The 2007 Federated Conference on
Rewriting, Deduction and Programming

Paris, France

June 25 – 29, 2007

WRS'07

The 7th International Workshop on Reduction Strategies in Rewriting and Programming

June 25th, 2007

Proceedings

Editor:
Jürgen Giesl

Preface

This volume contains the preliminary proceedings of the *7th International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2007)*. The final proceedings of WRS 2007 will be published as a volume of ENTCS (Electronic Notes in Theoretical Computer Science). The workshop was held in Paris, France, on June 25, 2007, as part of the *4th Federated Conference on Rewriting, Deduction, and Programming (RDP 2007)*.

The WRS workshop intends to promote and stimulate international 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. The previous editions of the workshop were: WRS 2001 (Utrecht, The Netherlands), WRS 2002 (Copenhagen, Denmark), WRS 2003 (Valencia, Spain), WRS 2004 (Aachen, Germany), WRS 2005 (Nara, Japan), and WRS 2006 (Seattle, USA).

Each submission to WRS 2007 was assigned to at least three Program Committee members, who reviewed the papers with the help of 7 external referees. Afterwards, the submissions were discussed by the Program Committee by means of Andrei Voronkov's *EasyChair* system. I am grateful to Andrei for providing his system which was very helpful for the management of the submissions and reviews and for the discussion of the Program Committee.

The program of WRS 2007 includes submitted 12 papers. In addition, WRS 2007 had two invited speakers, *Pierre-Etienne Moreau* and *Rachid Echahed*. I want to thank the invited speakers for their interesting and inspiring talks.

Many people helped to make WRS 2007 a success. In particular, I want to thank the organizers of RDP for their help (especially Ralf Treinen and Xavier Urbain). I am also very grateful to the members of the Program Committee and to the external reviewers for their careful work.

I would also like to thank the institutional sponsors of RDP 2007 without whom it would not have been possible to organize RDP 2007: the Conservatoire des Arts et Métiers (CNAM), the Centre National de la Recherche Scientifique (CNRS), the École Nationale Supérieure d'Informatique pour l'Industrie et l'Entreprise (ENSIEE), the GDR Informatique Mathématique, the Institut National de Recherche en Informatique et Automatique (INRIA) unit Futurs, and the Région Île de France.

Aachen, June, 2007 *Jürgen Giesl*

Program Chair

Jürgen Giesl

Program Committee

Sergio Antoy	*Portland State U, USA*
Horatiu Cirstea	*LORIA Nancy, France*
Manuel Clavel	*U Madrid, Spain*
Francisco Durán	*U Malaga, Spain*
Maribel Fernández	*KC London, UK*
Jürgen Giesl	*RWTH Aachen, Germany*
Bernhard Gramlich	*TU Vienna, Austria*
Salvador Lucas	*U Valencia, Spain*
Aart Middeldorp	*U Innsbruck, Austria*
Mizuhito Ogawa	*JAIST Nomi, Japan*
Vincent van Oostrom	*U Utrecht, The Netherlands*
Jaco van de Pol	*CWI Amsterdam, The Netherlands*
Masahiko Sakai	*U Nagoya, Japan*

External Reviewers

Paul Brauner
Guillem Godoy
Martin Korp
Ian Mackie
Antoine Reilles
Manfred Schmidt-Schauß
François-Régis Sinot

Contents

Innermost Termination of Rewrite Systems by Labeling[1]

René Thiemann

LuFG Informatik 2, RWTH Aachen
52074 Aachen, Germany

Aart Middeldorp

Institute of Computer Science, University of Innsbruck
6020 Innsbruck, Austria

Abstract

Semantic labeling is a powerful transformation technique for proving termination of term rewrite systems. The semantic part is given by a model or a quasi-model of the rewrite rules. A variant of semantic labeling is predictive labeling where the quasi-model condition is only required for the usable rules. In this paper we investigate how semantic and predictive labeling can be used to prove innermost termination. Moreover, we show how to reduce the set of usable rules for predictive labeling even further, both in the termination and the innermost termination case.

Keywords: Innermost Termination, Predictive Labeling, Semantic Labeling, Term Rewriting, Termination

1 Introduction

We start our discussion by illustrating the limitations of existing versions of semantic and predictive labeling on a concrete example. Consider the following rewrite system $\mathcal{R}$ where $x \div y$ generates a number between 0 and $\lfloor \frac{x}{y} \rfloor$:

$$x \geqslant 0 \to \mathsf{true} \quad (1)$$
$$0 \geqslant \mathsf{s}(y) \to \mathsf{false} \quad (2)$$
$$\mathsf{s}(x) \geqslant \mathsf{s}(y) \to x \geqslant y \quad (3)$$
$$x - 0 \to x \quad (4)$$
$$0 - y \to 0 \quad (5)$$
$$\mathsf{s}(x) - \mathsf{s}(y) \to x - y \quad (6)$$
$$\mathsf{id\text{-}inc}(x) \to x \quad (7)$$
$$\mathsf{id\text{-}inc}(x) \to \mathsf{s}(x) \quad (8)$$
$$x \div y \to \mathsf{if}(y \geqslant \mathsf{s}(0), x \geqslant y, x, y) \quad (9)$$
$$\mathsf{if}(\mathsf{false}, b, x, y) \to \mathsf{div\text{-}by\text{-}zero} \quad (10)$$
$$\mathsf{if}(\mathsf{true}, \mathsf{false}, x, y) \to 0 \quad (11)$$
$$\mathsf{if}(\mathsf{true}, \mathsf{true}, x, y) \to \mathsf{id\text{-}inc}((x - y) \div y) \quad (12)$$

[1] Supported by DFG (Deutsche Forschungsgemeinschaft) grant GI 274/5-1 and FWF (Austrian Science Fund) project P18763.

Proving termination of $\mathcal{R}$ is a difficult task. Consider the recursive calls of $\div$ and if in rules (9) and (12). Essentially, one has to find a well-founded order such that the argument x of if is larger than the argument $x - y$ of $\div$. To this end, one can use the fact that in the previous recursive call the terms $y \geqslant \mathsf{s}(\mathsf{0})$ and $x \geqslant y$ are both reducible to true. This knowledge is important as for $x = \mathsf{0}$ or $y = \mathsf{0}$ the term $x - y$ can be reduced to x. However, when using term orders one generates one separate constraint for each rule of $\mathcal{R}$. Thus, the knowledge of a previous recursive call is not directly available when building the constraint for rule (12). For example, polynomial interpretations with negative coefficients [5] are not expressive enough to solve the constraints of rules (9) and (12).

To solve this problem one can use the technique of *semantic labeling* [9]. We can take an algebra $\mathcal{A}$ over natural numbers $\mathbb{N}$ where we use the natural interpretation for the symbols $-$, s, $\mathsf{0}$, false, true, and $\geqslant$, i.e., $x -_{\mathcal{A}} y = \max(x-y, 0)$, $\mathsf{s}_{\mathcal{A}}(x) = x+1$, $\mathsf{0}_{\mathcal{A}} = \mathsf{false}_{\mathcal{A}} = 0$, $\mathsf{true}_{\mathcal{A}} = 1$, and $x \geqslant_{\mathcal{A}} y = 1$ if $x \geqslant y$, and 0 otherwise. Now, we can also provide *labeling functions* ℓ_f which define how to label the function symbol f in a term $f(t_1, \ldots, t_n)$, depending on the value of their arguments. E.g., we can choose $\ell_{\div}(n, m) = n$, $\ell_{\mathsf{if}}(b_1, b_2, n, m) = b_1 b_2 + \max(n-m, 0)$, and we do not label the remaining symbols. Then by labeling we get the (infinite) TRS $\mathsf{lab}(\mathcal{R})$ consisting of (1)–(8) together with the following rules, for all $i \geqslant j \geqslant 0$:

$$x \div_i y \to \mathsf{if}_j(y \geqslant \mathsf{s}(\mathsf{0}), x \geqslant y, x, y) \tag{13}$$
$$\mathsf{if}_i(\mathsf{false}, b, x, y) \to \mathsf{div\text{-}by\text{-}zero} \tag{14}$$
$$\mathsf{if}_i(\mathsf{true}, \mathsf{false}, x, y) \to \mathsf{0} \tag{15}$$
$$\mathsf{if}_{i+1}(\mathsf{true}, \mathsf{true}, x, y) \to \mathsf{id\text{-}inc}((x - y) \div_i y) \tag{16}$$

Termination of $\mathsf{lab}(\mathcal{R})$ is easily proved by LPO with precedence $\cdots \sqsupset \div_n \sqsupset \mathsf{if}_n \sqsupset \cdots \sqsupset \div_1 \sqsupset \mathsf{if}_1 \sqsupset \div_0 \sqsupset \mathsf{if}_0 \sqsupset \mathsf{id\text{-}inc} \sqsupset - \sqsupset \geqslant \sqsupset \mathsf{s} \sqsupset \mathsf{0} \sqsupset \mathsf{true} \sqsupset \mathsf{false}$. The result of semantic labeling is that if the algebra $\mathcal{A}$ is a model of $\mathcal{R}$ then termination of $\mathsf{lab}(\mathcal{R})$ implies termination of $\mathcal{R}$. However, it is impossible to give an interpretation $\mathsf{id\text{-}inc}_{\mathcal{A}}$ such that $\mathcal{A}$ is a model of $\mathcal{R}$, since there is a conflict between the rules (7) and (8).

One solution is to work with *quasi-models* where it is only required that the interpretation of each left-hand side of a rule is greater than or equal to the interpretation of the corresponding right-hand side. In [4] semantic labeling with quasi-models is extended to *predictive labeling* where $\mathcal{A}$ only has to be a quasi-model of the *usable rules*, the rules which define the function symbols that are needed to perform the labeling. In our example the usable rules are (1)–(6). And indeed $\mathcal{A}$ is a (quasi-) model of these rules. The problem when using quasi-models is the requirement that all interpretations have to be weakly monotone in all arguments. As $-_{\mathcal{A}}$ is not weakly monotone ($1 \geqslant 0$, but $3 -_{\mathcal{A}} 1 = 2 \not\geqslant 3 = 3 -_{\mathcal{A}} 0$) one cannot use the algebra $\mathcal{A}$ to prove termination of $\mathcal{R}$.

As a matter of fact, $\mathcal{R}$ is not terminating:

$$\begin{aligned}
\mathsf{s}(\mathsf{0}) \div \mathsf{id\text{-}inc}(\mathsf{0}) &\to \mathsf{if}(\mathsf{id\text{-}inc}(\mathsf{0}) \geqslant \mathsf{s}(\mathsf{0}), \mathsf{s}(\mathsf{0}) \geqslant \mathsf{id\text{-}inc}(\mathsf{0}), \mathsf{s}(\mathsf{0}), \mathsf{id\text{-}inc}(\mathsf{0})) \\
&\to^2 \mathsf{if}(\mathsf{s}(\mathsf{0}) \geqslant \mathsf{s}(\mathsf{0}), \mathsf{s}(\mathsf{0}) \geqslant \mathsf{s}(\mathsf{0}), \mathsf{s}(\mathsf{0}), \mathsf{id\text{-}inc}(\mathsf{0})) \\
&\to^4 \mathsf{if}(\mathsf{true}, \mathsf{true}, \mathsf{s}(\mathsf{0}), \mathsf{id\text{-}inc}(\mathsf{0}))
\end{aligned}$$

$$\begin{aligned}&\to\ \mathsf{id\text{-}inc}((\mathsf{s}(0) - \mathsf{id\text{-}inc}(0)) \div \mathsf{id\text{-}inc}(0))\\ &\to^2 (\mathsf{s}(0) - 0) \div \mathsf{id\text{-}inc}(0) \to \mathsf{s}(0) \div \mathsf{id\text{-}inc}(0) \to \cdots\end{aligned}$$

So there cannot be a version of predictive labeling with models and arbitrary interpretations. [2] Nevertheless, $\mathcal{R}$ is *innermost* terminating. Therefore we investigate whether one can use predictive labeling with models for innermost termination, where one can freely choose interpretations and where the algebra only has to be a model of the usable rules. As the previous results on predictive labeling only work for quasi-models, one cannot reuse them for innermost rewriting, e.g., Example 2.3 below shows that the main theorem of predictive labeling [4, Theorem 18] does not hold for innermost rewriting.

The remainder of this paper is organized as follows. In Section 2 we start the formal developments by recalling the basic definitions related to semantic labeling. We show that with respect to innermost termination semantic labeling is incomplete for both models and quasi-models and unsound for quasi-models. Soundness for models does hold and is shown in Section 3. By adapting the idea of predictive labeling to the innermost case we show that the model requirement is only needed for the usable rules induced by the labeling. The next contribution (Section 4) is the integration of an *argument filter*, i.e., a mapping from function symbols to sets of argument positions, to obtain even less usable rules than in [4] for innermost termination. This idea was already used in [3] where argument filters are employed to increase the power of term orders. In the context of semantic labeling, argument filters are used to express which arguments are ignored in interpretation and labeling functions. In Section 5 we return to termination. We show how to integrate argument filters with predictive labeling, resulting in a result that is strictly more powerful than the main theorem of [4]. Concluding remarks are given in Section 6.

2 Semantic Labeling for Innermost Termination

We assume that the reader is familiar with term rewriting [2]. Below we recall the basic definitions related to semantic labeling.

An algebra $\mathcal{A}$ over $\mathcal{F}$ is a pair $(A, \{f_{\mathcal{A}}\}_{f\in\mathcal{F}})$ consisting of a carrier A and, for every n-ary function symbol $f \in \mathcal{F}$, an interpretation function $f_{\mathcal{A}} \colon A^n \to A$. Given an assignment $\alpha \colon \mathcal{V} \to A$ we write $[\alpha]_{\mathcal{A}}(t)$ for the interpretation of the term t. An algebra $\mathcal{A}$ is a model of a rewrite system if $[\alpha]_{\mathcal{A}}(l) = [\alpha]_{\mathcal{A}}(r)$ for all rules $l \to r \in \mathcal{R}$ and all assignments α. If additionally, the carrier A is equipped with a well-founded order $>_A$ then $\mathcal{A}$ is a quasi-model if $[\alpha]_{\mathcal{A}}(l) \geqslant_A [\alpha]_{\mathcal{A}}(r)$ for all $l \to r \in \mathcal{R}$ and all assignments α.

For each function symbol f there also is a corresponding set $L_f \subseteq A$ of labels for f and if L_f is non-empty there also is a labeling function $\ell_f \colon A^n \to L_f$. The labeled signature $\mathcal{F}_{\mathsf{lab}}$ consists of n-ary function symbols f_a for every n-ary function symbol $f \in \mathcal{F}$ and label $a \in L_f$ together with all function symbols $f \in \mathcal{F}$ such that $L_f = \varnothing$. The labeling function ℓ_f determines the label of the root symbol f of a term $f(t_1, \ldots, t_n)$ based on the values of the arguments $t_1, \ldots, t_n$. For every assignment

[2] This answers a question raised in [4].

$\alpha\colon \mathcal{V} \to A$ the mapping $\mathsf{lab}_\alpha\colon \mathcal{T}(\mathcal{F},\mathcal{V}) \to \mathcal{T}(\mathcal{F}_{\mathsf{lab}},\mathcal{V})$ is inductively defined as follows:

$$\mathsf{lab}_\alpha(t) = \begin{cases} t & \text{if } t \text{ is a variable,} \\ f(\mathsf{lab}_\alpha(t_1),\dots,\mathsf{lab}_\alpha(t_n)) & \text{if } t = f(t_1,\dots,t_n) \text{ and } L_f = \varnothing, \\ f_a(\mathsf{lab}_\alpha(t_1),\dots,\mathsf{lab}_\alpha(t_n)) & \text{if } t = f(t_1,\dots,t_n) \text{ and } L_f \neq \varnothing \end{cases}$$

where a denotes the label $\ell_f([\alpha]_{\mathcal{A}}(t_1),\dots,[\alpha]_{\mathcal{A}}(t_n))$. The labeled TRS $\mathsf{lab}(\mathcal{R})$ over the signature $\mathcal{F}_{\mathsf{lab}}$ consists of the rules $\mathsf{lab}_\alpha(l) \to \mathsf{lab}_\alpha(r)$ for all $l \to r \in \mathcal{R}$ and $\alpha\colon \mathcal{V} \to A$. Moreover, if one uses quasi-models then one needs the set $\mathcal{D}\mathrm{ec} = \{f_a(x_1,\dots,x_n) \to f_b(x_1,\dots,x_n) \mid a,b \in L_f, a >_A b\}$ of decreasing rules. In this case every interpretation function $f_{\mathcal{A}}$ and every labeling function ℓ_f has to be weakly monotone, i.e., if $a \geqslant_A a'$ then $f_{\mathcal{A}}(a_1,\dots,a,\dots,a_n) \geqslant_A f_{\mathcal{A}}(a_1,\dots,a',\dots,a_n)$ and similarly for ℓ_f.

Zantema [9] obtained the following results for semantic labeling.

Lemma 2.1 *Let $\mathcal{R}$ be a TRS and $\mathcal{A}$ a non-empty algebra.*

(i) *If $\mathcal{A}$ is a model of $\mathcal{R}$ then $t \to_{\mathcal{R}} u$ implies $\mathsf{lab}_\alpha(t) \to_{\mathsf{lab}(\mathcal{R})} \mathsf{lab}_\alpha(u)$.*

(ii) *If $\mathcal{A}$ is a quasi-model of $\mathcal{R}$ then $t \to_{\mathcal{R}} u$ implies $\mathsf{lab}_\alpha(t) \to^+_{\mathsf{lab}(\mathcal{R})\cup\mathcal{D}\mathrm{ec}} \mathsf{lab}_\alpha(u)$.*

From Lemma 2.1 one obtains that $\mathcal{R}$ is terminating if and only if $\mathsf{lab}(\mathcal{R})\,(\cup\,\mathcal{D}\mathrm{ec})$ is terminating when $\mathcal{A}$ is a (quasi-)model of $\mathcal{R}$. Completeness is achieved by removing the labels of a possible infinite rewrite sequence of the labeled TRS. Soundness is proved by transforming a presupposed infinite rewrite sequence in $\mathcal{R}$ into an infinite rewrite sequence in $\mathsf{lab}(\mathcal{R})\,(\cup\,\mathcal{D}\mathrm{ec})$. This transformation is achieved by applying the labeling function $\mathsf{lab}_\alpha(\cdot)$ (for an arbitrary assignment α) to all terms in the infinite rewrite sequence of $\mathcal{R}$. Hence, semantic labeling is sound and complete for termination with respect to both models and quasi-models.

As first new contribution we show that semantic labeling is incomplete for innermost termination (Example 2.2) and that it is not even sound when using quasi-models (Example 2.3). We write $\xrightarrow{\mathsf{i}}_{\mathcal{R}}$ for the innermost rewrite relation of $\mathcal{R}$.

Example 2.2 Consider the TRS $\mathcal{R}$:

$$\begin{aligned} \mathsf{if}(\mathsf{true},x) &\to \mathsf{if}(\mathsf{test\text{-}ab}(x),x) & \mathsf{test\text{-}ab}(\mathsf{a}(x)) &\to \mathsf{test\text{-}b}(x) \\ \mathsf{a}(\mathsf{b}) &\to \mathsf{c} & \mathsf{test\text{-}b}(\mathsf{b}) &\to \mathsf{true} \end{aligned}$$

Note that $\mathcal{R}$ is innermost terminating. The reason is that $\mathsf{test\text{-}ab}(x)$ can only be evaluated to true if x is instantiated with $\mathsf{a}(\mathsf{b})$. But this is not allowed as $\mathsf{a}(\mathsf{b})$ is not in normal form. We choose the algebra $\mathcal{A}$ with carrier $A = \{0,1\}$, interpretations $\mathsf{if}_{\mathcal{A}}(x,y) = 0$, $\mathsf{b}_{\mathcal{A}} = \mathsf{c}_{\mathcal{A}} = \mathsf{true}_{\mathcal{A}} = 1$, $\mathsf{test\text{-}ab}_{\mathcal{A}}(x) = \mathsf{test\text{-}b}_{\mathcal{A}}(x) = \mathsf{a}_{\mathcal{A}}(x) = x$, and order $>_A = \varnothing$. Then $\mathcal{A}$ is a model (and thus also a quasi-model) of $\mathcal{R}$. Choosing $L_{\mathsf{a}} = A$, $\ell_{\mathsf{a}}(x) = x$, and $L_f = \varnothing$ for all other function symbols f we get the following labeled TRS $\mathsf{lab}(\mathcal{R})$:

$$\begin{aligned} \mathsf{if}(\mathsf{true},x) &\to \mathsf{if}(\mathsf{test\text{-}ab}(x),x) & \mathsf{test\text{-}ab}(\mathsf{a}_0(x)) &\to \mathsf{test\text{-}b}(x) & \mathsf{test\text{-}b}(\mathsf{b}) &\to \mathsf{true} \\ \mathsf{a}_1(\mathsf{b}) &\to \mathsf{c} & \mathsf{test\text{-}ab}(\mathsf{a}_1(x)) &\to \mathsf{test\text{-}b}(x) \end{aligned}$$

There are no decreasing rules. The following reduction shows that $\mathsf{lab}(\mathcal{R})$ is not innermost terminating:

$$\begin{aligned}\mathsf{if}(\mathsf{true},\mathsf{a}_0(\mathsf{b})) &\to_{\mathsf{lab}(\mathcal{R})} \mathsf{if}(\mathsf{test\text{-}ab}(\mathsf{a}_0(\mathsf{b})),\mathsf{a}_0(\mathsf{b})) \to_{\mathsf{lab}(\mathcal{R})} \mathsf{if}(\mathsf{test\text{-}b}(\mathsf{b}),\mathsf{a}_0(\mathsf{b}))\\ &\to_{\mathsf{lab}(\mathcal{R})} \mathsf{if}(\mathsf{true},\mathsf{a}_0(\mathsf{b})) \to_{\mathsf{lab}(\mathcal{R})} \cdots\end{aligned}$$

So semantic labeling is incomplete in the innermost case. The next example shows that semantic labeling with quasi-models is unsound in the innermost case.

Example 2.3 The TRS $\mathcal{R} = \{\mathsf{f}(\mathsf{a}(\mathsf{b})) \to \mathsf{f}(\mathsf{a}(\mathsf{b}))\}$ is obviously not innermost terminating. We choose the algebra $\mathcal{A}$ with carrier $A = \{0,1\}$, interpretations $\mathsf{b}_\mathcal{A} = \mathsf{f}_\mathcal{A}(x) = 1$, $\mathsf{a}_\mathcal{A}(x) = x$, and $>_A = >$, which is a (quasi-)model of $\mathcal{R}$. By taking $L_\mathsf{b} = L_\mathsf{f} = \varnothing$, $L_\mathsf{a} = A$, and $\ell_\mathsf{a}(x) = x$, we obtain the TRS $\mathsf{lab}(\mathcal{R}) \cup \mathcal{D}\mathrm{ec} = \{\mathsf{f}(\mathsf{a}_1(\mathsf{b})) \to \mathsf{f}(\mathsf{a}_1(\mathsf{b})),\ \mathsf{a}_1(x) \to \mathsf{a}_0(x)\}$. This TRS is innermost terminating because the second rule prohibits an innermost rewrite step with the first rule.

The previous example does not show that semantic labeling with models is unsound for innermost termination because there are no decreasing rules when using models. Indeed, in the next section we show the soundness of semantic labeling with models for innermost termination. Actually, we prove a stronger results by incorporating usable rules.

3 Predictive Labeling for Innermost Termination

Semantic labeling requires that the algebra is a model of all rules. This is in contrast to *predictive* labeling where the model condition only has to be satisfied for the *usable rules*, a concept introduced in [1]. We slightly modify the definition of usable rules by integrating the labeling. Here, $\mathcal{F}\mathrm{un}(t)$ denotes the set of all function symbols occurring in the term t.

Definition 3.1 Let $\mathcal{R}$ be a TRS and ℓ a labeling. We define the set of *usable symbols* $\mathcal{US}_\ell(t) \subseteq \mathcal{F}$ of a term t inductively. If $t \in \mathcal{V}$ then $\mathcal{US}_\ell(t) = \varnothing$. If $t = f(t_1,\ldots,t_n)$ then $\mathcal{US}_\ell(t)$ is the least set such that

(i) $\mathcal{US}_\ell(t_1) \cup \cdots \cup \mathcal{US}_\ell(t_n) \subseteq \mathcal{US}_\ell(t)$,

(ii) if $L_f \neq \varnothing$ then $\mathcal{F}\mathrm{un}(t_1) \cup \cdots \cup \mathcal{F}\mathrm{un}(t_n) \subseteq \mathcal{US}_\ell(t)$, and

(iii) if $l \to r \in \mathcal{R}$ and $\mathrm{root}(l) \in \mathcal{US}_\ell(t)$ then $\mathcal{F}\mathrm{un}(r) \subseteq \mathcal{US}_\ell(t)$.

