vec{y}/\vec{x}]$ be the substitution $[y_1/x_1, \ldots, y_m/x_m]$. We can extend this reduction as follows: $(\nu\mathcal{X}.(\pi'.Q' \mid R')) \mid R$ $=_\alpha (\nu\mathcal{Y}.(\pi'[\vec{y}/\vec{x}].Q'[\vec{y}/\vec{x}] \mid R'[\vec{y}/\vec{x}])) \mid R \xrightarrow{\mathcal{D},sc,nuup,*} (\nu\mathcal{Y}.((\pi'[\vec{y}/\vec{x}].Q'[\vec{y}/\vec{x}] \mid R'[\vec{y}/\vec{x}]) \mid R))$ $\xrightarrow{\mathcal{D},sc,assocr} (\nu\mathcal{Y}.(\pi'[\vec{y}/\vec{x}].Q'[\vec{y}/\vec{x}] \mid (R'[\vec{y}/\vec{x}] \mid R)))$. Clearly, only bound names of π' are renamed and thus the condition on the names holds.
- If $F = R \mid F'$ then the reasoning is analogous to the previous case.
- If $F = \nu x.(F')$ then by the induction hypothesis $(F'[\pi.Q]) \xrightarrow{\mathcal{D},sc,*} (\nu\mathcal{X}.(\pi'.Q' \mid R'))$ where $\mathsf{fn}(\pi) \cap \mathsf{fn}(F'[\pi.Q]) = \mathsf{fn}(\pi') \cap \mathcal{X}$. The same reduction can be performed in the $\mathcal{D}$-context $\nu x.([\cdot])$: $\nu x.(F'[\pi.Q]) \xrightarrow{\mathcal{D},sc,*} \nu x.(\nu\mathcal{X}.(\pi'.Q' \mid R'))$ and occurrences of x in π and π' are captured for both processes.
- If $F = !\,F'$ then by the induction hypothesis $F'[\pi.Q] \xrightarrow{\mathcal{D},sc,*} (\nu\mathcal{X}.(\pi'.Q' \mid R'))$ where $\mathsf{fn}(\pi) \cap \mathsf{fn}(F'[\pi.Q]) = \mathsf{fn}(\pi') \cap \mathcal{X}$. Suppose that $\mathcal{X} = \{x_1, \ldots, x_m\}$ and let $\mathcal{Y} = \{y_1, \ldots, y_m\}$ be fresh names, $[\vec{y}/\vec{x}]$ be the substitution $[y_1/x_1, \ldots, y_m/x_m]$, and P_0 be an α-renamed copy of $F'[\pi.Q]$. Then $!\,F'[\pi.Q] \xrightarrow{\mathcal{D},sc,replunfold} F'[\pi.Q] \mid !\,P_0 \xrightarrow{\mathcal{D},sc,*} (\nu\mathcal{X}.(\pi'.Q' \mid R')) \mid !\,P_0$ $=_\alpha (\nu\mathcal{Y}.(\pi'[\vec{y}/\vec{x}].Q'[\vec{y}/\vec{x}] \mid R'[\vec{y}/\vec{x}])) \mid !\,P_0 \xrightarrow{\mathcal{D},sc,nuup,*} \nu\mathcal{Y}.((\pi'[\vec{y}/\vec{x}].Q'[\vec{y}/\vec{x}] \mid R'[\vec{y}/\vec{x}]) \mid !\,P_0)$ $\xrightarrow{\mathcal{D},sc,assocr} \nu\mathcal{Y}.(\pi'[\vec{y}/\vec{x}].Q'[\vec{y}/\vec{x}] \mid (R'[\vec{y}/\vec{x}] \mid !\,P_0))$. The last process is of the required form and free names of π' remain free in $\pi'[\vec{y}/\vec{x}]$. ◀

Combining Lemmas 4.7 and 4.8 results in:

▶ **Theorem 4.9.** *If* $P \equiv \nu\mathcal{X}.(\pi.Q \mid R)$ *then* $P \xrightarrow{\mathcal{D},sc,*} \nu\mathcal{X}'.(\pi'.Q' \mid R')$ *s.t.* $\mathsf{fn}(\pi) \cap \mathcal{X} = \mathsf{fn}(\pi') \cap \mathcal{X}'$.

▶ **Corollary 4.10.** *For all* $x \in \mathcal{N}$ *and* $\mu \in \{x, \overline{x}\}$*:* $P \equiv\!\!\Rsh^{\mu}$ *iff* $P \xrightarrow{\mathcal{D},sc,*} Q$ *and* $Q \Rsh^{\mu}$.

Proof. This follows since $\xrightarrow{\mathcal{D},sc,*} \subset \equiv$, and from Theorem 4.9. ◀

4.3 Coincidence of $\sim$ and $\sim_{\mathcal{D}}$

We now prove our main result, by first showing that the observation predicates $\downarrow_{\mu}$,$\Downarrow_{\mu}$ remain unchanged if $\xrightarrow{sr}$ is replaced by $\xrightarrow{dsr}$. This implies that $\sim = \sim_{\mathcal{D}}$.

▶ **Proposition 4.11.** $\downarrow_{\mu} = \downarrow_{\mathcal{D},\mu}$.

Proof. We have to show two parts:

- If $P\downarrow_{\mathcal{D},\mu}$, then $P \xrightarrow{dsr,n} Q \xrightarrow{\mathcal{D},sc,*} Q'$ and $Q' \Rsh^{\mu}$. Theorem 4.6 shows that $P \xrightarrow{sr,n} Q$ and Corollary 4.10 shows that $Q \equiv\!\!\Rsh^{\mu}$. Thus we have $P\downarrow_{\mu}$.
- If $P\downarrow_{\mu}$ then $P \xrightarrow{sr,n} Q \wedge Q \equiv\!\!\Rsh^{\mu}$. Theorem 4.6 shows that $P \xrightarrow{dsr,n} Q' \xrightarrow{isca,*} Q$, Lemma 3.2 implies that $Q' \equiv\!\!\Rsh^{\mu}$, and by Corollary 4.10 $Q' \xrightarrow{dsr,*} Q''$ s.t. $Q'' \Rsh^{\mu}$. This shows $P\downarrow_{\mathcal{D},\mu}$. ◀

▶ **Proposition 4.12.** $\Downarrow_{\mu} = \Downarrow_{\mathcal{D},\mu}$.

Proof. We show the converse, i.e. $\uparrow_{\mu} = \uparrow_{\mathcal{D},\mu}$. Proposition 4.11 already implies $\Uparrow_{\mu} = \Uparrow_{\mathcal{D},\mu}$.

- Let P be a process with $P\uparrow_{\mathcal{D},\mu}$. Then $P \xrightarrow{dsr,n} Q$ and $Q\Uparrow_{\mathcal{D},\mu}$. Theorem 4.6 shows that $P \xrightarrow{sr,n} Q$. Since $Q\Uparrow_{\mathcal{D},\mu} \iff Q\Uparrow_{\mu}$, this implies $P\uparrow_{\mu}$.
- If $P\uparrow_{\mu}$, then $P \xrightarrow{sr,n} Q$ and $Q\Uparrow_{\mu}$. Theorem 4.6 shows that $P \xrightarrow{dsr,n} Q' \xrightarrow{isca,*} Q$. This also implies $Q' \equiv Q$ and thus $Q'\Uparrow_{\mu}$. Since $Q'\Uparrow_{\mathcal{D},\mu} \iff Q'\Uparrow_{\mu}$, this shows $P\uparrow_{\mathcal{D},\mu}$. ◀

The definitions of $\sim$ and $\sim_{\mathcal{D}}$ only differ in the used observation predicates. In the two previous propositions we have shown, that the observation predicates are identical, and thus:

▶ **Theorem 4.13.** $\sim = \sim_{\mathcal{D}}$.

A consequence of the previous theorem and Proposition 2.11 is:

▶ **Corollary 4.14.** *If* $P \equiv Q$ *then* $P \sim_{\mathcal{D}} Q$.

5 Conclusion

We have defined a reduction strategy for the synchronous π-calculus which makes conversions w.r.t. structural congruence explicit by reduction rules, and uses a restricted set of those conversions. We have shown that the new reduction strategy does not change the semantics of processes w.r.t. may- and should-testing equivalence. For further research, we may use the new reduction strategy for proving correctness of process transformations, e.g. automated computation of overlappings between transformation steps and $\mathcal{D}$-standard reductions. Also extensions of the π-calculus, may be the topic of further research.

