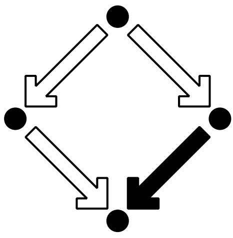
WRS 2008

Reduction Strategies in Rewriting and Programming
8th International Workshop

Proceedings

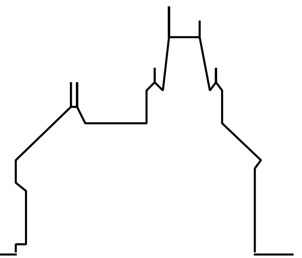
Editor: Aart Middeldorp

July 14, 2008
Castle of Hagenberg, Austria



RISC-Linz

Research Institute for Symbolic Computation
Johannes Kepler University
A-4040 Linz, Austria, Europe



WRS 2008

Reduction Strategies in Rewriting and Programming

8th International Workshop

Aart MIDDELDORP (editor)

Castle of Hagenberg, Austria
July 14, 2008

RISC-Linz Report Series No. 08-09

Editors: RISC-Linz Faculty
B. Buchberger, R. Hemmecke, T. Jebelean, M. Kauers, T. Kutsia, G. Landsmann, F. Lichtenberger, P. Paule, H. Rolletschek, J. Schicho, C. Schneider, W. Schreiner, W. Windsteiger, F. Winkler.

Supported by: Linzer Hochschulfonds, Upper Austrian Government, Austrian Federal Ministry of Science and Research (BMWF), Johann Radon Institute for Computational and Applied Mathematics of the Austrian Academy of Sciences (RICAM)

Copyright notice: Permission to copy is granted provided the title page is also copied.

Preface

This report contains the preliminary proceedings of the *8th International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2008)*. The workshop was held in the Castle of Hagenberg, Austria on July 14, 2008 and collocated with the 19th International Conference on Rewriting Techniques and Applications (RTA 2008).

The workshop promotes and stimulates research and collaboration in the area of strategies. It encourages the presentation of new directions, developments, and results as well as surveys and tutorials on existing knowledge in this area. Previous editions of the workshop were held in Utrecht (2001), Copenhagen (2002), Valencia (2003), Aachen (2004), Nara (2005), Seattle (2006), and Paris (2007).

WRS 2008 received 11 submissions. Each submission was assigned to at least 3 program committee members, who reviewed the papers with the help of 10 external referees. After careful deliberations the program committee decided to accept 8 papers, which are contained in this report. The program also included an invited talk by *Yves Lafont* on *Diagram Rewriting: Examples and Theory*

Many people helped to make WRS 2008 a success. I am grateful to the members of the program committee and the external reviewers for their work. A special thanks to Temur Kutsia, who chaired the organisation committee of RTA 2008 and affiliated events. The financial support of the following sponsors is gratefully acknowledged: Linzer Hochschulfonds, Upper Austrian Government, Austrian Federal Ministry of Science and Research (BMWF), and Johann Radon Institute for Computational and Applied Mathematics of the Austrian Academy of Sciences (RICAM) Finally, the EasyChair system of Andrei Voronkov considerably eased the task of the program chair.

Innsbruck, June 2008 *Aart Middeldorp*

Conference Organization

Program Chair

Aart Middeldorp

Program Committee

Elvira Albert	*Madrid*
Gabriele Keller	*Sydney*
Helene Kirchner	*Bordeaux*
Temur Kutsia	*Linz*
Ian Mackie	*Paris*
Aart Middeldorp	*Innsbruck*
Pierre-Etienne Moreau	*Nancy*
Michael Norrish	*Canberra*
Femke van Raamsdonk	*Amsterdam*
Kristoffer Rose	*Yorktown Heights*
Amr Sabry	*Bloomington*
Masahiko Sakai	*Nagoya*

Local Arrangements

Temur Kutsia

External Reviewers

Oana Andrei
Paul Brauner
Samir Genaim
Rémy Haemmerlé
Martin Korp
Naoki Nishida
Miguel Palomino
Yann Radenac
René Thiemann
Christian Urban

Table of Contents

New Developments in Environment Machines 1
Maribel Fernández, Nikolaos Siafakas

Strategy-Based Rewrite Semantics for Membrane Systems Preserves Maximal Concurrency of Rewrite Actions 16
Dorel Lucanu

Recognizing Strategies 31
Bastiaan Heeren, Johan Jeuring

Operational Termination of Conditional Rewriting with Built-in Numbers and Semantic Data Structures 46
Stephan Falke, Deepak Kapur

Completion as Post-Process in Program Inversion of Injective Functions 61
Naoki Nishida, Masahiko Sakai

A Transformational Approach to Prove Outermost Termination Automatically 76
Matthias Raffelsieper, Hans Zantema

Computing with Diagrams in Classical Logic 91
Pierre Lescanne, Dragiša Žunić

Closure of Tree Automata Languages under Innermost Rewriting . 110
Adrià Gascón, Guillem Godoy, Florent Jacquemard

New Developments in Environment Machines

Maribel Fernández[1] and Nikolaos Siafakas[2]

Department of Computer Science, King's College London, Strand, London WC2R 2LS, U.K.

Abstract

In this paper we discuss and compare abstract machines for the lambda-calculus, implementing various evaluation strategies. Starting from the well-known Categorical abstract machine (CAM) and Krivine's abstract machine (KAM), we develop two families of machines that differ in the way they treat environments. The first family is inspired by the work on closed reduction strategies, whereas the second is built in the spirit of the jumping machines based on the work done on Linear Logic

Keywords: λ-calculus, environment machines, explicit substitutions, Linear Logic.

1 Introduction

The λ-calculus is regarded as the theoretical foundation of functional programming languages. It can be thought of as a simple, lexically scoped programming language. Abstract machines are one of the tools one can use to provide a formal operational semantics to a programming language. Abstract machines are transition systems that bridge the gap between the specification of a dynamic semantics of a language and a concrete implementation. They may be considered as rewriting systems, where the rewriting rules have no superpositions.

There is no measure of the abstractness of an abstract machine, however, we will adopt some terminology: following [1] we refer to *abstract machines* as transition systems that accept abstract syntax-trees (in our case, pure λ-terms) as source-syntax, while machines that work with an instruction set will be called *virtual machines*, or *concrete machines* if there exists a hardware implementation that offers the particular instruction set.

We are interested in a particular kind of abstract machine, called *environment machine*. The components that are common to all environment machines are: the expression being evaluated, a control stack, and the environment which provides the bindings for the free variables in the expression. Functions are represented via

[1] Email: maribel.fernandez@kcl.ac.uk

[2] Email: nikolaos.siafakas@kcl.ac.uk

closures, that is, a piece of syntax coupled with an environment containing the mappings that provide values for the free variables.

Many of the environment machines that we encounter in the literature are resource unconscious: the memory model is often unspecified, yet referential transparency is guaranteed; some external machinery is always assumed (a garbage collector, a cloning device) or is given if the memory-layout is more concrete. This level of abstraction, although convenient when analyzing properties of the machines, is not sufficiently detailed from an implementation point of view. Linear Logic [14] addresses resource management issues in proofs, and the same techniques can be applied to the λ-calculus. Several abstract machines for the λ-calculus have been developed following this approach (see for example [8,21,10]), and also, at a more abstract level, several reduction strategies have been defined to take into account the management of resources in the β-reduction process (or more specifically, in the propagation of substitutions), see for example [11,12].

In this paper, starting from (variants of) two well-known abstract machines — Krivine's call-by-name machine [17], which we will call KAM, and the Categorical abstract machine [5], called CAM — we develop two families of machines that exploit recent work on evaluation strategies in the λ-calculus. The first family of machines, which we call λ_c-machines, is based on the work on closed reduction [11,12,22] and includes call-by-value and call-by-name machines that can be seen as a refinement of the CAM and KAM, respectively. More interestingly, this family of machines includes machines that perform reductions under abstractions, similarly to the closed-reduction machine discussed in [12,22]. The second family of machines, which we call δ-machines, are designed to facilitate compilation: the motivation is to have memory management included in the definition. The notion that we want to make significant here is that of a "virtual copy". The machines operate without the need for a garbage collector and each step may be accomplished in amortised $O(1)$ time.

All the machines described in this paper have been implemented; the prototypes are available from `http://www.dcs.kcl.ac.uk/pg/siafakas/`.

2 Background

The first *environment machine*, the SECD-machine [18], dates back to the 60s. It is a call-by-value machine, where arguments are evaluated from right-to-left. The Categorical Abstract Machine [5], also a call-by-value machine, is equipped with an instruction set and may be regarded as a virtual machine.

Krivine's call-by-name environment machine [17] is simple and influential. A lazy (call-by-need) version of the KAM can be found in [13]; the ZINC abstract machine [19] may be thought of as a call-by-value version of Krivine's abstract machine.

There are indeed many machines available in the literature that can be seen as variants of the KAM or the CAM (see the discussion in [1]). Most of these machines aim at reducing programs (i.e., closed λ-terms) to weak head normal form; they do not perform reductions under abstractions. A *strong reduction* version of the ZAM was studied in [16]. In terms of programming languages, one can think of reduction

under an abstraction as a "specialisation" of a function definition. The standard approach that is taken to achieve such a specialisation is to define a weak abstract machine which then calls new instances of itself inside the bodies of abstractions, using the normalisation by evaluation technique (see e.g. [4,12]).

Thus, environment machines can be classified according to the strategy of evaluation they implement (e.g., call-by-name, call-by-value), according to the kind of reduction performed (e.g., weak reduction, strong reduction), and also according to the way they associate environments to terms. For instance, in the CAM and the KAM, environments are associated to sub-terms of the term being evaluated, and provide information about the terms that should be substituted for the free variables in the sub-term. Later we will describe machines that will use only one environment instead of associating environments to sub-terms. To distinguish these two classes of machines, we say that the first class of machines use *local environments* whereas the latter use *global environments*.

3 Abstract machines with local environments

We start by presenting two weak machines with local environments that use names for variables, instead of the standard presentation using De Bruijn indices [9]. The machines are variants of the KAM and the CAM with Lisp-like associative lists to represent environments. Since we use names, we may have duplicate entries for keys in the list, however, keys that arise earlier in the list shadow keys that arise later in the list. In this way, one can implement maps in an non-destructive manner.

We then show how the management of resources (i.e., the environment) can be improved, using techniques inspired by the work on closed reduction [11,12].

3.1 A call-by-name environment machine with names

We specify the machine as a transition system, with a set of transition rules on configurations. A *configuration* of the machine consists of a λ-term and an environment (i.e., a closure), and a stack of closures. Environments are presented in a non-standard way as lists of substitutions. We use the infix, right associative (:) operator and the empty list is denoted by []. Substitutions and closures are represented as pairs, with a tag c (for closure) or s (for substitution). The machine is loaded with a term, an empty environment (when no other domain is provided in advance), and an empty stack; a successful run yields nothing but a closure. The transition rules for the machine are given in Table 1.

To prove the correctness of the machine we define, following [6], a calculus of closures with a call-by-name operational semantics. Terms are written $t[s]$ where t is a pure λ-term and s is a list of substitutions of the form (u, x) where x is a variable and u a term in the calculus, that is, each term $t[s]$ is a closure.

We can obtain the pure λ-term represented by a closure $t[s]$ by using s as a substitution in t; we denote the result as $Subst(t, s)$. If $Subst(t, s)$ is a closed λ-term, we say that the closure $t[s]$ is closed. The *values* of closed closures are abstractions $(\lambda x.t)[s]$. The relation $t[s] \to_{CBN} v$ defines the value v of a closed

Term	Env	Stack		Term	Env	Stack	Cond	Rule
tu	e	s	$\rightarrow$	t	e	$(u,e)^c : s$		Push
$\lambda x.t$	e	$(u,e')^c : s$	$\rightarrow$	t	$((u,e')^c, x)^s : e$	s		Pass
y	$((u,e')^c, x)^s : e$	s	$\rightarrow$	y	e	s	$x \not\equiv y$	EntF
y	$((u,e')^c, x)^s : e$	s	$\rightarrow$	u	e'	s	$x \equiv y$	EntT
$\lambda x.t$	e	$[]$	$\not\rightarrow$					Halt

Table 1
Call-by-name environment machine

closure $t[s]$ in a call-by-name strategy.

$$\frac{v \ value}{v \rightarrow_{CBN} v} \qquad \frac{t \rightarrow_{CBN} v}{x[(t,x):s] \rightarrow_{CBN} v} \qquad \frac{x[s] \rightarrow_{CBN} v}{x[(t,y):s] \rightarrow_{CBN} v}$$

$$\frac{t[s] \rightarrow_{CBN} (\lambda x.t')[s'] \quad t'[(u[s],x):s'] \rightarrow_{CBN} v}{(tu)[s] \rightarrow_{CBN} v}$$

Definition 3.1 (Compilation and De-compilation) *The de-compilation (or read-back) of a machine configuration $(t, e, [s_1 \dots s_n])$ $(n \geq 0)$ is a term obtained by*

$$decomp \quad (t, e, [(u_1, e_1')^c : \dots : (u_n, e_n')^c]) = t[e] \quad u_1[e_1'] \dots u_n[e_n']$$

The function comp compiles terms from the calculus of closures into machines states, using two auxiliary functions (compc compiles closures and comps compiles substitutions; they are mutually recursive):

$$comp \quad t[s] = (t, (comps \quad s), [])$$

where

$$\begin{aligned} comps \quad & [] && = [] \\ comps \quad & [(u,x):s] && = [(compc \quad u, x)^s : (comps \quad s)] \\ compc \quad & u[s] && = (u, comps \quad s)^c \end{aligned}$$

We say that a configuration in the abstract machine is closed *when $decomp(t, e, s)$ is a closed term.*

The following properties are used in the proof of correctness of the machine with respect to the call-by-name strategy:

Proposition 3.2 (i) *An irreducible, closed configuration in the call-by-name machine has the form $(\lambda x.t, e, [])$.*

Term	Env	Stack		Term	Env	Stack	Cond
tu	e	s	$\to$	t	e	$(u,e)^Q : s$	
$\lambda x.t$	e	$(u,e')^Q : s$	$\to$	u	e'	$(\lambda x.t,e)^P : s$	
$\lambda x.t$	e	$(\lambda x'.t',e')^P : s$	$\to$	t'	$((\lambda x.t,e)^c,x') : e'$	s	
y	$((u,e')^c,x)^s : e$	s	$\to$	y	e	s	$x \not\equiv y$
y	$((u,e')^c,x)^s : e$	s	$\to$	u	e'	s	$x \equiv y$
$\lambda x.t$	e	$[]$	$\not\to$				

Table 2
Call-by-value environment machine

(ii) *If* $(t,e,s) \to (t',e',s')$ *then* $(t,e,s \circ s'') \to (t',e',s' \circ s'')$*, where* $\circ$ *denotes list concatenation. Therefore, if* $(t,e,[]) \to^* (t',e',[])$ *also* $(t,e,s) \to^* (t',e',s)$ *for any* s*.*

Theorem 3.3 (Correctness of the Call-by-name Environment Machine) *Let* t *be a closed* λ*-term.*

(i) *If* $t[] \to_{CBN} v$ *using the call-by-name strategy then comp* $t[] = (t,[],[]) \to^* (u,e,[])$ *final, and* $(u,e,[]) =$ *comp* v*.*

(ii) *If comp* $t[] = (t,[],[]) \to^* (u,e,s)$*, where* (u,e,s) *is irreducible, then* $t \to_{CBN}$ *decomp*(u,e,s)*.*

3.2 A call-by-value environment machine with names

We now describe a machine isomorphic to the eager machine given in [10] and closely related to the Categorical Abstract Machine. The configurations of the machine consist of a λ-term, the environment (a list of substitutions mapping variables to closures; closures are terms with environments), and a stack that contains closures tagged either with Q or P. As before, closures and substitutions are represented as pairs. We attach a tag s to a pair of a variable and a closure to indicate that we are using it as a substitution; we attach a tag c to the closure part of the substitution. In the stack, closures are tagged with Q or P. We use the tag Q for arguments, and mark with a P functions stored in the stack while their arguments are evaluated. The transition rules for the abstract machine are given in Table 2.

It is easy to see that this machine is exactly the call-by-value machine defined in [10], but using names of variables instead of De Bruijn indices. The proof of correctness can be easily adapted from the one in [10].

3.3 Improving the management of resources

In the machines given above, environments carry a lot of useless information during computation due to the uncontrolled distribution of substitutions. A good representation of environments is essential in order to get efficient implementations.

Name	Term	Variable Constraint	Free Variables
Variable	x	$-$	$\{x\}$
Abstraction	$\lambda x.t$	$x \in \mathsf{fv}(t)$	$\mathsf{fv}(t) - \{x\}$
Application	tu	$\mathsf{fv}(t) \cap \mathsf{fv}(u) = \emptyset$	$\mathsf{fv}(t) \cup \mathsf{fv}(u)$
Erase	$\epsilon_x.t$	$x \notin \mathsf{fv}(t)$	$\mathsf{fv}(t) \cup \{x\}$
Copy	$\delta_x^{y,z}.t$	$x \notin \mathsf{fv}(t), y \neq z, \{y,z\} \subseteq \mathsf{fv}(t)$	$(\mathsf{fv}(t) - \{y,z\}) \cup \{x\}$
Closure	$t[\overline{(u,x)}]$	$\overline{x} \in \mathsf{fv}(t),\ \forall i \neq j, x_i \neq x_j, \mathsf{fv}(u) = \emptyset$	$\mathsf{fv}(t) - \overline{x}$

Table 3
λ_c-terms and variable constraints

Explicit substitution calculi provide a theoretical framework to explain how to deal with the environments. Linear Logic [14] adds explicit control over the assumptions in proofs, and some explicit substitution calculi exploit these ideas to improve the management of substitutions (see for example [20,11]). Thus, an explicit substitution calculus that controls the distribution of substitutions is a good candidate to specify environment machines in a resource conscious way. We will use a calculus of explicit substitutions that we call λ_c, inspired by [11].

Definition 3.4 (Terms in the λ_c-calculus) *We use x, y, z to denote variables, t, u, v to denote terms, and $\overline{o}$ to denote a sequence of elements $o_1, \ldots, o_n$. Table 3 shows the term constructions, together with the variable constraints that must be satisfied for each construction, and the associated free variables.*

The variable constraints imply that each variable occurs free in a term exactly once. Note that in closures, i.e., λ_c-terms of the form $t[s]$, s contains substitutions for variables that must occur free in t; it may contain several pairs (u, x), or none, in the latter case we write $t[id]$. In all the pairs (u, x) occurring in s the term u is closed. Also, unlike in the previous system, t may also contain closures.

A pure λ-term will be compiled into a λ_c-term by the function defined below.

Definition 3.5 (Compilation) *Let t be a λ-term. Its compilation $[\![t]\!]$ into λ_c is defined as: $[x_1] \ldots [x_n]\langle t\rangle$ where $\mathsf{fv}(t) = \{x_1, \ldots, x_n\}$, $n \geq 0$, we assume w.l.o.g. that the variables are processed in lexicographic order, and $\langle\cdot\rangle$ is defined by: $\langle x\rangle = x$, $\langle tu\rangle = \langle t\rangle\langle u\rangle$, and $\langle \lambda x.t\rangle = \lambda x.[x]\langle t\rangle$ if $x \in \mathsf{fv}(t)$, otherwise $\langle \lambda x.t\rangle = \lambda x.\epsilon_x.\langle t\rangle$. We define $[\cdot]\cdot$ below, using $t[x := u]$ to denote the usual (implicit) notion of substitution.*

$$[x]x = x \qquad [x](\lambda y.t) = \lambda y.[x]t$$
$$[x](\epsilon_y.t) = \epsilon_y.[x]t \qquad [x](\delta_y^{y',y''}.t) = \delta_y^{y',y''}.[x]t$$

$$\begin{aligned}[x](tu) &= \delta_x^{x',x''}.[x'](t[x := x'])[x''](u[x := x'']) && x \in \mathsf{fv}(t), x \in \mathsf{fv}(u), x', x''\ fresh \\ &= ([x]t)u && x \in \mathsf{fv}(t), x \notin \mathsf{fv}(u) \\ &= t([x]u) && x \in \mathsf{fv}(u), x \notin \mathsf{fv}(t)\end{aligned}$$

For example:

$$[\![(xy)(xy)]\!] = [x][y]\langle\!\langle(xy)(xy)\rangle\!\rangle = \delta_y^{y',y''}.\delta_x^{x',x''}.(x'y')(x''y'') \neq [y][x]\langle\!\langle(xy)(xy)\rangle\!\rangle$$

The compilation function returns a λ_c-term without closures. However, to load the abstract machine below, we assume there is a further step in the compilation which adds an empty closure $[id]$ to each sub-term. Although the compilation is defined on open terms, we demand that sufficient substitutions are provided in advance to obtain a closed term.

We will now investigate how call-by-name and call-by-value versions of λ_c-calculi may improve the KAM and CAM respectively. We start with a traditional operational semantics and derive the corresponding abstract machines. Next, we exploit the copying and erasing constructs to mix features from call-by-name and call-by-value. Finally, we discuss a more powerful strategy inspired by [11].

To define the strategies, we use an external function $\bullet$ to extend environments that we define via concatenation:

$$(t[s']) \bullet [s] = t[s' \circ s]$$

Since substitutions are closed, and each variable occurs at most once in s, the concatenation is commutative.

Call-by-name

We define first a call-by-name evaluation strategy for λ_c-terms in Figure 1. The operational semantics is given by a set of axioms and rules, defining a relation $t[s] \to^c_{CBN} v$ between closed closures and values. Values are λ_c-terms of the form $(\lambda y.t)[id]$, that is, the substitutions will be pushed inside abstractions to compute values. Usually, this operation is very costly due to the potential renamings to avoid capture of variables, but we rely on a closed reduction strategy, which does not need α-conversion [11].

The corresponding abstract machine is defined by the transition rules in Table 4 on configurations containing a λ_c-term and a stack of λ_c-terms.

The correctness of the machine with respect to the operational semantics is easy to prove, the proof follows the same lines as the correctness proofs mentioned earlier in this section. We state the main results below:

Proposition 3.6 (i) *An irreducible, closed configuration in the call-by-name λ_c machine has the form $((\lambda x.t)[id], [])$.*

(ii) *If $(t, s) \to (t', s')$ then $(t, s \circ s'') \to (t', s' \circ s'')$, where $\circ$ denotes list concatenation. Therefore, if $(t, []) \to^* (t', [])$ also $(t, s) \to^* (t', s)$ for any s.*

Theorem 3.7 (Correctness of the Call-by-name λ_c Environment Machine) *Let t be a closed λ_c-term obtained by compiling a closed λ-term u.*

$$\frac{t \to^c_{CBN} v}{x[(t,x)] \to^c_{CBN} v} \qquad \frac{}{(\lambda y.t)[id] \to^c_{CBN} (\lambda y.t)[id]}\,Ax \qquad \frac{(\lambda y.t \bullet [(u,x)])[s] \to^c_{CBN} v}{(\lambda y.t)[(u,x):s] \to^c_{CBN} v}$$

$$\frac{x \in fv(t) \quad ((t \bullet [(u',x)])u)[s] \to^c_{CBN} v}{(tu)[(u',x):s] \to^c_{CBN} v} \qquad \frac{x \in fv(u) \quad (t(u \bullet [(u',x)]))[s] \to^c_{CBN} v}{(tu)[(u',x):s] \to^c_{CBN} v}$$

$$\frac{t \to^c_{CBN} (\lambda x.r)[id] \quad r \bullet [(u,x)] \to^c_{CBN} v \quad fv(u) = \emptyset}{(tu)[id] \to^c_{CBN} v}\,Beta$$

$$\frac{(\delta^{y'y''}_y.t \bullet [(u,x)])[s] \to^c_{CBN} v}{(\delta^{y'y''}_y.t)[(u,x):s] \to^c_{CBN} v} \quad \frac{(t \bullet [(u,x'),(u,x'')]) \bullet [s] \to^c_{CBN} v}{(\delta^{x'x''}_x.t)[(u,x):s] \to^c_{CBN} v}\,Delta$$

$$\frac{(\epsilon_y.t \bullet [(u,x)])[s] \to^c_{CBN} v}{(\epsilon_y.t)[(u,x):s] \to^c_{CBN} v} \quad \frac{t \bullet [s] \to^c_{CBN} v}{(\epsilon_x.t)[(u,x):s] \to^c_{CBN} v}$$

Fig. 1. Call-by-name evaluation of λ_c-terms

Term	Stack		Term	Stack	Cond
$(tu)[(u',x):e]$	s	$\to^c_{CBN}$	$((t \bullet [(u',x)])u)[e]$	s	$x \in fv(t)$
$(tu)[(u',x):e]$	s	$\to^c_{CBN}$	$(t(u \bullet [(u',x)]))[e]$	s	$x \in fv(u)$
$x[(u,x)]$	s	$\to^c_{CBN}$	u	s	
$(\lambda x.t)[(u,y):e]$	s	$\to^c_{CBN}$	$(\lambda x.t \bullet [(u,y)])[e]$	s	$x \not\equiv y$
$(\epsilon_y.t)[(u,x):e]$	s	$\to^c_{CBN}$	$(\epsilon_y.t \bullet [(u,x)])[e]$	s	$x \not\equiv y$
$(\epsilon_x.t)[(u,x):e]$	s	$\to^c_{CBN}$	$t \bullet [e]$	s	
$(\delta^{y'y''}_y.t)[(u,x):e]$	s	$\to^c_{CBN}$	$(\delta^{y'y''}_y.t \bullet [(u,x)])[e]$	s	$x \not\equiv y$
$(\delta^{x'x''}_x.t)[(u,x):e]$	s	$\to^c_{CBN}$	$(t \bullet [(u,x'),(u,x'')]) \bullet [e]$	s	
$(tu)[]$	s	$\to^c_{CBN}$	t	$u:s$	
$(\lambda x.t)[]$	$u:s$	$\to^c_{CBN}$	$t \bullet [(u,x)]$	s	
$(\lambda x.t)[]$	$[]$	$\not\to^c_{CBN}$			

Table 4
Call-by-name λ_c abstract machine

(i) *If* $t[] \to^c_{CBN} v$ *then* $(t,[]) \to^* (v,[])$ *final.*

(ii) *If* $(t,[]) \to^* (v,[])$ *final, then* $t \to^c_{CBN} v$.

In order to define a call-by-value strategy, $\to^c_{CBV}$, for λ_c, we just need to replace

$$\frac{}{(\lambda y.t)[s] \to^c_{CBVN} (\lambda y.t)[s]}\,Ax' \qquad \frac{t \to^c_{CBVN} (\lambda x.r)[s] \quad r \bullet [(u,x) \circ s] \to^c_{CBVN} v}{(tu)[id] \to^c_{CBVN} v}\,Beta'$$

$$\frac{u \to^c_{CBVN} u' \quad t \bullet ([(u',x'),(u',x'')] \circ s) \to^c_{CBVN} v}{(\delta_x^{x'x''}.t)[(u,x) : s] \to^c_{CBVN} v}\,Delta'$$

Table 5
$\to^c_{CBVN}$-evaluation of λ_c-terms

the rule *Beta* in the definition of $\to^c_{CBN}$ by the following rule:

$$\frac{t \to^c_{CBV} (\lambda x.r)[id] \quad u \to^c_{CBV} v' \quad r \bullet [(v',x)] \to^c_{CBV} v \quad fv(u) = \emptyset}{(tu)[id] \to^c_{CBV} v}\,Beta$$

We omit the corresponding call-by-value environment machine.

The right time for evaluating the argument

The relations $\to^c_{CBV}$ and $\to^c_{CBN}$ differ in the way the argument is evaluated. We can take profit of the explicit copy constructs in the λ_c syntax, and trigger the evaluation of the argument just before copying. This gives us a reduction strategy that is in-between call-by-name and call-by-value (but it does not correspond exactly to a call-by-need strategy, since the existence of a copy construct does not imply neededness; see [11,12] for a more detailed discussion).

To define a strategy that evaluates substitutions before copying, we use the rules in Table 5 instead of the corresponding rules of $\to^c_{CBN}$ (see Figure 1). We also put some laziness in the way substitutions are propagated (we will avoid pushing substitutions through abstractions). In the corresponding abstract machine, we use tagged terms, in the same way as in the categorical abstract machine. We do reduce the substitution just before copying: in the operational semantics, it is easy to see that a substitution can evaluate only to an abstraction. We use Q to tag arguments as before, however, P now tags copying constructs instead of functions. The transition rules for the abstract machine are the rules in rows 1-7 in Table 4 together with the rules in Table 6.

Reduction under abstraction - improving the strategy

It is known [2] that no usual strategy based on environments can achieve Lévy's optimality. An efficient operational semantics that relaxes some of the demands of optimal reduction has been given in [11], where a high degree of sharing may be achieved. The idea is that we can take advantage of our ability to reduce under abstractions (since we are in an α-conversion free calculus), but not at the top-level, since we want to stop on a weak-head normal form. In particular, it is only useful to perform these extra reductions on a term which will be copied, in order to share these reductions. We refer the reader to [11] for the definition of an evaluation strategy for λ_c that interleaves a weak strategy with a stronger one, called only before an application of the δ rules.

Term	Stack		Term	Stack
$(tu)[id]$	s	$\rightarrow^c_{CBVN}$	t	$(u)^Q : s$
$(\delta^{x'x''}_x.t)[(u,x) : e]$	s	$\rightarrow^c_{CBVN}$	u	$((\delta^{x'x''}_x.t)[e])^P : s$
$(\lambda x.t)[e]$	$(u)^Q : s$	$\rightarrow^c_{CBVN}$	$t \bullet ([(u,x)] \circ e)$	s
$\underbrace{(\lambda_._)[_]}_{B}$	$((\delta^{x'x''}_x.t)[e])^P : s$	$\rightarrow^c_{CBVN}$	$t \bullet ([(B,x'),(B,x'')] \circ e)$	s
$(\lambda x.t)[e]$	$[]$	$\not\rightarrow^c_{CBVN}$		

Table 6
The Cbvn-machine

4 Abstract machines with global environments

In the latter machines, each environment is coupled with a term whereas the KAM and CAM machines explicitly define an environment pointer in their transitions. Yet another way to couple terms with environments is to define a global lookup-table, an array to be precise, that contains the bindings. Intuitively, we will use each variable name from a λ_c-term as an array-index: we work with the previously defined compilation with the main difference that every variable name appears uniquely in the compiled term. Additionally, since we work with arrays, we assume that variable names are natural numbers.

In the rest of this section, we present two experimental machines based on global environments. First we define a machine that works with linear terms (no instance of δ and ϵ-terms) and then we develop techniques that will allow us to move to the full case.

The linear J-Machine

Configurations consist of a pure linear term, an array (a) that stores unevaluated arguments and a stack (s) of pure terms. We write a_n to access the argument at array index n and $a \,|\, [(n,u)]$ to destructively update the element at array index n with u. The transitions of the machine are given below:

Term	GEnv	Stack		Term	GEnv	Stack
tu	a	s	$\rightarrow^g_{LIN}$	t	a	$u : s$
$\lambda n.t$	a	$u : s$	$\rightarrow^g_{LIN}$	t	$a \,|\, [(n,u)]$	s
n	a	s	$\rightarrow^g_{LIN}$	a_n	a	s
$\lambda n.t$	a	$[]$	$\not\rightarrow^g_{LIN}$			

The work-flow of the machine is simple: we jump along a fixed term while we update and retrieve global variables. Indeed, the machine was first defined in [10] in order to study the so called "jumping machines" in the context of Linear Logic. It

is easy to see that in the array, we do not overwrite any previously stored arguments since the machine does not copy any environments. The distinguishing feature of this machine is that it does not require any external machinery and every transition takes $O(1)$ time.

The δ-machine

The problem is the following: how do we use the previous array design in a non-linear way? Clearly, we cannot expect to fit all bindings in an array whose size is bound by the size of the term! Another problematic issue is the following: the machine does not group environments in lexical scopes and everything in the array is accessible at any time. In the sequel, we provide an ad-hoc machine that works with λ_c-terms in a call-by-value fashion.

In order to deal with the first problem, we assume the same unique name convention; however, since copying will take place, each array-index (variable name) cannot identify a single binding. Thus, let us look at the environment part of our configurations in more detail: every array index initially identifies an empty list structure. Each list contains triples of the form (c, u, to) where the first and last elements are strings $c := R \circ c \mid S \circ c \mid \epsilon$. Their purpose is to construct on the fly a *copy-address* such that we can keep a record when we jump into and out of "virtual copies" of terms. The middle element is reserved for arguments which are pure λ_c-terms. We use the auxiliary function $(a_n)_c^{find}$, which yields a triple from the list stored at array index n. The triple is found based on its first element c. We also use the auxiliary function $(a_n)_c^{del}$, which yields a list, where the triple identified by c is removed. Updating a list element of an array is done as before; the difference is that we may have multiple updates at several lists of the array: $a \mid [\overline{(n, l)}]$ where now $\overline{o}$ denotes a sequence of several index - element pairs.

Hence our configurations consist of the following:

- pure λ_c-terms
- the mentioned array of lists that builds the global environment
- a pointer to a *current-copy-address*: it indicates the current copy in which we work at a given time.
- a stack of λ_c-terms tagged with Q , P and also with a *copy-address*; hence we write: $(u)_c^{P|Q}$

The machine is loaded with a λ_c-term, the array of lists where each list is initialised to the empty one and we set the *current-copy-address* to ϵ. The latter indicates that initially, we do not work inside a copy. The transitions are defined in Table 7. There are two external functions, namely *copy* and *cleanup*, that deal with duplicating and cleaning up parts from the global environment. It is not too difficult to internalise these using a system of marks on the stack, but we prefer to keep them separate for simplicity. In both cases, we need one of our triples, a stack (r) to keep track of recursive calls, the free variables of the argument and a pointer to the global environment. Copying is carried out by the rules in Table 8. Cleaning up behaves in a similar fashion: all we need to do is to modify the first rule in Table 8 such that the copies tr' and tr'' are omitted in the definition (i.e. nothing

Term	Env	CurC	Stack	Term	Env	CurC	Stack
tu	a	c	s	t	a	c	$(u)^{Q}_{c} : s$
$\lambda n.t$	a	c	$(u)^{Q}_{c'} : s$	u	a	c'	$(\lambda n.t)^{P}_{c} : s$
$\lambda n.t$	a	c	$(\lambda n'.t')^{P}_{c'} : s$	t'	$a \mid [(n', (c', \lambda n.t, c) : a_{n'})]$	c'	s
n	a	c	s	u	$a \mid [(n, (a_n)^{del}_{c})]$	to	s
				where $(c, u, to) = (a_n)^{find}_{c}$			
$\delta^{n'n''}_{n}.t$	a	c	s	t	$copy((c, u.to), fv(u), a')$	c	s
				where $(c, u, to) = (a_n)^{find}_{c}$ $a' = a \mid \left[(n, (a_n)^{del}_{c}), (n', (c, u, R \circ to) : a_{n'}), (n'', (c, u, S \circ to) : a_{n''}) \right]$			
$\epsilon_n.t$	a	c	s	t	$cleanup((c, u.to), fv(u), a')$	c	s
				where $(c, u, to) = (a_n)^{find}_{c}$ $a' = a \mid [\, (n, (a_n)^{del}_{c}) \,]$			
$\lambda n.t$	a	c	$[]$				

Table 7
δ-machine transitions

triple	fvs	Env	stack	triple	fvs	Env	stack
(c, u, to)	$f' : f$	a	r	(c, u, to)	f	$a \mid [(f', tr' : tr'' : (a_{f'})^{del}_{to})]$	$(to, u', to') : r$
				where $(to, u', to') = (a_{f'})^{find}_{to}$ $tr' = ((S \circ to), u', (S \circ to'))$ $tr'' = ((R \circ to), u', (R \circ to'))$			
_	$[]$	a	$(c, u, to) : r$	(c, u, to)	$fv(u)$	a	r
_	$[]$	a	$[]$				

Table 8
Copying in the δ-machine

is concatenated). The effect of this is that we simply delete unwanted triples.

Example 4.1 We compile the term $(\lambda x.xx)((\lambda x.x)(\lambda x.x))$ into an λ_c-term and

α-convert variable names as appropriate. For the sake of clarity, we use names instead of natural numbers. Thus, the result of the compilation is: $(\lambda x.\delta_x^{x'\,x''}.x'x'')((\lambda y.y)(\lambda k.k))$. Additionally, we write u^c_{to} instead of (c, u, to). The transitions are shown in the following page. The essential point of the machine is to demonstrate how the *copy-address* is maintained at each transition: the δ-terms generate a *copy-address* for each copy and if we move inside a copy, we update the *current-copy-address* of the machine. The last line in the example shows the final state of the machine: here, the computation yields the identity function and the *current-copy-address* indicates that we jumped into the S-copy of the two copies that the δ-term generates. Notice that in this example, the externally defined systems are not utilised: this is because the terms that we copy have no free variables.

Discussion

The idea to use copy addresses comes from the Geometry of Interaction interpretation for Linear Logic [15] and the path based realisation in [21]. There, a λ-term is compiled into a reversible automaton whose states describe a traversal over a fixed Linear Logic proof-net and the configurations of the machine maintain a stratified representation of the contexts traversed so far. In particular, sharing in the net is maintained via contraction nodes (fan nodes) whose traversal builds $(R \mid S)^*$-strings to discriminate between shared contexts.

In our case, sharable information is represented via δ-nodes where we create copy addresses to discriminate between different copies. Counting copies seems to give a curious kind of scope, a "copy scope". From a different point of view, one may argue that the machine utilises on the fly α-conversion to maintain a unique variable name invariant: with such an interpretation, our variable names consist of a fixed part (given at initialisation) and a volatile part (the copy address) where we use the latter to maintain the invariant when we copy arguments. Notice that the bindings of each copy are tagged by distinct addresses. However, we have to know in what copy we are currently in so that we can pick up a correct binding. In a nutshell, the trick to make this happen is to remember the current copy address at the point of the call, the callee then creates a binding at the current copy address where we also remember the copy address of the caller and finally, at the time we use a binding, we "jump to" the argument and reset the copy address to the one that was active at the time of the call.

However, the copy addresses that we generate are a source of inefficiency because they can get quite big! In order to efficiently utilise associative data-structures, a more compact representation of the addresses would be useful. For instance, one could draw fresh addresses from a free-list. Finally, notice that there is no garbage collector involved. There is however a space leak if we do not make full copies of the copy address.

5 Conclusions

We have presented abstract machines for the λ-calculus that differ in the strategy implemented, and in the way the environments are manipulated. Applying techniques inspired by the work in Linear Logic, we have derived two families of

x	x'	x''	y	k	c	Term	Stack
$[]$	$[]$	$[]$	$[]$	$[]$	ϵ	$(\lambda x.\delta_x^{x'\,x''}.x'x'')((\lambda y.y)(\lambda k.k))$	$[]$
$[]$	$[]$	$[]$	$[]$	$[]$	ϵ	$\lambda x.\delta_x^{x'\,x''}.x'x''$	$((\lambda y.y)(\lambda k.k))_\epsilon^Q : []$
$[]$	$[]$	$[]$	$[]$	$[]$	ϵ	$(\lambda y.y)(\lambda k.k)$	$(\lambda x.\delta_x^{x'\,x''}.x'x'')_\epsilon^P : []$
$[]$	$[]$	$[]$	$[]$	$[]$	ϵ	$\lambda y.y$	$(\lambda k.k)_\epsilon^Q : (\lambda x.\delta_x^{x'\,x''}.x'x'')_\epsilon^P : []$
$[]$	$[]$	$[]$	$[]$	$[]$	ϵ	$\lambda k.k$	$(\lambda y.y)_\epsilon^P : (\lambda x.\delta_x^{x'\,x''}.x'x'')_\epsilon^P : []$
$[]$	$[]$	$[]$	$(\lambda k.k)_\epsilon^\epsilon : []$	$[]$	ϵ	y	$(\lambda x.\delta_x^{x'\,x''}.x'x'')_\epsilon^P : []$
$[]$	$[]$	$[]$	$[]$	$[]$	ϵ	$\lambda k.k$	$(\lambda x.\delta_x^{x'\,x''}.x'x'')_\epsilon^P : []$
$(\lambda k.k)_\epsilon^\epsilon : []$	$[]$	$[]$	$[]$	$[]$	ϵ	$\delta_x^{x'\,x''}.x'x''$	$[]$
$[]$	$(\lambda k.k)_R^\epsilon : []$	$(\lambda k.k)_S^\epsilon : []$	$[]$	$[]$	ϵ	$x'x''$	$[]$
$[]$	$(\lambda k.k)_R^\epsilon : []$	$(\lambda k.k)_S^\epsilon : []$	$[]$	$[]$	ϵ	x'	$(x'')_\epsilon^Q : []$
$[]$	$[]$	$(\lambda k.k)_S^\epsilon : []$	$[]$	$[]$	R	$(\lambda k.k)$	$(x'')_\epsilon^Q : []$
$[]$	$[]$	$(\lambda k.k)_S^\epsilon : []$	$[]$	$[]$	ϵ	x''	$(\lambda k.k)_R^P : []$
$[]$	$[]$	$[]$	$[]$	$[]$	S	$(\lambda k.k)$	$(\lambda k.k)_R^P : []$
$[]$	$[]$	$[]$	$[]$	$(\lambda k.k)_S^R : []$	R	k	$[]$
$[]$	$[]$	$[]$	$[]$	$[]$	S	$(\lambda k.k)$	$[]$

machines. In the first one, environments associate to each sub-term, locally, the information needed about the bindings of the free variables. A syntax for terms with explicit copying and erasing constructs allows us to incorporate some optimisations in the machines. The second family of machines also benefits from the use of a syntax with explicit copying and erasing, but uses a global environment. Machines with global environments are well-suited for compilation. In future work, we plan to derive a compiler for the λ-calculus based on the δ-machine and the machines working with the local environments.

References

[1] M.S. Ager, D. Biernacki, O. Danvy and J. Midtgaard. A Functional Correspondence between Evaluators and Abstract Machines. In *Proceedings of PPDP 2003*, ACM Press.

[2] A. Asperti and S. Guerrini. *The Optimal Implementation of Functional Programming Languages*, volume 45 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1998.

[3] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland Publishing Company, second, revised edition, 1984.

[4] U. Berger, M. Eberl, and H. Schwichtenberg. Normalization by evaluation, 1998.

[5] G. Cousineau, P.-L. Curien, and M. Mauny. The categorical abstract machine. *Science of Computer Programming*, 8:173–202, 1987.

[6] P.-L. Curien. An abstract framework for environment machines. *Theoretical Computer Science*, 82:389–402, 1991.

[7] P.-L. Curien. *Categorical Combinators, Sequential Algorithms and Functional Programming*. Progress in Theoretical Computer Science. Birkhauser, 1993.

[8] V. Danos and L. Regnier. Reversible, irreversible and optimal λ-machines. *Theoretical Computer Science*, 227(1–2):79–97, 1999.

[9] N. G. de Bruijn. Lambda calculus notation with nameless dummies. *Indagationes Mathematicae*, 34:381–392, 1972.

[10] M. Fernández and I. Mackie. Call by value lambda-graph rewriting — without rewriting. In *Proceedings of the Int. Conference on Graph Transformations (ICGT'02)*, Barcelona, 2002. Lecture Notes in Computer Science 2505, Springer, 2002.

[11] M. Fernández, I. Mackie and F-R. Sinot. Closed Reduction: Explicit Substitutions without alpha-conversion. In *Mathematical Structures in Computer Science*, 15(2), 2005.

[12] M. Fernández, I. Mackie and F-R. Sinot. Lambda-Calculus with Director Strings. In *Applicable Algebra in Engineering, Communication and Computing*, 15(6), pages 393–437, Springer, 2005.

[13] D. P. Friedman, A. Ghuloum, J. G. Siek and L. Winebarger. Improving the Lazy Krivine Machine. In *Higher-Order and Symbolic Computation*, 2003.

[14] J.-Y. Girard. Linear Logic. *Theoretical Computer Science*, 50(1):1–102, 1987.

[15] J.-Y. Girard. Geometry of interaction 1: Interpretation of System F. In R. Ferro, C. Bonotto, S. Valentini, and A. Zanardo, editors, *Logic Colloquium 88*, volume 127 of *Studies in Logic and the Foundations of Mathematics*, pages 221–260. North Holland Publishing Company, Amsterdam, 1989.

[16] B. Grégoire and X. Leroy. A compiled implementation of strong reduction. In *Proceedings of ICFP'02, Pittsburgh, Pennsylvania, USA*, 2002.

[17] J-L. Krivine. Un interprète du λ-calcul. Available online at `http://www.logique.jussieu.fr/~krivine`, 1985.

[18] P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6:308–320, 1964.

[19] X. Leroy. The ZINC experiment: an economical implementation of the ML language. Technical Report, INRIA Rocquencourt, 1990.

[20] I. Mackie. *The Geometry of Implementation*. PhD thesis, Department of Computing, Imperial College of Science, Technology and Medicine, September 1994.

[21] I. Mackie. The geometry of interaction machine. In *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages (POPL'95)*, pages 198–208. ACM Press, January 1995.

[22] F-R. Sinot. Stratégies efficaces et modèles d'implantation pour les langages fonctionnels. Thèse de doctorat, École Polytechnique, Palaiseau, France, 2006.

Strategy-Based Rewrite Semantics for Membrane Systems Preserves Maximal Concurrency of Rewrite Actions

Dorel Lucanu[1,2]

Faculty of Computer Science, Alexandru Ioan Cuza University, Iaşi, Romania

Abstract

We use a modal logic in order to show that the strategy-based rewrite semantics for membrane systems fully preserves the maximal concurrency of rewrite actions, whereas the maximal concurrency of communication actions and structural actions is partially preserved. Consequently, the strategy-based rewrite semantics describes more faithfully the behavior of the membrane systems than the rewrite logic-based semantics, which implements the maximal concurrency of the membrane systems only by interleaving concurrency.

Keywords: Rewrite Strategies, Strategy Controller, Membrane System, Modal logic, True Concurrency, Rewrite Logic.

1 Introduction

Membrane computing [17] deals with distributed and parallel computing models inspired from the structure and the functioning of living cells, as well as from the way the cells are organized. Such a model processes multisets of symbol-objects in a localized manner. The locality of processing refers to the fact that the evolution rules and evolving objects are encapsulated into compartments delimited by membranes. An essential role is also played by the communication among compartments and, eventually, with the environment.

There are several approaches [3,1] which describe a rewrite semantics based on rewriting logic (RL) [14,15]. Even if the use of RL framework seems to be a natural choice for specifying and analyzing membrane systems, the locality of evolution rules and the higher degree of the concurrency given by the maximal parallel rewriting (used in defining the behavior of these systems) is quite challenging. An alternative approach based on rewrite strategies and strategy controllers is given in [4]. The

[1] This work is partially supported by the PN II grant ID 393/2007.

[2] Email: dlucanu@info.uaic.ro

main idea is to separate the implementation of the control mechanisms of regions from the effective application of the evolution rules.

In [12] we show that RL-based semantics can describe the maximal parallel rewriting of the membrane systems only by interleaving semantics. In this paper we show that the strategy-based rewrite semantics defined in [4] preserves the maximal concurrency expressed by the maximal parallel rewriting. The concurrency degree of the communications and structural actions is the same in the RL-based semantics and strategy-based rewrite semantics. Since the two formalisms, membrane systems and strategy-based rewriting logic, are quite different, we use a simple modal logic as a common language for comparing the concurrency degrees of the two formalisms.

The paper is structured as follows. Section 2 briefly presents the membrane systems and rewriting logic. In Section 3 a Hennessy-Milner-like modal logic for membrane systems is introduced. Section 4 briefly recalls from [4] the strategy-based rewrite semantics for membrane systems. Section 5 includes the main results of the paper. The concurrency degree of an evolution step of a membrane system is compared with that of its implementation as a strategic rewrite using the modal logic introduced in Section 3. The paper ends with some concluding remarks.

Acknowledgments: The author would like to thank the anonymous referees for their useful remarks and comments.

2 Preliminaries

2.1 Membrane systems

In this paper we consider a particular case of membrane systems, namely that known as *transition P systems* [17]. Informally, a transition P system consists of: an alphabet of objects (a usual finite non-empty alphabet of abstract symbols identifying the objects), the membrane structure (it can be represented in many ways, but the most used one is by a string of labeled matching parentheses), the multisets of objects present in each region of the system (represented in the most compact way by strings of symbol-objects), the sets of evolution rules associated with each region, as well as the indication about the way the output is defined (see, e.g., Figure 1). Formally, a *transition P system* (of degree m) is a construct of the form $\Pi = (O, C, \mu, w_1, w_2, \ldots, w_m, R_1, R_2, \ldots, R_m, i_0)$, where:

(i) O is the (finite and non-empty) alphabet of *objects*,

(ii) $C \subseteq O$ is the set of *catalysts*,

(iii) μ is a *membrane structure*, consisting of m membranes, labeled with $1, 2, \ldots, m$; one says that the membrane structure, and hence the system, is of *degree* m,

(iv) $w_1, w_2, \ldots, w_m$ are multisets over O representing the multisets of objects present in the regions $1, 2, \ldots, m$ of the membrane structure (*contents*),

(v) $R_1, R_2, \ldots, R_m$ are finite sets of *evolution rules* associated with the regions $1, 2, \ldots, m$ of the membrane structure,

(vi) i_0 is either one of the labels $1, 2, \ldots, m$, and then the respective region is the *output region* of the system, or it is 0, and then the result of a computation is

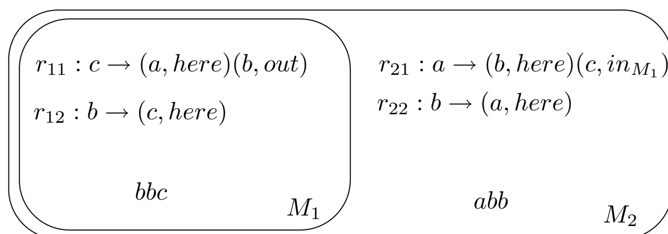


Fig. 1. A P system with two membranes

collected in the environment of the system.

A membrane structure is a hierarchically arranged set of membranes, contained in a distinguished external membrane called the *skin* membrane. Several membranes can be placed inside the skin membrane; a membrane without any other membrane inside it is said to be *elementary*. Each membrane determines a compartment, also called region, the space delimited from above by it and from below by the membranes placed directly inside, if any exists. Clearly, the correspondence membrane-region is one-to-one, that is why we sometimes use interchangeably these terms. The hierarchical structure of membranes is a rooted tree symbolically represented as a string of labeled matching parentheses. The rules have the form $r : u \to v$ or $r : u \to v\delta$, with u a non-empty multiset over O, v a multiset over $O \cup Tar$, where $Tar = \{here, out\} \cup \{in_j \mid 1 \leq j \leq m\}$, and δ a special object called *dissolving action*. The elements of Tar are called *target indications* and have the following meaning: an object having associated the indication *here* remains in the same region, one having associated the indication in_j goes immediately into the directly lower membrane j, and *out* indicates that the object has to exit the membrane, thus becoming an element of the region surrounding it. The rules can be cooperative (with u arbitrary), non-cooperative (with $u \in O \setminus C$), or catalytic (of the form $ca \to cv$ or $ca \to cv\delta$, with $a \in O \setminus C$, $c \in C$, and v a multiset over $(O \setminus C) \times Tar$); note that the catalysts never evolve and never change the region, they only help the other objects to evolve.

In this paper we associate a distinguished name M_j to each membrane j and the name M_j and the index j are used interchangeably.

A *configuration* $(\mu, w_1, \ldots, w_m)$ consists of the membrane structure μ and the multisets w_i of objects from its compartments. During the evolution of the system, both the multisets of objects and the membrane structure can change.

The objects evolve by means of evolution rules. In each time unit a transformation of a configuration of the system, called *transition*, takes place by applying the rules in each region, in a *non-deterministic and maximally parallel manner*. An evolution step in a given region consists in finding a maximal applicable multiset of rules, removing from the region all objects specified in the left hand sides of the chosen rules (with the multiplicities as indicated by the rules and by the number of times each rule is used), producing the objects from the right hand sides of rules, and then distributing these objects as indicated by the targets associated with them. If at least one of the rules introduces the dissolving action δ, then the membrane is dissolved, and its content becomes part of the immediately upper membrane, provided that this membrane was not dissolved at the same time, a case where we

stop in the first upper membrane which was not dissolved (at least the skin remains intact). The rules of the dissolved membranes are lost.

There are many extensions of transitional P systems, among which we mention here the use of priority relation over the evolution rules, the use of promoters and inhibitors, the non-deterministically choosing of the *in* target. The reader is invited to see [17] for a detailed presentation. We associate to each membrane a *control mechanism* specifying its particular way to evolve. Some examples of such mechanisms are presented in Section 3.

2.2 Rewriting Logic (RL)

We assume that the reader is familiar with the basic definitions and notations for many-sorted equational logic [9], term rewriting [5,6], membership equational logic (MEL) [16,7], and and rewriting logic [8,14,15].

Here we consider only *(unconditional) MEL-based rewrite theories* $\mathcal{R} = (\Sigma, E, R)$, where

- (Σ, E) is a MEL theory consisting of a MEL signature Σ and a set E of MEL axioms (membership axioms and equations), and
- R is a set of (universally quantified) labeled (unconditional) rewrite rules having the form $(\forall X)r : u \to v$, with $u, v \in \mathbb{T}_\Sigma(X)_s$ (= the set of terms of sort s and with variables in X) for some sort s and $Var(v) \subseteq Var(u) \subseteq X$.

The rewriting logic of a MEL-based rewrite theory $\mathcal{R}$ consists of

- sentences given by *rewrite sequents*, which are pairs of the form $(\forall X)t \to t'$, with $t, t' \in \mathbb{T}_\Sigma(X)_s$ for some sort s, and
- an *entailment relation* $\mathcal{R} \vdash (\forall X)t \to t'$ defined by a set of inference rules (see, e.g., [8,14] for details).

We give as examples the rewrite theories describing the control-free membranes (no restrictions regarding the application of the evolution rules are considered).

The Rewrite Theory Associated to an Elementary Membrane. The static description of the membranes is represented by the MEL theory (Σ_m, E_m), where Σ_m includes the sorts *Object*, *Soup*, and *HotSoup* with *Object* < *Soup* < *HotSoup*, (w, tar) : *HotSoup* if w : *Soup* and $tar \in Tar$, $\varepsilon : \ \to$ *HotSoup*, the concatenation $_\,_$: *HotSoup HotSoup* $\longrightarrow$ *HotSoup*. E_m includes axioms expressing the associativity and commutativity of $_\,_$ with ε the identity element. The complete description of a control-free membrane M is represented by the MEL rewrite theory $\mathcal{M} = (\Sigma_m \cup O, E_m, R)$, where O the set of object constants, and R includes the rewrite rules corresponding to the evolution rules. For the case of M_1 in Figure 1, O includes the constants a, b, c and R_1 includes the rules r_{11} and r_{12}. Using the inference rules for RL, we may deduce, e.g., that the one-step evolution of M_1 can be described by an one-step concurrent rewrite:

$$\frac{(\forall\emptyset)c \to (a, here)(b, out) \quad (\forall\emptyset)b \to (c, here)}{(\forall\emptyset)bbc \to (c, here)(c, here)(a, here)(b, out)}$$

Note that the above rewrite theory describes M_1 as an independent membrane. We

show below that the description of the behavior of M_1 by the rewrite theory corresponding to the whole system is more complicated.

The Rewrite Theory Associated to a Membrane System. The static description of membrane systems is represented by the MEL theory (Σ_p, E_p) consisting of (Σ_m, E_m) together with:

- a sort *MembraneName* together with a constant M : *MembraneName* for each membrane name M,
- a sort *Membrane* for states of both simple and composite membranes,
- a sort *MembraneBag* for multisets of membranes, together with its constructors: the subsort relation *Membrane* < *MembraneBag*, the constant *NULL* denoting the empty multiset, and the union of multisets
  $_,_ : \mathit{MembraneBag}\ \mathit{MembraneBag} \longrightarrow \mathit{MembraneBag}$ `[assoc comm id:` *NULL*`]`
- the constructors for *Membrane*: $\langle _|_ \rangle : \mathit{MembraneName}\ \mathit{HotSoup} \longrightarrow \mathit{Membrane}$ and $\langle _|_\{_\}\rangle : \mathit{MembraneName}\ \mathit{Soup}\ \mathit{MembraneBag} \longrightarrow \mathit{Membrane}$, together with the axiom $\langle M|w\,\{\mathit{NULL}\}\rangle = \langle M|w\rangle$.