The usable symbols of $\mathcal{R}$ are defined as

$$\mathcal{US}_\ell(\mathcal{R}) = \bigcup_{l\to r\in\mathcal{R}} \mathcal{US}_\ell(r)$$

and the *usable rules* of $\mathcal{R}$ are defined as

$$\mathcal{U}_\ell(\mathcal{R}) = \{l \to r \in \mathcal{R} \mid \mathrm{root}(l) \in \mathcal{US}_\ell(\mathcal{R})\}.$$

It can be shown that $\mathcal{US}_\ell(t) = \mathcal{G}_\ell(t)$ for the corresponding definition of $\mathcal{G}_\ell$ in [4, Definition 5]. However, there is a difference in the definition of $\mathcal{US}_\ell(\mathcal{R})$ and $\mathcal{G}_\ell(\mathcal{R})$

as in [4] both sides of a rule are considered, i.e., $\mathcal{G}_\ell(r)$ and $\mathcal{G}_\ell(l)$ are added for a rule $l \to r$. The difference is illustrated in the following example.

Example 3.2 Consider the TRS $\mathcal{R} = \{\mathsf{a} \to \mathsf{f}(\mathsf{g}(\mathsf{b})),\ \mathsf{g}(\mathsf{a}) \to \mathsf{c}\}$. Assuming $L_\mathsf{f} \neq \varnothing$ and $L_\mathsf{g} \neq \varnothing$, in [4] one obtains $\mathcal{G}_\ell(\mathcal{R}) = \{\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{f}, \mathsf{g}\}$ and thus both rules are usable. This is in contrast to Definition 3.1 where $\mathcal{US}_\ell(\mathcal{R}) = \{\mathsf{b}, \mathsf{c}, \mathsf{g}\}$ and hence $\mathcal{U}_\ell(\mathcal{R}) = \{\mathsf{g}(\mathsf{a}) \to \mathsf{c}\}$. The advantage of our definition is obvious: we get less usable rules. However, the property in [4] that one only needs interpretations for the symbols in $\mathcal{US}_\ell(\mathcal{R})$ is not valid anymore. To check the model condition for $\mathsf{g}(\mathsf{a}) \to \mathsf{c}$ and to label $\mathsf{g}(\mathsf{a})$ we need an interpretation $\mathsf{a}_\mathcal{A}$ for a.

From now on we assume a fixed TRS $\mathcal{R}$ and just write $\mathcal{US}_\ell$ instead of $\mathcal{US}_\ell(\mathcal{R})$ and $\mathcal{U}_\ell$ instead of $\mathcal{U}_\ell(\mathcal{R})$. Essentially, the aim of predictive (resp. semantic) labeling is to find a model for the usable (resp. all) rules and then try to prove innermost termination of $\mathsf{lab}(\mathcal{R})$ to ensure innermost termination of $\mathcal{R}$. As argued between Lemma 2.1 and Example 2.2, soundness of semantic labeling is proved by transforming an infinite reduction $t_1 \to_\mathcal{R} t_2 \to_\mathcal{R} \dots$ into an infinite reduction $\mathsf{lab}_\alpha(t_1) \to_{\mathsf{lab}(\mathcal{R})} \mathsf{lab}_\alpha(t_2) \to_{\mathsf{lab}(\mathcal{R})} \dots$ by using Lemma 2.1(i). However, in the predictive case this lemma does not hold if the algebra is not a model of all rules. To this end we consider a variant in which only reduction steps $t\sigma \xrightarrow{\mathsf{i}}_\mathcal{R} u$ are regarded where t satisfies $\mathcal{US}_\ell(t) \subseteq \mathcal{US}_\ell$ and where σ is a *normalized* substitution, i.e., where $\sigma(x)$ is in normal form for all $x \in \mathcal{V}$.

Lemma 3.3 *Let $\mathcal{A}$ be a model of $\mathcal{U}_\ell$, let $\mathcal{US}_\ell(t) \subseteq \mathcal{US}_\ell$, and let σ be a normalized substitution. If $t\sigma = C[l\sigma] \xrightarrow{\mathsf{i}}_\mathcal{R} C[r\sigma] = u$ is a reduction with rule $l \to r \in \mathcal{R}$ then*

(i) $\mathsf{lab}_\alpha(t\sigma) \xrightarrow{\mathsf{i}}_{\mathsf{lab}(\mathcal{R})} \mathsf{lab}_\alpha(u)$,

(ii) *there is a term t' such that $u = t'\sigma$ and $\mathcal{US}_\ell(t') \subseteq \mathcal{US}_\ell$, and*

(iii) *$\mathcal{F}\mathrm{un}(t) \subseteq \mathcal{US}_\ell$ implies both $\mathcal{F}\mathrm{un}(t') \subseteq \mathcal{US}_\ell$ and $[\alpha]_\mathcal{A}(t\sigma) = [\alpha]_\mathcal{A}(u)$.*

Note that Lemma 3.3(i) and (ii) will allow us to transform innermost reductions of $\mathcal{R}$ into infinite innermost reductions of $\mathsf{lab}(\mathcal{R})$. This is needed for the proof of the main theorem of this section (Theorem 3.4). Property (iii) is only needed to prove Lemma 3.3.

Proof. We perform structural induction on t. As σ is a normalized substitution t is not a variable, so let $t = f(t_1, \dots, t_n)$. We first consider a root reduction, i.e., $t\sigma = l\sigma \xrightarrow{\mathsf{i}}_\mathcal{R} r\sigma = u$. Let σ_lab be the substitution $\mathsf{lab}_\alpha \circ \sigma$ and let α_σ be the assignment $[\alpha]_\mathcal{A} \circ \sigma$. We have $\mathsf{lab}_\alpha(l\sigma) = \mathsf{lab}_{\alpha_\sigma}(l)\sigma_\mathsf{lab}$ and therefore obtain (i):

$$\mathsf{lab}_\alpha(t\sigma) = \mathsf{lab}_\alpha(l\sigma) = \mathsf{lab}_{\alpha_\sigma}(l)\sigma_\mathsf{lab} \xrightarrow{\mathsf{i}}_{\mathsf{lab}(\mathcal{R})} \mathsf{lab}_{\alpha_\sigma}(r)\sigma_\mathsf{lab} = \mathsf{lab}_\alpha(r\sigma) = \mathsf{lab}_\alpha(u).$$

Note that labeling does not introduce new redexes and hence the above reduction step is really an innermost step. The reason is that there are no decreasing rules as in Example 2.3. To obtain (ii) we choose $t' = r$. Then $u = t'\sigma$ is obviously satisfied and $\mathcal{US}_\ell(t') = \mathcal{US}_\ell(r) \subseteq \mathcal{US}_\ell$ follows by definition of $\mathcal{US}_\ell$. To prove (iii) let $\mathcal{F}\mathrm{un}(t) \subseteq \mathcal{US}_\ell$. Then $f \in \mathcal{US}_\ell$ and thus $l \to r \in \mathcal{U}_\ell$. Moreover, by the closure property in Definition 3.1(iii) we conclude $\mathcal{F}\mathrm{un}(r) \subseteq \mathcal{US}_\ell$. As the rule is usable we know that $\mathcal{A}$ is a model of this rule. Hence we can finish the root reduction case:

$$[\alpha]_{\mathcal{A}}(t\sigma) = [\alpha]_{\mathcal{A}}(l\sigma) = [\alpha_\sigma]_{\mathcal{A}}(l) = [\alpha_\sigma]_{\mathcal{A}}(r) = [\alpha]_{\mathcal{A}}(r\sigma) = [\alpha]_{\mathcal{A}}(u).$$

Now we consider a reduction below the root: $t_i\sigma = C'[l\sigma] \xrightarrow{\mathsf{i}}_{\mathcal{R}} C'[r\sigma] = u_i$ and $u = f(t_1\sigma, \ldots, u_i, \ldots, t_n\sigma)$. By Definition 3.1(i) we have $\mathcal{US}_\ell(t_i) \subseteq \mathcal{US}_\ell(t) \subseteq \mathcal{US}_\ell$. Hence, we can use the induction hypothesis for t_i. To prove (i) we consider two cases. First, if $L_f = \varnothing$ then

$$\begin{aligned}\mathsf{lab}_\alpha(t\sigma) = f(\mathsf{lab}_\alpha(t_1\sigma), \ldots, \mathsf{lab}_\alpha(t_i\sigma), \ldots, \mathsf{lab}_\alpha(t_n\sigma)) &\xrightarrow{\mathsf{i}}_{\mathsf{lab}(\mathcal{R})} \\ f(\mathsf{lab}_\alpha(t_1\sigma), \ldots, \mathsf{lab}_\alpha(u_i), \ldots, \mathsf{lab}_\alpha(t_n\sigma)) &= \mathsf{lab}_\alpha(u)\end{aligned}$$

directly proves (i). Otherwise, if $L_f \neq \varnothing$ then

$$\begin{aligned}\mathsf{lab}_\alpha(t\sigma) = f_a(\mathsf{lab}_\alpha(t_1\sigma), \ldots, \mathsf{lab}_\alpha(t_i\sigma), \ldots, \mathsf{lab}_\alpha(t_n\sigma)) &\xrightarrow{\mathsf{i}}_{\mathsf{lab}(\mathcal{R})} \\ f_a(\mathsf{lab}_\alpha(t_1\sigma), \ldots, \mathsf{lab}_\alpha(u_i), \ldots, \mathsf{lab}_\alpha(t_n\sigma))\end{aligned}$$

where $a = \ell_f([\alpha]_{\mathcal{A}}(t_1\sigma), \ldots, [\alpha]_{\mathcal{A}}(t_i\sigma), \ldots, [\alpha]_{\mathcal{A}}(t_n\sigma))$. It remains to show that $a = \ell_f([\alpha]_{\mathcal{A}}(t_1\sigma), \ldots, [\alpha]_{\mathcal{A}}(u_i), \ldots, [\alpha]_{\mathcal{A}}(t_n\sigma))$. To this end it suffices to prove $[\alpha]_{\mathcal{A}}(t_i\sigma) = [\alpha]_{\mathcal{A}}(u_i)$ which directly follows from the induction hypothesis (iii) since $\mathcal{F}\mathrm{un}(t_i) \subseteq \mathcal{US}_\ell(t_i) \subseteq \mathcal{US}_\ell$ by Definition 3.1(ii).

To show (ii) we first get a term t_i' with $t_i'\sigma = u_i$ and $\mathcal{US}_\ell(t_i') \subseteq \mathcal{US}_\ell$ by induction. We choose $t' = f(t_1, \ldots, t_i', \ldots, t_n)$ and directly obtain $t'\sigma = u$. To prove $\mathcal{US}_\ell(t') \subseteq \mathcal{US}_\ell$ we define $\mathcal{US}_\ell^k(t')$ to be like $\mathcal{US}_\ell(t')$ where we only apply closure (iii) in Definition 3.1 at most k times. Then it suffices to prove $\mathcal{US}_\ell^k(t') \subseteq \mathcal{US}_\ell$ for all $k \in \mathbb{N}$ which we do by an inner induction on k. We first consider closure (i). Here, we use $\mathcal{US}_\ell(t) \subseteq \mathcal{US}_\ell$ and Definition 3.1(i) to obtain $\mathcal{US}_\ell(t_1) \cup \cdots \cup \mathcal{US}_\ell(t_n) \subseteq \mathcal{US}_\ell$. Thus, $\mathcal{US}_\ell^k(t_1) \cup \cdots \cup \mathcal{US}_\ell^k(t_i') \cup \cdots \cup \mathcal{US}_\ell^k(t_n) \subseteq \mathcal{US}_\ell$ is also satisfied. For closure (ii) we only have to consider the case $L_f \neq \varnothing$. From $\mathcal{US}_\ell(t) \subseteq \mathcal{US}_\ell$ and Definition 3.1(ii) we conclude $\mathcal{F}\mathrm{un}(t_j) \subseteq \mathcal{US}_\ell$ for all $1 \leqslant j \leqslant n$. As $\mathcal{F}\mathrm{un}(t_i') \subseteq \mathcal{US}_\ell$ by induction hypothesis (iii), we are done. For closure (iii) let $f \in \mathcal{US}_\ell^k(t')$. If $f \in \mathcal{US}_\ell^{k-1}(t')$ then we only have to apply the inner induction hypothesis. Otherwise, there is a rule $l \to r$ with $\mathrm{root}(l) \in \mathcal{US}_\ell^{k-1}(t')$ and $f \in \mathcal{F}\mathrm{un}(r)$. From the inner induction hypothesis we obtain $\mathrm{root}(l) \in \mathcal{US}_\ell = \bigcup_{l' \to r'} \mathcal{US}_\ell(r')$. Thus, for some r' we have $\mathrm{root}(l) \in \mathcal{US}_\ell(r')$ and by Definition 3.1(iii) we know $f \in \mathcal{US}_\ell(r')$. But then $f \in \mathcal{US}_\ell$ as well.

To finally prove (iii) we assume $\mathcal{F}\mathrm{un}(t) \subseteq \mathcal{US}_\ell$. Then obviously $\mathcal{F}\mathrm{un}(t_i) \subseteq \mathcal{US}_\ell$. Thus, by induction hypothesis (iii) we know $\mathcal{F}\mathrm{un}(t_i') \subseteq \mathcal{US}_\ell$. So $\mathcal{F}\mathrm{un}(t') \subseteq \mathcal{US}_\ell$ is a consequence of $\mathcal{F}\mathrm{un}(t) \subseteq \mathcal{US}_\ell$. Moreover, we also obtain $[\alpha]_{\mathcal{A}}(t_i\sigma) = [\alpha]_{\mathcal{A}}(u_i)$ from the induction hypothesis (iii). Hence, we can finally prove (iii):

$$\begin{aligned}[\alpha]_{\mathcal{A}}(t\sigma) &= f_{\mathcal{A}}([\alpha]_{\mathcal{A}}(t_1\sigma), \ldots, [\alpha]_{\mathcal{A}}(t_i\sigma), \ldots, [\alpha]_{\mathcal{A}}(t_n\sigma)) \\ &= f_{\mathcal{A}}([\alpha]_{\mathcal{A}}(t_1\sigma), \ldots, [\alpha]_{\mathcal{A}}(u_i), \ldots, [\alpha]_{\mathcal{A}}(t_n\sigma)) = [\alpha]_{\mathcal{A}}(u).\end{aligned}$$

□

Theorem 3.4 *If $\mathcal{A}$ is a model of $\mathcal{U}_\ell$ then innermost termination of $\mathsf{lab}(\mathcal{R})$ implies innermost termination of $\mathcal{R}$.*

Proof. Suppose $\mathcal{R}$ is not innermost terminating. Then there is a minimal non-terminating term s which is not innermost terminating. By renaming the variables

of the rules used for the reductions we can assume that for every rewrite step in this infinite reduction the corresponding rule is instantiated by the same normalized substitution σ. By minimality of s, after a number of reductions there must be a root step, i.e., $s \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} l\sigma \xrightarrow{\mathsf{i}}_{\mathcal{R}} r\sigma$ for some rule $l \to r \in \mathcal{R}$ where $r\sigma$ is not innermost terminating. By definition of $\mathcal{US}_\ell$ we know $\mathcal{US}_\ell(r) \subseteq \mathcal{US}_\ell$. Hence, starting the infinite reduction with $r\sigma$ we can now simulate every reduction step with the corresponding labeled term $\mathsf{lab}_\alpha(r\sigma)$ using the labeled TRS $\mathsf{lab}(\mathcal{R})$. If $r\sigma \xrightarrow{\mathsf{i}}_{\mathcal{R}} r_1 \xrightarrow{\mathsf{i}}_{\mathcal{R}} r_2 \xrightarrow{\mathsf{i}}_{\mathcal{R}} \cdots$ then by Lemma 3.3(ii) we obtain terms $t_1, t_2, \ldots$ such that $r_i = t_i\sigma$ and $\mathcal{US}_\ell(t_i) \subseteq \mathcal{US}_\ell$. Using Lemma 3.3(i) we can finally prove that $\mathsf{lab}(\mathcal{R})$ is not innermost terminating:

$$\mathsf{lab}_\alpha(r\sigma) \xrightarrow{\mathsf{i}}_{\mathsf{lab}(\mathcal{R})} \mathsf{lab}_\alpha(r_1) = \mathsf{lab}_\alpha(t_1\sigma) \xrightarrow{\mathsf{i}}_{\mathsf{lab}(\mathcal{R})} \mathsf{lab}_\alpha(r_2) = \mathsf{lab}_\alpha(t_2\sigma) \xrightarrow{\mathsf{i}}_{\mathsf{lab}(\mathcal{R})} \ldots$$

□

With Theorem 3.4 it is now possible to prove innermost termination of the leading example with the specified algebra and the specified LPO.

4 Improved Labeling for Innermost Termination

We first modify the leading example to show a limitation of predictive labeling. Afterwards we present an improvement to overcome this limitation.

Example 4.1 We consider a reformulated version of the TRS in the leading example which uses an accumulator. Let $\mathcal{R}$ consist of the rules (1)–(8) together with the following rules:

$$\mathsf{quot}(x, y) \to \div(x, y, 0) \tag{17}$$
$$\div(x, y, z) \to \mathsf{if}(y \geqslant \mathsf{s}(0), x \geqslant y, x, y, z) \tag{18}$$
$$\mathsf{if}(\mathsf{false}, b, x, y, z) \to \mathsf{div\text{-}by\text{-}zero} \tag{19}$$
$$\mathsf{if}(\mathsf{true}, \mathsf{false}, x, y, z) \to z \tag{20}$$
$$\mathsf{if}(\mathsf{true}, \mathsf{true}, x, y, z) \to \div(x - y, y, \mathsf{id\text{-}inc}(z)) \tag{21}$$

The problem is that we cannot apply Theorem 3.4 with the given algebra $\mathcal{A}$; because $\mathsf{id\text{-}inc}$ now occurs below the labeled symbol $\div$, the problematic rules (7) and (8) are usable and $\mathcal{A}$ is not a model of these rules. However, the labeling function $\ell_\div$ ignores its third argument and thus, we do not need semantics for $\mathsf{id\text{-}inc}$ to compute the label for $\div$. Therefore, we would like to remove the $\mathsf{id\text{-}inc}$-rules from the set of usable rules. How this can be achieved is shown in the remainder of this section.

First, we need a notion to express which arguments of a function symbol should be ignored. To this end we use an *argument filter* which maps every symbol to the set of arguments that are not ignored. We further need a notion to express that an argument filter is suitable for an algebra and a labeling function. Argument filters were introduced in [1] and have been recently [3] used to reduce the usable rules in connection with the dependency pair method.

Definition 4.2 An *argument filter* is a mapping $\pi\colon \mathcal{F} \to 2^{\mathbb{N}}$ such that $\pi(f)$ is a subset of $\{1, \ldots, n\}$ for all $f \in \mathcal{F}$ with arity n. The application of an argument

filter π to a term t is denoted by $\pi(t)$ and defined as follows:

$$\pi(t) = \begin{cases} t & \text{if } t \text{ is a variable} \\ f(\pi(t_{i_1}), \ldots, \pi(t_{i_k})) & \text{if } t = f(t_1, \ldots, t_n) \text{ and } \pi(f) = \{i_1, \ldots, i_k\} \end{cases}$$

An algebra $\mathcal{A}$ is *π-conform* if $f_{\mathcal{A}}$ may depend on the i-th argument only if $i \in \pi(f)$. Similarly, a labeling function ℓ_f is *π-conform* if ℓ_f may depend on the i-th argument only if $i \in \pi(f)$.

From now on it is assumed that all algebras and labeling functions are π-conform. We refine Definition 3.1 to get less usable rules when regarding the argument filter.

Definition 4.3 Let $\mathcal{R}$ be a TRS, ℓ a labeling, and π an argument filter. We define the set $\mathcal{US}_{\ell,\pi}(t) \subseteq \mathcal{F}$ of *usable symbols with respect to* π of a term t inductively. If $t \in \mathcal{V}$ then $\mathcal{US}_{\ell,\pi}(t) = \varnothing$. If $t = f(t_1, \ldots, t_n)$ then $\mathcal{US}_{\ell,\pi}(t)$ is the least set such that

(i) $\mathcal{US}_{\ell,\pi}(t_1) \cup \cdots \cup \mathcal{US}_{\ell,\pi}(t_n) \subseteq \mathcal{US}_{\ell,\pi}(t)$,

(ii) if $L_f \neq \varnothing$ and $i \in \pi(f)$ then $\mathcal{F}\mathrm{un}(\pi(t_i)) \subseteq \mathcal{US}_{\ell,\pi}(t)$, and

(iii) if $l \to r \in \mathcal{R}$ and $\mathrm{root}(l) \in \mathcal{US}_{\ell,\pi}(t)$ then $\mathcal{F}\mathrm{un}(\pi(r)) \subseteq \mathcal{US}_{\ell,\pi}(t)$.

The usable symbols $\mathcal{US}_{\ell,\pi}(\mathcal{R})$ and the usable rules $\mathcal{U}_{\ell,\pi}(\mathcal{R})$ with respect to π are defined as

$$\mathcal{US}_{\ell,\pi}(\mathcal{R}) = \bigcup_{l \to r \in \mathcal{R}} \mathcal{US}_{\ell,\pi}(r) \quad \text{and} \quad \mathcal{U}_{\ell,\pi}(\mathcal{R}) = \{l \to r \in \mathcal{R} \mid \mathrm{root}(l) \in \mathcal{US}_{\ell,\pi}(\mathcal{R})\}.$$

As before, we assume a fixed TRS $\mathcal{R}$ and therefore just write $\mathcal{US}_{\ell,\pi}$ and $\mathcal{U}_{\ell,\pi}$ for $\mathcal{US}_{\ell,\pi}(\mathcal{R})$ and $\mathcal{U}_{\ell,\pi}(\mathcal{R})$. We now show how innermost termination of the TRS in Example 4.1 can be proved if one only has to find a model for the usable rules with respect to π.

Example 4.4 We choose $\pi(\div) = \{1, 2\}$ and $\pi(\mathsf{if}) = \{1, 2, 3, 4\}$ in Example 4.1. Then $\mathcal{A}$ and the labeling functions are π-conform and the usable rules are (1)–(6) as in the leading example. We obtain a similar labeled TRS and termination is proved by a similar LPO. One only has to extend the precedence for the new symbol quot by demanding $\mathsf{quot} \sqsupset \div_i$ for all $i \in \mathbb{N}$.

The only missing step is to extend the results of Lemma 3.3 and Theorem 3.4 to the refined version of usable rules in Definition 4.3.

Lemma 4.5 *Let $\mathcal{A}$ be a model of $\mathcal{U}_{\ell,\pi}$, let $\mathcal{US}_{\ell,\pi}(t) \subseteq \mathcal{US}_{\ell,\pi}$, and let σ be a normalized substation such that $t\sigma = C[l\sigma] \xrightarrow{\mathsf{i}}_{\mathcal{R}} C[r\sigma] = u$ for a rule $l \to r \in \mathcal{R}$. Then the following properties are satisfied:*

(i) $\mathsf{lab}_\alpha(t\sigma) \xrightarrow{\mathsf{i}}_{\mathsf{lab}(\mathcal{R})} \mathsf{lab}_\alpha(u)$,

(ii) *there is a term t' such that $u = t'\sigma$ and $\mathcal{US}_{\ell,\pi}(t') \subseteq \mathcal{US}_{\ell,\pi}$, and*

(iii) *$\mathcal{F}\mathrm{un}(\pi(t)) \subseteq \mathcal{US}_{\ell,\pi}$ implies both $\mathcal{F}\mathrm{un}(\pi(t')) \subseteq \mathcal{US}_{\ell,\pi}$ and $[\alpha]_{\mathcal{A}}(t\sigma) = [\alpha]_{\mathcal{A}}(u)$.*

Proof. The proof is completely similar to the proof of Lemma 3.3 where one replaces $\mathcal{US}_\ell$ by $\mathcal{US}_{\ell,\pi}$, $\mathcal{F}\mathrm{un}(t)$ by $\mathcal{F}\mathrm{un}(\pi(t))$, and $\mathcal{U}_\ell$ by $\mathcal{U}_{\ell,\pi}$. Therefore, we only give the three additional cases which arise when considering reductions below the root.