Acknowledgment. I thank Manfred Schmidt-Schauß for discussions and helpful comments on the paper. I thank René Thiemann for his great support on using AproVE and CeTA. I thank the anonymous reviewers for their valuable comments on the paper.

References

1 M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. In *CCS'97*, pages 36–47. ACM, 1997.
2 J. Engelfriet and T. Gelsema. A new natural structural congruence in the pi-calculus with replication. *Acta Inf.*, 40(6-7):385–430, 2004.
3 J. Engelfriet and T. Gelsema. An exercise in structural congruence. *Inf. Process. Lett.*, 101(1):1–5, 2007.
4 C. Fournet and G. Gonthier. A hierarchy of equivalences for asynchronous calculi. *J. Log. Algebr. Program.*, 63(1):131–173, 2005.
5 J. Giesl, P. Schneider-Kamp, and R. Thiemann. Automatic termination proofs in the dependency pair framework. In *IJCAR'06*, volume 4130 of *LNCS*, pages 281–286. Springer, 2006.
6 V. Khomenko and R. Meyer. Checking pi-calculus structural congruence is graph isomorphism complete. In *ACSD'09*, pages 70–79. IEEE, 2009.
7 C. Laneve. On testing equivalence: May and must testing in the join-calculus. Technical Report UBLCS 96-04, University of Bologna, 1996.
8 R. Milner. *Communicating and Mobile Systems: the π-calculus.* CUP, 1999.
9 R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, i & ii. *Inform. and Comput.*, 100(1):1–77, 1992.
10 J. H. Morris. *Lambda-Calculus Models of Programming Languages.* PhD thesis, MIT, 1968.
11 V. Natarajan and R. Cleaveland. Divergence and fair testing. In *ICALP'95*, volume 944 of *LNCS*, pages 648–659. Springer, 1995.
12 J. Niehren, D. Sabel, M. Schmidt-Schauß, and J. Schwinghammer. Observational semantics for a concurrent lambda calculus with reference cells and futures. *Electron. Notes Theor. Comput. Sci.*, 173:313–337, 2007.
13 G. D. Plotkin. Call-by-name, call-by-value, and the lambda-calculus. *Theoret. Comput. Sci.*, 1:125–159, 1975.
14 C. Priami. Stochastic pi-calculus. *Comput. J.*, 38(7):578–589, 1995.
15 C. Rau, D. Sabel, and M. Schmidt-Schauß. Correctness of program transformations as a termination problem. In *IJCAR'12*, volume 7364 of *LNCS*, pages 462–476. Springer, 2012.
16 A. Rensink and W. Vogler. Fair testing. *Inform. and Comput.*, 205(2):125–198, 2007.
17 D. Sabel and M. Schmidt-Schauß. A call-by-need lambda-calculus with locally bottom-avoiding choice: Context lemma and correctness of transformations. *Math. Structures Comput. Sci.*, 18(03):501–553, 2008.
18 D. Sabel and M. Schmidt-Schauß. A contextual semantics for Concurrent Haskell with futures. In *PPDP'11*, pages 101–112. ACM, 2011.
19 D. Sabel and M. Schmidt-Schauß. Conservative concurrency in Haskell. In *LICS'12*, pages 561–570. IEEE, 2012.
20 D. Sabel and M. Schmidt-Schauß. Contextual equivalence for the pi-calculus that can stop. Frank report 53, J. W. Goethe-Universität Frankfurt am Main, 2014. `http://www.ki.cs.uni-frankfurt.de/papers/frank/pi-stop-frank.pdf`.
21 D. Sangiorgi and D. Walker. *The π-calculus: a theory of mobile processes.* CUP, 2001.
22 M. Schmidt-Schauß, C. Rau, and D. Sabel. Algorithms for Extended Alpha-Equivalence and Complexity. In *RTA'13*, volume 21 of *LIPIcs*, pages 255–270. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2013.
23 M. Schmidt-Schauß and D. Sabel. Correctness of an STM Haskell implementation. In *ICFP'13*, pages 161–172. ACM, 2013.
24 R. Thiemann and C. Sternagel. Certification of termination proofs using CeTA. In *TPHOLs'09*, volume 5674 of *LNCS*, pages 452–468. Springer, 2009.

Contextual Equivalences in Call-by-Need and Call-By-Name Polymorphically Typed Calculi (Preliminary Report)

Manfred Schmidt-Schauß and David Sabel

Goethe-University, Frankfurt am Main
{schauss,sabel}@ki.cs.uni-frankfurt.de

Abstract

This paper presents a call-by-need polymorphically typed lambda-calculus with letrec, case, constructors and seq. The typing of the calculus is modelled in a system-F style. Contextual equivalence is used as semantics of expressions. We also define a call-by-name variant without letrec. We adapt several tools and criteria for recognizing correct program transformations to polymorphic typing, in particular an inductive applicative simulation.

1998 ACM Subject Classification F.4.2 Grammars and Other Rewriting Systems, F.3.2 Semantics of Programming Languages, D.3.1 Formal Definitions and Theory

Keywords and phrases functional programming, polymorphic typing, contextual equivalence, semantics

Digital Object Identifier 10.4230/OASIcs.WPTE.2014.63

1 Introduction

The goal of this paper is to present theoretical tools for recognizing correct program transformation in polymorphically typed, lazy functional programming languages like Haskell [4]. The intention is to take care of all program constructs of Haskell with operational significance. Thus the set of constructs like abstractions, applications, constructors, and case-expressions has to be extended by seq, the sequential-evaluation operator available in Haskell.

Our notion of correctness is based on the contextual equivalence, which equates programs if their termination behavior is identical if they are plugged in any surrounding larger program (i.e. any program context). Early work on the semantics of call-by-need evaluation can be found e.g. in [7]. Deep analyses of the contextual semantics of Haskell's core-language by investigating extended lambda-calculi were done e.g. in [8, 21, 20], but all these works consider the *untyped* variant of the core language.

In untyped calculi all program contexts have to be considered for the contextual equivalence while in typed calculi only typed programs are compared and only correctly typed contexts have to be considered. Hence in the typed setting the set of testing contexts is a subset of the used contexts in the untyped setting. Consequences are that correct program transformation in the untyped calculi are also correct in the typed calculi (provided that the transformation is type-preserving) and more importantly that typed calculi allow more correct program transformations than untyped calculi since the set of contexts is smaller and thus the contextual equivalence is less discriminating than in the untyped calculi.

Thus it is reasonable to also explore the semantics and the correctness of program transformations of typed program calculi. There are also some investigations in calculi with polymorphic types, letrec and seq [22] adapting parametricity to polymorphic calculi with

1st International Workshop on Rewriting Techniques for Program Transformations and Evaluation (WPTE'14).
Editors: Manfred Schmidt-Schauß, Masahiko Sakai, David Sabel, and Yuki Chiba; pp. 63–74

OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Type variables:	$a \in \mathcal{A}$		where $\mathcal{A}$ is the set of type variables
Term variables:	$x, x_i \in \mathcal{X}$		where $\mathcal{X}$ is the set of term variables
Types:	$\tau \in \mathit{Typ}$	$:= a \mid (\tau_1 \to \tau_2) \mid (K\ \tau_1 \dots \tau_{ar(K)})$	
Polymorphic types:	$\rho \in \mathit{PTyp}$	$:= \tau \mid \boldsymbol{\lambda} a.\rho$	
Expressions:	$e \in \mathit{Expr}_F$	$:= x : \rho \mid u \mid (e\ \tau) \mid (e_1\ e_2) \mid (c : \tau\ e_1\ \dots\ e_{ar(c)})$	
		$\mid (\texttt{seq}\ e_1\ e_2) \mid \texttt{letrec}\ x_1 : \rho_1 = e_1, \dots, x_n : \rho_n = e_n\ \texttt{in}\ e$	
		$\mid \texttt{case}_K\ e\ \texttt{of}\ (p_1\ \texttt{->}\ e_1) \dots (p_n\ \texttt{->}\ e_n)$	
		where there is a pattern p for every constructor in D_K	
Polymorphic expressions:	$u \in \mathit{PExpr}_F$	$:= x : \boldsymbol{\lambda} a.\rho \mid \lambda x : \tau.e \mid (u\ \tau) \mid \Lambda a.u$	
Case-patterns:	p	$:= (c : \tau\ x_1 : \tau_1 \dots x_{ar(c)} : \tau_{ar(c)})$	
		where x_i are different term variables	

Figure 1 Types and expressions of the language L_F.

seq, but not analyzing program transformations in depth. System F polymorphic calculi were first described in [3], are used in programming languages [10], and a variant of it is used in a Haskell compiler [4, 23].