A control-free membrane system Π is described by the rewrite theory $\mathcal{R}_\Pi = (\Sigma_p \cup O, E_p, R)$, where R includes the rewrite rules coming from all the component membranes together with the cooperation (interaction) rules (if any):

$$\begin{aligned}
\mathit{in}(M, M') : {} & \langle\, M \mid w_1(w_2, \mathit{in}_{M'})\ \{\ \langle\, M' \mid w'\ \{\ X\ \}\,\rangle, Y\ \}\,\rangle \rightarrow \\
& \langle\, M \mid w_1\ \{\ \langle\, M' \mid w' w_2\ \{\ X\ \}\,\rangle, Y\ \}\,\rangle \\
\mathit{out}(M', M) : {} & \langle\, M \mid w\ \{\ \langle\, M' \mid w_1'(w_2', \mathit{out})\ \{\ X\ \}\,\rangle, Y\ \}\,\rangle \rightarrow \\
& \langle\, M \mid w w_2'\ \{\ \langle\, M' \mid w_1'\ \{\ X\ \}\,\rangle, Y\ \}\,\rangle \\
\mathit{in\text{-}out}(M, M') : {} & \langle\, M \mid w_1(w_2, \mathit{in}_{M'})\ \{\ \langle\, M' \mid w_1', (w_2', \mathit{out})\ \{\ X\ \}\,\rangle, Y\ \}\,\rangle \rightarrow \\
& \langle\, M \mid w_1 w_2'\ \{\ \langle\, M' \mid w' w_2\ \{\ X\ \}\,\rangle, Y\ \}\,\rangle \\
\mathit{diss}(M', M) : {} & \langle\, M \mid w\ \{\ \langle\, M' \mid w'\delta\ \{\ X\ \}\,\rangle, Y\ \}\,\rangle \rightarrow \langle\, M \mid w w'\ \{\ X, Y\ \}\,\rangle
\end{aligned}$$

The first rule describes the transmission of a message from a parent membrane M to a child membrane M', the second one the transmission of a message from a child membrane M' to the parent membrane M, the third one an exchange of messages between M and M', and the fourth one the dissolving of the membrane M' (this is triggered by the presence of the object δ in the current content of M'). The above set of rewrite rules describing the possible interactions is not minimal. For instance, the effect of the rule *in-out* is equivalent to that of the sequential application of the rules *in* and *out*, in either order.

The rewrite theory $\mathcal{R}_\Pi$ does not include information about the locality of the rewrite rules. For instance, if Π is the system described in Figure 1, then logic defined by $\mathcal{R}_\Pi$ allows to apply r_{12} for both the content of M_1 and the content of M_2. There are different ways for describing the locality of the evolution rules: considering an operation $\mathit{rules}(M)$ - returning the rules of the membrane M [3,4], encoding the set of rules in the description of each membrane [1,12] and so on. In either of these cases the rewriting logic fails to describe an one-step evolution of a membrane by an one-step concurrent rewrite. For instance, if we encode the rules of M_1 by $r_{11} : \langle M_1|c\,W\rangle \rightarrow \langle M_1|(a, \mathit{here}), (b, \mathit{out})\,W\rangle$ and $r_{12} : \langle M_1|b\,W\rangle \rightarrow \langle M_1|(c, \mathit{here})\,W\rangle$, then these cannot be concurrently applied because they overlap.

3 A Modal Logic for Membrane Systems

In this section we define a Hennessy-Milner-like logic [10] able to express the concurrency degree (and the behavior) of a membrane system. We distinguish three kinds of actions in this logic:

(i) *rewrite actions* corresponding to evolution rules of a membrane. Such an action is denoted by the label of the involved rule. We assume that the rules have distinguished labels such that there is no ambiguity regarding the rule or the region the rule belongs to.

(ii) *communication actions* which describe how two parent-child regions communicate. Two kinds of communications are possible: $in(M, M')$ - the region M sends a message to region M' (M' is a child of M); and $out(M', M)$ - the region M' sends a message to surrounding region M.

(iii) *dissolving actions* $diss(M', M)$, meaning that the region M' is dissolved and its contents is sent to surrounding region M; the evolution rules of M' are lost. In general, we may consider *structural actions* meaning all actions aimed to modify the structure of the system.

We associate a modal language $\mathcal{L}_\Pi$ to a membrane system Π as follows:

(i) *true* is a formula in $\mathcal{L}_\Pi$;

(ii) if φ is a formula in $\mathcal{L}_\Pi$ and L a multiset of rewrite actions, then $\langle L\rangle\varphi$ is a formula in $\mathcal{L}_\Pi$;

(iii) if φ is a formula in $\mathcal{L}_\Pi$ and C a multiset of communication actions, then $\langle C\rangle\varphi$ is a formula in $\mathcal{L}_\Pi$;

(iv) if φ is a formula in $\mathcal{L}_\Pi$ and D a multiset of dissolving actions, then $\langle D\rangle\varphi$ is a formula in $\mathcal{L}_\Pi$;

(v) if φ_1 and φ_2 are formulas in $\mathcal{L}_\Pi$, then so are $\neg\varphi_1$ and $\varphi_1 \wedge \varphi_2$.

The other propositional connectors are added to $\mathcal{L}_\Pi$ in the usual way; e.g., *false* is the notation for $\neg true$. The modal operator $[A]\varphi$ is defined as $\neg\langle A\rangle\neg\varphi$, where A denotes a set of actions of the same type. An *elementary formula* is a formula which does not contain a subformula of the form $\langle A\rangle\varphi$ with A a set of communication or dissolving actions. The idea is that an elementary membrane can satisfy only elementary formulas.

We define first the satisfaction relation between elementary membranes and elementary formulas, where the operations over multisets are denoted in a similar way to that one used for sets.

(i) $M, w \models true$ for each w;

(ii) $M, w \models \varphi_1 \wedge \varphi_2$ iff $M, w \models \varphi_1$ and $M, w \models \varphi_2$;

(iii) $M, w \models \neg\varphi$ iff $M, w \not\models \varphi$;

(iv) $M, w \models \langle L\rangle\varphi$ iff there is a transition $w \to w'$ obtained by applying the rules designated by L according to the control mechanism of M and $M, w' \models \varphi$; in this case we say that the pair (w, w') is a *witness* of $\langle L\rangle true$,

where M ranges over the membrane names, w, w' range over the membrane contents.

We extend now the satisfaction relation to compound membranes. In the following Π is a system with the regions $M_1, \ldots, M_m$, μ and μ' range over the struc-

ture of Π, w_i, w'_i range over the contents of the membrane M_i, $\overline{w} = (w_1, \ldots, w_m)$, $\overline{w}' = (w'_1, \ldots, w'_m)$, L ranges over the nonempty sets of rewrite actions, C over the nonempty sets of communication actions, and D over the nonempty sets of dissolving actions.

(i) $\Pi, (\mu, w_1, \ldots, w_m) \models \mathit{true}$ for each $(\mu, w_1, \ldots, w_m)$;
(ii) $\Pi, (\mu, w_1, \ldots, w_m) \models \varphi_1 \wedge \varphi_2$ if and only if $\Pi, (\mu, w_1, \ldots, w_m) \models \varphi_1$ and $\Pi, (\mu, w_1, \ldots, w_m) \models \varphi_2$;
(iii) $\Pi, (\mu, w_1, \ldots, w_m) \models \neg\varphi$ iff $\Pi, (\mu, w_1, \ldots, w_m) \not\models \varphi$;
(iv) $\Pi, (\mu, w_1, \ldots, w_m) \models \langle L\rangle\varphi$ iff $L = L_1 \cup \ldots \cup L_m$ and there is $(\mu, w'_1, \ldots, w'_m)$ such that
  (a) for each i,
    if $L_i = \emptyset$, then $w'_i = w_i$, and
    if $L_i \neq \emptyset$, then (w_i, w'_i) is a witness of $\langle L_i\rangle \mathit{true}$ (hence $M_i, w_i \models \langle L_i\rangle \mathit{true}$)
  (b) and $\Pi, (\mu, w'_1, \ldots, w'_m) \models \varphi$;
  we say that $((\mu, \overline{w}), (\mu, \overline{w}'))$ is a *witness* of $\langle L\rangle \mathit{true}$;
(v) $\Pi, (\mu, w_1, \ldots, w_m) \models \langle C\rangle\varphi$ iff there is $(\mu, w'_1, \ldots, w'_{m'})$ such that for each i,

$$w'_i = (\cup(u \mid (u, \mathit{here}) \subseteq w_i)) \cup (\cup(v' \mid (\exists j)(v', \mathit{in}_{M_i}) \subseteq w_j \wedge \mathit{in}(M_j, M_i) \in C)) \cup\ (\cup(v'' \mid (v'', \mathit{out}) \subseteq w_j \wedge \mathit{out}(M_j, M_i) \in C));$$

  we say that $((\mu, \overline{w}), (\mu, \overline{w}'))$ is a *witness* of $\langle C\rangle \mathit{true}$;
(vi) $\Pi, (\mu, w_1, \ldots, w_m) \models \langle D\rangle\varphi$ iff there is $(\mu', w'_1, \ldots, w'_{m'})$ such that
  (a) μ' is obtained from μ by removing any M_j with $\mathit{diss}(M_j, M_i) \in D$ for certain M_i;
  (b) if M_i is in μ', then $w'_i = w_i\ \cup\ (\cup(w_k \mid \mathit{diss}(M_k, M_i) \in D^+))$ where D^+ is inductively defined by
    $D \subseteq D^+$,
    if $\mathit{diss}(M_j, M_i) \in D$ and $\mathit{diss}(M_k, M_j) \in D^+$, then $\mathit{diss}(M_k, M_i) \in D^+$;
  we say that $((\mu, \overline{w}), (\mu', \overline{w}'))$ is a *witness* of $\langle D\rangle \mathit{true}$;

Using the modal language $\mathcal{L}_\Pi$ with the satisfaction relation previously defined we are able to express the behavior of the system Π. A transition $(\mu, \overline{w}) \Rightarrow (\mu', \overline{w}')$ of a membrane system Π consists of

(i) either only rewrite actions, in that case $\mu' = \mu$ and there is a multiset L of rewrite actions such that $\Pi, (\mu, \overline{w}) \models \langle L\rangle \mathit{true}$ and for each multiset C of communications and multiset D of dissolvings, $\Pi, (\mu, \overline{w}') \models [C]\mathit{false}$ and $\Pi, (\mu, \overline{w}') \models [D]\mathit{false}$;
(ii) or only rewrite and communication actions, in that case $\mu' = \mu$ and there are a multiset L of rewrite actions and a multiset C of communications such that $\Pi, (\mu, \overline{w}) \models \langle L\rangle\langle C\rangle \mathit{true}$ and for each multiset D of dissolvings, $\Pi, (\mu, \overline{w}') \models [D]\mathit{false}$;
(iii) or only rewrite and dissolving actions, in that case there are a multiset L of rewrite actions and a multiset D of dissolvings such that $\Pi, (\mu, \overline{w}) \models \langle L\rangle\langle D\rangle \mathit{true}$ and for each multiset C of communications, $\Pi, (\mu, \overline{w}') \models [C]\mathit{false}$;
(iv) or rewrite, communication and dissolving actions, in that case there are a multiset L of rewrite actions, a multiset C of communications and a multiset D of dissolvings such that $\Pi, (\mu, \overline{w}) \models \langle L\rangle\langle C\rangle\langle D\rangle \mathit{true}$.

For instance, if Π_{12} is the membrane system represented in Figure 1, then

Π_{12}, $([_2[_1]_1]_2, bbc_{M_1}, abb_{M_2}) \models$

$\langle\{r_{11}, r_{12}, r_{12}, r_{21}, r_{22}, r_{22}\}\rangle\langle\{in(M_2, M_1), out(M_1, M_2)\}\rangle true.$

The modal logic can be extended in order to handle control mechanisms of the membranes. Since an evolution step of a membrane is described by a formula $\langle L\rangle true$, we may define a specialized satisfaction relation $\models_{ctrl}$ for each control *ctrl*: $M, w \models_{ctrl} \langle L\rangle true$ if and only if *ctrl* is the control of M and L is a set of rules which can be applied on w according to *ctrl*. Here are the definitions of some controls in this logic.

Maximal parallel rewriting (mpr). We use two specialized satisfaction relations to express that a multiset of rules is applied in a maximal parallel manner: $\models_{\text{mpr0}}$ - for "full maximal parallel rewriting", when each object of the current content is assigned to an evolution rule, and $\models_{\text{mpr}}$ - for the general case, when some objects cannot be assigned to any evolution rule.

(i) $M, u \models_{\text{mpr0}} \langle r\rangle true$ if $r : u \to v$ is an evolution rule of M;

(ii) $M, ww' \models_{\text{mpr0}} \langle L \cup L'\rangle true$ if $M, w \models_{\text{mpr0}} \langle L\rangle true$ and $M, w' \models_{\text{mpr0}} \langle L'\rangle true$;

(iii) $M, w \models_{\text{mpr}} \langle L\rangle true$ iff $w = w'z$ such that $M, w' \models_{\text{mpr0}} \langle L\rangle true$ and $M, z \models [r]false$ for each evolution rule $r : u \to v$ of M.

Maximal parallel rewriting with priorities (pri). We assume that there is a partial order $<$ over the evolution rules, where $r < r'$ stands for "r' has a greater priority than r". The definition of the corresponding satisfaction relation $\models_{\text{pri}}$ is reduced to that of $\models_{\text{mpr}}$:

$M, w \models_{\text{pri}} \langle L\rangle true$ iff $M, w \models_{\text{mpr}} \langle L\rangle true$ and $(\forall r, r')(r \in L, r < r'$ implies $M, w \not\models \langle r'\rangle true)$.

Rules with promoters. A rule with promoters, $r : u \to v|_p$, can be applied only if the promoters p are present in the current content. In the context of maximal parallel rewriting, the definition of the corresponding specialized satisfaction relation is as follows:

$M, w \models_{\text{prom}} \langle L\rangle true$ iff $M, w \models_{\text{mpr}} \langle L\rangle true$ and if $r \in L$ is a rule with promoter, $r : u \to v|_p$, then $p \subseteq w$.

Rules with inhibitors. A rule with inhibitors, $r : u \to v|_q$, can be applied only if the inhibitors q are *not* present in the current content. In the context of maximal parallel rewriting, the definition of the corresponding specialized satisfaction relation is as follows:

$M, w \models_{\text{inh}} \langle L\rangle true$ iff $M, w \models_{\text{mpr}} \langle L\rangle true$ and if $r \in L$ is a rule with inhibitor, $r : u \to v|_{\neg q}$, then $q \cap w = \varepsilon$ (recall that ε is the empty multiset).

Combinations between different controls are also possible, e.g., priorities and promoters.

The modal formulas supply information about the concurrency degree of the membrane systems. The *maximal concurrency* of a set A of actions is expressed by $\langle A\rangle true$ and $[A']false$ for each nonempty set $A' \neq A$: the maximality is expressed by $[A']false$ for all $A' \supset A$, and the concurrency is expressed by $[A']false$ for all $A' \subset A$.

The *interleaving concurrency* is expressed by a formula of the form $\langle r_1\rangle\langle r_2\rangle \mathit{true} \wedge \langle r_2\rangle\langle r_1\rangle \mathit{true} \wedge [\{r_1, r_2\}]\mathit{false}$, and the *true concurrency* is expressed by a formula of the form $\langle r_1\rangle\langle r_2\rangle \mathit{true} \wedge \langle r_2\rangle\langle r_1\rangle \mathit{true} \wedge \langle\{r_1, r_2\}\rangle \mathit{true}$ It is worth noting that the concurrency of the membrane systems is different from that of standard models of the true concurrency. For instance, the formula $\langle\{r_1, r_2\}\rangle \mathit{true} \implies \langle r_1\rangle\langle r_2\rangle \mathit{true}$ is valid for distributed labeled transition systems [11] and unsatisfiable for membranes having mpr as the control mechanism.

However, the modal language was designated as minimal with respect to the concurrency of the membrane systems. Besides the concurrency degree, there is other information which can be of interest regarding the current state: the structure of the system (relationships between regions), explicit description of the membership of a content to its own region etc. The description of the whole behavior of a membrane system can be obtained by enriching the modal language with state-based atomic formulas like $\mathit{childOf}(M, M')$, $\mathit{parentOf}(M, M')$, $\mathit{skin}(M)$, $\mathit{contentOf}(w, M')$, and so on. Using the state-based formulas, we can express the requirement as the communication is possible only between parent-child membranes: $\Pi, (\mu, \overline{w}) \models \langle\{\ldots \mathit{in}(M, M') \ldots\}\rangle \mathit{true}$ implies $\Pi, (\mu, \overline{w}) \models \mathit{parentOf}(M, M')$. Although their study could be interesting, these kind of formulas is out of the goal of this paper.

4 Strategy-based Rewrite Logic for Membrane Systems

Strategy-based rewrite logic for membrane systems was defined in [4] and specifies a membrane system Π by a triple $(\mathcal{R}_\Pi, \mathsf{STRAT}_\Pi, \mathsf{SCTRL}_\Pi)$, where $\mathcal{R}_\Pi$ is a rewrite theory that specifies the control-free system Π and defined as in Section 2.2, STRAT_Π specifies a strategy language for Π, and SCTRL_Π defines the strategy controllers for Π. A strategy controller is intended to equationally define the control mechanisms of Π. The rewrite strategies are used to guide the rewriting according to the operational semantics of Π. We briefly describe here the last two theories.

The Equational Theory STRAT_Π. We consider a minimal strategy language able to express the computations of a membrane system.

STRAT_Π defines the syntax for strategies and consisting of:

- a sort *RuleLabel* for representing rules, together with a membership axiom $r : \mathit{RuleLabel}$, for each rule $r : u \to v$ in R,
- a sort *Strategy* for strategies, and the subsort relation $\mathit{RuleLabel} < \mathit{Strategy}$,
- the strategy constructors for identity, failure, non-deterministic choice, and sequential composition respectively

$$\begin{aligned} \mathit{id} \;\; \mathit{fail} &: \longrightarrow \mathit{Strategy} \\ _ + _ &: \mathit{Strategy}\ \mathit{Strategy} \longrightarrow \mathit{Strategy}\ [\texttt{assoc comm}] \\ _;_ &: \mathit{Strategy}\ \mathit{Strategy} \longrightarrow \mathit{Strategy}\ [\texttt{assoc id}: \ \mathit{id}] \end{aligned}$$

- a congruence strategy operator for each of the constructors of *Soup*, *Membrane* and *MembraneBag*:

$$_\,_ : \mathit{Strategy}\ \mathit{Strategy} \longrightarrow \mathit{Strategy}\ [\texttt{assoc comm}]$$

$$\langle _|_ \rangle : \mathit{MembraneName}\ \mathit{Strategy} \rightarrow \mathit{Strategy}$$
$$\langle _|_\{_\} \rangle : \mathit{MembraneName}\ \mathit{Strategy}\ \mathit{Strategy} \longrightarrow \mathit{Strategy}$$
$$_,_ : \mathit{Strategy}\ \mathit{Strategy} \longrightarrow \mathit{Strategy}\ [\texttt{assoc comm}]$$

The strategy language defined by the above theory was designed having in mind mainly the control of the evolution rules. This language can be enriched with new constructs needed for defining other control mechanisms over evolution rules [2] or to add certain control over the interaction rules. In Section 5 we sketch out an extension for the case of cooperation rules.

The Equational Theory SCTRL_Π. SCTRL_Π is the MEL theory consisting of:

- a sort *StrategyController* - for *strategy controllers*, together with the constants *mpr, pri,...* – corresponding to membrane controllers, *rew* – corresponding to rewrite actions, *comm* – corresponding to communication actions, and *diss* – corresponding to dissolving actions,
- an operation $\mathit{getCtrl} : \mathit{MembraneName} \longrightarrow \mathit{StrategyController}$ which returns the constant corresponding to the control mechanism of the membrane.

The proof-theoretical semantics of the specification $(\mathcal{R}_\Pi, \mathsf{STRAT}_\Pi, \mathsf{SCTRL}_\Pi)$ is given by a MEL-theory $\mathit{Proof}(\Pi)$ which includes the semantics for strategies and the semantics for strategy controllers.
The semantics of strategies can be defined in different ways. In this paper we consider the set-theoretical semantics [13] defined by the following additional operations:

$$[\![_@_]\!] : \mathit{Strategy}\ \mathit{State} \longrightarrow \mathit{Set}\{\mathit{States}\}$$
$$[\![_@_]\!] : \mathit{Strategy}\ \mathit{Set}\{\mathit{States}\} \longrightarrow \mathit{Set}\{\mathit{States}\}$$

together with the following equations:

$$[\![id@t]\!] = t \qquad [\![fail@t]\!] = \emptyset$$
$$[\![r@t]\!] = \{t' \mid t \text{ rewritten directly modulo } E_p \text{ to } t' \text{ using the rule } r \text{ at top}\}$$
$$[\![s_1 + s_2@t]\!] = [\![s_1@t]\!] \cup [\![s_2@t]\!]$$
$$[\![s_1; s_2@t]\!] = [\![s_2@[\![s_1@t]\!]]\!]$$
$$[\![s_1 s_2@w_1 w_2]\!] = \{w'_1 w'_2 \mid w'_i \in [\![s_i@w_i]\!], i = 1, 2\}$$
$$[\![\langle M \mid s\rangle@\langle M \mid w\rangle]\!] = \{\langle M \mid w'\rangle \mid w' \in [\![s@w]\!]\}$$
$$[\![s_1, s_2@t_1, t_2]\!] = \{t'_1, t'_2 \mid t'_i \in [\![s_i@t_i]\!], i = 1, 2\}$$
$$[\![\langle M \mid s_1\{s_2\}\rangle@\langle M \mid w\{t\}\rangle]\!] = \{\langle M \mid w'\{t'\}\rangle \mid w' \in [\![s_1@w]\!], t' \in [\![s_2@t]\!]\}$$
$$[\![s@\emptyset]\!] = \emptyset \qquad [\![s@(T \cup \{t\})]\!] = [\![s@T]\!] \cup [\![s@t]\!]$$

where *State* is a supersort of *HotSoup*, *Membrane* and *MembraneBag*, w_i, w'_i are variables of sort *Soup*, t, t', t_i, t'_i are variables of sort *State*, T a variable of sort $\mathit{Set}\{\mathit{States}\}$, and s, s_i are variables of sort *Strategy*.

Definition 4.1 We say that two strategy terms s_1 and s_2 are equivalent in $\mathit{Proof}(\Pi)$, written $s_1 \equiv s_2$, if and only if $\mathit{Proof}(\Pi) \vdash [\![s_1@t]\!] = [\![s_2@t]\!]$ for all state terms t.

Proposition 4.2 *The following equivalences are true in* $Proof(\Pi)$*:*

$$\begin{array}{ll}
s + s \equiv s & id\ id \equiv id \\
s; fail \equiv fail; s \equiv fail & s + fail \equiv s \\
\langle M \mid s + s' \rangle \equiv \langle M \mid s \rangle + \langle M \mid s' \rangle & s\, fail \equiv fail\, s \equiv fail \\
\langle M \mid s_1 + s_1'\{s_2\} \rangle \equiv \langle M \mid s_1\{s_2\} \rangle + \langle M \mid s_1'\{s_2\} \rangle & s_1\,(s_2 + s_2') \equiv s_1\, s_2 + s_1\, s_2' \\
\langle M \mid s_1\{s_2 + s_2'\} \rangle \equiv \langle M \mid s_1\{s_2\} \rangle + \langle M \mid s_1\{s_2'\} \rangle & s_1, (s_2 + s_2') \equiv s_1, s_2 + s_1, s_2'
\end{array}$$

The semantics of the strategy controllers is given by means of a partial function

$$getStrat : StrategyController\ State \longrightarrow Strategy.$$

together with
– a set of equations defining $getStrat(ctrl, w)$ for each strategy controller constant $ctrl$ corresponding to a control mechanism (like mpr), where w is a variable of sort *Soup*, and
– a set of equations defining $getStrat(ctrl, t)$ for each strategy controller constant $ctrl \in \{rew, comm, diss\}$.
A transition (evolution step) is defined by $t \Rightarrow_{rew} t_1 \Rightarrow_{comm} t_2 \Rightarrow_{diss} t_3 = t'$ iff $Proof(\Pi) \vdash t_1 \in [\![getStrat(rew, t)@t]\!]$, $Proof(\Pi) \vdash t_2 \in [\![getStrat(comm, t_1)@t_1]\!]$, and $Proof(\Pi) \vdash t' \in [\![getStrat(diss, t_2)@t]\!]$.

The following result is a direct consequence of definition of $getStrat(ctrl, w)$ (see [4] for more details).

Proposition 4.3 *If w is a term of sort Soup, ctrl a membrane control mechanism, and $Proof(\Pi) \vdash getStrat(ctrl, w) = s$ with $s \neq fail$, then s is equivalent to a sum of strategy terms s_i of the form $r_{i_1} \dots r_{i_n} id$, where $r_{i_1} \dots r_{i_n}$ is a multiset of rule labels. Moreover, $[\![s_i@w]\!] \neq \emptyset$.*

5 Concurrency in Strategy-based Rewrite Semantics

Let Π be a membrane system and $\mathcal{SR}_\Pi = (\mathcal{R}_\Pi, \mathsf{STRAT}_\Pi, \mathsf{SCTRL}_\Pi)$ its representation as strategy-based rewrite theory. The static relationship between Π and $\mathcal{SR}_\Pi$ is given by a function ψ defined as follows:

(i) if w is a multiset of objects, then $\psi(w)$ is the corresponding ground term of sort *Soup*, denoted also by w (recall that each object of Π is defined as a constant of sort *Soup* in (Σ_p, E_p));
(ii) if M is an elementary membrane with the content w, then $\psi(M) = \langle M \mid w \rangle$;
(iii) if M is a compound membrane with the content w and the children $M_1, \dots, M_n$, then $\psi(M) = \langle M \mid w\,\{\psi(M_1), \dots, \psi(M_n)\} \rangle$;
(iv) $\psi(\mu, \overline{w}) = \psi(M)$, where M is the skin of Π (the relationship between $\overline{w}$ and $\psi(M)$ is implicitly given here).

The operational semantics correspondence is given by

$$\begin{aligned}
(\mu, \overline{w}) \Rightarrow_{ctrl} (\mu', \overline{w}') &\text{ iff } \psi(\mu, \overline{w}) \Rightarrow_{ctrl} \psi(\mu', \overline{w}') \\
&\text{ iff } Proof(\Pi) \vdash \psi(\mu', \overline{w}') \in [\![getStrat(ctrl, \psi(\mu, \overline{w}))@\psi(\mu, \overline{w})]\!]
\end{aligned}$$

where $ctrl \in \{rew, comm, diss\}$.

We associate a modal formula $\psi'(s)$ to a strategy term s as follows:

(i) $\psi'(id) = true$, $\psi'(fail) = false$;
(ii) $\psi'(r) = \langle r\rangle true$ if r is the label of a evolution/comunication/structural rewrite rule;
(iii) $\psi'(s_1 + s_2) = \psi'(s_1) \wedge \psi'(s_2)$;
(iv) $\psi'(s_1; s_2) = \langle A_1\rangle\varphi_2$ if $\psi'(s_1) = \langle A_1\rangle true$ and $\psi'(s_2) = \varphi_2$;
(v) $\psi'(s_1 s_2) = \langle A_1 \cup A_2\rangle(\varphi_1 \wedge \varphi_2)$ if $\psi'(s_i) = \langle A_i\rangle\varphi_i$ for $i = 1, 2$;
(vi) $\psi'(\langle M \mid s\rangle) = \psi'(s)$;
(vii) $\psi'(\langle M \mid s_1 \{s_2\}\rangle) = \langle A_1 \cup A_2\rangle(\varphi_1 \wedge \varphi_2)$ if $\psi'(s_i) = \langle A_i\rangle\varphi_i$ for $i = 1, 2$;
(viii) $\psi'(s_1, s_2) = \langle A_1 \cup A_2\rangle(\varphi_1 \wedge \varphi_2)$ if $\psi'(s_i) = \langle A_i\rangle\varphi_i$ for $i = 1, 2$.

The function ψ' is partial; for instance, $\psi'(s_1; s_2)$ is not defined for all s_1. Moreover, in order to have defined formulas like $\psi'(s\, id)$ we assume that $\varphi \equiv \langle\emptyset\rangle\varphi$.

5.1 Concurrency of Rewrite Actions

We have all the elements to compare the concurrency degrees of a membrane system Π and its strategy-based rewrite specification $(\mathcal{R}_\Pi, \mathsf{STRAT}_\Pi, \mathsf{SCTRL}_\Pi)$.

Lemma 5.1 *Let M be an elementary membrane and w its content. If $Proof(\Pi) \vdash getStrat(\langle\ M \mid w\ \rangle) = s$, then $\psi'(s)$ is well-defined and has the form $\langle L_1\rangle true \wedge \ldots \wedge \langle L_k\rangle true$.*

The proof of the above result follows directly from Proposition 4.3.

Lemma 5.2 *Let M be an elementary membrane and w its content. $M, w \models \langle L\rangle true$ if and only if $Proof(\Pi) \vdash getStrat(\langle\ M \mid w\ \rangle) = s$ and there is φ such that $\psi'(s) = \varphi \wedge \langle L\rangle true$.*

Proof. We assume that $L = \{r_1, \ldots, r_n\}$. If $M, w \models \langle L\rangle true$, then $s = s' + r_1 \ldots r_n id$ from the definition of *getStrat*. The converse implication follows by applying Proposition 4.3 and Lemma 5.1. □

Now we are able to prove the first main result of this paper:

Theorem 5.3 *Let L be a multiset of rewrite actions.*
1) If $\Pi, (\mu, \overline{w}) \models \langle L\rangle true$ then there are the strategy terms s, s' such that $\psi'(s') = \langle L\rangle true$ and $Proof(\Pi) \vdash getStrat(rew, \psi(\mu, \overline{w})) = s + s'$.
2) Conversely, if $Proof(\Pi) \vdash getStrat(rew, \psi(\mu, \overline{w})) = s$ then $\psi'(s)$ is well-defined and $\Pi, (\mu, \overline{w}) \models \psi'(s)$.

Proof. If $Proof(\Pi) \vdash getStrat(rew, \psi(\mu, \overline{w})) = s$, then s is a sum of strategies by Proposition 4.2 and each member of the sum defines a transition. The conclusions of the theorem follows by the operational semantics correspondence and Lemma 5.2. □

It is worth noting that if $Proof(\Pi) \vdash getStrat(\psi(\Pi, \overline{w})) = s$ and s_i is a sum member of s, then s_i can be implemented by a one-step concurrent $\mathcal{R}_\Pi$-rewrite. Therefore Theorem 5.3 says that *the maximal concurrency of the rewrite actions is preserved by the strategy-based rewrite semantics.*

5.2 Concurrency Of Communication Actions

Let us consider first a parent-child pair (M, M') of membranes. The communication between M and M' can be implemented either by interleaving the *in* and *out* rules, $in(M, M'); out(M', M) + out(M', M); in(M, M')$, or by using directly $in\text{-}out(M, M')$. Neither of the two options faithfully preserves the maximal concurrency of the membrane systems. However, the later option seems to be a better solution if we define $\psi'(in\text{-}out(M, M')) = \langle\{in(M, M'), out(M', M)\}\rangle true$. It is worth noting that for the case of interleaving concurrency we have $in(M, M'); out(M', M) \equiv out(M', M); in(M, M')$.

Let Π be a system with the structure $[_1[_2[_3]_3]_2]_1$. Then the interleaving concurrency cannot be avoided by the rewrite semantics even if the in-out rewrite rules are used. In fact, the best we can obtain is the strategy $in\text{-}out(M_1, M_2); in\text{-}out(M_2, M_3)+ in\text{-}out(M_2, M_3); in\text{-}out(M_1, M_2)$ (or an equivalent one if at least one communication is in only one direction).

However, the communications can happen concurrently at disjoint positions in the structure tree. If Π has the structure $[_1[_2[_3]_3]_2][_4[_5]_5]_4]_1$, then the communication can be described by the strategy

$\langle\ M_1 \mid id\ \{\ in\text{-}out(M_2, M_3), in\text{-}out(M_4, M_5)\ \}\ \rangle; in\text{-}out(M_1, M_2); in\text{-}out(M_1, M_4)$

or an equivalent one.

The *maximum concurrency degree of the communication which can be described in the rewrite semantics is that given by rewriting logic*. This is because the communication rewrite rules are global and not local as it is the case of evolution rules. In order to capture this concurrency degree in the strategy-based rewrite semantics, we have to add to the strategy language an operator

$$ca : Strategy\ Strategy \longrightarrow Strategy$$

for each communication action $ca \in \{in(M, M'), out(M, M'), in\text{-}out(M, M')\}$. A strategy expression $ca(s, s')$ means that s deals with the communications inside M, s' deals with the communications inside M', ca with the communication between M and M'. Here is an example:

$$in\text{-}out(M_1, M_2)(in\text{-}out(M_3, M_4), in\text{-}out(M_5, M_6))$$

where the structure is $[_1[_2]_2[_3[_4]_4]_3[_5[_6]_6]_5]_1$. These strategies are obtained by applying the unconditional replace axiom in the definition of RL. The definition of ψ' is extended by setting $\psi'(ca(s, s')) = \langle\{ca\} \cup C \cup C'\rangle(\varphi \wedge \varphi')$, where $\psi'(s) = \langle C\rangle\varphi$ and $\psi'(s') = \langle C'\rangle\varphi'$. Then, the strategy controller *comm* together with its semantics will compute a strategy with maximum concurrency degree.

We conclude now the second main result, namely that *the true concurrency of the communication actions is partially preserved by the strategy-based rewrite semantics, i.e., it is described by a combination of true concurrency and interleaving concurrency.* This is formalized by the following result:

Theorem 5.4 *Let C be a set of communication actions. Then $\Pi, (\mu, \overline{w}) \models \langle C\rangle true$ if and only if there is a strategy term s, a partition $C_1 \uplus \ldots \uplus C_k$ of C, and a bijection $f : \{1, \ldots, k\} \longrightarrow \{1, \ldots, k\}$ such that $Proof(\Pi) \vdash getStrat(comm, \psi(\mu, \overline{w})) = s$ and $\psi'(s) = \langle C_{f(1)}\rangle \ldots \langle C_{f(k)}\rangle true$.*

5.3 Concurrency Of Structural Actions

If the structural actions include only dissolvings, then we get a similar conclusion to that one for the communication actions. However, since the structural actions change the structure of the system, the maximal concurrency of structural actions cannot always be described by interleaving concurrency in the strategy-based rewrite semantics. For instance, the double-dissolving given by

$$\Pi, ([_1[_2[_3]_3]_2]_1, w_1, w_2\delta, w_3\delta) \models \langle\{\mathit{diss}(M_2, M_1), \mathit{diss}(M_3, M_2)\}\rangle \mathit{true}$$

is described either by a strategy s with $\psi'(s) = \langle \mathit{diss}(M_2, M_1)\rangle\langle \mathit{diss}(M_3, M_1)\rangle \mathit{true}$ or by a strategy s' with $\psi'(s') = \langle \mathit{diss}(M_3, M_2)\rangle\langle \mathit{diss}(M_2, M_1)\rangle \mathit{true}$. Obviously $s \equiv s'$ but this equivalence does not define a interleaving concurrency. Therefore we have a weaker result:

Theorem 5.5 *Let D be a set of structural actions. Then $\Pi, (\mu, \overline{w}) \models \langle D\rangle \mathit{true}$ if and only if there is a strategy term s and a partition $D_1 \uplus \ldots \uplus D_k$ of D such that $\mathit{Proof}(\Pi) \vdash \mathit{getStrat}(\mathit{diss}, \psi(\mu, \overline{w})) = s$ and $\psi'(s) = \langle D_1\rangle \ldots \langle D_k\rangle \mathit{true}$.*

6 Conclusion

In this paper we give a partial answer to the question if it is possible to define a rewrite semantics for the membrane systems. It was recently shown [12] that rewriting logic-based semantics cannot preserve the maximal concurrency of the rewrite actions. The main reason is the locality of the evolution rules. The rewrite rules encoding the evolution rules belonging to a region must share this locality and hence they cannot be applied concurrently.

In this paper we show that the strategy-based rewrite semantics introduced in [4] preserves the maximal concurrency of the rewrite actions. In the strategy-based rewrite semantics the control mechanisms of the membranes are modeled by strategy controllers. A strategy controller analyzes the current state and computes a strategy term describing all possible transitions from the current state. The strategy term corresponding to a membrane can be computed in such a way it preserves the concurrency degree given by the control mechanism. The semantics of the strategy controllers is equationally defined and therefore it does not affect the behavior described by the strategy-based theory.

Regarding the concurrency of the cooperation (communication and structural) actions, the two rewrite semantics are equivalent. Since the rewrite rules governing these actions are global, the concurrency given by the rewriting logic is the maximum we can obtain.

In a recent paper, Şerbănuţă et al. [19] show that the framework K [18] is suitable to faithfully describe the behavior of the P systems. Their result is based on the following two facts:
1) a special encoding of the P systems by tagging the rewrite rules and the objects with the path in the structure-tree from the root to the membrane M, and
2) the rewriting rules in K can be applied concurrently even when they overlap, assuming that they do not change the overlapped portion of the term (may overlap on "read only" parts).

In this paper we consider the particular case of transition P systems [17]. There is a large variety of P systems. We think that the strategy-based rewrite semantics can faithfully describe almost all mechanisms used for controlling the evolution rules. It remains to investigate what happens with the concurrency degree of the cooperation actions for different more general structures, e.g., tissue-like structures or neural-like structures.

References

[1] Agrigoroaiei, O. and G. Ciobanu, *Rewriting Logic Specification of Membrane Systems with Promoters and Inhibitors*, in: *Proceedings of WRLA*, 2008, pp. 1–16, to appear in ENTCS.

[2] Andrei, O., G. Ciobanu and D. Lucanu, *Expressing Control Mechanisms in P systems by Rewriting Strategies*, in: H. J. Hoogeboom, G. Paun, G. Rozenberg and A. Salomaa, editors, *Workshop on Membrane Computing*, Lecture Notes in Computer Science **4361** (2006), pp. 154–169.

[3] Andrei, O., G. Ciobanu and D. Lucanu, *A rewriting logic framework for operational semantics of membrane systems*, Theoretical Computer Science **373** (2007), pp. 163 – 181.

[4] Andrei, O. and D. Lucanu, *Strategy-Based Proof Calculus for Membrane Systems*, in: *Proceedings of WRLA*, 2008, pp. 17–34, to appear in ENTCS.

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

[6] Bezem, M., J. Klop and R. de Vrijer, editors, "Term Rewriting Systems." Cambridge Tracts in Theoretical Computer Science **I**, Cambridge University Press, 2003.

[7] Bouhoula, A., J.-P. Jouannaud and J. Meseguer, *Specification and proof in membership equational logic*, Theoretical Computer Science **236** (2000), pp. 35–132.

[8] Bruni, R. and J. Meseguer, *Semantic foundations for generalized rewrite theories*, Theoretical Computer Science **360** (2006), pp. 386–414.

[9] Goguen, J. A. and J. Meseguer, *Order-Sorted Algebra I: Equational Deduction for Multiple Inheritance, Overloading, Exceptions and Partial Operations*, Theoretical Computer Science **105** (1992), pp. 217–273.

[10] Hennessy, M. and R. Milner, *Algebraic laws for nondeterminism and concurrency*, J. of ACM **32** (1985), pp. 137–161.

[11] Lodaya, K., M. Mukund, R. Ramanujam and P. Thiagarajan, *Models and logics for true concurrency*, in: P. Thiagarajan, editor, *Some Models and Logics for Concurrency*, Advanced School on the Algebraic, Logical and Categorical Foundations of Concurrency, Gargnano del Garda, 1991 .

[12] Lucanu, D., *Rewriting Logic-based Semantics of Membrane Systems and the Maximal Concurrency*, in: *Proceedings of Prague International Workshop on Membrane Computing* (2008), pp. 23–34.

[13] Martí-Oliet, N., J. Meseguer and A. Verdejo, *A Rewriting Semantics for Maude Strategies*, in: *Proceedings of WRLA*, 2008, pp. 207–226, to appear in ENTCS.

[14] Meseguer, J., *Conditional Rewriting Logic as a Unified Model of Concurrency*, Theoretical Computer Science **96** (1992), pp. 73–155.

[15] Meseguer, J., *Rewriting Logic as a Semantic Framework for Concurrency: a Progress Report*, in: *Proceedings of the 7th International Conference on Concurrency Theory*, Lecture Notes in Computer Science **1119**, 1996, pp. 331–372.

[16] Meseguer, J., *Membership algebra as a logical framework for equational specification*, in: Francesco Parisi-Presicce, editor, *WADT*, Lecture Notes in Computer Science **1376** (1997), pp. 18–61.

[17] Paun, G., "Membrane Computing. An Introduction," Springer, 2002.

[18] Roşu, G., *K: A Rewriting-Based Framework for Computations – Preliminary version*, Technical Report Department of Computer Science UIUCDCS-R-2007-2926 and College of Engineering UILU-ENG-2007-1827, University of Illinois at Urbana-Champaign (2007).

[19] Şerbănuţă, T. F., G. Ştefănescu and G. Roşu, *Defining and executing p-systems with structured data in k*, in: *WMC'08*, 2008, to appear.

Recognizing Strategies

Bastiaan Heeren[1,2]

School of Computer Science, Open Universiteit Nederland
P.O.Box 2960, 6401 DL Heerlen, The Netherlands

Johan Jeuring[3]

School of Computer Science, Open Universiteit Nederland and ICS, Universiteit Utrecht

Abstract

We use strategies to specify how a wide range of exercises can be solved incrementally, such as bringing a logic proposition to disjunctive normal form, reducing a matrix, or calculating with fractions. With such a strategy, we can automatically generate worked-out solutions, track the progress of a student by inspecting submitted intermediate answers, and report back suggestions in case the student deviates from the strategy. Because we can calculate all kinds of feedback automatically from a strategy specification, it becomes less labor-intensive and less ad-hoc to specify new exercise domains and exercises within that domain.
A strategy describes valid sequences of transformation rules that solve the exercise at hand, which turns tracking intermediate steps into a parsing problem. This is a promising view at the problem because it allows us to take advantage of many years of experience in parsing sentences of context-free languages, and transfer this knowledge and technology to the domain of stepwise solving exercises.
In this paper we work out the similarities between parsing and solving exercises incrementally, and we discuss the implementation of a recognizer for strategies. We present a full implementation of such a recognizer, and discuss a number of design choices we have made. In particular, we discuss the use of a fixed point combinator to deal with repetition, and labels to mark positions in the strategy.

Keywords: grammars, parsing, strategies, exercise assistants, combinator languages

1 Introduction

Strategies are used in many domains such as programming, rewriting, compiler construction, and theorem proving. We recently realized that strategies also play an important role in exercise assistants that support incrementally solving exercises in mathematics, logics, physics, etc. [6]. In the intelligent tutoring systems field, a strategy is called procedural knowledge, a production system, or a procedural plan. In this field, strategies are not used to rewrite terms, but they are used to check that a user performs the correct steps towards a solution for an exercise.

[1] This work was made possible by the support of the SURF Foundation, the higher education and research partnership organisation for Information and Communications Technology (ICT). For more information about SURF, please visit http://www.surf.nl. We thank Alex Gerdes and the anonymous reviewers for their constructive comments.

[2] Email: bastiaan.heeren@ou.nl

[3] Email: johanj@cs.uu.nl

An exercise such as "rewrite the following arithmetic expression containing fractions to its normal form", consists of the expression to be rewritten, the rules with which the expression can be rewritten (and possibly also some known buggy rules), and a strategy to guide or direct the rewriting. If a user solves the exercise incrementally, we can check at each step whether or not the step is a valid rewrite step, and whether or not this rewrite step is valid according to the strategy. In a way we check whether or not the sequence of rewrite steps performed by the user is a prefix of a sentence in the language specified by the strategy.

This paper briefly explains a language for specifying strategies for exercises, and it shows how we use this language for recognizing valid sequences of rewrite steps. The paper has two contributions:

- It discusses the design choices when constructing a strategy language for specifying exercises, and a strategy recognizer for such a language.
- It shows how we can use the strategy language to check whether or not user-input is correct with respect to the strategy specified for an exercise.

The information provided by our strategy recognizer is necessary for determining what kind of feedback to give to a user of an exercise assistant. At the moment the feedback provided by exercise assistants is almost always limited to correct/incorrect. Using the diagnosis given by our strategy recognizer we can improve a lot on this.

This paper is organized as follows. Section 2 introduces our language for specifying strategies for exercises. We illustrate the language with a strategy for adding fractions: this strategy is used as a running example throughout the paper. Section 3 shows how we have implemented the components of our strategy language to obtain a strategy recognizer, and discusses the main design choices. We then present three extensions to our strategy recognizer in Section 4. Section 5 shows how the strategy language can be used for diagnosing possible problems in the user input. The last two sections (6 and 7) discuss related work and ongoing research, and draw conclusions.

2 A strategy language

Before we introduce our strategy language, which is inspired by context-free grammars (CFG), we give an example strategy.

Example 2.1 Consider the problem of adding two fractions, for example, $\frac{2}{5}$ and $\frac{2}{3}$: if the result is an improper fraction (the numerator is larger than or equal to the denominator), then it should be converted to a mixed number. Figure 1 displays four rewrite rules on fractions. The three rules on the right (B1 to B3) are buggy rules that capture common mistakes. A possible strategy to solve this type of exercise is the following:

- *Step 1.* Find the least common denominator (lcd) of the fractions: let this be n
- *Step 2.* Rename the fractions such that n is the denominator
- *Step 3.* Add the fractions by adding the numerators
- *Step 4.* Simplify the fraction if it is improper

$$\frac{a}{c} + \frac{b}{c} = \frac{a+b}{c} \quad \text{[ADD]}$$
$$\frac{a}{b} \times \frac{c}{d} = \frac{a \times c}{b \times d} \quad \text{[MUL]}$$
$$\frac{b}{c} = \frac{a \times b}{a \times c} \quad \text{[RENAME]}$$
$$\frac{a+b}{b} = 1 + \frac{a}{b} \quad \text{[SIMPL]}$$

$$\frac{a}{b} + \frac{c}{d} \neq \frac{a+c}{b+d} \quad \text{[B1]}$$
$$a \times \frac{b}{c} \neq \frac{a \times b}{a \times c} \quad \text{[B2]}$$
$$a + \frac{b}{c} \neq \frac{a+b}{c} \quad \text{[B3]}$$

Fig. 1. Transformation rules in the domain of fractions

A context-free grammar distinguishes terminal and non-terminal symbols, and has a set of production rules. For our strategy language, we take a different approach and use combinators instead. Strategies are constructed in the following way:

- **Transformation rule.** Such a rule is the smallest building block to construct composite strategies, and closely corresponds to a terminal symbol in a CFG. Occasionally, we write *symbol* r for some transformation rule r to distinguish the strategy from the transformation rule.
- **Sequence.** We can combine strategies s and t and put them in sequence, for which we write $s \lessdot\!\star\!\gtrdot t$. A production rule in a CFG is a sequence of symbols.
- **Choice.** Another way to combine strategies is by choice: we write $s <\!|\!> t$ for choosing between strategy s and strategy t. In CFGs, choice is introduced by having multiple production rules for a non-terminal symbol.
- **Unit elements.** We introduce two special elements that are the units for sequence and choice. The strategy *succeed* always succeeds (unit element for $\lessdot\!\star\!\gtrdot$), whereas *fail* always fails (unit element for $<\!|\!>$).
- **Labels.** Because our primary interest in the strategy language is to automatically calculate feedback from it, we need some mechanism to mark positions in the strategy, for example, to encode the hierarchical structure of a strategy, or to refine the textual feedback that is associated with a certain position in the strategy. For this purpose, we introduce labels. Labeling a strategy s with some label ℓ is written as *label* $\ell\ s$. The exact representation of a label is irrelevant.
- **Recursion.** We need a way to deal with recursion, and for this we introduce a fixed point combinator. We write *fix* f, where f is the function of which we take the fixed point. Hence, the function f takes a strategy and returns one, and such that the property $\mathit{fix}\ f = f\ (\mathit{fix}\ f)$ holds.

Example 2.2 Repetition, zero or more occurrences of something, is a well-known recursion pattern. We can define this pattern using our fixed point recursion combinator:

$$\mathit{many}\ s = \mathit{fix}\ (\lambda x \rightarrow \mathit{succeed} <\!|\!> (s \lessdot\!\star\!\gtrdot x))$$

The strategy that applies transformation rule r zero or more times would thus be:

$$\begin{aligned}
&\mathit{many}\ (\mathit{symbol}\ r) \\
&\quad = \mathit{succeed} <\!|\!> (\mathit{symbol}\ r \lessdot\!\star\!\gtrdot \mathit{many}\ (\mathit{symbol}\ r)) \\
&\quad = \mathit{succeed} <\!|\!> (\mathit{symbol}\ r \lessdot\!\star\!\gtrdot (\mathit{succeed} <\!|\!> (\mathit{symbol}\ r \lessdot\!\star\!\gtrdot \mathit{many}\ (\mathit{symbol}\ r)))) \\
&\quad = \ldots
\end{aligned}$$

Example 2.3 We use the strategy combinators to turn the informal strategy description from Example 2.1 into a strategy specification:

$$\begin{array}{lll} \mathit{addFractions} = \mathit{label}\ \ell_0 & (& \mathit{label}\ \ell_1\ \mathit{ruleLCD} \\ & \lessdot\!\star\!\gtrdot & \mathit{label}\ \ell_2\ (\mathit{repeat}\ (\mathit{somewhere}\ \mathit{ruleRename})) \\ & \lessdot\!\star\!\gtrdot & \mathit{label}\ \ell_3\ \mathit{ruleAdd} \\ & \lessdot\!\star\!\gtrdot & \mathit{label}\ \ell_4\ (\mathit{try}\ \mathit{ruleSimpl}) \\ &) & \end{array}$$

The strategy contains the labels ℓ_0 to ℓ_4, and uses the transformation rules given in Figure 1. The transformation *ruleLCD* is somewhat different: it does not change the term, but it calculates the least common denominator and stores this in an environment. The rule RENAME for renaming a fraction uses the computed lcd to determine the value of a in its right-hand side. Rules that do not change the term but only the context in which an exercise is solved are so-called *administrative rules*.

The definition of *addFractions* contains the strategy combinators *repeat*, *somewhere*, and *try*. In an earlier paper [6], we discussed how these combinators, and many others, can be defined conveniently in terms of the strategy language. The combinator *repeat* is a variant of the *many* combinator: it applies its argument strategy exhaustively. The check that the strategy can no longer be applied is an administrative rule. The definition of *somewhere* is another example of an administrative rule: this combinator changes the focus in the abstract syntax tree before it applies its argument strategy. The zipper data structure [8] can be used to keep a point of focus.

2.1 Semantics of the strategy language

Before we move on to the implementation, we make our strategy combinators more precise by defining the language that is generated by a strategy. Such a language is a set of sequences of transformation rules.

$$\begin{array}{ll} \mathit{language}\ (s \lessdot\!\star\!\gtrdot t) & = \{\, xy \mid x \in \mathit{language}\ s, y \in \mathit{language}\ t \,\} \\ \mathit{language}\ (s <\!|\!> t) & = \mathit{language}\ s \cup \mathit{language}\ t \\ \mathit{language}\ (\mathit{fix}\ f) & = \mathit{language}\ (f\ (\mathit{fix}\ f)) \\ \mathit{language}\ (\mathit{label}\ \ell\ s) & = \mathit{language}\ s \\ \mathit{language}\ (\mathit{symbol}\ r) & = \{r\} \\ \mathit{language}\ \mathit{succeed} & = \{\epsilon\} \\ \mathit{language}\ \mathit{fail} & = \emptyset \end{array}$$

This definition tells us whether a sequence of rules follows a strategy or not: the sequence of rules should be a sentence in the language generated by the strategy, or a prefix of a sentence since we solve exercises incrementally. Not all sequences make sense, however. An exercise gives us an initial term (say t_0), and we are only interested in sequences of rules that can be applied successively to this term. Suppose that we have terms (denoted by t_i) and rules (denoted by r_i), and let t_{i+1} be the result of applying rule r_i to term t_i. A possible derivation that starts with t_0 can be depicted in the following way:

$$t_0 \xrightarrow{r_0} t_1 \xrightarrow{r_1} t_2 \xrightarrow{r_2} t_3 \xrightarrow{r_3} \ldots$$

To be precise, applying a rule to a term can yield multiple results, but most domain rules, such as the rules for fractions in Figure 1, return at most one term. Running a strategy with an initial term returns a set of terms, and is specified by:

$$\mathit{run}\ s\ t_0 = \{\ t_{n+1} \mid r_0 \mathinner{..} r_n \in \mathit{language}\ s, \forall_{i \in 0 \ldots n} : t_{i+1} \in \mathit{apply}\ r_i\ t_i\ \}$$

Recognizing a strategy comes down to tracing the steps that a student is taking, but how would a tool get the sequence of rules? In exercise assistants that offer free input, users submit intermediate terms. Therefore, the tool first has to determine which of the known rules has been applied, or even which combination of rules has been used. Discovering which rule has been used is obviously an important part of an exercise assistant, and it influences the quality of the generated feedback. It is, however, not the topic of this paper. An alternative to free input is to let users select a rule, which is then applied automatically to the current term. In this setup, it is no longer a problem to detect which rule has been used.

3 Design of a strategy recognizer

In this section we discuss the design of a strategy recognizer. Instead of designing our own recognizer, we could reuse existing parsing libraries and tools. There are many excellent parser generators and various parser combinator libraries around [9,13], and these are often highly optimized and efficient in both their time and space behavior. However, the problem we are facing is quite different from other parsing applications. To start with, efficiency is no longer a key concern. Because we are recognizing applications of rewrite rules applied by a student, the length of the input is very limited. Our experience until now is that speed poses no serious constraints on the design of the library. A second difference is that we are not building an abstract syntax tree.

The following issues are important for a strategy recognizer, but are not (sufficiently) addressed in traditional parsing libraries:

(i) We are only interested in sequences of transformation rules that can be applied successively to some initial term, and this is hard to express in most libraries. Parsing approaches that start by analyzing the grammar for constructing a parsing table will not work in our setting because they can not take the current term into account.

(ii) The ability to diagnose errors in the input highly influences the quality of the feedback services. It is not enough to detect that the input is incorrect, but we also want to know at which point the input deviates from the strategy, and what is expected at this point. Some of the more advanced parser tools have error correcting facilities, which helps diagnosing an error to some extent.

(iii) Exercises are solved incrementally, and therefore we do not only have to recognize full sentences, but also prefixes. Backtracking and look-ahead can not be used because we want to recognize strategies at each intermediate step.

(iv) Labels help to describe the structure of a strategy in the same way as nonterminals do in a grammar. For a good diagnosis it is vital that a recognizer knows at each intermediate step where it is in the strategy.

(v) A strategy should be serializable, for instance because we want to communicate with other e-learning tools and environments.

In earlier attempts to design a recognizer library for strategies, we tried to reuse an existing error-correcting parser combinator library [13], but failed because of the reasons listed above. The library we develop in this paper is written in the functional programming language Haskell [12]. The code in this paper is almost complete and conforms to the Haskell 98 standard. Although the code is relatively short, we want to emphasize that the library has been tested in practice on different domains. For instance, strategies implemented for the domain of linear algebra are more complex than the strategy for fractions reported in this paper. These strategies will be used in several courses during 2008.

3.1 Representing grammars

Because strategies are grammars, we start by exploring a suitable representation for grammars. The data type for grammars is based on the alternatives of the strategy language discussed in Section 2:

```
data Grammar a = Grammar a :⋆: Grammar a
               | Grammar a :|: Grammar a
               | Rec Int (Grammar a)
               | Symbol a | Var Int | Succeed | Fail deriving Show
```

The type variable *a* in this definition is an abstraction for the type of the symbols: for strategies, the symbols are rules. The first design choice is how to represent recursive grammars, for which we use the constructors *Rec* and *Var*. A *Rec* binds all the *Var*s in its scope that have the same integer. We assume that all our grammars are closed, i.e., there are no free occurrences of variables. This data type makes it easy to manipulate and analyze grammars. Alternative representations for recursion are higher-order fixed point functions, or nameless terms using de Bruijn indices.

Labels are absent and will be added later. Observe that we use the constructors :⋆: and :|: for sequence and choice, respectively (instead of the combinators $<\star>$ and $<|>$ introduced earlier). Haskell infix constructors have to start with a colon, but the real motivation is that we use $<\star>$ and $<|>$ as smart constructors.

3.2 Smart constructors

A smart constructor is a normal function that in addition to constructing a value performs some checks or some simplifications. We use smart constructors for simplifying grammars, and to obtain a normal form. We introduce a smart constructor for every alternative of the *Grammar* data type: the functions *symbol*, *var*, *succeed*, and *fail* do nothing special, but are introduced for consistency.