First, to prove (i) one has to show $\ell_f([\alpha]_{\mathcal{A}}(t_1\sigma),\ldots,[\alpha]_{\mathcal{A}}(t_i\sigma),\ldots,[\alpha]_{\mathcal{A}}(t_n\sigma)) = \ell_f([\alpha]_{\mathcal{A}}(t_1\sigma),\ldots,[\alpha]_{\mathcal{A}}(u_i),\ldots,[\alpha]_{\mathcal{A}}(t_n\sigma))$ as before. If $i \in \pi(f)$ then one can conclude $\mathcal{F}\mathrm{un}(\pi(t_i)) \subseteq \mathcal{US}_{\ell,\pi}$ and proceed as in the proof of Lemma 3.3. Otherwise, $i \notin \pi(f)$ and thus, the equality is valid as ℓ_f ignores its i-th argument. Second, to prove (ii) one has to show $\mathcal{US}_{\ell,\pi}(t') \subseteq \mathcal{US}_{\ell,\pi}$ by looking at the closure properties (i) and (ii) of Definition 4.3. When considering (ii) one cannot conclude $\mathcal{F}\mathrm{un}(\pi(t_i)) \subseteq \mathcal{US}_{\ell,\pi}$ if $i \notin \pi(f)$. However, in that case $\mathcal{F}\mathrm{un}(\pi(t'_i)) \subseteq \mathcal{US}_{\ell,\pi}$ is not required to satisfy (ii). Finally, to prove (iii) one gets the additional case $i \notin \pi(f)$. Then $\mathcal{F}\mathrm{un}(\pi(t')) = \mathcal{F}\mathrm{un}(\pi(t)) \subseteq \mathcal{US}_{\ell,\pi}$ as $\pi(t) = \pi(t')$. Moreover, using the fact that $f_{\mathcal{A}}$ ignores its i-th argument immediately yields $[\alpha]_{\mathcal{A}}(t\sigma) = [\alpha]_{\mathcal{A}}(u)$. □

We are now ready to present the result about improved predictive labeling where under the assumption of π-conformity one only has to find a model for the usable rules with respect to π. As demonstrated in Example 4.1 and Example 4.4 this clearly extends Theorem 3.4.

Theorem 4.6 *Let π be an argument filter. If $\mathcal{A}$ is a model of $\mathcal{U}_{\ell,\pi}$ and if both $\mathcal{A}$ and all labeling functions are π-conform then innermost termination of $\mathsf{lab}(\mathcal{R})$ implies innermost termination of $\mathcal{R}$.*

Proof. Just replace Lemma 3.3 by Lemma 4.5 in the proof of Theorem 3.4. □

A possible extension of Theorem 4.6 is to redefine Definition 4.3 such that $\mathcal{US}_{\ell,\pi}(t_i) \subseteq \mathcal{US}_{\ell,\pi}(t)$ is only required if $i \in \pi(f)$. However the following example shows that this extension is unsound.

Example 4.7 Consider the TRS $\{\mathsf{f}(\mathsf{g}(\mathsf{a})) \to \mathsf{f}(\mathsf{g}(\mathsf{b})),\ \mathsf{b} \to \mathsf{a}\}$. We choose the algebra with carrier $A = \{0,1\}$ and interpretations $\mathsf{f}_{\mathcal{A}}(x) = \mathsf{g}_{\mathcal{A}}(x) = \mathsf{a}_{\mathcal{A}} = 0$ and $\mathsf{b}_{\mathcal{A}} = 1$. For the labeling we use $L_{\mathsf{f}} = L_{\mathsf{a}} = L_{\mathsf{b}} = \varnothing$, $L_{\mathsf{g}} = A$, and $\ell_{\mathsf{g}}(x) = x$. Then both the algebra and the labeling functions are π-conform for the argument filter π defined by $\pi(\mathsf{f}) = \pi(\mathsf{a}) = \pi(\mathsf{b}) = \varnothing$ and $\pi(\mathsf{g}) = \{1\}$. However, using the alternative definition of $\mathcal{US}_{\ell,\pi}(t)$ we get $\mathcal{US}_{\ell,\pi} = \varnothing$ and hence, $\mathcal{A}$ is a model for the usable rules. Thus, the extension cannot be sound as the labeled TRS $\{\mathsf{f}(\mathsf{g}_0(\mathsf{a})) \to \mathsf{f}(\mathsf{g}_1(\mathsf{b})),\ \mathsf{b} \to \mathsf{a}\}$ is terminating while $\mathcal{R}$ is not innermost terminating.

In the next section we combine the idea of usable rules with respect to an argument filter with predictive labeling for full rewriting.

5 Improved Predictive Labeling for Termination

Example 5.1 We consider the TRS $\mathcal{R}$ consisting of (7), (8), and

$$\mathsf{nonZero}(\mathsf{0}) \to \mathsf{false} \quad (22) \qquad \mathsf{random}(x) \to \mathsf{rand}(x,\mathsf{0}) \quad (26)$$
$$\mathsf{nonZero}(\mathsf{s}(x)) \to \mathsf{true} \quad (23) \qquad \mathsf{rand}(x,y) \to \mathsf{if}(\mathsf{nonZero}(x),x,y) \quad (27)$$
$$\mathsf{p}(\mathsf{s}(x)) \to x \quad (24) \qquad \mathsf{if}(\mathsf{false},x,y) \to y \quad (28)$$
$$\mathsf{p}(\mathsf{0}) \to \mathsf{0} \quad (25) \qquad \mathsf{if}(\mathsf{true},x,y) \to \mathsf{rand}(\mathsf{p}(x),\mathsf{id\text{-}inc}(y)) \quad (29)$$

Here, $\mathsf{random}(x)$ generates a random number between 0 and x. We use the algebra $\mathcal{A}$ with carrier $\mathbb{N}$ and natural interpretations $\mathsf{p}_{\mathcal{A}}(x) = \max(x-1,0)$, $\mathsf{s}_{\mathcal{A}}(x) = x+1$,

$\mathsf{0}_\mathcal{A} = \mathsf{false}_\mathcal{A} = 0$, $\mathsf{true}_\mathcal{A} = 1$, and $\mathsf{nonZero}_\mathcal{A}(x) = 0$ if $x = 0$, and 1 otherwise. If one takes the standard order $>$ on $\mathbb{N}$ then $\mathcal{A}$ is a $\sqcup$-algebra [4] and a quasi-model for rules (22)–(25). Moreover, for the labeling with $L_{\mathsf{rand}} = L_{\mathsf{if}} = \mathbb{N}$, $\ell_{\mathsf{rand}}(n, m) = n$, $\ell_{\mathsf{if}}(b, n, m) = b + \max(n - 1, 0)$, and $L_f = \varnothing$ for all other function symbols, both $\mathcal{A}$ and the labeling functions are monotone. Hence, one can apply predictive labeling of [4] to obtain the terminating TRS $\mathsf{lab}(\mathcal{R}) \cup \mathcal{D}\mathrm{ec}$ which can be proved by an LPO. Unfortunately, the usable rules according to [4] include the critical rules (7) and (8) as argument filters are not considered when computing the usable rules. Thus, the requirements of predictive labeling [4, Theorem 18] are not satisfied. Therefore, we now extend the results of [4] and show how to integrate argument filters in the termination case where we only obtain the usable rules (22)–(25). Indeed, all requirements of our new Theorem 5.10 are satisfied and we can conclude termination of $\mathcal{R}$ by proving termination of $\mathsf{lab}(\mathcal{R}) \cup \mathcal{D}\mathrm{ec}$.

As in [4], for improved predictive labeling in the termination case we do not allow arbitrary algebras but one has to use a so-called $\sqcup$-*algebra* ([4, Definition 8]).

Definition 5.2 Let $\mathcal{A}$ be an algebra and let $>_A$ be a well-founded order on the carrier A. We say that $(\mathcal{A}, >_A)$ is a $\sqcup$-*algebra* if for all finite subsets $X \subseteq A$ there exists a least upper bound $\bigsqcup X$ of X in A.

In the remainder of this section we assume that $\mathcal{R}$ is a *finitely branching* TRS, π an argument filter, and $(\mathcal{A}, >_A)$ a $\sqcup$-algebra such that all interpretations $f_\mathcal{A}$ and all labeling functions ℓ_f are weakly monotone and π-conform, and $\mathcal{U}_{\ell,\pi} \subseteq \geqslant_A$.

As in the previous sections we cannot directly achieve the result of Lemma 2.1(ii) to transform infinite $\mathcal{R}$ reductions into infinite reductions of $\mathsf{lab}(\mathcal{R}) \cup \mathcal{D}\mathrm{ec}$ since $\mathcal{A}$ is not a quasi-model of all rules in $\mathcal{R}$. Therefore, we introduce an alternative interpretation function $[\alpha]^*_\mathcal{A}(\cdot)$ for all terminating terms ($\mathcal{SN}$) similar to [4, Definition 9]. However, one has to perform a minor modification due to the difference between $\mathcal{US}_\ell$ and $\mathcal{G}_\ell$, cf. Example 3.2.

Definition 5.3 Let $t \in \mathcal{SN}$ and α an assignment. We define the interpretation $[\alpha]^*_\mathcal{A}(t)$ inductively as follows where $t' = f_\mathcal{A}([\alpha]^*_\mathcal{A}(t_1), \ldots, [\alpha]^*_\mathcal{A}(t_n))$:

$$[\alpha]^*_\mathcal{A}(t) = \begin{cases} \alpha(x) & \text{if } t \text{ is a variable,} \\ t' & \text{if } t = f(t_1, \ldots, t_n) \text{ and } f \in \mathcal{US}_{\ell,\pi}, \\ \bigsqcup \{[\alpha]^*_\mathcal{A}(u) \mid t \to_\mathcal{R} u\} \cup \{t'\} & \text{if } t = f(t_1, \ldots, t_n) \text{ and } f \notin \mathcal{US}_{\ell,\pi}. \end{cases}$$

Note that the recursion in the definition of $[\alpha]^*_\mathcal{A}(\cdot)$ terminates because the union of $\to_\mathcal{R}$ and the proper superterm relation $\rhd$ is a well-founded relation on $\mathcal{SN}$. Further note that the operation $\bigsqcup$ is applied only to finite sets as $\mathcal{R}$ is assumed to be finitely branching.

The induced labeling function [4, Definition 10] can be defined for terminating and for minimal non-terminating terms ($\mathcal{T}^\infty$) but not for arbitrary terms in $\mathcal{T}(\mathcal{F}, \mathcal{V})$.

Definition 5.4 Let $t \in \mathcal{SN} \cup \mathcal{T}^\infty$ and α an assignment. We define the labeled

term $\mathsf{lab}^*_\alpha(t)$ inductively as follows:

$$\mathsf{lab}^*_\alpha(t) = \begin{cases} t & \text{if } t \text{ is a variable,} \\ f(\mathsf{lab}^*_\alpha(t_1), \ldots, \mathsf{lab}^*_\alpha(t_n)) & \text{if } t = f(t_1, \ldots, t_n) \text{ and } L_f = \varnothing, \\ f_a(\mathsf{lab}^*_\alpha(t_1), \ldots, \mathsf{lab}^*_\alpha(t_n)) & \text{if } t = f(t_1, \ldots, t_n) \text{ and } L_f \neq \varnothing \end{cases}$$

where $a = \ell_f([\alpha]^*_{\mathcal{A}}(t_1), \ldots, [\alpha]^*_{\mathcal{A}}(t_n))$.

The following lemma compares the predicted semantics of an instantiated terminating term to the original semantics of the uninstantiated term, in which the substitution becomes part of the assignment.

Definition 5.5 Given an assignment α and a substitution σ such that $\sigma(x) \in \mathcal{SN}$ for all variables x, the assignment α^*_σ is defined as $[\alpha]^*_{\mathcal{A}} \circ \sigma$ and the substitution $\sigma_{\mathsf{lab}^*_\alpha}$ as $\mathsf{lab}^*_\alpha \circ \sigma$.

Lemma 5.6 *If $t\sigma \in \mathcal{SN}$ then $[\alpha]^*_{\mathcal{A}}(t\sigma) \geqslant_A [\alpha^*_\sigma]_{\mathcal{A}}(t)$. If in addition $\mathcal{F}\mathrm{un}(\pi(t)) \subseteq \mathcal{US}_{\ell,\pi}$ then $[\alpha]^*_{\mathcal{A}}(t\sigma) = [\alpha^*_\sigma]_{\mathcal{A}}(t)$.*

Proof. We use structural induction on t. If $t \in \mathcal{V}$ then $[\alpha]^*_{\mathcal{A}}(t\sigma) = ([\alpha]^*_{\mathcal{A}} \circ \sigma)(t) = [\alpha^*_\sigma]_{\mathcal{A}}(t)$. Suppose $t = f(t_1, \ldots, t_n)$. We distinguish two cases.

(i) If $f \in \mathcal{US}_{\ell,\pi}$ then

$$\begin{aligned}[\alpha]^*_{\mathcal{A}}(t\sigma) = f_{\mathcal{A}}([\alpha]^*_{\mathcal{A}}(t_1\sigma), \ldots, [\alpha]^*_{\mathcal{A}}(t_n\sigma)) &\geqslant_A \\ f_{\mathcal{A}}([\alpha^*_\sigma]_{\mathcal{A}}(t_1), \ldots, [\alpha^*_\sigma]_{\mathcal{A}}(t_n)) &= [\alpha^*_\sigma]_{\mathcal{A}}(t)\end{aligned}$$

where the inequality follows from the induction hypothesis (note that $t_i\sigma \in \mathcal{SN}$ for all $i = 1, \ldots, n$) and the weak monotonicity of $f_{\mathcal{A}}$. If $\mathcal{F}\mathrm{un}(\pi(t)) \subseteq \mathcal{US}_{\ell,\pi}$ and $i \in \pi(f)$ then $\mathcal{F}\mathrm{un}(\pi(t_i)) \subseteq \mathcal{US}_{\ell,\pi}$ and thus $[\alpha]^*_{\mathcal{A}}(t_i\sigma) = [\alpha^*_\sigma]_{\mathcal{A}}(t_i)$ according to the induction hypothesis. Since $f_{\mathcal{A}}$ is π-conform, the inequality is turned into an equality.

(ii) If $f \notin \mathcal{US}_{\ell,\pi}$ then

$$\begin{aligned}[\alpha]^*_{\mathcal{A}}(t\sigma) &= \bigsqcup\{\cdots\} \cup \{f_{\mathcal{A}}([\alpha]^*_{\mathcal{A}}(t_1\sigma), \ldots, [\alpha]^*_{\mathcal{A}}(t_n\sigma))\} \\ &\geqslant_A f_{\mathcal{A}}([\alpha]^*_{\mathcal{A}}(t_1\sigma), \ldots, [\alpha]^*_{\mathcal{A}}(t_n\sigma)) \geqslant_A [\alpha^*_\sigma]_{\mathcal{A}}(t)\end{aligned}$$

again using weak monotonicity of $f_{\mathcal{A}}$ and the induction hypothesis. As in this case $\mathcal{F}\mathrm{un}(\pi(t)) \not\subseteq \mathcal{US}_{\ell,\pi}$, we have already proved the second part of the lemma. □

The next lemma does the same for labeled terms. Since the label of a function symbol only depends on the semantics of its arguments, we can only deal with terminating and minimal non-terminating terms.

Lemma 5.7 *If $t\sigma \in \mathcal{SN} \cup \mathcal{T}^\infty$ then $\mathsf{lab}^*_\alpha(t\sigma) \to^*_{\mathcal{D}\mathrm{ec}} \mathsf{lab}_{\alpha^*_\sigma}(t)\sigma_{\mathsf{lab}^*_\alpha}$. If in addition $\mathcal{US}_{\ell,\pi}(t) \subseteq \mathcal{US}_{\ell,\pi}$ then $\mathsf{lab}^*_\alpha(t\sigma) = \mathsf{lab}_{\alpha^*_\sigma}(t)\sigma_{\mathsf{lab}^*_\alpha}$.*

Proof. We use structural induction on t. If t is a variable then $\mathsf{lab}^*_\alpha(t\sigma) = t\sigma_{\mathsf{lab}^*_\alpha} = \mathsf{lab}_{\alpha^*_\sigma}(t)\sigma_{\mathsf{lab}^*_\alpha}$. Otherwise $t = f(t_1, \ldots, t_n)$. Note that $t_1, \ldots, t_n \in \mathcal{SN}$. The induction hypothesis yields $\mathsf{lab}^*_\alpha(t_i\sigma) \to^*_{\mathcal{D}\mathrm{ec}} \mathsf{lab}_{\alpha^*_\sigma}(t_i)\sigma_{\mathsf{lab}^*_\alpha}$ for all $i = 1, \ldots, n$. Moreover,

whenever $\mathcal{US}_{\ell,\pi}(t) \subseteq \mathcal{US}_{\ell,\pi}$ then by Definition 4.3(i) $\mathcal{US}_{\ell,\pi}(t_i) \subseteq \mathcal{US}_{\ell,\pi}$ for every i and thus $\mathsf{lab}^*_\alpha(t_i\sigma) = \mathsf{lab}_{\alpha^*_\sigma}(t_i)\sigma_{\mathsf{lab}^*_\alpha}$ by the induction hypothesis. We distinguish three cases.

(i) If $L_f = \varnothing$ then

$$\begin{aligned}\mathsf{lab}^*_\alpha(t\sigma) &= f(\mathsf{lab}^*_\alpha(t_1\sigma), \ldots, \mathsf{lab}^*_\alpha(t_n\sigma))\\ &\to^*_{\mathcal{D}\mathrm{ec}} f(\mathsf{lab}_{\alpha^*_\sigma}(t_1)\sigma_{\mathsf{lab}^*_\alpha}, \ldots, \mathsf{lab}_{\alpha^*_\sigma}(t_n)\sigma_{\mathsf{lab}^*_\alpha})\\ &= f(\mathsf{lab}_{\alpha^*_\sigma}(t_1), \ldots, \mathsf{lab}_{\alpha^*_\sigma}(t_n))\sigma_{\mathsf{lab}^*_\alpha} = \mathsf{lab}_{\alpha^*_\sigma}(f(t_1, \ldots, t_n))\sigma_{\mathsf{lab}^*_\alpha}.\end{aligned}$$

Of course, if $\mathcal{US}_{\ell,\pi}(t) \subseteq \mathcal{US}_{\ell,\pi}$ then there are no reduction steps.

(ii) If $L_f \neq \varnothing$ and $\mathcal{US}_{\ell,\pi}(t) \not\subseteq \mathcal{US}_{\ell,\pi}$ then $\mathsf{lab}^*_\alpha(t\sigma) = f_a(\mathsf{lab}^*_\alpha(t_1\sigma), \ldots, \mathsf{lab}^*_\alpha(t_n\sigma)) \to^*_{\mathcal{D}\mathrm{ec}} f_a(\mathsf{lab}_{\alpha^*_\sigma}(t_1)\sigma_{\mathsf{lab}^*_\alpha}, \ldots, \mathsf{lab}_{\alpha^*_\sigma}(t_n)\sigma_{\mathsf{lab}^*_\alpha})$ and

$$\begin{aligned}\mathsf{lab}_{\alpha^*_\sigma}(t)\sigma_{\mathsf{lab}^*_\alpha} &= f_b(\mathsf{lab}_{\alpha^*_\sigma}(t_1), \ldots, \mathsf{lab}_{\alpha^*_\sigma}(t_n))\sigma_{\mathsf{lab}^*_\alpha}\\ &= f_b(\mathsf{lab}_{\alpha^*_\sigma}(t_1)\sigma_{\mathsf{lab}^*_\alpha}, \ldots, \mathsf{lab}_{\alpha^*_\sigma}(t_n)\sigma_{\mathsf{lab}^*_\alpha})\end{aligned}$$

with $a = \ell_f([\alpha]^*_{\mathcal{A}}(t_1\sigma), \ldots, [\alpha]^*_{\mathcal{A}}(t_n\sigma))$ and $b = \ell_f([\alpha^*_\sigma]_{\mathcal{A}}(t_1), \ldots, [\alpha^*_\sigma]_{\mathcal{A}}(t_n))$. Lemma 5.6 yields $[\alpha]^*_{\mathcal{A}}(t_i\sigma) \geqslant_A [\alpha^*_\sigma]_{\mathcal{A}}(t_i)$ for all $i = 1, \ldots, n$. Because the labeling function ℓ_f is weakly monotone in all its coordinates, $a \geqslant_A b$. If $a >_A b$ then $\mathcal{D}\mathrm{ec}$ contains the rewrite rule $f_a(x_1, \ldots, x_n) \to f_b(x_1, \ldots, x_n)$ and thus (also if $a = b$) $f_a(\mathsf{lab}_{\alpha^*_\sigma}(t_1)\sigma_{\mathsf{lab}^*_\alpha}, \ldots, \mathsf{lab}_{\alpha^*_\sigma}(t_n)\sigma_{\mathsf{lab}^*_\alpha}) \to^*_{\mathcal{D}\mathrm{ec}} \mathsf{lab}_{\alpha^*_\sigma}(t)\sigma_{\mathsf{lab}^*_\alpha}$. We conclude that $\mathsf{lab}^*_\alpha(t\sigma) \to^*_{\mathcal{D}\mathrm{ec}} \mathsf{lab}_{\alpha^*_\sigma}(t)\sigma_{\mathsf{lab}^*_\alpha}$.

(iii) If $L_f \neq \varnothing$ and $\mathcal{US}_{\ell,\pi}(t) \subseteq \mathcal{US}_{\ell,\pi}$ then $\mathsf{lab}^*_\alpha(t\sigma) = f_a(\mathsf{lab}^*_\alpha(t_1\sigma), \ldots, \mathsf{lab}^*_\alpha(t_n\sigma)) = f_a(\mathsf{lab}_{\alpha^*_\sigma}(t_1)\sigma_{\mathsf{lab}^*_\alpha}, \ldots, \mathsf{lab}_{\alpha^*_\sigma}(t_n)\sigma_{\mathsf{lab}^*_\alpha}) = f_a(\mathsf{lab}_{\alpha^*_\sigma}(t_1), \ldots, \mathsf{lab}_{\alpha^*_\sigma}(t_n))\sigma_{\mathsf{lab}^*_\alpha}$ and

$$\mathsf{lab}_{\alpha^*_\sigma}(t)\sigma_{\mathsf{lab}^*_\alpha} = f_b(\mathsf{lab}_{\alpha^*_\sigma}(t_1), \ldots, \mathsf{lab}_{\alpha^*_\sigma}(t_n))\sigma_{\mathsf{lab}^*_\alpha}$$

with $a = \ell_f([\alpha]^*_{\mathcal{A}}(t_1\sigma), \ldots, [\alpha]^*_{\mathcal{A}}(t_n\sigma))$ and $b = \ell_f([\alpha^*_\sigma]_{\mathcal{A}}(t_1), \ldots, [\alpha^*_\sigma]_{\mathcal{A}}(t_n))$. We need to show that $a = b$. Because ℓ_f is π-conform, this amounts to showing $[\alpha]^*_{\mathcal{A}}(t_i\sigma) = [\alpha^*_\sigma]_{\mathcal{A}}(t_i)$ for $i \in \pi(f)$. If we can show that $\mathcal{F}\mathrm{un}(\pi(t_i)) \subseteq \mathcal{US}_{\ell,\pi}$, this follows from Lemma 5.6. (Note that $t_i\sigma \in \mathcal{SN}$ as $t\sigma \in \mathcal{SN} \cup \mathcal{T}^\infty$.) But this can directly be concluded from $\mathcal{F}\mathrm{un}(\pi(t_i)) \subseteq \mathcal{US}_{\ell,\pi}(t) \subseteq \mathcal{US}_{\ell,\pi}$ by closure property (ii) of Definition 4.3.

□

We further need to know that the predicted semantics decreases when rewriting.

Lemma 5.8 *Let $t, u \in \mathcal{SN}$. If $t \to_{\mathcal{R}} u$ then $[\alpha]^*_{\mathcal{A}}(t) \geqslant_A [\alpha]^*_{\mathcal{A}}(u)$.*

Proof. We perform structural induction on t. Obviously, t is not a variable, so let $t = f(t_1, \ldots, t_n)$. If $f \notin \mathcal{US}_{\ell,\pi}$ then $[\alpha]^*_{\mathcal{A}}(t) = \bigsqcup\{[\alpha]^*_{\mathcal{A}}(v) \mid t \to_{\mathcal{R}} v\} \cup \{\cdots\} \geqslant_A [\alpha]^*_{\mathcal{A}}(u)$ since $[\alpha]^*_{\mathcal{A}}(u) \in \{[\alpha]^*_{\mathcal{A}}(v) \mid t \to_{\mathcal{R}} v\}$. Thus, for the remaining proof we may assume $f \in \mathcal{US}_{\ell,\pi}$. We consider two cases.