In this paper we focus our investigations on a polymorphically typed calculus and thus we introduce a polymorphically typed lambda-calculus L_F with letrec, case, constructors and seq that models sharing on the expression level. The (predicative) polymorphism in the calculus is modelled in a system-F style by type abstractions. Predicative typing shows up in the formation rule for application, where the argument is not permitted to be polymorphic. As a second calculus, we present a call-by-name calculus L_P without letrec, together with a fully abstract translation $T : L_F \to L_P$. There are type-erasing translations into untyped variants of the calculi (see [20]).

$$\begin{array}{lcc} \text{typed:} & L_F & \xrightarrow{\quad T \quad} & L_P \\ & \downarrow{\scriptstyle \varepsilon} & & \downarrow{\scriptstyle \varepsilon_P} \\ \text{untyped:} & L_{LR} \text{ in [20]} & & L_{lcc} \text{ in [20]} \end{array}$$

The results in this paper are: The correctness of a large set of program transformations in L_F and L_P (see Corollary 4.5 and Proposition 5.3). By stand-alone proofs in the respective calculi we obtain a context lemma in L_F (Proposition 4.7); and a sound and complete applicative simulation $\preccurlyeq_P$ in L_P (Theorem 5.6), which implies a ciu-Theorem 5.9 as a replacement for the context-lemma in L_P. By analogy, and since the calculi are deterministic, we obtain that the fixpoint operators for $\preccurlyeq_P$ are continuous, and so $\preccurlyeq_P = \preccurlyeq_{P,\omega}$ where the latter is defined inductively (see Theorem 4.10). Using the fully abstract typed translation T, the results in L_P can be transferred to L_F, using the methods on Q-similarity in [20].

2 Syntax of the Polymorphic Typed Call-By-Need Lambda Calculus

We define the polymorphically typed language L_F which employs cyclic sharing using a letrec [2] and is like a core-language of Haskell [4], and uses ideas of system-F polymorphism. The syntax of types and expressions is shown in Fig. 1. There are two classes of types. Types $\tau \in \mathit{Typ}$ are like extended monomorphic types, where type variables are allowed, but are treated more or less as constants. Polymorphic types $\rho \in \mathit{PTyp}$ allow to explicitly quantify type variables by $\boldsymbol{\lambda}$. Expressions are built from variables (which always occur with their type), abstractions, applications, type abstractions and applications, `seq`-expressions to

$$\frac{e : \tau'}{(\lambda x : \tau.e) : \tau \to \tau'} \qquad \frac{e : \rho}{\Lambda a.e : \boldsymbol{\lambda} a.\rho} \qquad \frac{e : \boldsymbol{\lambda} a.\rho}{(e\ \tau) : \rho[\tau/a]} \qquad \frac{e : \tau_1 \to \tau_2 \quad e' : \tau_1}{(e\ e') : \tau_2}$$

$$\frac{e_1 : \rho_1 \quad \ldots \quad e_n : \rho_n \quad e : \rho}{(\texttt{letrec}\ x_1 : \rho_1 = e_1, \ldots, x_n : \rho_n = e_n\ \texttt{in}\ e) : \rho} \qquad \frac{e_1 : \rho \quad e_2 : \rho'}{(\texttt{seq}\ e_1\ e_2) : \rho'}$$

$$\frac{\begin{array}{l} e_1 : \tau_1, \ldots, e_n : \tau_n \quad \tau = \tau_1 \to \ldots \to \tau_n \to \tau_{n+1} \\ c : \boldsymbol{\lambda} a_1, \ldots, a_m.\tau'' \text{ is the general type of } c \\ \text{there are } \tau'_1, \ldots, \tau'_m : \tau''[\tau'_1/a_1, \ldots, \tau'_m/a_m] = \tau \end{array}}{(c : \tau\ e_1, \ldots e_n) : \tau_{n+1}} \qquad \frac{e : \tau \quad p_i : \tau \quad e_i : \rho}{(\texttt{case}_K\ e\ \texttt{of}\ (p_1\ \texttt{->}\ e_1) \ldots (p_n\ \texttt{->}\ e_n)) : \rho}$$

Figure 2 Typing Rules for L_F.

model strict evaluation, recursive `letrec`, constructor applications and `case`-expressions. We assume that there are type-constructors K given with their respective arity, denoted $ar(K)$, similar as Haskell-style data- and type constructors (see [9]). We assume that the constant type constructors `Bool`, `Nat` and the unary `List` are already defined. For every type-constructor K of arity $ar(K)$, there is a set $D_K \neq \emptyset$ of data constructors, such that $K_1 \neq K_2 \implies D_{K_1} \cap D_{K_2} = \emptyset$. Every (data) constructor c of K comes with a type $\boldsymbol{\lambda} a_1, \ldots, a_{ar(K)}.\tau_1 \to \ldots \to \tau_{ar(c)} \to K\ a_1 \ldots a_{ar(K)}$. We assume that the following is available: $D_{\texttt{Bool}} = \{\texttt{True}, \texttt{False}\}$, with $\texttt{True} : \texttt{Bool}$, $\texttt{False} : \texttt{Bool}$, $D_{\texttt{List}} = \{\texttt{Nil}, \texttt{Cons}\}$, with $\texttt{Nil} : \boldsymbol{\lambda} a.\texttt{List}\ a$, $\texttt{Cons} : \boldsymbol{\lambda} a.a \to \texttt{List}\ a \to \texttt{List}\ a$, and $D_{\texttt{Nat}} = \{0, \texttt{Succ}\}$, where $0 : \texttt{Nat}$ and $\texttt{Succ} : \texttt{Nat} \to \texttt{Nat}$.

The scoping is as expected: For expressions, λx, Λa, $(p \to \ldots)$ open a scope, and `letrec` opens a recursive scope. For types, $\boldsymbol{\lambda} a$ opens a scope. Types of the expressions are (generalized) monomorphic and the polymorphic aspect is the type computation via type abstractions and beta-reduction for type applications. For the reduction, the idea is that types could be omitted from reduction without any change in the operational reduction sequence (see Sect. 4.1). The body u of a type abstractions $\Lambda a.u$ are syntactically restricted (see Fig. 1), which implies that reduction cannot generate a `letrec`-expression as its body. We will explain the reason for this restriction in Remark 3.5 below. Note that the syntax permits a polymorphic non-termination expression (i.e. a "bot").

▶ **Example 2.1.** An example is the polymorphic combinator $\mathbf{K} := \Lambda a.\Lambda b.\lambda x : a.\lambda y : b.x$. It is polymorphic in the type variables a, b.

▶ **Remark 2.2.** It is known that there is a practically problematic danger of growth of type expressions during reduction, as reported in [23], but this could be defeated by using dags for types. We could model this by a `let` for types, which, however, would lead to notational overhead. So, for simplicity, we treat the types in a non-sharing way.

A generalized monomorphic type-system is used to to form correctly typed expressions where $\boldsymbol{\lambda}$ is also permitted in the syntax of types. The typing rules are in Fig. 2. Typing the polymorphic combinator $\mathbf{K} := \Lambda a.\Lambda b.\lambda x : a.\lambda y : b.x.$ results in $\boldsymbol{\lambda} a.\boldsymbol{\lambda} b.a \to (b \to a)$.

▶ **Definition 2.3** (Contexts)**.** An L_F-*context* $\mathbb{C} \in Ctxt_F$ is an L_F-expression that has a single occurrence of the *hole* $[\cdot : \rho]$ of (polymorphic) type ρ and is itself well-typed.

This represents contexts, where the hole maybe at polymorphic expressions.