The smart constructor $<\star>$ for sequences removes the unit element *Succeed*, and propagates the absorbing element *Fail*. Because the input is processed from left to right, we associate sequences to the right. Pay close attention to the occurrences of the smart constructors and the actual constructors in the following definition:

```
(<*>) :: Grammar a → Grammar a → Grammar a
Succeed <*> t       = t
s       <*> Succeed = s
Fail    <*> _       = fail
_       <*> Fail    = fail
(s :*: t) <*> u     = s :*: (t <*> u)
s       <*> t       = s :*: t
```

For choices, we remove occurrences of *Fail*, and we nest alternatives to the right:

```
(<|>) :: Grammar a → Grammar a → Grammar a
Fail    <|> t    = t
s       <|> Fail = s
(s :|: t) <|> u  = s :|: (t <|> u)
s       <|> t    = s :|: t
```

The smart constructor for recursive grammars checks that there is at least one free occurrence of the variable in the body: a *Rec* is built only if this is the case.

```
rec :: Int → Grammar a → Grammar a
rec i s = if i ∈ freeVars s then Rec i s else s
```

Calculating the set of free variables of a grammar is straightforward, although we have to take care of shadowing binders.

Finally, we define a constructor function for fixed points on grammars, which gives us another way to specify recursive grammars:

```
fix :: (Grammar a → Grammar a) → Grammar a
```

This function can be implemented using *rec* and *var*: the only difficulty in defining *fix* is to discover which integer can be used. We omit the implementation details.

3.3 Empty and firsts

For recognizing sentences, we have to define the functions *empty* and *firsts*. The function *empty* tests whether the empty sentence is part of the language.

```
empty :: Grammar a → Bool
empty (s :*: t)  = empty s ∧ empty t
empty (s :|: t)  = empty s ∨ empty t
empty (Rec i s)  = empty s
empty Succeed    = True
empty _          = False
```

The last definition covers the cases for *Fail*, *Symbol*, and *Var*. The most interesting definition is for the pattern (*Rec i s*): it calls *empty* recursively on *s* as there is no need to inspect recursive occurrences.

The function *firsts* returns a list with all symbols that can appear as the first symbol of a sentence. For each symbol, the function also returns the remaining grammar, i.e., the sentences that can appear after that symbol.

$$
\begin{aligned}
&\mathit{firsts} :: \mathit{Grammar}\ a \to [(a, \mathit{Grammar}\ a)]\\
&\mathit{firsts}\ (s :\!\star\!: t) &&= [(a, s' \mathbin{<\!\star\!>} t) \mid (a, s') \leftarrow \mathit{firsts}\ s] \mathbin{+\!\!+}\\
& &&\ (\textbf{if}\ \mathit{empty}\ s\ \textbf{then}\ \mathit{firsts}\ t\ \textbf{else}\ [\,])\\
&\mathit{firsts}\ (s :\!|\!: t) &&= \mathit{firsts}\ s \mathbin{+\!\!+} \mathit{firsts}\ t\\
&\mathit{firsts}\ (\mathit{Rec}\ i\ s) &&= \mathit{firsts}\ (\mathit{replaceVar}\ i\ (\mathit{Rec}\ i\ s)\ s)\\
&\mathit{firsts}\ (\mathit{Symbol}\ a) &&= [(a, \mathit{succeed})]\\
&\mathit{firsts}\ _ &&= [\,]
\end{aligned}
$$

For a sequence $(s :\!\star\!: t)$, we determine which symbols can appear first for s, and we change the results to reflect that t is part of the remaining grammar. Furthermore, if s can be empty, then we also have to look at the *firsts* for t. For choices, we simply combine the results for both operands. If the grammar is a single symbol, then this symbol appears first, and the remaining strategy is *succeed* (we are done). To find the firsts for $(\mathit{Rec}\ i\ s)$, we have to look inside the body s. All occurrences of this recursion point are replaced by the grammar itself before we call *firsts* again. The replacement is performed by a helper-function: $\mathit{replaceVar}\ i\ s\ t$ replaces all free occurrences of $(\mathit{Var}\ i)$ in t by s.

The function *nonempty* removes the empty sentence from a grammar, and is defined using *firsts*:

$$
\begin{aligned}
&\mathit{nonempty} :: \mathit{Grammar}\ a \to \mathit{Grammar}\ a\\
&\mathit{nonempty}\ s = \mathit{foldr}\ (\mathbin{<\!|\!>})\ \mathit{fail}\ [\mathit{symbol}\ a \mathbin{<\!\star\!>} t \mid (a, t) \leftarrow \mathit{firsts}\ s]
\end{aligned}
$$

Example 3.1 The repetition combinator *many* can be defined in the following way:

$$
\begin{aligned}
&\mathit{many} :: \mathit{Grammar}\ a \to \mathit{Grammar}\ a\\
&\mathit{many}\ s = \mathit{rec}\ 0\ (\mathit{succeed} \mathbin{<\!|\!>} (\mathit{nonempty}\ s \mathbin{<\!\star\!>} \mathit{var}\ 0))
\end{aligned}
$$

It can also be expressed using the function *fix*, resulting in the definition given in Example 2.2. We have to apply *nonempty* to strategy s to avoid a left-recursive grammar specification: this also holds when we use *fix*. In Section 4.2 we explain how left recursion can be avoided by analyzing the grammar that is constructed.

3.4 Running a strategy

So far, nothing specific about recognizing strategies has been discussed. A strategy is a grammar over rewrite rules: with the functions *empty* and *firsts* we can run a strategy with an initial term:

$$
\begin{aligned}
&\mathit{run} :: \mathit{Grammar}\ (\mathit{Rule}\ a) \to a \to [a]\\
&\mathit{run}\ s\ a = [a \mid \mathit{empty}\ s] \mathbin{+\!\!+} [c \mid (r, t) \leftarrow \mathit{firsts}\ s, b \leftarrow \mathit{apply}\ r\ a, c \leftarrow \mathit{run}\ t\ b]
\end{aligned}
$$

The list of results returned by *run* consists of two parts: the first part tests whether $\mathit{empty}\ s$ holds, and if so, it yields the singleton list containing the term a. The

second part takes care of the non-empty alternatives. Let r be one of the symbols that can appear first in strategy s (r is a rewrite rule). We are only interested in r if it can be applied to the current term a. It is irrelevant how the type *Rule* is defined, except that applying a rule to a term returns a list of results. We run the remainder of the strategy (that is, t) with the result of the application of rule r.

The function *run* can produce an infinite list. In most cases, however, we are only interested in a single result (and rely on lazy evaluation). The part that considers the empty sentence is put at the front to return sentences with few rewrite rules early. Nonetheless, the definition returns results in a depth-first manner. We define a variant of *run* which exposes breadth-first behavior:

$$
\begin{array}{l}
\mathit{run}' :: \mathit{Grammar}\ (\mathit{Rule}\ a) \to a \to [[a]] \\
\mathit{run}'\ s\ a = [a \mid \mathit{empty}\ s] : \mathit{merge}\ [\mathit{run}'\ t\ b \mid (r, t) \leftarrow \mathit{firsts}\ s, b \leftarrow \mathit{apply}\ r\ a] \\
\quad \textbf{where}\ \mathit{merge} = \mathit{map}\ \mathit{concat} \circ \mathit{transpose}
\end{array}
$$

The function *run′* produces a list of lists: results are grouped by the number of rewrite steps that have been applied, thus making explicit the breadth-first nature of the function. The helper-function *merge* merges the results of the recursive calls: by transposing the list of results, we combine results with the same number of steps.

3.5 Labels

Labels are not included in the *Grammar* data type. We introduce two mutually recursive types for strategies that can have labeled parts:

$$
\begin{array}{ll}
\textbf{data}\ \mathit{LabeledStrategy} & l\ a = \mathit{Label}\ l\ (\mathit{Strategy}\ l\ a) \\
\textbf{type}\ \mathit{Strategy} & l\ a = \mathit{Grammar}\ (\mathit{Either}\ (\mathit{Rule}\ a)\ (\mathit{LabeledStrategy}\ l\ a))
\end{array}
$$

A labeled strategy is a strategy with a label (of type l). A strategy is a grammar where the symbols are either rules or labeled strategies. For this choice, we use the *Either* data type: rules are tagged with the *Left* constructor, labeled strategies are tagged with *Right*. With the type definitions above, we can have grammars over other grammars, and the nesting can be arbitrarily deep.

Excluding labels from the *Grammar* data type is a design choice. Functions that work on the *Grammar* data type don't have to deal with labels, which makes it, for example, simpler to manipulate grammars. A disadvantage of our solution is that symbols in a strategy must be tagged *Left* or *Right*. In our actual implementation we circumvent the tagging by overloading the strategy combinators. As a result, strategies can really be defined as the specification in Example 2.3.

Now that we can label parts of a strategy, we want to keep track at which point in the strategy we are, and we do so without changing the underlying machinery. We start with defining the *Step* helper data type:

$$
\textbf{data}\ \mathit{Step}\ l\ a = \mathit{Enter}\ l \mid \mathit{Step}\ (\mathit{Rule}\ a) \mid \mathit{Exit}\ l\ \textbf{deriving}\ \mathit{Show}
$$

A step is a rewrite rule (constructor *Step*), or a constructor to indicate that we entered (or left) a labeled part of the strategy. A labeled strategy can be turned into a grammar over steps in the following way:

$$
\begin{array}{l}
\mathit{withSteps} :: \mathit{LabeledStrategy}\ l\ a \rightarrow \mathit{Grammar}\ (\mathit{Step}\ l\ a) \\
\mathit{withSteps}\ (\mathit{Label}\ l\ s) \ = \ \mathit{symbol}\ (\mathit{Enter}\ l) \\
\qquad\qquad\qquad\qquad\quad <\star> \mathit{mapSymbol}\ (\mathit{either}\ (\mathit{symbol} \circ \mathit{Step})\ \mathit{withSteps})\ s \\
\qquad\qquad\qquad\qquad\quad <\star> \mathit{symbol}\ (\mathit{Exit}\ l)
\end{array}
$$

For each label, we introduce symbols that mark the beginning and the end of that label. We use the function *mapSymbol* to transform strategy s to a grammar of steps. The function $\mathit{mapSymbol} :: (a \rightarrow \mathit{Grammar}\ b) \rightarrow \mathit{Grammar}\ a \rightarrow \mathit{Grammar}\ b$ applies its argument function f to all symbols in a grammar. Note that f returns a grammar, and therefore can be used to change symbols and to flatten a grammar of grammars in one traversal. Each symbol of s is either a rewrite rule or a labeled strategy (see the type definition of *Strategy*): a rewrite rule becomes a symbol with a step, and a labeled strategy is handled by calling the function *withSteps* recursively.

To run a grammar with steps, we first have to overload the function *apply* such that it also works on *Step*, and generalize the types of *run* and run' accordingly. The step data type gives us more information, as we show in our next example.

Example 3.2 Suppose that we run the strategy of Example 2.3 on the term $\frac{2}{5} + \frac{2}{3}$: what would be the result? Of course, we would expect to get the derivation:

$$
\frac{2}{5} + \frac{2}{3} \ = \ \frac{6}{15} + \frac{2}{3} \ = \ \frac{6}{15} + \frac{10}{15} \ = \ \frac{16}{15} \ = \ 1\frac{1}{15}
$$

The final answer, $1\frac{1}{15}$, is indeed what we get. In fact, this term is returned twice because the strategy does not specify which of the fractions should be renamed first, which results in two different derivations. It is much more informative to step through the above derivation and see the intermediate steps.

$$
\begin{array}{lllll}
[\mathit{Enter}\ \ell_0, & \mathit{Enter}\ \ell_1, & \mathit{Step}\ \mathit{ruleLCD}, & \mathit{Exit}\ \ell_1, & \mathit{Enter}\ \ell_2, \\
\ \mathit{Step}\ \mathit{down}, & \mathit{Step}\ \underline{\mathit{ruleRename}}, & \mathit{Step}\ \mathit{up}, & \mathit{Step}\ \mathit{down}, & \mathit{Step}\ \underline{\mathit{ruleRename}}, \\
\ \mathit{Step}\ \mathit{up}, & \mathit{Step}\ \mathit{not}, & \mathit{Exit}\ \ell_2, & \mathit{Enter}\ \ell_3, & \mathit{Step}\ \underline{\mathit{ruleAdd}}, \\
\ \mathit{Exit}\ \ell_3, & \mathit{Enter}\ \ell_4, & \mathit{Step}\ \underline{\mathit{ruleSimpl}}, & \mathit{Exit}\ \ell_4, & \mathit{Exit}\ \ell_0]
\end{array}
$$

The list has twenty steps, but only four correspond to actual steps from the derivation: the rules of those steps are underlined. The other rules are administrative: the rules *up* and *down* are introduced by the *somewhere* combinator, whereas *not* comes from the use of *repeat*. Also observe that each *Enter* step has a matching *Exit* step. In principle, a label can be visited multiple times by a strategy.

4 Extensions

The previous section presents the core of our work on recognizing strategies: strategies can be labeled, and with the functions *empty* and *firsts* we can run a strategy. In this section we present three extensions to illustrate the flexibility of our approach.

4.1 Parallel strategies

Suppose that we want to run the strategies s and t in parallel, denoted by $s <||> t$. This operation makes sense in the domain of rewriting: for example, two parts have

to be reduced, and steps to reduce any of the two parts can be interleaved until we are done with both sides. In theory, we can express two strategies that run in parallel in terms of sequences and choices. In practice, however, such a translation does not scale because the grammar will grow tremendously.

In our setup, it is relatively easy to add a new constructor for parallel strategies to the *Grammar* data type, and we will further explore this approach.

```
data Grammar a = ... | Grammar a :||: Grammar a
```

Just as we did for sequences and choices, we first introduce a smart constructor for parallel strategies, which expresses that it has *Succeed* as its unit element, *Fail* as its absorbing element, and that the combinator is associative:

```
(<||>) :: Grammar a → Grammar a → Grammar a
Succeed <||> t          = t
s       <||> Succeed    = s
Fail    <||> _          = fail
_       <||> Fail       = fail
(s :||: t) <||> u       = s :||: (t <||> u)
s       <||> t          = s :||: t
```

Next, we extend the definitions of *empty* and *firsts* with a new case:

```
empty (s :||: t) = empty s ∧ empty t
firsts (s :||: t) = [(a, s' <||> t ) | (a, s') ← firsts s] ++
                    [(a, s  <||> t') | (a, t') ← firsts t]
```

Other functions that operate on the *Grammar* data type (such as *freeVars* and *mapSymbol*) have to be extended as well, but these changes are minimal. Using a generic traversal library [10] can further reduce the impact of adding a constructor.

In a similar way, we can define useful variants on this combinator, such as a left-biased parallel combinator (which continues with its left operand strategy whenever this is possible), or a parallel combinator that stops as soon as one of its operand strategies is finished.

4.2 Removing left recursion

Because we can inspect the grammar, we can detect and remove left recursion in a grammar. Left-recursive definitions cause the function *firsts* to loop, and are therefore not desirable. Fortunately, removing left recursion from a context-free grammar is a standard procedure, and we can transfer this knowledge directly to our combinator approach. Consider the following left-recursive context-free grammar:

$$X \;::=\; Xb \mid Xc \mid a \mid \epsilon$$

We proceed by grouping the left-recursive alternatives (Xb and Xc) and the remaining alternatives (a and ϵ), and arrive at the following grammar:

$$X \;::=\; a \mid aY \mid \epsilon \mid Y \qquad\qquad Y \;::=\; b \mid bY \mid c \mid cY$$

We briefly sketch how this procedure can be used for our *Grammar* data type. Given a *Rec* constructor, we want to make sure that none of its *Var*s appears first. We replace all variables of the *Rec* in question by a special symbol. Then, we use *firsts* to analyze the grammar, and we divide the alternatives in the left-recursive cases and the remaining cases. With the function *empty* we check for the ϵ alternative. In the last step, we combine all alternatives as we did for the context-free grammar. If necessary, we repeat the procedure until the left recursion has disappeared. Other existing grammar analyses can be reused in a similar way.

Example 4.1 We extend the smart constructors for recursive grammars (*rec* and *fix*) and let them check for left recursion. It is now safe to apply the function *firsts* to the left-recursive grammar $\mathit{fix}\ (\lambda x \rightarrow (x \mathbin{<\star>} \mathit{symbol}\ \texttt{'a'}) \mathbin{<|>} \mathit{succeed})$. Another example is $\mathit{fix}\ (\lambda x \rightarrow x)$, which is simplified to *fail*.

4.3 Serializing the remaining strategy

In a recent project, we offered strategies as a service to the MathDox system [3]. For this binding, we designed a stateless protocol for diagnosing intermediate answers submitted by students. One obstacle in establishing this binding was how to communicate the remaining strategy, which is part of the state of an exercise, back and forth. The representation of a strategy is finite and can be serialized. This is not very appealing because strategies can become quite large, which means that it takes longer to process a request.

The remainder of a strategy can also be encoded as a list of integers, with the extra benefit that the rewrite rules applied so far can be recovered. The encoding is rather simple: the integers in the list only encode which element of the *firsts* set has to be used. A *Prefix* associates symbols (rewrite rules) with the integers from the encoding, and contains the remaining grammar we are interested in:

```
data Prefix a = Prefix [(Int, a)] (Grammar a)
```

This data type is called *Prefix* because we are in the middle of a derivation, which means that we have a prefix of a sentence. The following function constructs a prefix from a list of integers and a labeled strategy:

```
makePrefix :: [Int] → LabeledStrategy l a → Prefix (Step l a)
makePrefix is = f [] is ∘ withSteps
  where f acc []       s = Prefix (reverse acc) s
        f acc (i : is) s = case drop i (firsts s) of
                             (a, t) : _ → f ((i, a) : acc) is t
                             _          → error "invalid prefix"
```

The local function f has an accumulating argument which builds up a list in reverse order for efficiency reasons. This explains why the list *acc* has to be reversed in the case for the empty list. Each integer i from the list is used to select the ith element of the list returned by the function *firsts*.

Example 4.2 Let us compute the list of integers that encodes the full derivation of our running example (see Example 3.2):

$[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0]$

The list contains twenty elements, just like the list of steps. The list is dominated by zeros, which is not a coincident. At most places in our grammar, there is just one path that can be followed, witnessed by the fact that *firsts* returns a singleton list at these positions. A simple optimization is to not add an integer for the cases where *firsts* offers no choice.

5 Error diagnosis and hints

In this section, we explain how strategies can help in diagnosing student errors and reporting useful feedback and hints. We have implemented several kinds of feedback for our own exercise assistants, but also for other e-learning systems that use our feedback services as a back-end. The simplest form of feedback is correct/incorrect for the final answer, which is the kind of feedback that is offered by several online tools. A useful extension to this categorization is to check for equivalence between the submitted term and the initial term. With this, we can distinguish a correct but not yet final answer from an incorrect answer. In the rest of this section we present a number of scenarios in the fraction domain to illustrate the possibilities.

Example 5.1 A student submits $\frac{16}{15}$ as the final answer. The exercise assistant reports back to the student that his answer is correct, but with a gentle reminder that the exercise is not yet finished. In this scenario, the strategy tells exactly which step needs to be done: the improper fraction should be simplified.

Example 5.2 A term is submitted as an intermediate step: the rule used is recognized, but according to the strategy it shouldn't have been applied. For instance, [Rename] is recognized, but the denominators of the fractions are already equal. The student can be warned that although the step is correct, it is better to do something else.

Worked-out problems can be generated from a strategy, showing all the steps to go from the initial term to the expected answer. A worked-out problem is the presentation of a sentence that is generated by the strategy. The next step in a derivation can be calculated with the function *firsts*. The information computed with *firsts* can be presented in different ways: the hints can be very general or very specific, for instance by using the levels of the labels in the strategy. A strategy can complete the exercise, and therefore, progress information, such as the number of steps remaining, is available.

Example 5.3 A student has no clue how to add the fractions $\frac{2}{5}$ and $\frac{2}{3}$, and presses the hint button. The system reports the hint: "make the denominators equal". The fact that the denominators are not yet equal can be concluded from the strategy. If this does not help the student, the system can emit a more specific message stating that the fractions should be renamed such that the denominators become 15. This number is calculated and present in the environment in which the strategy is run. A final hint could suggest to rewrite the part $\frac{2}{5}$ into $\frac{6}{15}$.

Strategies can work together with buggy rules: these rules capture common mistakes, and help to report specialized messages for specific (but often occurring)

errors. In addition to the buggy rules, it is also possible to formulate buggy strategies, i.e., common procedural mistakes. In our fraction domain, for example, a buggy strategy would be to make the numerators equal before adding fractions.

Example 5.4 A student submits $\frac{4}{8}$ as the solution to the exercise $\frac{2}{5}+\frac{2}{3}$. Because the terms are not equivalent, the buggy rules are considered (B1 to B3 from Figure 1), and in this case, rule B1 matches. A special message associated with this rule (e.g., "it is not sound to add the numerators and the denominators of the fractions: rename the fractions first") is reported to the student.

Strategies and rules are essentially the same, except that the structure of a strategy is made explicit. Hence, it is straightforward to turn a strategy into a rule, or a part of a strategy with a certain label. This is convenient if following a strategy becomes routine, and a step-wise approach is no longer helpful to the student. Similarly, a tool can ask a student to solve the entire problem first, and decompose the problem in steps if the submitted answer is not correct.

Example 5.5 A student is asked to provide the final answer to a question, and in case it is incorrect, the exercise tool poses sub-problems to the student. These sub-problems can be calculated automatically from the strategy by looking at the labels. The strategy for adding fractions, for instance, can be decomposed in 4 steps.

6 Related work

There are many tools that offer students an environment in which they can solve exercises incrementally, such as MathDox [3] and ActiveMath [5]. Most of these tools are limited to correct/incorrect feedback, because it is often difficult and laborious in these systems to diagnose mistakes. However, some tools use external domain reasoners for making a diagnosis, which is exactly what our strategy recognizer has to offer. Some work has been done on diagnosing student mistakes on the level of rewrite rules [2,7,15].

In this paper, we discuss the design and implementation of a strategy recognizer, which makes it possible to use strategies for improving error diagnosis. Strategies for specifying exercises are introduced in a different paper [6]. By viewing strategy recognition as a parsing problem, we take advantage of almost 50 years of experience in parsing sentences of context-free languages. The strategy language on which our work is based is very similar to languages that are used in parser libraries [9,13], but also to strategic programming languages such as Stratego [11,14] and Elan [1], data conversion libraries [4], and languages in other domains.

7 Conclusions

This paper presents a complete implementation of a recognizer for strategies. A strategy describes valid sequences of rewrite rules, and is very similar to context-free grammars. Knowledge and experience from the field of parsing sentences can be transferred to the domain of stepwise solving exercises. One example of such a transfer is the grammar transformation to remove left recursion.

Although it is tempting to reuse existing parsing tools and libraries, a closer look at the problem reveals subtle differences that make the existing tools unsuitable for dealing with the problem we are facing. Nevertheless, the strategy combinators that we selected as our starting point are inspired by context-free grammars. Some design choices were discussed, in particular how to deal with recursion, and how to mark positions in a strategy. In Section 5 we have shown how strategies can be used to report improved feedback.

We will continue our research on strategy recognizers in several directions. We are working on creating bindings with a number of existing tutoring tools, such as ActiveMath [5]. Protocols are needed to exchange information with such an environment, and we are working on developing and standardizing these protocols. Our tool has a binding with MathDox [3], and has recently been used in a classroom setting. We have collected data from these session, and preliminary analyses show that providing feedback on the strategy level improves far transfer: students that received feedback on the strategic level did better in advanced exercises. A final area that requires further investigation is how to make strategies (and the associated feedback messages) more accessible to teachers.

References

[1] Borovanský, P., C. Kirchner, H. Kirchner and C. Ringeissen, *Rewriting with strategies in ELAN: A functional semantics*, International Journal of Foundations of Computer Science **12** (2001), pp. 69–95.

[2] Bouwers, E., "Improving Automated Feedback – a Generic Rule-Feedback Generator," Master's thesis, Utrecht University, Department of Information and Computing Sciences (2007).

[3] Cohen, A., H. Cuypers, E. Reinaldo Barreiro and H. Sterk, *Interactive mathematical documents on the web*, in: *Algebra, Geometry and Software Systems* (2003), pp. 289–306.

[4] Cunha, A. and J. Visser, *Strongly typed rewriting for coupled software transformation*, Electronic Notes in Theoretical Computer Science **174** (2007), pp. 17–34.

[5] Goguadze, G., A. González Palomo and E. Melis, *Interactivity of exercises in ActiveMath*, in: *International Conference on Computers in Education, ICCE 2005*, 2005.

[6] Heeren, B., J. Jeuring, A. v. Leeuwen and A. Gerdes, *Specifying strategies for exercises*, in: S. Autexier et al., editor, *Proceedings of MKM 2008*, LNAI **5144** (2008), pp. 430–445.

[7] Hennecke, M., "Online Diagnose in intelligenten mathematischen Lehr-Lern-Systemen (in German)," Ph.D. thesis, Hildesheim University (1999), fortschritt-Berichte VDI Reihe 10, Informatik / Kommunikationstechnik; 605. Düsseldorf: VDI-Verlag.

[8] Huet, G., *Functional Pearl: The Zipper*, Journal of Functional Programming **7** (1997), pp. 549–554.

[9] Hutton, G., *Higher-order Functions for Parsing*, Journal of Functional Programming **2** (1992), pp. 323–343.

[10] Lämmel, R. and S. Peyton Jones, *Scrap your boilerplate: a practical approach to generic programming*, ACM SIGPLAN Notices **38** (2003), pp. 26–37, TLDI'03.

[11] Lämmel, R., E. Visser and J. Visser, *The Essence of Strategic Programming* (2003), 20 p.; Draft as of October 8, 2003; Available from http://homepages.cwi.nl/~ralf/eosp/paper.pdf.

[12] Peyton Jones et al., S., "Haskell 98, Language and Libraries. The Revised Report," Cambridge University Press, 2003, a special issue of the Journal of Functional Programming, see also http://www.haskell.org/.

[13] Swierstra, S. D. and L. Duponcheel, *Deterministic, error-correcting combinator parsers*, in: J. Launchbury, E. Meijer and T. Sheard, editors, *Advanced Functional Programming*, LNCS **1129** (1996), pp. 184–207.

[14] Visser, E., Z.-e.-A. Benaissa and A. Tolmach, *Building program optimizers with rewriting strategies*, in: *ICFP '98*, 1998, pp. 13–26.

[15] Zinn, C., *Supporting tutorial feedback to student help requests and errors in symbolic differentiation*, in: M. Ikeda, K. Ashley and T.-W. Chan, editors, *ITS 2006*, LNCS **4053** (2006), pp. 349–359.

Operational Termination of Conditional Rewriting with Built-in Numbers and Semantic Data Structures ⋆

Stephan Falke[1] Deepak Kapur[2]

Department of Computer Science, University of New Mexico, Albuquerque, NM 87131, USA

Abstract

While ordinary conditional rewrite systems are more elegant than unconditional ones, they still have limited expressive power since semantic data structures, such as sets or multisets, cannot be modeled elegantly. Extending our work presented at RTA 2008 [9], the present paper defines a class of conditional rewrite systems that allows the use of semantic data structures and supports built-in natural numbers, including constraints taken from Presburger arithmetic. The framework is both expressive and natural. Rewriting is performed using a combination of normalized equational rewriting with recursive evaluation of conditions and validity checking of instantiated constraints.
Termination is one of the most important properties of any kind of rewriting. For conditional systems, it is not sufficient to only show well-foundedness of the rewrite relation, but it also has to be ensured that evaluation of the conditions terminates. These properties are captured by the notion of *operational termination*. In this work, we show that operational termination for the class of conditional rewrite systems discussed above can be reduced to (regular) termination of unconditional systems using a syntactic transformation. Powerful methods for showing termination of unconditional systems are presented in [9].

Keywords: Conditional term rewriting, operational termination, semantic data structures

1 Introduction

Conditional term rewrite systems operating on free data structures provide a powerful framework for specifying algorithms. This approach has successfully been taken by the system Maude [4]. Many algorithms, however, operate on semantic data structures like finite sets, multisets, or sorted lists (e.g., using Java's collection classes or the OCaml extension Moca [3]). Constructors used to generate such data structures satisfy certain properties, i.e., they are not free. For example, finite sets can be generated using the empty set, singleton sets, and set union. Set union is associative, commutative, idempotent, and has the empty set as unit element. Such semantic data structures can be modeled using equational axioms.

⋆ Partially supported by NSF grant CCF-0541315.

[1] Email: spf@cs.unm.edu

[2] Email: kapur@cs.unm.edu

Extending our work presented at CADE 2007 and RTA 2008 [7,9], the present paper introduces conditional constrained equational rewrite systems (CCESs) which have three components: (i) $\mathcal{R}$, a set of conditional constrained rewrite rules for specifying algorithms on semantic data structures, (ii) $\mathcal{S}$, a set of constrained rewrite rules on constructors, and (iii) $\mathcal{E}$, a set of equations on constructors. Here, (ii) and (iii) are used for modeling semantic data structures where normalization with $\mathcal{S}$ yields normal forms that are unique up to equivalence w.r.t. $\mathcal{E}$. The constraints for $\mathcal{R}$ and $\mathcal{S}$ are Boolean combinations of atomic formulas of the form $s \simeq t$ and $s > t$ from Presburger arithmetic. Rewriting with such a system is performed using a combination of normalized rewriting [12] with evaluation of conditions and validity checking of instantiated constraints. Before matching a redex with the left side of a rule, the redex is first normalized with $\mathcal{S}$. Additionally, the rewrite step is only performed if the instantiated conditions of the rule can be established by recursively rewriting them and if the instantiated constraint of the rule is valid. The difference between conditions and constraints in a rule is thus operational.

Example 1.1 *This example shows a quicksort algorithm that takes a set and returns a list. It is a modification of an example from [2] that is widely used in the literature on conditional rewriting. Sets are constructed using* $\emptyset$ *and* ins*, where* ins *adds an element to a set. The semantics of sets is modeled using* $\mathcal{S}$ *and* $\mathcal{E}$ *as follows.*

$$\begin{aligned} \mathcal{E}\text{:}\quad \mathsf{ins}(x, \mathsf{ins}(y, zs)) &\approx \mathsf{ins}(y, \mathsf{ins}(x, zs)) \\ \mathcal{S}\text{:}\quad \mathsf{ins}(x, \mathsf{ins}(x, ys)) &\to \mathsf{ins}(x, ys) \end{aligned}$$

Quicksort is specified by the following conditional constrained rewrite rules.

$$\begin{array}{rcll} \mathsf{app}(\mathsf{nil}, zs) & \to & zs & \\ \mathsf{app}(\mathsf{cons}(x, ys), zs) & \to & \mathsf{cons}(x, \mathsf{app}(ys, zs)) & \\ \mathsf{split}(x, \emptyset) & \to & \langle \emptyset, \emptyset \rangle & \\ \mathsf{split}(x, zs) \to^* \langle zl, zh \rangle \mid \mathsf{split}(x, \mathsf{ins}(y, zs)) & \to & \langle \mathsf{ins}(y, zl), zh \rangle & [\![x > y]\!] \\ \mathsf{split}(x, zs) \to^* \langle zl, zh \rangle \mid \mathsf{split}(x, \mathsf{ins}(y, zs)) & \to & \langle zl, \mathsf{ins}(y, zh) \rangle & [\![x \not> y]\!] \\ \mathsf{qsort}(\emptyset) & \to & \mathsf{nil} & \\ \mathsf{split}(x, ys) \to^* \langle yl, yh \rangle \mid \mathsf{qsort}(\mathsf{ins}(x, ys)) & \to & \mathsf{app}(\mathsf{qsort}(yl), \mathsf{cons}(x, \mathsf{qsort}(yh))) & \end{array}$$

Here, $\mathsf{split}(x, ys)$ *returns a pair of sets* $\langle yl, yh \rangle$ *where* yl *contains all* $y \in ys$ *such that* $x > y$ *and* yh *contains all* $y \in ys$ *such that* $x \not> y$. ◇

One of the most important properties of a CCES is that a rewrite engine operating with it always terminates. For this, it has to be shown that the rewrite relation is well-founded and that the evaluation of the conditions terminates. These properties can be characterized by the notion of *operational termination* [11].[3] The recursive nature of conditional rewriting is reflected in an inference systems for proving that a term s can be reduced to a term t, and operational termination is the property that this inference system does not allow infinite derivations.

[3] Another commonly used characterization is *effective termination*, see, e.g., [13]. However, as argued in [11], operational termination better captures the behaviour of actual rewrite engines.

The present paper shows that operational termination of a conditional system can be reduced to termination of an unconditional system using a syntactic transformation. This transformation is similar to the transformation used for ordinary conditional rewriting, see, e.g., [13, Definition 7.2.48]. Powerful methods based on dependency pairs for showing termination of unconditional systems are presented in [9], and in combination with the current paper these methods can be used for showing operational termination of CCESs as well.

This paper is organized as follows. In Section 2, the rewrite relation is defined. In Section 3, we formally define the notion of operational termination and show that termination and operational termination coincide for unconditional systems. Section 4 introduces a transformation from conditional systems into unconditional ones. We show that termination of the transformed system implies operational termination of the original system. The omitted proofs may be found in the full version of this paper [8], and [6] contains several nontrivial conditional systems whose operational termination can be shown by applying the transformation presented in this paper and using the termination techniques presented in [9].

2 Conditional Normalized Rewriting with Constraints

We assume familiarity with the concepts and notations of term rewriting [1]. We consider terms over two sorts, `nat` and `univ`, and we use an initial signature $\mathcal{F}_{\mathcal{PA}} = \{\mathsf{0}, \mathsf{1}, +\}$ using only sort `nat`. Properties of natural numbers are modelled using the set $\mathcal{PA} = \{x + (y + z) \approx (x + y) + z,\ x + y \approx y + x,\ x + \mathsf{0} \approx x\}$ of equations. For each $k \in \mathbb{N} - \{0\}$, we denote the term $\mathsf{1} + \ldots + \mathsf{1}$ (with k occurrences of $\mathsf{1}$) by k.

We then extend $\mathcal{F}_{\mathcal{PA}}$ by a finite sorted signature $\mathcal{F}$. We omit stating the sorts explicitly in examples if they can be inferred. In the following we assume that all terms, contexts, context replacements, substitutions, rewrite rules, equations, etc. are sort correct. For any syntactic construct c we let $\mathcal{V}(c)$ denote the set of variables occurring in c. The root symbol of a term s is denoted by $\mathrm{root}(s)$. The root position of a term is denoted by λ. For an arbitrary set $\mathcal{E}$ of equations and terms s, t we write $s \to_{\mathcal{E}} t$ iff there exist an equation $u \approx v \in \mathcal{E}$, a substitution σ, and a position $p \in \mathcal{P}os(s)$ such that $s|_p = u\sigma$ and $t = s[v\sigma]_p$. The symmetric closure of $\to_{\mathcal{E}}$ is denoted by $\vdash\dashv_{\mathcal{E}}$, and the reflexive transitive closure of $\vdash\dashv_{\mathcal{E}}$ is denoted by $\sim_{\mathcal{E}}$. For two terms s, t we write $s \sim_{\mathcal{E}}^{>\lambda} t$ iff $s = f(s_1, \ldots, s_n)$ and $t = f(t_1, \ldots, t_n)$ such that $s_i \sim_{\mathcal{E}} t_i$ for all $1 \leq i \leq n$, i.e., if equations are only applied below the root.

An *atomic $\mathcal{PA}$-constraint* has the form $\top$ (truth), $s \simeq t$ (equality) or $s > t$ (greater) for terms $s, t \in \mathcal{T}(\mathcal{F}_{\mathcal{PA}}, \mathcal{V})$. The set of *$\mathcal{PA}$-constraints* is defined to be the closure of the set of atomic $\mathcal{PA}$-constraints under $\neg$ (negation) and $\wedge$ (conjunction). Validity (the constraint is true for all assignments) and satisfiability (the constraint is true for some assignment) of $\mathcal{PA}$-constraints are defined as usual, where we take the set of natural numbers as universe of concern. We also speak of $\mathcal{PA}$-validity and $\mathcal{PA}$-satisfiability. These properties are decidable [15].

The rewrite rules that we consider are ordinary conditional rewrite rules together with a $\mathcal{PA}$-constraint C.

Definition 2.1 (Conditional Constrained Rewrite Rule) *A* conditional constrained rewrite rule *has the form* $s_1 \to^* t_1, \ldots, s_n \to^* t_n \mid l \to r[\![C]\!]$ *such that*

(i) $l, r \in \mathcal{T}(\mathcal{F} \cup \mathcal{F}_{\mathcal{PA}}, \mathcal{V})$ *such that* $\text{root}(l) \in \mathcal{F}$,

(ii) $s_i, t_i \in \mathcal{T}(\mathcal{F} \cup \mathcal{F}_{\mathcal{PA}}, \mathcal{V})$,

(iii) C *is a* $\mathcal{PA}$*-constraint,*

(iv) $\mathcal{V}(r) \subseteq \mathcal{V}(l) \cup \bigcup_{j=1}^{n} \mathcal{V}(t_j)$, *and*

(v) $\mathcal{V}(s_i) \subseteq \mathcal{V}(l) \cup \bigcup_{j=1}^{i-1} \mathcal{V}(t_j)$ *for all* $1 \leq i \leq n$.[4]

The difference between conditions and constraints in a rule is operational. Conditions need to be evaluated by recursively rewriting them, while constraints are checked using a decision procedure for $\mathcal{PA}$-validity. This distinction will be formalized in Definition 2.7. In a rule $l \to r[\![\top]\!]$ the constraint $\top$ will be omitted. For a set $\mathcal{R}$ of constrained rewrite rules, the set of *defined symbols* is given by $\mathcal{D}(\mathcal{R}) = \{f \mid f = \text{root}(l) \text{ for some } s_1 \to^* t_1, \ldots, s_n \to^* t_n \mid l \to r[\![C]\!] \in \mathcal{R}\}$. The set of *constructors* is $\mathcal{C}(\mathcal{R}) = \mathcal{F} - \mathcal{D}(\mathcal{R})$. Note that according to this definition, the symbols from $\mathcal{F}_{\mathcal{PA}}$ are considered to be neither defined symbols nor constructors.

Properties of non-free data structures are modelled using *constructor equations* and *constructor rules*. Constructor equations need to be linear and regular.

Definition 2.2 (Constructor Equations) *A* constructor equation *has the form* $u \approx v$ *for terms* $u, v \in \mathcal{T}(\mathcal{C}(\mathcal{R}), \mathcal{V})$ *such that* $u \approx v$ *has* identical unique variables *(is* i.u.v.*), i.e.,* u *and* v *are linear and* $\mathcal{V}(u) = \mathcal{V}(v)$.

Similar to conditional constrained rewrite rules, constructor rules have a $\mathcal{PA}$-constraint that will guard when a rule is applicable.

Definition 2.3 (Constructor Rules) *A* constructor rule *is a rule* $l \to r[\![C]\!]$ *with* $l, r \in \mathcal{T}(\mathcal{C}(\mathcal{R}), \mathcal{V})$ *and a* $\mathcal{PA}$*-constraint* C *where* $\text{root}(l) \in \mathcal{C}(\mathcal{R})$ *and* $\mathcal{V}(r) \subseteq \mathcal{V}(l)$.

Again, constraints C of the form $\top$ will be omitted in constructor rules. Constructor rules and equations give rise to the following rewrite relation. It is based on extended rewriting [14] but requires that the $\mathcal{PA}$-constraint of the constructor rule is $\mathcal{PA}$-valid after being instantiated by the matcher. For this, we require that variables of sort `nat` are instantiated by terms over $\mathcal{F}_{\mathcal{PA}}$ in order to ensure that $\mathcal{PA}$-validity of the instantiated $\mathcal{PA}$-constraint can be decided by a decision procedure for $\mathcal{PA}$-validity.[5]

Definition 2.4 ($\mathcal{PA}$-based Substitutions) *Let* σ *be a substitution. Then* σ *is* $\mathcal{PA}$-based *iff* $\sigma(x) \in \mathcal{T}(\mathcal{F}_{\mathcal{PA}}, \mathcal{V})$ *for all variables* x *of sort* `nat`.

Definition 2.5 (Constructor Rewrite Relation) *Let* $\mathcal{E}$ *be a finite set of constructor equations and let* $\mathcal{S}$ *be a finite set of constructor rules. Then* $s \to_{\mathcal{PA}\|\mathcal{E}\backslash\mathcal{S}} t$ *iff there exist a constructor rule* $l \to r[\![C]\!] \in \mathcal{S}$*, a position* $p \in \mathcal{P}os(s)$*, and a* $\mathcal{PA}$*-based substitution* σ *such that*

(i) $s|_p \sim_{\mathcal{E}\cup\mathcal{PA}} l\sigma$,[6]

(ii) $C\sigma$ *is* $\mathcal{PA}$*-valid, and*

(iii) $t = s[r\sigma]_p$.

[4] Using the notation of [13], the last two conditions yield deterministic type 3 rules.

[5] This requirement can be relaxed slighlty by requiring that only those variables of sort `nat` that occur in the $\mathcal{PA}$-constraint need to be instantiated by terms over $\mathcal{F}_{\mathcal{PA}}$.

[6] Recall that $\mathcal{PA}$ also denotes the set of equations introduced above.

We write $s \to^{>\lambda}_{\mathcal{PA}\|\mathcal{E}\backslash\mathcal{S}} t$ iff $s \to_{\mathcal{PA}\|\mathcal{E}\backslash\mathcal{S}} t$ at a position $p \neq \lambda$, and $s \xrightarrow{!}{}^{>\lambda}_{\mathcal{PA}\|\mathcal{E}\backslash\mathcal{S}} t$ iff s reduces to t in zero or more $\to^{>\lambda}_{\mathcal{PA}\|\mathcal{E}\backslash\mathcal{S}}$ steps and t is a normal form w.r.t. $\to^{>\lambda}_{\mathcal{PA}\|\mathcal{E}\backslash\mathcal{S}}$.

We combine conditional constrained rewrite rules and constructor rules and equations into a conditional constrained equational system (CCES).

Definition 2.6 (CCES) *A* CCES *has the form* $(\mathcal{R}, \mathcal{S}, \mathcal{E})$ *for a finite set* $\mathcal{R}$ *of conditional constrained rewrite rules, a finite set* $\mathcal{S}$ *of constructor rules, and a finite set* $\mathcal{E}$ *of constructor equations such that*

(i) $\mathcal{S}$ *is right-linear, i.e., variables occur at most once in* r *for all* $l \to r[\![C]\!] \in \mathcal{S}$*,*

(ii) $\sim_{\mathcal{E}\cup\mathcal{PA}}$ *commutes over* $\to_{\mathcal{PA}\|\mathcal{E}\backslash\mathcal{S}}$*, i.e.,* $\sim_{\mathcal{E}\cup\mathcal{PA}} \circ \to_{\mathcal{PA}\|\mathcal{E}\backslash\mathcal{S}} \subseteq \to_{\mathcal{PA}\|\mathcal{E}\backslash\mathcal{S}} \circ \sim_{\mathcal{E}\cup\mathcal{PA}}$*, and*

(iii) $\to_{\mathcal{PA}\|\mathcal{E}\backslash\mathcal{S}}$ *is convergent modulo* $\sim_{\mathcal{E}\cup\mathcal{PA}}$*, i.e.,* $\to_{\mathcal{PA}\|\mathcal{E}\backslash\mathcal{S}}$ *is terminating and we have* $\leftarrow^*_{\mathcal{PA}\|\mathcal{E}\backslash\mathcal{S}} \circ \to^*_{\mathcal{PA}\|\mathcal{E}\backslash\mathcal{S}} \subseteq \to^*_{\mathcal{PA}\|\mathcal{E}\backslash\mathcal{S}} \circ \sim_{\mathcal{E}\cup\mathcal{PA}} \circ \leftarrow^*_{\mathcal{PA}\|\mathcal{E}\backslash\mathcal{S}}$*.*

The commutation property intuitively states that if $s \sim_{\mathcal{E}\cup\mathcal{PA}} s'$ and $s' \to_{\mathcal{PA}\|\mathcal{E}\backslash\mathcal{S}} t'$, then $s \to_{\mathcal{PA}\|\mathcal{E}\backslash\mathcal{S}} t$ for some $t \sim_{\mathcal{E}\cup\mathcal{PA}} t'$. If $\mathcal{S}$ does not already satisfy this property then it can be achieved by adding *extended rules* [14,10]. It might be hard to check the conditions on $\to_{\mathcal{PA}\|\mathcal{E}\backslash\mathcal{S}}$ automatically and an implementation might thus be restricted to some commonly used data structures for which these properties have been established manually. Several examples are listed in Figure 1. The rule "$(*)$" is needed in order to make $\sim_{\mathcal{E}\cup\mathcal{PA}}$ commute over $\to_{\mathcal{PA}\|\mathcal{E}\backslash\mathcal{S}}$. The constructor $\langle\cdot\rangle$ creates a singleton set or multiset, respectively.

If $\mathcal{R}$ is unconditional (i.e., $n = 0$ for all $s_1 \to^* t_1, \ldots, s_n \to^* t_n \mid l \to r[\![C]\!]$ in $\mathcal{R}$), a CCES will also be called a *CES* [9]. The rewrite relation corresponding to a CCES is an extension of the rewrite relation considered in [9].

Definition 2.7 (Conditional Rewrite Relation) *Let* $(\mathcal{R}, \mathcal{S}, \mathcal{E})$ *be a CCES. The rewrite relation* $\xrightarrow{\mathcal{S}}_{\mathcal{PA}\|\mathcal{E}\backslash\mathcal{R}}$ *is the least relation satisfying* $s \xrightarrow{\mathcal{S}}_{\mathcal{PA}\|\mathcal{E}\backslash\mathcal{R}} t$ *iff there exist a conditional constraint rewrite rule* $s_1 \to^* t_1, \ldots, s_n \to^* t_n \mid l \to r[\![C]\!]$ *in* $\mathcal{R}$*, a position* $p \in \mathcal{P}os(s)$*, and a* $\mathcal{PA}$*-based substitution* σ *such that*

(i) $s|_p \xrightarrow{!}{}^{>\lambda}_{\mathcal{PA}\|\mathcal{E}\backslash\mathcal{S}} \circ \sim^{>\lambda}_{\mathcal{E}\cup\mathcal{PA}} l\sigma$*,*

(ii) $C\sigma$ *is* $\mathcal{PA}$*-valid,*

(iii) $s_i\sigma \xrightarrow{\mathcal{S}}{}^*_{\mathcal{PA}\|\mathcal{E}\backslash\mathcal{R}} \circ \sim_{\mathcal{E}\cup\mathcal{PA}} t_i\sigma$ *for all* $1 \leq i \leq n$*, and*

(iv) $t = s[r\sigma]_p$*.*

Notice that the restriction to $\mathcal{PA}$-based substitution enforces a kind of innermost rewriting for function symbols with resulting sort `nat`. The least relation satisfying Definition 2.7 can be obtained by an inductive construction, similarly to ordinary conditional rewriting (see, e.g., [13]).

Example 2.8 *Continuing Example 1.1 we now illustrate* $\xrightarrow{\mathcal{S}}_{\mathcal{PA}\|\mathcal{E}\backslash\mathcal{R}}$*. Consider* $t = \mathsf{qsort}(\mathsf{ins}(1, \mathsf{ins}(3, \mathsf{ins}(1, \emptyset))))$ *and the* $\mathcal{PA}$*-based substitution* $\sigma = \{x \mapsto 3, ys \mapsto \mathsf{ins}(1, \emptyset), yl \mapsto \mathsf{ins}(1, \emptyset), yh \mapsto \emptyset\}$*. We have* $t \xrightarrow{!}{}^{>\lambda}_{\mathcal{PA}\|\mathcal{E}\backslash\mathcal{S}} \mathsf{qsort}(\mathsf{ins}(1, \mathsf{ins}(3, \emptyset))) \sim^{>\lambda}_{\mathcal{E}\cup\mathcal{PA}} \mathsf{qsort}(\mathsf{ins}(x, ys))\sigma$ *and thus* $t \xrightarrow{\mathcal{S}}_{\mathcal{PA}\|\mathcal{E}\backslash\mathcal{R}} \mathsf{app}(\mathsf{qsort}(\mathsf{ins}(1, \emptyset)), \mathsf{cons}(3, \mathsf{qsort}(\emptyset)))$ *using the third rule for* qsort*, provided* $\mathsf{split}(3, \mathsf{ins}(1, \emptyset)) \xrightarrow{\mathcal{S}}{}^*_{\mathcal{PA}\|\mathcal{E}\backslash\mathcal{R}} \circ \sim_{\mathcal{E}\cup\mathcal{PA}} \langle \mathsf{ins}(1, \emptyset), \emptyset \rangle$*. In*

	Constructors	$\mathcal{E}$	$\mathcal{S}$
Sorted lists	$\mathsf{nil}, \mathsf{cons}$		$\mathsf{cons}(x, \mathsf{cons}(y, zs)) \to$ $\mathsf{cons}(y, \mathsf{cons}(x, zs))[\![x > y]\!]$
Multi-sets	$\emptyset, \mathsf{ins}$	$\mathsf{ins}(x, \mathsf{ins}(y, zs)) \approx$ $\mathsf{ins}(y, \mathsf{ins}(x, zs))$	
Multi-sets	$\emptyset, \langle\cdot\rangle, \cup$	$x \cup (y \cup z) \approx (x \cup y) \cup z$ $x \cup y \approx y \cup x$	$x \cup \emptyset \to x$
Sets	$\emptyset, \mathsf{ins}$	$\mathsf{ins}(x, \mathsf{ins}(y, zs)) \approx$ $\mathsf{ins}(y, \mathsf{ins}(x, zs))$	$\mathsf{ins}(x, \mathsf{ins}(x, ys)) \to$ $\mathsf{ins}(x, ys)$
Sets	$\emptyset, \langle\cdot\rangle, \cup$	$x \cup (y \cup z) \approx (x \cup y) \cup z$ $x \cup y \approx y \cup x$	$x \cup \emptyset \to x$ $x \cup x \to x$ $(x \cup x) \cup y \to x \cup y \quad (*)$
Sorted sets	$\emptyset, \mathsf{ins}$		$\mathsf{ins}(x, \mathsf{ins}(y, zs) \to$ $\mathsf{ins}(y, \mathsf{ins}(x, zs))[\![x > y]\!]$ $\mathsf{ins}(x, \mathsf{ins}(y, zs)) \to$ $\mathsf{ins}(x, zs)[\![x \simeq y]\!]$

Fig. 1. Commonly used data structures.

order to verify this, we use the second split*-rule. For this, we first need to check that the instantiated constraint* $3 \not> 1$ *is* $\mathcal{PA}$*-valid. Furthermore we need to show that* $\mathsf{split}(3, \emptyset) \xrightarrow{\mathcal{S}}^*_{\mathcal{PA}\|\mathcal{E}\backslash\mathcal{R}} \circ \sim_{\mathcal{E}\cup\mathcal{PA}} \langle\emptyset, \emptyset\rangle$*, which is established by the first* split*-rule. Reducing* $\mathsf{app}(\mathsf{qsort}(\mathsf{ins}(1, \emptyset)), \mathsf{cons}(3, \mathsf{qsort}(\emptyset)))$ *eventually produces* $\mathsf{cons}(1, \mathsf{cons}(3, \mathsf{nil}))$*.* $\diamond$

It can be shown [8] that whenever $s \sim_{\mathcal{E}\cup\mathcal{PA}} s'$ and $s \xrightarrow{\mathcal{S}}_{\mathcal{PA}\|\mathcal{E}\backslash\mathcal{R}} t$, then $s' \xrightarrow{\mathcal{S}}_{\mathcal{PA}\|\mathcal{E}\backslash\mathcal{R}} t'$ for some $t' \sim_{\mathcal{E}\cup\mathcal{PA}} t$, i.e., $\sim_{\mathcal{E}\cup\mathcal{PA}}$ commutes over $\xrightarrow{\mathcal{S}}_{\mathcal{PA}\|\mathcal{E}\backslash\mathcal{R}}$.

Lemma 2.9 *For any CCES* $(\mathcal{R}, \mathcal{S}, \mathcal{E})$ *we have* $\sim_{\mathcal{E}\cup\mathcal{PA}} \circ \xrightarrow{\mathcal{S}}_{\mathcal{PA}\|\mathcal{E}\backslash\mathcal{R}} \subseteq \xrightarrow{\mathcal{S}}_{\mathcal{PA}\|\mathcal{E}\backslash\mathcal{R}} \circ \sim_{\mathcal{E}\cup\mathcal{PA}}$*. Furthermore, the* $\xrightarrow{\mathcal{S}}_{\mathcal{PA}\|\mathcal{E}\backslash\mathcal{R}}$ *steps can be performed using the same conditional constrained rewrite rule and* $\mathcal{PA}$*-based substitution.*

3 Termination and Operational Termination

Termination of a CCES means that there is no term that starts an infinite $\xrightarrow{\mathcal{S}}_{\mathcal{PA}\|\mathcal{E}\backslash\mathcal{R}}$ reduction, i.e., that the relation $\xrightarrow{\mathcal{S}}_{\mathcal{PA}\|\mathcal{E}\backslash\mathcal{R}}$ is well-founded. As is well-known, termination is not the only crucial property of conditional rewriting. In order to get a decidable rewrite relation it additionally has to be ensured that evaluation of the conditions terminates. As argued in [11], the notion of *operational termination* is a

$$(\mathsf{Refl}) \quad \frac{}{s \rightarrow^* t} \quad \text{if } s \sim_{\mathcal{E}\cup\mathcal{PA}} t$$

$$(\mathsf{Tran}) \quad \frac{s \rightarrow t \qquad t \rightarrow^* u}{s \rightarrow^* u}$$

$$(\mathsf{Repl}) \quad \frac{s_1\sigma \rightarrow^* t_1\sigma \qquad \cdots \qquad s_n\sigma \rightarrow^* t_n\sigma}{s \rightarrow t}$$

$$\begin{aligned} \text{if} \quad & s_1 \rightarrow^* t_1, \ldots, s_n \rightarrow^* t_n \mid l \rightarrow r[\![C]\!] \in \mathcal{R}, \\ & p \in \mathcal{P}os(s), \\ & \sigma \text{ is } \mathcal{PA}\text{-based}, \\ & s|_p \xrightarrow{!}{}^{>\lambda}_{\mathcal{PA}\|\mathcal{E}\backslash\mathcal{S}} \circ \sim^{>\lambda}_{\mathcal{E}\cup\mathcal{PA}} l\sigma, \\ & C\sigma \text{ is } \mathcal{PA}\text{-valid, and} \\ & t = s[r\sigma]_p. \end{aligned}$$

Fig. 2. Derivation rules.

natural choice for this since it better captures the behavior of actual rewrite engines than other commonly used notions like *effective termination* [13].

As in [11], the recursive nature of conditional rewriting is reflected in an inference system that aims at proving $s \xrightarrow{\mathcal{S}}_{\mathcal{PA}\|\mathcal{E}\backslash\mathcal{R}} t$ or $s \xrightarrow{\mathcal{S}}{}^*_{\mathcal{PA}\|\mathcal{E}\backslash\mathcal{R}} t$. Operational termination is then characterized by the absence of infinite proof trees for this inference system.

Definition 3.1 (Proof Trees) *Let $(\mathcal{R}, \mathcal{S}, \mathcal{E})$ be a CCES. The set of (finite)* proof trees *for $(\mathcal{R}, \mathcal{S}, \mathcal{E})$ and the* head *of a proof tree are inductively defined as follows.*

(i) *An* open goal *G, where G is either $s \rightarrow t$ or $s \rightarrow^* t$ for some terms s, t, is a proof tree. In this case* $\text{head}(G) = G$ *is the head of the proof tree.*

(ii) *A* derivation tree*, denoted by*

$$T = \frac{T_1 \qquad \cdots \qquad T_n}{G}(\Delta)$$

is a proof tree, where G is as in the first case, Δ is one of the derivation rules in Figure 2, and $T_1, \ldots, T_n$ are proof trees such that

$$\frac{\text{head}(T_1) \qquad \cdots \qquad \text{head}(T_n)}{G}$$

is an instance of Δ. In this case, $\text{head}(T) = G$.

A proof tree is closed *iff it does not contain any open goals.*

Example 3.2 *We again consider the CCES for quicksort from Examples 1.1 and 2.8. Then* $\mathsf{qsort}(\mathsf{ins}(1, \mathsf{ins}(3, \mathsf{ins}(1, \emptyset)))) \to \mathsf{app}(\mathsf{qsort}(\mathsf{ins}(1, \emptyset)), \mathsf{cons}(3, \mathsf{qsort}(\emptyset)))$ *is an open goal and*

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{split}(3,\emptyset) \to \langle\emptyset,\emptyset\rangle}\,(\mathsf{Repl}) \qquad \dfrac{}{\langle\emptyset,\emptyset\rangle \to^* \langle\emptyset,\emptyset\rangle}\,(\mathsf{Refl})}{\mathsf{split}(3,\emptyset) \to^* \langle\emptyset,\emptyset\rangle}\,(\mathsf{Tran})}{\mathsf{split}(3,\mathsf{ins}(1,\emptyset)) \to \langle\mathsf{ins}(1,\emptyset),\emptyset\rangle}\,(\mathsf{Repl}) \qquad \dfrac{}{\langle\mathsf{ins}(1,\emptyset),\emptyset\rangle \to^* \langle\mathsf{ins}(1,\emptyset),\emptyset\rangle}\,(\mathsf{Refl})}{\mathsf{split}(3,\mathsf{ins}(1,\emptyset)) \to^* \langle\mathsf{ins}(1,\emptyset),\emptyset\rangle}\,(\mathsf{Tran})}{\mathsf{qsort}(\mathsf{ins}(1,\mathsf{ins}(3,\mathsf{ins}(1,\emptyset)))) \to \mathsf{app}(\mathsf{qsort}(\mathsf{ins}(1,\emptyset)),\mathsf{cons}(3,\mathsf{qsort}(\emptyset)))}\,(\mathsf{Repl})$$

is a closed proof tree with this goal as its head. $\diamond$

An infinite proof tree is a sequence of proof trees such that each member of the sequence is obtained from its immediate predecessor by expanding open goals.