(i) First we consider a root reduction $t = l\sigma \to_{\mathcal{R}} r\sigma = u$. As $\mathrm{root}(l) = \mathrm{root}(t) = f \in \mathcal{US}_{\ell,\pi}$ we know $l \to r \in \mathcal{U}_{\ell,\pi}$ and $\mathcal{F}\mathrm{un}(\pi(r)) \subseteq \mathcal{US}_{\ell,\pi}$ due to closure property (iii) in Definition 4.3. From the assumption $\mathcal{U}_{\ell,\pi} \subseteq \geqslant_A$ we infer $l \geqslant_A r$. Using Lemma 5.6 we obtain $[\alpha]^*_{\mathcal{A}}(t) = [\alpha]^*_{\mathcal{A}}(l\sigma) \geqslant_A [\alpha^*_\sigma]_{\mathcal{A}}(l) \geqslant_A [\alpha^*_\sigma]_{\mathcal{A}}(r) = [\alpha]^*_{\mathcal{A}}(r\sigma) = [\alpha]^*_{\mathcal{A}}(u)$.

(ii) Next assume a reduction $t \to_{\mathcal{R}} f(t_1, \ldots, u_i, \ldots, t_n) = u$ below the root where $t_i \to_{\mathcal{R}} u_i$. The induction hypothesis yields $[\alpha]^*_{\mathcal{A}}(t_i) \geqslant_A [\alpha]^*_{\mathcal{A}}(u_i)$ and thus

$$\begin{aligned}[\alpha]^*_{\mathcal{A}}(t) &= f_{\mathcal{A}}([\alpha]^*_{\mathcal{A}}(t_1), \ldots, [\alpha]_{\mathcal{A}}(t_i), \ldots, [\alpha]_{\mathcal{A}}(t_n)) \\ &\geqslant_A f_{\mathcal{A}}([\alpha]^*_{\mathcal{A}}(t_1), \ldots, [\alpha]_{\mathcal{A}}(u_i), \ldots, [\alpha]_{\mathcal{A}}(t_n)) = [\alpha]^*_{\mathcal{A}}(u)\end{aligned}$$

by weak monotonicity of $f_{\mathcal{A}}$.

□

We are now ready for the key lemma, which states that rewrite steps between terminating and minimal non-terminating terms can be labeled.

Lemma 5.9 *Let $t, u \in \mathcal{SN} \cup \mathcal{T}^\infty$. If $t \to_{\mathcal{R}} u$ then $\mathsf{lab}^*_\alpha(t) \to^+_{\mathsf{lab}(\mathcal{R}) \cup \mathcal{D}\mathrm{ec}} \mathsf{lab}^*_\alpha(u)$.*

Proof. We use structural induction on t. Obviously $t = f(t_1, \ldots, t_n)$. For a root reduction $t = l\sigma \to_{\mathcal{R}} r\sigma = u$ we infer $\mathsf{lab}^*_\alpha(t) = \mathsf{lab}^*_\alpha(l\sigma) \to^*_{\mathcal{D}\mathrm{ec}} \mathsf{lab}_{\alpha^*_\sigma}(l)\sigma_{\mathsf{lab}^*_\alpha} \to_{\mathsf{lab}(\mathcal{R})} \mathsf{lab}_{\alpha^*_\sigma}(r)\sigma_{\mathsf{lab}^*_\alpha} = \mathsf{lab}^*_\alpha(r\sigma) = \mathsf{lab}^*_\alpha(u)$ by Lemma 5.7. Otherwise, we have $u = f(t_1, \ldots, u_i, \ldots, t_n)$ with $t_i \to_{\mathcal{R}} u_i$. We obtain $\mathsf{lab}^*_\alpha(t_i) \to^+_{\mathsf{lab}(\mathcal{R}) \cup \mathcal{D}\mathrm{ec}} \mathsf{lab}^*_\alpha(u_i)$ from the induction hypothesis. We distinguish two cases.

(i) If $L_f = \varnothing$ then $\mathsf{lab}^*_\alpha(t) = f(\mathsf{lab}^*_\alpha(t_1), \ldots, \mathsf{lab}^*_\alpha(t_i), \ldots, \mathsf{lab}^*_\alpha(t_n)) \to^+_{\mathsf{lab}(\mathcal{R}) \cup \mathcal{D}\mathrm{ec}} f(\mathsf{lab}^*_\alpha(t_1), \ldots, \mathsf{lab}^*_\alpha(u_i), \ldots, \mathsf{lab}^*_\alpha(t_n)) = \mathsf{lab}^*_\alpha(u)$.

(ii) If $L_f \neq \varnothing$ then

$$\begin{aligned}\mathsf{lab}^*_\alpha(t) &= f_a(\mathsf{lab}^*_\alpha(t_1), \ldots, \mathsf{lab}^*_\alpha(t_i), \ldots, \mathsf{lab}^*_\alpha(t_n)) \\ &\to^+_{\mathsf{lab}(\mathcal{R}) \cup \mathcal{D}\mathrm{ec}} f_a(\mathsf{lab}^*_\alpha(t_1), \ldots, \mathsf{lab}^*_\alpha(u_i), \ldots, \mathsf{lab}^*_\alpha(t_n))\end{aligned}$$

with $a = \ell_f([\alpha]^*_{\mathcal{A}}(t_1), \ldots, [\alpha]^*_{\mathcal{A}}(t_i), \ldots, [\alpha]^*_{\mathcal{A}}(t_n))$ and

$$\mathsf{lab}^*_\alpha(u) = f_b(\mathsf{lab}^*_\alpha(t_1), \ldots, \mathsf{lab}^*_\alpha(u_i), \ldots, \mathsf{lab}^*_\alpha(t_n))$$

with $b = \ell_f([\alpha]^*_{\mathcal{A}}(t_1), \ldots, [\alpha]^*_{\mathcal{A}}(u_i), \ldots, [\alpha]^*_{\mathcal{A}}(t_n))$. Because $t_i \in \mathcal{SN}$ we can use Lemma 5.8 to obtain $[\alpha]^*_{\mathcal{A}}(t_i) \geqslant_A [\alpha]^*_{\mathcal{A}}(u_i)$. Hence, $a \geqslant_A b$ by weak monotonicity of ℓ_f and thus $f_a(\mathsf{lab}^*_\alpha(t_1), \ldots, \mathsf{lab}^*_\alpha(u_i), \ldots, \mathsf{lab}^*_\alpha(t_n)) \to^*_{\mathcal{D}\mathrm{ec}} \mathsf{lab}^*_\alpha(u)$.

□

We now have all the ingredients to prove the soundness of improved predictive labeling for termination.

Theorem 5.10 *Let $\mathcal{R}$ be a TRS, let π be an argument filter, and let $(\mathcal{A}, >_A)$ be a $\sqcup$-algebra such that $\mathcal{A}$ is a quasi-model of $\mathcal{U}_{\ell,\pi}$ and all interpretation and labeling functions are weakly monotone and π-conform. If $\mathsf{lab}(\mathcal{R}) \cup \mathcal{D}\mathrm{ec}$ is terminating then so is $\mathcal{R}$.*

Proof. Note that for every term $t \in \mathcal{T}^\infty$ there exist a rewrite rule $l \to r \in \mathcal{R}$, a substitution σ, and a subterm u of r such that $t \xrightarrow{\geq\epsilon}{}^* l\sigma \xrightarrow{\epsilon} r\sigma \trianglerighteq u\sigma$ and $l\sigma, u\sigma \in \mathcal{T}^\infty$. Let α be an arbitrary assignment. We will apply lab^*_α to the terms in the above sequence. From Lemma 5.9 we obtain $\mathsf{lab}^*_\alpha(t) \to^*_{\mathsf{lab}(\mathcal{R}) \cup \mathcal{D}\mathrm{ec}} \mathsf{lab}^*_\alpha(l\sigma)$. Since $r\sigma$ need not be an element of $\mathcal{T}^\infty$, we cannot apply Lemma 5.9 to the step $l\sigma \xrightarrow{\epsilon} r\sigma$. Instead we use Lemma 5.7 to obtain $\mathsf{lab}^*_\alpha(l\sigma) \to^*_{\mathcal{D}\mathrm{ec}} \mathsf{lab}_{\alpha^*_\sigma}(l)\sigma_{\mathsf{lab}^*_\alpha}$. Since $\mathsf{lab}_{\alpha^*_\sigma}(l) \to$

$\mathsf{lab}_{\alpha^*_\sigma}(r) \in \mathsf{lab}(\mathcal{R})$, $\mathsf{lab}_{\alpha^*_\sigma}(l)\sigma_{\mathsf{lab}^*_\alpha} \to_{\mathsf{lab}(\mathcal{R})} \mathsf{lab}_{\alpha^*_\sigma}(r)\sigma_{\mathsf{lab}^*_\alpha}$. Because u is a subterm of r, $\mathsf{lab}_{\alpha^*_\sigma}(r)\sigma_{\mathsf{lab}^*_\alpha} \trianglerighteq \mathsf{lab}_{\alpha^*_\sigma}(u)\sigma_{\mathsf{lab}^*_\alpha}$. From closure property (i) of Definition 4.3 we infer $\mathcal{US}_{\ell,\pi}(u) \subseteq \mathcal{US}_{\ell,\pi}(r)$. Since r is a right-hand side of a rewrite rule of $\mathcal{R}$, $\mathcal{US}_{\ell,\pi}(r) \subseteq \mathcal{US}_{\ell,\pi}$. Hence $\mathcal{US}_{\ell,\pi}(u) \subseteq \mathcal{US}_{\ell,\pi}$. Lemma 5.7 now yields $\mathsf{lab}_{\alpha^*_\sigma}(u)\sigma_{\mathsf{lab}^*_\alpha} = \mathsf{lab}^*_\alpha(u\sigma)$. Putting everything together, we obtain $\mathsf{lab}^*_\alpha(t) \to^+_{\mathsf{lab}(\mathcal{R})\cup\mathcal{D}\mathrm{ec}} \cdot \trianglerighteq \mathsf{lab}^*_\alpha(u\sigma)$. Now suppose that $\mathcal{R}$ is non-terminating. Then $\mathcal{T}^\infty$ is non-empty and thus there is an infinite sequence $t_1 \xrightarrow{>\epsilon}{}^* \cdot \xrightarrow{\epsilon} \cdot \trianglerighteq t_2 \xrightarrow{>\epsilon}{}^* \cdot \xrightarrow{\epsilon} \cdot \trianglerighteq \cdots$ By the above argument, this sequence is transformed into $\mathsf{lab}^*_\alpha(t_1) \to^+_{\mathsf{lab}(\mathcal{R})\cup\mathcal{D}\mathrm{ec}} \cdot \trianglerighteq \mathsf{lab}^*_\alpha(t_2) \to^+_{\mathsf{lab}(\mathcal{R})\cup\mathcal{D}\mathrm{ec}} \cdot \trianglerighteq \cdots$ By introducing appropriate contexts, the latter sequence gives rise to an infinite reduction in $\mathsf{lab}(\mathcal{R}) \cup \mathcal{D}\mathrm{ec}$, contradicting the assumption that $\mathcal{R}$ is terminating. □

6 Conclusion

We have analyzed how the powerful technique of semantic labeling can be used to prove innermost termination. It turned out that semantic labeling can be used for models but not for quasi-models. We extended our results to predictive labeling such that one only has to find a model for the usable as opposed to all rules. This approach was further improved by incorporating argument filters. The latter extension was finally integrated with predictive labeling for termination.

The results presented in this paper should be implemented in order to test their effectiveness and combined with dependency pairs [1] to increase their applicability. Semantic [9] and predictive [4] labeling with infinite (quasi-)models for termination have been implemented in the automatic termination prover TPA [6]. The underlying theory is worked out in [8] and [7]. In the latter paper predictive labeling for termination is combined with dependency pairs. Modifying these results to cover innermost termination is straightforward. Incorporating argument filterings will increase the search space but otherwise poses no challenge. We anticipate that the power of TPA and other termination provers will be increased by the results of this paper.

References

[1] Arts, T. and J. Giesl, *Termination of term rewriting using dependency pairs*, TCS **236** (2000), pp. 133–178.

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

[3] Giesl, J., R. Thiemann, P. Schneider-Kamp and S. Falke, *Mechanizing and improving dependency pairs*, JAR **37** (2006), pp. 155–203.

[4] Hirokawa, N. and A. Middeldorp, *Predictive labeling*, in: *Proc. 17th RTA*, LNCS **4098**, 2006, pp. 313–327.

[5] Hirokawa, N. and A. Middeldorp, *Tyrolean termination tool: Techniques and features*, I&C **205** (2007), pp. 474–511.

[6] Koprowski, A., *TPA: Termination proved automatically*, in: *Proc. 17th RTA*, LNCS, 2006, pp. 275–266.

[7] Koprowski, A. and A. Middeldorp, *Predictive labeling with dependency pairs using SAT*, in: *Proc. 21st CADE*, LNAI **4603**, 2007, pp. 410–425.

[8] Koprowski, A. and H. Zantema, *Recursive path ordering for infinite labelled rewrite systems*, in: *Proc. 3rd IJCAR*, LNAI **4130**, 2006, pp. 332–346.

[9] Zantema, H., *Termination of term rewriting by semantic labelling*, FI **24** (1995), pp. 89–105.

Decidability of Innermost Termination and Context-Sensitive Termination for Semi-Constructor Term Rewriting Systems

Keita Uchiyama [1] Masahiko Sakai [2] Toshiki Sakabe [3]

Graduate School of Information Science
Nagoya University
Furo-cho, Chikusa-ku, Nagoya, 464-8603, Japan

Abstract

Yi and Sakai [9] showed that the termination problem is a decidable property for the class of semi-constructor term rewriting systems, which is a superclass of the class of right ground term rewriting systems. The decidability was shown by the fact that every non-terminating TRS in the class has a loop. In this paper we modify the proof of [9] to show that both innermost termination and μ-termination are decidable properties for the class of semi-constructor TRSs.

Keywords: Context-Sensitive Termination, Dependency Pair, Innermost Termination

1 Introduction

Termination is one of the central properties of term rewriting systems (TRSs for short), where we say a TRS terminates if it does not admit any infinite reduction sequence. Since termination is undecidable in general, several decidable classes have been studied [4,5,6,8,9]. The class of semi-constructor TRSs is one of them [9], where a TRS is in this class if for every right hand sides of rules its all subterms having defined symbol at root position are ground.

Innermost reduction, the strategy which rewrites innermost redexes, is used for call-by-value computation. are indicated by specifying arguments of function symbols. Some non-terminating TRSs are terminating by context-sensitive reduction without loss of computational ability. The termination property with respect to innermost (resp. context-sensitive) reduction is called innermost (resp. context-sensitive) termination. Since innermost termination and context-sensitive termina-

[1] Email: uchiyama@sakabe.i.is.nagoya-u.ac.jp
[2] Email: sakai@is.nagoya-u.ac.jp
[3] Email: sakabe@is.nagoya-u.ac.jp

tion are also undecidable in general, methods for proving these terminations have been studied [1,2].

In this paper, we prove that innermost termination and context-sensitive termination for semi-constructor TRSs are decidable properties. The proof is done by using notions of dependency pairs [1,2].

2 Preliminaries

We assume the reader is familiar with the standard definitions of term rewriting systems [3] and here we just review the main notations used in this paper.

A *signature* $\mathcal{F}$ is a set of function symbols, where every $f \in \mathcal{F}$ is associated with a non-negative integer by an arity function: $\mathrm{arity}\colon \mathcal{F} \to \mathbb{N}(= \{0, 1, 2, \ldots\})$. The set of all *terms* built from a signature $\mathcal{F}$ and a countable infinite set $\mathcal{V}$ of *variables* such that $\mathcal{F} \cap \mathcal{V} = \emptyset$, is represented by $\mathcal{T}(\mathcal{F}, \mathcal{V})$. The set of *ground terms* is $\mathcal{T}(\mathcal{F}, \emptyset)$ ($\mathcal{T}(\mathcal{F})$ for short). The set of variables occurring in a term t is denoted by $\mathrm{Var}(t)$.

The set of all *positions* in a term t is denoted by $\mathcal{P}\mathrm{os}(t)$ and ε represents the root position. $\mathcal{P}\mathrm{os}(t)$ is: $\mathcal{P}\mathrm{os}(t) = \{\varepsilon\}$ if $t \in \mathcal{V}$, and $\mathcal{P}\mathrm{os}(t) = \{\varepsilon\} \cup \{iu \mid 1 \leq i \leq n, u \in \mathcal{P}\mathrm{os}(t_i)\}$ if $t = f(t_1, \ldots, t_n)$. Let C be a *context* with a hole $\Box$. We write $C[t]$ for the term obtained from C by replacing $\Box$ with a term t. Especially, we specify an occurrence position p of a hole by $C[\,]_p$. We say t is a subterm of s if $s = C[t]$ for some context C. We denote the *subterm relation* by $\trianglelefteq$, that is, $t \trianglelefteq s$ if t is a subterm of s, and $t \triangleleft s$ if $t \trianglelefteq s$ and $t \neq s$. The *root symbol* of a term t is denoted by $\mathrm{root}(t)$.

A *substitution* θ is a mapping from $\mathcal{V}$ to $\mathcal{T}(\mathcal{F}, \mathcal{V})$ such that the set $\mathrm{Dom}(\theta) = \{x \in \mathcal{V} \mid \theta(x) \neq x\}$ is finite. We usually identify a substitution θ with the set $\{x \mapsto \theta(x) \mid x \in \mathrm{Dom}(\theta)\}$ of variable bindings. In the following, we write $t\theta$ instead of $\theta(t)$.

A *rewrite rule* $l \to r$ is a directed equation which satisfies $l \notin \mathcal{V}$ and $\mathrm{Var}(r) \subseteq \mathrm{Var}(l)$. A *term rewriting system* TRS is a finite set of rewrite rules. A *redex* is a term $l\theta$ for a rule $l \to r$ and a substitution θ. A term containing no redex is called a *normal form*. A substitution θ is *normal* if $x\theta$ is in normal forms for every x. The *reduction relation* $\xrightarrow{p}_R \subseteq \mathcal{T}(\mathcal{F}, \mathcal{V}) \times \mathcal{T}(\mathcal{F}, \mathcal{V})$ associated with a TRS R and position p is defined as follows: $s \xrightarrow{p}_R t$ if there exist a rewrite rule $l \to r \in R$, a substitution θ, and a context $C[\,]_p$ such that $s = C[l\theta]_p$ and $t = C[r\theta]_p$, we say that s is reduced to t by contracting redex $l\theta$. We sometimes write $\to_R$ for $\xrightarrow{p}_R$ by omitting p.

A redex is *innermost*, if its all proper subterms are in normal forms. If s is reduced to t by contracting an innermost redex, then $s \to_R t$ is said to be an *innermost reduction* denoted by $s \xrightarrow[in]{}_R t$.

Proposition 2.1 *For a TRS R, if there is a reduction $s \xrightarrow[in]{}_R t$, then $C[s] \xrightarrow[in]{}_R C[t]$ for any context C.*

A mapping $\mu : \mathcal{F} \to \mathcal{P}(\mathbb{N})$ is a *replacement map* (or $\mathcal{F}$-map) if $\mu(f) \subseteq \{1, \ldots, \mathrm{arity}(f)\}$. The set of *$\mu$-replacing positions* $\mathcal{P}\mathrm{os}_\mu(t)$ of a term t is: $\mathcal{P}\mathrm{os}_\mu(t) = \{\varepsilon\}$, if $t \in \mathcal{V}$ and $\mathcal{P}\mathrm{os}_\mu(t) = \{\varepsilon\} \cup \{iu \mid 1 \leq i \leq n, i \in \mu(f), u \in \mathcal{P}\mathrm{os}_\mu(t_i)\}$, if $t = f(t_1, \ldots, t_n)$. The set of all *μ-replacing variables* of t is $\mathrm{Var}_\mu(t) = \{x \in \mathrm{Var}(t) \mid \exists p \in \mathcal{P}\mathrm{os}_\mu(t), \exists C, C[x]_p = t\}$. The *$\mu$-replacing subterm relation* $\trianglelefteq_\mu$ is given by $s \trianglelefteq_\mu t$

if there is $p \in \mathcal{P}os_\mu(t)$ such that $t = C[s]_p$. A context $C[\ ]_p$ is *μ-replacing* denoted by $C_\mu[\ ]_p$ if $p \in \mathcal{P}os_\mu(C[\ ]_p)$. A *context-sensitive rewriting system* is a TRS with an $\mathcal{F}$-map. If $s \xrightarrow{p} t$ and $p \in \mathcal{P}os_\mu(s)$, then $s \xrightarrow{p} t$ is said to be *μ-reduction* denoted by $s \overrightarrow{\ }_{\mu R} t$.

Let $\to$ be a binary relation on terms, the transitive closure of $\to$ is denoted by $\to^+$. The transitive and reflexive closure of $\to$ is denoted by $\to^*$. If $s \to^* t$, then we say that there is a *$\to$-sequence* starting from s to t or t is *$\to$-reachable from s*. We write $s \to^k t$ if t is $\to$-reachable from s with k steps. A term t *terminates* with respect to $\to$ if there exists no infinite $\to$-sequence starting from t.

Example 2.2 Let $R_1 = \{g(x) \to h(x),\ h(d) \to g(c),\ c \to d\}$ and $\mu_1(g) = \mu_1(h) = \emptyset$ [10]. A μ_1-reduction sequence starting from $g(d)$ is $g(d) \overrightarrow{\ }_{\mu_1 R_1} h(d) \overrightarrow{\ }_{\mu_1 R_1} g(c)$. We can not reduce $g(c)$ to $g(d)$ because c is not μ_1-replacing subterm of $g(c)$.

Proposition 2.3 *For a TRS R and $\mathcal{F}$-map μ, if there is a reduction $s \overrightarrow{\ }_{\mu R} t$, then $C_\mu[s] \overrightarrow{\ }_{\mu R} C_\mu[t]$ for any μ-replacing context C_μ.*

For a TRS R (and $\mathcal{F}$-map μ), we say that R terminates (resp. innermost terminates, μ-terminates) if every term terminates with respect to $\to_R$ (resp. $\overrightarrow{\ }_{in R}$, $\overrightarrow{\ }_{\mu R}$).

For a TRS R, a function symbol $f \in \mathcal{F}$ is *defined* if $f = \mathrm{root}(l)$ for some rule $l \to r \in R$. The set of all defined symbols of R is denoted by $D_R = \{\mathrm{root}(l) \mid l \to r \in R\}$. A term t has a *defined root symbol* if $\mathrm{root}(t) \in D_R$.

Let R be a TRS over a signature $\mathcal{F}$. The signature $\mathcal{F}^\sharp$ denotes the union of $\mathcal{F}$ and $D_R^\sharp = \{f^\sharp \mid f \in D_R\}$ where $\mathcal{F} \cap D_R^\sharp = \emptyset$ and $f^\sharp$ has the same arity as f. We call these fresh symbols *dependency pair symbols*. We define a notation $t^\sharp$ by $t^\sharp = f^\sharp(t_1, \ldots, t_n)$ if $t = f(t_1, \ldots, t_n)$ and $f \in D_R$, $t^\sharp = x$ if $t = x$ and $x \in \mathcal{V}$. If $l \to r \in R$ and u is a subterm of r with a defined root symbol and $u \not\trianglelefteq l$, then the rewrite rule $l^\sharp \to u^\sharp$ is called a *dependency pair* of R. The set of all dependency pairs of R is denoted by $\mathrm{DP}(R)$.

Definition 2.4 [Semi-Constructor TRS] A term $t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ is a *semi-constructor* term if every term s such that $s \trianglelefteq t$ and $\mathrm{root}(s) \in D_R$ is ground. A TRS R is a semi-constructor system if r is a semi-constructor term for every rule $l \to r \in R$.

Example 2.5 The TRS $R_2 = \{a \to g(f(a)),\ f(g(x)) \to h(f(a), x)\}$ is a semi-constructor TRS.

A TRS R is called *right-ground* if for every $l \to r \in R$, r is ground.

Proposition 2.6 *The following statements hold:*

(i) *Right-ground TRSs are semi-constructor systems.*

(ii) *For a semi-constructor TRS R, all rules in DP(R) are right-ground.*

3 Decidability of Innermost Termination for Semi-Constructor TRSs

Decidability of termination for semi-constructor TRSs is proved based on the observation that there exists an infinite reduction sequence having a loop if it is not

terminating [9]. In this section, we prove the decidability of innermost termination in similar way.