$(e_1\ e_2)^{\mathsf{sub}\vee\mathsf{top}}$	$\to (e_1^{\mathsf{sub}}\ e_2)^{\mathsf{sub}} \qquad e_1 \neq \Lambda a.e'$
$(\texttt{letrec}\ Env\ \texttt{in}\ e)^{\mathsf{top}}$	$\to (\texttt{letrec}\ Env\ \texttt{in}\ e^{\mathsf{sub}})^{\mathsf{sub}}$
$(\texttt{letrec}\ x = e, Env\ \texttt{in}\ \mathbb{C}[x^{\mathsf{sub}}])$	$\to (\texttt{letrec}\ x = e^{\mathsf{sub}}, Env\ \texttt{in}\ \mathbb{C}[x^{\mathsf{sub}}])$
$(\texttt{letrec}\ x = e_1, y = \mathbb{C}[x^{\mathsf{sub}}], Env\ \texttt{in}\ e_2)$	$\to (\texttt{letrec}\ x = e_1^{\mathsf{sub}}, y = \mathbb{C}[x^{\mathsf{sub}}], Env\ \texttt{in}\ e_2)$
$(\texttt{seq}\ e_1\ e_2)^{\mathsf{sub}\vee\mathsf{top}}$	$\to (\texttt{seq}\ e_1^{\mathsf{sub}}\ e_2)^{\mathsf{sub}}$
$(\texttt{case}\ e\ \texttt{of}\ alts)^{\mathsf{sub}\vee\mathsf{top}}$	$\to (\texttt{case}\ e^{\mathsf{sub}}\ \texttt{of}\ alts)^{\mathsf{sub}}$
$((\Lambda a.u)\ \tau)^{\mathsf{sub}\vee\mathsf{top}}$	$\to ((\Lambda a.e^{\mathsf{sub}})\ \tau)^{\mathsf{sub}}$; then stop with success
$(\Lambda a.u)^{\mathsf{sub}\vee\mathsf{top}}$	$\to (\Lambda a.u^{\mathsf{sub}})^{\mathsf{sub}}$
$\mathsf{sub}\vee\mathsf{top}$ means label sub or top.	

Figure 3 Searching the normal-order redex using labels.

3 Small-Step Operational Semantics of L_F

A reduction step consists of: (1) finding a normal-order redex, then (2) applying a reduction rule. Instead of defining step (1) by a syntactic definition reduction contexts – which is notationally complex in L_F (see e.g. [21] for a similar language), we define the search by a labeling algorithm which uses two atomic labels sub, top, where top means the top-expression, and sub means "subterm" (in a `letrec`-expression). For an expression e the labeling algorithm starts with e^{top}, where e has no further inner labels top or sub. Then the rules of Fig. 3 are applied exhaustively. The role of top and sub is to prevent to label positions inside deep `letrec`-expressions. The labeling algorithm fails, if a loop is detected, which happens if a to-be-labeled position is already labeled sub, and otherwise, if no more rules are applicable or if the labeling algorithm has to stop, it succeeds. If we apply the labeling algorithm to contexts, then the contexts where the hole will be labeled with sub, or top are called the *reduction contexts*. We denote reduction contexts with $\mathbb{R}$. Note that for the ease of reading, we omit the types of variables and constructors in the notation.

▶ **Definition 3.1.** *Normal-order reduction* rules are defined in Fig. 4, where we assume that the labeling algorithm was used successfully before. Otherwise no normal-order reduction is applicable. In the presentation of the rules we only present the to-be-reduced subexpression. We also assume that the topmost redex according to the rule is the *normal-order redex*.

Note that the guiding principle in the cp-rules is to copy only values, i.e. polymorphic abstractions or cv-expressions. It is easy to verify that normal-order reduction is unique.

▶ **Definition 3.2.** A *cv-expression* is an expression of the form $(c\ x_1 \ldots x_n)$ where c is a constructor and x_i are variables. A *polymorphic abstraction* is an expression of the form $\Lambda a_1, \ldots, \Lambda a_n.u$, where $n \geq 0$ and u is an abstraction. Let $\mathbb{W} \in \mathit{WCtxt}$ denote contexts according to the grammar $\mathbb{W} \in \mathit{WCtxt} ::= [\cdot] \mid (\texttt{letrec}\ Env\ \texttt{in}\ [\cdot])$. A *weak head normal form* (WHNF) is a polymorphic abstraction, or of the form $W[w]$, where w is a constructor application or an abstraction.

▶ **Lemma 3.3.** *Reduction does not change the type of expressions.*

▶ **Example 3.4.** As an example we reduce an expression including the polymorphic combinator $\mathbf{K} := \Lambda a.\Lambda b.\lambda x : a.\lambda y : b.x$. Applying it to the type `Bool`, the constant `True` of type `Bool` and a type variable a' is as follows: $(\Lambda a.\Lambda b.\lambda x : a, \lambda y : b.x)$ `Bool` a' `True` results after three normal-order reductions in $(\texttt{letrec}\ x : \texttt{Bool} = \texttt{True}\ \texttt{in}\ \lambda y : a'.x)$.

(lbeta)	$((\lambda x:\tau.e_1)^{\mathsf{sub}}\ e_2) \to \texttt{letrec}\ x:\tau = e_1\ \texttt{in}\ e_2$
(Lbeta)	$((\Lambda a.u)^{\mathsf{sub}}\ \tau) \to u[\tau/a]$
(cp-in)	$\texttt{letrec}\ x = v^{\mathsf{sub}}, Env\ \texttt{in}\ \mathbb{C}[x^{\mathsf{sub}}] \to \texttt{letrec}\ x = v, Env\ \texttt{in}\ \mathbb{C}[v]$ where v is a polymorphic abstraction, or a cv-expression
(cp-e)	$\texttt{letrec}\ x = v^{\mathsf{sub}}, y = \mathbb{C}[x^{\mathsf{sub}}], Env\ \texttt{in}\ e \to \texttt{letrec}\ x = v, y = \mathbb{C}[v], Env\ \texttt{in}\ e$ where v is a polymorphic abstraction, or a cv-expression
(cpcx-in)	$\texttt{letrec}\ x = (c:\tau\ e_1 \dots e_n)^{\mathsf{sub}}, Env\ \texttt{in}\ \mathbb{C}[x^{\mathsf{sub}}] \to \texttt{letrec}\ x = (c:\tau\ x_1 \dots x_n), x_1:\tau_1 = e_1, \dots, x_n:\tau_n = e_n, Env\ \texttt{in}\ \mathbb{C}[(c\ x_1 \dots x_n)]$ where the types τ_i are computed as the type of e_i
(cpcx-e)	$\texttt{letrec}\ x = (c:\tau\ e_1 \dots e_n)^{\mathsf{sub}}, y = \mathbb{C}[x^{\mathsf{sub}}], Env\ \texttt{in}\ e \to \texttt{letrec}\ x = (c:\tau\ e_1 \dots e_n), x_1:\tau_1 = e_1, \dots, x_n:\tau_n = e_n, y = \mathbb{C}[(c\ x_1 \dots x_n)], Env\ \texttt{in}\ e$ where the types τ_i are computed as the type of e_i
(case)	$(\texttt{case}\ (c\ e_1 \dots e_n)^{\mathsf{sub}}\ \texttt{of}\ \dots((c\ y_1:\tau_1 \dots y_n:\tau_n)\ \texttt{->}\ e)\dots) \to \texttt{letrec}\ y_1:\tau_1 = e_1, \dots, y_n:\tau_n = e_n\ \texttt{in}\ e$
(case)	$(\texttt{case}\ c^{\mathsf{sub}}\ \texttt{of}\ \dots(c\ \texttt{->}\ e)\dots) \to e$
(seq)	$(\texttt{seq}\ v^{\mathsf{sub}}\ e) \to e$ if v is a constructor application or a polymorphic abstraction
(llet-e)	$\texttt{letrec}\ Env_1, x = (\texttt{letrec}\ Env_2\ \texttt{in}\ e_1)^{\mathsf{sub}}\ \texttt{in}\ e_2 \to \texttt{letrec}\ Env_1, Env_2, x = e_1\ \texttt{in}\ e_2$
(llet-in)	$\texttt{letrec}\ Env_1\ \texttt{in}\ (\texttt{letrec}\ Env_2\ \texttt{in}\ e)^{\mathsf{sub}} \to \texttt{letrec}\ Env_1, Env_2\ \texttt{in}\ e$
(lapp)	$((\texttt{letrec}\ Env\ \texttt{in}\ e_1)^{\mathsf{sub}}\ e_2) \to \texttt{letrec}\ Env\ \texttt{in}\ (e_1\ e_2)$
(lseq)	$(\texttt{seq}\ (\texttt{letrec}\ Env\ \texttt{in}\ e_1)^{\mathsf{sub}}\ e_2) \to \texttt{letrec}\ Env\ \texttt{in}\ (\texttt{seq}\ e_1\ e_2)$
(lcase)	$(\texttt{case}\ (\texttt{letrec}\ Env\ \texttt{in}\ e)^{\mathsf{sub}}\ \texttt{of}\ alts) \to \texttt{letrec}\ Env\ \texttt{in}\ (\texttt{case}\ e\ \texttt{of}\ alts)$

Figure 4 Normal-order rules.