Definition 3.3 (Prefixes of Proof Trees, Infinite Proof Trees) *A proof tree* T *is a* prefix *of a proof tree* T'*, written* $T \subset T'$*, if there are one or more open goals* $G_1, \ldots, G_n$ *in* T *such that* T' *is obtained from* T *by replacing each* G_i *by a derivation tree* T_i *with* $\mathrm{head}(T_i) = G_i$*. An* infinite proof tree *is an infinite sequence* $\{T_i\}_{i \geq 0}$ *of finite proof trees such that* $T_i \subset T_{i+1}$ *for all* $i \geq 0$*.*

The notion of *well-formed proof trees* captures the operational behavior of a rewrite engine that evaluates the conditions of a rewrite rule from left to right.

Definition 3.4 (Well-formed Proof Trees) *A proof tree* T *is* well-formed *if it is either an open goal, a closed proof tree, or a derivation tree of the form*

$$\frac{T_1 \qquad \cdots \qquad T_n}{G}(\Delta)$$

where T_j *is a well-formed proof tree for all* $1 \leq j \leq n$ *and there is an* $i \leq n$ *such that* T_i *is not closed,* T_j *is closed for all* $j < i$*, and* T_k *is an open goal for all* $k > i$*. An infinite proof tree is* well-formed *if it consists of well-formed proof trees.*

As mentioned above, *operational termination* is characterized by the absence of infinite well-formed proof trees.

Definition 3.5 (Operational Termination) *A CCES* $(\mathcal{R}, \mathcal{S}, \mathcal{E})$ *is* operationally terminating *iff there are no infinite well-formed proof trees.*

It can be shown that the notions of termination and operational termination coincide for unconditional systems [8].

Lemma 3.6 *Let* $(\mathcal{R}, \mathcal{S}, \mathcal{E})$ *be a CES. Then* $(\mathcal{R}, \mathcal{S}, \mathcal{E})$ *is operationally terminating iff* $(\mathcal{R}, \mathcal{S}, \mathcal{E})$ *is terminating.*

4 Elimination of Conditions

In order to show operational termination of a CCES $(\mathcal{R}, \mathcal{S}, \mathcal{E})$, we transform it into a CES $(\mathcal{U}(\mathcal{R}), \mathcal{S}, \mathcal{E})$ such that operational termination of $(\mathcal{R}, \mathcal{S}, \mathcal{E})$ is implied by operational termination of $(\mathcal{U}(\mathcal{R}), \mathcal{S}, \mathcal{E})$. We then check for termination of $(\mathcal{U}(\mathcal{R}), \mathcal{S}, \mathcal{E})$,

which, by Lemma 3.6, is equivalent to operational termination of $(\mathcal{U}(\mathcal{R}), \mathcal{S}, \mathcal{E})$. The transformation generalizes the classical one for ordinary conditional rewriting (see, e.g., [13, Definition 7.2.48]) to rewriting with equations, normalization, and constraints. An extension of the classical transformation to context-sensitive rewriting with equations was proposed in [5]. Our presentation is influenced by that paper.

Definition 4.1 (Transformation $\mathcal{U}$) *Let $\rho : s_1 \to^* t_1, \ldots, s_n \to^* t_n \mid l \to r[\![C]\!]$ be a conditional constrained rewrite rule. Then $\mathcal{U}(\rho)$ is defined by*

$$\begin{aligned}
\textit{if } n = 0 \textit{ then } \mathcal{U}(\rho) &= \{ \rho \} \\
\textit{if } n > 0 \textit{ then } \mathcal{U}(\rho) &= \{ l \to U_1^\rho(s_1, x_1^*)[\![C]\!] \} \cup & (1)\\
&\quad \{ U_{i-1}^\rho(t_{i-1}, x_{i-1}^*) \to U_i^\rho(s_i, x_i^*)[\![C]\!] \mid 2 \le i \le n \} \cup & (2)\\
&\quad \{ U_n^\rho(t_n, x_n^*) \to r[\![C]\!] \} & (3)
\end{aligned}$$

Here, the U_i^ρ are fresh function symbols and, for $1 \le i \le n$, the expression x_i^* denotes the sorted list of variables in the set $\mathcal{V}(l) \cup \mathcal{V}(t_1) \cup \ldots \cup \mathcal{V}(t_{i-1})$ according to some fixed order on the set $\mathcal{V}$ of all variables. For a finite set $\mathcal{R}$ of conditional constrained rewrite rules we let $\mathcal{U}(\mathcal{R}) = \bigcup_{\rho \in \mathcal{R}} \mathcal{U}(\rho)$.

Example 4.2 *Continuing Examples 1.1, 2.8, and 3.2 we get the following unconditional constrained rewrite rules.*

$$\begin{aligned}
\mathsf{app}(\mathsf{nil}, zs) &\to zs \\
\mathsf{app}(\mathsf{cons}(x, ys), zs) &\to \mathsf{cons}(x, \mathsf{app}(ys, zs)) \\
\mathsf{split}(x, \emptyset) &\to \langle \emptyset, \emptyset \rangle \\
\mathsf{split}(x, \mathsf{ins}(y, zs)) &\to \mathsf{U}_1(\mathsf{split}(x, zs), x, y, zs) \quad [\![x > y]\!] \\
\mathsf{U}_1(\langle zl, zh \rangle, x, y, zs) &\to \langle \mathsf{ins}(y, zl), zh \rangle \quad [\![x > y]\!] \\
\mathsf{split}(x, \mathsf{ins}(y, zs)) &\to \mathsf{U}_2(\mathsf{split}(x, zs), x, y, zs) \quad [\![x \not> y]\!] \\
\mathsf{U}_2(\langle zl, zh \rangle, x, y, zs) &\to \langle zl, \mathsf{ins}(y, zh) \rangle \quad [\![x \not> y]\!] \\
\mathsf{qsort}(\emptyset) &\to \mathsf{nil} \\
\mathsf{qsort}(\mathsf{ins}(x, ys)) &\to \mathsf{U}_3(\mathsf{split}(x, ys), x, ys) \\
\mathsf{U}_3(\langle yl, yh \rangle, x, ys) &\to \mathsf{app}(\mathsf{qsort}(yl), \mathsf{cons}(x, \mathsf{qsort}(yh)))
\end{aligned}$$

In order to ease readability we used simplified names for the function symbols U_i^ρ from Definition 4.1. Termination of this system is shown in [6, Appendix D.3]. $\Diamond$

In order to show that $(\mathcal{R}, \mathcal{S}, \mathcal{E})$ is operationally terminating if $(\mathcal{U}(\mathcal{R}), \mathcal{S}, \mathcal{E})$ is operationally terminating we make use of the following lemma.

Lemma 4.3 *For any well-formed proof tree T for $(\mathcal{R}, \mathcal{S}, \mathcal{E})$ whose head goal is either $s \to t$ or $s \to^* t$, there exists a well-formed proof tree $\beta(T)$ for $(\mathcal{U}(\mathcal{R}), \mathcal{S}, \mathcal{E})$ whose head goal is $s \to^* t$. Furthermore, if $T \subset T'$ for some T', then $\beta(T) \subset \beta(T')$.*

Before proving Lemma 4.3 we make two preliminary remarks about proof trees. These properties will be used freely in the following.

Property 4.4 *Given the proof tree*

$$\frac{T_1 \qquad \cdots \qquad T_n}{s \to t}$$

and a term $s' \sim_{\mathcal{E} \cup \mathcal{PA}} s$*, it is possible to construct the proof tree*

$$\frac{T_1 \qquad \cdots \qquad T_n}{s' \to t'}$$

where $t' \sim_{\mathcal{E} \cup \mathcal{PA}} t$ *is given by Lemma 2.9.* □

Property 4.5 *Given the proof tree*

$$\cfrac{\cfrac{T_1}{s_0 \to s_1} \qquad \cfrac{\cfrac{T_2}{s_1 \to s_2} \qquad \cfrac{\vdots}{} \quad \cfrac{\cfrac{T_n}{s_{n-1} \to s_n} \qquad \cfrac{}{s_n \to^* t}\,(\mathsf{Refl})}{}\,(\mathsf{Tran})}{s_1 \to^* t}\,(\mathsf{Tran})}{s \to^* t}\,(\mathsf{Tran})$$

with $s_0 = s$ *and a term* $s' \sim_{\mathcal{E} \cup \mathcal{PA}} s$*, it is possible to construct the proof tree*

$$\cfrac{\cfrac{T_1}{\widetilde{s_0} \to \widetilde{s_1}} \qquad \cfrac{\cfrac{T_2}{\widetilde{s_1} \to \widetilde{s_2}} \qquad \cfrac{\vdots}{} \quad \cfrac{\cfrac{T_n}{\widetilde{s_{n-1}} \to \widetilde{s_n}} \qquad \cfrac{}{\widetilde{s_n} \to^* t}\,(\mathsf{Refl})}{}\,(\mathsf{Tran})}{\widetilde{s_1} \to^* t}\,(\mathsf{Tran})}{s' \to^* t}\,(\mathsf{Tran})$$

where $\widetilde{s_0} = s'$ *and* $\widetilde{s_i} \sim_{\mathcal{E} \cup \mathcal{PA}} s_i$ *for all* $0 \leq i \leq n$*. Here, the* $\widetilde{s_i}$ *are given by Lemma 2.9. Notice that* $\widetilde{s_n} \sim_{\mathcal{E} \cup \mathcal{PA}} t$ *since* $\widetilde{s_n} \sim_{\mathcal{E} \cup \mathcal{PA}} s_n$ *and* $s_n \sim_{\mathcal{E} \cup \mathcal{PA}} t$*.* □

Proof of Lemma 4.3

Assume that T is a well-formed proof tree for $(\mathcal{R}, \mathcal{S}, \mathcal{E})$ whose head goal is either $s \to t$ or $s \to^* t$. The construction of $\beta(T)$ is done by induction on the structure of T. There are two cases, depending on whether the head goal of T is of the form $s \to^* t$ or $s \to t$.

I. The head goal is $s \to^* t$:
If the inference rule (Refl) is applied to $s \to^* t$ then we are immediately done. Otherwise, the inference rule (Tran) is applied to $s \to^* t$. First, we assume that T is closed. Then, T has the shape

$$\cfrac{\cfrac{T_1}{s_0 \to s_1} \qquad \cfrac{\cfrac{T_2}{s_1 \to s_2} \qquad \cfrac{\vdots}{} \quad \cfrac{\cfrac{T_n}{s_{n-1} \to s_n} \qquad \cfrac{}{s_n \to^* t}\,(\mathsf{Refl})}{}\,(\mathsf{Tran})}{s_1 \to^* t}\,(\mathsf{Tran})}{s \to^* t}\,(\mathsf{Tran})$$

where $s_0 = s$ and $s_n \sim_{\mathcal{E}\cup\mathcal{PA}} t$. By the inductive hypothesis we can assume that each subtree

$$U_i = \frac{T_i}{s_{i-1} \to s_i}$$

has a transformed tree $\beta(U_i)$ of the form

$$\cfrac{\cfrac{T_i^1}{s_{i-1} \to s_i^1} \qquad \cfrac{\cfrac{T_i^2}{s_i^1 \to s_i^2} \qquad \cfrac{\vdots}{} \qquad \cfrac{\cfrac{T_i^{k_i}}{s_i^{k_i-1} \to s_i^{k_i}} \qquad \cfrac{}{s_i^{k_i} \to^* s_i}\,(\mathsf{Refl})}{}\,(\mathsf{Tran})}{s_i^1 \to^* s_i}\,(\mathsf{Tran})}{s_{i-1} \to^* s_i}\,(\mathsf{Tran})$$

The proof tree $\beta(T)$ is now built by suitably "gluing" the $\beta(U_i)$ together.

$$\cfrac{\cfrac{T_1^1}{\widetilde{s_0} \to \widetilde{s_1^1}} \qquad \cfrac{\cfrac{T_1^2}{\widetilde{s_1^1} \to \widetilde{s_1^2}} \qquad \vdots \qquad \cfrac{\cfrac{T_1^{k_1}}{\widetilde{s_1^{k_1-1}} \to \widetilde{s_1^{k_1}}} \qquad \cfrac{\cfrac{T_2^1}{\widetilde{s_1^{k_1}} \to \widetilde{s_2^1}} \qquad \vdots \qquad \cfrac{\cfrac{T_n^1}{\widetilde{s_{n-1}^{k_n}} \to \widetilde{s_n^1}} \qquad \cfrac{\cfrac{T_n^2}{\widetilde{s_n^1} \to \widetilde{s_n^2}} \qquad \vdots \qquad \cfrac{\cfrac{T_n^{k_n}}{\widetilde{s_n^{k_n-1}} \to \widetilde{s_n^{k_n}}} \qquad \cfrac{}{\widetilde{s_n^{k_n}} \to^* t}\,(\mathsf{Refl})}{\widetilde{s_n^{k_n-1}} \to^* t}\,(\mathsf{Tran})}{\widetilde{s_n^1} \to^* t}\,(\mathsf{Tran})}{\widetilde{s_{n-1}^{k_n}} \to^* t}\,(\mathsf{Tran})}{\widetilde{s_1^{k_1}} \to^* t}\,(\mathsf{Tran})}{\widetilde{s_1^{k_1-1}} \to^* t}\,(\mathsf{Tran})}{\widetilde{s_1^1} \to^* t}\,(\mathsf{Tran})}{s \to^* t}\,(\mathsf{Tran})$$

If T is not closed since some leftmost T_j^i is not closed, then $\beta(T)$ needs to be cut at the level of T_j^i. In either case, $\beta(T)$ is a well-formed proof tree if T is well-formed and $\beta(T) \subset \beta(T')$ if $T \subset T'$.

II. The head goal is $s \to t$:
Again, we first assume that T is closed. Then, it has the shape

$$\cfrac{\cfrac{S_1}{s_1\sigma \to^* t_1\sigma} \qquad \cdots \qquad \cfrac{S_n}{s_n\sigma \to^* t_n\sigma}}{s \to t}\,(\mathsf{Repl})$$

for some rule $\rho : s_1 \to^* t_1, \ldots, s_n \to^* t_n \mid l \to r[\![C]\!]$ from $\mathcal{R}$. In order to ease notation, we assume that the position in the (Repl) rule is $p = \lambda$, i.e., $s \xrightarrow{!}{}^{>\lambda}_{\mathcal{PA}\|\mathcal{E}\backslash\mathcal{S}} \circ \sim_{\mathcal{E}\cup\mathcal{PA}} l\sigma$ and $t = r\sigma$. If the constrained rewrite rule that is used is

unconditional, then this rule is also present in $\mathcal{U}(\mathcal{R})$ and we obtain the following proof tree for $(\mathcal{U}(\mathcal{R}), \mathcal{S}, \mathcal{E})$:

$$\dfrac{\dfrac{}{s \to t}\,\textsf{(Repl)} \qquad \dfrac{}{t \to^* t}\,\textsf{(Refl)}}{s \to^* t}\,\textsf{(Tran)}$$

Otherwise, $\mathcal{U}(\mathcal{R})$ contains rules of the form (1), (2) and (3) from Definition 4.1. We construct proof trees for $(\mathcal{U}(\mathcal{R}), \mathcal{S}, \mathcal{E})$ with the following head goals:

$$\begin{array}{rcll}
U_n^\rho(t_n, x_n^*)\sigma & \to^* & r\sigma & (G_n)\\
U_n^\rho(s_n, x_n^*)\sigma & \to^* & r\sigma & (H_n)\\
U_{n-1}^\rho(t_{n-1}, x_{n-1}^*)\sigma & \to^* & r\sigma & (G_{n-1})\\
U_{n-1}^\rho(s_{n-1}, x_{n-1}^*)\sigma & \to^* & r\sigma & (H_{n-1})\\
 & \vdots & & \\
U_1^\rho(t_1, x_1^*)\sigma & \to^* & r\sigma & (G_1)\\
U_1^\rho(s_1, x_1^*)\sigma & \to^* & r\sigma & (H_1)\\
s & \to^* & t & (K)
\end{array}$$

For the following, notice that $C\sigma$ is $\mathcal{PA}$-valid by assumption.

(i) Proof tree for (G_n): We can construct the proof tree

$$\dfrac{\dfrac{}{U_n^\rho(t_n, x_n^*)\sigma \to r\sigma}\,\textsf{(Repl)} \qquad \dfrac{}{r\sigma \to^* r\sigma}\,\textsf{(Refl)}}{U_n^\rho(t_n, x_n^*)\sigma \to^* r\sigma}\,\textsf{(Tran)}$$

using rule (3) from Definition 4.1.

(ii) Proof tree for (H_k) using the proof tree for (G_k): We assume that we have already constructed a proof tree T_k for the goal $(G_k) = U_k^\rho(t_k\sigma, x_k^*\sigma) \to^* r\sigma$. By induction on the tree structure, we can assume that the subtree

$$P_k = \dfrac{S_k}{s_k\sigma \to^* t_k\sigma}$$

has a transformed tree $\beta(P_k)$ of the form

$$\dfrac{\dfrac{T_k^1}{u_0 \to u_1} \qquad \dfrac{\dfrac{T_k^2}{u_1 \to u_2} \qquad \dfrac{\vdots \quad \dfrac{\dfrac{T_k^l}{u_{l-1} \to u_l} \qquad \dfrac{}{u_l \to^* t_k\sigma}\,\textsf{(Refl)}}{\;}\,\textsf{(Tran)}}{\;}}{u_1 \to^* t_k\sigma}\,\textsf{(Tran)}}{s_k\sigma \to^* t_k\sigma}\,\textsf{(Tran)}$$

where $u_0 = s_k\sigma$ and $u_l \sim_{\mathcal{E}\cup\mathcal{PA}} t_k\sigma$. Then, we can construct a proof tree for

the goal $U^\rho_k(s_k\sigma, x^*_k\sigma) \to^* r\sigma$ as follows:

$$\dfrac{\dfrac{T^1_k}{u'_0 \to u'_1} \quad \dfrac{\dfrac{T^2_k}{u'_1 \to u'_2} \quad \dfrac{\vdots \quad \dfrac{\dfrac{T^l_k}{u'_{l-1} \to u'_l} \quad \dfrac{T_k}{U^\rho_k(u_l, x^*_k\sigma) \to^* r\sigma}}{u'_{l-1} \to^* r\sigma}\,(\mathsf{Tran})}{}}{u'_1 \to^* r\sigma}\,(\mathsf{Tran})}{U^\rho_k(s_k\sigma, x^*_k\sigma) \to^* r\sigma}\,(\mathsf{Tran})$$

where $u'_i = U^\rho_k(u_i, x^*_k\sigma)$.

(iii) Proof tree for (G_{k-1}) using the proof tree for (H_k): We assume that we have already constructed a proof tree T_k for the goal $(H_k) = U^\rho_k(s_k\sigma, x^*_k\sigma) \to^* r\sigma$. Then, we can construct a proof tree for the goal $U^\rho_{k-1}(t_{k-1}\sigma, x^*_{k-1}\sigma) \to^* r\sigma$ as follows:

$$\dfrac{\dfrac{}{U^\rho_{k-1}(t_{k-1}\sigma, x^*_{k-1}\sigma) \to U^\rho_k(s_k\sigma, x^*_k\sigma)}\,(\mathsf{Repl}) \qquad T_k}{U^\rho_{k-1}(t_{k-1}\sigma, x^*_{k-1}\sigma) \to^* r\sigma}\,(\mathsf{Tran})$$

where the (Repl) step uses rule (2) from Definition 4.1.

(iv) Proof tree for (K) using the proof tree for (H_1): We assume that we have already constructed a proof tree T_1 for the goal $(H_1) = U^\rho_1(s_1\sigma, x^*_1\sigma) \to^* r\sigma$. Then, we can construct a proof tree for the goal $s \to^* t$ as follows:

$$\dfrac{\dfrac{}{s \to U^\rho_1(s_1\sigma, x^*_1\sigma)}\,(\mathsf{Repl}) \qquad T_1}{s \to^* t}\,(\mathsf{Tran})$$

where the (Repl) step uses rule (1) from Definition 4.1.

As in case I., if the original proof tree is not closed, then the transformed tree is cut at some level. In either case, $\beta(T)$ is well-formed if T is well-formed and $\beta(T) \subset \beta(T')$ if $T \subset T'$. □

Lemma 4.3 now easily implies the following result.

Theorem 4.6 *$(\mathcal{R}, \mathcal{S}, \mathcal{E})$ is operationally terminating if $(\mathcal{U}(\mathcal{R}), \mathcal{S}, \mathcal{E})$ is operationally terminating.*

In combination with Lemma 3.6 we get the key result of the present paper.

Corollary 4.7 *$(\mathcal{R}, \mathcal{S}, \mathcal{E})$ is operationally terminating if $(\mathcal{U}(\mathcal{R}), \mathcal{S}, \mathcal{E})$ is terminating.*

Example 4.8 *The following CCES specifies the sieve of Eratosthenes.* $\mathsf{primes}(x)$ *returns a list containing the prime numbers up to x. In this example we have $\mathcal{S} = \mathcal{E} = \emptyset$.*

$$\mathsf{primes}(x) \to \mathsf{sieve}(\mathsf{nats}(2, x))$$

$$\mathsf{nats}(x, y) \to \mathsf{nil} \qquad [\![x > y]\!]$$

$$\begin{aligned}
\mathsf{nats}(x,y) &\to \mathsf{cons}(x,\mathsf{nats}(x+1,y))\ [\![x \not> y]\!]\\
\mathsf{sieve}(\mathsf{nil}) &\to \mathsf{nil}\\
\mathsf{sieve}(\mathsf{cons}(x,ys)) &\to \mathsf{cons}(x,\mathsf{sieve}(\mathsf{filter}(x,ys)))\\
\mathsf{filter}(x,\mathsf{nil}) &\to \mathsf{nil}\\
\mathsf{isdiv}(x,y) \to^* \mathsf{true}\ \mid\ \mathsf{filter}(x,\mathsf{cons}(y,zs)) &\to \mathsf{filter}(x,zs)\\
\mathsf{isdiv}(x,y) \to^* \mathsf{false}\ \mid\ \mathsf{filter}(x,\mathsf{cons}(y,zs)) &\to \mathsf{cons}(y,\mathsf{filter}(x,zs))\\
\mathsf{isdiv}(x,0) &\to \mathsf{true} & [\![x > 0]\!]\\
\mathsf{isdiv}(x,y) &\to \mathsf{false} & [\![x > y \wedge y > 0]\!]\\
\mathsf{isdiv}(x,x+y) &\to \mathsf{isdiv}(x,y) & [\![x > 0]\!]
\end{aligned}$$

Using Definition 4.1 we obtain the following $\mathcal{U}(\mathcal{R})$.

$$\begin{aligned}
\mathsf{primes}(x) &\to \mathsf{sieve}(\mathsf{nats}(2,x))\\
\mathsf{nats}(x,y) &\to \mathsf{nil} & [\![x > y]\!]\\
\mathsf{nats}(x,y) &\to \mathsf{cons}(x,\mathsf{nats}(x+1,y)) & [\![x \not> y]\!]\\
\mathsf{sieve}(\mathsf{nil}) &\to \mathsf{nil}\\
\mathsf{sieve}(\mathsf{cons}(x,ys)) &\to \mathsf{cons}(x,\mathsf{sieve}(\mathsf{filter}(x,ys)))\\
\mathsf{filter}(x,\mathsf{nil}) &\to \mathsf{nil}\\
\mathsf{filter}(x,\mathsf{cons}(y,zs)) &\to \mathsf{U}_1(\mathsf{isdiv}(x,y),x,y,zs)\\
\mathsf{U}_1(\mathsf{true},x,y,zs) &\to \mathsf{filter}(x,zs)\\
\mathsf{filter}(x,\mathsf{cons}(y,zs)) &\to \mathsf{U}_2(\mathsf{isdiv}(x,y),x,y,zs)\\
\mathsf{U}_2(\mathsf{false},x,y,zs) &\to \mathsf{cons}(y,\mathsf{filter}(x,zs))\\
\mathsf{isdiv}(x,0) &\to \mathsf{true} & [\![x > 0]\!]\\
\mathsf{isdiv}(x,y) &\to \mathsf{false} & [\![x > y \wedge y > 0]\!]\\
\mathsf{isdiv}(x,x+y) &\to \mathsf{isdiv}(x,y) & [\![x > 0]\!]
\end{aligned}$$

By Corollary 4.7 the CCES $(\mathcal{R},\emptyset,\emptyset)$ is operationally terminating if the unconditional CES $(\mathcal{U}(\mathcal{R}),\emptyset,\emptyset)$ is terminating. Termination of $(\mathcal{U}(\mathcal{R}),\emptyset,\emptyset)$ is shown in [6, Appendix F.2]. ◇

5 Conclusions and Future Work

We have presented conditional constrained equational rewrite systems for specifying algorithms. Rewriting with these systems is based on normalized equational rewriting combined with evaluation of conditions and validity checking of instantiated constraints. Semantic data structures like finite sets, multisets, and sorted lists are modeled using constructor rules and equations. Natural numbers are built-in and

constraints are taken from Presburger arithmetic.

We have shown that operational termination of such conditional systems can be reduced to termination of unconditional systems using a syntactic transformation. Powerful methods based on dependency pairs for showing termination of unconditional systems are presented in [9]. These methods can thus be used for showing operational termination of conditional systems as well. Using this approach, operational termination of several nontrivial conditional systems is shown in [6].

We will next study properties apart from operational termination. In particular, we will investigate confluence and sufficient completeness. Orthogonal to this, we plan to generalize the rewrite relation by considering other built-in theories, most importantly integers instead of natural numbers.

References

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

[2] Bertling, H. and H. Ganzinger, *Completion-time optimization of rewrite-time goal solving*, in: N. Dershowitz, editor, *Proceedings of the 3rd Conference on Rewriting Techniques and Applications (RTA '89)*, Lecture Notes in Computer Science **355** (1989), pp. 45–58.

[3] Blanqui, F., T. Hardin and P. Weis, *On the implementation of construction functions for non-free concrete data types*, in: R. D. Nicola, editor, *Proceedings of the 16th European Symposium on Programming (ESOP '07)*, Lecture Notes in Computer Science **4421** (2007), pp. 95–109.

[4] Clavel, M., F. Durán, S. Eker, Patrick, Lincoln, N. Martí-Oliet, J. Meseguer and C. Talcott, "All About Maude–A High-Performance Logical Framework," Lecture Notes in Computer Science **4350**, Springer-Verlag, 2007.

[5] Durán, F., S. Lucas, C. Marché, J. Meseguer and X. Urbain, *Proving operational termination of membership equational programs*, Higher-Order and Symbolic Computation **21** (2008), pp. 59–88.

[6] Falke, S. and D. Kapur, *Dependency pairs for rewriting with built-in numbers and semantic data structures*, Technical Report TR-CS-2007-21, Department of Computer Science, University of New Mexico (2007), available at http://www.cs.unm.edu/research/tech-reports/.

[7] Falke, S. and D. Kapur, *Dependency pairs for rewriting with non-free constructors*, in: F. Pfenning, editor, *Proceedings of the 21st Conference on Automated Deduction (CADE '07)*, Lecture Notes in Artificial Intelligence **4603** (2007), pp. 426–442.

[8] Falke, S. and D. Kapur, *Operational termination of conditional rewriting with built-in numbers and semantic data structures*, Technical Report TR-CS-2007-22, Department of Computer Science, University of New Mexico (2007), available at http://www.cs.unm.edu/research/tech-reports/.

[9] Falke, S. and D. Kapur, *Dependency pairs for rewriting with built-in numbers and semantic data structures*, in: A. Voronkov, editor, *Proceedings of the 19th Conference on Rewriting Techniques and Applications (RTA '08)*, Lecture Notes in Computer Science **5117** (2008), pp. 94–109.

[10] Giesl, J. and D. Kapur, *Dependency pairs for equational rewriting*, in: A. Middeldorp, editor, *Proceedings of the 12th Conference on Rewriting Techniques and Applications (RTA '01)*, Lecture Notes in Computer Science **2051** (2001), pp. 93–108.

[11] Lucas, S., C. Marché and J. Meseguer, *Operational termination of conditional term rewriting systems*, Information Processing Letters **95** (2005), pp. 446–453.

[12] Marché, C., *Normalized rewriting: An alternative to rewriting modulo a set of equations*, Journal of Symbolic Computation **21** (1996), pp. 253–288.

[13] Ohlebusch, E., "Advanced Topics in Term Rewriting," Springer-Verlag, 2002.

[14] Peterson, G. E. and M. E. Stickel, *Complete sets of reductions for some equational theories*, Journal of the ACM **28** (1981), pp. 233–264.

[15] Presburger, M., *Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt*, in: *Comptes Rendus du Premier Congrès de Mathématiciens des Pays Slaves*, 1929, pp. 92–101.

Completion as Post-Process in Program Inversion of Injective Functions

Naoki Nishida [1] Masahiko Sakai [2]

Graduate School of Information Science, Nagoya University
Nagoya, Japan

Abstract

Given a constructor term rewriting system defining injective functions, the inversion compiler in [19,20] generates a confluent conditional term rewriting system defining completely the inverse relations of the injective functions, and then the compiler unravels the conditional system into an unconditional term rewriting system. In general, the unconditional system is not confluent and thus not computationally equivalent to the conditional system. In this paper, we propose a modification of Knuth-Bendix completion procedure as a post-process of the inversion compiler. Given a confluent and operationally terminating conditional system, the procedure takes the unraveled one of the conditional system as input, and it returns a convergent system that is computationally equivalent to the conditional system if it halts successfully. We also adapt the modification to the conditional systems that are not confluent but innermost-confluent. The implementation of our method succeeds in generating innermost-convergent inverse systems for all examples shown by Kawabe et al. where all main and axillary functions are injective.

Keywords: unraveling, convergence, functional programming, conditional term rewriting system

1 Introduction

Given a constructor TRS (term rewriting system), the inversion compiler proposed in [19,20] first generates a CTRS (conditional TRS) as an intermediate result, and then transforms the CTRS into a TRS that is equivalent to the CTRS with respect to *inverse computation*. The first phase of the compiler is *local inversion*; for every constructor TRS, the first phase generates a CTRS, called an *inverse system*, that completely represents the inverse relation of the reduction relation represented by the constructor TRS. The second phase employs (a variant of) Ohlebusch's *unraveling* [21]. *Unravelings* are transformations based on Marchiori's approach [14], that transform CTRSs into TRSs. Note that we call all variants of Marchiori's unravelings *unravelings* because they satisfy the condition [14,15] of being unravelings.

Unfortunately, the compiler cannot always generate TRSs that are computationally equivalent to the intermediate CTRSs due to a character of unravelings, that

[1] Email: nishida@is.nagoya-u.ac.jp
[2] Email: sakai@is.nagoya-u.ac.jp

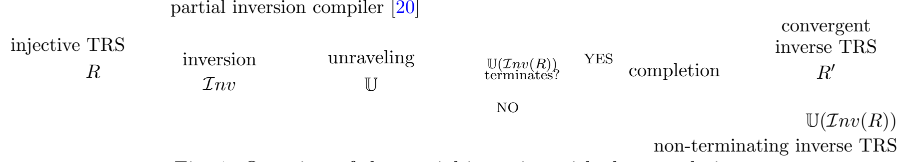


Fig. 1. Overview of the partial inversion with the completion.

is, the unraveled TRSs of CTRSs may have unexpected normal forms that represent dead ends of wrong choices at branches of evaluating conditional parts of the CTRSs. These wrong choices are captured by critical pairs of the unraveled TRSs, each of which originates two rewrite rules corresponding to the 'correct' and 'wrong' choices, where any rules looking like 'wrong choice' must be necessary elsewhere. Note that it is decidable whether a normal form is desired or unexpected: a normal form of the unraveled TRSs is an unexpected one if it contains an extra defined symbol introduced by the unraveling.

In program inversion by the compiler, this problem arises even if functions defined in given constructor TRSs are injective. For this reason, the resultant TRSs do not define functions and thus the inversion compiler is less applicable to injective functions in practical functional programming languages — it is easy to translate the functional programs to constructor TRSs, but difficult to translate the resultant TRSs of the compiler back into functional programs.

In this paper, we propose a modification of the *Knuth-Bendix completion procedure* in order to transform the unraveled TRSs of confluent and operationally terminating CTRSs into convergent (and possibly non-overlapping) TRSs that are computationally equivalent to the CTRSs. Unfortunately, the procedure does not always halt. However, if the procedure halts successfully and the resultant convergent TRSs are non-overlapping, then the resultant systems can be easily translated back into functional programs due to non-overlappingness. We takes the modified completion procedure as a post-process into the program inversion of injective functions (Fig. 1 and Section 4). Through this approach, we show that unravelings are useful not only in analyzing properties but also in modifying programs (unraveled TRSs, especially inverse programs).

Consider the following functional program in Standard ML where $\mathsf{Snoc}(xs, y)$ produces the list obtained from xs by adding y as the last element:

```
fun Snoc( [], y ) = [y]
  | Snoc( (x::xs), y ) = x :: Snoc( xs, y );
```

We can easily translate the above program into the following constructor TRS:

$$R_1 = \{\ \mathsf{Snoc}(\mathsf{nil}, y) \to (y :: \mathsf{nil}),\quad \mathsf{Snoc}((x :: xs), y) \to (x :: \mathsf{Snoc}(xs, y))\ \}$$

where $(t :: ts)$ abbreviates the list $\mathsf{cons}(t, ts)$. The compiler inverts R_1 into the following CTRS in the first phase [3]:

$$\mathcal{I}nv(R_1) = \begin{cases} \mathsf{InvSnoc}([y]) \to \langle \mathsf{nil}, y\rangle \\ \mathsf{InvSnoc}((x :: ys)) \to \langle (x :: xs), y\rangle \Leftarrow \mathsf{InvSnoc}(ys) \to \langle xs, y\rangle \end{cases}$$

[3] To simplify discussions, we omit describing special rules in the form of $\mathsf{Inv}F(F(x_1, \ldots, x_n)) \to \langle x_1, \ldots, x_n\rangle$ [20,19] because they are meaningless for inverse computation in dealing with call-by-value systems. The special rules are necessary only for inverse computation of normalizing computation in term rewriting.

where $[t_1, t_2, \ldots, t_n]$ abbreviates the list $\mathsf{cons}(t_1, \mathsf{cons}(t_2, \cdots, \mathsf{cons}(t_n, \mathsf{nil}) \cdots))$ and each tuple of n terms $t_1, \ldots, t_n$ is denoted by $\langle t_1, \ldots, t_n \rangle$ that can be represented as terms by introducing an n-ary constructor. The compiler unravels the CTRS $\mathcal{I}nv(R_1)$ into the following TRS in the second phase:

$$\mathbb{U}(\mathcal{I}nv(R_1)) = \left\{ \begin{array}{c} \mathsf{InvSnoc}([y]) \to \langle \mathsf{nil}, y \rangle, \\ \mathsf{InvSnoc}((x :: ys)) \to U_1(\mathsf{InvSnoc}(ys), x, ys), \\ U_1(\langle xs, y \rangle, x, ys) \to \langle (x :: xs), y \rangle \end{array} \right\}$$

The introduced symbol U_1 is used for evaluating the conditional part $\mathsf{InvSnoc}(ys) \to \langle xs, y \rangle$ of the second rule in $\mathcal{I}nv(R_1)$. The term $\mathsf{Snoc}([\mathsf{a}, \mathsf{b}], \mathsf{c})$ has a unique normal form $[\mathsf{a}, \mathsf{b}, \mathsf{c}]$ but $\mathsf{InvSnoc}([\mathsf{a}, \mathsf{b}, \mathsf{c}])$ has two normal forms, a solution $\langle [\mathsf{a}, \mathsf{b}], \mathsf{c} \rangle$ of inverse computation and an unexpected normal form $U_1(U_1(U_1(\mathsf{InvSnoc}(\mathsf{nil}), \mathsf{c}, \mathsf{nil}), \mathsf{b}, [\mathsf{c}]), \mathsf{a}, [\mathsf{b}, \mathsf{c}])$. In this example, it appears to be easy to translate from the CTRS $\mathcal{I}nv(R_1)$ into a functional program directly because we can easily determine an appropriate priority of conditional rules in $\mathcal{I}nv(R_1)$. However, such a direct translation is difficult in general because we cannot decide which rules have priority of the application to terms. The restricted compiler in [1] is useless for this case because R_1 is out of the scope. It is probably impossible that one transforms input systems into equivalent systems from which the compiler generates the inverse systems without overlapping. To avoid this problem, it has been shown in [18] that the transformation in [24] is suitable as the second phase of the compiler, in the sense of producing convergent systems. However, the generated systems contain some special symbols and *overlapping* rules. For this reason, it is difficult to translate the convergent but *overlapping* TRS into a functional program (see Section 5).

Roughly speaking, non-confluence of $\mathbb{U}(\mathcal{I}nv(R_1))$ comes from the critical pair $\langle \langle \mathsf{nil}, x \rangle, U_1(\mathsf{InvSnoc}(\mathsf{nil}), x, \mathsf{nil}) \rangle$ between the first and second rules in $\mathbb{U}(\mathcal{I}nv(R_1))$. In this case, the application of the first rule is the correct choice and that of the second is the wrong, that is, $\langle \mathsf{nil}, x \rangle$ is the correct result and $U_1(\mathsf{InvSnoc}(\mathsf{nil}), x, \mathsf{nil})$ is the wrong recursive call of U_1 containing the dead end $\mathsf{InvSnoc}(\mathsf{nil})$. From this observation, by adding the rule $U_1(\mathsf{InvSnoc}(\mathsf{nil}), x, \mathsf{nil}) \to \langle \mathsf{nil}, x \rangle$, the unexpected normal form of $\mathsf{InvSnoc}([\mathsf{a}, \mathsf{b}, \mathsf{c}])$ can be reduced to the solution. This added rule provides a path from the wrong branch of inverse computation to the correct branch. Due to this rule, the new TRS is confluent. This process just corresponds to the behavior of *completion*. Therefore, *completion* is expected to solve the non-confluence of TRSs obtained by the inversion compiler.

This paper illustrates all of the following:

- under the *call-by-value* evaluation of operationally terminating deterministic CTRSs, *simulation-soundness* on the innermost reduction is preserved by Ohlebusch's unraveling (Subsection 3.2);
- given a (innermost-)confluent and operationally terminating CTRS, the completion procedure takes the unraveled TRS (evaluated by the innermost reduction) as input, and returns a (innermost-)convergent TRSs that are computationally equivalent to the CTRSs if the procedure halts successfully (Subsection 3.3);

- to deal with TRSs whose termination is not provable by any reduction orders without dependency analysis (e.g., lexicographic path orders), we employ the completion with termination provers, following the approach in [28] (Subsection 3.5).

We also show that an implementation of the completion procedure succeeds in generating convergent TRSs from all the unraveled TRSs of CTRSs obtained by the inversion compiler [19] from injective functions shown by Kawabe et al. [8] where all axillary functions are also injective, and we show an informal translation of the *non-overlapping* TRSs obtained by the procedure into functional programs (Subsection 3.4). Note that we do not consider *sorts*; however, the framework in this paper is easily extended to many-sorted systems.

2 Preliminaries

Here, we will review the following basic notations of term rewriting [2,22].

Throughout this paper, we use $\mathcal{V}$ as a countably infinite set of *variables*. The set of all *terms* over a *signature* $\mathcal{F}$ and $\mathcal{V}$ is denoted by $\mathcal{T}(\mathcal{F},\mathcal{V})$. The set of all variables appearing in either of terms $t_1,\ldots,t_n$ is represented by $\mathcal{V}ar(t_1,\ldots,t_n)$. The *identity* of terms s and t is denoted by $s \equiv t$. For a term t and a position p of t, the notation $t|_p$ represents the subterm of t at p. The function symbol at the *root position* ε of t is denoted by $\mathtt{root}(t)$. The notation $C[t_1,\ldots,t_n]_{p_1,\ldots,p_n}$ represents the term obtained by replacing each $\Box$ at position p_i of an n-hole *context* C with term t_i for $1 \leq i \leq n$. The *domain* and *range* of a *substitution* σ are denoted by $\mathcal{D}om(\sigma)$ and $\mathcal{R}an(\sigma)$, respectively. The application $\sigma(t)$ of substitution σ to t is abbreviated to $t\sigma$.

An *(oriented) conditional rewrite rule* over $\mathcal{F}$ is a triple (l,r,c), denoted by $l \to r \Leftarrow c$, such that l is a non-variable term in $\mathcal{T}(\mathcal{F},\mathcal{V})$, r is a term in $\mathcal{T}(\mathcal{F},\mathcal{V})$, and c is of form of $s_1 \to t_1 \wedge \cdots \wedge s_n \to t_n$ $(n \geq 0)$ of terms s_i and t_i in $\mathcal{T}(\mathcal{F},\mathcal{V})$. In particular, the conditional rewrite rule $l \to r \Leftarrow c$ is said to be an *(unconditional) rewrite rule* if $n = 0$, and we may abbreviate it to $l \to r$. We sometimes attach a unique label ρ to a rule $l \to r \Leftarrow c$ by denoting $\rho : l \to r \Leftarrow c$, and we use the label to refer to the rule. To simplify notations, we may write labels instead of the corresponding rules. An *(oriented) conditional rewriting system* (*CTRS*, for short) R over a signature $\mathcal{F}$ is a finite set of conditional rewrite rules over $\mathcal{F}$. The *rewrite relation* of R is denoted by $\to_R$. To specify the applied position p and rule ρ, we write $\to_R^p$ or $\to_R^{[p,\rho]}$. A conditional rewrite rule $\rho : l \to r \Leftarrow s_1 \to t_1 \cdots s_k \to t_k$ is called *deterministic* if $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l,t_1,\ldots,t_k)$ and $\mathcal{V}ar(s_i) \subseteq \mathcal{V}ar(l,t_1,\ldots,t_{i-1})$ for $1 \leq i \leq k$. The CTRS R is called a *deterministic CTRS* (a *DCTRS* for short) if all rules in R are deterministic. *Operational termination* of DCTRSs is such that no infinite reductions exist in existing rewrite engines [13]: a CTRS R is *operationally terminating* (*OP-SN*, for short) if for any terms s and t, any proof tree attempting to prove that $s \xrightarrow{*}_R t$ cannot be infinite.

Throughout this paper, we assume that a signature $\mathcal{F}$ consists of a set $\mathcal{D}$ of defined symbols and a set $\mathcal{C}$ of constructors: $\mathcal{F} = \mathcal{D} \uplus \mathcal{C}$. Let R be a CTRS over $\mathcal{F}$. The sets $\mathcal{D}_R$ and $\mathcal{C}_R$ of all *defined symbols* and all *constructors* of R are defined as $\mathcal{D}_R = \{\mathtt{root}(l) \mid l \to r \Leftarrow c \in R\}$ and $\mathcal{C}_R = \mathcal{F} \setminus \mathcal{D}_R$, respectively. We suppose that $\mathcal{D}_R \subseteq \mathcal{D}$ and $\mathcal{C}_R \subseteq \mathcal{C}$. Terms in $\mathcal{T}(\mathcal{C},\mathcal{V})$ are called *constructor terms*. The CTRS

R is called a *constructor system* if every rule $f(t_1, \ldots, t_n) \to r \Leftarrow c$ in R satisfies $\{t_1, \ldots, t_n\} \subseteq \mathcal{T}(\mathcal{C}, \mathcal{V})$.

We use the notion of *context-sensitive reduction* in [12]. Let $\mathcal{F}$ be a signature. A *context-sensitive condition* (*replacement mapping*) μ is a mapping from $\mathcal{F}$ to a set of natural numbers such that $\mu(f) \subseteq \{1, \ldots, n\}$ for n-ary symbols f *in* $\mathcal{F}$. When $\mu(f)$ is not defined explicitly, we assume that $\mu(f) = \{1, \ldots, n\}$. The *context-sensitive reduction* of the *context-sensitive TRS* (R, μ) of a TRS R and a *replacement map* μ is denoted by $\to_{(R,\mu)}$: $\to_{(R,\mu)} = \{(s,t) \mid s \to^p_R t, p \in \mathcal{O}_\mu(s)\}$. The innermost reduction of $\to_{(R,\mu)}$ is denoted by $\underset{\mathrm{i}}{\to}_{(R,\mu)}$: $\underset{\mathrm{i}}{\to}_{(R,\mu)} = \{(s,t) \mid s \to^p_R t, p \in \mathcal{O}_\mu(s), (\forall q > p.\ q \in \mathcal{O}(s)$ implies that $s|_q$ is irreducible$)\}$.

Let $l_i \to r_i$ $(i = 1, 2)$ be two rules whose variables have been renamed such that $\mathcal{V}ar(l_1, r_1) \cap \mathcal{V}ar(l_2, r_2) = \emptyset$. Let p be a position in l_1 such that $l_1|_p$ is not a variable and let θ be a most general unifier of $l_1|_p$ and l_2. This determines a *critical pair* $\langle r_1\theta, (l_1\theta)[r_2\theta]_p \rangle$. If $p = \varepsilon$, then the critical pair is called an *overlay*. If two rules give rise to a critical pair, we say that they *overlap*. We denote the set of critical pairs constructed by rules in a TRS R by $CP(R)$. We also denote the set of critical pairs between rules in R and another TRS R' by $CP(R, R')$. Moreover, $CP_\varepsilon(R)$ denotes the set of *overlays* of R.

Let R and R' be CTRSs such that normal forms are computable (i.e., $\to_R$ and $\to_{R'}$ are well-defined), and T be a set of terms. Roughly speaking, R' is *computationally equivalent* to R with respect to T if there exist mappings ϕ and ψ such that if R terminates on a term $s \in T$ admitting a unique normal form t, then R' also terminates on $\phi(s)$ and for any of its normal forms t', we have $\psi(t') = t$ [24]. In this paper, we assume that ϕ and ψ are the identity mappings.

Let $\underset{1}{\to}$ and $\underset{2}{\to}$ two binary relations on terms, and T' and T'' be sets of terms. We say that $\underset{1}{\to} = \underset{2}{\to}$ *in* $T' \times T''$ ($\underset{1}{\to} \supseteq \underset{2}{\to}$ *in* $T' \times T''$, respectively) if $\underset{1}{\to} \cap (T' \times T'') = \underset{2}{\to} \cap (T' \times T'')$ ($\underset{1}{\to} \cap (T' \times T'') \supseteq \underset{2}{\to} \cap (T' \times T'')$, respectively). Especially, we say that $\underset{1}{\to} = \underset{2}{\to}$ *in* T' (and $\underset{1}{\to} \supseteq \underset{2}{\to}$ *in* T') if $T' = T''$.

3 Completion to Unraveled TRSs

In this section, we show that the completion of the unraveled TRSs produces convergent TRSs that are computationally equivalent to the corresponding CTRSs. To adapt to call-by-value computation, we show *simulation-soundness* of the unraveling for DCTRSs with respect to innermost reduction.

3.1 Unraveling for DCTRSs

We first give the definition of Ohlebusch's unraveling [21]. Given a finite set X of variables, we denote by $\overrightarrow{X}$ the sequence of variables in X without repetitions (in some fixed order).

Definition 3.1 Let R be a DCTRS over a signature $\mathcal{F}$. For every conditional rewrite rule $\rho : l \to r \Leftarrow s_1 \to t_1 \wedge \cdots \wedge s_k \to t_k$, let $|\rho|$ denote the number k of conditions in ρ. For every conditional rule $\rho \in R$, we prepare k 'fresh' function symbols $U^\rho_1, \ldots, U^\rho_{|\rho|}$ not in $\mathcal{F}$, called *U symbols*, in the transformation. We transform ρ into

a set $\mathbb{U}(\rho)$ of $k+1$ unconditional rewrite rules as follows:

$$\mathbb{U}(\rho) = \left\{ \, l \to U_1^\rho(s_1, \overrightarrow{X_1}), \;\; U_1^\rho(t_1, \overrightarrow{X_1}) \to U_2^\rho(s_2, \overrightarrow{X_2}), \;\; \cdots, \;\; U_k^\rho(t_k, \overrightarrow{X_k}) \to r \, \right\}$$

where $X_i = \mathcal{V}ar(l, t_1, \ldots, t_{i-1})$. The system $\mathbb{U}(R) = \bigcup_{\rho \in R} \mathbb{U}(\rho)$ is an unconditional TRS over the extended signature $\mathcal{F}_\mathbb{U} = \mathcal{F} \cup \{ U_i^\rho \mid \rho \in R, 1 \leq i \leq |\rho| \}$.

Note that the definition of $\mathbb{U}$ is essentially equivalent to that in [21,23].

An unraveling U is *simulation-sound* (*simulation-preserving* and *simulation-complete*, respectively) for a DCTRS R over $\mathcal{F}$ if the following holds: for all s and $t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$, $s \xrightarrow{*}_R t$ if ('only if' and 'iff', respectively) $s \xrightarrow{*}_{U(R)} t$. Note that simulation-preservingness is a necessary condition of being unravelings. Roughly speaking, the computational equivalence is equivalent to the combination of simulation-completeness and normal-form uniqueness. The unraveling $\mathbb{U}$ is not simulation-sound for every DCTRS [22]. To avoid this difficulty of non-'simulation-soundness' of $\mathbb{U}$, a restriction to the rewrite relations of the unraveled TRSs is shown in [23], which is done by the *context-sensitive* condition given by the replacement map μ such that $\mu(U_i^\rho) = \{1\}$ for every U_i^ρ in Definition 3.1. We denote the context-sensitive TRS $(\mathbb{U}(R), \mu)$ by $\mathbb{U}_{\rm cs}(R)$. We consider $\mathbb{U}_{\rm cs}$ as an unraveling from CTRSs to context-sensitive TRSs.

Theorem 3.2 ([23]) *For every DCTRS R over $\mathcal{F}$, $\xrightarrow{*}_R = \xrightarrow{*}_{\mathbb{U}_{\rm cs}(R)}$ in $\mathcal{T}(\mathcal{F}, \mathcal{V})$.*

3.2 Call-by-Value Evaluation

To adapt computation of DCTRSs to call-by-value evaluation of functional programs, we define an 'innermost-like' reduction of DCTRSs, called *operationally innermost reduction*. This notion removes the context-sensitivity for simulation-soundness from the corresponding reduction.

For a binary relation $\to$ on terms, the binary relation $\xrightarrow{*}^!$ is defined as $\{ \, (s,t) \mid s \xrightarrow{*} t, t \in \mathit{NF}_\to \, \}$ where $\mathit{NF}_\to$ is the set of normal forms with respect to $\to$. Let R be an OP-SN DCTRS. The *n-level operationally innermost reduction* $\xrightarrow[(n),\mathrm{i}]{}_R$ is defined as follows: $\xrightarrow[(0),\mathrm{i}]{}_R = \emptyset$, and $\xrightarrow[(n+1),\mathrm{i}]{}_R = \xrightarrow[(n),\mathrm{i}]{}_R \cup \{ \, (C[l\sigma], C[r\sigma]) \mid l \to r \Leftarrow s_1 \to t_1 \wedge \cdots s_k \to t_k \in R, \mathcal{R}an(\sigma) \subseteq \mathit{NF}_{\xrightarrow[(n),\mathrm{i}]{}_R}, \forall i.\ s_i\sigma \xrightarrow[(n),\mathrm{i}]{*}{}^!_R t_i\sigma \, \}$. The *operationally innermost reduction* $\xrightarrow[\mathrm{i}]{}_R$ of R is defined as $\bigcup_{i \geq 0} \xrightarrow[(i),\mathrm{i}]{}_R$. Note that if R is a TRS, then the operationally innermost reduction of R is equivalent to the ordinary innermost reduction. Note that the ordinary innermost reduction is not well-defined for every CTRS [6]. However, both the ordinary and operationally innermost reductions of OP-SN CTRSs are well-defined.

For OP-SN DCTRSs, Theorem 3.2 holds for $\xrightarrow[\mathrm{i}]{}_R$ and $\xrightarrow[\mathrm{i}]{}_{\mathbb{U}_{\rm cs}(R)}$.

Theorem 3.3 *For every OP-SN DCTRS R over $\mathcal{F}$, $\xrightarrow[\mathrm{i}]{*}_R = \xrightarrow[\mathrm{i}]{*}_{\mathbb{U}_{\rm cs}(R)}$ in $\mathcal{T}(\mathcal{F}, \mathcal{V})$.*

The proof of Theorem 3.3 follows the proof of Theorem 3.2 (see the appendix of [17]). The context-sensitive constraint is not necessary for the innermost reduction.

Theorem 3.4 *For every DCTRS R over $\mathcal{F}$, $\xrightarrow[\mathrm{i}]{*}_{\mathbb{U}(R)} = \xrightarrow[\mathrm{i}]{*}_{\mathbb{U}_{\rm cs}(R)}$ in $\mathcal{T}(\mathcal{F}, \mathcal{V})$.*

Proof. It is clear that $\xrightarrow[\mathrm{i}]{*}_{\mathbb{U}(R)} \supseteq \xrightarrow[\mathrm{i}]{*}_{\mathbb{U}_{\mathrm{cs}}(R)}$ in $\mathcal{T}(\mathcal{F},\mathcal{V})$. It follows from the notion of innermost and context-sensitive reductions that for a term reachable from terms in $\mathcal{T}(\mathcal{F},\mathcal{V})$, every term in any *irreducible positions* determined by the replacement map is a normal form. Thus, $\xrightarrow[\mathrm{i}]{}_{\mathbb{U}(R)} \subseteq \xrightarrow[\mathrm{i}]{}_{\mathbb{U}_{\mathrm{cs}}(R)}$ in $\mathcal{T}(\mathcal{F},\mathcal{V})$, and hence $\xrightarrow[\mathrm{i}]{*}_{\mathbb{U}(R)} \subseteq \xrightarrow[\mathrm{i}]{*}_{\mathbb{U}_{\mathrm{cs}}(R)}$ in $\mathcal{T}(\mathcal{F},\mathcal{V})$. □

According to Theorem 3.4, when evaluating terms by innermost reductions of $\mathbb{U}_{\mathrm{cs}}(R)$, we can treat $\mathbb{U}(R)$ without the context-sensitive constraint determined by $\mathbb{U}$.

For OP-SN DCTRSs, we have the following simulation-completeness.

Corollary 3.5 *For every OP-SN DCTRS R over $\mathcal{F}$, $\xrightarrow[\mathrm{i}]{*}_R = \xrightarrow[\mathrm{i}]{*}_{\mathbb{U}(R)}$ in $\mathcal{T}(\mathcal{F},\mathcal{V})$.*

3.3 Applying Completion to Unraveled TRSs

In this subsection, we apply the completion procedure to the unraveled TRSs of CTRSs in order to transform them into convergent TRSs that are computationally equivalent to the CTRSs.

First, we introduce the Knuth-Bendix completion procedure [10,25]. Since we add an automated post-process into the inversion compiler, we here use the automated procedure instead of the ordinary completion based on inference rules [2].

Definition 3.6 Let E be a finite set of equations over $\mathcal{F}$, and $\succ$ be a reduction order. Let $E_{(0)} = E$, $R_{(0)} = \emptyset$ and $i = 0$, we apply the following steps:

1. (Orientation) select $s \approx t \in E_{(i)}$ such that $s \succ t$;
2. (Composition) $R' := \{ l \to r' \mid l \to r \in R_{(i)}, r \xrightarrow[\mathrm{i}]{*!}_{R_{(i)} \cup \{s \to t\}} r' \}$;
3. (Deduction) $E' := (E_{(i)} \setminus \{s \approx t\}) \cup CP(\{s \to t\}, R' \cup \{s \to t\})$;
4. (Collapse) $R_{(i+1)} := \{s \to t\} \cup \{ l \to r \mid l \to r \in R', l \not\sqsupseteq s\}$;
5. (Simplification & Deletion)
   $E_{(i+1)} := \{ s'' \approx t'' \mid s' \approx t' \in E', s' \xrightarrow[\mathrm{i}]{*!}_{R_{(i+1)}} s'' \not\equiv t'' \xleftarrow[\mathrm{i}]{*!}_{R_{(i+1)}} t' \}$;
6. if $E_{(i+1)} \neq \emptyset$ then $i := i + 1$ and go to step 1.