Definition 3.1 [loop] A *reduction sequence* loops if it contains $t \to^+ C[t]$ for some context C, and *head-loops* if containing $t \to^+ t$.

Proposition 3.2 *If there exists an innermost sequence that loops, then there exists an infinite innermost sequence.*

Definition 3.3 [Innermost DP-chain] For a TRS R, a sequence of the elements of $\mathrm{DP}(R)$ $s_1^\sharp \to t_1^\sharp, s_2^\sharp \to t_2^\sharp, \ldots$ is an *innermost dependency chain* if there exist substitutions $\tau_1, \tau_2, \ldots$ such that $s_i^\sharp\tau_i$ is in normal forms and $t_i^\sharp\tau_i \xrightarrow[in]{}_R^* s_{i+1}^\sharp\tau_{i+1}$ holds for every i.

Theorem 3.4 ([2]) *For a TRS R, R does not innermost terminate if and only if there exists an infinite innermost dependency chain.*

Let $\mathcal{T}_\infty^{\overrightarrow{in}_R}$ denote the set of all $\trianglerighteq$-*minimal non-terminating terms* for $\xrightarrow[in]{}_R$, here "$\trianglerighteq$-minimal" is used in the sense that all its proper subterms terminate.

Definition 3.5 [$\mathcal{C}$-min] For a TRS R, let $\mathcal{C} \subseteq \mathrm{DP}(R)$. An infinite reduction sequence in $R \cup \mathcal{C}$ in the form $t_1^\sharp \xrightarrow[in]{}_{R\cup\mathcal{C}} t_2^\sharp \xrightarrow[in]{}_{R\cup\mathcal{C}} t_3^\sharp \xrightarrow[in]{}_{R\cup\mathcal{C}} \cdots$ with $t_i \in \mathcal{T}_\infty^{\overrightarrow{in}_R}$ for all $i \geq 1$ is called a $\mathcal{C}$-*min innermost reduction sequence*. We use $\mathcal{C}_{min}^{in}(t^\sharp)$ to denote the set of all $\mathcal{C}$-min innermost reduction sequence starting from $t^\sharp$.

Proposition 3.6 ([2]) *Given a TRS R, the following statements hold:*

(i) *If there exists an infinite innermost dependency chain, then $\mathcal{C}_{min}^{in}(t^\sharp) \neq \emptyset$ for some $\mathcal{C} \subseteq DP(R)$ and $t \in \mathcal{T}_\infty^{\overrightarrow{in}_R}$.*

(ii) *For any sequence in $\mathcal{C}_{min}^{in}(t^\sharp)$, reduction by rules of R takes place below the root while reduction by rules of $\mathcal{C}$ takes place at the root.*

(iii) *For any sequence in $\mathcal{C}_{min}^{in}(t^\sharp)$, there is at least one rule in $\mathcal{C}$ which is applied infinitely often.*

Lemma 3.7 ([2]) *For two terms s and s', $s^\sharp \xrightarrow[in]{}_{R\cup\mathcal{C}}^* s'^\sharp$ implies $s \xrightarrow[in]{}_R^* C[s']$ for some context C.*

Proof. We use induction on the number n of reduction steps in $s^\sharp \xrightarrow[in]{}_{R\cup\mathcal{C}}^n s'^\sharp$. In the case that $n = 0$, it holds with $C = \Box$. Let $n \geq 1$. Then we have $s^\sharp \xrightarrow[in]{}_{R\cup\mathcal{C}}^{n-1} s''^\sharp \xrightarrow[in]{}_{R\cup\mathcal{C}} s'^\sharp$ for some $s''^\sharp$. By the induction hypothesis, $s \xrightarrow[in]{}_R^* C[s'']$.
• Consider the case that $s''^\sharp \xrightarrow[in]{}_R s'^\sharp$. Since $s'' \xrightarrow[in]{}_R s'$, we have $C[s''] \xrightarrow[in]{}_R C[s']$ by Proposition 2.1. Hence $s \xrightarrow[in]{}_R^* C[s']$.
• Consider the case that $s''^\sharp \xrightarrow[in]{}_{\mathcal{C}} s'^\sharp$. Since s'' is a normal form with respect to $\to_R$, we have $s'' \xrightarrow[in]{}_R C'[s']$ by the definition of dependency pairs. $C[s''] \xrightarrow[in]{}_R C[C'[s']]$, by Proposition 2.1. Hence $s \xrightarrow[in]{}_R^* C[C'[s']]$. □

Lemma 3.8 *For a semi-constructor TRS R, the following statements are equivalent:*

(i) *R does not innermost terminate.*

(ii) *There exists* $l^\sharp \to u^\sharp \in DP(R)$ *such that* sq *head-loops for some* $\mathcal{C} \subseteq DP(R)$ *and* $sq \in \mathcal{C}^{in}_{min}(u^\sharp)$.

Proof. ((ii) ⇒ (i)) : It is obvious from Lemma 3.7, and Proposition 3.2. ((i) ⇒ (ii)) : By Theorem 3.4 there exists an infinite innermost dependency chain. By Proposition 3.6(i), there exists a sequence $sq \in \mathcal{C}^{in}_{min}(t^\sharp)$. By Proposition 3.6(ii),(iii), there exists some rule $l^\sharp \to u^\sharp \in \mathcal{C}$ which is applied at root position in sq infinitely often. By Proposition 2.6(ii), $u^\sharp$ is ground. Thus sq contains a subsequence $u^\sharp \xrightarrow[in]{}{}^*_{R \cup DP(R)} \cdot \to_{\{l^\sharp \to u^\sharp\}} u^\sharp$, which head-loops. □

Theorem 3.9 *Innermost termination of semi-constructor TRSs is decidable.*

Proof. The decision procedure for the innermost termination of a semi-constructor TRS R is as follows: consider all terms $u_1, u_2, \ldots, u_n$ corresponding to the right-hand sides of $\mathrm{DP}(R) = \{l_i^\sharp \to u_i^\sharp \mid 1 \le i \le n\}$, and simultaneously generate all innermost reduction sequences with respect to R starting from $u_1, u_2, \ldots, u_n$. It halts if it enumerates all reachable terms exhaustively or it detects a looping reduction sequence $u_i \xrightarrow[in]{}{}^+_R C[u_i]$ for some i.

Suppose R does not innermost-terminate. By Lemma 3.8, 3.7, we have a looping reduction sequence $u_i \xrightarrow[in]{}{}^+_R C[u_i]$ for some i and C, which we eventually detect. If R innermost terminates, then the execution of the reduction sequence generation eventually stops since it is finitely branching. Moreover it does not detect a looping sequence, otherwise it contradicts to Proposition 3.2. Thus the procedure decides innermost termination of R in finitely many steps. □

4 Decidability of Context-Sensitive Termination for Semi-Constructor TRSs

The proof of decidability for innermost termination is straightforward as above. However, the one for context-sensitive termination is not so straightforward because of the existence of dependency pair with a variable in the right-hand side.

Definition 4.1 [μ-Loop] Let $\to$ be a relation on terms and μ be an $\mathcal{F}$-map. A *reduction sequence μ-loops* if it contains $t' \to^+ C_\mu[t']$ for some context C_μ.

Example 4.2 Let $\mu_2(f) = \mu_2(h) = \{1\}$ and $\mu_2(g) = \emptyset$. For R_2 (in Example 2.5), the μ_2-reduction sequence $f(a) \xrightarrow[\mu_2]{}_{R_2} f(g(f(a))) \xrightarrow[\mu_2]{}_{R_2} h(f(a), f(a)) \xrightarrow[\mu_2]{}_{R_2} \cdots$ is μ_2-looping.

Proposition 4.3 *If there exists a μ-looping μ-reduction sequence, then there exists an infinite μ-reduction sequence.*

Definition 4.4 [Context-Sensitive Dependency Pairs [1]] Let R be a TRS and μ be an $\mathcal{F}$-map. We define $\mathrm{DP}(R, \mu) = \mathrm{DP}_\mathcal{F}(R, \mu) \cup \mathrm{DP}_\mathcal{V}(R, \mu)$ to be the set of *context-sensitive* dependency pairs (CS-DPs) where:

$$\mathrm{DP}_\mathcal{F}(R, \mu) = \{l^\sharp \to u^\sharp \mid l \to r \in R, u \trianglelefteq_\mu r, \mathrm{root}(u) \in D_R, u \not\trianglelefteq_\mu l\}$$
$$\mathrm{DP}_\mathcal{V}(R, \mu) = \{l^\sharp \to x \mid l \to r \in R, x \in \mathrm{Var}^\mu(r) \setminus \mathrm{Var}^\mu(l)\}$$

Example 4.5 Consider TRS R_2 (in Example 2.5) and $\mathcal{F}$-map μ_2 (in Example 4.2). $\mathrm{DP}_{\mathcal{F}}(R_2, \mu_2) = \{f^\sharp(g(x)) \to f^\sharp(a)\}$ and $\mathrm{DP}_{\mathcal{V}}(R_2, \mu_2) = \{f^\sharp(g(x)) \to x\}$.

Definition 4.6 [Context-Sensitive Semi-Constructor TRS] For an $\mathcal{F}$-map μ, a TRS R is a *μ-semi-constructor TRS* if all rules in $\mathrm{DP}_{\mathcal{F}}(R, \mu)$ are right-ground.

For a given TRS R and an $\mathcal{F}$-map μ, we define $\mu^\sharp$ by $\mu^\sharp(f) = \mu(f)$ for $f \in \mathcal{F}$, and $\mu^\sharp(f^\sharp) = \mu(f)$ for $f \in D_R$. We write $s \trianglerighteq^\sharp_\mu t^\sharp$ for $s \trianglerighteq_\mu t$

Definition 4.7 [Context-Sensitive dependency chain] For a TRS R and $\mathcal{F}$-map μ, a sequence of the elements of $\mathrm{DP}(R, \mu)$ $s_1^\sharp \to t_1^\sharp,\ s_2^\sharp \to t_2^\sharp,\ \ldots$ is a *context-sensitive dependency chain* if there exist substitutions $\tau_1, \tau_2, \ldots$ satisfying both:

- $t_i^\sharp \tau_i \xrightarrow[\mu^\sharp]{}{}^*_R s_{i+1}^\sharp \tau_{i+1}$, if $t_i^\sharp \notin \mathcal{V}$
- $x\tau_i \trianglerighteq^\sharp_\mu u_i^\sharp \xrightarrow[\mu^\sharp]{}{}^*_R s_{i+1}^\sharp \tau_{i+1}$ for some term u_i, if $t_i^\sharp = x$.

For a given TRS and an $\mathcal{F}$-map μ, let $\mathcal{T}_\infty^{\overrightarrow{\mu}_R}$ denote the set of all $\trianglerighteq_\mu$-*minimal non-terminating terms* for $\overrightarrow{\mu}_R$,

Example 4.8 Consider TRS R_2 (in Example 2.5) and $\mathcal{F}$-map μ_2 (in Example 4.2). $f(a), f(g(f(a))) \in \mathcal{T}_\infty^{\overrightarrow{\mu_2}_{R_2}}$ and $f(f(a)), h(f(a), f(a)) \notin \mathcal{T}_\infty^{\overrightarrow{\mu_2}_{R_2}}$.

Theorem 4.9 ([1]) *For a TRS R and an $\mathcal{F}$-map μ, there exists an infinite context-sensitive dependency chain if and only if R does not μ-terminate.*

Let R be a TRS, μ be an $\mathcal{F}$-map and $\mathcal{C} \subseteq \mathrm{DP}(R, \mu)$. We define $\underset{\mu}{\hookrightarrow}_{R,\mathcal{C}}$ as $(\xrightarrow[\mu^\sharp]{}_{\mathcal{C}_\mathcal{F}} \cup (\xrightarrow[\mu^\sharp]{}_{\mathcal{C}_\mathcal{V}} \cdot \trianglerighteq^\sharp_\mu) \cup \xrightarrow[\mu^\sharp]{}_R)$ where $\mathcal{C}_\mathcal{F} = \mathcal{C} \cap \mathrm{DP}_{\mathcal{F}}(R, \mu)$ and $\mathcal{C}_\mathcal{V} = \mathcal{C} \cap \mathrm{DP}_{\mathcal{V}}(R, \mu)$.

Definition 4.10 [μ-$\mathcal{C}$-min] Let R be a TRS, μ be an $\mathcal{F}$-map, An infinite sequence of terms in the form $t_1^\sharp \underset{\mu}{\hookrightarrow}_{R,\mathcal{C}} t_2^\sharp \underset{\mu}{\hookrightarrow}_{R,\mathcal{C}} t_3^\sharp \underset{\mu}{\hookrightarrow}_{R,\mathcal{C}} \cdots$ is called a *$\mathcal{C}$-min μ-sequence* if $t_i \in \mathcal{T}_\infty^{\overrightarrow{\mu}_R}$ for $i \geq 1$.

We use $\mathcal{C}^\mu_{min}(t^\sharp)$ to denote the set of all $\mathcal{C}$-min μ-sequences starting from $t^\sharp$.

Example 4.11 The sequence $f^\sharp(a) \underset{\mu_2}{\hookrightarrow}_{R_2,\mathcal{C}} f^\sharp(g(f(a))) \underset{\mu_2}{\hookrightarrow}_{R_2,\mathcal{C}} f^\sharp(a) \underset{\mu_2}{\hookrightarrow}_{R_2,\mathcal{C}} \cdots$ is $\mathcal{C}$-min μ-sequence.

Proposition 4.12 ([1]) *Given a TRS R and an $\mathcal{F}$-map μ, the following statements hold:*

(i) *If there exists an infinite context-sensitive dependency chain, then $\mathcal{C}^\mu_{min}(t^\sharp) \neq \emptyset$ for some $\mathcal{C} \subseteq DP(R, \mu)$ and $t \in \mathcal{T}_\infty^{\overrightarrow{\mu}_R}$.*

(ii) *For any sequence in $\mathcal{C}^\mu_{min}(t^\sharp)$, a reduction with $\xrightarrow[\mu^\sharp]{}_R$ takes place below the root while reductions with $\xrightarrow[\mu^\sharp]{}_{\mathcal{C}_\mathcal{F}}$ and $\xrightarrow[\mu^\sharp]{}_{\mathcal{C}_\mathcal{V}}$ take place at the root.*

(iii) *For any sequence in $\mathcal{C}^\mu_{min}(t^\sharp)$, there is at least one rule in $\mathcal{C}$ which is applied infinitely often.*

Lemma 4.13 *For two terms s and t, $s^\sharp \underset{\mu}{\hookrightarrow}^*_{R,\mathcal{C}} t^\sharp$ implies $s \xrightarrow[\mu]{}{}^*_R C_\mu[t]$ for some context C_μ.*

Proof. We use induction on length n of the sequence. In the case that $n = 0$, it holds trivially. Let $n \geq 1$. Then we have $s^\sharp \hookrightarrow^*_{\mu R,\mathcal{C}} u^\sharp \hookrightarrow_{\mu R,\mathcal{C}} t^\sharp$ for some u.

• In the case that $u^\sharp \xrightarrow[\mu^\sharp]{}_{\mathcal{C}_\mathcal{F}} t^\sharp$, we have $u \xrightarrow[\mu]{}_R C'_\mu[t]$ by the definition of dependency pairs.

• In the case that $u^\sharp \xrightarrow[\mu^\sharp]{}_{\mathcal{C}_\mathcal{V}} v \trianglerighteq^\sharp_\mu t^\sharp$, we have $u \xrightarrow[\mu]{}_R C''_\mu[v]$ by the definition of dependency pairs and $v = C'''_\mu[t]$. Thus $u \xrightarrow[\mu]{}_R C''_\mu[C'''_\mu[t]] = C'_\mu[t]$.

• In the case that $u^\sharp \xrightarrow[\mu^\sharp]{}_R t^\sharp$, we have $u \xrightarrow[\mu]{}_R C'_\mu[t]$ for $C'_\mu[\,] = \Box$.

Therefore $s \xrightarrow[\mu]{}^*_R C_\mu[u] \xrightarrow[\mu]{}_R C_\mu[C'_\mu[t]]$ by the induction hypothesis and Proposition 2.3. □

For a TRS R and $\mathcal{F}$-map μ, we say R is free from infinite variable dependency chain (FFIVDC) if and only if there exists no infinite context-sensitive dependency chain consists of $\mathrm{DP}_\mathcal{V}(R, \mu)$. If R is FFIVDC, then $\mathcal{C}^\mu_{min}(t^\sharp) = \emptyset$ for any $\mathcal{C} \subseteq \mathrm{DP}_\mathcal{F}(R, \mu)$ and a term t.

Lemma 4.14 *Let μ be an $\mathcal{F}$-map. If μ-semi-constructor TRS R is FFIVDC, then the following statements are equivalent:*

(i) *R does not μ-terminate.*

(ii) *There exists $l^\sharp \to u^\sharp \in DP_\mathcal{F}(R, \mu)$ such that sq head-loops for some $sq \in \mathcal{C}^\mu_{min}(u^\sharp)$.*

Proof. ((ii) ⇒ (i)) : It is obvious from Lemma 4.13, and Proposition 4.3. ((i) ⇒ (ii)) : By Theorem 4.9 there exists an infinite context-sensitive dependency chain. By Proposition 4.12(i), there exists a sequence $sq \in \mathcal{C}^\mu_{min}(t^\sharp)$. By Proposition 4.12(ii),(iii) and the fact that R is FFIVDC, there is some rule in $l^\sharp \to u^\sharp \in \mathcal{C}_\mathcal{F}$ which is applied at root reduction in sq infinitely often.
By definition 4.6, $u^\sharp$ is ground. Thus sq contains a subsequence $u^\sharp \hookrightarrow^+_{\mu R,\mathcal{C}} u^\sharp$, which head-loops and is in $\mathcal{C}^\mu_{min}(u^\sharp)$. □

Lemma 4.15 *Let μ be an $\mathcal{F}$-map. If μ-semi-constructor TRS R is FFIVDC, then μ-termination of R is decidable.*

Proof. The decision procedure for μ-termination of a μ-semi-constructor TRS R is as follows: consider all terms $u_1, u_2, \ldots, u_n$ corresponding to the right-hand sides of $\mathrm{DP}_\mathcal{F}(R, \mu) = \{l^\sharp_i \to u^\sharp_i \mid 1 \leq i \leq n\}$, and simultaneously generate all μ-reduction sequences with respect to R starting from $u_1, u_2, \ldots, u_n$. It halts if it enumerates all reachable terms exhaustively or it detects a μ-looping reduction sequence $u_i \xrightarrow[\mu]{}^+_R C_\mu[u_i]$ for some i.

Suppose R does not μ-terminate. By Lemma 4.14, 4.13, we have a μ-looping reduction sequence $u_i \xrightarrow[\mu]{}^+_R C_\mu[u_i]$ for some i and C_μ, which we eventually detect. If R μ-terminates, then the execution of the reduction sequence generation eventually stops since it is finitely branching. Moreover it does not detect a μ-looping sequence, otherwise it contradicts to Proposition 4.3. Thus the procedure decides μ-termination of R in finitely many steps. □

We have to check FFIVDC property in order to use Lemma 4.15. The following Proposition provides its sufficient condition. The set $\mathrm{DP}^1_\mathcal{V}(R, \mu)$ is a subset of

$\mathrm{DP}_{\mathcal{V}}(R,\mu)$ defined as follows

$$\mathrm{DP}^1_{\mathcal{V}}(R,\mu) = \{f^\sharp(u_1,\ldots,u_k) \to x \in \mathrm{DP}_{\mathcal{V}}(R,\mu) \mid \exists i, 1 \le i \le k, i \notin \mu(f), x \in Var(u_i)\}$$

Proposition 4.16 ([1]) *Let R be a TRS, μ be an $\mathcal{F}$-map and $\mathcal{C} = DP^1_{\mathcal{V}}(R,\mu)$. $\mathcal{C}^\mu_{min}(t^\sharp) = \emptyset$ for any term t.*

Since $\mathrm{DP}^1_{\mathcal{V}}(R,\mu) = \mathrm{DP}_{\mathcal{V}}(R,\mu)$ implies that R is FFIVDC by Proposition 4.16, the following theorem directly follows from Lemma 4.15.

Theorem 4.17 *Let μ be an $\mathcal{F}$-map and R be a μ-semi-constructor TRS. μ-termination of R is decidable if $DP_{\mathcal{V}}(R,\mu) = DP^1_{\mathcal{V}}(R,\mu)$.*

In the following, we show that μ-termination of semi-constructor TRSs (not μ-semi-constructor) is decidable.

Lemma 4.18 *Consider a reduction $s^\sharp = C_{\mu^\sharp}[l\theta]_p \xrightarrow[\mu^\sharp]{}_R t^\sharp = C_{\mu^\sharp}[r\theta]_p = C'[u]_q$ where $s,u \in \mathcal{T}_\infty^{\overrightarrow{\mu}_R}$ and $q \in \mathcal{P}\mathrm{os}(t) \setminus \mathcal{P}\mathrm{os}_\mu(t)$. Then one of the following statements hold*

(i) $s \rhd u$

(ii) $v\theta = u$ *and* $r = C''[v]_{q'}$ *for some* θ, $v \notin \mathcal{V}$, C'', *and* $q' \in \mathcal{P}\mathrm{os}(r) \setminus \mathcal{P}\mathrm{os}_\mu(r)$

Proof. Since $q \in \mathcal{P}\mathrm{os}(t) \setminus \mathcal{P}\mathrm{os}_\mu(t)$, p is not below or equal to q. In the case that p and q are in parallel positions, $s \rhd u$ trivially holds. In the case that p is above q, it is obvious that $s \rhd u$ hold or, $v\theta = u$ and $r = C''[v]_{q'}$ for some θ, $v \notin \mathcal{V}$, C''. Here the fact that $q' \in \mathcal{P}\mathrm{os}(r) \setminus \mathcal{P}\mathrm{os}_\mu(r)$ follows from $p \in \mathcal{P}\mathrm{os}_\mu(t)$ and $q \notin \mathcal{P}\mathrm{os}_\mu(t)$. □

Lemma 4.19 *Let R be a semi-constructor TRS, μ be an $\mathcal{F}$-map. For a $\mathcal{C}$-min μ-sequence $s^\sharp_1 \xrightarrow[\mu^\sharp]{}^*_R t^\sharp_1 \xrightarrow[\mu^\sharp]{}_{\mathcal{C}_\mathcal{V}} u_1 \trianglerighteq^\sharp_\mu s^\sharp_2 \cdots$ with no reduction by rules in $\mathcal{C}_\mathcal{F}$, one of following statements hold for each i:*

(i) $s_i \rhd s_{i+1}$

(ii) *There exists* $l^\sharp \to s^\sharp_{i+1} \in DP(R)$ *for some* l

Proof. We have $t^\sharp_i = C[s_{i+1}]_q$ for some $q \in \mathcal{P}\mathrm{os}(t_i) \setminus \mathcal{P}\mathrm{os}_\mu(t_i)$. We show (i) or the following (ii') by induction on the number n of steps of $s^\sharp_i \xrightarrow[\mu^\sharp]{}^n_R t^\sharp_i = C[s_{i+1}]$.

(ii') There exist a reduction by $l \to r$ in $s^\sharp_i \xrightarrow[\mu^\sharp]{}^*_R t^\sharp_i$ and $l^\sharp \to s^\sharp_{i+1} \in \mathrm{DP}(R)$

• In the case that $n = 0$, trivially $s_i = t_i \rhd s_{i+1}$.

• In the case that $n > 0$, let $s^\sharp_i \xrightarrow[\mu^\sharp]{}_R s'^\sharp \xrightarrow[\mu^\sharp]{}^{n-1}_R t^\sharp_i = C[s_{i+1}]_q$. By the induction hypothesis, the condition (ii') or $s' \rhd s_{i+1}$ follows. In the former case, it is trivial. In the latter case, by Lemma 4.18, we have $s_i \rhd s_{i+1}$, or, we have $v\theta = s_{i+1}$ and $r = C'[v]_{q'}$ for some $l \to r \in R$, θ, $v \notin \mathcal{V}$, C' and $q' \in \mathcal{P}\mathrm{os}(r) \setminus \mathcal{P}\mathrm{os}_\mu(r)$ by Lemma 4.18. Hence $v\theta = v$ due to $\mathrm{root}(s_{i+1}) \in D_R$ and Proposition 2.6(ii). Therefore (ii') follows. □

One may think that the Lemma 4.19 would hold even if $\mathrm{DP}(R)$ were replaced with $\mathrm{DP}(R,\mu)$. However, it does not hold from the following counter example.