▶ **Remark 3.5.** On typed and untyped sharing: There is a constellation that has to be excluded (by syntax): expressions of the form $e = \Lambda a_1 \dots a_n.(\texttt{letrec}\ Env\ \texttt{in}\ t)$. The technical problem is that the (cp)-rules want to copy these expressions. However, in the untyped case, this is not possible, but instead a let-shifting can be done. In the typed case the scoping of the types prevents this let-shifting, and so the expression is stuck: it cannot be further reduced. Analyzing the usage at runtime of the expressions in the environment Env, it turns out that it does not make sense to share them among differently typed copies, since it is not type-safe. So the intention can only be to copy Env together with the abstraction. But then the elements in Env are not really useful, since they must have all types. Due to this conflict with the untyped reduction, the body e in $\Lambda a.e$ is syntactically restricted to expressions $u \in PExpr_F$. I.e., `letrec`-expressions and also expressions which may reduce to a `letrec`-expression (e.g. an application) are forbidden for e. If e is a variable, then it must have a type of the form $\boldsymbol{\lambda}.\rho$ which again ensures that the variable cannot be replaced by arbitrary expressions. For the same reasons, we also forbid constructors at the position for e in $\Lambda a.e$. Thus allowed expressions for e are type applications, abstractions and polymorphic variables in Λ-bodies, which also enforces potential reductions in the body. We permit only $\Lambda a_1 \dots a_n.\lambda x.e$ as proper polymorphic WHNFs.

3.1 Contextual Equivalence

In this section we define contextual equivalence for typed expressions. We introduce convergence as the observable behavior of expressions. Expressions are contextually equivalent if their convergence behavior is indistinguishable in all program contexts.

▶ **Definition 3.6.** Let $e \in L_F$. A normal order reduction sequence of e is called a *(normal-order) evaluation* if the last term is a WHNF. We write $e{\downarrow}e'$ (e *converges*) iff there is an evaluation starting from e ending in WHNF e' (we omit e', if it is of no interest). Otherwise, if there is no evaluation of e, we write $e{\Uparrow}$.

▶ **Definition 3.7** (Contextual Preorder and Equivalence)**.** *Contextual preorder* $\leq_F$ and *contextual equivalence* $\sim_F$ are defined for equally typed expressions. For e_1, e_2 of type ρ:

$$\begin{array}{lll} e_1 \leq_F e_2 & \text{iff} & \forall \mathbb{C}[\cdot : \rho] : \tau : \text{ If } \mathbb{C}[e_1] \text{ and } \mathbb{C}[e_2] \text{ are closed, then } (\mathbb{C}[e_1]{\downarrow} \implies \mathbb{C}[e_2]{\downarrow}) \\ e_1 \sim_F e_2 & \text{iff} & e_1 \leq_F e_2 \quad \wedge \quad e_2 \leq_F e_1 \end{array}$$

Note that we are only interested in top-expressions of closed type. It is standard to show that $\leq_F$ is a precongruence, i.e. a compatible partial order, and that $\sim_F$ is a congruence, i.e. a compatible equivalence relation. Also, a progress lemma holds for closed expressions e: If no reduction is possible, then e is a WHNF, or the search for a redex fails.

4 Correctness of Program Transformations and Translations

A *typed program transformation* P is a binary relation on L_F-expressions, such that $(e_1, e_2) \in P$ is only valid for well-formed e_1, e_2 of equal type. The restriction of P to a type ρ is denoted with P_ρ. A program transformation P is called *correct* iff for all ρ and all $(e_1, e_2) \in P_\rho$, the contextual equivalence relation $e_1 \sim_F e_2$ holds. Analogously, for untyped programs, a program transformation P is a binary relation on untyped expressions and it is correct if $P \subseteq \sim$. Disproving the correctness of a (typed or untyped) program transformation is often easy, since a counter example consisting of a program context which distinguishes two related expressions by their convergence behavior is sufficient.

▶ **Definition 4.1.** Let L_1, L_2 be two (typed or untyped) calculi with a notion of expressions, contexts, may-convergence $\downarrow_i$, and contextual preorder $\leq_i$. A translation ψ from calculus L_1 in L_2 (with equal set of types) mapping expressions to expressions and contexts into contexts where types are retained and $\psi([\cdot]) = [\cdot]$ is **1. convergence equivalent** if $e{\downarrow_1} \iff \psi(e){\downarrow_2}$; **2. compositional** if $\psi(\mathbb{C}[e]) = \psi(\mathbb{C})[\psi(e)]$; **3. adequate** if $\psi(e_1) \leq_2 \psi(e_2) \implies e_1 \leq_1 e_2$; and **4. fully abstract** if $\psi(e_1) \leq_2 \psi(e_2) \iff e_1 \leq_1 e_2$.

4.1 Importing Results from Untyped Calculi

The untyped language L_{LR} has the same syntax as L_F except that variables have no type, and that type abstractions $\Lambda a.e$ and type applications $(e\ \tau)$ are not permitted. The normal order reduction for L_{LR} is defined as for L_F where the rule (Lbeta) is not used. The semantics of L_{LR} was investigated e.g. in [20, 21].

▶ **Definition 4.2.** The *translation* ε translates L_F-expressions into untyped expressions L_{LR} where we assume that the data types and the constructors are the same. It is defined as $\varepsilon(\Lambda a.e) := \varepsilon(e)$, $\varepsilon(e\ \tau) := \varepsilon(e)$, and on the other constructs ε acts homomorphically, removing type labels and types.

Since the untyped reduction is the same as the typed reduction if the types and (Lbeta)-reductions are ignored, and since the untyped WHNFs are exactly the typed WHNFs with the types removed, ε is convergence equivalent. Since it is also independent of the surrounding context, it is also compositional:

▶ **Lemma 4.3.** *The translation ε is convergence equivalent and compositional.*

(gc)	$(\texttt{letrec}\ x_1 = e_1, \ldots, x_n = e_n\ \texttt{in}\ e) \to e$ if no x_i occurs free in e
(gc)	$(\texttt{letrec}\ x_1 = e_1, \ldots, x_n = e_n, y_1 = e'_1, \ldots, y_m = e'_m\ \texttt{in}\ e)$ $\to (\texttt{letrec}\ y_1 = e'_1, \ldots, y_m = e'_m\ \texttt{in}\ e)$ if no x_i occurs free in e nor in any e'_j
(gcp)	$(\texttt{letrec}\ x = e, \mathit{Env}\ \texttt{in}\ \mathbb{C}[x]) \to (\texttt{letrec}\ x = e, \mathit{Env}\ \texttt{in}\ \mathbb{C}[e])$
(gcp)	$(\texttt{letrec}\ x = e_1, y = \mathbb{C}[x], \mathit{Env}\ \texttt{in}\ e_2) \to (\texttt{letrec}\ x = e_1, y = \mathbb{C}[e_1], \mathit{Env}\ \texttt{in}\ e_2)$
(gcp)	$(\texttt{letrec}\ x = \mathbb{C}[x], \mathit{Env}\ \texttt{in}\ e) \to (\texttt{letrec}\ x = \mathbb{C}[\mathbb{C}[x]], \mathit{Env}\ \texttt{in}\ e)$

Figure 5 Further program transformations.

This implies that it is also adequate (see for example [17]).

▶ **Corollary 4.4.** *The translation ε is adequate, which means that equivalences from L_{LR} also hold in L_F.*

Proving correctness of program transformations is in general a hard task, since all contexts need to be taken into account. In e.g. [8, 21, 12] methods to prove correctness of program transformations for untyped letrec calculi were developed. As a first step we will use the result of [21] to lift untyped program equivalences into the typed calculus. In the calculus introduced in [21] the normal order reduction is slightly different, but it is easy to show that these differences do not change the convergence behavior of untyped expressions (a proof of this coincidence for an extended calculus can be in [15], see also [20]). This implies that all reduction rules of Fig. 4 and the optimizations *garbage collection* (gc) and *general copying* (gcp) (see Fig.5) are correct program transformations for L_F (for (gcp) see [16]).