Note that $l \sqsupseteq s$ if there are some $C[\ ]$ and θ such that $l \equiv C[s\theta]$. Note that the procedure does not always halt. Suppose that the procedure halts successfully at $i = k$ (hence $E_{(k)} = \emptyset$). Then, R_k is convergent, and R_k satisfies $\xleftrightarrow{*}_E = \xleftrightarrow{*}_{R_k}$ [2]. Note that when there is no rule to select at the Orientation step, the procedure halts in failure.

The usual purpose of the completion is to generate TRSs that are equivalent to given equation sets. In contrast to the usual purpose, we would like the completion to transform unraveled TRSs $\mathbb{U}(R)$ into convergent TRSs as executable programs that are computationally equivalent to the original CTRSs R. For this reason, we start the completion procedure from $(CP(\mathbb{U}(R)), \{ l \to r \in \mathbb{U}(R) \mid \nexists l' \to r' \in \mathbb{U}(R), l \sqsupset l' \})$ where $\mathbb{U}(R) \subseteq \succ$. Moreover, consistency of the normal forms of $\mathbb{U}(R)$ (that is, they are also normal forms of the modified system) is necessary for preserving computational equivalence of R. For this requirement, we add the side condition '$\mathtt{root}(s)$ is a U symbol' to the Orientation step:

1. (Orientation†) *select $s \approx t \in E_{(i)}$ such that $s \succ t$ and $\mathtt{root}(s)$ is a U symbol;*

Due to the side condition of the Orientation step, and the basic character of the completion procedure [2], the completion procedure produces convergent TRSs that are computationally equivalent to the input TRSs.

Theorem 3.7 *Let R be an OP-SN DCTRS over $\mathcal{F}$, and $\succ$ be a reduction order such that $\mathbb{U}(R) \subseteq \succ$. Let $E_0 = CP(\mathbb{U}(R))$, $R_0 = \{l \to r \in \mathbb{U}(R) \mid \nexists l' \to r \in \mathbb{U}(R), l \sqsupset l'\}$, and R' be a TRS obtained by the completion procedure from (E_0, R_0) with $\succ$. Then, (1) R' is convergent and (2) $\xrightarrow{*}^{!}_{\mathbb{U}(R)} = \xrightarrow{*}^{!}_{R'}$ in $\mathcal{T}(\mathcal{F},\mathcal{V})$.*

Proof. It follows from the side condition '$\mathtt{root}(s)$ is a U symbol' of the Orientation that $NF_{\mathbb{U}(R)}(\mathcal{F},\mathcal{V}) = NF_{R'}(\mathcal{F},\mathcal{V})$. It also follows from the correctness of the completion (Theorem 7.3.5 in [2]) that R' is convergent and $\xleftrightarrow{*}_{CP(\mathbb{U}(R))\cup\mathbb{U}(R)} \subseteq \xrightarrow{*}_{R'} \cdot \xleftarrow{*}_{R'}$. Let $\underset{1}{\to} = \{(s,t) \mid s,t \in \mathcal{T}(\mathcal{F},\mathcal{V}),\ s \xrightarrow{+}^{!}_{R} t\}$ and $\underset{2}{\to} = \{(s,t) \mid s,t \in \mathcal{T}(\mathcal{F},\mathcal{V}),\ s \xrightarrow{+}^{!}_{R'} t\}$. Then, we have $\underset{1}{\overset{*}{\leftrightarrow}} \subseteq \underset{2}{\overset{*}{\leftrightarrow}}$, confluence of $\underset{2}{\to}$, termination of $\underset{1}{\to}$, and $NF_1 = NF_2$ where NF_i is the set of normal forms with respect to $\to$. Therefore, it follows from Theorem 3.3 in [26] that $\underset{1}{\overset{*}{\to}} = \underset{2}{\overset{*}{\to}}$ in $\mathcal{T}(\mathcal{F},\mathcal{V})$, and hence $\xrightarrow{*}^{!}_{R} = \xrightarrow{*}^{!}_{R'}$ in $\mathcal{T}(\mathcal{F},\mathcal{V})$. □

Note that (2) implies $NF_{\mathbb{U}(R)}(\mathcal{F},\mathcal{V}) = NF_{R'}(\mathcal{F},\mathcal{V})$.

As we have already described, we would like to modify systems on a call-by-value interpretation. The innermost reduction is necessary for modeling some primitive functions used in some examples of program inversion (R_{du} in Subsection 4.2). Since such TRSs are not confluent but innermost-confluent, the completion procedure possibly fails in modifying those TRSs. To solve this problem and to obtain innermost-convergent TRSs instead of convergent TRSs, we show an initial setting of the completion to unraveled TRSs.

Theorem 3.8 *Let R be an OP-SN DCTRS over $\mathcal{F}$, and $\succ$ be a reduction order such that $\mathbb{U}(R) \subseteq \succ$. Let $E_0 = CP_\varepsilon(\mathbb{U}(R))$, $R_0 = \{l \to r \in \mathbb{U}(R) \mid \nexists l' \to r \in \mathbb{U}(R), l \sqsupset l'\}$, and R' be a TRS obtained by the completion procedure from (E_0, R_0) with $\succ$. Then, (1) R' is innermost-convergent and (2) $\underset{\mathrm{i}}{\overset{*}{\to}}^{!}_{\mathbb{U}(R)} = \underset{\mathrm{i}}{\overset{*}{\to}}^{!}_{R'}$ in $\mathcal{T}(\mathcal{F},\mathcal{V})$.*

Proof. We here show the sketch of the proof. The full proof will be found in the appendix of [17]. It follows from the side condition '$\mathtt{root}(s)$ is a U symbol' of the Orientation that $NF_{\mathbb{U}(R)}(\mathcal{F},\mathcal{V}) = NF_{R'}(\mathcal{F},\mathcal{V})$. Following the correctness proof of the ordinary completion (Section 7.3 and 7.4 in [2]), it can be shown that $\underset{\mathrm{i}}{\overset{*}{\leftrightarrow}}_{\mathbb{U}(R)} \subseteq \underset{\mathrm{i}}{\overset{*}{\to}}_{R'} \cdot \underset{\mathrm{i}}{\overset{*}{\leftarrow}}_{R'}$ and R' is innermost-convergent. The remainder of the proof is similar to the corresponding part of the proof of Theorem 3.7. □

Note that (2) implies $NF_{\mathbb{U}(R)}(\mathcal{F},\mathcal{V}) = NF_{R'}(\mathcal{F},\mathcal{V})$.

The following condition is necessary for the completion procedure to halt 'successfully'[4]: for every term $s \in \mathcal{T}(\mathcal{F},\mathcal{V})$, its normal form over $\mathcal{F}$ with respect to the innermost reduction is unique, that is, for all normal forms t_1 and $t_2 \in NF_{\mathbb{U}(R)}(\mathcal{F},\mathcal{V})$, if $t_1 \underset{\mathrm{i}}{\overset{*}{\leftarrow}}_{\mathbb{U}(R)} s \underset{\mathrm{i}}{\overset{*}{\to}}_{\mathbb{U}(R)} t_2$, then $t_1 \equiv t_2$. Note that if R is confluent, then $\mathbb{U}(R)$ satisfies this property. If $t_1 \underset{\mathrm{i}}{\overset{*}{\leftarrow}}_{\mathbb{U}(R)} s \underset{\mathrm{i}}{\overset{*}{\to}}_{\mathbb{U}(R)} t_2$ and $t_1 \not\equiv t_2$, then the added side condition '$\mathtt{root}(s)$ is a U symbol' prevents t_1 and t_2 from being joinable.

[4] Notice that this condition is not sufficient for the procedure to halt; In other words, the procedure halts (or keeps running) 'unsuccessfully' if the input system does not satisfy this condition.

Example 3.9 Consider the non-convergent TRS $\mathbb{U}(\mathcal{I}nv(R_1))$ in Section 1 again. Given the *lexicographic path order* (LPO) $\succ_{\mathsf{lpo}}$ determined by the precedence $>$ with $\mathsf{InvSnoc} > U_1 > \mathsf{cons} > \mathsf{nil} > \langle\ \rangle$, we obtain the following convergent and non-overlapping TRS by the completion procedure (in 4 cycles):

$$R_2 = \left\{ \begin{array}{l} \mathsf{InvSnoc}((x :: ys)) \to U_1(\mathsf{InvSnoc}(ys), x, ys), \\ U_1(\langle xs, y\rangle, x, ys) \to \langle (x :: xs), y\rangle, \qquad U_1(\mathsf{InvSnoc}(\mathsf{nil}), x, \mathsf{nil}) \to (\mathsf{nil}, x) \end{array} \right\}$$

The completion removes the rule $\mathsf{InvSnoc}(\mathsf{cons}(y, \mathsf{nil})) \to \langle \mathsf{nil}, y\rangle$ from $\mathbb{U}(\mathcal{I}nv(R_1))$ and the resultant TRS R_2 are non-overlapping. This removal is effective in translating the TRS into a functional program.

Unfortunately, the completion procedure does not always halt even if the inputs are restricted to unraveled TRSs. For example, the completion does not halt for the unraveled TRS obtained from Example 7.1.5 in [22] although there exists an appropriate convergent TRS which is equivalent to the unraveled TRS.

3.4 Translation Back into Functional Programs

In this subsection, we informally discuss translations from convergent and non-overlapping TRS R_2 into functional programs in Standard ML. The translations have not been automated yet but we believe that the automation is neither so difficult nor so surprising. It is difficult to translate $\mathcal{I}nv(R_1)$ or $\mathbb{U}(\mathcal{I}nv(R_1))$ into functional programs because deciding a priority of rewrite rules is difficult in general. On the other hand, we do not have to consider such a priority for R_2 that is computationally equivalent to $\mathcal{I}nv(R_1)$ because R_2 is not only confluent but also non-overlapping.

The U symbols U_i^{ρ} introduced by the unraveling are often considered to express `let`, `if` or `case` clauses in functional programming languages. In the rewrite rules of R_2, the U symbol U_1 plays the role of a `case` clause as follows:

```
case InvSnoc( ys ) of (xs,y) => ( x::xs, y )
            | InvSnoc( [] ) => ( [], y )
```

where `InvSnoc( [] )` is not well-formed in the syntax of Standard ML. It is natural to write this fragment by introducing the extra `case` clause for `ys` as follows:

```
case ys of [] => ( [], y )
        | _  => (case InvSnoc( ys ) of (xs,y) => ( x::xs, y ) )
```

Thus, we translate the TRS R_2 into the following program:

```
fun InvSnoc( (x::ys) ) =
    case ys of [] => ( [], x )
             | _ => (case InvSnoc(ys) of (xs,y) => (x::xs,y) );
```

Other approaches to translations are possible. For example, we can consider U_1 as the composition of `if` and `let` clauses or as a 'local function' defined in $\mathsf{InvSnoc}$.

3.5 Completion with Termination Provers

In this subsection, we show an example where the completion consults with termination provers. This idea is firstly introduced in [28].

Consider the following unraveled TRS:

$$R_3 = \mathbb{U}(\mathcal{I}nv(R_1)) \cup \left\{ \begin{array}{l} \mathsf{InvSnocRev}(\mathsf{nil}) \to \langle \mathsf{nil} \rangle, \\ \mathsf{InvSnocRev}(y) \to \mathsf{U}_2(\mathsf{InvSnoc}(y), y), \\ \mathsf{U}_2(\langle z, x \rangle, y) \to \mathsf{U}_3(\mathsf{InvSnocRev}(z), x, y, z), \\ \mathsf{U}_3(\langle x_1 \rangle, x, y, z) \to \langle \mathsf{cons}(x, x_1) \rangle \end{array} \right\}$$

In contrast to the case of $\mathsf{InvSnoc}$, there is no LPO $\succ_{\mathsf{lpo}}$ with $R_3 \subseteq \succ_{\mathsf{lpo}}$. Detecting such a path-based reduction order (e.g., LPO and recursive path order) in advance may be impossible or there might be no such path-based order. Thus, analyzing the dependencies of defined symbols is necessary to prove the termination of R_3.

To achieve this kind of analysis, we introduce termination provers to the completion procedure. We modify Orientation, following the approach shown in [28]:

1. (Orientation‡) *select* $s \approx t \in E_{(i)}$ *such that* $\bigcup_{j=0}^{i} R_{(j)} \cup \{s \to t\}$ *is terminating, and* $\mathtt{root}(s)$ *is a U symbol;*

In the case of innermost reduction, it is enough to check innermost-termination of $\bigcup_{j=0}^{i} R_{(j)} \cup \{s \to t\}$. This setting enables us to employ existing termination provers at each Orientation step.

In this mechanism, the TRS R_3 is transformed by the procedure (in 2 cycles) into the following convergent and non-overlapping TRS:

$$R_2 \cup \left\{ \begin{array}{l} \mathsf{InvSnocRev}(v) \to \mathsf{U}_2(\mathsf{InvSnoc}(v), v), \\ \mathsf{U}_2(\langle w, x \rangle, v) \to \mathsf{U}_3(\mathsf{U}_2(\mathsf{InvSnoc}(w), w), v, w, x), \\ \mathsf{U}_3(\langle xs \rangle, v, w, x) \to \langle \mathsf{cons}(x, xs) \rangle, \quad \mathsf{U}_3(\mathsf{InvSnoc}(\mathsf{nil}), x, \mathsf{nil}) \to \langle \mathsf{nil}, x \rangle \end{array} \right\}$$

4 Completion as Post-Process in Program Inversion

In this section, we apply the unraveling $\mathbb{U}$ and the completion procedure to CTRSs generated by the partial inversion compiler [20], that is, we apply the completion as a *post-process* of $\mathbb{U}(\mathcal{I}nv(\cdot))$ to the unraveled TRSs. First, we briefly introduce the feature of inverse systems for injective functions. Next, we show the results of experiments by an implementation of the framework. We employ the partial inversion $\mathcal{I}nv$ in [20] that generates a partial inverse CTRS from a pair of a given constructor TRS and a specification that we do not describe in detail here. For a defined symbol F, the defined symbol $\mathsf{Inv}F$ introduced by $\mathcal{I}nv$ represents a full inverse of F. We assume that a constructor TRS defines a main injective function, and that the specification requires a full inverse of the main function.

4.1 Inverse CTRSs of Injective Functions

We first define *injectivity* of TRSs [18], and then give sufficient condition for input constructor TRSs whose inverse CTRSs generated by $\mathcal{I}nv$ are convergent.

Definition 4.1 Let R be a convergent constructor TRS. A defined symbol F of R is called *injective (with respect to normal forms)* if the binary relation

$\{(\langle s_1,\ldots,s_n\rangle,t) \mid s_1,\ldots,s_n,t \in NF_R(\mathcal{F},\mathcal{V}), F(s_1,\ldots,s_n) \xrightarrow{*}_R t\}$ is an injective mapping. The TRS R is called *injective (with respect to normal forms)* if all of its defined symbols are injective.

For example, the TRS R_1 in Section 1 is injective. Note that every injective TRS is non-erasing [18].

The following defined symbol Reverse computes the reverses of given lists:

$$R_4 = \left\{ \begin{array}{cc} \mathsf{Reverse}(xs) \to \mathsf{Rev}(xs,\mathsf{nil}), & \mathsf{Rev}(\mathsf{nil},ys) \to ys, \\ \mathsf{Rev}(\mathsf{cons}(x,xs),ys) \to \mathsf{Rev}(xs,\mathsf{cons}(x,ys)) & \end{array} \right\}$$

The inverse TRS of the above TRS is generated as follows:

$$\mathbb{U}(\mathcal{I}nv(R_4)) = \{\,\cdots,\ \mathsf{InvRev}(z) \to U_4(\mathsf{InvRev}(z),z),\ \cdots\}.$$

Reverse is injective but Rev is not. Thus, R_4 is not injective. In this case, the TRS $\mathbb{U}(\mathcal{I}nv(R_4))$ is not terminating. For this reason, we restrict ourselves to injective functions whose inverse TRSs are terminating. In [18], a sufficient condition has been shown for the full inversion compiler in [19] to generate convergent inverse CTRSs from injective TRSs. The condition is also effective for the partial inversion compiler $\mathcal{I}nv$ [20].

Theorem 4.2 *Let R be a non-erasing innermost-convergent constructor TRS. If $F \in \mathcal{D}_R$ is injective, then for all s, t_1 and $t_2 \in NF_{\mathcal{I}nv(R)}(\mathcal{F},\mathcal{V})$, $t_1 \xleftarrow[\mathrm{i}]{*}_{\mathbb{U}(\mathcal{I}nv(R))} \mathsf{Inv}F(s) \xrightarrow[\mathrm{i}]{*}_{\mathbb{U}(\mathcal{I}nv(R))} t_2$ implies $t_1 \equiv t_2$. Suppose that for every rule $F(u_1,\ldots,u_n) \to r$ in R, if r is not a variable, then the root symbol of r does not depend [5] on F. Then, the CTRS $\mathcal{I}nv(R)$ is OP-SN, and the TRS $\mathbb{U}(\mathcal{I}nv(R))$ is terminating.*

The proof of Theorem 4.2 follows the proof of Theorem 4 in [18] (see the appendix of [17]). Note that $\mathbb{U}(\mathcal{I}nv(R))$ is not always confluent even if $\mathcal{I}nv(R)$ is confluent. The first claim in Theorem 4.2 shows that given an injective TRS R, $\mathbb{U}(\mathcal{I}nv(R))$ satisfies the necessary condition (described above Example 3.9) for successful run of the completion. When a constructor TRS R does not satisfy the condition in Theorem 4.2 for preserving termination, we directly check the (innermost-)termination of $\mathbb{U}(\mathcal{I}nv(R))$. In other words, when R satisfies the second assumption in Theorem 4.2, we are free of the termination check of $\mathbb{U}(\mathcal{I}nv(R))$ that is less efficient than the check of satisfying the second assumption.

4.2 Experiments

In this section, we report the results of applying an implementation of our approach based on Theorem 3.8 and the Orientation§ step to several samples.

The implementation of the completion procedure is based on the ML programs shown in [2]. In our implementation, we use the following weight w for equations: $w(s \simeq t) = (size(s) + depth(s)) \times 2 + (size(t) + depth(t))$ where $s \succ t$, and $size(u)$ and $depth(u)$ are the term-size and term-depth of u, respectively. The weight w is one of weights that come from experience. At every Orientation step, the implementation selects an equation whose weight is the minimum in equations $s \approx t$

[5] An n-ary symbol G of R *depends on a symbol* F if (G,F) is in the transitive closure of the relation $\{\,(G',F') \mid G'(\cdots) \to C[F'(\cdots)] \in R\,\}$.

such that $(\bigcup_{j=0}^{i} R_{(i)}) \cup \{s \to t\}$ is innermost terminating. The implementation is written in Standard ML of New Jersey, and it was executed under OS Vine Linux 4.2, on an Intel Pentium 4 CPU at 3 GHz and 1 GByte of primary memory. The implementation consults with AProVE 1.2 [4] by system call in SML/NJ as a termination prover at the Orientation§ step that checks the innermost termination. The implementation checks the innermost termination of input TRSs in advance. The timeout for checking termination is 300 seconds in every call of the prover.

In [8], the results of the experiments for the inversion compiler LRinv [8,9] running on 15 samples [6] are shown where LRinv succeeds in inverting all of the examples. Those examples are written in the scheme script Gauche: 5 scripts on 'list manipulation' (`snoc.fct`, `snocrev.fct`, `reverse.fct`, and so on), 3 on 'number manipulation', 4 on 'encoding and decoding' (`treelist.fct`, and so on), and 2 on 'printing and parsing'. The inverse TRSs of the scripts `snoc.fct`, `snocrev.fct` and `reverse.fct` correspond to the TRSs $\mathbb{U}(\mathcal{I}nv(R_1))$, R_3 and $\mathbb{U}(\mathcal{I}nv(R_4))$, respectively. None of the constructor TRSs corresponding to the scripts `reverse.fct`, `unbin.fct`, `treepath.fct`, `pack.fct` and `pack-bin.fct` are injective. The CTRSs obtained by $\mathcal{I}nv$ from them are not OP-SN. We excluded those non-'OP-SN' examples from experiments.

In the examples, there is a special primitive operator `du` defined as follows: $\mathtt{du}(\langle x\rangle) = \langle x, x\rangle$, $\mathtt{du}(\langle x, x\rangle) = \langle x\rangle$, and $\mathtt{du}(\langle x, y\rangle) = \langle x, y\rangle$ if $x \neq y$. We encode this operator as the following terminating TRS:

$$R_{\mathsf{du}} = \left\{ \begin{array}{ccc} \mathsf{Du}(\langle x\rangle) \to \langle x, x\rangle, & \mathsf{Du}(\langle x, y\rangle) \to \mathsf{EqChk}(\mathsf{EQ}(x, y)), & \\ \mathsf{EqChk}(\langle x\rangle) \to \langle x\rangle, & \mathsf{EqChk}(\mathsf{EQ}(x, y)) \to \langle x, y\rangle, & \mathsf{EQ}(x, x) \to \langle x\rangle \end{array} \right\}$$

Since R_{du} has no *overlay*, R_{du} is *locally innermost-confluent*, and hence, R_{du} is *innermost-confluent* [11]. Under the innermost reduction, the TRS can simulate computation of `du`. The operator `du` is an inverse of itself [8,9]. Thus, the TRS R_{du} is also an inverse system of itself. For this reason, exceptionally, the inversion compiler does not produce any rules of InvDup but introduces Du instead of InvDup.

Table 1 summarizes the results of the experiments for our approach running on 10 of the 15 examples previously mentioned, which are easily translated to TRSs [7]. The second column labeled with 'CR by [1]' shows whether the input TRS of the example is in the class shown in [1], in which the corresponding inverse TRS is orthogonal and thus confluent. In that case, the implemented procedure only checks innermost-termination of the inverse TRS. The third column labeled with 'SN by Th. 4.2' shows whether the input TRS satisfies the conditions in Theorem 4.2, that is, the corresponding inverse TRSs are terminating. The fourth column shows the results of completion ('success $\surd$', 'fail' or 'timeout') with the numbers of running 'cycles' in the sense of Definition 3.6. The numbers of cycles are equivalent to times of applying Orientation. As described above, the implementation checks the innermost termination of input TRSs before the completion procedure starts. Thus, we have the results 'success (0 cycle) and 1 call of provers'. The sixth column

[6] Unfortunately, the site shown in [8] is not accessible now. The examples are also described briefly as functional programs in [9], and some of the detailed programs can be found in [9].

[7] The detail of the experiments will be available from the following URL:
`http://www.trs.cm.is.nagoya-u.ac.jp/repius/experiments/`

Table 1
the results of the experiments

example	CR by [1]	SN by Th.4.2	innermost CR&SN by completion			
			result (cycles)	call AProVE	time	¬overlap
du (primitive)			√ (0 cycle)	1 time	0.64 s	
snoc.fct		√	√ (1 cycle)	2 times	2.12 s	√
snocrev.fct		√	√ (2 cycles)	3 times	4.61 s	√
double.fct			√ (1 cycle)	2 times	2.32 s	
mirror.fct			√ (2 cycles)	3 times	4.10 s	
zip.fct	√	√	√ (0 cycle)	1 time	1.04 s	√
inc.fct		√	√ (1 cycle)	2 times	2.53 s	√
octbin.fct	√	√	√ (0 cycle)	1 time	1.28 s	√
treelist.fct		√	timeout (0 cycle)	timeout at 1st time	timeout	—
print-sexp.fct			√ (6 cycles)	7 times	35.53 s	√
print-xml.fct			√ (2 cycles)	3 times	14.02 s	
treelist.fct†		√	√ (4 cycles)	5 times	40.92 s	

†The improved transformation [16] of $\mathbb{U}$ is applied instead of $\mathbb{U}$.

shows the average time of 5 trials. The rightmost column shows whether or not the resultant TRSs are non-overlapping (*surd* means the resultant is non-overlapping). None of the resultant TRSs has overlay while part of them are overlapping.

In the experiments, the procedure failed in modifying `treelist.fct` because of a timeout in pre-checking the innermost termination [8]. For the intermediate CTRS R_5 generated from `treelist.fct`, the improvement of $\mathbb{U}$ shown in [16] is effective. Note that the improvement proposed for the variant $\mathbb{U}'$ of $\mathbb{U}$ is also applicable to $\mathbb{U}$ where $\mathbb{U}'$ shown in [3,20,19] is obtained by setting $X_i = \mathcal{V}ar(l, t_1, \ldots, t_{i-1}) \cap \mathcal{V}ar(r, t_i, s_{i+1}, t_{i+1}, \ldots, s_k, t_k)$ in Definition 3.1. $\mathbb{U}$ unravels one of the conditional rules into 5 rules but the improved transformation $\mathbb{U}'$ of $\mathbb{U}$ unravels the rule into 4. Surprisingly, the completion succeeds in modifying the TRS $\mathbb{U}'(R_5)$ (see the result on the bottom line of Table 1). Remark that in other examples, there is no difference between applications of $\mathbb{U}$ and $\mathbb{U}'$ to the inverse CTRSs.

We tried to prove by TTT [7] innermost-termination of $\mathbb{U}(R_5)$ and $\mathbb{U}'(R_5)$. The results are the same with AProVE 1.2: 'timeout' and 'success', respectively. Moreover, AProVE 1.2 succeeded in proving termination of both $\mathbb{U}(R_5)$ and $\mathbb{U}'(R_5)$, and TTT did not in either of those TRSs.

5 Comparison with Related Work

Completion procedures are used for solving word problems, for transforming equations to equivalent convergent systems, or for proving inductive theorems. As far as we know, there is no application of completion to program modification, and there is no program transformation based on unravelings in order to produce computationally equivalent systems. The method in this paper does not always succeed for every confluent and OP-SN DCTRSs while the latest transformation [24] based on Viry's approach [27] always succeeds. Consider the example in Section 1 again. To eliminate the unexpected normal form $U_1(U_1(U_1(\mathsf{InvSnoc}(\mathsf{nil}), \mathsf{c}, \mathsf{nil}), \mathsf{b}, [\mathsf{c}]), \mathsf{a}, [\mathsf{b}, \mathsf{c}])$ from the set of normal forms, the transformation in [24] is effective in the sense

[8] The current 'web interface' version of AProVE succeeds in proving this innermost termination.

of producing convergent systems. By the transformation, we obtain the following convergent TRS instead of $\mathbb{U}(\mathcal{I}nv(R_1))$:

$$\left\{ \begin{array}{r} \mathsf{InvSnoc}(\mathsf{cons}(y,\mathsf{nil}),z) \to \{\langle \mathsf{nil}, y\rangle\}, \\ \mathsf{InvSnoc}(\mathsf{cons}(x,ys),\bot) \to \mathsf{InvSnoc}(\mathsf{cons}(x,ys),\{\mathsf{InvSnoc}(ys,\bot)\}), \\ \mathsf{InvSnoc}(\mathsf{cons}(x,ys),\{\langle xs,y\rangle\}) \to \{\langle \mathsf{cons}(x,xs),y\rangle\}, \qquad\qquad\qquad \\ \mathsf{InvSnoc}(\{xs\},z) \to \{\mathsf{InvSnoc}(xs,\bot)\}, \quad \{\{x\}\} \to \{x\} \end{array} \right\}$$

$$\cup \{\, c(x_1,\ldots,\{x_i\},\ldots,x_n) \to \{c(x_1,\ldots,x_n)\} \mid c \text{ is a constructor, } n \geq 1 \,\}$$

where $\{\ \}$ and $\bot$ are special function symbols not in the original signature. In this system, the term $\mathsf{InvSnoc}([\mathsf{a},\mathsf{b},\mathsf{c}],\bot)$ has a unique normal form $\{\langle[\mathsf{a},\mathsf{b}],\mathsf{c}\rangle\}$. However, it is difficult to translate the convergent but *overlapping* TRS into a functional program because the system contains special symbols $\{\ \}$ and $\bot$ and *overlapping* rules.

On the other hand, the modified completion in this paper unexpectedly succeeded for all examples on program inversion we tried, except for functions that call non-injective functions such as ones including accumulators. The resultant systems of our method are not always non-overlapping (equivalently non-overlay for innermost reduction), while the results of [24] are always overlapping. One of our future works is to find a subclass in which the completion halts successfully, and then to compare this framework with the transformation [24] based on Viry's approach [27].

The inversion compiler LRinv, the closest one to the method in this paper, has been proposed for injective functions written in a functional language [8,5,9]. This compiler translates source programs into programs in a grammar language, and then inverts the grammar programs into inverse grammar programs. To eliminate nondeterminism in the inverse programs, their compiler applies *LR parsing* to the inverse programs. The classes for which LR parsing and the completion procedure work successfully are not well known, which makes it difficult to compare LRinv and our method. However, LRinv succeeds in generating inverse functions from the scripts `reverse.fct`, `unbin.fct`, `treepath.fct`, `pack.fct` and `pack-bin.fct` while the method in this paper is not applicable to those scripts. From this fact, LRinv seems to be stronger than the method in this paper but there must be a plenty of room on improving the principle of inversion used in the partial inversion complier in [20]. As future work, we plan to extend the partial inversion compiler for functions with accumulators such as Rev.

Acknowledgement

We are grateful to Germán Vidal and the anonymous reviewers for many comments for improving this paper. We also thank Tsubasa Sakata for his discussing correctness of the completion procedure on innermost reduction. This work is partly supported by MEXT. KAKENHI #18500011, #20300010 and #20500008 and Kayamori Foundation of Informational Science Advancement.

References

[1] Almendros-Jiménez, J. M. and G. Vidal, *Automatic partial inversion of inductively sequential functions*, in: *Proc. of the 18th International Symposium on Implementation and Application of Functional*

Languages, Lecture Notes in Computer Science **4449** (2006), pp. 253–270.

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

[3] Durán, F., S. Lucas, J. Meseguer, C. Marché and X. Urbain, *Proving termination of membership equational programs*, in: *Proc. of the 2004 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation* (2004), pp. 147–158.

[4] Giesl, J., P. Schneider-Kamp and R. Thiemann, *Automatic termination proofs in the dependency pair framework*, in: *Proc. of the 3rd International Joint Conference on Automated Reasoning*, Lecture Notes in Computer Science **4130** (2006), pp. 281–286.

[5] Glück, R. and M. Kawabe, *A method for automatic program inversion based on LR(0) parsing*, Fundam. Inform. **66** (2005), no. 4, pp. 367–395.

[6] Gramlich, B., *On the (non-)existence of least fixed points in conditional equational logic and conditional rewriting*, in: *Proc. 2nd Int. Workshop on Fixed Points in Computer Science – Extended Abstracts* (2000), pp. 38–40.

[7] Hirokawa, N. and A. Middeldorp, *Tyrolean termination tool: Techniques and features*, Information and Computation **205** (2007), no. 4, pp. 474–511.

[8] Kawabe, M. and Y. Futamura, *Case studies with an automatic program inversion system*, in: *Proc. of the 21st Conference of Japan Society for Software Science and Technology*, 6C-3, 2004, pp. 1–5.

[9] Kawabe, M. and R. Glück, *The program inverter LRinv and its structure*, in: *Proc. of the 7th International Symposium on Practical Aspects of Declarative Languages*, Lecture Notes in Computer Science **3350** (2005), pp. 219–234.

[10] Knuth, D. E. and P. B. Bendix, *Simple word problems in universal algebra*, in: *Computational Problems in Abstract Algebra* (1970), pp. 263–297.

[11] Krishna Rao, M. R. K., *Relating confluence, innermost-confluence and outermost-confluence properties of term rewriting systems*, Acta Informatica **33** (1996), no. 6, pp. 595–606.

[12] Lucas, S., *Context-sensitive computations in functional and functional logic programs*, Journal of Functional and Logic Programming **1998** (1998), no. 1.

[13] Lucas, S., C. Marché and J. Meseguer, *Operational termination of conditional term rewriting systems*, Information Processing Letters **95** (2005), no. 4, pp. 446–453.

[14] Marchiori, M., *Unravelings and ultra-properties*, in: *Proc. of the 5th International Conference on Algebraic and Logic Programming*, Lecture Notes in Computer Science **1139** (1996), pp. 107–121.

[15] Marchiori, M., *On deterministic conditional rewriting*, Computation Structures Group, Memo 405, MIT Laboratory for Computer Science (1997).

[16] Nishida, N., T. Mizutani and M. Sakai, *Transformation for refining unraveled conditional term rewriting systems*, in: *Proc. of the Sixth International Workshop on Reduction Strategies in Rewriting and Programming*, Electronic Notes in Theoretical Computer Science **174** (2007), Issue 10, pp. 75–95.

[17] Nishida, N. and M. Sakai, *Completion as Post-Process in Program Inversion of Injective Functions*, `http://www.trs.cm.is.nagoya-u.ac.jp/~nishida/papers/` (2008), the full version of this paper.

[18] Nishida, N., M. Sakai and T. Kato, *Convergent term rewriting systems for inverse computation of injective functions*, in: *Proc. of the 9th International Workshop on Termination* (2007), pp. 77–81.

[19] Nishida, N., M. Sakai and T. Sakabe, *Generation of inverse computation programs of constructor term rewriting systems*, The IEICE Trans. Inf.& Syst. **J88-D-I** (2005), no. 8, pp. 1171–1183 (in Japanese).

[20] Nishida, N., M. Sakai and T. Sakabe, *Partial inversion of constructor term rewriting systems*, in: *Proc. of the 16th International Conference on Rewriting Techniques and Applications*, Lecture Notes in Computer Science **3467** (2005), pp. 264–278.

[21] Ohlebusch, E., *Termination of logic programs: Transformational methods revisited*, Applicable Algebra in Engineering, Communication and Computing **12** (2001), no. 1-2, pp. 73–116.

[22] Ohlebusch, E., "Advanced Topics in Term Rewriting," Springer-Verlag, 2002.

[23] Schernhammer, F. and B. Gramlich, *On proving and characterizing operational termination of deterministic conditional rewrite systems*, in: *Proc. of the 9th International Workshop on Termination* (2007), pp. 82–85.

[24] Serbanuta, T.-F. and G. Rosu, *Computationally equivalent elimination of conditions*, in: *Proc. of the 17th International Conference on Rewriting Techniques and Applications*, Lecture Notes in Computer Science **4098** (2006), pp. 19–34.

[25] Terese, "Term Rewriting Systems," Cambridge University Press, 2003.

[26] Toyama, Y., *How to prove equivalence of term rewriting systems without induction*, Theoretical Computer Science **90** (1991), no. 2, pp. 369–390.

[27] Viry, P., *Elimination of conditions*, Journal of Symbolic Computation **28** (1999), no. 3, pp. 381–401.

[28] Wehrman, I., A. Stump and E. M. Westbrook, *Slothrop: Knuth-Bendix completion with a modern termination checker*, in: *Proc. of the 17th International Conference on Term Rewriting and Applications*, Lecture Notes in Computer Science **4098** (2006), pp. 287–296.

A transformational approach to prove outermost termination automatically

Matthias Raffelsieper[1]

Department of Computer Science, TU Eindhoven, P.O. Box 513, 5600 MB Eindhoven, The Netherlands

Hans Zantema[2]

Department of Computer Science, TU Eindhoven, P.O. Box 513, 5600 MB Eindhoven, The Netherlands

Institute for Computing and Information Sciences, Radboud University Nijmegen, P.O. Box 9010, 6500 GL Nijmegen, The Netherlands

Abstract

We present a transformation from a generalized form of left-linear TRSs, called *quasi left-linear* TRSs, to TRSs such that outermost termination of the original TRS can be concluded from termination of the transformed TRS. In this way we can apply state-of-the-art termination tools for automatically proving outermost termination of any given quasi left-linear TRS. Experiments show that this works well for non-trivial examples, some of which could not be automatically proven outermost terminating before. Therefore, our approach substantially increases the class of systems that can be shown outermost terminating automatically.

1 Introduction

A lot of work has been done on automatically proving termination and innermost termination. However, also termination with respect to the outermost strategy makes sense. For instance, this is the standard strategy in the functional programming language Haskell [10], and it can be specified in CafeOBJ [4] and Maude [2]. We will focus on the most general variant of the outermost strategy: reducing a redex is always allowed if it is not a proper subterm of another redex. Termination with respect to this strategy we shortly call *outermost termination*. This is different from the approaches for proving termination of Haskell presented in [7,15], where termination is only proven for a specific set of terms (ground instantiations of a so-called *start term*) and not for every possible term. Furthermore, the language

[1] Email: M.Raffelsieper@tue.nl

[2] Email: H.Zantema@tue.nl

Haskell does not allow overlapping rules, i.e., there is at most one rule applicable for every term.

Let's consider the following example. The infinite list $0 : 1 : 2 : 3 : \cdots$ can be generated by applying the non-terminating rule

$$\mathsf{from}(x) \;\to\; x : \mathsf{from}(\mathsf{s}(x))$$

to the start term $\mathsf{from}(0)$. If we want to check for overflow, e.g., numbers should not be n or higher, we could add the rule

$$\mathsf{s}^n(x) : xs \;\to \mathsf{overflow}.$$

Now for sure the TRS remains non-terminating since it still contains the first non-terminating rule. But we expect the combined system to be outermost terminating. This is the kind of examples for which we want to prove outermost termination automatically.

Until now Cariboo [9] was the only tool having facilities for proving outermost termination. Its approach is a stand alone one, which does not make use of the huge effort of the last years to improve the power of termination tools. For making use of the impressive power of termination tools, the natural approach is to make a transformation from TRSs to TRSs such that the modified termination property (in our case outermost termination) of the original TRS can be concluded from termination of the transformed TRS. In the past, a similar approach was successfully applied to context-sensitive termination [5] and liveness problems [8].

The Termination Problem DataBase (TPDB) [14] already contains 6 outermost examples that really require a consideration of the outermost strategy. If no strategy is regarded, then all of these examples are non-terminating. For these examples, both Cariboo and our presented transformation together with a termination prover for term rewrite systems without a strategy can prove outermost termination. As will be shown later, using the presented transformation we can prove outermost termination for systems where Cariboo fails to do so. Therefore, this approach increases the number of term rewrite systems that can be shown outermost terminating. However, it is not the case that it supersedes Cariboo: There are also examples where Cariboo succeeds while the transformed TRS cannot be shown terminating by any of the termination provers that we tried.

The presented approach deals with ground outermost termination. We will see that when fixing the signature there may be a difference between outermost termination and ground outermost termination, but by adding fresh constants there is no difference any more. Therefore it is not a restriction to focus on ground outermost termination.

The crucial ingredient of our transformation is *anti-matching*: for L being a set of terms such that it matches all terms that can be rewritten with the given TRS, we need a set S_L such that any term matches with a term in S_L if and only if it does not match with a term in L. It turns out that if all terms in L are linear, then a finite set S_L satisfying this requirement can easily be constructed, while there are sets L containing non-linear terms such that every set S_L satisfying this property is infinite. That's why we restrict to the class of *quasi left-linear* TRSs, which are all

TRSs where a left-hand side is an instance of a linear left-hand side. Clearly, this class also includes all left-linear TRSs.

Based on this anti-matching we give a straightforward transformation T such that every infinite outermost reduction with respect to any quasi left-linear TRS R gives rise to an infinite $T(R)$-reduction. We experimented with several variants of the basic transformation and chose the most powerful one in our final definition of the transformation.

This paper is structured as follows: After introducing the used notations in Section 2, we present our transformation and prove soundness in Section 3. There, we assume that we can construct a set of terms that match those terms not being matched by a left-hand side. This problem of anti-matching is treated in Section 4, which proves that our transformation can be applied automatically for quasi left-linear TRSs. In Section 5 we give a short description of our implementation of the transformation and present a number of examples. We conclude in Section 6.

2 Preliminaries

This section shall briefly introduce the notations used in this paper. For an introduction of term rewriting see for example [1,16].

We consider the set $\mathcal{T}(\Sigma, \mathcal{V})$ of all *terms* over a set $\mathcal{V}$ of *variables* and a finite, non-empty set Σ of *function symbols*, called *signature*, where each $f \in \Sigma$ is associated with a natural number called its *arity*. If the arity of $f \in \Sigma$ is $n \in \mathbb{N}$, then we denote this by $\text{arity}(f) = n$. Instead of $\mathcal{T}(\Sigma, \emptyset)$ we also write $\mathcal{T}(\Sigma)$, which we call the set of *ground terms*. For a term $t \in \mathcal{T}(\Sigma, \mathcal{V})$ we write $\mathcal{V}(t)$ to denote the set of variables occurring in t. For a non-variable term $t = f(t_1, \ldots, t_n)$ we say that f is the *root* of t, denoted by $\mathsf{root}(t)$. A term $t \in \mathcal{T}(\Sigma, \mathcal{V})$ is called *linear* if every variable occurs at most once in t; we write $\mathcal{T}_{\text{lin}}(\Sigma, \mathcal{V})$ for the set of all linear terms in $\mathcal{T}(\Sigma, \mathcal{V})$.

A *position* of a term $t \in \mathcal{T}(\Sigma, \mathcal{V})$ is a sequence of natural numbers, the set of all positions of a term t is denoted $Pos(t)$. The empty sequence is denoted as ϵ. Such a position $\pi \in Pos(t)$ identifies a *subterm* of t, which is written $t|_\pi$. The term that we get from replacing the subterm $t|_\pi$ by another term $s \in \mathcal{T}(\Sigma, \mathcal{V})$ is denoted $t[s]_\pi$.

A *substitution* $\sigma : \mathcal{V} \to \mathcal{T}(\Sigma, \mathcal{V})$ is written as $\sigma = \{x_1 := t_1, \ldots, x_m := t_m\}$, which denotes the mapping $\sigma(x) = x$ and $\sigma(x_i) = t_i$ for all $x \neq x_i$ and $1 \leq i \leq m$. The set of all substitutions over Σ and $\mathcal{V}$ is denoted as $\text{SUB}(\Sigma, \mathcal{V})$. The application of a substitution $\sigma \in \text{SUB}(\Sigma, \mathcal{V})$ to a term $t \in \mathcal{T}(\Sigma, \mathcal{V})$ is denoted $t\sigma$ and replaces all variables by their corresponding terms. Such a term $t\sigma$ is called an *instance* of t. Two terms $s, t \in \mathcal{T}(\Sigma, \mathcal{V})$ are said to *unify*, if a *unifier* $\sigma \in \text{SUB}(\Sigma, \mathcal{V})$ exists such that $s\sigma = t\sigma$. A term $s \in \mathcal{T}(\Sigma, \mathcal{V})$ is said to *match* a term $t \in \mathcal{T}(\Sigma, \mathcal{V})$, if a substitution $\sigma \in \text{SUB}(\Sigma, \mathcal{V})$ exists such that $s\sigma = t$.

A standard property relating linearity and unification is the following.

Lemma 2.1 *Let $t, u \in \mathcal{T}_{\text{lin}}(\Sigma, \mathcal{V})$ be two linear terms with $\mathcal{V}(t) \cap \mathcal{V}(u) = \emptyset$ that do not unify. Then there is a position $\pi \in Pos(t) \cap Pos(u)$ such that $t|_\pi$ and $u|_\pi$ are no variables and $\mathsf{root}(t|_\pi) \neq \mathsf{root}(u|_\pi)$.*

Proof (Sketch). We refer to [16, Section 7.7], where the standard Martelli-Montanari unification algorithm is given. For linear terms $t, u \in \mathcal{T}_{\text{lin}}(\Sigma, \mathcal{V})$ with

$\mathcal{V}(t) \cap \mathcal{V}(u) = \emptyset$ we have as invariant of the algorithm that every variable occurs at most once. Thus, the only way to get the result **fail** is by having two terms that have different root symbols. □

A *Term Rewrite System (TRS)* is a set $R \subseteq \mathcal{T}(\Sigma, \mathcal{V}) \times \mathcal{T}(\Sigma, \mathcal{V})$, where instead of $(\ell, r) \in R$ we write $\ell \to r \in R$. The set of left-hand sides of a TRS R is denoted $\text{lhs}(R)$ and is defined as $\text{lhs}(R) = \{\ell \mid \ell \to r \in R\}$. A TRS is called *left-linear* if ℓ is linear for all $\ell \in \text{lhs}(R)$ and we call it *quasi left-linear* if every $\ell \in \text{lhs}(R)$ is an instance of a linear $\ell' \in \text{lhs}(R)$. A term $s \in \mathcal{T}(\Sigma, \mathcal{V})$ *rewrites* to a term $t \in \mathcal{T}(\Sigma, \mathcal{V})$ with a rule $\ell \to r \in R$ at a position $\pi \in Pos(s)$, denoted by $s \to_{\ell \to r, \pi} t$, iff there exists a substitution $\sigma \in \text{SUB}(\Sigma, \mathcal{V})$ such that $s|_\pi = \ell\sigma$ and $t = s[r\sigma]_\pi$. The term $s|_\pi$ is called *redex*. Instead of $s \to_{\ell \to r, \pi} t$ we also write $s \to_{R,\pi} t$, $s \to_R t$, $s \to_\pi t$, or $s \to t$ if the term rewrite system R and/or the position π are clear from the context. If for a term $s \in \mathcal{T}(\Sigma, \mathcal{V})$ and a position $\pi \in Pos(s)$ we have for all $t \in \mathcal{T}(\Sigma, \mathcal{V})$ that $s \not\to_{R,\pi} t$ holds, then we also write $s \not\to_{R,\pi}$ or $s \not\to_\pi$.

A term $s \in \mathcal{T}(\Sigma, \mathcal{V})$ *outermost rewrites* to a term $t \in \mathcal{T}(\Sigma, \mathcal{V})$ with a rule $\ell \to r \in R$ at a position $\pi \in Pos(s)$, denoted $s \xrightarrow{o}_{\ell \to r, \pi} t$, iff $s \to_{\ell \to r, \pi} t$ and for all positions $\pi' \in Pos(s)$ with $\pi = \pi' \pi''$ for some $\pi'' \in \mathbb{N}^*$ with $\pi'' \neq \epsilon$ we have that $s \not\to_{R,\pi'}$.

A TRS R is called *terminating* (*outermost terminating*), iff there is no infinite sequence $t_1, t_2, t_3, \ldots \in \mathcal{T}(\Sigma, \mathcal{V})$ of terms with $t_i \to_R t_{i+1}$ ($t_i \xrightarrow{o}_R t_{i+1}$) for all $i \in \mathbb{N}$. A TRS R is called *ground terminating* (*outermost ground terminating*), iff there is no infinite sequence $t_1, t_2, t_3 \ldots \in \mathcal{T}(\Sigma)$ of ground terms such that $t_i \to_R t_{i+1}$ ($t_i \xrightarrow{o}_R t_{i+1}$) for all $i \in \mathbb{N}$.

The following example shows that outermost termination for arbitrary terms may differ from outermost ground termination.

Example 2.2 Consider the following term rewrite system R over the signature $\Sigma = \{\mathsf{f}, \mathsf{a}, \mathsf{b}\}$:

$$R = \begin{cases} \mathsf{f}(\mathsf{a}, x) & \to & \mathsf{a} & \qquad \mathsf{f}(x, \mathsf{a}) & \to & \mathsf{f}(x, \mathsf{b}) \\ \mathsf{f}(\mathsf{b}, x) & \to & \mathsf{a} & \qquad \mathsf{b} & \to & \mathsf{a} \\ \mathsf{f}(\mathsf{f}(x, y), z) & \to & \mathsf{a} & & & \end{cases}$$

For arbitrary terms we have the infinite outermost reduction

$$\mathsf{f}(x, \mathsf{a}) \to \mathsf{f}(x, \mathsf{b}) \to \mathsf{f}(x, \mathsf{a}) \to \cdots$$

However, when instantiating x by any arbitrary ground term $t \in \mathcal{T}(\Sigma)$, then one of the three rules on the left is applicable at the root position. Hence, the reduction $\mathsf{f}(t, \mathsf{b}) \to \mathsf{f}(t, \mathsf{a})$ is no longer an outermost reduction. In fact, the above term rewrite system is outermost ground terminating, as we will show later.

However, this difference only occurs when fixing the signature. It is easy to see that by replacing variables in any infinite outermost reduction by fresh constants, the result is an infinite outermost ground reduction. For quasi left-linear TRSs adding one fresh constant suffices.

3 The Transformation

The idea of the transformation is to only allow a reduction when a certain control symbol down marks the current redex. After having reduced a term, the control symbol is replaced by another control symbol up that is moved outwards. Only when the root of the term is encountered, then the control symbol is replaced by the down symbol again. In order to find the next outermost redex, the symbol down may only descend into subterms when no left-hand side is applicable to the term. For this purpose, we need a set S such that its elements match exactly those terms that are not matched by a left-hand side. For quasi left-linear TRSs such a set always exists and can be constructed automatically.

Theorem 3.1 *For a quasi left-linear TRS R, there exists a finite, computable, and (up to variable renaming) unique set $S \subseteq \mathcal{T}_{\text{lin}}(\Sigma, \mathcal{V})$ of linear terms such that for all ground terms $t \in \mathcal{T}(\Sigma)$ the two statements are equivalent:*

- *For all $\ell \in \text{lhs}(R)$ and all $\tau \in \text{SUB}(\Sigma, \mathcal{V})$ we have $t \neq \ell\tau$,*
- *$t = s\sigma$ for some $s \in S$ and some $\sigma \in \text{SUB}(\Sigma, \mathcal{V})$.*

The proof of this theorem is given in the following section. Using such a set, we can now define the transformation formally.

Definition 3.2 Let R be a TRS over a signature Σ. Let $S \subseteq \mathcal{T}(\Sigma, \mathcal{V})$ be such that for all $t \in \mathcal{T}(\Sigma)$ we have that $s \in S$ and $\sigma \in \text{SUB}(\Sigma, \mathcal{V})$ exist with $s\sigma = t$, iff for all $\ell \in \text{lhs}(R)$ and all $\tau \in \text{SUB}(\Sigma, \mathcal{V})$ we have $\ell\tau \neq t$. Choose four fresh unary symbols $\mathsf{top}, \mathsf{up}, \mathsf{down}, \mathsf{block} \notin \Sigma$, and let $\Sigma^\natural = \{f^\natural \mid f \in \Sigma, \text{arity}(f) > 0\}$ be such that $\Sigma \cap \Sigma^\natural = \emptyset$. The TRS $T(R)$ over the signature $\Sigma \cup \Sigma^\natural \cup \{\mathsf{top}, \mathsf{up}, \mathsf{down}, \mathsf{block}\}$ is defined to consist of the following rules:

- $\mathsf{down}(\ell) \to \mathsf{up}(r)$, for all rules $\ell \to r$ of R;
- $\mathsf{top}(\mathsf{up}(x)) \to \mathsf{top}(\mathsf{down}(x))$;
- $\mathsf{down}(f(t_1, \ldots, t_n)) \to f^\natural(\mathsf{block}(t_1), \ldots, \mathsf{down}(t_i), \ldots, \mathsf{block}(t_n))$, for all $f(t_1, \ldots, t_n) \in S$ and all $i \in \{1, \ldots, n\}$;
- $f^\natural(\mathsf{block}(x_1), \ldots, \mathsf{up}(x_i), \ldots, \mathsf{block}(x_n)) \to \mathsf{up}(f(x_1, \ldots, x_n))$, for all $f \in \Sigma$ and all $i \in \{1, \ldots, n\}$, where $\text{arity}(f) = n$ and $x_1, \ldots, x_n$ are distinct variables.

Because of Theorem 3.1, this transformation can be performed automatically for any finite quasi left-linear TRS. For an infinite TRS R, we clearly have that $T(R)$ is infinite, too. The TRS $T(R)$ can also become infinite for a TRS R that is not quasi left-linear, since then any set S satisfying the required property of the above definition might be infinite. This is demonstrated in the following section. However, the soundness of the transformation does not depend on this restriction.

In the transformation, we introduce the already mentioned symbols down and up to control the position of the next redex. The symbol top is used to denote the root position of the term, where the search of the next redex has to turn downwards again. The last of these fresh control symbols is the symbol block. Its purpose is to disallow evaluations where an up symbol appears at the root position of a term without having applied a rule of the form $\mathsf{down}(\ell) \to \mathsf{up}(r)$. Furthermore, we create a new marked symbol $f^\natural$ for every function symbol f of the original rewrite system,

having the same arity as f. This allows to use different interpretations for these symbols in the often used reduction pairs. Thus, it can be distinguished whether a symbol is above or below one of the control symbols.

An outermost rewrite step can be modelled by a sequence of steps in the transformed system. This is shown in the following lemma.

Lemma 3.3 *Let R be a TRS over a signature Σ, let $u, v \in \mathcal{T}(\Sigma)$.*
If $u \xrightarrow{o}_R v$, then $\mathsf{down}(u) \to^+_{T(R)} \mathsf{up}(v)$.

Proof. Let $u, v \in \mathcal{T}(\Sigma)$ be two ground terms, let $u \xrightarrow{o}_{\ell \to r, \pi} v$ for some rule $\ell \to r \in R$ and some position $\pi \in Pos(u)$. Induction is done over the length of π.

In case $\pi = \epsilon$, we directly have the rule $\mathsf{down}(\ell) \to \mathsf{up}(r) \in T(R)$, which shows the desired property.

Otherwise, let $\pi = i\,\pi'$ for some $\pi' \in Pos(u|_i)$ and let $u = f(u_1, \ldots, u_n)$. Hence, $u \not\to_\epsilon$ and $v = f(u_1, \ldots, u_{i-1}, v_i, u_{i+1}, \ldots, u_n)$ for some $v_i \in \mathcal{T}(\Sigma)$. Since u is a ground term, we have $u\tau = u$ and $u\tau = u \neq \ell'\sigma$ for all $\tau, \sigma \in \mathrm{SUB}(\Sigma, \mathcal{V})$ and all $\ell' \in \mathrm{lhs}(R)$. Hence, from the requirement in Definition 4.1 we get that there is a term $s \in S$ and a substitution $\sigma \in \mathrm{SUB}(\Sigma)$ such that $s\sigma = u$. Let $s = f(s_1, \ldots, s_n)$. Then a rule $\mathsf{down}(f(s_1, \ldots, s_n)) \to f^\natural(\mathsf{block}(s_1), \ldots, \mathsf{down}(s_i), \ldots, \mathsf{block}(s_n)) \in T(R)$ exists that is applicable to the term $\mathsf{down}(u)$ and therefore gives the reduction $\mathsf{down}(u) = \mathsf{down}(f(u_1, \ldots, u_n)) \to f^\natural(\mathsf{block}(u_1), \ldots, \mathsf{down}(u_i), \ldots, \mathsf{block}(u_n))$. For the subterm u_i we have $u_i \xrightarrow{o}_{\ell \to r, \pi'} v_i$. Here, the induction hypothesis is applicable and yields a reduction $\mathsf{down}(u_i) \to^+ \mathsf{up}(v_i)$. When applying this together with the rule $f^\natural(\mathsf{block}(x_1), \ldots, \mathsf{up}(x_i), \ldots, \mathsf{block}(x_n)) \to \mathsf{up}(f(x_1, \ldots, x_n)) \in T(R)$ the desired result can be shown:

$$\begin{aligned} \mathsf{down}(u) = \mathsf{down}(f(u_1, \ldots, u_n)) &\to f^\natural(\mathsf{block}(u_1), \ldots, \mathsf{down}(u_i), \ldots, \mathsf{block}(u_n)) \\ &\to^+ f^\natural(\mathsf{block}(u_1), \ldots, \mathsf{up}(v_i), \ldots, \mathsf{block}(u_n)) \\ &\to \mathsf{up}(f(u_1, \ldots, u_{i-1}, v_i, u_{i+1}, \ldots, u_n)) = \mathsf{up}(v) \end{aligned}$$

□

Using the above lemma we can prove the main theorem which shows that the presented transformation is sound, i.e., from the termination of the transformed TRS the outermost ground termination of the original TRS can be concluded.

Theorem 3.4 *Let R be a TRS over a signature Σ for which $T(R)$ is terminating. Then R is outermost ground terminating.*

Proof. Assume R is not outermost ground terminating. Then there is an infinite outermost reduction $t_1 \xrightarrow{o}_R t_2 \xrightarrow{o}_R \ldots$ for some ground terms $t_1, t_2, \ldots \in \mathcal{T}(\Sigma)$. For each $t_i \xrightarrow{o}_R t_{i+1}$ we have that $\mathsf{down}(t_i) \to^+_{T(R)} \mathsf{up}(t_{i+1})$ by Lemma 3.3. Due to the rule $\mathsf{top}(\mathsf{up}(x)) \to \mathsf{top}(\mathsf{down}(x))$ we obtain an infinite reduction in the transformed system $T(R)$,

$$\mathsf{top}(\mathsf{down}(t_1)) \to^+_{T(R)} \mathsf{top}(\mathsf{up}(t_2)) \to_{T(R)} \mathsf{top}(\mathsf{down}(t_2)) \to^+_{T(R)} \ldots,$$

contradicting termination of $T(R)$. □

Let us now remark on the situation when (arbitrary) outermost termination shall be considered, not just outermost ground termination. For a quasi left-linear TRS R over the signature Σ we create a new TRS R' which has the same rules as R, but now is defined to be over the signature $\Sigma' = \Sigma \cup \{c\}$, where $c \notin \Sigma$ is a fresh constant (i.e., it has arity 0). Then an infinite reduction $t_1 \xrightarrow{o}_R t_2 \xrightarrow{o}_R t_3 \xrightarrow{o}_R \dots$ for some terms $t_1, t_2, t_3, \dots \in \mathcal{T}(\Sigma, \mathcal{V})$ implies that $t_1\sigma_1 \xrightarrow{o}_{R'} t_2\sigma_2 \xrightarrow{o}_{R'} t_3\sigma_3 \xrightarrow{o}_{R'} \dots$ is an infinite reduction of ground terms $t_i\sigma_i \in \mathcal{T}(\Sigma)$ for substitutions $\sigma_i = \{x := c \mid x \in \mathcal{V}(t_i)\}$ for $i \in \mathbb{N}$. This holds, since no left-hand side of the rewrite system R' matches a subterm of $t_i\sigma_i$ which R does not match, because no left-hand side of R' contains the constant c. In the other direction, one can replace a symbol c in an infinite reduction w.r.t. R' by a fresh variable, giving an infinite reduction w.r.t. R. Therefore, we have that the term rewrite system R is outermost terminating, iff the term rewrite system R' is outermost ground terminating. Such a TRS R' can then be handled by our transformation to show outermost termination of R.