Example 4.20 Let $R_3 = \{f(g(x)) \to x,\ g(b) \to g(f(g(b)))\}$, $\mu_3(f) = \{1\}$ and $\mu_3(g) = \emptyset$. There exists a $\mathcal{C}$-min μ_3-sequence $f^\sharp(g(b)) \xrightarrow[\mu_3^\sharp\ R_3]{} f^\sharp(g(f(g(b)))) \xrightarrow[\mu_3^\sharp\ \mathcal{C}_\mathcal{V}]{}$ $f(g(b)) \trianglerighteq^\sharp_{\mu_3} f^\sharp(g(b))$ where $\mathcal{C}_\mathcal{V} = \mathrm{DP}_\mathcal{V}(R_3, \mu_3)$.

A dependency pair whose right-hand side is $f^\sharp(g(b))$ is in $\mathrm{DP}(R)$ but not in $\mathrm{DP}(R, \mu)$.

Lemma 4.21 *For a semi-constructor TRS R and an $\mathcal{F}$-map μ, the following statements are equivalent:*

(i) *R does not μ-terminate.*

(ii) *There exists $l^\sharp \to u^\sharp \in DP(R)$ such that sq head-loops for some $sq \in \mathcal{C}^\mu_{min}(u^\sharp)$.*

Proof. ((ii) $\Rightarrow$ (i)) : It is obvious from Lemma 4.13, and Prop 4.3. ((i)$\Rightarrow$ (ii)) : By Theorem 4.9 there exists a context-sensitive dependency chain. By Proposition 4.12(i), there exists a sequence $sq \in \mathcal{C}^\mu_{min}(t^\sharp)$. By Proposition 4.12(ii),(iii), there exists a rule in $\mathcal{C}$ applied at root position in sq infinitely often.
• Consider the case that there exists a rule $l^\sharp \to r^\sharp \in \mathcal{C}_\mathcal{F}$ with infinite use in sq. Since u is ground by Proposition 4.12(ii) and $\mathcal{C}_\mathcal{F} \subseteq DP(R)$, sq has a subsequence $u^\sharp \underset{\mu}{\hookrightarrow}^+_{R,\mathcal{C}} u^\sharp$.
• Otherwise, sq has a infinite subsequence without use of rules in $\mathcal{C}_\mathcal{F}$. The subsequence is in $\mathcal{C}^\mu_{min}(s^\sharp)$ for some $s^\sharp$. Then the condition (ii) of Lemma 4.19 holds for infinitely many i's, otherwise we have an infinite sequence $s_k \rhd s_{k+1} \rhd \cdots$ for some k, which is a contradiction. Hence there exists a $l^\sharp \to u^\sharp \in \mathrm{DP}(R)$ such that $u^\sharp$ occurs more than twice in sq. Thus the sequence $u^\sharp \underset{\mu}{\hookrightarrow}^+_{R,\mathcal{C}} u^\sharp$ appears in sq. □

Theorem 4.22 *μ-termination of semi-constructor TRSs is decidable.*

Proof. The decision procedure for μ-termination of a semi-constructor TRS R is as follows: consider all terms $u_1, u_2, \ldots, u_n$ corresponding to the right-hand sides of $\mathrm{DP}(R) = \{l_i^\sharp \to u_i^\sharp \mid 1 \le i \le n\}$, and simultaneously generate all μ-reduction sequences with respect to R starting from $u_1, u_2, \ldots, u_n$. It halts if it enumerates all reachable terms exhaustively or it detects a μ-looping reduction sequence $u_i \xrightarrow[\mu]{}^+_R C_\mu[u_i]$ for some i.

Suppose R does not μ-terminate. By Lemma 4.21 and 4.13, we have a μ-looping reduction sequence $u_i \xrightarrow[\mu]{}^+_R C_\mu[u_i]$ for some i and C_μ, which we eventually detect. If R μ-terminates, then the execution of the reduction sequence generation eventually stops since it is finitely branching. Moreover it does not detect a μ-looping sequence, otherwise it contradicts to Proposition 4.3. Thus the procedure decides μ-termination of R in finitely many steps. □

5 Some Extension and Example

5.1 Innermost

In this subsection, we extend the class for which innermost termination is decidable by using the dependency graph.

Lemma 5.1 *Let R be a TRS whose innermost termination is equivalent to the non-existence of an innermost dependency chain that contains infinite use of right-ground dependency pairs. Then innermost termination of R is decidable.*

Proof. We apply the procedure which is used in proof of Lemma 3.9 starting with terms $u_1, u_2, \ldots, u_n$, where $u_i^\sharp$'s are all ground right-hand sides of dependency pairs. Suppose R is innermost non-terminating, we have an innermost dependency chain with infinite use of a right-ground dependency pair. Similarly to the semi-constructor case, we have a looping sequence $u_i \xrightarrow[in]{}{}^+_R C[u_i]$, which can be detected by the procedure. □

Definition 5.2 [Innermost DP-Graph [2]] The *innermost dependency graph* of a TRS R is a directed graph whose nodes are the dependency pairs and there is an arc from $s^\sharp \to t^\sharp$ to $u^\sharp \to v^\sharp$ if there exist normal substitutions σ and τ such that $t^\sharp\sigma \xrightarrow[in]{}{}^*_R u^\sharp\tau$ and $u^\sharp\tau$ is in normal forms with respect to R.

An approximated dependency graph is a graph that contains the innermost dependency graph as subgraph. Computable such graphs are proposed in [2], for example.

Theorem 5.3 *Let R be a TRS and G be an approximated dependency graph of R. If at least one node in the cycle is right-ground for every cycle of G, then innermost termination of R is decidable.*

Proof. From Lemma 5.1. □

Example 5.4 Let $R_3 = \{f(s(x)) \to g(x),\ g(s(x)) \to f(s(0))\}$. Then $\mathrm{DP}(R_3) = \{f^\sharp(s(x)) \to g^\sharp(x),\ g^\sharp(s(x)) \to f^\sharp(s(0))\}$. The innermost dependency graph of R_3 has one cycle, which contains a right ground node [Fig. 1]. The innermost termination of R_3 is decidable by Theorem 5.3. Actually we know R_3 is innermost terminating from the procedure in the proof of Theorem 3.9 since all innermost reduction sequences from $f(s(0))$ terminate.

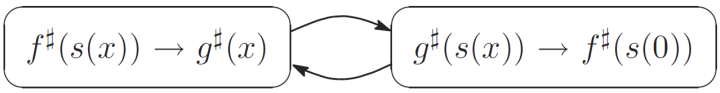


Fig. 1. The innermost DP-Graph of R_3

Example 5.5 Let $R_4 = \{a \to b,\ f(a,x) \to x,\ f(x,b) \to g(x,x),\ g(b,x) \to h(f(a,a),x)\}$. Then $\mathrm{DP}(R_4) = \{f^\sharp(x,b) \to g^\sharp(x,x),\ g^\sharp(b,x) \to f^\sharp(a,a),\ g^\sharp(b,x) \to a^\sharp\}$. The innermost dependency graph of R_4 has one cycle, which contains a right ground node [Fig. 2]. The innermost termination of R_4 is decidable by Theorem 5.3. Actually we know R_4 is not innermost terminating from the procedure in the proof of Theorem 3.9 by detecting the looping sequence $f(a,a) \xrightarrow[in]{}_{R_4} f(b,b) \xrightarrow[in]{}_{R_4} g(b,b) \xrightarrow[in]{}_{R_4} h(f(a,a),b)$.

5.2 Context-Sensitive

We extend the class for which μ-termination is decidable by using the dependency graph. The class extended in this subsection is the class that satisfy the condition

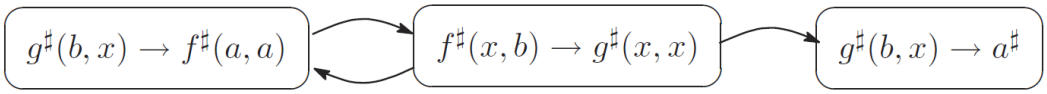


Fig. 2. The innermost DP-Graph of R_4

of Theorem 4.17.

Lemma 5.6 *Let R be a TRS and μ be an $\mathcal{F}$-map. If μ-termination of R is equivalent to the non-existence of a context-sensitive dependency chain that contains infinite use of right-ground rule in $DP_{\mathcal{F}}(R,\mu)$. Then μ-termination of R is decidable.*

Proof. We apply the procedure which used on proof of Lemma 4.22 starting with terms $u_1, u_2, \ldots, u_n$, where $u_i^\sharp$'s are all ground right-hand sides of rules in $\mathrm{DP}_{\mathcal{F}}(R,\mu)$. Suppose R is non-μ-termination, we have an context-sensitive dependency chain with infinite use of right-ground rule in $\mathrm{DP}_{\mathcal{F}}(R,\mu)$. Similarly to the μ-semi-constructor case, we have a looping sequence $u_i \xrightarrow[\mu]{}{}^{+}_{R} C_\mu[u_i]$, which can be detected by the procedure. □

Definition 5.7 [Context-Sensitive DP-Graph [1]] The *Context-Sensitive dependency graph* of a TRS R and an $\mathcal{F}$-map μ is directed graph whose nodes are CS-dependency pairs:

(i) There is an arc from $s \to t \in \mathrm{DP}_{\mathcal{F}}(R,\mu)$ to $u \to v \in \mathrm{DP}(R,\mu)$ if there exist substitutions σ and τ such that $t\sigma \xrightarrow[\mu^\sharp]{}{}^{*}_{R} u\tau$.

(ii) There is an arc from $s \to t \in \mathrm{DP}_{\mathcal{V}}(R,\mu)$ to each dependency pair $u \to v \in \mathrm{DP}(R,\mu)$.

Similarly to the innermost case, a computable approximated dependency graph is proposed for context-sensitive DP-graph[1].

Theorem 5.8 *Let R be a TRS, μ be an $\mathcal{F}$-map and G be an approximated DP-graph of R. μ-termination of R is decidable if one of followings hold for every cycle in G.*

(i) *The cycle contains at least one node that is right-ground.*

(ii) *All nodes of the cycle are elements in $DP^1_{\mathcal{V}}(R,\mu)$.*

Proof. From Lemma 5.6 and Theorem 4.16. □

Example 5.9 Let $R_5 = \{h(x) \to g(x,x),\ g(a,x) \to f(b,x),\ f(x,x) \to h(a),\ a \to b\}$, $\mu_5(f) = \mu_5(g) = \mu_5(h) = \{1\}$ and $\mu_5(a) = \mu_5(b) = \emptyset$ [7]. Then $\mathrm{DP}(R_5,\mu_5) = \{h^\sharp(x) \to g^\sharp(x,x),\ g^\sharp(a,x) \to f^\sharp(b,x),\ f^\sharp(x,x) \to h^\sharp(a),\ f^\sharp(x,x) \to a^\sharp\}$. The context-sensitive dependency graph of R_5 and μ_5 has one cycle, which contains a right ground node [Fig.3]. The μ_5-termination of R_5 is decidable by Theorem 5.8. Actually we know R_5 is μ_5-terminating from the procedure in the proof of Theorem 4.15 since all μ_5-reduction sequences from $h(a)$ terminates.

Example 5.10 Let $\mu_6(f) = \{2\}$, $\mu_6(g) = \mu_6(h) = \{1\}$ and $\mu_6(a) = \mu_6(b) = \emptyset$. Consider μ_6-termination of R_5. The context-sensitive dependency graph of R_5

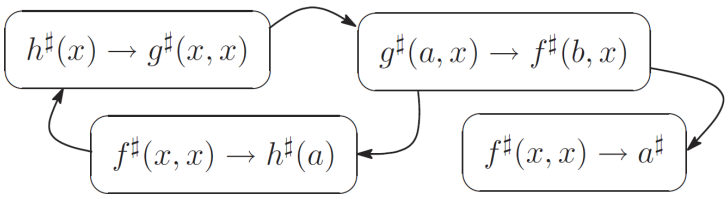


Fig. 3. The innermost DP-Graph of R_5 and μ_5

and μ_6 is same as one of R_5 and μ_5 [Fig.3]. The μ_6-termination of R_5 is decidable by Theorem 5.8. By the decision procedure, we can detect the μ_6-looping sequence $h(a) \xrightarrow[\mu_6]{}_{R_5} g(a,a) \xrightarrow[\mu_6]{}_{R_5} f(b,a) \xrightarrow[\mu_6]{}_{R_5} f(b,b) \xrightarrow[\mu_6]{}_{R_5} h(a)$. Thus R_5 is non-μ_5-terminating.

The class of TRS that satisfy the condition of Theorem 5.8 is a superclass of the class of TRS that satisfy the condition of Theorem 4.17. However, the semi-constuctor class and none of these classes are included from each other.

Acknowledgement

We would like to thank the anonymous referees for their helpful comments and remarks. This work is partly supported by MEXT.KAKENHI #18500011 and #16300005.

References

[1] B. Alarcón, R. Gutiérrez, and S. Lucas. Context-Sensitive Dependency Pairs. In *the 26th Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 4337 of *Lecture Notes in Computer Science*, pages 298–309, 2006.

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

[3] F. Baader and T. Nipkow. *Term rewriting and all that.* Cambridge University Press, 1998.

[4] N. Dershowitz. Termination of Linear Rewriting Systems. In *the 8th International Colloquium on Automata, Languages and Programming*, volume 115 of *Lecture Notes in Computer Science*, pages 448–458, 1981.

[5] G. Godoy and A. Tiwari. Termination of Rewrite Systems with Shallow Right-Linear, Collapsing, and Right-Ground Rules. In *the 20th International Conference on Automated Deduction*, volume 3632 of *Lecture Notes in Computer Science*, pages 164–176, 2005.

[6] G. Huet and D. Lankford. On the Uniform Halting Problem for Term Rewriting Systems. Technical report, INRIA, 1978.

[7] S. Lucas. Proving Termination of Context-Sensitive Rewriting by Transformation. *Information and Computation*, 204:1782–1846, 2006.

[8] T. Nagaya and Y. Toyama. Decidability for left-linear growing term rewriting systems. *Information and Computation*, 178:499–514, 2002.

[9] Y. Wang and M. Sakai. Decidability of Termination for Semi-Constructor TRSs, Left-Linear Shallow TRSs and Related Systems. In *the 17th International Conference on Rewriting Techniques and Applications*, volume 4098 of *Lecture Notes in Computer Science*, pages 343–356, 2006.

[10] H. Zantema. Termination of context-sensitive rewriting. In *the 8th Conference on Rewriting Techniques and Applications*, volume 1232 of *Lecture Notes in Computer Science*, pages 172–186, 1997.

Termination of Lazy Rewriting Revisited

Felix Schernhammer and Bernhard Gramlich

TU Wien, Austria, email: {felixs,gramlich}@logic.at

Abstract

Lazy rewriting is a proper restriction of term rewriting that dynamically restricts the reduction of certain arguments of functions in order to obtain termination. In contrast to context-sensitive rewriting reductions at such argument positions are not completely forbidden but delayed. Based on the observation that the only existing (non-trivial) approach to prove termination of such lazy rewrite systems is flawed, we develop a modified approach for transforming lazy rewrite systems into context-sensitive ones that is sound and complete with respect to termination.

1 Introduction

In functional programming languages, evaluations are often carried out in a lazy fashion. This means that in the evaluation of an expression, the result of certain subexpressions is not computed until it is known that the particular result is actually needed. A very similar idea is used in lazy rewriting ([FKW00]) where the reduction of certain subterms is delayed as long as possible.

Initially, lazy rewriting was introduced by [FKW00] in a graph rewriting setting (although the basic underlying idea is much older, cf. e.g. [FW76], [Str89], [Pv93]). However, for the termination analysis of lazy rewrite systems it is favorable to consider term rewriting instead of graph rewriting. Therefore, we will use the notion of lazy rewriting introduced in [Luc02b].

Restrictions of term rewriting have been studied over the last decades and have been used both for practical implementations of specification languages (cf. e.g. strategy annotations in Maude [CDE+03]) and for theoretical results (cf. e.g. [Luc98,Luc06], [GL06]). Some of the most sophisticated approaches like *on-demand rewriting* ([Luc01]) and *rewriting with on-demand strategy annotations* ([AEGL03]) incorporate lazy evaluation features. Thus, a better understanding of lazy rewriting may contribute to an improved understanding and analysis of these more recent approaches.

In [Luc02b] a transformation from lazy rewrite systems into context-sensitive ones was proposed, which was supposed to preserve non-termination and conjectured to be complete w.r.t. termination. Unfortunately, a counterexample (see Example 3.3 below) proves that this transformation is unsound w.r.t. termination. In this paper we repair the transformation and prove both soundness and completeness of the new transformation w.r.t. termination.

In Section 2 of this paper we will give basic definitions and introduce basic notations of lazy rewriting. In Section 3 we introduce the transformation of [Luc02b] and give a counterexample to its soundness w.r.t. termination. We propose a modified

version of the transformation which is proved to be sound and complete w.r.t. termination. Section 4 contains a discussion of the presented results and concludes. [1]

2 Preliminaries

We assume familiarity with the basic concepts and notations in term rewriting as well as context-sensitive term rewriting as provided for instance in [BN98,Luc98].

As in [FKW00] and [Luc02b] we are concerned with left-linear lazy rewrite systems in this work.

General assumption: Throughout the paper we assume that all lazy rewrite systems are *left-linear* [2] and *finite*.

Lazy rewriting operates on labelled terms. Each function and variable symbol of a term has either an *eager* label e or a *lazy* label l which we will write as superscripts. So given a signature $\Sigma = \{f_1, ..., f_n\}$, we consider a new signature $\Sigma' = \{f_1^e, f_1^l, ..., f_n^e, f_n^l\}$. We denote by V' the set of labelled variables, so $\mathcal{T}(\Sigma', V')$ is the set of labelled terms of a labelled signature Σ'. Following [Luc02b] we use a replacement map μ to specify for each function $f \in \Sigma$ which arguments should be evaluated eagerly. Given a replacement map μ we define the *canonical labelling* of terms as a mapping $label_\mu : \mathcal{T}(\Sigma, V) \to \mathcal{T}(\Sigma', V')$, where Σ' is the labelled signature and V' are the labelled variables [Luc02b]:

$$\begin{aligned} label_\mu(t) &= label^e_\mu(t) \\ label^\alpha_\mu(x) &= x^\alpha (\alpha \in \{e, l\}) \\ label^\alpha_\mu(f(t_1, ..., t_n)) &= f^\alpha(label^{\alpha_1}_\mu(t_1), ..., label^{\alpha_n}_\mu(t_n)) \\ &\quad \text{where } \alpha_i = e \text{ if } i \in \mu(f), \ l \text{ otherwise, and } \alpha \in \{e, l\} \end{aligned}$$

Given a labelled term t, the unlabelled term $erase(t)$ is constructed from t by omitting all labels. A position p of a term t is said to be *eager* (resp. *lazy*), if the symbol at the root of the subterm starting at position p of t has an *eager* (resp. *lazy*) label. Note that the *lazy* positions of a term are not the same as the non-replacing positions in context-sensitive rewriting. The reason is that in lazy rewriting eager positions may occur below lazy ones whereas in context-sensitive rewriting all positions which are below a non-replacing position are non-replacing.

However, rewrite steps may only be performed at so-called *active* positions. A position p is called *active* if all positions on the path from the root to p are eager.

Definition 2.1 ([Luc02b], [FKW00]) *The* active *positions of a labelled term t (denoted $Act(t)$) are recursively defined as follows.*

- *the root position ϵ of t is* active
- *if p is an* active *position and position $p.i$ is* eager*, then position $p.i$ is* active.

[1] Due to lack of space, the proofs of some auxiliary results (Propositions 3.7, 3.8 and 3.9 and Lemmata 3.13, 3.14, 3.19, 3.20 and 3.21 have been omitted here. They can be found in the full version of the paper at `http://www.logic.at/people/schernhammer/papers/wrs07-long.pdf`.

[2] Nevertheless, for clarity we will mention this assumption in the main results.

Note that given an unlabelled term t and a replacement map μ, the active positions of $label_\mu(t)$ are exactly the replacing positions w.r.t. context-sensitive rewriting.

Definition 2.2 ([Luc02b], [FKW00]) *Let $l \in \mathcal{T}(\Sigma, V)$ be linear, $t \in \mathcal{T}(\Sigma', V')$ be a labelled term and let p be an* active *position of t. Then l matches $t|_p$ modulo laziness if either*

- *$l \in V$ or*
- *If $l = f(l_1, ..., l_n)$ and $t|_p = f^e(t_1^\alpha, ..., t_n^\alpha)$ $(\alpha \in \{e, l\})$, then for all* eager *subterms t_i^e, l_i matches modulo laziness t_i^e.*

If t_i^l at positions $p.i$ is a lazy subterm and $l_i \notin V$, then position $p.i$ is called essential.

When writing t^e (resp. t^l) we mean that the term t has an eager (resp. lazy) root label. Informally, a matching modulo laziness is a partial matching ignoring (possible) clashes at lazy positions. Positions where such clashes occur may be activated (i.e., their label may be changed from lazy to eager).

Definition 2.3 ([Luc02b]) *Let $\mathcal{R} = (\Sigma, R)$ be a (left-linear) TRS. Let t be a labelled term and let l be the left-hand side of a rule of R. If l matches modulo laziness $t|_p$, and this matching gives rise to an essential position $p.i$ $(t|_{p.i} = f^l(t_1, ..., t_n))$, then $t \xrightarrow{A} t[f^e(t_1, ..., t_n)]_{p.i}$. The relation $\xrightarrow{A}$ is called* activation relation.

Definition 2.4 ([Luc02b]) *Let l be the (linear) left-hand side of a rewrite rule and let t be a labelled term. If l matches $erase(t)$, then the mapping $\sigma_{l,t} : Var(l) \to \mathcal{T}(\Sigma', V')$ is defined as follows. For all $x \in V$, with $l|_q = x$: $\sigma_{l,t}(x) = t|_q$.*

Informally, $\sigma_{l,t}$ is the matcher when matching l against t, where one adds the appropriate labels of t.

This substitution is modified to operate on labelled terms in the following way, yielding the mapping $\sigma\colon V' \to \mathcal{T}(\Sigma', V')$ [Luc02b]:

$$\sigma(x^e) = \begin{cases} y^e & \text{if } \sigma_{l,u}(x) = y^\alpha \in V' \\ f^e(t_1, ..., t_n) & \text{if } \sigma_{l,u}(x) = f^\alpha(t_1, ..., t_n) \end{cases}$$

$$\sigma(x^l) = \begin{cases} y^l & \text{if } \sigma_{l,u}(x) = y^\alpha \in V' \\ f^l(t_1, ..., t_n) & \text{if } \sigma_{l,u}(x) = f^\alpha(t_1, ..., t_n) \end{cases}$$

σ is homeomorphically extended to a mapping $\mathcal{T}(\Sigma', V') \to \mathcal{T}(\Sigma', V')$ as usual.

Definition 2.5 ([Luc02b]) *Let $\mathcal{R} = (\Sigma, R)$ be a (left-linear) TRS with replacement map μ. The* active rewrite *relation $\xrightarrow{R}_\mu$: $\mathcal{T}(\Sigma', V') \times \mathcal{T}(\Sigma', V')$ is defined as follows: Let t be a labelled term such that the left-hand side of a rewrite rule $l \to r$ matches $erase(t|_p)$ with $\sigma_{l,t|_p}$ and let $p \in Act(t)$. Then $t \xrightarrow{R}_\mu t[\sigma(label_\mu(r))]_p$.*

Informally, the active rewrite relation $\xrightarrow{R}_\mu$ performs rewrite steps according to rewrite rules as usual at active positions, where labels are considered. The lazy rewrite relation $\xrightarrow{LR}_\mu$ is the union of the activation relation and the active rewrite relation.

Definition 2.6 ([Luc02b]) *Let $\mathcal{R}$ be a (left-linear) TRS and let μ be a replacement map for $\mathcal{R}$. The lazy rewrite relation $\overset{LR}{\to}_\mu$ induced by $(\mathcal{R}, \mu)$ is the union of the two relations $\overset{A}{\to}$ and $\overset{R}{\to}_\mu$ ($\overset{LR}{\to}_\mu = \overset{A}{\to} \cup \overset{R}{\to}_\mu$).*

Definition 2.7 *Let $\mathcal{R}$ be a TRS with a replacement map μ. Then $\mathcal{R}$ is $LR(\mu)$-terminating if there is no infinite $\overset{LR}{\to}_\mu$-sequence starting from a term t, whose labelling is canonical or more liberal (i.e., whenever $label_\mu(erase(t))|_p$ is eager, then $t|_p$ is eager as well).*

Informally, we call a labelled term t more liberal than its canonically labelled version $label_\mu(erase(t))$ if it has strictly more eager labels. The reason for considering terms with canonical or more liberal labelling in the definition of $LR(\mu)$-termination, is that only such terms appear in lazy reduction sequences starting from canonically labelled terms, in which we are actually interested.