▶ **Corollary 4.5.** *The reductions rules from Figs. 4 and 5 are correct program transformations in L_F, and can be used in any context.*

4.2 Context Lemma

For the context lemma we first define the $\leq$-relation for reduction contexts. A context lemma for a similar polymorphic calculus with a (more complex) type labeling but without explicit type abstractions and applications is in [15].

▶ **Definition 4.6.** For a polymorphic type ρ and e_1, e_2 of type ρ, let $e_1 \leq_{F,R} e_2$ hold if for all reduction contexts $\mathbb{R}[\cdot : \rho]$ such that $\mathbb{R}[e_1], \mathbb{R}[e_2]$ are closed: $\mathbb{R}[e_1]{\downarrow} \implies \mathbb{R}[e_2]{\downarrow}$.

▶ **Proposition 4.7** (Context Lemma for L_F)**.** *Let ρ be a polymorphic type and e_1, e_2 be of type ρ. Then $e_1 \leq_{F,R} e_2 \iff e_1 \leq_F e_2$.*

The proof is standard, e.g. it follows the technique explained in [18]. However, the proof technique relies on the proper use of sharing. For instance, the proof technique breaks down in a call-by-name calculus (like L_P in Sect. 5).

The context lemma 4.7 immediately implies:

▶ **Corollary 4.8.** *If e_1, e_2 are closed expressions of equal type with $e_1{\Uparrow}, e_2{\Uparrow}$, then $e_1 \sim_F e_2$.*

4.3 Inductive Similarity For L_F

As a further proof tool for showing contextual equivalences, we define an improved similarity definition in L_F, which is often superior to the context lemma.

▶ **Definition 4.9.** We define $\preccurlyeq_\omega := \bigcap_{n\geq 0} \preccurlyeq_n$ where for $n \geq 0$, $\preccurlyeq_n$ is defined on closed L_F-expressions e_1, e_2 of the same type as follows:

1. $e_1 \preccurlyeq_0 e_2$ is always true.
2. $e_1 \preccurlyeq_n e_2$ for $n > 0$ holds if the following conditions hold:
   a. if $e_1{\downarrow}\Lambda a.e_1'$, then $e_2{\downarrow}\Lambda a.e_2'$, and for all τ: $e_1'[\tau/a] \preccurlyeq_{n-1} e_2'[\tau/a]$.
   b. if $e_1{\downarrow}\mathbb{W}[\lambda x:\tau.e_1']$, then
   $e_2{\downarrow}\mathbb{W}'[\lambda x:\tau.e_2']$ and for all closed $e:\tau$: $\mathbb{W}[\lambda x.e_1']\ e \preccurlyeq_{n-1} \mathbb{W}'[\lambda x.e_2']\ e$.
   c. if $e_1{\downarrow}\mathbb{W}[c\ e_1' \ldots e_m']$, then $e_2{\downarrow}\mathbb{W}'[c\ e_1'' \ldots e_m'']$ and for all i: $\mathbb{W}[e_i'] \preccurlyeq_{n-1} \mathbb{W}'[e_i'']$.

The proof of soundness (and completeness) of inductive similarity w.r.t. contextual preorder can be constructed similar to an analogous proof in the untyped case (see [20]). We sketch the proof: The language L_P is a polymorphically typed *call-by-name* lambda-calculus with fixpoint combinators, but no `letrec` (see Sect. 5). In L_P conincidence of contextual preorder and inductive similarity can be shown by proving soundness and completeness of an applicative similarity using Howe's method (Theorem 5.6). Using some further arguments which rely on the continuity of the fixpoint combinators show the coincidence of applicative and inductive similarity (Theorem 5.8). Then a translation $T : L_F \to L_P$ is defined (below in Sect. 5.2) which replaces `letrec`-expressions by fixpoint combinators. Since the translation T is fully abstract and surjective on the equivalence classes of contextual equivalence, Theorem 5.8 can be lifted into L_F which shows the following theorem:

▶ **Theorem 4.10.** $\preccurlyeq_\omega^o = \leq_F$

As a corollary we show that ε is not fully abstract.

▶ **Proposition 4.11.** *The translation ε is not fully abstract.*

Proof. In L_F the expressions $e_1 = \lambda x : \texttt{Bool}.\texttt{case}_{\texttt{Bool}}\ x\ \texttt{of}\ (\texttt{True}\ \texttt{->}\ x)\ (\texttt{False}\ \texttt{->}\ x)$ and $e_2 = \lambda x : \texttt{Bool}.x$ are contextually equivalent. This follows by Theorem 4.10 and since the possible arguments can be classified as equivalent to $\bot$ (a nonterminating expression of type `Bool`), `True` or `False`, and the result is equivalent or equal in all cases. However, $\varepsilon(e_1)$ and $\varepsilon(e_2)$ are different in L, which can be seen by applying them to $\lambda x.x$. ◀

As an example, we show the equivalence of other polymorphic expressions.

▶ **Proposition 4.12.** *In the language L_F the two expressions $e_1 = \Lambda a.\lambda x : (\texttt{List}\ a).x$ and $e_2 = \Lambda a.\lambda x : (\texttt{List}\ a).\texttt{case}\ x\ of\ (\texttt{Nil}\ \texttt{->}\texttt{Nil})\ (\texttt{Cons}\ y_1\ y_2\ \texttt{->}\texttt{Cons}\ y_1\ y_2)$ of polymorphic type $\boldsymbol{\lambda} a.\texttt{List}\ a \to \texttt{List}\ a$ are contextually equivalent.*

Proof. We use inductive similarity. Application to τ yields two monomorphic abstractions. Further application can only be to arguments without WNHF, or with WHNF $\mathbb{W}[\texttt{Nil}]$ or $\mathbb{W}[\texttt{Cons}\ e_1'\ e_2']$. In all cases, the results are obviously contextually equivalent and thus applicative similar. ◀

5 A Call-by-Name Polymorphic Lambda Calculus

We present a call-by-name polymorphic lambda calculus L_P as a second calculus, with built-in multi-fixpoint constructions Ψ for representing mutual recursive functions.

We argue that there is a fully abstract translation T from L_F into the calculus L_P. To demonstrate the power, we show that there is a polymorphic applicative simulation in L_P that is useful for recognizing equivalences. The calculus L_P is related to the lazy lambda calculus [1], however L_P is more expressive and typed.

Polymorphic expressions:	$u \in PExpr_P$	$::= x : \boldsymbol{\lambda} a.\rho \mid \lambda x:\tau.e \mid (u\ \tau) \mid \Lambda a.u$
Expressions:	$e, e_i \in Expr_P$	$::= x : \rho \mid u \mid (e\ \tau) \mid (e_1\ e_2) \mid (c : \tau\ e_1 \ldots e_{ar(c)})$ $\mid (\texttt{case}_K\ e\ \texttt{of}\ alts) \mid (\texttt{seq}\ e_1\ e_2) \mid \Psi_{i,n}\overline{x:\rho}.\overline{e}$
Reduction contexts:	$\mathbb{R}_P \in \mathcal{R}_P$	$::= [\cdot] \mid (\mathbb{R}_P\ e) \mid \texttt{case}_K\ \mathbb{R}_P\ \texttt{of}\ alts \mid \texttt{seq}\ \mathbb{R}_P\ e$

Normal order reduction:

(rnbeta) $\mathbb{R}_P[((\lambda x.e_1)\ e_2)] \xrightarrow{P} \mathbb{R}_P[e_1[e_2/x]]$

(rncase) $\mathbb{R}_P[\texttt{case}_K\ (c\ e_1 \ldots e_{ar(c)})\ \texttt{of}\ \ldots((c\ x_1 \ldots x_{ar(c)})\ \texttt{->}\ e)\ldots)]$
$\xrightarrow{P} e[e_1/x_1, \ldots, e_{ar(c)}/x_{ar(c)}]$

(rnseq) $\mathbb{R}_P[\texttt{seq}\ v\ e] \xrightarrow{P} \mathbb{R}_P[e]$, if v is an L_P- WHNF.