4 Anti-matching

We consider a term rewrite system R and consider a set L matching all terms that can be rewritten by R, for example $L = \text{lhs}(R)$. For our transformation of this term rewrite system we have to find a set S_L of terms that describe the terms which cannot be rewritten by R. Clearly, this is only depending on the left-hand sides. Rewriting w.r.t. a TRS is done by matching the left-hand sides to some other terms. Thus, we want to find a set S_L of terms that match those ground terms not matched by the terms contained in L. One can imagine that there are several possible sets that satisfy this condition. Our goal is to select the smallest such set and to be able to construct it finitely when this is possible.

This section presents the general problem of finding a set of terms that match the terms not matched by some terms contained in another set. Only at the end of this section we will restrict ourselves to the case of linear terms, where we will see that for this restriction the set is finite and can be computed.

The problem of finding a set of terms that describe the complement of a set of terms is similar to the problem considered by Lassez and Marriot [13], where an explicit representation of a set is being searched that is described using counter examples. But their focus is on machine learning, therefore it is hard to directly apply their results. We also want to mention the concept of *anti-patterns* as introduced in [11]. This is more general since it allows to introduce negation of patterns at any position in a term, while we are only interested in the negation of a complete pattern. However, this work is not applicable here, since we want a representation of a set that does not match a given term, while an anti-pattern matching problem is to decide whether an anti-pattern matches a single ground term.

Below, we first define a set S'_L of terms that satisfy the desired property. This set is usually infinite and contains quite a number of redundant terms, i.e., terms that are already matched by other terms contained in S'_L. Thus, we define another set S_L that consists only of the minimal elements of S'_L w.r.t. an order that expresses whether one term matches the other.

Definition 4.1 Let $L \subseteq \mathcal{T}(\Sigma, \mathcal{V})$ be a set of terms. On terms we define the preorder $\leq$ by

$$t \leq u \iff \exists \sigma : t\sigma = u$$

which induces the definition of its strict part to be

$$t < u \iff t \leq u \wedge \neg(u \leq t).$$

Now S_L is defined to be the set of minimal elements of the set of terms that do not unify with elements of L, i.e.,

$$\begin{aligned} S'_L &= \{t \in \mathcal{T}(\Sigma, \mathcal{V}) \mid \nexists \ell \in L, \sigma, \tau \in \mathrm{SUB}(\Sigma, \mathcal{V}) : \ell\sigma = t\tau\} \\ S_L &= \{t \in S'_L \mid \nexists u \in S'_L : u < t\} \end{aligned}$$

One might wonder why unification is considered, while term rewriting is concerned with matching. This becomes clear when formulating what kind of terms we are looking for: the set of terms that *match* those terms which are not *matched* by left-hand sides. This means we have to consider two matchings at the same time, when assuming that the set of variables are disjoint then this gives rise to a unification problem.

As a next step we show that the set S'_L is closed under substitution. This is of interest, since we want to consider the ground terms that are matched by a term contained in S'_L. Thus, it should be the case that every instantiation of a term from S'_L is also contained in S'_L, such that especially this holds for ground instances.

Lemma 4.2 $\{s\sigma \in \mathcal{T}(\Sigma, \mathcal{V}) \mid s \in S'_L, \sigma \in \mathrm{SUB}(\Sigma, \mathcal{V})\} = S'_L$.

Proof. "$\supseteq$": Holds trivially for $\sigma = id$.

"$\subseteq$": Let $s \in S'_L, \sigma \in \mathrm{SUB}(\Sigma, \mathcal{V})$. Then $s\sigma \in \{s\sigma \in \mathcal{T}(\Sigma, \mathcal{V}) \mid s \in S'_L, \sigma \in \mathrm{SUB}(\Sigma, \mathcal{V})\}$. We have that $\ell\sigma' \neq s\tau'$ for all $\ell \in L$ and all substitutions $\sigma', \tau' \in \mathrm{SUB}(\Sigma, \mathcal{V})$. Therefore, especially $\ell\sigma' \neq s\sigma\tau'$ holds for all substitutions $\sigma', \tau' \in \mathrm{SUB}(\Sigma, \mathcal{V})$. Hence, $s\sigma \in S'_L$. □

The set S_L is derived from the set S'_L by taking only the minimal elements of S'_L w.r.t. the order $>$. In order to be able to show that these minimal elements exist, we have to first show that this order is well-founded, i.e., there are no infinite descending chains.

Lemma 4.3 *The relation $>$ from Definition 4.1 is well-founded.*

Proof. Assume, $>$ was not well-founded, i.e., there exists an infinite sequence $t_1 > t_2 > \ldots$ for some $t_1, t_2, \ldots \in \mathcal{T}(\Sigma, \mathcal{V})$. Thus, for every $i > 1$ we have $t_{i-1}\tau \neq t_i$ for all $\tau \in \mathrm{SUB}(\Sigma, \mathcal{V})$ and substitutions $\sigma_i \in \mathrm{SUB}(\Sigma, \mathcal{V})$ exist such that $t_{i-1} = t_i\sigma_i$. For a term $t \in \mathcal{T}(\Sigma, \mathcal{V})$, let $\#_\Sigma(t) \in \mathbb{N}$ denote the number of symbols from Σ contained in t and let $\#_{2\times\mathcal{V}}(t) \in \mathbb{N}$ denote the number of variables that occur more than once in t. If for a variable $x_i \in \mathcal{V}(t_i)$ we have $\sigma_i(x_i) \notin \mathcal{V}$, then we see that $\#_\Sigma(t_{i-1}) = \#_\Sigma(t_i\sigma_i) >_\mathbb{N} \#_\Sigma(t_i)$. Otherwise, we have $\sigma_i(x) \in \mathcal{V}$ for all $x \in \mathcal{V}$. Then there must be two variables $x_i, y_i \in \mathcal{V}(t_i)$ with $x_i \neq y_i$ and $\sigma_i(x_i) = \sigma_i(y_i) \in \mathcal{V}$, since otherwise we could define $\tau = \{y := x \mid \sigma_i(x) = y\}$ and would get $t_{i-1}\tau = t_i$. Hence,

in this case we see that $\#_\Sigma(t_{i-1}) = \#_\Sigma(t_i)$ and $\#_{2\times\mathcal{V}}(t_{i-1}) >_\mathbb{N} \#_{2\times\mathcal{V}}(t_i)$. Because the lexicographic combination of two well-founded orders is also well-founded, we have a contradiction since we have constructed an infinite descending chain for the lexicographic combination of $>_\mathbb{N}$ on $\#_\Sigma$ and $>_\mathbb{N}$ on $\#_{2\times\mathcal{V}}$. □

When removing larger elements from a set w.r.t. $\leq$, then all terms that are matched by removed terms are still being matched by some term in the set. This is proven in the next lemma and will be used to show that S_L still matches the same terms as S'_L.

Lemma 4.4 $\{u\sigma \mid u \in U, \sigma \in \mathrm{SUB}(\Sigma,\mathcal{V})\} = \{u\sigma \mid u \in U \cup U', \sigma \in \mathrm{SUB}(\Sigma,\mathcal{V})\}$ *for every* $U, U' \subseteq \mathcal{T}(\Sigma,\mathcal{V})$ *with* $U' = \{u' \mid \exists u \in U : u \leq u'\}$.

Proof. "$\subseteq$": trivial, since $U \subseteq U \cup U'$.

"$\supseteq$": Let $u' \in U \cup U'$, let $\sigma' \in \mathrm{SUB}(\Sigma,\mathcal{V})$. If $u' \in U$, then the property trivially holds. So let $u' \in U' \setminus U$. Then a $u \in U$ exists such that $u \leq u'$, i.e., there is a substitution $\tau \in \mathrm{SUB}(\Sigma,\mathcal{V})$ such that $u' = u\tau$. Hence, $u'\sigma' = u\tau\sigma' \in \{u\sigma \mid u \in U, \sigma \in \mathrm{SUB}(\Sigma,\mathcal{V})\}$. □

For the set S'_L it should be intuitively clear that all terms that are not matched by a term contained in L are matched by a term in that set. Using the above lemma, we can now show that this already holds for the set S_L.

Lemma 4.5 $\{s\sigma \in \mathcal{T}(\Sigma,\mathcal{V}) \mid s \in S_L, \sigma \in \mathrm{SUB}(\Sigma,\mathcal{V})\} = S'_L$.

Proof. "$\subseteq$": Since $S_L \subseteq S'_L$, this holds due to Lemma 4.2.

"$\supseteq$": Since $>$ is well-founded as shown in Lemma 4.3, the existence of the minimal elements in S_L is guaranteed. Thus, Lemma 4.4 shows the desired property. □

This allows us to prove that the ground terms matched by S_L are indeed those terms that are not matched by the set L.

Lemma 4.6 $\mathcal{T}(\Sigma) \setminus \{\ell\sigma \in \mathcal{T}(\Sigma) \mid \ell \in L, \sigma \in \mathrm{SUB}(\Sigma,\mathcal{V})\} = \{s\sigma \in \mathcal{T}(\Sigma) \mid s \in S_L, \sigma \in \mathrm{SUB}(\Sigma,\mathcal{V})\}$.

Proof. This lemma is shown in two steps: First it is proven that $\{\ell\sigma \in \mathcal{T}(\Sigma) \mid \ell \in L, \sigma \in \mathrm{SUB}(\Sigma,\mathcal{V})\} \cap \{s\sigma \in \mathcal{T}(\Sigma) \mid s \in S_L, \sigma \in \mathrm{SUB}(\Sigma,\mathcal{V})\} = \emptyset$ (showing "$\supseteq$"), and in the second step it is shown that $\mathcal{T}(\Sigma) = \{\ell\sigma \in \mathcal{T}(\Sigma) \mid \ell \in L, \sigma \in \mathrm{SUB}(\Sigma,\mathcal{V})\} \cup \{s\sigma \in \mathcal{T}(\Sigma) \mid s \in S_L, \sigma \in \mathrm{SUB}(\Sigma,\mathcal{V})\}$ (showing "$\subseteq$").

For the first step, let $t \in \{\ell\sigma \in \mathcal{T}(\Sigma) \mid \ell \in L, \sigma \in \mathrm{SUB}(\Sigma,\mathcal{V})\} \cap \{s\sigma \in \mathcal{T}(\Sigma) \mid s \in S_L, \sigma \in \mathrm{SUB}(\Sigma,\mathcal{V})\}$. Thus, there exist $\ell \in L$ and $\sigma_\ell \in \mathrm{SUB}(\Sigma,\mathcal{V})$ such that $t = \ell\sigma_\ell$ and there exist $s \in S_L \subseteq S'_L$ and $\sigma_s \in \mathrm{SUB}(\Sigma,\mathcal{V})$ such that $t = s\sigma_s$. Putting this together gives $\ell\sigma_\ell = t = s\sigma_s$, which is a contradiction to the definition of S'_L.

To show the second step, we observe that clearly $\{\ell\sigma \in \mathcal{T}(\Sigma) \mid \ell \in L, \sigma \in \mathrm{SUB}(\Sigma,\mathcal{V})\} \cup \{s\sigma \in \mathcal{T}(\Sigma) \mid s \in S_L, \sigma \in \mathrm{SUB}(\Sigma,\mathcal{V})\} \subseteq \mathcal{T}(\Sigma)$. So it remains to be shown that $\{\ell\sigma \in \mathcal{T}(\Sigma) \mid \ell \in L, \sigma \in \mathrm{SUB}(\Sigma,\mathcal{V})\} \cup \{s\sigma \in \mathcal{T}(\Sigma) \mid s \in S_L, \sigma \in \mathrm{SUB}(\Sigma,\mathcal{V})\} \supseteq \mathcal{T}(\Sigma)$. For that purpose, let $t \in \mathcal{T}(\Sigma)$ be an arbitrary ground term. In case there exist $\ell \in L$ and $\sigma_\ell \in \mathrm{SUB}(\Sigma,\mathcal{V})$ such that $\ell\sigma_\ell = t$ the property has been shown. Otherwise, we may assume that for all $\ell \in L$ and all substitutions $\sigma \in \mathrm{SUB}(\Sigma,\mathcal{V})$ that satisfy $\ell\sigma \in \mathcal{T}(\Sigma)$ we have $\ell\sigma \neq t$. Since t is a ground term,

$\mathcal{V}(t) = \emptyset$ holds. This means that for any substitution $\tau \in \mathrm{SUB}(\Sigma, \mathcal{V})$ we have $t\tau = t$. Furthermore, for every term $t' \in \mathcal{T}(\Sigma, \mathcal{V})$ with $\mathcal{V}(t') \neq \emptyset$ it holds that $t \neq t'$ which allows to conclude that $t \neq \ell\sigma'$ for all substitutions $\sigma' \in \mathrm{SUB}(\Sigma, \mathcal{V})$ where $\mathcal{V}(\ell\sigma') \neq \emptyset$. Putting this together, we get that for all substitutions $\sigma, \tau \in \mathrm{SUB}(\Sigma, \mathcal{V})$ it holds that $\ell\sigma \neq t\tau = t$. From the definition of S'_L we get $t \in S'_L$, and hence $t \in \{s\sigma \in \mathcal{T}(\Sigma) \mid s \in S_L, \sigma \in \mathrm{SUB}(\Sigma, \mathcal{V})\}$ by Lemma 4.5. □

In the following we restrict ourselves to sets L that only contain linear terms. It should be observed that this also covers the case of a quasi left-linear TRS R: for such a TRS we can define L to be the set of all linear left-hand sides of R and have that L still matches the same terms as $\mathrm{lhs}(R)$, due to Lemma 4.4. We want to show that for a linear set L the set S_L is finite. For that purpose we need the depth of a term, which is defined as follows.

Definition 4.7 The *depth* of a term $t \in \mathcal{T}(\Sigma, \mathcal{V})$ is defined as $\mathrm{depth}(t) = 0$ if $t \in \mathcal{V}$ and $\mathrm{depth}(f(t_1, \ldots, t_n)) = 1 + \max\{\mathrm{depth}(t_1), \ldots, \mathrm{depth}(t_n)\}$ for $t = f(t_1, \ldots, t_n)$.

The *depth* of a finite set $T \subseteq \mathcal{T}(\Sigma, \mathcal{V})$ is defined as the maximum over the depths of the terms it contains, i.e., $\mathrm{depth}(T) = \max\{\mathrm{depth}(t) \mid t \in T\}$.

Then, we have that for example $\mathrm{depth}(\mathsf{f}(x, y)) = 1$, while $\mathrm{depth}(\mathsf{f}(\mathsf{a}, y)) = 2$ for the signature $\Sigma = \{\mathsf{f}, \mathsf{a}\}$. Using this notion of depth, we can now prove the following lemma. It provides an upper bound on the depth of the terms contained in S_L for sets L containing only linear terms.

Lemma 4.8 *For a set $L \subseteq \mathcal{T}_{\mathrm{lin}}(\Sigma, \mathcal{V})$ of linear terms we have that* $\mathrm{depth}(s) \leq \mathrm{depth}(L)$ *for all $s \in S_L$.*

Proof. Assume, there exists a $s \in S_L \subseteq S'_L$ with $\mathrm{depth}(s) > \mathrm{depth}(L)$. Thus, we have that $s\sigma \neq \ell\tau$ for all $\ell \in L$ and all substitutions $\sigma, \tau \in \mathrm{SUB}(\Sigma, \mathcal{V})$. W.l.o.g. we may assume that $\mathcal{V}(s)$ and $\mathcal{V}(\ell)$ are disjoint for all $\ell \in L$. Lemma 2.1 shows that for every $\ell \in L$ a position $\pi_\ell \in Pos(\ell) \cap Pos(s)$ exists such that $\mathsf{root}(\ell|_{\pi_\ell}) \neq \mathsf{root}(s|_{\pi_\ell})$. By definition of $\mathrm{depth}(L)$, we have $|\pi_\ell| < \mathrm{depth}(L)$. Let $\mathrm{trunc}_L(s) \in \mathcal{T}(\Sigma, \mathcal{V})$ denote the term that is derived from s by replacing all subterms at positions of length $\mathrm{depth}(L)$ by fresh variables. By construction, we have $\mathrm{depth}(\mathrm{trunc}_L(s)) = \mathrm{depth}(L)$, $\mathrm{trunc}_L(s) < s$, and $\mathsf{root}(s|_\pi) = \mathsf{root}(\mathrm{trunc}_L(s)|_\pi)$ for all $\pi \in Pos(s)$ with $|\pi| < \mathrm{depth}(L)$. Hence, $\mathsf{root}(\mathrm{trunc}_L(s)|_\pi) = \mathsf{root}(s|_\pi) \neq \mathsf{root}(\ell|_{\pi_\ell})$, i.e., for all substitutions $\sigma, \tau \in \mathrm{SUB}(\Sigma, \mathcal{V})$ we have $\mathrm{trunc}_L(s)\sigma \neq \ell\tau$. Thus $\mathrm{trunc}_L(s) \in S'_L$, which contradicts the minimality of s. □

Furthermore, we only have to consider linear terms for the set S_L, if we are only interested in the matching of ground terms.

Lemma 4.9 *For a set $L \subseteq \mathcal{T}_{\mathrm{lin}}(\Sigma, \mathcal{V})$ of linear terms, we have for every $t \in \mathcal{T}(\Sigma) \cap S'_L$ that $s \in S_L \cap \mathcal{T}_{\mathrm{lin}}(\Sigma, \mathcal{V})$ and $\sigma \in \mathrm{SUB}(\Sigma, \mathcal{V})$ exist with $s\sigma = t$.*

Proof. Let $t \in \mathcal{T}(\Sigma) \cap S'_L$. Then for all $\ell \in L$ and all $\tau \in \mathrm{SUB}(\Sigma, \mathcal{V})$ we have that $\ell\tau \neq t$. Since $\mathcal{T}(\Sigma) \subseteq \mathcal{T}_{\mathrm{lin}}(\Sigma, \mathcal{V})$, we get from Lemma 2.1 that a $\pi_\ell \in Pos(\ell) \cap Pos(t)$ exists with $\mathsf{root}(t|_{\pi_\ell}) \neq \mathsf{root}(\ell|_{\pi_\ell})$.

There exist $s \in S_L$ and $\sigma \in \mathrm{SUB}(\Sigma, \mathcal{V})$ with $s\sigma = t$, due to Lemma 4.5. In case $s \in \mathcal{T}_{\mathrm{lin}}(\Sigma, \mathcal{V})$ nothing has to be proven.

Otherwise, we start with the term $\text{lin}(s)$ that is created from s by replacing every occurrence of a variable by a fresh variable, thereby generating a linear term. Then clearly, there is a substitution σ' such that $\text{lin}(s)\sigma' = t$. If there is an $\ell' \in L$ and a substitution $\tau \in \text{SUB}(\Sigma, \mathcal{V})$ such that $\text{lin}(s)\tau = \ell'\tau$ (where we assume that $\mathcal{V}(\text{lin}(s)) \cap \mathcal{V}(\ell') = \emptyset$), then we replace the variable at a position π_s that is a prefix of $\pi_{\ell'}$ by $f(x_1, \ldots, x_n)$, where $f = \mathsf{root}(t|_{\pi_s})$, $\text{arity}(f) = n$, and the x_i are fresh variables. This variable must exist, otherwise ℓ' would match t. This process is repeated until there are no more ℓ' that unify with the thereby constructed term s'. By construction s' is linear and does not unify with any term from L. Furthermore, this term is minimal in S'_L w.r.t. $>$, therefore $s' \in S_L$, which shows our claim. $\Box$

From the above lemmas, we can give the following construction of a set S for a set L of linear terms that satisfies the requirement of Definition 3.2. Let $d = \text{depth}(L)$ be the maximal depth of terms occurring in L. Start by S' being the finite set of all linear terms up to renaming of variables of depth $\leq d$. Next remove all terms from S' that unify with L. Finally initialize S to S' and remove all non-minimal elements t from S, i.e., every term t for which a $u \in S$ exists with $u < t$ is removed from S. From Lemmas 4.8 and 4.9 we then know that all ground terms that are not matched by L are matched by S.

Using this construction and the above lemmas, we can now show Theorem 3.1. It states that for a quasi left-linear TRS R a finite, computable, and unique set S exists that matches exactly those terms that $\text{lhs}(R)$ does not match. Please note that we only have to consider a linear set L that matches all ground terms matched by $\text{lhs}(R)$, as we already observed above.

Proof of Theorem 3.1. Let $L \subseteq \mathcal{T}_{\text{lin}}(\Sigma, \mathcal{V})$ be the finite set of linear left-hand sides of R. Then L matches all terms that can be rewritten by R due to Lemma 4.4. Let $S_L \subseteq \mathcal{T}(\Sigma, \mathcal{V})$ be defined as given in Definition 4.1. As we can see from Lemma 4.6, we have that for all ground terms $t \in \mathcal{T}(\Sigma)$ it holds that $t \in \{s\sigma \in \mathcal{T}(\Sigma) \mid s \in S_L, \sigma \in \text{SUB}(\Sigma, \mathcal{V})\}$ iff $t \notin \{\ell\sigma \in \mathcal{T}(\Sigma) \mid \ell \in L, \sigma \in \text{SUB}(\Sigma, \mathcal{V})\}$. From Lemma 4.8 we get that S_L is finite, since, up to variable renaming, only finitely many terms whose depth is less than or equal to $\text{depth}(L)$ exist for a finite signature Σ. Lemma 4.9 shows that $S = S_L \cap \mathcal{T}_{\text{lin}}(\Sigma, \mathcal{V})$, and finally the sketched construction shows that the set S is computable and unique since the minimal elements w.r.t. $>$ are unique. $\Box$

Finally, we want to further analyze the case of TRSs that are not quasi left-linear. For this purpose, let $L = \{\mathsf{f}(x, x)\}$ be the left-hand sides of a TRS over the signature $\Sigma = \{\mathsf{f}, \mathsf{g}\}$. Then for every $n \in \mathbb{N}$ we have $\mathsf{f}(x, \mathsf{g}^n(x)) \in S'_L$. Furthermore, there is no term $s \neq \mathsf{f}(x, \mathsf{g}^n(x)) \in S'_L$ such that $s\sigma = \mathsf{f}(x, \mathsf{g}^n(x))$, which shows that S_L is infinite. To show that this is not due to choosing the set S_L, we prove the proposition below, stating that S_L is the smallest set that has the desired property.

Proposition 4.10 *Let $L \subseteq \mathcal{T}(\Sigma, \mathcal{V})$. For every $S \subseteq \mathcal{T}(\Sigma, \mathcal{V})$ that satisfies $\forall t \in \mathcal{T}(\Sigma) : (\exists s \in S, \sigma \in \text{SUB}(\Sigma, \mathcal{V}) : t = s\sigma) \iff \neg(\exists \ell \in L, \tau \in \text{SUB}(\Sigma, \mathcal{V}) : t = \ell\tau)$ we have $S_L \subseteq S \subseteq S'_L$, where we disregard variable renamings.*

Proof. The inclusion $S \subseteq S'_L$ can be seen directly from the definition of S'_L.

Assume, there is such a set $S \subseteq \mathcal{T}(\Sigma, \mathcal{V})$ with $S_L \not\subseteq S$. Then, there is a term $s' \in S_L$ such that $s' \notin S$. Furthermore, it must be the case that $\{s\sigma \mid s \in S, \sigma \in$

$\mathrm{SUB}(\Sigma, \mathcal{V})\} = \{s\sigma \mid s \in S_L, \sigma \in \mathrm{SUB}(\Sigma, \mathcal{V})\} = S'_L$, i.e., there must be an $s \in S$ and a $\sigma \in \mathrm{SUB}(\Sigma, \mathcal{V})$ such that $s\sigma = s'$. This implies that $s \leq s'$. In case we also have $s' \leq s$, then $s' \in S$, contradicting our assumption. But otherwise $s < s'$ holds, which contradicts the minimality of s'. □

As a consequence of Proposition 4.10 and the previously observed fact that for $L = \{\mathsf{f}(x, x)\}$ it holds that $S_L \supseteq \{\mathsf{f}(x, \mathsf{g}^n(x)) \mid n \in \mathbb{N}\}$, we conclude that any set S that matches those terms which are not matched by a term in L must be infinite, since already $S_L \subseteq S$ is infinite.

5 Implementation and Experiments

We have implemented the transformation as described in the previous sections. Even though the construction of the set S_L of terms that match those terms not matched by the set L of left-hand sides of the input term rewrite system can certainly be improved, the complete transformation only takes a neglegible amount of time for all of the following examples.

Our implementation allows for a number of different variants of the transformation to be used. In Section 3 only one of these was presented, this proved to be the most effective one in our experiments. In detail, one can chose whether or not to add the blocking symbol when the symbol down descends into a term that is not matched by a left-hand side of the original term rewrite system. Also, one can chose whether the symbols upon descending should be rewritten to a marked version of that symbol. As a last option, one can also use a modified version of the rules for the up symbol, however this modification proved itself not to be effective.

The transformed system was then used as input for the termination provers Jambox [3], TTT2 [12], and AProVE [6], which were the strongest tools of the 2007 termination competition in the TRS category [14]. The reason why we used multiple tools was that the transformation turned out to produce rewrite systems for which sometimes one tool succeeded in proving termination of the transformed TRS, while at least one of the other tools was unable to do so.

Below we present some examples. First, we want to show that Example 2.2 really is outermost ground terminating, as claimed above. When this example is transformed, the following TRS is created:

Example 5.1 *(Transformation of Example 2.2)*

$$T(R) = \begin{cases} \mathsf{top}(\mathsf{up}(x)) & \to & \mathsf{top}(\mathsf{down}(x)) & \mathsf{down}(\mathsf{b}) & \to & \mathsf{up}(\mathsf{a}) \\ \mathsf{down}(\mathsf{f}(x, \mathsf{a})) & \to & \mathsf{up}(\mathsf{f}(x, \mathsf{b})) & \mathsf{down}(\mathsf{f}(\mathsf{a}, x)) & \to & \mathsf{up}(\mathsf{a}) \\ \mathsf{f}^\natural(\mathsf{block}(x), \mathsf{up}(y)) & \to & \mathsf{up}(\mathsf{f}(x, y)) & \mathsf{down}(\mathsf{f}(\mathsf{b}, x)) & \to & \mathsf{up}(\mathsf{a}) \\ \mathsf{f}^\natural(\mathsf{up}(x), \mathsf{block}(y)) & \to & \mathsf{up}(\mathsf{f}(x, y)) & \mathsf{down}(\mathsf{f}(\mathsf{f}(x, y), z)) & \to & \mathsf{up}(\mathsf{a}) \end{cases}$$

It can be observed that in the transformed TRS there are no rules that allow the symbol down to descend into a term. This holds because we have $S_L = \{\mathsf{a}\}$, such that no rules are created for it. The transformed TRS can easily be shown terminating within a short amount of time by all of the considered tools. For the next example, this is not the case anymore.

Example 5.2

$$R_2 = \begin{cases} \mathsf{a} & \to & \mathsf{f}(\mathsf{a}) \\ \mathsf{f}(\mathsf{f}(\mathsf{f}(\mathsf{f}(\mathsf{f}(x))))) & \to & \mathsf{b} \end{cases}$$

Both AProVE and TTT2 can show termination of $T(R_2)$, while Jambox fails to do so. What is also interesting is that TTT2 uses RFC Match Bounds to show this, while AProVE uses only Dependency Pairs and a large number of rewriting steps, but is able to find this proof much faster than TTT2.

The next example proved to be rather difficult for all of the considered tools. It is the example from the introduction for the case $n = 1$.

Example 5.3

$$R_3 = \begin{cases} \mathsf{from}(x) & \to & x : \mathsf{from}(\mathsf{s}(x)) \\ \mathsf{s}(x) : xs & \to & \mathsf{overflow} \end{cases}$$

This example could only be proven terminating by the tool Jambox, both AProVE and TTT2 failed. However, the techniques used by Jambox to prove termination, namely semantic labelling and polynomial orders, are also implemented in both of the other tools. Hence, this clearly shows that proving termination is also strongly dependent on heuristics and/or search encodings.

In the examples that we considered so far, we had that always the right-hand side of the rule that caused the outermost ground termination was a ground term. This is different in the next example.

Example 5.4

$$R_4 = \begin{cases} \mathsf{f}(\mathsf{f}(\mathsf{g}(x))) & \to & x \\ \mathsf{g}(\mathsf{b}) & \to & \mathsf{f}(\mathsf{g}(\mathsf{b})) \end{cases}$$

This transformed TRS can be shown terminating by the tools TTT2 and Jambox, while AProVE fails.

In the example below, it is the case that the right-hand sides are not always either growing or detecting a term that has grown too large.

Example 5.5

$$R_5 = \begin{cases} \mathsf{f}(\mathsf{f}(x, y), z) & \to & \mathsf{c} \\ \mathsf{f}(x, \mathsf{f}(y, z)) & \to & \mathsf{f}(\mathsf{f}(x, y), z) \\ \mathsf{a} & \to & \mathsf{f}(\mathsf{a}, \mathsf{a}) \end{cases}$$

The transformed TRS $T(R_5)$ can be shown terminating by both AProVE and Jambox, while for this example TTT2 fails to show termination. If the first rule is changed to $\mathsf{f}(\mathsf{f}(x, y), z) \to \mathsf{f}(\mathsf{c}, x)$, then only AProVE can show the transformed TRS to be terminating.

Next, we want to consider the 6 examples contained in the Termination Problem DataBase (TPDB) [14] that require outermost termination analysis. All of these examples can be proven outermost ground terminating by the tool Cariboo [9], which was mentioned in the introduction. Of these 6 examples, 5 are left-linear, therefore they can be directly handled by our approach. For these, we can show outermost ground termination using Jambox as termination prover. The last example shall be considered in more detail below.

Example 5.6 *(Outermost Example 6)*

$$R_6 = \begin{cases} \mathsf{f}(x,x) & \to & \mathsf{f}(\mathsf{i}(x),\mathsf{g}(\mathsf{g}(x))) & \quad \mathsf{f}(x,\mathsf{i}(\mathsf{g}(x))) & \to & \mathsf{a} \\ \mathsf{f}(x,y) & \to & x & \quad \mathsf{f}(x,\mathsf{i}(x)) & \to & \mathsf{f}(x,x) \\ \mathsf{g}(x) & \to & \mathsf{i}(x) & & & \end{cases}$$

As can be seen above, this example has non-linear left-hand sides for the function symbol f. However, these left-hand sides are all instances of the left-hand side $\mathsf{f}(x,y)$, which makes this TRS quasi left-linear. Hence, we only have to consider the set $L = \{\mathsf{f}(x,y), \mathsf{g}(x)\}$ of linear terms, from which we then compute S_L to be $S_L = \{\mathsf{a}, \mathsf{i}(x)\}$. Using this set, our transformation yields a finite TRS $T(R_6)$, whose termination can be proven using any of the three considered tools.

Finally, we want to compare the strength of our approach against that of Cariboo. The following example is non-terminating for normal rewriting, since already the rule $\mathsf{h}(x) \to \mathsf{f}(\mathsf{h}(x))$ allows an infinite reduction.

Example 5.7

$$R_7 = \begin{cases} \mathsf{f}(\mathsf{h}(x)) & \to & \mathsf{f}(\mathsf{i}(x)) & \quad \mathsf{h}(x) & \to & \mathsf{f}(\mathsf{h}(x)) \\ \mathsf{f}(\mathsf{i}(x)) & \to & x & \quad \mathsf{i}(x) & \to & \mathsf{h}(x) \end{cases}$$

Cariboo is unable to prove outermost ground termination of the TRS R_7, while the transformed TRS $T(R_7)$ can be proven terminating by all considered tools. Also Example 5.4 and both variants of Example 5.5 cannot be proven outermost ground terminating by Cariboo.

There are also examples where Cariboo succeeds, whereas our transformation fails. First of all, Cariboo can also handle examples that are not quasi left-linear, while our transformation is not applicable in this case. But there are also quasi left-linear examples where Cariboo can prove outermost ground termination, but none of the considered tools can prove termination of the transformed TRS. Such an example is given below.

Example 5.8

$$R_8 = \begin{cases} \mathsf{from}(x) & \to & \mathsf{cons}(x, \mathsf{from}(\mathsf{s}(x))) \\ \mathsf{cons}(\mathsf{s}(\mathsf{s}(x)), xs) & \to & \mathsf{nil} \end{cases}$$

This example can be shown terminating by Cariboo, whereas for all termination provers the transformed TRS $T(R_8)$ is too hard. Please note that this is only a slightly modified version of Example 5.3, where instead of one s symbol now two such symbols are required.

6 Conclusion

We have presented a transformation such that outermost ground termination of a TRS follows from termination of the transformed TRS. This transformation is sound for arbitrary term rewrite systems, but only for finite quasi left-linear term rewrite systems the transformed term rewrite system is finite and can be constructed

automatically. For this class of term rewrite systems we implemented the transformation and we were able to prove ground outermost termination for a number of non-trivial examples. When comparing the presented approach to the existing one implemented in the tool Cariboo [9], then we have shown that our approach can prove term rewrite systems to be outermost ground terminating where the existing approach fails. However, there are also examples where Cariboo succeeds but our transformation fails. Especially, Cariboo is not limited to quasi left-linear TRSs, but also when considering only such TRSs there are examples showing this. However, some of these quasi left-linear examples might, due to the nature of our approach, be proven outermost ground terminating in the future using the presented transformation, when the underlying termination provers become even more powerful.

A next step is to consider whether the presented transformation is complete, i.e., whether from the non-termination of the transformed TRS one can conclude that the original TRS is not outermost ground terminating.

References

[1] Baader, F. and T. Nipkow, “Term Rewriting and All That,” Cambridge University Press, Cambridge, UK, 1998.

[2] Clavel, M., F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer and C. Talcott, *The Maude 2.0 System*, in: *Proceedings of the 14th Conference on Rewriting Techniques and Applications*, Lecture Notes in Computer Science **2706** (2003), pp. 76–87.

[3] Endrullis, J., *Jambox 2.0e*, downloadable from http://joerg.endrullis.de.

[4] Futatsugi, K. and R. Diaconescu, editors, “CafeOBJ Report,” World Scientific Publishing Company, 1998.

[5] Giesl, J. and A. Middeldorp, *Transformation Techniques for Context-Sensitive Rewrite Systems*, Journal of Functional Programming **14** (2004), pp. 379–427.

[6] Giesl, J., P. Schneider-Kamp and R. Thiemann, *AProVE 1.2: Automatic Termination Proofs in the Dependency Pair Framework*, in: *Proceedings of the 3rd International Joint Conference on Automatic Reasoning*, Lecture Notes in Computer Science **4130** (2006), pp. 281–286, downloadable from http://aprove.informatik.rwth-aachen.de.

[7] Giesl, J., S. Swiderski, P. Schneider-Kamp and R. Thiemann, *Automated Termination Analysis for Haskell: From Term Rewriting to Programming Languages*, in: *Proceedings of the 17th Conference on Rewriting Techniques and Applications*, Lecture Notes in Computer Science **4098**, 2006, pp. 297–312.

[8] Giesl, J. and H. Zantema, *Liveness in Rewriting*, in: *Proceedings of the 14th Conference on Rewriting Techniques and Applications*, Lecture Notes in Computer Science **2706** (2003), pp. 321–336.

[9] Gnaedig, I. and H. Kirchner, *Termination of rewriting under strategies*, ACM Transactions on Computational Logic (2007), to appear, available at http://tocl.acm.org/accepted/315gnaedig.ps.

[10] Jones, S. P., editor, “Haskell 98 Language and Libraries: The Revised Report,” Cambridge University Press, 2003, also available from http://www.haskell.org/definition.

[11] Kirchner, C., R. Kopetz and P.-E. Moreau, *Anti-Pattern Matching*, in: *Proceedings of the 16th European Symposium on Programming*, Lecture Notes in Computer Science **4421** (2007), pp. 110–124.

[12] Korp, M., C. Sternagel, H. Zankl and A. Middeldorp, *Tyrolean Termination Tool 2 (TTT2)*, downloadable from http://colo6-c703.uibk.ac.at/ttt2.

[13] Lassez, J.-L. and K. Marriot, *Explicit Representation of Terms Defined by Counter Examples*, Journal of Automated Reasoning **3** (1987), pp. 301–317.

[14] Marché, C. and H. Zantema, *The Termination Competition*, in: *Proceedings of the 18th Conference on Rewriting Techniques and Applications*, Lecture Notes in Computer Science **4533** (2007), pp. 303–313, see also http://www.lri.fr/~marche/termination-competition.

[15] Panitz, S. E. and M. Schmidt-Schauss, *TEA: Automatically proving termination of programs in a non-strict higher order functional language*, in: *Proceedings of the 4th International Symposium on Static Analysis*, Lecture Notes in Computer Science **1302**, 1997, pp. 345–360.

[16] Terese, “Term Rewriting Systems,” Cambridge University Press, Cambridge, UK, 2003.

Computing with Diagrams in Classical Logic

Pierre Lescanne and Dragiša Žunić

Université de Lyon, ENS de Lyon, CNRS (LIP), 46 allée d'Italie, 69364 Lyon, France

1 Introduction

For a long time it was considered not possible to give constructive semantics to classical logic, but only to intuitionistic and linear logic. Recent works have shown [12,13] that this is actually possible if one gives up on the principle that the computational semantics is a confluent rewrite system.

In this paper we present two non-confluent higher order rewrite systems, the first called the ${}^{*}\mathcal{X}$ calculus defined on terms, and the second called the ${}^{d}\mathcal{X}$ calculus defined on diagrams. ${}^{*}\mathcal{X}$ represents a computational interpretation of Gentzen's sequent system G1 for classical logic, while ${}^{d}\mathcal{X}$ whose diagrams are naturally derived from ${}^{*}\mathcal{X}$-terms, captures the essence of classical computation. In both calculi reduction rules satisfy interface preservation and type preservation.

The first computational interpretation for classical logic in the sequent calculus was presented by Herbelin [1,6], while a more direct correspondence with a standard sequent formulation of classical logic was presented by Urban [12]. This research first lead to the $\mathcal{X}$ calculus [9,14], to which the the ideas of making weakening and contraction explicit are applied (as studied through λlxr calculus by Kesner and Lengrand [7]), yielding the ${}^{*}\mathcal{X}$ calculus. Due to this explicitness and the linearity constraints, the terms (and therefore the proofs) can be seen as diagrams, a fact which is nicely illustrated by the diagram for Peirce's law near the end of this paper.

2 The sequent calculus $G1$

Among the three systems presented by Kleene [8], $G1$ is the closest to the original formulation of Gentzen [3]. Despite the fact that Gentzen and Kleene present explicitly exchange rules, which is not the case here, we keep the name $G1$ (Figure 1). Latin symbols $A, B, ...$ are used to denote formulas and Greek symbols $\Gamma, \Delta, \Gamma', \Delta', ...$ to denote contexts, which are in this framework multisets of formulas. Exchange rules are handled by using the multiset data structure, whereas the other structural rules, namely *weakening* and *contraction* are explicitly given. The axiom rules do

$$\frac{}{A \vdash A}\,(ax) \qquad \frac{\Gamma \vdash A, \Delta \qquad \Gamma', A \vdash \Delta'}{\Gamma, \Gamma' \vdash \Delta, \Delta'}\,(cut)$$

$$\frac{\Gamma \vdash A, \Delta \qquad \Gamma', B \vdash \Delta'}{\Gamma, \Gamma', A \to B \vdash \Delta, \Delta'}\,(L\!\to) \qquad \frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \to B, \Delta}\,(R\!\to)$$

$$\frac{\Gamma \vdash \Delta}{\Gamma, A \vdash \Delta}\,(weak\text{-}L) \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash A, \Delta}\,(weak\text{-}R)$$

$$\frac{\Gamma, A, A \vdash \Delta}{\Gamma, A \vdash \Delta}\,(cont\text{-}L) \qquad \frac{\Gamma \vdash A, A, \Delta}{\Gamma \vdash A, \Delta}\,(cont\text{-}R)$$

Fig. 1. Sequent system $G1$

not involve arbitrary contexts. Inference rules with two premises, namely $(L \to)$ and *(cut)*, are given in the context-splitting style, which means that when looking bottom-up the contexts of a conclusion is split by premises. It has been shown in [11] that if a context-sharing style was applied one obtains an equivalent system, i. e., a system that proves the same sequents.

3 The calculus $^*\mathcal{X}$

Terms are built from *names* which should not be confused with *variables*. The difference lies in the fact that a variable can be replaced by an arbitrary term, while a name can be only *renamed* (that is, substituted by another name). The reader will notice the presence of hats on some names, which is a notation borrowed from *Principia Mathematica* [16] and denotes the binding of a name. The syntax of $^*\mathcal{X}$ is presented in Figure 2, where $x, y, z...$ range over an infinite set of innames and $\alpha, \beta, \gamma...$ range over an infinite set of outnames.

$P, Q ::=$	$\langle x.\alpha \rangle$	*capsule*
$\mid$	$\widehat{x}\, P\, \widehat{\beta} \cdot \alpha$	*exporter*
$\mid$	$P\, \widehat{\alpha}\ [x]\ \widehat{y}\, Q$	*importer*
$\mid$	$P\widehat{\alpha} \dagger \widehat{x}Q$	*cut*
$\mid$	$x \odot P$	*left-eraser*
$\mid$	$P \odot \alpha$	*right-eraser*
$\mid$	$x{<}^{\widehat{x_1}}_{\widehat{x_2}}\langle P]$	*left-duplicator*
$\mid$	$[P\rangle^{\widehat{\alpha_1}}_{\widehat{\alpha_2}}{>}\alpha$	*right-duplicator*

Fig. 2. The syntax of $^*\mathcal{X}$

Free and bound names Names can be free or bound. The sets of *free innames* and *free outnames* are defined in Figure 3.

S	$I(S)$	$O(S)$
$\langle x.\alpha\rangle$	x	α
$\widehat{x}\, P\, \widehat{\beta} \cdot \alpha$	$I(P)\backslash\{x\}$	$(O(P)\backslash\{\beta\})\cup\{\alpha\}$
$P\,\widehat{\alpha}\ [x]\ \widehat{y}\, Q$	$I(P)\cup(I(Q)\backslash\{y\})\cup\{x\}$	$(O(P)\backslash\{\alpha\})\cup O(Q)$
$P\widehat{\alpha}\ \dagger\ \widehat{x}Q$	$I(P)\cup(I(Q)\backslash\{x\})$	$(O(P)\backslash\{\alpha\})\cup O(Q)$
$x \odot P$	$I(P)\cup\{x\}$	$O(P)$
$P \odot \alpha$	$I(P)$	$O(P)\cup\{\alpha\}$
$x{<}^{\widehat{x_1}}_{\widehat{x_2}}\langle P]$	$(I(P)\backslash\{x_1,\ x_2\})\cup\{x\}$	$O(P)$
$[P\rangle^{\widehat{\alpha_1}}_{\widehat{\alpha_2}}{>}\alpha$	$I(P)$	$(O(P)\backslash\{\alpha_1,\ \alpha_2\})\cup\{\alpha\}$

Fig. 3. Free names

The notation $I(P)$, $O(P)$ and $N(P)$ denotes a *set* of *free innames*, *free outnames* and *free names* of a term P, respectively. Thus $N(P) = I(P)\cup O(P)$. If we wish to see those sets as *lists*, we use the following notation: $\mathcal{I}^P$, $\mathcal{O}^P$ and $\mathcal{N}^P$, respectively.

A name which occurs in P and which is not free is called a *bound name*. Notice that the same construction often binds two names. This can be either two innames, two outnames or an inname and an outname. Moreover, these names sometimes belong to different subterms, as in the case of an importer or a cut. To denote the binding of all names in one list, we use simply $\widehat{\mathcal{I}}$, $\widehat{\mathcal{O}}$.

It is sometimes needed to use $\overline{I}(P)$ instead of $\mathcal{I}^P$, and similarly for outnames $\overline{O}(P)$, and names in general $\overline{N}(P)$. The bar is used to denote that we see the given set of names as a list, according to the total order which can be defined for the set of names. To exclude a name from a list, for instance, outname α, we write $\mathcal{O}^P\backslash\alpha$.

Renaming We define the operation $P\{x/y\}$ which denotes the *renaming* of a free name y in P by a fresh name x. It is a meta-operation which replaces a unique occurrence of a free name by another free name. Therefore it is simpler than the meta-substitution of λ-calculus, which denotes the substitution of a free variable (which can occur arbitrary number of times) by an arbitrary term.

Indexing We introduce a special kind of renaming, called *indexing*, in order to simplify the syntax of the reduction rules. For example $P_i = ind(P, N(P), i)$ means that P_i is obtained by indexing free names in P by index i, where $i \in N$. Simple notation P_i for cases such as this one will be used when possible. We assume that indexing always creates fresh names. As we use it indexing preserves linearity.

Convention on names We adopt a convention on names: "a name is never both

bound and free in the same term". Terms are defined up to α-conversion, that is, the renaming of bound names does not change them.

Modules A module is a part of a term (not a subterm) of the form $\widehat{\alpha} \not\nearrow \widehat{x}Q$ (*left-module*) and $P\widehat{\beta} \not\nwarrow \widehat{y}$ (*right-module*) which percolates through the structure of that term (and its subterms) during the computation, as specified by the so-called "propagation rules". It resembles the explicit substitution. We say that α and y are the *handles* of $\widehat{\alpha} \not\nearrow \widehat{x}Q$ and $P\widehat{\beta} \not\nwarrow \widehat{y}$, respectively. Two *modules are independent* if the handle of one module does not bind a free name inside the other module, and vice-versa, as follows:

independent modules	conditions
$\widehat{\alpha} \not\nearrow \widehat{x}Q,\ \widehat{\beta} \not\nearrow \widehat{y}R$	$\alpha \notin N(R),\ \beta \notin N(Q)$
$P\widehat{\alpha} \not\nwarrow \widehat{x},\ \ Q\widehat{\beta} \not\nwarrow \widehat{y}$	$x \notin N(Q),\ y \notin N(P)$
$P\widehat{\alpha} \not\nwarrow \widehat{x},\ \ \widehat{\beta} \not\nearrow \widehat{y}R$	$x \notin N(R),\ \beta \notin N(P)$

Linearity In ${}^*\mathcal{X}$, we consider only the *linear* terms, which means:
– Every name has at most one free occurrence, and
– Every binder does bind an actual occurrence of a name (and thus only one)

Although the ${}^*\mathcal{X}$-syntax produces non-linear terms, every non-linear term can be translated into a linear one, using duplicators and erasers. For instance, $\langle x.\alpha\rangle \odot \alpha$, which has two free occurrences of α, can be represented in ${}^*\mathcal{X}$ by the term $[\langle x.\alpha_1\rangle \odot \alpha_2\rangle\frac{\widehat{\alpha_1}}{\widehat{\alpha_2}}{>}\alpha$ (notice the role of a duplicator). The term $\widehat{x}\,\langle x.\alpha\rangle\,\widehat{\beta}\cdot\gamma$ binds no free name and corresponds to the linear term $\widehat{x}\,(\langle x.\alpha\rangle \odot \beta)\,\widehat{\beta}\cdot\gamma$ (notice the role of an eraser). The following tables define specific names which are called *principal*.

a term	princip. names
$\langle x.\alpha\rangle$	x, α
$\widehat{x}\,P\,\widehat{\beta}\cdot\alpha$	α
$P\,\widehat{\alpha}\,[x]\,\widehat{y}\,Q$	x
$P\widehat{\alpha}\ \dagger\ \widehat{x}Q$	none

a term	princip. names
$x \odot P$	x
$P \odot \alpha$	α
$x{<}\frac{\widehat{x_1}}{\widehat{x_2}}\langle P]$	x
$[P\rangle\frac{\widehat{\alpha_1}}{\widehat{\alpha_2}}{>}\alpha$	α

We say that a name is *logical* if it is introduced by either: a capsule, an importer or an exporter (terms that correspond to logical inference rules). We say that a name is *structural* if it is introduced by either an eraser or a duplicator (terms that correspond to structural inference rules). We say that a name is L-principal (S-principal) in a term P if it is both logical (structural) and principal in P.

Lemma 3.1 *Every term has at least a free logical outname.*

Some abbreviations

instead of	we write
$x_1{\odot}(...(x_n \odot P)...)$	$x_1 \odot\ ...x_n \odot P$
$(...(P \odot \alpha_1)\ ...){\odot}\alpha_n$	$P \odot \alpha_1\ ... \odot \alpha_n$

instead of	we write
$x_1{<}\frac{\widehat{y_1}}{\widehat{z_1}}\langle ...x_n{<}\frac{\widehat{y_n}}{\widehat{z_n}}\langle P]...]$	$(x_1, ...,\ x_n){<}\frac{(\widehat{y_1},...,\widehat{y_n})}{(\widehat{z_1},...,\widehat{z_n})}\langle P]$
$[...[P\rangle\frac{\widehat{\beta_1}}{\widehat{\gamma_1}}{>}\alpha_1...\rangle\frac{\widehat{\beta_n}}{\widehat{\gamma_n}}{>}\alpha_n$	$[P\rangle\frac{(\widehat{\beta_1},...,\widehat{\beta_n})}{(\widehat{\gamma_1},...,\widehat{\gamma_n})}{>}(\alpha_1, ..., \alpha_n)$

4 Reduction rules

Reduction rules are grouped into: *Activation rules (left and right)*, *Structural actions (left and right)*, *Deactivation rules (left and right)*, *Logical actions*, *Propagation rules (left and right)*.

Congruence rules We assume some simple congruence rules which originate from the sequent calculus.

Commuting names in a duplicator

$$\begin{array}{rcl} x{<}^{\widehat{x_1}}_{\widehat{x_2}}\langle P] & \equiv & x{<}^{\widehat{x_2}}_{\widehat{x_1}}\langle P] \\ [P\rangle^{\widehat{\alpha_1}}_{\widehat{\alpha_2}}{>}\alpha & \equiv & [P\rangle^{\widehat{\alpha_2}}_{\widehat{\alpha_1}}{>}\alpha \end{array}$$

Permuting independent duplicators

$$\begin{array}{rcl} x{<}^{\widehat{x_1}}_{\widehat{x_2}}\langle y{<}^{\widehat{y_1}}_{\widehat{y_2}}\langle P]] & \equiv & y{<}^{\widehat{y_1}}_{\widehat{y_2}}\langle x{<}^{\widehat{x_1}}_{\widehat{x_2}}\langle P]] \\ [[P\rangle^{\widehat{\alpha_1}}_{\widehat{\alpha_2}}{>}\alpha\rangle^{\widehat{\beta_1}}_{\widehat{\beta_2}}{>}\beta & \equiv & [[P\rangle^{\widehat{\beta_1}}_{\widehat{\beta_2}}{>}\beta\rangle^{\widehat{\alpha_1}}_{\widehat{\alpha_2}}{>}\alpha \\ [x{<}^{\widehat{x_1}}_{\widehat{x_2}}\langle P]\rangle^{\widehat{\alpha_1}}_{\widehat{\alpha_2}}{>}\alpha & \equiv & x{<}^{\widehat{x_1}}_{\widehat{x_2}}\langle [P\rangle^{\widehat{\alpha_1}}_{\widehat{\alpha_2}}{>}\alpha] \end{array}$$

The conditions in the first rule treating the duplicators are $y \notin \{x_1, x_2\}$ and $x \notin \{y_1, y_2\}$ and in the second $\beta \notin \{\alpha_1, \alpha_2\}$ and $\alpha \notin \{\beta_1, \beta_2\}$. The third rule allows us to drop parenthesis and use a simplified notation

$$x{<}^{\widehat{x_1}}_{\widehat{x_2}}\langle P\rangle^{\widehat{\alpha_1}}_{\widehat{\alpha_2}}{>}\alpha \qquad \text{and more generally} \qquad \mathcal{I}{<}^{\widehat{\mathcal{I}_1}}_{\widehat{\mathcal{I}_2}}\langle P\rangle^{\widehat{\mathcal{O}_1}}_{\widehat{\mathcal{O}_2}}{>}\mathcal{O}$$

where $\mathcal{I}$ and $\mathcal{O}$ are lists of names. When $\mathcal{I} = ()$, we write $[P\rangle^{\widehat{\mathcal{O}_1}}_{\widehat{\mathcal{O}_2}}{>}\mathcal{O}$. The case $\mathcal{O} = ()$ is not possible as stated by Lemma 3.1.
When the names are triplicated, one can do it in any order:

$$\begin{array}{rcl} z{<}^{\widehat{y}}_{\widehat{x_3}}\langle y{<}^{\widehat{x_1}}_{\widehat{x_2}}\langle P]] & \equiv & z{<}^{\widehat{x_1}}_{\widehat{y}}\langle y{<}^{\widehat{x_2}}_{\widehat{x_3}}\langle P]] \\ [[P\rangle^{\widehat{\alpha_1}}_{\widehat{\alpha_2}}{>}\beta\rangle^{\widehat{\beta}}_{\widehat{\alpha_3}}{>}\gamma & \equiv & [[P\rangle^{\widehat{\alpha_2}}_{\widehat{\alpha_3}}{>}\beta\rangle^{\widehat{\alpha_1}}_{\widehat{\beta}}{>}\gamma \end{array}$$

This can be seen as an associativity of names bound by a ternary duplicator.
Permuting the erasers: The following rule suggests that we may drop parenthesis and write $x \odot P \odot \alpha$, and more generally we may write: $\mathcal{I} \odot P \odot \mathcal{O}$.

$$\begin{array}{rcl} y \odot x \odot P & \equiv & x \odot y \odot P \\ P \odot \alpha \odot \beta & \equiv & P \odot \beta \odot \alpha \\ (x \odot P) \odot \alpha & \equiv & x \odot (P \odot \alpha) \end{array}$$

Activation rules Activation rules handle the intrinsic non-determinism of the classical cut-elimination. More precisely, during cut-elimination we have to choose in which of the left or the right subtree to push the cut further. For that, the syntax is extended with two operators called *active cuts*:

$$P\widehat{\alpha} \nearrow \widehat{x}Q \quad | \quad P\widehat{\alpha} \nwarrow \widehat{x}Q$$

A cut can be activated only towards an eraser or a duplicator. This means it can be left-activated only if one has either an eraser or a duplicator on the left, and

similarly, it can be right-activated only if one has an eraser or a duplicator on the right. See Figure 4. In other words, to activate a cut towards terms corresponding to logical inference rules is not allowed.

Left :

$(act\text{-}L) \;:\; P\widehat{\alpha} \dagger \widehat{x}Q \;\to\; P\widehat{\alpha} \nearrow \widehat{x}Q$, if α not L-principal for P

Right :

$(act\text{-}R) \;:\; P\widehat{\alpha} \dagger \widehat{x}Q \;\to\; P\widehat{\alpha} \nwarrow \widehat{x}Q$, if x not L-principal for Q

Fig. 4. Activation rules

Terms $P\widehat{\alpha} \nearrow \widehat{x}Q$ and $P\widehat{\alpha} \nwarrow \widehat{x}Q$ are essentially different. This becomes obvious if we take the example when both, α and x, are introduced by erasers. If $P = M \odot \alpha$ and $Q = x \odot N$, where M and N are arbitrary terms

$$(M \odot \alpha)\widehat{\alpha} \nearrow \widehat{x}(x \odot N) \;\to\; (\mathcal{I}^N \setminus x) \odot M \odot \mathcal{O}^N$$
$$(M \odot \alpha)\widehat{\alpha} \nwarrow \widehat{x}(x \odot N) \;\to\; \mathcal{I}^M \odot N \odot (\mathcal{O}^M \setminus \alpha)$$

is Lafont's example [5], which illustrates the non-determinism of classical cut-elimination, here coded in $^*\mathcal{X}$.

Structural actions Structural actions consist of four reduction rules, specifying *erasure* and *duplication* by referring to the situation when an active cut faces an eraser or a duplicator. Structural actions are given in Figure 5 and related to those in intuitionistic logic [2,7].

Deactivation rules See Figure 6.