Note that $LR(\mu)$-termination and well-foundedness of $\overset{LR}{\to}_\mu$ do not coincide in general.

Example 2.8 *Consider the TRS $g(f(a), c) \to a$, $h(x, f(b)) \to g(x, h(x, x))$ with a replacement map $\mu(f) = \mu(g) = \{1\}$ and $\mu(h) = \{1, 2\}$. This system is $LR(\mu)$-terminating. This can be shown with the transformation of Definition 3.6 and Theorem 3.22. However, $\overset{LR}{\to}_\mu$ is not well-founded:*

$$\underline{g^e(f^e(b^l), h^l(f^e(b^l), f^e(b^l)))} \overset{LR}{\to}_\mu g^e(f^e(b^l), h^e(f^e(b^l), f^e(b^l))) \overset{LR}{\to}_\mu g^e(f^e(b^l), \underline{g^e(f^e(b^l), h^l(f^e(b^l), f^e(b^l)))}) \overset{LR}{\to}_\mu \dots .$$

3 Transforming Lazy Rewrite Systems

We start with the definition of the transformation of [Luc02b], because it provides the basic ideas for our new one. The main idea of the transformation is to explicitly mimic activation steps of lazy rewriting through special activation rules in the transformed system which basically exchange function symbols to make them more eager (this goes back to [Ngu01]). Activations in lazy rewriting are possible at positions which correspond to a non-variable position of the left-hand side of some rule in a partial matching. This is why in the transformation we are concerned with non-variable lazy positions of left-hand sides of rules.

The transformation is iterative. In each iteration new rules are created until a fixpoint is reached. The following definition identifies for a rule $l \to r$ and a position p the positions $p.i$ which are lazy in $label_\mu(l)$. These positions are dealt with in parallel in one step of the transformation.

Definition 3.1 ([Luc02b]) *Let $l \to r$ be a rewrite rule and p a non-variable position of l, then*

$$\mathcal{I}(l, p) = \{i \in \{1, ..., ar(root(l|_p))\} \mid i \notin \mu(root(l|_p)) \wedge p.i \in Pos_\Sigma(l)\}$$

Definition 3.2 ([Luc02b]) *Let $\mathcal{R} = (\Sigma, R)$ be a TRS with replacement map μ and let $\mathcal{I}(l, p) = \{i_1, \dots, i_n\} \neq \emptyset$ for some rule $l \to r \in R$ and $p \in Pos_\Sigma(l)$ where $root(l|_p) = f$. The transformed system $\mathcal{R}^\diamond = (\Sigma^\diamond, R^\diamond)$ and $\mu^\diamond$ are defined as follows:*

- $\Sigma^\diamond = \Sigma \cup \{f_j \mid 1 \leq j \leq n\}$
- $\mu^\diamond(f_j) = \mu(f) \cup \{i_j\}$ *for all* $1 \leq j \leq n$ *and* $\mu^\diamond(g) = \mu(g)$ *for all* $g \in \Sigma$
- $R' = R - \{l \to r\} \cup \{l'_j \to r \mid 1 \leq j \leq n\} \cup \{l[x]_{p.i_j} \to l'_j[x]_{p.i_j} \mid 1 \leq j \leq n\}$

where $l'_j = l[f_j(l|_{p.1}, ..., l|_{p.m})]_p$ *if* $ar(f) = m$, x *is a fresh variable and* f_j *are new function symbols of arity* $ar(f_j) = ar(f)$.

The transformation of Definition 3.2 is iterated until arriving at a system $\mathcal{R}^\natural = (\Sigma^\natural, R^\natural)$ and $\mu^\natural$ such that $\mathcal{I}(l, p) = \emptyset$ for every rule $l \to r \in R^\natural$ and every position $p \in Pos_\Sigma(l)$.

In [Luc02b] it remains unspecified how the pair l, p is selected in one step of the transformation. However, the order in which those pairs are considered can be essential.

Example 3.3 *Consider the TRS*

$$f(g(a), a) \to a \qquad b \to f(g(c), b)$$

with a replacement map $\mu(f) = \{1\}$ *and* $\mu(g) = \emptyset$. *This system is not* $LR(\mu)$*-terminating:*

$$b^e \overset{LR}{\to}_\mu f^e(g^e(c^l), b^l) \overset{LR}{\to}_\mu f^e(g^e(c^l), b^e) \overset{LR}{\to}_\mu f^e(g^e(c^l), f^e(g^e(c^l), b^l)) \overset{LR}{\to}_\mu \dots$$

However, if we start the transformation with the first rule and position $p = \epsilon$, *and consider position* 1 *of the first rule in the second step of the transformation, then we arrive at the context-sensitive system*

$$\begin{aligned} f_2(g_1(a), a) &\to a & f(g'_1(a), x) &\to f_2(g(a), x) \\ f_2(g(x), a) &\to f_2(g_1(x), a) & f(g(x), y) &\to f(g'_1(x), y) \\ b &\to f(g(c), b) \end{aligned}$$

with $\mu(f) = \mu(g_1) = \mu(g'_1) = \{1\}$ *and* $\mu(f_2) = \{1, 2\}$. *This system is* μ*-terminating (proved with AProVE [GTSK06]). The lazy reduction sequence starting from* b *cannot be mimicked anymore, because due to the two transformation steps first the argument of* g *has to be activated which prevents the activation of the* b *in the second argument of* f.

In Lucas' transformation, positions that are dealt with last during the transformation must be activated first in rewrite sequences of the transformed system. This can be seen in Example 3.3 where $\mathcal{I}(f(g(a), a), \epsilon)$ is considered in the first step of the transformation, but position 2 must be activated after position 1.1 (whose activation is enabled by a later transformation step considering $\mathcal{I}(f(g(a), x), 1)$).

Thus, the order in which lazy positions of rules are dealt with during the transformation is the reversed order in which they may be activated in the resulting transformed system. Hence, as we want to be able to activate more outer positions before more inner ones, we consider more inner lazy positions first in our new transformation.

Despite considering more inner positions first in the transformation, we do not

want to prioritize any lazy positions. Thus, we define $\mathcal{I}(l)$ which identifies the innermost lazy positions in a term with respect to a given replacement map μ.

Definition 3.4

$$\mathcal{I}(l) = \{p \in Pos_\Sigma(l) \mid p \text{ is lazy in } label_\mu(l) \wedge \\ \wedge (\nexists q \in Pos_\Sigma(l) : q \text{ lazy in } label_\mu(l) \wedge q > p)\}$$

Since all new function symbols, which are introduced by the transformation, are substituted for function symbols of the original signature, we define the mapping *orig* from the signature of the transformed system into the original signature which identifies for each new function symbol the original one for which it was substituted.

Definition 3.5 *Let $\mathcal{R} = (\Sigma, R)$ be a TRS with replacement map μ. If in one step of the transformation $f \in \Sigma$ is replaced by a new function symbol f', then $orig(f') = f$. Furthermore, if f' is substituted for a function symbol $g \notin \Sigma$, then $orig(f') = orig(g)$. For function symbols $h \in \Sigma$, we set $orig(h) = h$ and for variables we have $orig(x) = x$.*

The actual transformation proceeds in 3 stages. First, a set of initial activation rules is created. These rules allow the activation of one *innermost* position of a left-hand side of the original rules of the lazy TRS. As already indicated, by a rule activating position $p.i$ we mean a rule $l \to r$ where l and r differ only in the function symbol at position p and $p.i$ is replacing in r but non-replacing in l.

In the second stage one rule $l \to r$ (activating a position p) created in stage 1 (or stage 2) is replaced by a set of rules, such that each lazy *innermost* position q of l may be activated by rules where p is non-replacing in both sides, and another set of rules which activate p where q is replacing in both sides (thus such a positions q must be activated before p). This construction is repeated until the rules obtained do not have any lazy (non-variable) positions. We would like to point out that as we consider *innermost* positions of terms in stage one and one step of stage two in our transformation, the outermost lazy positions of the initial rules of the lazy system are dealt with last. So these are the positions which may be activated first in reduction sequences of the transformed system.

In the third phase of the transformation for each rule of the original lazy system one active rewrite rule is created which uses the new extended signature.

Definition 3.6 *Let $\mathcal{R} = (\Sigma, R)$ be a TRS with replacement map μ. The transformed system $\widetilde{\mathcal{R}} = (\widetilde{\Sigma}, \widetilde{R})$ with $\widetilde{\mu}$ is constructed in the following three stages.*

1 **Generation of Initial Activation Rules.** *The transformed signature $\overline{\Sigma} \supseteq \Sigma$ and the set $A(l)$ for every rule $l \to r \in R$ are defined as the least sets satisfying*

$$l[x]_{p.i} \to l'[x]_{p.i} \in A(l) \text{ if } p.i \in \mathcal{I}(l) \text{ and } l' = l[f_i(l|_{p.1}, ..., l|_{p.n})]_p \qquad (1)$$
$$\wedge\ f_i \in \widetilde{\Sigma}$$
$$\wedge\ orig(g) = orig(h) \wedge \widetilde{\mu}(g) = \widetilde{\mu}(h) \Rightarrow g = h \text{ for all } g, h \in \widetilde{\Sigma}$$

where $\widetilde{\mu}$ is defined by $\widetilde{\mu}(f) = \mu(f)$ for all $f \in \Sigma$ and $\widetilde{\mu}(f_i) = \mu(orig(f_i)) \cup \{i\}$ if f_i was introduced in (1). Then we have $\widetilde{R} := \bigcup_{l \to r \in R} A(l)$.

2 **Saturation of Activation Rules.**

2.a **Processing one Activation Rule.** *Let* $\widetilde{R} = A(l_1) \cup ... \cup A(l_n)$ *and let* $l \to r \in A(l_i)$ *for some* $i \in \{1, ..., n\}$ *such that* $\mathcal{I}(l)$ *is not empty. Then we modify the set* $A(l_i)$ *in the following way*

$$A(l_i) = A(l_i) - \{l \to r\} \cup \{l[x]_{p.i} \to l'[x]_{p.i}\} \cup \{l' \to r'\}$$

for all $p.i \in \mathcal{I}(l)$ *where* $l' = l[f_i(l|_{p.1}, ..., l|_{p.n})]_p$ *and* $r' = r[f_i'(r|_{p.1}, ..., r|_{p.n})]_p$. *If there is no* $g \in \widetilde{\Sigma}$ *with* $orig(g) = orig(f_i)$ *and* $\widetilde{\mu}(g) = \widetilde{\mu}(root(l|_p)) \cup \{i\}$, *then* $\widetilde{\Sigma} = \widetilde{\Sigma} \cup \{f_i\}$ *and* $\widetilde{\mu}(f) = \widetilde{\mu}(root(l|_p)) \cup \{i\}$, *otherwise* $f_i = g$. *Analogously, if there is no* $g \in \widetilde{\Sigma}$ *with* $orig(g) = orig(f_i')$ *and* $\widetilde{\mu}(g) = \widetilde{\mu}(root(r|_p)) \cup \{i\}$, *then* $\widetilde{\Sigma} = \widetilde{\Sigma} \cup \{f_i'\}$ *and* $\widetilde{\mu}(f_i') = \widetilde{\mu}(root(r|_p)) \cup \{i\}$, *otherwise* $f_i' = g$.

$$\widetilde{R} := \bigcup_{l \to r \in R} A(l)$$

2.b **Iteration.** *Step 2.a is iterated until for all rules* $l \to r$ *of* $\widetilde{R}$ *we have that* $\mathcal{I}(l) = \emptyset$.

3 **Generation of Active Rewrite Rules.** *For each rule* $l \to r \in R$ *we add one active rewrite rule to* $\widetilde{R}$ *as follows. For every position* $p \in Pos_\Sigma(l)$, *we consider the set*

$$Symb(p, l) = \{root(r'|_p) \mid l' \to r' \in A(l) \wedge p \in Pos_\Sigma(r')\}.$$

The function symbol which is least restrictive in this set (i.e. the maximal element of $\widetilde{\mu}(f)$ *w.r.t. the subset relation of all* $f \in Symb(p, l)$*) is unique (cf. Proposition 3.9). We write* $maxSymb(p, l)$. *Then, we set*

$$\widetilde{R} := \widetilde{R} \cup \bigcup_{l \to r \in R} l'' \to r$$

where l'' *is given by* $Pos(l) = Pos(l'')$, $root(l''|_p) = maxSymb(p, l)$ *for all* $p \in Pos_\Sigma(l)$ *and* $root(l''|_p) = root(l|_p)$ *for all* $p \in Pos_V(l)$. *The signature of the transformed system is not altered in this stage.*

We have the following important properties of the transformation.

Proposition 3.7 *Let* $\mathcal{R}$ *be a TRS with replacement map* μ *and let* $\widetilde{\mathcal{R}} = (\widetilde{\Sigma}, \widetilde{R})$ *be the transformed system with replacement map* $\widetilde{\mu}$. *For* $f, g \in \widetilde{\Sigma}$

$$orig(f) = orig(g) \wedge \widetilde{\mu}(f) = \widetilde{\mu}(g) \Rightarrow f = g$$

Proposition 3.8 *The transformation of Definition 3.6 terminates and yields a finite transformed system for every TRS* $\mathcal{R}$ *and every replacement map* μ.

Proposition 3.9 *Let* $\mathcal{R}$ *be a TRS with replacement map* μ. *Let* $\widetilde{\mathcal{R}}$ *and* $\widetilde{\mu}$ *be the TRS (resp. replacement map) obtained after stages* 1 *and* 2 *of the transformation of Definition 3.6. Then the symbol* $maxSymb(p, l)$ *is unique for every rule* $l \to r \in R$ *and every* $p \in Pos_\Sigma(l)$.

Example 3.10 *Consider the TRS from Example 3.3*

$$f(g(a), a) \to a \qquad b \to f(g(c), b)$$

with a replacement map μ, s.t. $\mu(f) = \{1\}$ and $\mu(g) = \emptyset$. In the first stage of the transformation we have $\mathcal{I}(l_1) = \{1.1, 2\}$ and the following two initial activation *rules are added.*

$$f(g(x), a) \to f(g_1(x), a) \qquad f(g(a), x) \to f_2(g(a), x)$$

with $\widetilde{\mu}(g_1) = \{1\}$ and $\widetilde{\mu}(f_2) = \{1, 2\}$. In step 2.a, the first of these rules is replaced by

$$f(g(x), y) \to f_2(g(x), y) \qquad f_2(g(x), a) \to f_2(g_1(x), a)$$

and in the second iteration the second rule is replaced by

$$f(g(x), y) \to f(g_1(x), y) \qquad f(g_1(a), x) \to f_2(g_1(a), x).$$

Finally, the following active rewrite rules *are added:*

$$f_2(g_1(a), a) \to a \qquad b \to f(g(c), b).$$

Hence, the system $\widetilde{\mathcal{R}}$ is

$$\begin{array}{ll} f(g(x), y) \to f_2(g(x), y) & f_2(g(x), a) \to f_2(g_1(x), a) \\ f(g(x), y) \to f(g_1(x), y) & f(g_1(a), x) \to f_2(g_1(a), x) \\ f_2(g_1(a), a) \to a & b \to f(g(c), b) \end{array}$$

with $\widetilde{\mu}(f) = \widetilde{\mu}(g_1) = \{1\}$, $\widetilde{\mu}(f_2) = \{1, 2\}$ and $\widetilde{\mu}(g) = \emptyset$. $\widetilde{\mathcal{R}}$ is not $\widetilde{\mu}$-terminating:

$$\underline{b} \to_{\widetilde{\mu}} \underline{f(g(c), b)} \to_{\widetilde{\mu}} f_2(g(c), \underline{b}) \to_{\widetilde{\mu}} f_2(g(c), \underline{f(g(c), b)}) \to_{\widetilde{\mu}} \dots$$

The rest of the paper is concerned with the proof of soundness and completeness of the transformation of Definition 3.6 w.r.t. termination. First, we will deal with the simpler case of completeness.

Theorem 3.11 *Let $\mathcal{R} = (\Sigma, R)$ be a left-linear TRS with replacement map μ; and let $\widetilde{\mathcal{R}} = (\widetilde{\Sigma}, \widetilde{R})$, $\widetilde{\mu}$ be the transformed system (resp. replacement map) according to Definition 3.6. If $\mathcal{R}$ is $LR(\mu)$-terminating, then $\widetilde{\mathcal{R}}$ is $\widetilde{\mu}$-terminating.*

Proof. We will prove the result indirectly by showing that every infinite $\widetilde{\mathcal{R}}_{\widetilde{\mu}}$-derivation implies the existence of an infinite lazy $\mathcal{R}$-derivation. Assume there is an infinite $\widetilde{\mathcal{R}}_{\widetilde{\mu}}$-sequence starting from a term t. Then we construct an infinite lazy reduction sequence starting from the labelled term t' defined by

$$Pos(t) = Pos(t') \wedge \forall p \in Pos(t) : (orig(root(t|_p)) = root(erase(t'|_p)) \wedge t'|_p \text{ is eager iff } label_{\widetilde{\mu}}(t)|_p \text{ is eager}).$$

In this case we write $t' \approx t$. Note that t' is labelled canonically or more liberally as $\mu(orig(f)) \subseteq \mu(f)$ for all $f \in \widetilde{\Sigma}$. Now consider a $\widetilde{\mu}$-step $t \to_{\widetilde{\mu}} s$ and a labelled term t' with $t' \approx t$. We will prove that there is a labelled term s', such that $t' \overset{LR}{\to}_{\mu} s'$ and $s' \approx s$. We make a case distinction on the type of $\widetilde{\mu}$-step.

(i) First assume the step is an activation step. Then there is an activation rule $l' \to l''$ in $\widetilde{R}$ which can be applied to t. This activation rule stems from a rule

$l \to r \in R$, and we have that $orig(root(l'|_p)) = root(l|_p)$ for all non-variable positions p of l'. Furthermore, all variable positions of l' which are non-variable in l are lazy in $label_{\widetilde{\mu}}(l')$ and thus in t'. Hence, l matches modulo laziness t' and the same position as in t can be activated yielding s' with $s' \approx s$ (note that the active positions of t' are exactly the replacing positions of t).

(ii) If the step $t \to_{\widetilde{\mu}} s$ is an active rewrite step, a rule $l' \to r$ matches (a subterm of) t. This rule is the transformed version of a rule $l \to r \in R$ with $orig(root(l'|_p)) = root(l|_p)$ for all $p \in Pos(l) = Pos(l')$. Thus, l matches $erase(t')$ and the rule can be applied to t' yielding s' with $s' \approx s$. The reason is that $orig(root(s'|_p)) = root(s|_p)$ for all position of s (note that the right hand-sides of the rules applied to t and t' are identical). Regarding the labels of s' assume that the rewrite steps were performed at a position q (in t and t'). For all positions $o \in Pos(t)$ with $o||q \vee o < q$ we have $s'|_o$ is eager if and only if $label_{\widetilde{\mu}}(s)|_o$ is eager because this has already been the case in t' and t. Furthermore, positions $q.o$ where $o \in Pos(r)$ are eager in s' if and only if they are eager in $label_{\widetilde{\mu}}(s)$ because of the canonical labelling of r inside s'. Finally, positions $p.o$ where $o \notin Pos(r)$ are eager in s' if and only if they are eager in $label_{\widetilde{\mu}}(s)$, because a proper superterm of each term $s'|_{p.o}$ occurred already in t' and thus, if an eager position of s' had not been eager in $label_{\widetilde{\mu}}(s)$ (or vice versa), then this would be a contradiction to $t' \approx t$. □

The soundness proof is similar to the completeness proof in the sense that given an infinite lazy reduction sequence, we are going to construct a corresponding infinite reduction sequence in the transformed system. So assume there is an infinite lazy reduction sequence in a TRS $\mathcal{R}$ with replacement map μ. The first observation is that every lazy reduction sequence naturally corresponds to a context-free $\to_{\mathcal{R}}$ sequence, which performs the active rewrite steps of the lazy reduction sequence.

We will construct a $\to_{\widetilde{\mu}}$-reduction sequence in the transformed system $\widetilde{\mathcal{R}}$, that corresponds to a context-free $\to_{\mathcal{R}}$-sequence. Terms in the context-free $\to_{\mathcal{R}}$-sequence and terms in the corresponding $\widetilde{\mu}$-sequence are in a special relationship.

Definition 3.12 *Let $\mathcal{R} = (\Sigma, R)$ be a TRS, μ a replacement map and let $s, t \in \mathcal{T}(\Sigma, V)$ be two terms. Abusing notation we write $s \to^*_{\mu^c} t$ if and only if*

(i) for all positions $p \in Pos^\mu(t)$ we have $root(t|_p) = root(s|_p)$, and

(ii) for all minimal positions $q \in Pos(t) \setminus Pos^\mu(t)$ we have $s|_q \to^*_\mu s'$ and $s' \to^*_{\mu^c} t|_q$.

The idea behind $\to^*_{\mu^c}$ is that context-free reduction steps which occur at positions that are in the replacing part of the simulating term should be simulated, thus the replacing parts of two terms s and t with $s \to^*_{\mu^c} t$ must be entirely equal. On the other hand, context-free steps that occur at positions which are forbidden in the simulating term are ignored. Yet, if the forbidden subterm in which they occur eventually gets activated, then these steps may still be simulated.

As minimal non-replacing positions in a term are always strictly below the root, the recursive description of $\to^*_{\mu^c}$ in Definition 3.12 is well-defined.

We have $s = t \Rightarrow s \to^*_{\mu^c} t$. Figure 1 illustrates the correspondence between a lazy

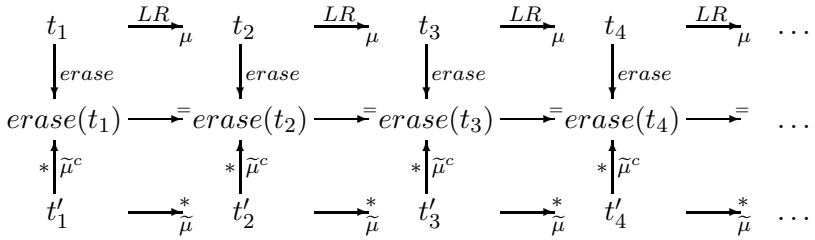


Fig. 1. Relation between the various rewrite sequences occurring in the soundness proof.

reduction sequence, the corresponding context-free one, and the $\to_{\widetilde{\mu}}$-sequence. Note that if the lazy reduction sequence is infinite, then there are infinitely many non-empty steps in the context-free reduction sequence, as every labelled term admits only finitely many activation steps.

In the first part of the soundness proof we show the existence of a $\to_{\widetilde{\mu}}$-sequence of the shape as in Figure 1.

The next lemma provides a criterion for the existence of an activation rule in the transformed system, that is able to activate a certain position in a term t over the new signature.

Lemma 3.13 *Let $(\mathcal{R} = (\Sigma, R), \mu)$ be a TRS with replacement map and let $(\widetilde{\mathcal{R}} = (\widetilde{\Sigma}, \widetilde{R}), \widetilde{\mu})$ be the system obtained by the transformation of Definition 3.6. Let $t \in \mathcal{T}(\widetilde{\Sigma}, V)$ be a term and $\alpha\colon l \to r \in R$ a rewrite rule of the original TRS, such that the following preconditions hold.*

(i) For all replacing *positions p in t with $p \in Pos_\Sigma(l)$: $orig(root(t|_p)) = root(l|_p)$.*

(ii) For all positions $p.i$ that are variable positions in l we have that $t|_q \in \mathcal{T}(\Sigma, V)$ for some $q \leq p$.

Then, every position q, which is minimal non-replacing in t and non-variable in l, can be activated (i.e. we have $t \to_{\widetilde{\mu}} t'$ such that q is $\widetilde{\mu}$-replacing in t' and $orig(root(t|_p)) = orig(root(t'|_p))$ for all $p \in Pos(t)$).

The next lemma establishes the relationship between a context-free reduction sequence and a corresponding $\to_{\widetilde{\mu}}$ reduction of Figure 1.