(rntype) $\mathbb{R}_P[(\Lambda a.e)\ \tau] \xrightarrow{P} \mathbb{R}_P[e[\tau/a]]$

(rnfix) $\mathbb{R}_P[\Psi_{i,n}\overline{x}.\overline{e}] \xrightarrow{P} \mathbb{R}_P[(e_i[\Psi_{1,n}\overline{x}.\overline{e}/x_1, \ldots, \Psi_{n,n}\overline{x}.\overline{e}/x_n])]$

Figure 6 Syntax and normal order reduction $\xrightarrow{P}$ of L_P.

5.1 The Calculus L_P

▶ **Definition 5.1.** The calculus L_P is defined as follows. The set $Expr_P$ of L_P-expressions is that of L_F, where `letrec` is removed, see Fig.6. $(\Psi_{i,n}\overline{x:\rho}.\overline{e})$ is a family of multi-fixpoint-operators, where $1 \leq i \leq n$ and where $\overline{x}$ means $x_1, \ldots, x_n$, similarly for e.

The typing rules are according to Fig. 2 with the additional rule for the Ψ-operator:

$$\frac{\text{for } i = 1, \ldots, n\text{: } e_i : \rho_i, x_i : \rho_i}{(\Psi_{i,n}\overline{x:\rho}.\overline{e}) : \rho_i}$$

WHNFs in L_P are (polymorphic) abstractions $\Lambda a_1. \ldots .\Lambda a_n.\lambda x : \rho.e$ and constructor applications. In Fig. 6 the reduction rules and the normal order reduction $\xrightarrow{P}$ for L_P using reduction contexts $\mathbb{R}_P$ are given. The contextual preorder $\leq_P$ and contextual equality $\sim_P$ are defined as above for the calculus L_F, where convergence to L_P-WHNFs and where holes in contexts are permitted to have polymorphic type ρ, but the context itself must have plain type.

Note that our syntax permits a "polymorphic bot": $\Psi_{1,1}\ x{:}(\boldsymbol{\lambda} a.a).x$. An example is polymorphic `length` of lists (type-labels in the notation are partially omitted):

$$\begin{aligned}\texttt{length} \ := \ & (\Psi_{1,1} \mathit{len}{:}\texttt{List}\ a \to \texttt{Nat}.\Lambda a.\lambda xs{:}\texttt{List}\ a. \\ & \quad \texttt{case}\ xs\ \texttt{of}\ (\texttt{Nil}\ \texttt{->}\ 0)\ (\texttt{Cons}\ y\ ys\ \texttt{->}\ \texttt{Succ}\ (\mathit{len}\ a\ ys)))\end{aligned}$$

Our formulation is a bit more general than that in [10] for system F and ML, which corresponds to Milner type checking, whereas our formulation permits differently typed occurrences of a recursive polymorphic functions in its defining body, and so corresponds to iterative (polymorphic) type checking.

5.2 On the Translation $T : L_F \to L_P$

In order to define the typed translation $T : L_F \to L_P$, we adapt the combined translation from the untyped variant in [20].

▶ **Definition 5.2.** The translation $T : L_F \to L_P$ is defined as:
$T(\texttt{letrec}\ x_1 = e_1; \ldots, x_n = e_n\ \texttt{in}\ e') := T(e')[(\Psi_{1,n}\overline{x}.\overline{f})/x_1, \ldots, (\Psi_{n,n}\overline{x}.\overline{f})/x_n]$ and where $f_i = \lambda x_1, \ldots x_n.T(e_i)$ for $i = 1, \ldots, n$, and it is homomorphically on other constructs.

▶ **Proposition 5.3.** *The translation $T : L_F \to L_P$ is fully abstract and surjective on contextual-equivalence classes.*

Proof. This can be derived from the untyped version in [20]. ◀

For inheriting correct translations we again use a translation ε_P into an untyped call-by-name calculus L_{lcc} in [20] by forgetting the types (and allowing also untyped expressions), and with $\varepsilon_P(\Psi_{1,1}x.s) = \mathbf{Y}\ (\varepsilon_P(\lambda x.s))$, where $\mathbf{Y}$ is the untyped fixpoint combinator in L_{lcc}. Similarly, we can translate $\Psi_{i,n}(\ldots)$ using the multi fixpoint combinators in [20].

▶ **Proposition 5.4.** *The translation $\varepsilon_P : L_P \to L_{lcc}$ is convergence equivalent and compositional, hence adequate, but it is not fully abstract. The reduction rules of L_P are correct.*

Proof. Only the case of a type abstraction requires an extra argument. It is sufficient that the contexts used for testing have the same type of the hole as the expressions. Then adequacy of the translation ε_P can be used. ◀

5.3 Applicative Simulation in L_P

In the following we use binary relations η on closed expressions (of the same type). We need closing substitutions σ which are defined as mapping free variables (of plain type) to closed expressions of the same type, and all type variables to plain types. This extension to type variables is the key to apply applicative simulation also to polymorphic functions. The open extension η^o of η is the relation on open expressions, where $e_1\ \eta^o\ e_2$ is valid iff for all substitutions σ where $\sigma(e_1), \sigma(e_2)$ are closed, the relation $\sigma(e_1)\ \eta\ \sigma(e_2)$ holds. We will also use the restriction of a binary relation η to closed expressions which is denoted as η^c.

▶ **Definition 5.5** (Applicative Similarity in L_P)**.** Let η be a binary relation on closed L_P-expressions, where only expression of equal syntactic type can be related. Let F_P be the operator on relations on closed L_P-expressions s.t. $e_1\ F_P(\eta)\ e_2$ holds iff

- $e_1{\downarrow_P}\lambda x.e_1' \implies \left(e_2{\downarrow_P}\lambda x.e_2' \text{ and } e_1'\ \eta^o\ e_2' \right)$
- $e_1{\downarrow_P}(c\ e_1' \ldots e_n') \implies \left(e_2{\downarrow_P}(c\ e_1'' \ldots e_n'') \text{ and the relation } e_i'\ \eta\ e_i'' \text{ holds for all } i \right)$
- $e_1{\downarrow_P}\Lambda a.e_1' \implies \left(e_2{\downarrow_P}\Lambda a.e_2' \text{ and } e_1'\ \eta^o\ e_2' \right)$

Applicative similarity $\preccurlyeq_P$ is defined as the greatest fixpoint of the operator F_P. *Mutual similarity* $\simeq_P$ is defined as $e_1 \simeq_P e_2$ iff $e_1 \preccurlyeq_P e_2 \wedge e_2 \preccurlyeq_P e_1$.

Note that the operator F_P is monotone, hence the greatest fixpoint $\preccurlyeq_P$ exists.

Howe's method [5, 6] to show that $\preccurlyeq_P$ is a pre-congruence and equal to $\leq_P^c$ can be applied without unexpected changes, see also [11] and [20, Sect. 4.2] where the only extra feature is the typing. For completeness, we show $\leq_P^c \subseteq \preccurlyeq_P$ by proving that contextual equivalence in L_P satisfies the fixpoint conditions of F_P and then we use coinduction. By the properties of $\preccurlyeq_P$ this implies that $\leq_P \subseteq \preccurlyeq_P$ also holds for open expressions.

▶ **Theorem 5.6.** *$\leq_P^c = \preccurlyeq_P$, and $\leq_P = \preccurlyeq_P^o$.*

Proof. Only the completeness part is missing. We have to analyze the three conditions of Definition 5.5, where we use Proposition 5.4 several times.

1. If $e_1 \leq_P e_2$, and $e_1{\downarrow_P}\lambda x : \tau.e_1'$, then clearly $e_2{\downarrow_P}\lambda x : \tau.e_2'$. Since beta-reduction is correct, also for all closed expressions $e : \tau\ e_1\ e \leq_P e_2\ e$, since $\leq_P$ is a precongruence, and since reduction sequences are correct w.r.t. $\sim_P$; thus $e_1'[e/x] \leq_P e_2'[e/x]$.