There is a *duality* between activation and *deactivation* groups of rules. One can be obtained from the other by reversing the arrow, and negating the side conditions. Activation and deactivation rules are designed in a way that does not allow looping, that is, the side conditions do not allow activation followed by deactivation of a cut, or vice versa.

Logical actions The purpose of logical actions is to define reductions when L-principal names are involved in a cut. See Figure 7.

First two logical rules define *merging* of a capsule with another term by using the renaming operation. The merging of a capsule with another element follows Urban's local cut-elimination procedure ([12], p. 50), whereas a slightly different approach was taken in [14,9].

The third logical action describes the direct interaction between an exporter and an importer, which results in *inserting* the (immediate) subterm of an exporter between the two (immediate) subterms of an importer.

Propagation rules

Propagation rules describe the propagation of a cut through the structure of terms. The propagation of a cut over another inactive cut is enabled, which allows representing elegantly β-reduction (see [15], chap. 6) The rules are divided into "left" and "right" symmetric groups, see Figures 8 and 9.

Left :

$(\not\nearrow\text{-}eras)$: $(P \odot \alpha)\widehat{\alpha} \not\nearrow \widehat{x}Q \quad \rightarrow \quad \mathcal{I}^Q \odot P \odot \mathcal{O}^Q$

$(\not\nearrow\text{-}dupl)$: $([P\rangle\frac{\widehat{\alpha_1}}{\widehat{\alpha_2}}{>}\alpha)\widehat{\alpha} \not\nearrow \widehat{x}Q \quad \rightarrow \quad \mathcal{I}^Q{<}\frac{\widehat{\mathcal{I}_1^Q}}{\widehat{\mathcal{I}_2^Q}} \left\langle (P\widehat{\alpha_1} \not\nearrow \widehat{x_1}Q_1)\widehat{\alpha_2} \not\nearrow \widehat{x_2}Q_2 \right\rangle \frac{\widehat{\mathcal{O}_1^Q}}{\widehat{\mathcal{O}_2^Q}}{>}\mathcal{O}^Q$

where:
$\mathcal{I}^Q = \overline{I}(Q) \setminus x,\ \mathcal{O}^Q = \overline{O}(Q)$ and $Q_i = ind(Q,\ N(Q),\ i)$ for $i = 1, 2$.

Right :

$(\nwarrow\text{-}eras)$: $P\widehat{\alpha} \nwarrow \widehat{x}(x \odot Q) \quad \rightarrow \quad \mathcal{I}^P \odot Q \odot \mathcal{O}^P$

$(\nwarrow\text{-}dupl)$: $P\widehat{\alpha} \nwarrow \widehat{x}(x{<}\frac{\widehat{x_1}}{\widehat{x_2}}\langle Q]) \quad \rightarrow \quad \mathcal{I}^P{<}\frac{\widehat{\mathcal{I}_1^P}}{\widehat{\mathcal{I}_2^P}} \left\langle P_2\widehat{\alpha_2} \nwarrow \widehat{x_2}(P_1\widehat{\alpha_1} \nwarrow \widehat{x_1}Q) \right\rangle \frac{\widehat{\mathcal{O}_1^P}}{\widehat{\mathcal{O}_2^P}}{>}\mathcal{O}^P$

where:
$\mathcal{I}^P = \overline{I}(P),\ \mathcal{O}^P = \overline{O}(P) \setminus \alpha$ and $P_i = ind(P,\ N(P),\ i)$ for $i = 1, 2$.

Fig. 5. Structural actions

Left :

$(\not\nearrow\text{-}deact)$: $P\widehat{\alpha} \not\nearrow \widehat{x}Q \rightarrow P\widehat{\alpha} \dagger \widehat{x}Q$, if α is L-principal for P

Right :

$(\nwarrow\text{-}deact)$: $P\widehat{\alpha} \nwarrow \widehat{x}Q \rightarrow P\widehat{\alpha} \dagger \widehat{x}Q$, if x is L-principal for Q

Fig. 6. Deactivation rules

$(ren\text{-}L)$: $\langle y.\alpha\rangle\widehat{\alpha} \dagger \widehat{x}Q \quad \rightarrow \quad Q\{y/x\}$

$(ren\text{-}R)$: $P\widehat{\alpha} \dagger \widehat{x}\langle x.\beta\rangle \quad \rightarrow \quad P\{\beta/\alpha\}$

$(ei\text{-}insert)$: $(\widehat{y}\,P\,\widehat{\beta}\cdot\alpha)\widehat{\alpha} \dagger \widehat{x}(Q\,\widehat{\gamma}\ [x]\ \widehat{z}\,R) \quad \rightarrow \quad either \begin{cases} (Q\widehat{\gamma} \dagger \widehat{y}P)\widehat{\beta} \dagger \widehat{z}R \\ Q\widehat{\gamma} \dagger \widehat{y}(P\widehat{\beta} \dagger \widehat{z}R) \end{cases}$

Fig. 7. Logical actions

For instance the rule $(exp\not\nearrow\text{-}prop)$ shows how an active cut (in fact, a module $\widehat{\beta} \not\nearrow \widehat{y}R$) enters from the right-hand side through an exporter, up to its immediate subterm. Propagation through an importer or a cut requires a side condition to decide to which of the two immediate subterms the module will go.

The rules $(cut(c)\not\nearrow\text{-}prop)$ and $(\nwarrow cut(c)\text{-}prop)$ require additional explanations. They handle the case of propagation over a cut with a capsule whose both names are cut-names. If we exclude these rules from the system, we could construct an infinite reduction sequence such as:
$(P\widehat{\alpha} \dagger \widehat{x}\langle x.\beta\rangle)\widehat{\beta} \dagger \widehat{y}R \rightarrow ... \rightarrow P\widehat{\alpha} \dagger \widehat{x}(\langle x.\beta\rangle\widehat{\beta} \dagger \widehat{y}R) \rightarrow ... \rightarrow (P\widehat{\alpha} \dagger \widehat{x}\langle x.\beta\rangle)\widehat{\beta} \dagger \widehat{y}R$.
Besides that, the solution offered is intuitive as we would expect the terms

$$(P\widehat{\alpha} \dagger \widehat{x}\langle x.\beta\rangle)\widehat{\beta} \not\nearrow \widehat{y}R \quad \text{and} \quad P\widehat{\alpha} \nwarrow \widehat{x}(\langle x.\beta\rangle\widehat{\beta} \dagger \widehat{y}R)$$

to reduce to the same term $P\widehat{\alpha} \dagger \widehat{y}R$, in this case.

Structural rules and convergence

Let us have a close look at the structural rules of duplication. For example the

$(exp\not\nearrow\text{-}prop)$	: $(\widehat{x}\, P\, \widehat{\gamma} \cdot \alpha)\widehat{\beta} \not\nearrow \widehat{y}R$	$\rightarrow\ \widehat{x}\,(P\widehat{\beta} \not\nearrow \widehat{y}R)\,\widehat{\gamma} \cdot \alpha,$	$\alpha \neq \beta$
$(imp\not\nearrow\text{-}prop_1)$	: $(P\,\widehat{\alpha}\,[x]\,\widehat{z}Q)\widehat{\beta} \not\nearrow \widehat{y}R$	$\rightarrow\ (P\widehat{\beta} \not\nearrow \widehat{y}R)\,\widehat{\alpha}\,[x]\,\widehat{z}Q,$	$\beta \in O(P)$
$(imp\not\nearrow\text{-}prop_2)$	: $(P\,\widehat{\alpha}\,[x]\,\widehat{z}Q)\widehat{\beta} \not\nearrow \widehat{y}R$	$\rightarrow\ P\,\widehat{\alpha}\,[x]\,\widehat{z}(Q\widehat{\beta} \not\nearrow \widehat{y}R),$	$\beta \in O(Q)$
$(cut(c)\not\nearrow\text{-}prop)$	: $(P\widehat{\alpha} \dagger \widehat{x}\langle x.\beta\rangle)\widehat{\beta} \not\nearrow \widehat{y}R$	$\rightarrow\ P\widehat{\alpha} \dagger \widehat{y}R$	
$(cut\not\nearrow\text{-}prop_1)$	: $(P\widehat{\alpha} \dagger \widehat{x}Q)\widehat{\beta} \not\nearrow \widehat{y}R$	$\rightarrow\ (P\widehat{\beta} \not\nearrow \widehat{y}R)\widehat{\alpha} \dagger \widehat{x}Q,$	$\beta \in O(P),\ Q \neq \langle x.\beta\rangle$
$(cut\not\nearrow\text{-}prop_2)$	: $(P\widehat{\alpha} \dagger \widehat{x}Q)\widehat{\beta} \not\nearrow \widehat{y}R$	$\rightarrow\ P\widehat{\alpha} \dagger \widehat{x}(Q\widehat{\beta} \not\nearrow \widehat{y}R),$	$\beta \in O(Q),\ Q \neq \langle x.\beta\rangle$
$(L\text{-}eras\not\nearrow\text{-}prop)$	: $(x \odot P)\widehat{\beta} \not\nearrow \widehat{y}R$	$\rightarrow\ x \odot (P\widehat{\beta} \not\nearrow \widehat{y}R)$	
$(R\text{-}eras\not\nearrow\text{-}prop)$	: $(P \odot \alpha)\widehat{\beta} \not\nearrow \widehat{y}R$	$\rightarrow\ (P\widehat{\beta} \not\nearrow \widehat{y}R) \odot \alpha,$	$\alpha \neq \beta$
$(L\text{-}dupl\not\nearrow\text{-}prop)$	: $(x{<}^{\widehat{x_1}}_{\widehat{x_2}}\langle P])\widehat{\beta} \not\nearrow \widehat{y}R$	$\rightarrow\ x{<}^{\widehat{x_1}}_{\widehat{x_2}}\langle P\widehat{\beta} \not\nearrow \widehat{y}R]$	
$(R\text{-}dupl\not\nearrow\text{-}prop)$	: $([P\rangle^{\widehat{\alpha_1}}_{\widehat{\alpha_2}}{>}\alpha)\widehat{\beta} \not\nearrow \widehat{y}R$	$\rightarrow\ [P\widehat{\beta} \not\nearrow \widehat{y}R\rangle^{\widehat{\alpha_1}}_{\widehat{\alpha_2}}{>}\alpha,$	$\alpha \neq \beta$

Fig. 8. Left propagation

$(\not\nwarrow exp\text{-}prop)$	: $P\widehat{\alpha} \not\nwarrow \widehat{x}(\widehat{y}\,Q\,\widehat{\beta} \cdot \gamma)$	$\rightarrow\ \widehat{y}\,(P\widehat{\alpha} \not\nwarrow \widehat{x}Q)\,\widehat{\beta} \cdot \gamma$	
$(\not\nwarrow imp\text{-}prop_1)$	: $P\widehat{\alpha} \not\nwarrow \widehat{x}(Q\,\widehat{\beta}\,[y]\,\widehat{z}\,R)$	$\rightarrow\ (P\widehat{\alpha} \not\nwarrow \widehat{x}Q)\,\widehat{\beta}\,[y]\,\widehat{z}\,R,$	$x \in I(Q)$
$(\not\nwarrow imp\text{-}prop_2)$	: $P\widehat{\alpha} \not\nwarrow \widehat{x}(Q\,\widehat{\beta}\,[y]\,\widehat{z}\,R)$	$\rightarrow\ Q\,\widehat{\beta}\,[y]\,\widehat{z}\,(P\widehat{\alpha} \not\nwarrow \widehat{x}R),$	$x \in I(R)$
$(\not\nwarrow cut(c)\text{-}prop)$	: $P\widehat{\alpha} \not\nwarrow \widehat{x}(\langle x.\beta\rangle\widehat{\beta} \dagger \widehat{y}R)$	$\rightarrow\ P\widehat{\alpha} \dagger \widehat{y}R$	
$(\not\nwarrow cut\text{-}prop_1)$	: $P\widehat{\alpha} \not\nwarrow \widehat{x}(Q\widehat{\beta} \dagger \widehat{y}R)$	$\rightarrow\ (P\widehat{\alpha} \not\nwarrow \widehat{x}Q)\widehat{\beta} \dagger \widehat{y}R,$	$x \in I(Q),\ Q \neq \langle x.\beta\rangle$
$(\not\nwarrow cut\text{-}prop_2)$	: $P\widehat{\alpha} \not\nwarrow \widehat{x}(Q\widehat{\beta} \dagger \widehat{y}R)$	$\rightarrow\ Q\widehat{\beta} \dagger \widehat{y}(P\widehat{\alpha} \not\nwarrow \widehat{x}R),$	$x \in I(R),\ Q \neq \langle x.\beta\rangle$
$(\not\nwarrow L\text{-}eras\text{-}prop)$	: $P\widehat{\alpha} \not\nwarrow \widehat{x}(y \odot Q)$	$\rightarrow\ y \odot (P\widehat{\alpha} \not\nwarrow \widehat{x}Q),$	$x \neq y$
$(\not\nwarrow R\text{-}eras\text{-}prop)$	: $P\widehat{\alpha} \not\nwarrow \widehat{x}(Q \odot \beta)$	$\rightarrow\ (P\widehat{\alpha} \not\nwarrow \widehat{x}Q) \odot \beta$	
$(\not\nwarrow L\text{-}dupl\text{-}prop)$	: $P\widehat{\alpha} \not\nwarrow \widehat{x}(y{<}^{\widehat{y_1}}_{\widehat{y_2}}\langle Q])$	$\rightarrow\ y{<}^{\widehat{y_1}}_{\widehat{y_2}}\langle P\widehat{\alpha} \not\nwarrow \widehat{x}Q],$	$x \neq y$
$(\not\nwarrow R\text{-}dupl\text{-}prop)$	: $P\widehat{\alpha} \not\nwarrow \widehat{x}([Q\rangle^{\widehat{\beta_1}}_{\widehat{\beta_2}}{>}\beta)$	$\rightarrow\ [P\widehat{\alpha} \not\nwarrow \widehat{x}Q\rangle^{\widehat{\beta_1}}_{\widehat{\beta_2}}{>}\beta$	

Fig. 9. Right propagation

rule ($\not\nearrow$-*dupl*):

$$([P\rangle^{\widehat{\alpha_1}}_{\widehat{\alpha_2}}{>}\alpha)\widehat{\alpha} \not\nearrow \widehat{x}Q \quad \rightarrow \quad \mathcal{I}^Q{<}^{\widehat{\mathcal{I}^Q_1}}_{\widehat{\mathcal{I}^Q_2}} \left\langle (P\widehat{\alpha_1} \not\nearrow \widehat{x_1}Q_1)\widehat{\alpha_2} \not\nearrow \widehat{x_2}Q_2 \right\rangle^{\widehat{\mathcal{O}^Q_1}}_{\widehat{\mathcal{O}^Q_2}}{>}\mathcal{O}^Q$$

yields the term $(P\widehat{\alpha_1} \not\nearrow \widehat{x_1}Q_1)\widehat{\alpha_2} \not\nearrow \widehat{x_2}Q_2$ in the context of a certain number of left and right contractions. Notice that this term contains two left modules, namely $\widehat{\alpha_1} \not\nearrow \widehat{x_1}Q_1$ and $\widehat{\alpha_2} \not\nearrow \widehat{x_2}Q_2$, which are independent since $\alpha_1, \alpha_2 \in P$ (and therefore $\alpha_2 \notin Q_1,\ \alpha_1 \notin Q_2$). One may wonder whether the order of modules is relevant, that is, whether the two terms $S_1 = (P\widehat{\alpha_1} \not\nearrow \widehat{x_1}Q_1)\widehat{\alpha_2} \not\nearrow \widehat{x_2}Q_2$ and $S_2 = (P\widehat{\alpha_2} \not\nearrow \widehat{x_2}Q_2)\widehat{\alpha_1} \not\nearrow \widehat{x_1}Q_1$ should be considered the same. Intuitively, in a given situation, there is no reason to prefer using one term over another.

Definition 4.1 (NF-Joinability) *We say that the two terms P and Q are* NF-joinable *if they share the same set of normal forms, denoted $P \downarrow_{NF} Q$.*

Theorem 4.2 (Convergence) *Let $P,\ Q,\ P_1, ...P_n,\ Q_1, ...Q_n$ be arbitrary terms*

and $k \mapsto \bar{k}$ *an arbitrary permutation of* $(1, ...n)$*, then*
(a) $(...(P\widehat{\alpha_1} \nearrow \widehat{x_1} Q_1)...)\widehat{\alpha_n} \nearrow \widehat{x_n} Q_n \downarrow_{NF} (...(P\widehat{\alpha_{\bar{1}}} \nearrow \widehat{x_{\bar{1}}} Q_{\bar{1}})...)\widehat{\alpha_{\bar{n}}} \nearrow \widehat{x_{\bar{n}}} Q_{\bar{n}}$
where $\{\widehat{\alpha_1} \nearrow \widehat{x_1} Q_1, ...\widehat{\alpha_n} \nearrow \widehat{x_n} Q_n\}$ *is a set of independent modules.*

(b) $P_1\widehat{\alpha_1} \nwarrow \widehat{x_1}(...(P_n\widehat{\alpha_n} \nwarrow \widehat{x_n} Q)...) \downarrow_{NF} P_{\bar{1}}\widehat{\alpha_{\bar{1}}} \nwarrow \widehat{x_{\bar{1}}}(...(P_{\bar{n}}\widehat{\alpha_{\bar{n}}} \nwarrow \widehat{x_{\bar{n}}} Q)...)$
where $\{P_1\widehat{\alpha_1} \nwarrow \widehat{x_1}, ...P_n\widehat{\alpha_n} \nwarrow \widehat{x_n}\}$ *is a set of independent modules.*

Theorem 4.3 (Basic properties of $\rightarrow$)

(i) *Preservation of* free names*: If* $P \rightarrow Q$ *then* $N(P) = N(Q)$*.*

(ii) *Preservation of* linearity*: If* P *is linear and* $P \rightarrow Q$ *then* Q *is linear.*

Simplification rules We define the *simplification rules*, denoted $\dashrightarrow$, which can be seen as an efficient way to simplify terms. They are not reduction rules since they do not involve cuts. The point is that applying a duplicator to an eraser is of no interest and can be avoided by using simplification rules, as defined by:

$$
\begin{array}{lcll}
(s_L) & : & x{<}^{\widehat{y}}_{\widehat{z}}\langle z \odot P] & \dashrightarrow \; P\{x/y\} \\
(s_R) & : & [P \odot \gamma\rangle^{\widehat{\beta}}_{\widehat{\gamma}}{>}\alpha & \dashrightarrow \; P\{\alpha/\beta\}
\end{array}
$$

They are applied before the reduction rules, that is, we give them high priority during computation. One can see them as a kind of garbage collection, as they simplify computation by preventing the situation when we duplicate a term to erase one or both copies in the next step.

It is easy to see that the simplification rules preserve the set of free names, linearity and types. The rules can be given in a more general way:

$$
\begin{array}{lcll}
(s^g_L) & : & \mathcal{I}{<}^{\widehat{\mathcal{I}_1}}_{\widehat{\mathcal{I}_2}}\langle \mathcal{I}_2 \odot P] & \dashrightarrow \; P\{\mathcal{I}/\mathcal{I}_1\} \\
(s^g_R) & : & [P \odot \mathcal{O}_2\rangle^{\widehat{\mathcal{O}_1}}_{\widehat{\mathcal{O}_2}}{>}\mathcal{O} & \dashrightarrow \; P\{\mathcal{O}/\mathcal{O}_1\}
\end{array}
$$

5 The type assignment system for $^*\mathcal{X}$

We only consider here linear terms to which we will add type information. Given a set T of basic types, a type is given by $A, B ::= T \mid A \rightarrow B$.

The type of an $^*\mathcal{X}$-term is expressed as $P \because \Gamma \vdash \Delta$, which is the *type assignment* of the $^*\mathcal{X}$-term P, where Γ is a context (antecedent) whose domain consists of free innames of P and Δ is a context (succedent) whose domain consists of free outnames of P. Contexts are sets of pairs (name, formula). By forgetting the names one gets sequents of $G1$ where contexts are multisets of formulas. For example, Γ as a set of type declarations for innames could be $x : A,\ y : B$, while Δ as a set of declarations for outnames could be $\alpha : A,\ \beta : A \rightarrow B,\ \gamma : C$. Comma in the expression Γ, Δ stands for the set union.

We will say that a term P is *typable* if there exist contexts Γ and Δ such that $P \because \Gamma \vdash \Delta$ holds in the system of inference rules given by Figure 10. If we remove the term decoration in the type system of Figure 10, we get the classical sequent

$$\dfrac{}{\langle x.\alpha\rangle \because\ x:A\vdash \alpha:A}\ (ax) \qquad \dfrac{P \because\ \Gamma\vdash\alpha:A,\Delta \qquad Q\because\ \Gamma',x:A\vdash\Delta'}{P\widehat{\alpha}\dagger\widehat{x}Q\because\ \Gamma,\Gamma'\vdash\Delta,\Delta'}\ (cut)$$

$$\dfrac{P\because\ \Gamma\vdash\alpha:A,\Delta \qquad Q\because\ \Gamma',y:B\vdash\Delta'}{P\,\widehat{\alpha}\,[x]\,\widehat{y}\,Q\because\ \Gamma,\Gamma',x:A\to B\vdash\Delta,\Delta'}\ (L\!\to) \qquad \dfrac{P\because\ \Gamma,x:A\vdash\alpha:B,\Delta}{\widehat{x}\,P\,\widehat{\alpha}\cdot\beta\because\ \Gamma\vdash\beta:A\to B,\Delta}\ (R\!\to)$$

$$\dfrac{P\because\ \Gamma\vdash\Delta}{x\odot P\because\ \Gamma,x:A\vdash\Delta}\ (cont\text{-}L) \qquad \dfrac{P\because\ \Gamma\vdash\Delta}{P\odot\alpha\because\ \Gamma\vdash\alpha:A,\Delta}\ (cont\text{-}R)$$

$$\dfrac{P\because\ \Gamma,x:A,y:A\vdash\Delta}{z{<}^{\widehat{x}}_{\widehat{y}}\langle P]\because\ \Gamma,z:A\vdash\Delta}\ (weak\text{-}L) \qquad \dfrac{P\because\ \Gamma\vdash\alpha:A,\beta:A,\Delta}{[P\rangle^{\widehat{\alpha}}_{\widehat{\beta}}{>}\gamma\because\ \Gamma\vdash\gamma:A,\Delta}\ (weak\text{-}R)$$

Fig. 10. The type system for $^*\mathcal{X}$

calculus as presented in Figure 1. As an illustration we give a term for the Peirce's law.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\langle x.\alpha_1\rangle\because\ x:A\vdash\alpha_1:A}\ (ax)}{\langle x.\alpha_1\rangle\odot\beta\because\ x:A\vdash\alpha_1:A,\beta:B}\ (weak\text{-}R)}{\widehat{x}\,(\langle x.\alpha_1\rangle\odot\beta)\,\widehat{\beta}\cdot\gamma\because\ \vdash\alpha_1:A,\gamma:A\to B}\ (\to\! R) \qquad \dfrac{}{\langle y.\alpha_2\rangle\because\ y:A\vdash\alpha_2:A}\ (ax)}{(\widehat{x}\,(\langle x.\alpha_1\rangle\odot\beta)\,\widehat{\beta}\cdot\gamma)\,\widehat{\gamma}\,[z]\,\widehat{y}\,\langle y.\alpha_2\rangle\because\ z:(A\to B)\to A\vdash\alpha_1:A,\alpha_2:A}\ (\to\! L)}{[(\widehat{x}\,(\langle x.\alpha_1\rangle\odot\beta)\,\widehat{\beta}\cdot\gamma)\,\widehat{\gamma}\,[z]\,\widehat{y}\,\langle y.\alpha_2\rangle\rangle^{\widehat{\alpha_1}}_{\widehat{\alpha_2}}{>}\alpha\because\ z:(A\to B)\to A\vdash\alpha:A}\ (cont\text{-}R)}{\widehat{z}\,([(\widehat{x}\,(\langle x.\alpha_1\rangle\odot\beta)\,\widehat{\beta}\cdot\gamma)\,\widehat{\gamma}\,[z]\,\widehat{y}\,\langle y.\alpha_2\rangle\rangle^{\widehat{\alpha_1}}_{\widehat{\alpha_2}}{>}\alpha)\,\widehat{\alpha}\cdot\delta\because\ \vdash\delta:((A\to B)\to A)\to A}\ (\to\! R)$$

We can also assign a prooftree to the term $\lambda xyz.xz(yz)$ known as the S-combinator of λ-calculus. Typing this term is left to the reader (see [15] Sec. 4.4)

$$\widehat{\omega}\,(\widehat{u}\,(\widehat{x}\,(x{<}^{\widehat{x_1}}_{\widehat{x_2}}\langle\underline{\langle x_2.\epsilon\rangle\,\widehat{\epsilon}\,[w]\,\widehat{v}\,\underline{((\underline{\langle x_1.\delta\rangle\,\widehat{\delta}\,[u]\,\widehat{y}\,\langle y.\beta\rangle})\,\widehat{\beta}\,[v]\,\widehat{z}\,\langle z.\gamma\rangle)}}])\,\widehat{\gamma}\cdot\eta)\,\widehat{\eta}\cdot\theta)\,\widehat{\theta}\cdot\alpha$$

Theorem 5.1 (Witness reduction)

1. *If* $S \because\ \Gamma\vdash\Delta$ *and* $S\to S'$, *then* $S'\because\ \Gamma\vdash\Delta$
2. *If* $S \because\ \Gamma\vdash\Delta$ *and* $S\dashrightarrow S'$, *then* $S'\because\ \Gamma\vdash\Delta$

6 Diagrammatic classical computing

After designing and studying the $^*\mathcal{X}$ calculus, a linear model of computation which introduces explicit erasure and duplication, it was natural to think about a diagrammatic representation enabled by linearity and explicitness of name handling. Thus we obtained the diagrammatic calculus $^d\mathcal{X}$, which abstracts away from unessential part of $^*\mathcal{X}$-computation. Having both of these calculi at hand makes it easier to understand each of them, and through them the rather complex classical cut-elimination.

The basic notion of $^d\mathcal{X}$ is *port* and it corresponds to the notion of a *name* in $^*\mathcal{X}$. Ports represent the interface of a diagram. Like for names, there are two kinds of ports - entering ports, called *in-ports*, and exiting, called *out-ports*. We imagine

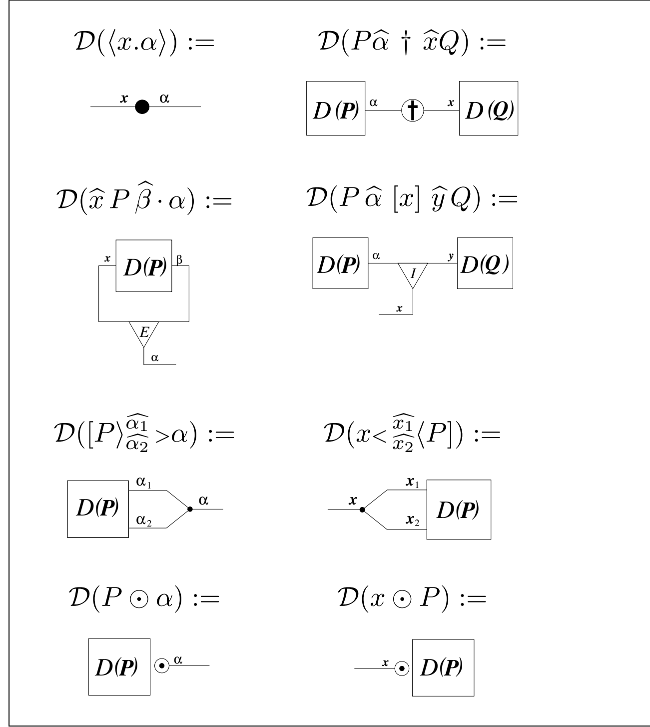


Fig. 11. Encoding the terms into diagrams

a diagram as a circuit with entering and exiting wires (See Figure 12) where P is an actual diagram. By convention in-ports are always coming from the left, and out-ports are always going to the right. Latin letters $x, y, z, \ldots$ are used to denote in-ports and Greek $\alpha, \beta, \gamma, \ldots$ to denote out-ports.

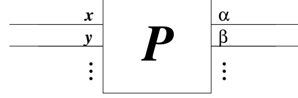


Fig. 12. A generic diagram

6.1 The syntax

The syntax of the diagrammatic calculus ${}^d\mathcal{X}$ defined in Figure 13 consists of eight basic diagram constructors which are directly inspired by ${}^*\mathcal{X}$-terms, and which can be seen as their two-dimensional view. The constructors are called: (1) in-out, (2) cut, (3) $\mathcal{E}$-fan, (4) $\mathcal{I}$-fan, (5)&(6) black-holes, (7)&(8) forks. The translation $\mathcal{D}$ from terms to diagrams is defined inductively on the structure of terms by Figure 11. It is important to notice that the diagrams are always built from within, starting from the ground diagram called *in-out*. Higher order diagrams are built by respecting the rules defined by the syntax.

Because a diagram represents a class of terms, the terminology used to denote diagrams is different from that used for ${}^*\mathcal{X}$ (see Figure 15).

Convention on ports We use the convention that the ports of a diagram are

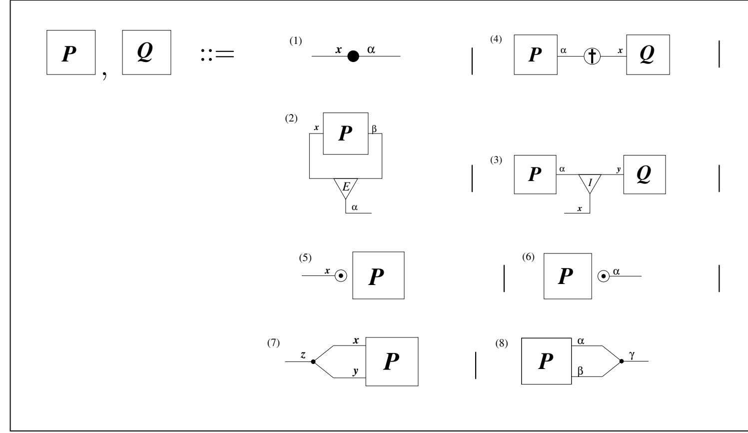


Fig. 13. The diagrammatic syntax

presented only when they are relevant. The set of ports $\mathcal{P}$ of a diagram is presented in Figure 14. A port is said to be *logical* if it is created by an in-out, $\mathcal{I}$-fan, or an $\mathcal{E}$-fan. *Structural ports* are the ones created by black holes and forks.

Diagram	Ports ($\mathcal{P}$)
	$\{x, \alpha\}$
	$\{\alpha\} \cup \mathcal{P}(P)$
	$\{x\} \cup \mathcal{P}(P) \cup \mathcal{P}(Q)$
	$\mathcal{P}(P) \cup \mathcal{P}(Q)$

Diagram	Ports ($\mathcal{P}$)
	$\{x\} \cup \mathcal{P}(P)$
	$\{\alpha\} \cup \mathcal{P}(P)$
	$\{z\} \cup \mathcal{P}(P)$
	$\{\gamma\} \cup \mathcal{P}(P)$

Fig. 14. The ports

Lemma 6.1 *Each diagram has at least one logical out-port.*

6.2 The reduction rules

Reduction rules are divided into four main groups. These are *logical* and *structural* actions and also *activation* and *deactivation* rules. Each group (except logical) can itself be split into two symmetric left and right subgroups. The activation and deactivation groups are dual.

The diagrammatic calculus ${}^{d}\mathcal{X}$ has less rules than ${}^{*}\mathcal{X}$. Namely the group called "propagation rules" does not exist in ${}^{d}\mathcal{X}$. The basic purpose of this group in ${}^{*}\mathcal{X}$ is to propagate the cut through the structure of terms, until it reaches a point where the propagation ends. In general, a propagation ends when a module, for example $\widehat{\alpha} \not\dagger \widehat{x}Q$, reaches an actual place where a free name α occurs in the syntactic

${}^{d}\mathcal{X}$*-diagrams*	${}^{*}\mathcal{X}$*-terms*	*Logic*
in-out	capsule	axiom
$\mathcal{E}$-fan	exporter	→R-intro
$\mathcal{I}$-fan	importer	→L-intro

${}^{d}\mathcal{X}$*-diagrams*	${}^{*}\mathcal{X}$*-terms*	*Logic*
fork	duplicator	contraction
black hole	eraser	weakening
dagger	cut	cut
port	name	named proposition

Fig. 15. The terminology

representation. In ${}^{d}\mathcal{X}$, since we are in a two-dimensional space, the propagation rules are lost which also suggests that they are not an essential part of the computation.

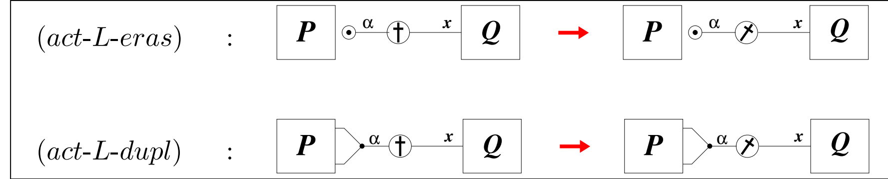
The reduction procedure aims at the elimination of †'s (daggers) and captures the essence of the classical cut-elimination, like proof nets for linear logic [4]. Therefore it is *non-deterministic* and *non-confluent*, like classical cut-elimination.

Following *convention on ports*, we draw only the relevant ports in reduction rules, i.e., those which are involved in reducing, or created by the reduction rule.

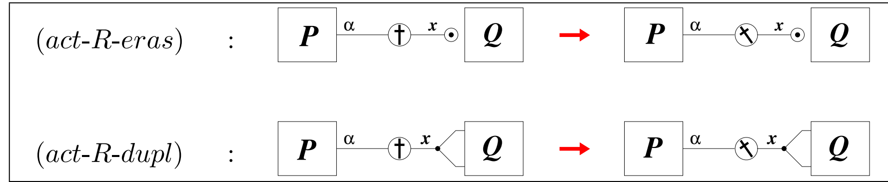
6.2.1 *Activation rules*

As *a picture is worth a thousand words* we present the rules with no comments.

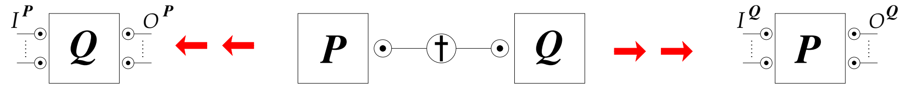
- **Left-activation.**



- **Right-activation.**



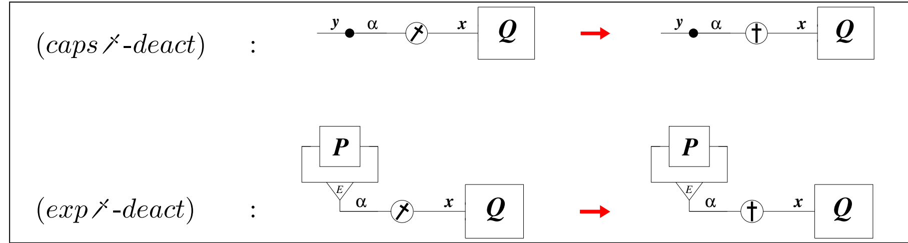
Lafont's example [5] is



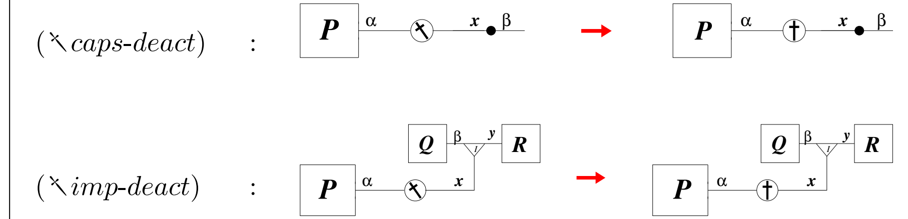
Active cuts are introduced into the system to express non-determinism. The rules make sure that when a dagger is activated, it will be treated by some other reduction rule.

6.2.2 *Deactivation rules*

- **Left-deactivation.**



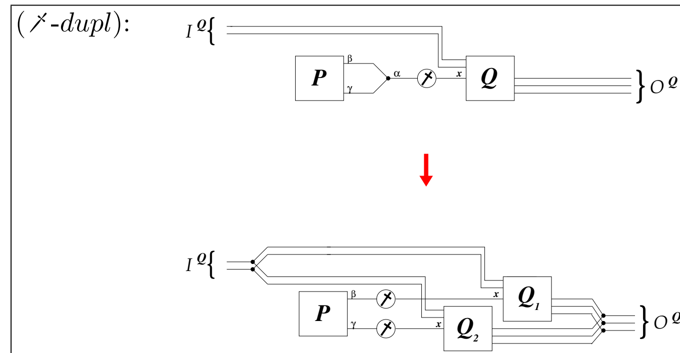
• **Right-deactivation.**

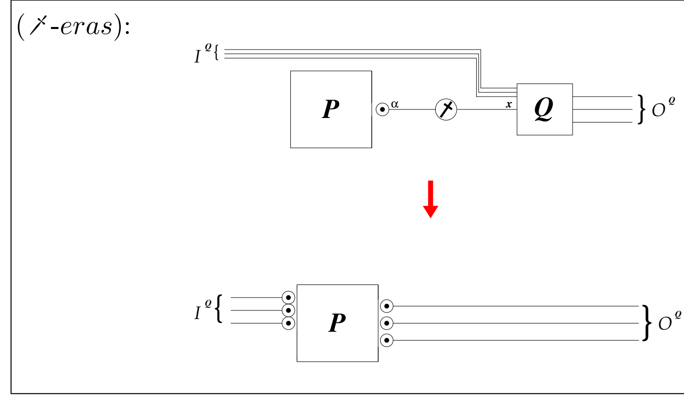


Activation and deactivation are dual. Indeed by negating the side condition and by changing the direction of the arrow in an activation rule, one gets an deactivation rule and vice versa.

6.2.3 Structural actions

Structural actions define *duplication* and *erasure* of diagrams. They are implemented by using *forks* and a *black holes*, which correspond to erasers and duplicators in $^*\mathcal{X}$. We have pairs of symmetric rules: (↗*-erasure)* and (↖*-erasure)*, and (↗*-duplication)* and (↖*-duplication)*. By lack of space, we give only the rules for left actions. Those for right actions are symmetric (for details see [15]).

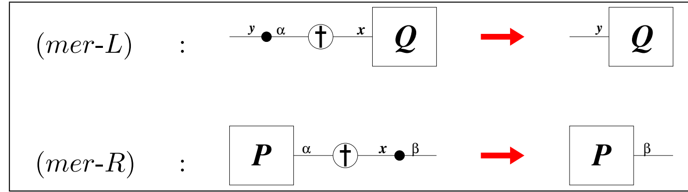




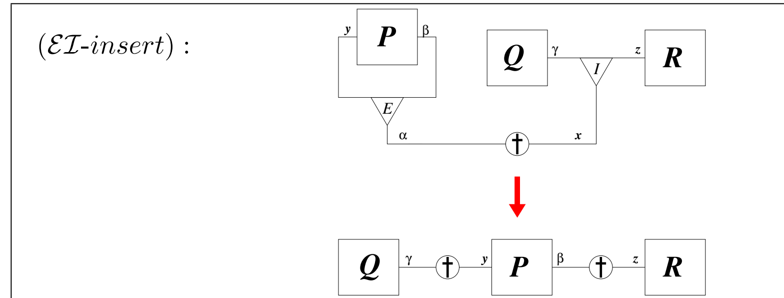
The rule (*†-duplication)* defines an action which occurs when a left-activated dagger faces a right fork. We duplicate a term on the right-hand side of a dagger.

6.2.4 *Logical actions*

- **Merging.**



- **Inserting.** The last rule is called *insertion* and describes how to reduce a diagram that connects an $\mathcal{E}$-fan to an $\mathcal{I}$-fan through a dagger. The term P is inserted between the terms Q and R.



6.3 *Diagram simplification*

Simplification rules (Figure 16) are denoted by $\dashrightarrow$. They are used to simplify diagrams when possible, but they are not reduction rules in the sense that they do not involve cuts. These rules have no computational meaning. Furthermore, rewriting in an arbitrary diagram all branches of the form and in optimizes further computations, in the sense that many useless duplications of a term are avoided. The case of two branches of a fork ending in black holes is a special case. Thus is rewritten in , and similarly is rewritten in . The set of ports is preserved by reductions and simplification

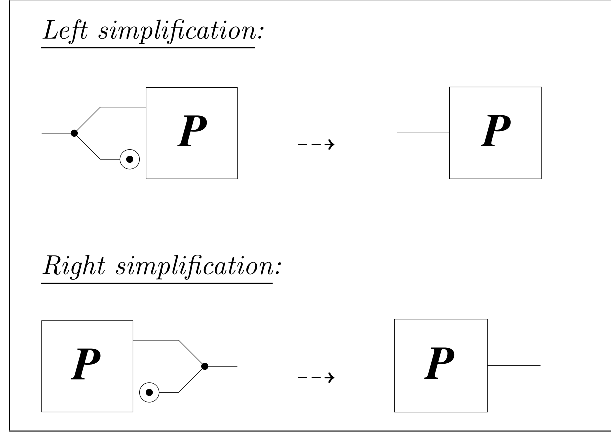


Fig. 16. Diagram simplification

rules.

Theorem 6.2 (Preservation of ports)
1. If $P \rightarrow Q$ then $\mathcal{P}(P) = \mathcal{P}(Q)$
2. If $P \dashrightarrow Q$ then $\mathcal{P}(P) = \mathcal{P}(Q)$

7 The type assignment system for ${}^{d}\mathcal{X}$

Given a set T of basic types, a type is given by $A, B ::= T \mid A \rightarrow B$. The type assignment of a diagram P is given by an expression $P \,{:}\!\cdot\; \Gamma \vdash \Delta$. Here Γ stands for a set of type declarations for in-ports, while Δ stands for a set of declarations for out-ports. If P is a diagram, an expression $P \,{:}\!\cdot\; \Gamma \vdash \Delta$ is used to denote a *type assignment*. The type system is presented in Figure 19.

Theorem 7.1 (Witness reduction)
1. If $D_1 \,{:}\!\cdot\; \Gamma \vdash \Delta$ and $D_1 \rightarrow D_2$ then $D_2 \,{:}\!\cdot\; \Gamma \vdash \Delta$
2. If $D_1 \,{:}\!\cdot\; \Gamma \vdash \Delta$ and $D_1 \dashrightarrow D_2$ then $D_2 \,{:}\!\cdot\; \Gamma \vdash \Delta$

As an illustration we give two diagrams as representations of proofs: the Peirce's law: $\vdash ((A \rightarrow B) \rightarrow A) \rightarrow A$ (Figure 17), and the **S** combinator: $\vdash (A \rightarrow B \rightarrow C) \rightarrow (A \rightarrow B) \rightarrow A \rightarrow C$ (Figure 18).

8 Non determinism

The diagrammatic calculus ${}^{d}\mathcal{X}$ is non-deterministic and non-confluent, which is in accordance with the properties of classical cut-elimination. This is due to the following reasons.

The direction of activation. Activation rules are new. Introduced by Urban and Bierman in [13] as "commuting cuts", they showed that it is not necessary to restrict this choice in order to achieve strong normalization. The non-determinism introduced by activation rules leads to non-confluence.

Actual boundaries of diagrams. Structural actions define erasure and duplication of

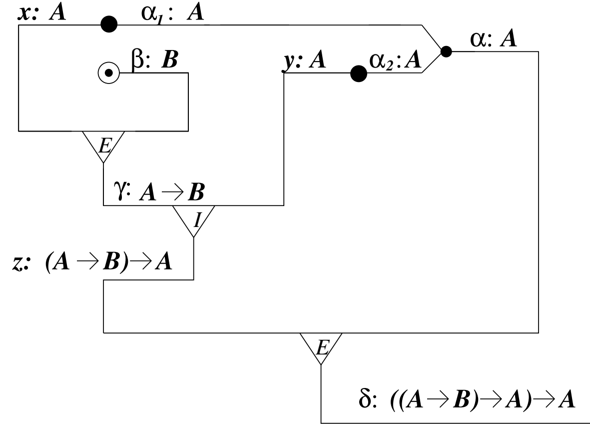


Fig. 17. The diagram for Peirce's law

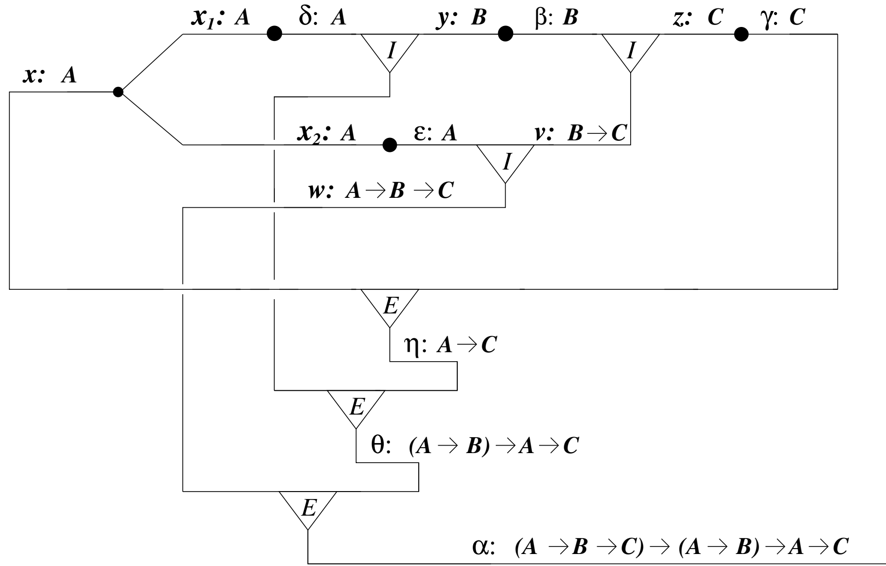


Fig. 18. The diagram for the combinator **S**

diagrams, but the choice of which specific diagram is erased or duplicated is non-deterministic (as in [10] we do not use boxes). The diagram to duplicate (erase) can be any subdiagram whose port is involved in a dagger operation. It is still an open question whether this leads to non-confluence or not.

9 Conclusion

This paper provides an insight into rather complex classical computations. We extract its essential part through diagrammatic representation. ${}^{*}\mathcal{X}$ is the first classical calculus to introduce terms for explicit erasure and duplication. From this low level language we derive a higher abstraction model - the diagrammatic calculus ${}^{d}\mathcal{X}$.

However ${}^{*}\mathcal{X}$-terms are not in one-to-one correspondence with ${}^{d}\mathcal{X}$-diagrams, and this is the starting point of a research aimed at revealing which exactly ${}^{*}\mathcal{X}$-terms

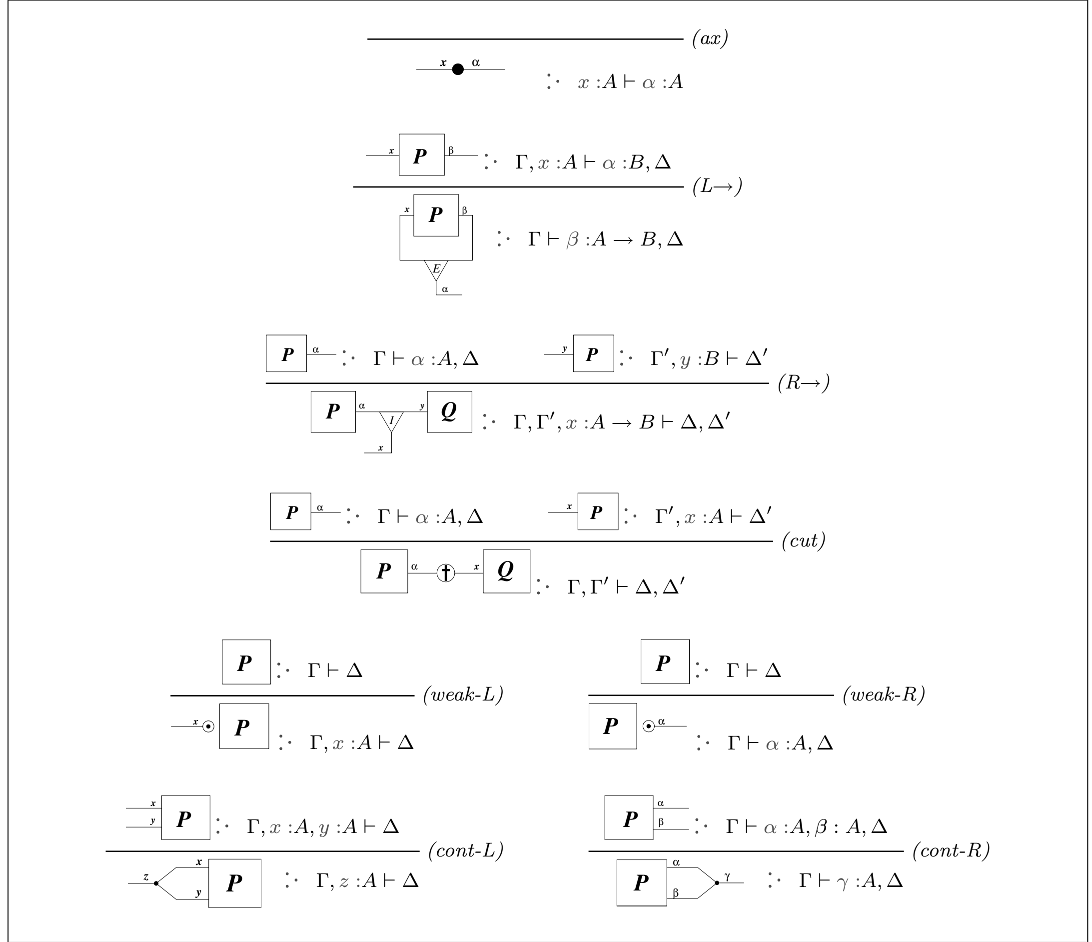


Fig. 19. The type system for ${}^{d}\mathcal{X}$

should be considered the same; this lead to the design of the ${}^{©}\mathcal{X}$ calculus (see [15], part III), with congruence relation on terms, which proposes a more faithful correspondence with the diagrammatic calculus. Namely, one diagram of ${}^{d}\mathcal{X}$ corresponds to a congruence class of ${}^{©}\mathcal{X}$.

References

[1] Curien, P.-L. and H. Herbelin, *The duality of computation*, in: *Proc. 5 th ACM SIGPLAN Int. Conf. on Functional Programming (ICFP'00)*, 2000, pp. 233–243.

[2] David, R. and B. Guillaume, *A lambda-calculus with explicit weakening and explicit substitution*, Mathematical Structures in Computer Science **11** (2001), pp. 169–206.

[3] Gentzen, G., *Untersuchungen über das logische Schließen*, Math. Z. **39** (1935), pp. 176–210, 405–431.

[4] Girard, J.-Y., *Linear logic*, Theoretical Computer Science **50** (1987), pp. 1–102.

[5] Girard, J.-Y., Y. Lafont and P. Taylor, "Proofs and Types," Cambridge Tracts in Theoret Computer Science **7**, Cambridge U. Press, 1989.

[6] Herbelin, H., "C'est maintenant qu'on calcule, au cœur de la dualité," Habilitation à diriger les recherches, Université Paris XI (2007).

[7] Kesner, D. and S. Lengrand, *Ressource operators for lambda-calculus.*, Inform. Comput. **205** (2007), pp. 419–473, long version.

[8] Kleene, S., "Introduction to Metamathematics," Number 1 in Bibliotheca mathematica, North-Holland, 1952, revised edition, Wolters-Noordhoff, 1971.

[9] Lengrand, S., *Call-by-value, call-by-name, and strong normalization for the classical sequent calculus*, , **86**, 2003.

[10] Robinson, E., *Proof nets for classical logic*, Journal of Logic and Computation **13** (2003), pp. 777–797.

[11] Troelstra, A. S. and H. Schwichtenberg, "Basic Proof Theory," Cambridge U. Press, New York, NY, USA, 1996.

[12] Urban, C., "Classical Logic and Computation," Ph.D. thesis, Cambridge U. (2000).

[13] Urban, C. and G. M. Bierman, *Strong normalisation of cut-elimination in classical logic*, in: *Typed Lambda Calculus and Applications*, Lecture Notes in Computer Science **1581** (1999), pp. 365–380.

[14] van Bakel, S., S. Lengrand and P. Lescanne, *The language $\mathcal{X}$: circuits, computations and classical logic*, in: *Proc.9th Italian Conf. on Theoretical Computer Science (ICTCS'05)*, Lecture Notes in Computer Science **3701**, 2005, pp. 81–96.

[15] Žunić, D., "Computing with Sequents and Diagrams in Classical Logic-Calculi $^*\mathcal{X}$, $^d\mathcal{X}$ and $^{©}\mathcal{X}$," Ph.D. thesis, ENS de Lyon (2007).

[16] Whitehead, A. N. and B. Russell, "Principia Mathematica," Cambridge U. Press, 1925, 2nd edition.

Closure of Tree Automata Languages under Innermost Rewriting

Adrià Gascón{1}, Guillem Godoy{1} & Florent Jacquemard{2} [2]

{1} Technical University of Catalonia, Jordi Girona 1, Barcelona, Spain. [1]

{2} INRIA Futurs & LSV, UMR CNRS/ENS Cachan, France.

Abstract

Preservation of regularity by a term rewrite system (TRS) states that the set of reachable terms from a tree automata (TA) language (aka regular term set) is also a TA language. It is an important and useful property, and there have been many works on identifying classes of TRS ensuring it; unfortunately, regularity is not preserved for restricted classes of TRS like shallow TRS. Nevertheless, this property has not been studied for important strategies of rewriting like the innermost strategy – which corresponds to the *call by value* computation of programming languages.

We prove that the set of innermost-reachable terms from a TA language by a shallow TRS is not necessarily regular, but it can be recognized by a tree automaton with equality and disequality constraints between brothers. As a consequence we conclude decidability of regularity of the reachable set of terms from a TA language by innermost rewriting and shallow TRS. This result is in contrast with plain (not necessarily innermost) rewriting for which we prove undecidability. We also show that, like for plain rewriting, innermost rewriting with linear and right-shallow TRS preserves regularity.

Introduction

Finite representations of infinite sets of terms are useful in many areas of computer science. The choice of a formalism for this purpose depends on its expressiveness, but also on its computational properties. Finite-state Tree Automata (TA) [3] are a well studied formalism for representing term languages, due to their good computational and expressiveness properties. They are used in many fields of computer science, from a theoretical and a practical point of view. For instance, for the analysis of systems or programs, when configurations can be represented by trees (e.g. concurrent processes with parallel and sequential composition operators) TA provide a finite representation of possibly infinite sets of configurations.

Term rewriting is a general formalism for the symbolic evaluation of terms by replacement of some patterns by others, following oriented equations, or rewrite

[1] The first two authors were supported by Spanish Min. of Educ. and Science by the FORMALISM project (TIN2007-66523) and by the LOGICTOOLS-2 project (TIN2007-68093-C02-01)

[2] adriagascon@gmail.com, ggodoy@lsi.upc.edu, florent.jacquemard@lsv.ens-cachan.fr

rules, given in a finite set (a term rewrite system, or TRS). Plain rewriting is sometimes too general, and in many contexts rewriting is applied with specific strategies giving a finer representation of the system behaviour. This is the case of the innermost strategy, which corresponds to the *call by value* computation of programming languages, where arguments are fully evaluated before the application of the function.

In the above application to system verification, transitions in infinite state systems can usually be represented by rewrite rules. There have been many studies of the connections between tree automata and rewriting, and a central property in this domain is the *preservation of regularity*. It states that for any given regular language L (which means that L is accepted by a TA), the set of reachable terms from L by a TRS R, denoted $R^*(L)$ is also regular. Preservation of regularity has been widely studied. The first result of this kind was that preservation of regularity holds for every ground TRS, as shown in [16]. In [14] this property was established for linear (variables occur at most once in every left-hand and right-hand side of a rule) and right-flat (the right-hand sides of the rules have height 0 or 1) TRS. There have been several extensions of this result, e.g. [6,10,13,15,5], and [13] represents a breakthrough since the left-linearity condition (linearity of left-hand sides of rules of the TRS) was dropped. However, in all the above cases, the condition of right-linearity remains necessary and in fact, a rewrite rule like $g(x) \to f(x, x)$ does not preserve regularity. Moreover, only plain rewriting is considered in these works, except in [5] where the bottom-up strategy is considered; there have been (up to our knowledge) no studies of regularity preservation under the innermost strategy.

The aim of this work is to study the preservation of regularity for innermost rewriting, and to identify a class of TRS for which better results can be found under the innermost strategy than under plain rewriting. We consider the class of shallow (all variables occur at depth 0 or 1 in the terms of the rules) TRS. Although the shallow case seems restrictive, for plain rewriting, shallow TRS do not preserve regularity. Moreover, several interesting properties of TRS, like reachability, joinability, confluence [12] and termination [8], are undecidable for shallow TRS, while adding certain linearity restrictions allows the decidability of all these problems [13,15,9,8]. Hence, from a theoretical point of view, the shallow case draws a frontier for decidability when one considers classes of TRS defined by syntactic restrictions.

Our main result (Theorem 4.6, Section 4.2) is that, given a regular language L and a *shallow* TRS R, the set $R^*(L)$ of terms reachable from L using R with the innermost strategy is recognized by tree automata extended with equality and disequality constraints between brothers in their state transitions. This kind of automata, which we call BTTA, was introduced in [2] as an extension of TA, and it has also good closure and decidability properties, but with worst complexity than standard TA. This is in contrast with the situation with plain rewriting: $R^*(L)$ is in general neither a TA nor a BTTA language under the same hypotheses (Proposition 3.2, Section 3).

One of the classical techniques for proving results of preservation of regularity consists of adding transitions to the automaton recognizing the starting language L, in order to simulate rule applications of R and recognize also all the terms reachable

from L. Apparently, this completion technique which worked well for standard TA (in all the regularity preservation results cited so far) does not work for general shallow TRS. Innermost rewriting cannot be simulated by TA transitions, despite it does operate almost in a bottom-up fashion for shallow TRS [5]. The reason follows from two other results of the paper:

- First, we show that innermost rewriting with flat TRS (TRS whose all left-hand-side and right-hand-sides of rules have depth at most one) does not preserve regularity (Proposition 4.2, Section 4). As a consequence, we need to consider BTTA instead of standard TA.
- Second, flat and linear TRS do neither preserve BTTA-recognizably (Proposition 4.3, Section 4.1). Consequently, TA completion cannot work in this case.

The main result is obtained in two steps. First, we reduce the problem of representing the reachable terms from a regular set to the reachable terms from a constant. Next, we give a direct construction of a BTTA recognizing the reachable terms from a constant. It is based on a representation of the set of reachable terms introduced in [7] using constrained terms. As an immediate consequence of the main result, we obtain from [1] that given a regular language L and a shallow TRS R, it is decidable whether $R^*(L)$ is regular for innermost rewriting. In contraposition, we prove undecidability of regularity of $R^*(L)$ for plain (not necessarily innermost) rewriting.

Another positive result (Theorem 5.3, Section 5.1) is that, like for plain rewriting, innermost rewriting with linear and right-shallow TRS preserves regular languages. This result has been independently obtained in [11] In our case it is proved with a non trivial adaptation of the tree automata completion technique of e.g. [14,10]. The cases of plain and innermost rewriting are different in essence to treat, and some subtle differences need to be introduced. We show in particular that even though TA completion permits to establish that right-linear and right-flat TRS (*i.e.* when left-hand sides of rules might be not linear) preserve regular languages under plain rewriting, we show that this property is no longer true for under innermost rewriting (Proposition 5.4, Section 5.2).

1 Preliminaries

We use standard notation from the term rewriting literature [4]. A *signature* Σ is a finite set of function symbols with arity. We write Σ_m for the subset of function symbols of Σ of arity m. Given an infinite set $\mathcal{V}$ of variables, the set of terms built over Σ and $\mathcal{V}$ is denoted $\mathcal{T}(\Sigma, \mathcal{V})$, and the subset of ground terms is denoted $\mathcal{T}(\Sigma)$. The set of variables occurring in a term $t \in \mathcal{T}(\Sigma, \mathcal{V})$ is denoted $vars(t)$. A *substitution* σ is a mapping from $\mathcal{V}$ to $\mathcal{T}(\Sigma, \mathcal{V})$. The application of a substitution σ to a term t is written $\sigma(t)$, and is the homomorphic extension of σ to $\mathcal{T}(\Sigma, \mathcal{V})$.

A term t is identified as usual to a function from its set of *positions* (strings of positive integers) $Pos(t)$ to symbols of $\mathcal{F}$ and $\mathcal{V}$. We note Λ the empty string (root position). The length of a position p is denoted $|p|$. The *height* of a term t, denoted $h(t)$, is the maximum of $\{|p| \bigm| p \in Pos(t)\}$. A subterm of t at position p is written $t|_p$, and the replacement in t of the subterm at position p by u denoted $t[u]_p$.

Rewriting and the innermost strategy.

A *term rewriting system* (TRS) over a signature Σ is a finite set of rewrite rules $\ell \to r$, where $\ell \in \mathcal{T}(\Sigma, \mathcal{V}) \setminus \mathcal{V}$ (it is called left-hand side of the rule) and $r \in \mathcal{T}(\Sigma, vars(\ell))$ (it is called right-hand side). A term $s \in \mathcal{T}(\Sigma, \mathcal{V})$ rewrites to t by a TRS R at a position p of s with a substitution σ, denoted $s \xrightarrow[R,p,\sigma]{} t$ (p and σ may be omitted in this notation) if there is a rewrite rule $\ell \to r \in R$ such that $s|_p = \sigma(\ell)$ and $t = s[\sigma(r)]_p$. In this case, s is said to be *reducible*. The set of irreducible terms, also called R-*normal-forms*, is denoted by NF_R. The transitive and reflexive closure of $\xrightarrow[R]{}$ is denoted $\xrightarrow[R]{*}$. Given $L \subseteq \mathcal{T}(\Sigma)$, we note $R^*(L) = \{t \mid \exists s \in L, s \xrightarrow[R]{*} t\}$. The above rewrite step is called *innermost* if all proper subterms of $s|_p$ are R-normal forms. In this case, we write $s \xrightarrow[R]{\imath} t$, and $\xrightarrow[R]{\curlywedge}$ for the the transitive and reflexive closure of this relation, and $R^{\curlywedge}(L)$ for $\{t \mid \exists s \in L, s \xrightarrow[R]{\curlywedge} t\}$. We shall also use the notations $\mathsf{NF}^*_R(s)$ and $\mathsf{NF}^{\curlywedge}_R(s)$ (with $s \in \mathcal{T}(\mathcal{F}, \mathcal{V})$) for resp. $R^*(\{s\}) \cap \mathsf{NF}_R$ and $R^{\curlywedge}(\{s\}) \cap \mathsf{NF}_R$.

A TRS is called *linear* (resp. *right-linear*, *left-linear*) if every variable occurs at most once in each term (resp. right-hand side, left-hand side) of the rules. It is called *shallow* (resp. *right-shallow*, *left-shallow*) if variables occur at depth 0 or 1 in the terms (resp. in the right-hand sides, in the left-hand sides) of the rules and *flat* (resp. *right-flat*, *left-flat*) if the terms (resp. the right-hand sides, the left-hand sides) in the rules have height at most 1. A rule $\ell \to r$ is called *collapsing* if r is a variable.

2 Tree automata with constraints between brothers

A *tree automaton* (TA) A on a signature Σ is a tuple $(Q, Q^{\mathsf{f}}, \Delta)$ where Q is a finite set of nullary state symbols, disjoint from Σ, $Q^{\mathsf{f}} \subseteq Q$ is the subset of final states and Δ is a set of ground rewrite rules of the form: $f(q_1, \ldots, q_m) \to q$, or $q_1 \to q$ (ε-transition) where $f \in \Sigma_m$, and $q_1, \ldots, q_m, q \in Q$ (q is called the *target* state of the rule).

A *Bogaert-Tison tree automaton* (BTTA, or tree automaton with constraints between brothers) is defined like a TA except that its states are unary and its transitions are constrained rewrite rules of the form $f\big(q_1(x_1), \ldots, q_m(x_m)\big) \to q\big(f(x_1, \ldots, x_m)\big)\, [\![c]\!]$, or ε-transitions $q_1(x_1) \to q(x_1)$, where $x_1, \ldots, x_m$ are distinct variables and the constraint c is a Boolean combination of equalities $x_i = x_j$. Equivalently, the constraint c can be defined as a partition P of $\{1, \ldots, m\}$ with the same meaning as a conjunction of equalities $x_i = x_j$ for the indexes i, j such that $i \equiv_P j$, and disequalities $x_i \neq x_j$ for the indexes i, j such that $i \not\equiv_P j$. Following the notations of [2,3], the above transitions are written $f(q_1, \ldots, q_m) \xrightarrow{c} q$ (or $f(q_1, \ldots, q_m) \to q$ when c is *true*) and $q_1 \to q$, and every equality $x_i = x_j$ (resp. disequality $x_i \neq x_j$) in the constraint c is written $i = j$ (resp. $i \neq j$). Note that every TA is the special case of BTTA whose constraints are all equal to *true*.

The *language* $L(A, q)$ of a BTTA A in state q is the set of ground terms *accepted* in state q by A, i.e. the terms t such that $t \xrightarrow[\Delta]{*} q(t)$. The language $L(A)$ of A is $\bigcup_{q \in Q^{\mathsf{f}}} L(A, q)$ and a set of ground terms is called *regular* (resp. *BT-regular*) if it is the language of a TA (resp. BTTA).

A BTTA A is called *deterministic* (resp. *complete*) if for every term $t \in \mathcal{T}(\Sigma)$,

there is at most (resp. at least) one state q such that $t \in L(A,q)$. If A is deterministic and complete, this unique state is denoted $A(t)$. A BTTA A is *normalized* if it does not contain ϵ-transitions, constraints in transitions are defined using partitions, and for every function symbol f with arity m, states $q_1,\dots,q_m$ and partition P of $\{1,\dots,m\}$, A contains exactly one rule of the form $f(q_1,\dots,q_m) \xrightarrow{P} q$. A normalized BTTA A is deterministic and complete, and any BTTA can be transformed into a normalized one recognizing the same language. If A is normalized, we write $A(t,P)$, for a flat term $t \in \mathcal{T}(\Sigma \cup Q)$, to denote the unique state q such that $t \xrightarrow{P} q$ is a transition of A. BTTA are useful for representing the set of normal forms of certain classes of TRS, like flat TRS, see *e.g.* [3].

Lemma 2.1 *[3] Let R be a flat TRS over Σ. There exists a normalized BTTA $B = (Q_B, Q^{\mathsf{f}}_B, \Delta_B)$ on Σ which recognizes the set of ground R-normal forms. Moreover $|Q_B \setminus Q^{\mathsf{f}}_B| = 1$.*

3 Closure under plain rewriting with shallow TRS

Right-(shallow and linear) TRS preserve regularity [13]. It is well known that right-linearity cannot be omitted, as the following example shows.

Example 3.1 Let us consider the TRS $R := \{g(x) \to f(x,x)\}$ and the regular language $L = \{g^n(a) \mid n \geq 0\} = \{a, g(a), g(g(a)), \dots\}$. The set $R^*(L)$ is not regular because its intersection with the regular set $\mathcal{T}(\{f,a\})$ is the non-regular set *Bin* of complete binary trees whose internal nodes are labeled by f and whose leaves are labeled by a, and the class of regular tree languages is closed under intersection. $\diamond$

We show below that considering BTTA does not help in this case.

Proposition 3.2 *In general, $R^*(L)$ is not BT-regular when L is a regular tree language and R a flat TRS.*

Proof. Let us consider R, L and *Bin* as in Example 3.1. The set $R^*(L)$ is not BT-regular. Indeed, its intersection with the regular (hence BT-regular) set $L_2 := \{f(s,t) \mid s \in \mathcal{T}(\{g,a\}), t \in \mathcal{T}(\{f,a\})\}$ is the subset L' of terms $f(s,t) \in L_2$ with $t \in Bin$ and $h(s) = h(t)$. This latter set is not BT-regular, as shown below. It follows that $R^*(L)$ is not BT-regular because the class of BT-regular tree languages is closed under intersection [2].

Let us now show that L' is not BT-regular. Assume that it is recognized by a BTTA $A = (Q, Q^{\mathsf{f}}, \Delta)$ on Σ with n states, and for all $i \geq 1$ let $f(s_i,t_i)$ be the term of L' with $h(s) = h(t) = i$. For each i, there exists a reduction sequence $f(s_i,t_i) \xrightarrow[\Delta]{*} q(f(s_i,t_i))$ with $q \in Q^{\mathsf{f}}$, and we consider the last rule ρ_i of Δ applied in this reduction sequence. There exist two distinct indexes $i_1, i_2 \geq 1$ such that $\rho_{i_1} = \rho_{i_2}$. Let $f(q_1(x_1), q_2(x_2)) \xrightarrow{c} q(f(x_1,x_2))$ be this unique rule of Δ. Note that the constraint c does not contain the equality $x_1 = x_2$, actually c may be $x_1 \neq x_2$ or *true*. In both cases, it follows that $f(s_{i_1}, t_{i_2}) \xrightarrow[\Delta]{*} f(q_1(s_{i_1}), q_2(t_{i_2})) \xrightarrow[\rho_{i_1}]{} q(f(s_{i_1}, t_{i_2}))$. This is contradiction with the fact that $f(s_{i_1}, t_{i_2}) \notin L'$ because $h(s_{i_1}) \neq h(t_{i_2})$. $\square$

4 Closure under innermost rewriting with shallow TRS

The essential problem in Example 3.1 and Proposition 3.2 relies on the fact that after an application of the rule $g(x) \to f(x,x)$ on a term $g(t)$, producing $f(t,t)$, the following application of rewrite rules can change the two occurrences of t in different ways, producing terms $f(t_1,t_2)$ with $t_1 \neq t_2$. The equality constraints of BTTA have not the expressive power to capture the relation relating t_1 and t_2 (i.e. that both are reachable from a common term). The situation is getting better when the innermost strategy is applied.

Example 4.1 When we apply the rule $g(x) \to f(x,x)$ (Example 3.1) with the innermost strategy, the subterm where it is applied must be $g(t)$ for a R-normal form t. Hence, in the term $f(t,t)$ obtained, t cannot be modified by rewriting. Hence $R^{\curlywedge}(L) = \{g^n(t) \mid t \in Bin\}$ is BT-regular. ◇

Note however that $R^{\curlywedge}(L)$ is not regular in the above example.

Proposition 4.2 *In general, $R^{\curlywedge}(L)$ is not regular when L is a regular tree language L and R a flat TRS.*

4.1 Closure of BTTA languages with flat TRS

Linear and flat TRS preserve regularity [14]. This result cannot be extended to BT-regularity, neither for plain nor innermost rewriting.

Proposition 4.3 *In general, $R^*(L)$ and $R^{\curlywedge}(L)$ are not BT-regular when L is BT-regular and R is a flat and linear TRS.*

Proof. The tree language $L = \{h(f^n(0), f^n(0)) \mid n \geq 0\}$ is recognized by the following BTTA, with one equality constraint tested at the root position: $\big(\{q, q^{\mathsf{f}}\}, \{q^{\mathsf{f}}\}, \{0 \to q, f(q) \to q, h(q,q) \xrightarrow{1=2} q^{\mathsf{f}}\}\big)$. Note that L is not regular. Let us consider the flat and linear TRS $R = \{f(x) \to g(x)\}$ and the regular tree language $L' = \{h(f^n(0), g^m(0)) \mid n \geq 0\}$. The closure $R^*(L) \cap L' = \{h(f^n(0), g^n(0))\}$ is not BT-recognizable, hence $R^*(L)$ is neither BT-recognizable. This is also true if we consider innermost rewriting. □

4.2 Closure of TA languages with shallow TRS

The classical approach for proving preservation of regularity [10,13,15] consists of completing a TA recognizing the original language L with new rules inferred using R. This method cannot be generalized to BT-regular languages, according to Proposition 4.3. Therefore, we follow a different approach. We first prove that given L regular and R flat, we can generate a new TRS R_c over an extended signature including a new constant c such that $R_c^{\curlywedge}(\{c\})$ coincides with $R^{\curlywedge}(L)$ on the given signature. This simple and enabling result permits to represent the set of terms reachable from a regular term set as the set of terms reachable from a constant. Later, we show how to compute a BTTA recognizing the reachable terms from a constant. To this end we make use of some results in [7] on innermost rewriting with shallow TRS.

Simplifying assumptions on the signature and the TRS.

From now on in this section, we assume that all terms are built from a given fixed signature Σ which contains several constant symbols and only one non-constant symbol f of arity m. We assume moreover that the TRS R is flat. Such assumptions, already used *e.g.* in [7], can be made without loss of generality for the problem considered here.

Reduction to reachable terms from constants.

Our intention is to reduce the effort of characterizing the reachable terms from a regular language L to just characterizing the reachable terms from a single constant. This is possible using the common idea of adding the inverse rules of an automaton A recognizing L to the rewrite system. The generation of the terms of L starting from the final states of A before any rewrite step application is ensured by the innermost strategy.

Lemma 4.4 *For every flat TRS R and regular language L, over a signature Σ, there exists an extension $\Sigma' \supset \Sigma$, a constant $c \in \Sigma' \setminus \Sigma$ and a flat TRS R_c over Σ' such that $R_c^{\curlywedge}(\{c\}) \cap \mathcal{T}(\Sigma) = R^{\curlywedge}(L)$.*

Weak normal forms and constrained terms.

From [7] we have the following definitions and results. A term t is a *weak normal form* if it is either a constant or a term of the form $t = f(t_1, \ldots, t_m)$ such that every t_i is either a constant or a normal form.

A *constraint* C is a partial function $C : \mathcal{V} \to \mathcal{P}(\Sigma_0)$ ($\mathcal{P}(\Sigma_0)$ denotes the powerset of Σ_0 minus the empty set) i.e. an assignment from variables to non-empty sets of constants. We say that a substitution σ is a *solution* of a constraint C (with respect to a TRS R) if for all x in $dom(C)$, $\sigma(x) \in \mathsf{NF}_R^{\curlywedge}(C(x)) \setminus \Sigma_0$. A *constrained term* is a pair denoted $t|C$, where t is a flat term and C is a constraint, with $dom(C) = vars(t)$. A term $\sigma(t)$ is called an *instance* of $t|C$ if σ is a solution of C. Note that every instance of a constrained term is a weak normal form.

In [7] it is shown how to compute for every flat TRS R and for every constant $c \in \Sigma_0$ two sets of constrained terms r_c and $\overline{r}_c$ satisfying the following properties.

- (a) for every $t|C \in r_c$, there exists at least one solution of C, and all instances of $t|C$ are innermost-reachable from c.
- (b) for every $t|C \in \overline{r}_c$, all non-constant normal form instances of $t|C$ are innermost-reachable from c.
- (c) for every weak normal form s innermost-reachable from c, there exists some constrained term $t|C \in r_c$ such that s is an instance of $t|C$.
- (d) for every non-constant normal form s innermost-reachable from c, there exists some constrained term $f(t_1, \ldots, t_m)|C \in \overline{r}_c$ such that s is an instance of $f(t_1, \ldots, t_m)|C$.

Recognizing terms reachable from constants.

We assume some sets r_c and $\overline{r_c}$ as above and we construct a normalized BTTA A_R which recognizes terms reachable from constants in Σ_0 using R. For this purpose, we shall use the BTTA B of Lemma 2.1 recognizing the ground normal forms of R. Let $Q_0 = \{q \in Q_B \mid \exists d \in \Sigma_0, B(d) = q\}$, and $Q_1 = Q_B^{\mathsf{f}} \setminus Q_0$. Without loss of generality we assume that only constants lead to states in Q_0. Thus, the states of Q_1 characterize the set of non-constant R-normal-form. The states of A_R are pairs $\langle S, q\rangle$, where $S \subseteq \Sigma_0$ and $q \in Q_B$. The intuitive idea is that a term t will lead to $\langle S, q\rangle$ with A_R if it leads to q with B and S is the set of all constants that R-reach t. To this end, the set of transition rules contains:

- $b \to \big\langle \{d \mid b|\emptyset \in r_d\}, B(b)\big\rangle$, for every constant b.
- $f\big(\langle S_1, q_1\rangle, \ldots, \langle S_m, q_m\rangle\big) \xrightarrow{P} \big\langle S, B\big(f(q_1, \ldots, q_m), P\big)\big\rangle$, for every $S_1, \ldots, S_m \subseteq \Sigma_0$, $q_1, \ldots, q_m \in Q_B$, and partition P of $\{1, \ldots, m\}$, and where S is the set of constants $c \in \Sigma_0$ such that there exists $f(\alpha_1, \ldots, \alpha_m)|C \in (r_c \cup \overline{r_c})$ with:

  i. $\forall 1 \leq i \leq m$, if $\alpha_i \in \Sigma_0$ then $\alpha_i \in S_i$ and if $\alpha_i \in \mathcal{V}$ then $C(\alpha_i) \subseteq S_i$ and $q_i \in Q_1$,

  ii. $\forall 1 \leq i < j \leq m$, if $\alpha_i = \alpha_j \in \mathcal{V}$ then $i \equiv_P j$ and if $f(\alpha_1, \ldots, \alpha_m)|C \in \overline{r_c} \setminus r_c$ then $B\big(f(q_1 \ldots q_m), P\big) \in Q_1$ and every $\alpha_i \in \Sigma_0$ $(1 \leq i \leq m)$ is in NF_R.

By construction, the automaton A_R is normalized. The following lemma states the correctness of its construction.

Lemma 4.5 *For all $t \in \mathcal{T}(\Sigma)$ $A_R(t) = \big\langle \{d \in \Sigma_0 \mid d \xrightarrow[R]{\curlywedge} t\}, B(t)\big\rangle$.*

Proof. Let $A_R(t) = \langle S, q\rangle$. It is straightforward by construction that $B(t) = q$. We prove $S = \{d \in \Sigma_0 \mid d \xrightarrow[R]{\curlywedge} t\}$ by induction on the size of t.
We first consider the case where t is a constant. By definition of A_R, $S = \{d \mid (t|\emptyset) \in r_d\}$. It suffices to see that the conditions $d \xrightarrow[R]{\curlywedge} t$ and $(t|\emptyset) \in r_d$ are equivalent when t is a constant. This is a consequence of conditions (a) and (d) above.

Now, assume that t is not a constant. Then, t is of the form $f(t_1, \ldots, t_m)$. Let $\langle S_1, q_1\rangle = A_R(t_1)$, $\ldots$, $\langle S_m, q_m\rangle = A_R(t_m)$. By induction hypothesis, $B(t_i) = q_i$ and $S_i = \{d \in \Sigma_0 \mid d \xrightarrow[R]{\curlywedge} t_i\}$, for every $i \in \{1, \ldots, m\}$. Let $f(\langle S_1, q_1\rangle, \ldots, \langle S_m, q_m\rangle) \xrightarrow{P} \langle S, q\rangle$ be the rule fired in the last applied transition of $A_R(t)$. Then, it holds that $i \equiv_P j$ iff $t_i = t_j$. We prove the two inclusions of $S = \{d \in \Sigma_0 \mid d \xrightarrow[R]{\curlywedge} t\}$ separately.

Direction $\subseteq$. Let $c \in S$. By construction of A_R there exists a constrained term $f(\alpha_1, \ldots, \alpha_m)|C \in (r_c \cup \overline{r_c})$ for which the above conditions i and ii hold.

We obtain an instance of $f(\alpha_1, \ldots, \alpha_m)|C$ by defining a substitution σ for the α_i's that are variables in $\mathit{vars}(f(\alpha_1, \ldots, \alpha_m))$ with $\sigma(\alpha_i) := t_i$. The substitution σ is well defined: if for different i and j we have $\alpha_i = \alpha_j \in \mathcal{V}$, by condition (ii), it implies that $i \equiv_P j$ and then $t_i = t_j$. It holds that every of such $\sigma(\alpha_i)$ is a non-constant normal form, because $\alpha_i \in \mathcal{V}$ implies $q_i = B(t_i) \in Q_1$ due to condition (i). Moreover, $\sigma(\alpha_i)$ is reachable from $C(\alpha_i)$ because $\alpha_i \in \mathcal{V}$ implies $C(\alpha_i) \subseteq S_i$ and $S_i = \{d \in \Sigma_0 \mid d \xrightarrow[R]{\curlywedge} t_i\}$. Altogether, it follows that $\sigma(f(\alpha_1, \ldots, \alpha_m))$ is an instance of $f(\alpha_1, \ldots, \alpha_m)|C \in (r_c \cup \overline{r_c})$.
We know that either $f(\alpha_1, \ldots, \alpha_m)|C \in r_c$ or $f(\alpha_1, \ldots, \alpha_m)|C \in \overline{r_c} \setminus r_c$. On the one hand, if $f(\alpha_1, \ldots, \alpha_m)|C \in r_c$ then, by condition a, $c \xrightarrow[R]{\curlywedge} \sigma(f(\alpha_1, \ldots, \alpha_m))$. On the other hand, if $f(\alpha_1, \ldots, \alpha_m)|C \in \overline{r_c} \setminus r_c$ then our conditions imply that

$B(f(q_1,\ldots,q_m),P) \in Q_1$, and hence $q \in Q_1$ and t is a non-constant normal form. Moreover, the α_i's that are constants are also normal forms. For every one of these constants α_i we know that $\alpha_i \in S_i$, and hence we also have $\alpha_i \xrightarrow[R]{\curlywedge} t_i$. But since this α_i is a normal form it follows that $\alpha_i = t_i$. This implies that $\sigma(f(\alpha_1,\ldots,\alpha_m)) = t$, and hence, that t is a non-constant normal form that is an instance of $f(\alpha_1,\ldots,\alpha_m)|C \in \overline{r_c}$, and by condition b, $c \xrightarrow[R]{\curlywedge} \sigma(f(\alpha_1,\ldots,\alpha_m))$.

Once we know that $c \xrightarrow[R]{\curlywedge} \sigma(f(\alpha_1,\ldots,\alpha_m))$, it suffices to show that $\sigma(f(\alpha_1,\ldots,\alpha_m)) \xrightarrow[R]{\curlywedge} t$ in order to conclude. By the definition of σ, terms $\sigma(f(\alpha_1,\ldots,\alpha_m))$ and t can only differ in the positions i such that α_i is a constant. But in such cases we know that $\alpha_i \in S_i$, and using $S_i = \{d \in \Sigma_0 \mid d \xrightarrow[R]{\curlywedge} t_i\}$ we obtain $\alpha_i \xrightarrow[R]{\curlywedge} t_i$. Hence, $\sigma(f(\alpha_1,\ldots,\alpha_m)) \xrightarrow[R]{\curlywedge} t$ follows.

Direction $\supseteq$. Let c be such that there exists a rewrite sequence $c \xrightarrow[R]{\curlywedge} t$. Since t is not a constant, the previous derivation can be written by making explicit the last rewrite step at position Λ as ($>\Lambda$ represents any position other than Λ):

$$c \xrightarrow[R]{\curlywedge}\xrightarrow[R,\Lambda]{} f(s_1,\ldots,s_m) \xrightarrow[R,>\Lambda]{\curlywedge} t = f(t_1,\ldots,t_m)$$

Hence, there exist (sub-)derivations $s_i \xrightarrow[R]{\curlywedge} t_i$. The term $s = f(s_1,\ldots,s_m)$ is a weak normal form, and hence, by condition c, there exists a constrained term $u|C \in r_c$ such that s is an instance of $u|C$. At this point, either there exists such a u of the form $f(\alpha_1,\ldots,\alpha_m)$, or every u satisfying this condition is a variable. In the second case, s is necessarily a normal form, and hence, by condition d, there exists a constrained term $f(\alpha_1,\ldots,\alpha_m)|C$ in $\overline{r}_c$ such that s is an instance of $f(\alpha_1,\ldots,\alpha_m)|C$. For proving that $c \in S$, it suffices to show that the conditions i and ii hold.

If a certain α_i is a constant, then it coincides with s_i, which R-reaches t_i. Since $S_i = \{d \in \Sigma_0 \mid d \xrightarrow[R]{\curlywedge} t_i\}$, it necessarily contains α_i.

If a certain α_i is a variable, then s_i coincides with t_i and is a non-constant normal form reachable from $C(\alpha_i)$. Hence, $q_i = B(t_i)$ is in Q_1, and again since $S_i = \{d \in \Sigma_0 \mid d \xrightarrow[R]{\curlywedge} t_i\}$, it necessarily includes $C(\alpha_i)$.

If $\alpha_i = \alpha_j \in \mathcal{V}$ then $s_i = s_j$ and since both are normal forms we also have $t_i = t_j$, from which $i \equiv_P j$ follows.

In the case where $f(\alpha_1,\ldots,\alpha_m)|C$ belongs to $\overline{r}_c \setminus r_c$, $f(s_1,\ldots,s_m)$ is a non-constant normal form. Therefore, $q = B(f(q_1,\ldots,q_m),P) \in Q_1$ and all the constants α_i are also normal forms. □

Given a flat TRS R and a regular L, the BTTA A_{R_c} (for R_c associated to R as in Lemma 4.4), restricted to the signature Σ of R, recognizes $R^{\curlywedge}(L)$ by marking as accepting states the pairs $\langle S,q\rangle$ such that $c \in S$, according to Lemmas 4.4 and 4.5. This permits to conclude the proof of Theorem 4.6.

Theorem 4.6 *$R^{\curlywedge}(L)$ is BT-regular when L is regular and R is shallow.*

A term t is *reachable* from another term s if there exists a rewrite sequence that transforms s into t. Two terms s and t are *joinable* if there exists a term u reachable from s and t. Ground reachability and joinability (the restriction of these problems to ground terms) are undecidable for flat TRS [12]. Theorem 4.6 show that they become decidable when the innermost strategy is applied.

Corollary 4.7 *Ground reachability and joinability are decidable for ground terms for innermost rewriting with shallow TRS.*

Proof. When restricting to innermost rewriting, t is reachable from s iff $t \in R^{\curlywedge}(\{s\})$. Since $\{s\}$ is a regular language when s is ground, $R^{\curlywedge}(\{s\})$ is BT-regular by Theorem 4.6. Therefore ground reachability reduces to the membership problem for BTTA, which is decidable.

Similarly, s and t are joinable iff $R^{\curlywedge}(\{s\}) \cap R^{\curlywedge}(\{t\}) \neq \emptyset$. By Theorem 4.6 and closure of BT-languages under Boolean operations [3], we obtain a reduction of ground joinability to the emptiness problem for BTTA, which is also decidable. $\Box$

In [1] the decidability of the regularity of a BTTA was shown. Combining this result with Theorem 4.6 we obtain the following corollary.

Corollary 4.8 *Given a regular language L and a shallow TRS R, it is decidable whether $R^{\curlywedge}(L)$ is regular.*

This result does not hold when we deal with plain rewriting. In [12] it has been proved that reachability of flat TRS is undecidable by reducing the Post correspondence problem into $0 \xrightarrow[R]{*} 1$. We show below how to extend R into R_0 such that ${R_0}^*(0)$ is regular iff $0 \xrightarrow[R]{*} 1$.

Theorem 4.9 *Given a regular language L and a flat TRS R, it is undecidable whether $R^*(L)$ is regular.*

Proof. In [12] it is proved that reachability of flat TRS is undecidable by reducing a PCP instance P into a TRS R over a signature including $\{0, 1\}$ such that P has a solution iff $0 \xrightarrow[R]{*} 1$. The reduction in [12] also satisfies that if P has no solution, the 0 does not reach any term containing 1 nor any term containing 0 properly.

This reduction can be modified by adding new symbols $\{f, h, g, a, b, c\}$ to the current signature Σ, and adding two new sets of rules to R: $R_1 = \{0 \to f(a, b), a \to g(a), b \to g(b), a \to c, b \to c, f(x, x) \to h(x, x)\}$ and R_2 containing all the necessary rules for making $R_2^*(1)$ to be $\mathcal{T}(\Sigma \cup \{f, h, a, b, c\})$. The rules of R ensure $(R \cup R_1 \cup R_2)^*(0)$ to be a non-regular language, unless $0 \xrightarrow[R]{*} 1$. Note that if P has solution, then $0 \xrightarrow[R]{*} 1$, and hence $(R \cup R_1 \cup R_2)^*(0)$ is $\mathcal{T}(\Sigma \cup \{f, h, a, b, c\})$, which is regular. Otherwise, if P has no solution, then 0 does not reach any term containing 1, nor containing 0 properly, and hence $(R \cup R_1 \cup R_2)^*(0) \cap \mathcal{T}(\{h, c\})$ is the set $\{h(g^n(c), g^n(c)) \mid n > 0\}$, which is not regular. $\Box$

5 Innermost rewriting and right-shallow TRS

In this section, we study the closure of regular languages under innermost rewriting with TRS whose right-hand sides of rules are shallow. We show that regularity is preserved by innermost rewriting with linear right-shallow TRS (Subsection 5.1), but not by innermost rewriting with right-(linear and flat) (non left-linear) TRS (Subsection 5.2).

5.1 TA languages and linear and right-shallow TRS

First, we observe that every right-shallow TRS R can be transformed into a right-flat TRS R' (on an extended signature) such that for all $s, t \in \mathcal{T}(\Sigma)$, $s \xrightarrow[R]{\curlywedge} t$ iff $s \xrightarrow[R']{\curlywedge} t$. The idea is to add a new constant c_r and a rule $c_r \to r$ for every ground proper subterm r of a right-hand side of a rule of R, and to replace r by c_r in all the right-hand sides of R.

Let $A = (Q, Q^{\mathsf{f}}, \Delta)$ be a deterministic and complete TA on Σ recognizing a tree language L, and let R be a linear and right-flat TRS. For all $c \in \Sigma_0$ we denote as q_c the unique state of Q such that $c \to q_c \in \Delta$. We assume moreover *wlog* that $L(A, q_c) = \{c\}$.

We construct a finite sequence of TA $A_0, A_1, \ldots$ whose last element recognizes $R^{\curlywedge}(L)$. The construction of the sequence is incremental. Every A_{k+1} is obtained from A_k by the addition of some new transitions, such that if some term s is recognized by A_k and s rewrites (in one step of innermost rewriting) to t, then t is recognized by A_{k+1}.

In order to restrict to innermost rewriting, we shall use a complete and deterministic TA $B = (Q_B, Q^{\mathsf{f}}_B, \Delta_B)$ (without ε-transitions) recognizing the ground R-normal forms (see *e.g.* [3] for its construction). As in Lemma 2.1, we can assume that B has only one non-accepting state q_{reject}. Let A_0 be a TA recognizing $L(A)$: $A_0 := \big(Q \times Q_B, Q^{\mathsf{f}} \times Q_B, \Delta_0\big)$ where Δ_0 is the set of transitions $f\big(\langle q_1, q'_1\rangle, \ldots, \langle q_m, q'_m\rangle\big) \to \langle q, q'\rangle$ such that $f(q_1, \ldots, q_m) \to q \in \Delta$ and $f(q'_1, \ldots, q'_m) \to q' \in \Delta_B$.

The addition of transition rules to A_k, giving A_{k+1}, is defined by the superposition of rules of R into a sequence of transitions of Δ_k. More precisely, $A_{k+1} \setminus A_k$ contains all the transitions which can be constructed from a rewrite rule $\ell \to r$ of R (we let $\ell = f(\ell_1, \ldots, \ell_m)$) and a substitution θ of the variables of ℓ into states of $Q \times Q^{\mathsf{f}}_B$ whose accepted language wrt A_k is not empty, such that: $\theta(\ell) \xrightarrow[\Delta_k]{*} \langle q_0, q_{\mathsf{reject}}\rangle$, and the last step of the above reduction is $f\big(\langle q_1, q'_1\rangle, \ldots, \langle q_m, q'_m\rangle\big) \xrightarrow[\Delta_k]{} \langle q_0, q_{\mathsf{reject}}\rangle$ and for all $i \le m$, $q'_i \ne q_{\mathsf{reject}}$. There are two cases for the transitions of $A_{k+1} \setminus A_k$:

- case 1: r is a variable. In this case, $r \in \mathit{vars}(\ell)$. Let $\langle \overline{q}, \overline{q}'\rangle = \theta(r)$, we add the ε-transition $\langle \overline{q}, \overline{q}'\rangle \to \langle q_0, \overline{q}'\rangle$.
- case 2: $r = g(r_1, \ldots, r_m)$. We add all the transitions $g\big(\langle \overline{q}_1, \overline{q}'_1\rangle, \ldots, \langle \overline{q}_m, \overline{q}'_m\rangle\big) \to \langle q_0, \overline{q}'\rangle$ such that $g(\overline{q}'_1, \ldots, \overline{q}'_m) \to \overline{q}' \in \Delta_B$ and for each $i \le m$, if r_i is a variable then $\langle \overline{q}_i, \overline{q}'_i\rangle := \theta(r_i)$, otherwise, if r_i is a constant then $\overline{q}_i$ is q_{r_i} and there is no restriction for $\overline{q}'_i$.

All the TAs have the same state set, hence the construction terminates with a fixpoint denoted A^*. We can now show that $L(A^*) = R^{\curlywedge}(L(A))$, more precisely, that for all $t \in \mathcal{T}(\Sigma)$, $t \in L\big(A^*, \langle q, q'\rangle\big)$ iff $t \in L(B, q')$ and there exists $s \in L(A, q)$ such that $s \xrightarrow[R]{\curlywedge} t$. To this end we follow the principle of the proofs given *e.g.* in [14,10,13], but some technical difficulties appear when we try to replace a subterm by another subterm while preserving an execution with Δ^*. They are solved thanks to the following technical Lemma 5.2.

The if direction is proved by induction on the number of rewrite steps in $s \xrightarrow[R]{\curlywedge} t$, using Lemma 5.2. The other direction is proved by an induc-

tion on the multiset associated to the derivation $t \xrightarrow[\Delta^*]{*} \langle q, q' \rangle$ by mapping each transition rule ρ used to the least index i of the A_i to which ρ belongs.

Lemma 5.2 *For all $t \in \mathcal{T}(\Sigma \cup Q_{A^*})$, if $t\big[\langle q_0, q_{\mathsf{reject}} \rangle\big]_p \xrightarrow[\Delta^*]{*} \langle q, q_{\mathsf{reject}} \rangle$ then, for all $q' \in Q_B$, there exists $q'' \in Q_B$ such that $t[\langle q_0, q' \rangle]_p \xrightarrow[\Delta^*]{*} \langle q, q'' \rangle$.*

Proof. First of all, we note that no ε-transition is applied in $t \xrightarrow[\Delta^*]{*} \langle q, q_{\mathsf{reject}} \rangle$ at any position $p' \leq p$ *wrt* the prefix ordering. This is due to the fact that the ε-transitions are always of the form $\langle _, q_{nr} \rangle \to \langle _, q_{nr} \in Q_B \rangle$ for some q_{nr} different from q_{reject}, since this kind of transitions are derived in case 1 using rules where the right-hand side is a variable, and hence, it is necessarily instantiated by a normal form.

Hence, it suffices to prove the statement of the lemma for the case where t is of the form $f(\langle q_1, q_1' \rangle, \ldots, \langle q_m, q_m' \rangle)$, p is a certain position i, and there is just one rewrite step derivation with a rule of the form $f(\langle q_1, q_1' \rangle, \ldots, \langle q_i, q_i' \rangle, \ldots, \langle q_m, q_m' \rangle) \to \langle q, q_{\mathsf{reject}} \rangle$. The global statement is then obtained inductively. In this case, q_i is q_0, and q_i' is q_{reject}. If the rule $f(\langle q_1, q_1' \rangle, \ldots, \langle q_i, q_i' \rangle, \ldots, \langle q_m, q_m' \rangle) \to \langle q, q_{\mathsf{reject}} \rangle$ is in Δ_0, by the definition of Δ_0 we have that for any $q' \in Q_B$ there exists also an alternative rule of the form $f(\langle q_1, q_1' \rangle, \ldots, \langle q_i, q' \rangle, \ldots, \langle q_m, q_m' \rangle) \to \langle q, q'' \rangle$, for some q'', and we are done. Otherwise, assume that this rule $f(\langle q_1, q_1' \rangle, \ldots, \langle q_i, q_i' \rangle, \ldots, \langle q_m, q_m' \rangle) \to \langle q, q_{\mathsf{reject}} \rangle$ is not in Δ_0. Then it has been derived by case 2 using a rule $\ell \to f(r_1, \ldots, r_m) \in R$. Moreover, by the conditions of case 2 and the fact that q_i' is q_{reject} we know that r_i is not a variable, and hence, it is a constant. Again by the conditions of case 2, for any $q' \in Q_B$ there exists also an alternative rule of the form $f(\langle q_1, q_1' \rangle, \ldots, \langle q_i, q' \rangle, \ldots, \langle q_m, q_m' \rangle) \to \langle q, q'' \rangle$, for some q'', and we are done. □

The number $|A^*|$ of states of A^* is at most $|A| \times |B|$, and the number of rules of A^* is polynomial in the same measure[3], if we assume as usual that the maximum arity of a function symbol is fixed for the problem.

Theorem 5.3 *$R^{\curlywedge}(L)$ is regular when L is regular and R is linear and right-shallow.*

Proof.

- Direction $\Leftarrow$ We prove it by induction on the number of rewrite steps of $s \xrightarrow[R]{\curlywedge} t$. For 0 rewrite steps we have $s = t$, and hence, $t \xrightarrow[\Delta]{*} q$. From the construction of Δ_0, $t \xrightarrow[\Delta_0]{*} \langle q, q' \rangle$ follows, and since $\Delta_0 \subseteq \Delta^*$ we also have $t \xrightarrow[\Delta^*]{*} \langle q, q' \rangle$, and we are done. Hence, assume that there is at least one rewrite step in $s \xrightarrow[R]{\curlywedge} t$. We can write this derivation by making explicit the last rewrite step as $s \xrightarrow[R]{\curlywedge} t[\sigma(l)]_p \xrightarrow[\ell \to r, p, \sigma]{\imath} t[\sigma(r)]_p = t$. By induction hypothesis, $t[\sigma(\ell)]_p \xrightarrow[\Delta^*]{*} \langle q, q_{\mathsf{reject}} \rangle$ (the q_{reject} is due to the fact that $t[\sigma(\ell)]_p$ is not a normal form). By re-ordering the rule applications of Δ^*, we can assume that this derivation is of the form

$$t[\sigma(\ell)]_p \xrightarrow[\Delta^*]{*} t[\theta(\ell)]_p \xrightarrow[\Delta^*]{*} t[\langle q_0, q_{\mathsf{reject}} \rangle]_p \xrightarrow[\Delta^*]{*} \langle q, q_{\mathsf{reject}} \rangle$$

where θ is a mapping from variables to states of Δ^*. By the conditions for adding new rules into the Δ_j's, we also have $\theta(r) \xrightarrow[\Delta^*]{*} \langle q_0, q_0' \rangle$ for some $q_0' \in Q_B$, and since the variables of r occur also in ℓ, there exists a derivation $\sigma(r) \xrightarrow[\Delta^*]{*}$

[3] Note however that $|B|$ can be exponential in the size of R in worst case.

$\theta(r) \xrightarrow[\Delta^*]{*} \langle q_0, q_0' \rangle$, and hence, by Lemma 5.2,

$$t[\sigma(r)]_p \xrightarrow[\Delta^*]{*} t[\theta(r)]_p \xrightarrow[\Delta^*]{*} t[\langle q_0, q_0' \rangle]_p \xrightarrow[\Delta^*]{*} \langle q, q'' \rangle$$

for some q''. But this q'' is necessarily q' since Δ^* simulates B on the second component of the pairs, and we are done.

- Direction $\Rightarrow$ The fact that $t \xrightarrow[B]{*} q'$ follows directly from the construction of B: note that all the Δ_i's always simulate the execution of B for the second component of the pair of states. For proving that $\exists s : (s \xrightarrow[\Delta]{*} q \wedge s \xrightarrow[R]{\curlywedge} t)$, we do an induction on the measure defined as follows. For a concrete derivation $t \xrightarrow[\Delta^*]{*} \langle q, q' \rangle$, we call $\mathsf{Rules}(t \xrightarrow[\Delta^*]{*} \langle q, q' \rangle)$ the multiset of all rules used in it, and define $\mathsf{Ind}(t \xrightarrow[\Delta^*]{*} \langle q, q' \rangle)$ as the multiset obtained by replacing every occurrence of a rule $\ell \to r$ in $\mathsf{Rules}(t \xrightarrow[\Delta^*]{*} \langle q, q' \rangle)$ by the least index i such that $\ell \to r \in \Delta_i$. We compare indexes of derivations by the multiset extension of the usual ordering on natural numbers, and prove the statement of the lemma by induction on this ordering.

  Now, we distinguish cases depending on whether the last rewrite step of $t \xrightarrow[\Delta^*]{*} \langle q, q' \rangle$ is an ε-transition or not.

  Assume first that it is an ε-transition. Then, this derivation can be written of the form $t \xrightarrow[\Delta^*]{*} \langle q_0, q' \rangle \xrightarrow[\Delta^*]{} \langle q, q' \rangle$ for a certain state $\langle q_0, q' \rangle$ of Q_{Δ^*}. The rule $\langle q_0, q' \rangle \to \langle q, q' \rangle$ is not in Δ_0, and has been necessarily added into some Δ_i with $i > 0$ by case 1, using a collapsing rule $\ell \to x \in R$ and a substitution θ such that $\theta(x) = \langle q_0, q' \rangle$ and $\theta(\ell) \xrightarrow[\Delta_{i-1}]{*} \langle q, q_{\mathsf{reject}} \rangle$. Moreover, for the rest of variables y of ℓ different from x, there exist terms $t_y \in \mathcal{T}(\Sigma)$ such that $t_y \xrightarrow[\Delta_{i-1}]{*} \theta(y)$.

  We define a substitution σ such that $\sigma(y) = t_y$ for every of such y, and $\sigma(x) = t$. The substitution σ is well defined because R is right-linear.

  Hence, we have a one rewrite step derivation $\sigma(\ell) \xrightarrow[\ell \to x, \sigma]{\imath} t$, which is innermost due to the conditions for the addition of the rule to Δ_i (condition $q_i' \neq q_{\mathsf{reject}}$ there). Now, note that the multiset $\mathsf{Ind}(t \xrightarrow[\Delta^*]{*} \langle q_0, q' \rangle)$ has just one less i than $\mathsf{Ind}(t \xrightarrow[\Delta^*]{*} \langle q, q' \rangle)$, and that the multiset $\mathsf{Ind}(\theta(\ell) \xrightarrow[\Delta_{i-1}]{*} \langle q, q_{\mathsf{reject}} \rangle)$ and the respective multisets $\mathsf{Ind}(t_y \xrightarrow[\Delta_{i-1}]{*} \theta(y))$ contain numbers smaller than i. Therefore, composing the previous derivations we can construct a derivation $\sigma(\ell) \xrightarrow[\Delta^*]{*} \langle q, q_{\mathsf{reject}} \rangle$ smaller than $t \xrightarrow[\Delta^*]{*} \langle q, q' \rangle$. Hence, by induction hypothesis, there exists a term s such that $s \xrightarrow[R]{\curlywedge} \sigma(\ell)$ and $s \xrightarrow[\Delta]{*} q$, and, since t is R-reachable from s, we are done.

  Now, assume that the last rewrite step of $t \xrightarrow[\Delta^*]{*} \langle q, q' \rangle$ is not an ε-transition. Hence, this derivation can be written by making explicit the last step as $t \xrightarrow[\Delta^*]{*} f(\langle q_1, q_1' \rangle, \ldots, \langle q_m, q_m' \rangle) \xrightarrow[\Delta^*]{} \langle q, q' \rangle$. Thus, if we write t of the form $f(t_1, \ldots, t_m)$, we can take (sub-)derivations $t_1 \xrightarrow[\Delta^*]{*} \langle q_1, q_1' \rangle$, $\ldots$, $t_m \xrightarrow[\Delta^*]{*} \langle q_m, q_m' \rangle$. Every $\mathsf{Ind}(t_i \xrightarrow[\Delta^*]{*} \langle q_i, q_i' \rangle)$ is smaller than $\mathsf{Ind}(t \xrightarrow[\Delta^*]{*} \langle q, q' \rangle)$, and thus, by induction hypothesis there exist terms $s_1, \ldots s_m$ and derivations $s_i \xrightarrow[\Delta]{*} q_i$ and $s_i \xrightarrow[R]{\curlywedge} t_i$.

  At this point we distinguish cases depending on whether the rule $f(\langle q_1, q_1' \rangle, \ldots, \langle q_m, q_m' \rangle) \to \langle q, q' \rangle$ belongs to Δ_0 or not. Assume first that it belongs to Δ_0. Then, by the definition of Δ_0, there exists also a rule $f(q_1, \ldots, q_m) \to q$ in Δ. Therefore, by composing this and the previous derivations, we have $f(s_1, \ldots, s_m) \xrightarrow[\Delta]{*} q$, but also $f(s_1, \ldots, s_m) \xrightarrow[R]{\curlywedge} f(t_1, \ldots, t_m) = t$, and we are done.

Now, assume that $f(\langle q_1, q'_1\rangle, \ldots, \langle q_m, q'_m\rangle) \to \langle q, q'\rangle$ has been added by case 2 into some Δ_i with $i > 0$ using a rule of the form $\ell \to f(r_1, \ldots, r_m)$ and a substitution θ satisfying the following properties. On the one side, $\theta(\ell) \xrightarrow[\Delta_{i-1}]{*} \langle q, q_{\mathsf{reject}}\rangle$. On the other side, for every r_i, if r_i is a variable then $\langle q_i, q'_i\rangle$ is $\theta(r_i)$ and $q'_i \neq q_{\mathsf{reject}}$, and otherwise, if r_i is a constant then q_i is q_{r_i} and there is no restriction on q'_i. Moreover, for the variables y occurring in ℓ and not in $f(r_1, \ldots, r_m)$ there exist terms $t_y \in \mathcal{T}(\Sigma)$ such that $t_y \xrightarrow[\Delta_{i-1}]{*} \theta(y)$. We define a substitution σ such that $\sigma(y) = t_y$ for every of such y, and $\sigma(r_i) = t_i$ for the r_i that are variables. As above, σ is well defined because R is right-linear. Hence, we have a one rewrite step derivation $\sigma(\ell) \xrightarrow[\ell \to f(r_1,\ldots,r_m),\sigma]{} \sigma(f(r_1, \ldots, r_m))$, which is innermost due to the conditions for the addition of the rule to Δ_i, again. The terms $\sigma(f(r_1, \ldots, r_m))$ and $t = f(t_1, \ldots, t_m)$ can differ only in the positions i such that r_i is a constant. For an i of this kind, q_i is q_{r_i}, and the only term of $\mathcal{T}(\Sigma)$ that can be derived into q_{r_i} with Δ is r_i. Therefore, s_i is r_i, and hence, there exists a derivation $r_i \xrightarrow[R]{\curlywedge} t_i$.

This implies that $\sigma(f(r_1, \ldots, r_m)) \xrightarrow[R]{\curlywedge} t$. To conclude, it suffices to see that there exists a term s such that $s \xrightarrow[R]{\curlywedge} \sigma(\ell)$ and $s \xrightarrow[\Delta]{*} q$. To this end, we will prove that there exists a derivation $\sigma(\ell) \xrightarrow[\Delta^*]{*} \langle q, q_{\mathsf{reject}}\rangle$ smaller than $t \xrightarrow[\Delta^*]{*} \langle q, q'\rangle$, and the statement will follow by induction hypothesis. But this is identical to what we have done in a previous case. Note that the multiset $\mathsf{Ind}(\theta(\ell) \xrightarrow[\Delta_{i-1}]{*} \langle q, q_{\mathsf{reject}}\rangle)$ and the respective multisets $\mathsf{Ind}(t_y \xrightarrow[\Delta_{i-1}]{*} \theta(y))$ contain numbers smaller than i. Moreover, for the variables x that occur in $f(r_1, \ldots, r_m)$, left-linearity ensures that the indexes are preserved in these cases. Therefore, composing the previous derivations we can construct a derivation $\sigma(\ell) \xrightarrow[\Delta^*]{*} \langle q, q_{\mathsf{reject}}\rangle$ smaller than $t \xrightarrow[\Delta^*]{*} \langle q, q'\rangle$, and we are done.

□

5.2 *Closure of TA languages with right-(linear and flat) TRS*

When we drop the restriction that R is left-linear in Theorem 5.3, we lose regularity preservation with innermost rewriting. This is in contrast with plain rewriting, where regularity is preserved for right-linear and right-shallow TRS [13].

Proposition 5.4 *In general, $R^{\curlywedge}(L)$ is not BT-regular when L is regular and R is right-linear and right-flat.*

Proof. Let $L = \{f(f(a, a), c)\}$, and $R = \{f(x, c) \to x, f(g(x), x) \to h(x), h(x) \to h(x), a \to g(a), a \to b\}$. The intersection of $R^{\curlywedge}(L)$ with the language of all terms containing only the symbols f, g, b is the set $\{f(g^n(b), g^m(b)) \mid n \neq m + 1\}$, which is not BT-regular. □

6 Conclusion and further work

We have covered much of the cases of closure of TA and BTTA languages by innermost rewriting, providing results for each case. The positive results are that the set of terms innermost-reachable from a regular language with a shallow TRS is BT-regular, and it is regular when the TRS is linear and right-shallow. Moreover,

given a shallow TRS, regularity of the reachable terms from a regular language is decidable. Other consequences are the decidability of the problems of *ground reachability*, *ground joinability* and *regular tree model checking* (given two regular languages L_{init} and L_{bad} and the TRS R, do we have $R^{\curlywedge}(L_{\mathsf{init}}) \cap L_{\mathsf{bad}} = \emptyset$?) for innermost rewriting with TRS in the above classes.

As future work, it could be interesting to consider other variants of tree automata with more general or different constraints, and to consider other strategies of rewriting different from innermost.

References

[1] B. Bogaert, F. Seynhaeve, and S. Tison. The recognizability problem for tree automata with comparisons between brothers. In Wolfgang Thomas, editor, *Foundations of Software Science and Computation Structure, Second International Conference, FoSSaCS'99*, volume 1578 of *Lecture Notes in Computer Science*, pages 150–164. Springer, March 1999.

[2] B. Bogaert and S. Tison. Equality and disequality constraints on direct subterms in tree automata. In *International Symposium on Theoretical Aspects of Computeter Science*, pages 161–171, 1992.

[3] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 2007.

[4] N. Dershowitz and J. P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science (Vol. B: Formal Models and Semantics)*, pages 243–320, Amsterdam, 1990. North-Holland.

[5] I. Durand and G. Sénizergues. Bottom-up rewriting is inverse recognizability preserving. In *18th Int. Conf. Term Rewriting and Applications (RTA'07)*, volume 4533 of *LNCS*, pages 107–121. Springer, 2007.

[6] R. Gilleron and S. Tison. Regular tree languages and rewrite systems. *Fundam. Inform.*, 24(1/2):157–174, 1995.

[7] G. Godoy and E. Huntingford. Innermost-reachability and innermost-joinability are decidable for shallow term rewrite systems. In F. Baader, editor, *Term Rewriting and Applications, 18th International Conference, RTA*, volume 4533 of *LNCS*, pages 184–199. Springer, June 2007.

[8] G. Godoy, E. Huntingford, and A.Tiwari. Termination of rewriting with right-flat rules. In F. Baader, editor, *Term Rewriting and Applications, 18th International Conference, RTA*, volume 4533 of *LNCS*, pages 200–213. Springer, June 2007.

[9] G. Godoy and A. Tiwari. Confluence of shallow right-linear rewrite systems. In C.-H. Luke Ong, editor, *19th International Workshop of Computer Science Logic, CSL*, volume 3634 of *LNCS*, pages 541–556. Springer, August 2005.

[10] F. Jacquemard. Decidable approximations of term rewriting systems. In *Rewriting Techniques and Applications, 7th International Conference*, pages 362–376, 1996.

[11] Y. Kojima and M. Sakai. Innermost reachability and context sensitive reachability properties are decidable for linear right-shallow term rewriting systems. In *Rewriting Techniques and Applications, 19th International Conference, 2008*, to appear.

[12] I. Mitsuhashi, M. Oyamaguchi, and F. Jacquemard. The confluence problem for flat TRSs. In *Proc. 8th Intl. Conf. on Artificial Intelligence and Symbolic Computation (AISC'06)*, volume 4120 of *LNAI*, pages 68–81. Springer, 2006.

[13] T. Nagaya and Y. Toyama. Decidability for left-linear growing term rewriting systems. In *Proc. 10th Intl. Conf. RTA*, volume 1631 of *LNCS*, pages 256–270. Springer, 1999.

[14] K. Salomaa. Deterministic tree pushdown automata and monadic tree rewriting systems. *J. Comput. Syst. Sci.*, 37:367–394, 1988.

[15] T. Takai, Y. Kaji, and H. Seki. Right-linear finite path overlapping term rewriting systems effectively preserve recognizability. In *Rewriting Techniques and Applications, RTA*, volume 1833 of *LNCS*, pages 246–260, 2000.

[16] W.S.Brainerd. Tree generating regular systems. *Information and Control*, 4:217–231, 1969.

Author Index

Falke, Stephan ... 46
Fernandez, Maribel ... 1

Gascón, Adrià ... 110
Godoy, Guillem ... 110

Heeren, Bastiaan ... 31

Jacquemard, Florent ... 110
Jeuring, Johan ... 31

Kapur, Deepak ... 46

Lescanne, Pierre ... 91
Lucanu, Dorel ... 16

Nishida, Naoki ... 61

Raffelsieper, Matthias ... 76

Sakai, Masahiko ... 61
Siafakas, Nikolaos ... 1

Zantema, Hans ... 76
Žunić, Dragiša ... 91