2. If $e_1 \leq_P e_2$ and $e_1{\downarrow_P}(c\ e'_1 \ldots e'_n)$, then $e_2{\downarrow_P}v_2$. Since $\leq_P$ is a precongruence, reduction is correct, and using simple contexts like `case` $[\cdot]$ `of` $((c\ x_1 \ldots x_n)$ `->True`$) \ldots (p'$ `->`$\bot)$ and `case` $[\cdot]$ `of` $((c\ x_1 \ldots x_n)$ `->`$x_i) \ldots (p'$ `->`$\bot)$, we see that v_2 is of the form $(c\ e''_1 \ldots e''_n)$, where $e'_i \leq_P e''_i$ for all i.
3. The last case is the type abstraction and type substitution. The same arguments as above can be used, by plugging the expressions e_1, e_2 into a context $([\cdot.]\ \tau)$. ◀

5.4 Inductive Similarity For L_P

We define the inductive version of applicative similarity in L_P:

▶ **Definition 5.7.** We define $\preccurlyeq_{P,\omega} := \bigcap_{n\geq 0} \preccurlyeq_{P,n}$ where for $n \geq 0$, $\preccurlyeq_{P,n}$ is defined on closed L_P-expressions e_1, e_2 of the same type as follows:

1. $e_1 \preccurlyeq_{P,0} e_2$ is always true.
2. $e_1 \preccurlyeq_{P,n} e_2$ for $n > 0$ holds if the following conditions hold:
   a. If $e_1{\downarrow}\Lambda a.e'_1$, then $e_2{\downarrow}\Lambda a.e'_2$, and for all τ: $e'_1[\tau/a] \preccurlyeq_{P,n-1} e'_2[\tau/a]$.
   b. if $e_1{\downarrow}\lambda x:\tau.e'_1$ then $e_2{\downarrow}\lambda x:\tau.e'_2$ and for all closed $e:\tau$: $e'_1[e/x] \preccurlyeq_{P,n-1} e'_2[e/x]$.
   c. if $e_1{\downarrow}(c\ e'_1 \ldots e'_m)$ then $e_2{\downarrow}(c\ e''_1 \ldots e''_m)$ and for all i: $e'_i \preccurlyeq_{P,n-1} e''_i$.

   Proving continuity of the fixpoint operator of $\preccurlyeq_P$ as in (see [20]), we obtain:

▶ **Theorem 5.8.** $\preccurlyeq^o_{P,\omega} = \leq_P$

A corollary is a ciu-Theorem: Let $e_1 \leq_{ciu} e_2$ for two L_P-expressions e_1, e_2 of equal type iff for all closed L_P-reduction contexts $\mathbb{R}_P$, and all (well-typed) substitutions σ where $\sigma(e_1)$ and $\sigma(e_2)$ are closed: $\mathbb{R}_P[\sigma(e_1)]{\downarrow} \implies \mathbb{R}_P[\sigma(e_2)]{\downarrow}$.

▶ **Theorem 5.9.** $\leq_{ciu} = \leq_P$

Proof. We apply the knowledge about applicative simulation $\preccurlyeq$: If $e_1 \preccurlyeq^o e_2$, then for all σ where $\sigma(e_1), \sigma(e_2)$ are closed: $\sigma(e_1) \preccurlyeq \sigma(e_2)$. Since we already know that $\preccurlyeq$ is a pre-congruence, we also obtain $\mathbb{R}_P[\sigma(e_1)] \preccurlyeq \mathbb{R}_P[\sigma(e_2)]$, and so $\mathbb{R}_P[\sigma(e_1)]{\downarrow} \implies \mathbb{R}_P[\sigma(e_2)]{\downarrow}$.

We show that the ciu-relation implies $\preccurlyeq^o$: For closed e_1, e_2 it holds: Since the calculus L_P is deterministic, it is sufficient to restrict the test to the contexts $[\cdot]\ e'_1 \ldots\ e'_m$ for all $m \geq 0$ and all closed e'_i (see Theorem 5.8). So, if the ciu-condition for closed expressions holds, we obtain $e_1 \leq_P e_2$ and so $e_1 \preccurlyeq e_2$. The ciu-relation for non-closed e_1, e_2 is equivalent to the open extension of $\preccurlyeq$, i.e., it implies $e_1 \preccurlyeq^o e_2$, and thus we have the equality: $\preccurlyeq^o = \leq_{ciu}$. ◀

6 Conclusion

Using a system-F-like extension of untyped extended lambda-calculi with case, constructors, and seq, and call-by-need and call-byname variants, we present several tools for recognizing correct transformations This could potentially be used in lazy functional programming languages like Haskell.

Further research may be to investigate a polymorphic variant of (the non-deterministic language) Concurrent Haskell with futures (CHF) [14, 13]. A non-deterministic extension of L_F and L_P with `amb` appears unrealistic, since there are counterexamples for combinations of `letrec` [19] and since call-by-name and call-need nondeterminism are very different.

Acknowledgment. We thank the anonymous reviewers for their valuable comments.

References

1 S. Abramsky. The lazy lambda calculus. In D. A. Turner, editor, *Research Topics in Functional Programming*, pages 65–116. Addison-Wesley, 1990.

2 Z. M. Ariola and J. W. Klop. Lambda calculus with explicit recursion. *Inform. and Comput.*, 139(2):154–233, 1997.

3 J.-Y. Girard, P. Taylor, and Y. Lafont. *Proofs and Types*. CUP, 1994.

4 Haskell-community. The Haskell Programming Language, 2014. `http://www.haskell.org`.

5 D. Howe. Equality in lazy computation systems. In *LICS'89*, pages 198–203, 1989.

6 D. Howe. Proving congruence of bisimulation in functional programming languages. *Inform. and Comput.*, 124(2):103–112, 1996.

7 J. Launchbury. A natural semantics for lazy evaluation. In *POPL'93*, pages 144–154. ACM, 1993.

8 A. K. D. Moran, D. Sands, and M. Carlsson. Erratic fudgets: A semantic theory for an embedded coordination language. *Sci. Comput. Program.*, 46(1-2):99–135, 2003.

9 S. Peyton Jones. *Haskell 98 language and libraries: the Revised Report*. CUP, 2003.

10 B. C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

11 A. M. Pitts. Howe's method for higher-order languages. In *Advanced Topics in Bisimulation and Coinduction*, volume 52 of *Cambridge Tracts in Theoretical Computer Science*, chapter 5, pages 197–232. CUP, November 2011. (chapter 5).

12 D. Sabel and M. Schmidt-Schauß. A call-by-need lambda-calculus with locally bottom-avoiding choice: Context lemma and correctness of transformations. *Math. Structures Comput. Sci.*, 18(03):501–553, 2008.

13 D. Sabel and M. Schmidt-Schauß. A contextual semantics for Concurrent Haskell with futures. In *PPDP'11*, pages 101–112, New York, NY, USA, July 2011. ACM.

14 D. Sabel and M. Schmidt-Schauß. Conservative concurrency in Haskell. In *LICS'12*, pages 561–570. IEEE, 2012.

15 D. Sabel, M. Schmidt-Schauß, and F. Harwath. Reasoning about contextual equivalence: From untyped to polymorphically typed calculi. In *INFORMATIK 2009 (ATPS'09)*, volume 154 of *LNI*, pages 369; 2931–45, 2009.

16 M. Schmidt-Schauß. Correctness of copy in calculi with letrec. In *RTA'08*, volume 4533 of *LNCS*, pages 329–343. Springer, 2007.

17 M. Schmidt-Schauß, E. Machkasova, and D. Sabel. Extending Abramsky's lazy lambda calculus: (non)-conservativity of embeddings. In *RTA'13*, volume 21 of *LIPIcs*, pages 239–254, Dagstuhl, Germany, 2013. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

18 M. Schmidt-Schauß and D. Sabel. On generic context lemmas for higher-order calculi with sharing. *Theoret. Comput. Sci.*, 411(11-13):1521 – 1541, 2010.

19 M. Schmidt-Schauß, D. Sabel, and E. Machkasova. Counterexamples to applicative simulation and extensionality in non-deterministic call-by-need lambda-calculi with letrec. *Inf. Process. Lett.*, 111(14):711–716, 2011.

20 M. Schmidt-Schauß, D. Sabel, and E. Machkasova. Simulation in the call-by-need lambda-calculus with letrec, case, constructors, and seq. Frank report 49, Goethe-Universität Frankfurt, 2012.

21 M. Schmidt-Schauß, M. Schütz, and D. Sabel. Safety of Nöcker's strictness analysis. *J. Funct. Programming*, 18(04):503–551, 2008.

22 J. Voigtländer and P. Johann. Selective strictness and parametricity in structural operational semantics, inequationally. *Theor. Comput. Sci*, 388(1–3):290–318, 2007.

23 D. Vytiniotis and S. Peyton Jones. Evidence Normalization in System FC (Invited Talk). In *RTA'13*, volume 21 of *LIPIcs*, pages 20–38, Dagstuhl, Germany, 2013. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.