Lemma 3.14 *Let $(\mathcal{R} = (\Sigma, R), \mu)$ be a TRS with replacement map and let $(\widetilde{\mathcal{R}} = (\widetilde{\Sigma}, \widetilde{R}), \widetilde{\mu})$ be the system obtained by the transformation of Definition 3.6. Let s and t be terms from $\mathcal{T}(\Sigma, V)$, such that $s \to^*_{\widetilde{\mu}^c} t$. If $t \xrightarrow{p} t^*$ (with a rule $l \to r$) and $p \in Pos^{\widetilde{\mu}}(s)$, then $s \to^+_{\widetilde{\mu}} s^*$ and $s^* \to^*_{\widetilde{\mu}^c} t^*$. Otherwise, if $t \xrightarrow{p} t^*$ and $p \notin Pos^{\widetilde{\mu}}(s)$, then $s \to^*_{\widetilde{\mu}} s^*$ and $s^* \to^*_{\widetilde{\mu}^c} t^*$.*

Unfortunately, the last lemma and the correspondence of lazy, context-free and $\to_{\widetilde{\mu}}$-reduction sequences of Figure 1 is not sufficient to prove the existence of an infinite $\to_{\widetilde{\mu}}$-sequence in the presence of an infinite lazy reduction sequence, since there may be only finitely many non-empty $\to_{\widetilde{\mu}}$-reductions in the simulating sequence.

Example 3.15 *Consider the TRS $\mathcal{R}$*

$$a \to f(a) \qquad f(b) \to b$$

with a replacement map $\mu(f) = \emptyset$*. The transformed system* $\widetilde{\mathcal{R}}$ *is*

$$a \to f(a) \qquad f(x) \to f_1(x) \qquad f_1(b) \to b$$

with $\widetilde{\mu}(f) = \emptyset$, $\widetilde{\mu}(f_1) = \{1\}$*. We have the following lazy reduction sequence*

$$\underline{a^e} \overset{LR}{\to}_\mu f^e(\underline{a^l}) \overset{LR}{\to}_\mu f^e(\underline{a^e}) \overset{LR}{\to}_\mu \dots$$

which corresponds to the context-free sequence

$$a \to f(a) \to f(f(a)) \to \dots.$$

Consider a corresponding sequence in the system $\widetilde{\mathcal{R}}$*,*

$$a \to_{\widetilde{\mu}} f(a).$$

Then we could activate a *in* $f(a)$ *according to rule* 2 *of the transformed system. However, it is a priori not clear whether such a step should be performed when trying to simulate an infinite reduction sequence. The following example illustrates the potential problems.*

Example 3.16 *Consider the non-terminating TRS* $\mathcal{R}$

$$f(g(x)) \to f(g(x)) \qquad g(a) \to g(b) \qquad a \to c$$

with a replacement map $\mu(f) = \{1\}$ *and* $\mu(g) = \emptyset$*. The transformed system* $\widetilde{\mathcal{R}}$ *is*

$$f(g(x)) \to f(g(x)) \quad g(x) \to g_1(x)$$
$$g_1(a) \to g(b) \qquad a \to c$$

with $\widetilde{\mu}(f) = \widetilde{\mu}(g_1) = \{1\}$ *and* $\widetilde{\mu}(g) = \emptyset$*. Consider the following context-free reduction sequence.*

$$f(g(a)) \to f(g(c)) \to f(g(c)) \to \dots$$

If we activate position 1.1 *in* $f(g(a))$ *in the simulating* $\to_{\widetilde{\mu}}$*-sequence, we cannot further simulate the sequence, i.e. we get*

$$f(g(a)) \to_{\widetilde{\mu}} f(g_1(a)) \to_{\widetilde{\mu}} f(g_1(c)),$$

but the term $f(g_1(c))$ *is a* $\to_{\widetilde{\mu}}$*-normal form.*

The crucial difference why the activation of a subterm is essential in Example 3.15 and unnecessary in Example 3.16 is that in the former example the activated subterm itself initiates an infinite lazy reduction sequence. This observation will be used in the second part of the soundness proof (cf. Theorem 3.22).

With the following definition we intend to identify labelled terms in an infinite lazy reduction sequence with non-terminating subterms that have possibly been activated. For such terms t, the predicate $mininf(t)$ does not hold.

Definition 3.17 *Let* Σ *be a signature and* μ *be a replacement map for* Σ*. A labelled term* t *is said to be* minimal non-terminating *if it admits an infinite lazy*

reduction sequence and for each eager labelled proper subterm $t|_p$ of t, either $t|_p$ does not initiate an infinite lazy rewrite sequence, or position p is eager in the term $label_\mu(erase(t))$. We write $mininf(t)$ if t has this property.

Definition 3.18 *Let $\mathcal{R} = (\Sigma, R)$ be a TRS with replacement map μ. Let t be a labelled term t and $t \overset{LR}{\to}_\mu s$ be an activation step. This activation step is called* inf-activating *(thus it is an* inf-activation step*) if and only if $mininf(t)$ but* not *$mininf(s)$.*

It is easy to see that whenever $mininf(t)$ holds for a labelled term t, there is no active position $p \in Act(t)$ which is non-active in $label_\mu(erase(t))$, such that $t|_p$ initiates an infinite lazy reduction sequence.

In the second part of the soundness proof we will show that each infinite lazy reduction sequence contains either an inf-activation step or an active rewrite step $s \overset{LR}{\to}_\mu t$ at position p, such that position p is μ-replacing in $erase(s)$. Furthermore, such steps result in non-empty simulations by the $\to_{\widetilde{\mu}}$-sequence.

Lemma 3.19 *Let $\mathcal{R} = (\Sigma, R)$ be a TRS with replacement map μ. Let t be a labelled term satisfying $mininf(t)$. Then we have:*

(i) If $t \overset{LR}{\to}_\mu s$ with an inf-activation step at position q_1 activating position q_2 and $q_1 < p \leq q_2$ is the maximal (w.r.t. $\leq$) eager position in s which does initiate an infinite reduction sequence s.t. $t|_p$ does not, then we have $mininf(s|_p)$.

(ii) If $t \overset{LR}{\to}_\mu s$ with any other step than in (i) (i.e. activation steps which are not inf-activating, or active rewrite steps), then $mininf(s)$.

Lemma 3.20 *Let $\mathcal{R} = (\Sigma, R)$ be a TRS with a replacement map μ. Let $t \in \mathcal{T}(\Sigma, V)$ be an unlabelled term. If t initiates an infinite context-free reduction sequence with infinitely many root reduction steps, then a labelled term t' initiates an infinite lazy reduction sequence if $erase(t') = t$ and t' has an eager root label.*

The next lemma characterizes infinite lazy reduction sequences starting from minimal non-terminating labelled terms. It states that in such an infinite lazy reduction sequence after finitely many steps there is either an active rewrite step $s_i \overset{LR}{\to}_\mu s_{i+1}$ at some position p which is active in $label_\mu(erase(s_i))$ or there is an inf-activation step. We already proved in Lemma 3.14 that active rewrite steps at such positions can be simulated by a non-empty sequence in the transformed system (remember that the active rewrite steps of a lazy reduction sequence correspond to a context-free derivation). In Theorem 3.22 we will prove that the same is true for inf-activation steps.

Lemma 3.21 *Let $\mathcal{R} = (\Sigma, R)$ be a TRS with a replacement map μ. Let t_0 be a labelled term with the property $mininf(t_0)$. Let $P : t_0 \overset{LR}{\to}_\mu t_1 \overset{LR}{\to}_\mu \ldots \overset{LR}{\to}_\mu t_n \overset{LR}{\to}_\mu \ldots$ be an infinite lazy reduction sequence starting from t_0. Then, either there is an active rewrite step $t_i \overset{LR}{\to}_\mu t_{i+1}$ at position p, where p is active in $label_\mu(erase(t_i))$, or there is an* inf-activation *step in P.*

Theorem 3.22 *Let $(\mathcal{R} = (\Sigma, R), \mu)$ be a left-linear TRS with replacement map and*

let $(\widetilde{\mathcal{R}} = (\widetilde{\Sigma}, \widetilde{R}), \widetilde{\mu})$ *the system obtained by the transformation of Definition 3.6. If* $\widetilde{\mathcal{R}}$ *is* $\widetilde{\mu}$*-terminating, then* $\mathcal{R}$ *is* $LR(\mu)$*-terminating.*

Proof. We will show that the existence of an infinite lazy reduction sequence $P : t_0 \overset{LR}{\to}_\mu t_1 \overset{LR}{\to}_\mu \dots$ (where t_0 is canonically or more liberally labelled) implies the existence of an infinite reduction sequence in the transformed system. The following invariant will be maintained for every labelled term t_i on an infinite reduction sequence P. Let $s_0 \to_{\widetilde{\mu}} s_1 \to_{\widetilde{\mu}} \dots$ be the simulating reduction sequence we are going to construct:

There is a s_j such that

$$s_j|_o \to^*_{\widetilde{\mu}^c} erase(t_i|_o) \wedge mininf(t_i|_o)$$

and position o is $\widetilde{\mu}$-replacing in s_j and active in t_i. Furthermore, $t_i|_o$ is at least as eager as its canonically labelled version (i.e. whenever $label_\mu(erase(t_i|_o))$ has an eager label at some position q, then the label of $t_i|_o$ is eager at that position, too). Note that the latter condition is trivially fullfilled by all terms t_i in P, and thus by all subterms as no "deactivations" are possible in lazy rewriting and active rewrite steps only introduce canonically labelled terms.

We show that a finite subsequence of P implies the existence of a non-empty reduction sequence in the transformed system which preserves the invariant. As each term $t_i|_o$ itself initiates an infinite lazy reduction sequence this suffices to show that there is an infinite reduction sequence in the transformed system.

In order to apply Lemma 3.21 we assume $mininf(t_0)$. This minimality constraint can be satisfied, as w.l.o.g. we can find a t_0 such that *each* proper subterm of t_0 with an eager label does not initiate an infinite lazy reduction sequence.

The infinite reduction sequence we are going to construct in the transformed system starts with the term $s_0 = erase(t_0)$. We have $s_0 \to^*_{\widetilde{\mu}^c} erase(t_0)$.

Lemma 3.21 states that in the lazy reduction sequence starting from t_0 there is either an active rewrite step $t_i \overset{LR}{\to}_\mu t_{i+1}$ at position p such that p is active in $label_\mu(erase(t_i))$, or there is an inf-activation step. Let $t_j \overset{LR}{\to}_\mu t_{j+1}$ be the first step, which is of one of the two kinds.

The goal is to show that the reduction sequence $t_0 \overset{LR}{\to}^*_\mu t_{j+1}$ can be simulated by a sequence $s_0 \to^+_{\widetilde{\mu}} s_i$ such that $s_i|_o \to^*_{\widetilde{\mu}^c} erase(t_{j+1|_o})$ and $mininf(t_{j+1}|_o)$ holds for some position o which is active in t_{j+1} and replacing in s_i. We make a case distinction on whether the step $t_j \overset{LR}{\to}_\mu t_{j+1}$ is an active rewrite step or an *inf-activation* step.

(i) Assume the step $t_j \overset{LR}{\to}_\mu t_{j+1}$ is an active rewrite step at position p such that p is active in $label_\mu(erase(t_j))$. We have (according to Lemma 3.14) $s_0 \to^*_{\widetilde{\mu}} s_i$ and $s_i \to^*_{\widetilde{\mu}^c} erase(t_j)$. If position p is active in $label_\mu(erase(t_j))$ then p is replacing in s_i (note that $s_i \in \mathcal{T}(\Sigma, V)$). Thus, with Lemma 3.14 we have $s_0 \to^*_{\widetilde{\mu}} s_i \to^+_{\widetilde{\mu}} s_{i+1}$ and $s_{i+1} \to^*_{\widetilde{\mu}^c} erase(t_{j+1})$. Furthermore, we have $mininf(t_{j+1})$ according to Lemma 3.19.

(ii) Assume the step $t_j \overset{LR}{\to}_\mu t_{j+1}$ is an inf-activation step. Again by Lemma 3.14

we have $s_0 \to_{\widetilde{\mu}}^* s_i$ and $s_i \to_{\widetilde{\mu}^c}^* erase(t_j)$ ($s_i \in \mathcal{T}(\Sigma, V)$). The matching modulo laziness of a rule with t_j was established at a position q_{inf} which is active in $label_\mu(erase(t_j))$. The reason is that otherwise in t_j there were a non-terminating active subterm which is non-active in $label_\mu(erase(t_j))$. Thus, $mininf(t_j)$ would not hold, which contradicts Lemma 3.19.

The fact that the activation step from t_j to t_{j+1} is inf-activating implies that there is a unique maximal active subterm $t_{j+1}|_p$ of t_{j+1} which is non-active in $label_\mu(erase(t_{j+1}))$ and initiates an infinite lazy reduction sequence. For this position p we have $p \leq q$ where q is the position that is activated in the inf-activation step: If we had $p > q$ or $p||q$, then $t_j|_p = t_{j+1}|_p$. Furthermore, as $label_\mu(erase(t_j)) = label_\mu(erase(t_{j+1}))$, this would contradict $mininf(t_j)$.

Note that, since the position q_{inf} where the matching modulo laziness was established in t_j is active in $label_\mu(erase(t_j))$, we have that $q_{inf} < p \leq q$.

In the simulating sequence we will activate position p in the term s_i. We note that p is non-replacing in s_i (as it is non-active in $label_\mu(erase(t_j))$), but it is not necessarily minimal non-replacing. Thus, in order to activate position p in s_i, we possibly need to activate positions $o < p$ in s_i first.

Let $o < p$ be the minimal non-replacing position in s_i. According to Lemma 3.13 we can activate o in s_i yielding s_i'. Note that as t_j is at least as eager as $label_\mu(erase(t_j))$, we have $orig(root(s_i|_{q_{inf}.q'})) = root(erase(t_j|_{q_{inf}.q'}))$ for every replacing position $q_{inf}.q'$ of s_i and $root(erase(t_j|_{q_{inf}.q'})) = l|_{q'}$ for some rule $l \to r$. Then, as $s_i \to_{\widetilde{\mu}^c}^* erase(t_j)$, we have $s_i' \overset{\geq o}{\to}_{\widetilde{\mu}}^* s_i''$ such that $s_i''|_o \to_{\widetilde{\mu}^c}^* erase(t_j|_o)$ according to Definition 3.12. Position o is replacing in s_i''. If there is still a non-replacing position $o' < p$ in s_i'', it is again activated and s_i'' is reduced to a term s_i''' such that $s_i'''|_{o'} \to_{\widetilde{\mu}^c}^* erase(t_j|_{o'})$. This construction is repeated until position p is replacing in a term s_i^* and we have $s_i^*|_p \to_{\widetilde{\mu}^c}^* erase(t_j|_p)$.

Note that $s_i^*|_p$ is a term over the original signature Σ, so it does not contain any function symbols introduced by the transformation. Clearly, we have that $s_i^*|_p \to_{\widetilde{\mu}^c}^* erase(t_{j+1}|_p)$, since $erase(t_j) = erase(t_{j+1})$. Finally, according to Lemma 3.19 we have $mininf(t_{j+1}|_p)$.

Now given an infinite lazy reduction sequence P starting from a labelled term t_0 and a term s_0 with $s_0 \to_{\widetilde{\mu}^c}^* erase(t_0)$, we have shown that a *finite* subsequence $t_0 \overset{LR}{\to}_\mu^+ t_i$ of a special shape implies the existence of a nonempty sequence $s_0 \to_{\widetilde{\mu}}^+ s_j$ such that $s_j|_p \to_{\widetilde{\mu}^c}^* erase(t_i|_p)$, where p is active in t_i and replacing in s_j, $mininf(t_i|_p)$ holds and $t_i|_p$ initiates an infinite lazy reduction sequence. Thus, again the infinite lazy sequence starting at $t_i|_p$ has a finite subsequence, that can be simulated by a non-empty reduction sequence in the transformed system. By repeating this construction, we get an infinite reduction sequence in the transformed system starting at s_0. □

4 Discussion

In the proof of Theorem 3.22 we saw that the CSRS obtained by our transformation cannot simulate lazy reduction sequences in a one-to-one fashion. When simulating

infinite lazy reduction sequences, after every inf-activation step in the lazy reduction an entirely new infinite lazy sequence was considered, namely the one initiated by the activated subterm. Thus, the question arises whether we can define a transformation from lazy rewrite systems into context-sensitive ones, such that the transformed system is able to fully simulate the lazy reduction system. We conjecture that this is indeed possible ([Sch07]).

Regarding the size of the transformed system, we have that the number of rules created by our transformation is in general exponentially higher than the number of lazy non-variable subterms in left-hand sides of rules of the lazy system. This leads to the question whether our transformation is practical in proofs of termination of real world lazy TRSs. Currently, it is unclear if termination of the resulting systems can easily be inferred by automated termination tools. However, this would be most desirable in practice. Experiments to answer this question have yet to be performed.

References

[AEGL03] M. Alpuente, S. Escobar, B. Gramlich, and S. Lucas. On-demand strategy annotations revisited. Technical Report DSIC–II/18/03, UPV, Valencia, Spain, July 2003.

[BN98] F. Baader and T. Nipkow. *Term rewriting and all that.* Cambridge University Press, New York, NY, USA, 1998.

[CDE+03] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. The Maude 2.0 system. In R. Nieuwenhuis, ed., *Proc. RTA'03*, LNCS 2706, pp. 76–87. Springer, 2003.

[FKW00] W. Fokkink, J. Kamperman, and P. Walters. Lazy rewriting on eager machinery. *ACM Trans. Program. Lang. Syst.*, 22(1):45–86, 2000.

[FW76] D. P. Friedman and D. S. Wise. CONS should not evaluate its arguments. In S. Michaelson and R. Milner, eds., *Proc. 3rd ICALP*, pp. 257–284. Edinburgh University Press, 1976.

[GL06] B. Gramlich and S. Lucas. Generalizing Newman's Lemma for left-linear rewrite systems. In F. Pfenning, ed., *Proc. RTA'06,*, LNCS 4098, pp. 66–80. Springer, 2006.

[GTSK06] J. Giesl, R. Thiemann, and P. Schneider-Kamp. AProVE 1.2: Automatic termination proofs in the dependency pair framework. In Ulrich Furbach and Natarajan Shankar eds., *Proc. IJCAR'06,* LNCS 4130, pp. 281–286, 2006.

[HMJ76] P. Henderson and J. H. Morris Jr. A lazy evaluator. In *Conference Record of the Third ACM Symp. on Principles of Programming Languages, Atlanta, Georgia, Jan. 1976*, pp. 95–103, 1976.

[Luc98] S. Lucas. Context-sensitive computations in functional and functional logic programs. *Journal of Functional and Logic Programming*, 1998(1), January 1998.

[Luc01] S. Lucas. Termination of on-demand rewriting and termination of OBJ programs. In H. Sondergaard, ed., *Proc. PPDP'01*, pp. 82–93, September 2001. ACM Press, New York.

[Luc02a] S. Lucas. Context-sensitive rewriting strategies. *Inform. and Comput.*, 178(1):294–343, 2002.

[Luc02b] S. Lucas. Lazy rewriting and context-sensitive rewriting. *In M. Hanus, ed., Proc. WFLP'01,* ENTCS 64, Elsevier, 2002.

[Luc06] S. Lucas. Proving termination of context-sensitive rewriting by transformation. *Information and Computation*, 204(1):1782–1846, January 2006.

[Ngu01] Q. H. Nguyen. Compact normalisation trace via lazy rewriting. *Proc. WRS'01,* ENTCS 57, 2001.

[Pv93] M. J. Plasmeijer and M. C. J. D. van Eekelen. *Functional Programming and Parallel Graph Rewriting.* Addison-Wesley, 1993.

[Str89] R. Strandh. Classes of equational programs that compile into efficient machine code. In N. Dershowitz, ed., *Proc. RTA'89*, LNCS 355, pp. 449–461. Springer, April 1989.

[Sch07] F. Schernhammer. On context-sensitive term rewriting. Master's thesis, Vienna University of Technology, Feb. 2007.

Undecidable Properties on Length-Two String Rewriting Systems

Masahiko Sakai[1]

Graduate School of Information Science, Nagoya University, furo-cho Chikusa-ku, Nagoya 4648603 Japan

Yi Wang[2]

Graduate School of Arts and Science, The University of Tokyo, Komaba Meguro-ku, Tokyo 1538914 Japan

Abstract

Length-two string rewriting systems are length preserving string rewriting systems that consist of length-two rules. This paper shows that confluence, termination, left-most termination and right-most termination are undecidable properties for length-two string rewriting systems. This results mean that these properties are undecidable for the class of linear term rewriting systems in which depth-two variables are allowed in both-hand sides of rules.

1 Introduction

Confluence and termination are both generally undecidable for term rewriting systems (TRSs) and for string rewriting systems. Hence several decidable classes have been studied. Confluence is a decidable property for terminating TRSs [11], ground TRSs [14], linear shallow TRSs [6] and shallow right-linear TRSs [7]. Termination is a decidable property for ground TRSs [9], right ground TRSs [3], TRSs that consist of right-ground rules, collapsing rules and shallow right-linear rules [8], and related class of shallow left-linear TRSs [16].

There are also results on undecidable classes. Confluence is an undecidable property for semi-constructor TRSs [12] and flat TRSs [10,13]. Caron showed that termination is an undecidable property for length preserving string rewriting systems [2].

String rewriting systems (SRSs) are said to be length preserving if the left-hand side and the right-hand side of each rule have the same length. Especially

[1] Email: sakai@is.nagoya-u.ac.jp
[2] Email: wangyi@graco.c.u-tokyo.ac.jp

they are length-two systems if all of the lengths are two. In this paper, we show confluence, termination, left-most termination and right-most termination are undecidable properties for length-two SRSs. Firstly we show those for length preserving SRSs by reducing the Post's correspondence problem, which is known to be undecidable, to termination problem and to confluence problem for length preserving SRSs. Then we give a transformation of length preserving SRSs to length-two SRSs that preserves both confluence and termination properties.

The class of length-two SRSs is a subclass of linear term rewriting systems in which depth-two variables are allowed in both-hand sides of rules. Thus the undecidability for these class of term rewriting systems also obtained. In that sence, the undecidability results in this paper shed new light on the borderline between decidability and undecidability for TRSs.

2 Preliminaries

Let Σ be an alphabet. A *string rewrite rule* is a pair of strings $l, r \in \Sigma^*$, denoted by $l \to r$. A finite set of string rewrite rules is called a *string rewriting system* (SRS). A string is called a *redex* if it is the left-hand side of a rule. An SRS $\mathcal{R}$ induces a *rewrite step* relation $\underset{\mathcal{R}}{\to}$ defined as $s \underset{\mathcal{R}}{\to} t$ if there exist $u, v \in \Sigma^*$ and a rule $l \to r$ in $\mathcal{R}$ such that $s = ulv$ and $t = urv$. Especially it is *left-most* (resp. *right-most*) if l is the left-most (resp. right-most) redex in s. We use $\overset{+}{\underset{\mathcal{R}}{\to}}$ for the transitive closure of $\underset{\mathcal{R}}{\to}$ and $\overset{*}{\underset{\mathcal{R}}{\to}}$ for the reflexive-transitive closure of $\underset{\mathcal{R}}{\to}$. We use $\underset{\mathcal{R}}{\leftrightarrow}$ for $\underset{\mathcal{R}}{\leftarrow} \cup \underset{\mathcal{R}}{\to}$. We write $\overset{k}{\underset{\mathcal{R}}{\to}}$ for the relation with k rewrite steps. A (possibly infinite) sequence $s_0 \underset{\mathcal{R}}{\to} s_1 \underset{\mathcal{R}}{\to} \cdots$ is called a reduction sequence.

We say that string s is *terminating* if every reduction sequence starting from s is finite. We say that strings s_1 and s_2 are *joinable* if $s_1 \overset{*}{\underset{\mathcal{R}}{\to}} s \overset{*}{\underset{\mathcal{R}}{\leftarrow}} s_2$ for some s, denoted by $s_1 \downarrow_{\mathcal{R}} s_2$. A string s is *confluent* if $s_1 \downarrow_{\mathcal{R}} s_2$ for any $s_1 \overset{*}{\underset{\mathcal{R}}{\leftarrow}} s \overset{*}{\underset{\mathcal{R}}{\to}} s_2$. An SRS $\mathcal{R}$ is *confluent* (*terminating*) if all strings are confluent (terminating).

In this paper, the notation $|u|$ represents the length of string u. The notation $\underbrace{a \cdots a}_{m}$ denotes the string that consists of m symbols of a. We refer $\{r \to l \mid l \to r \in \mathcal{R}\}$ by $\mathcal{R}^{-1}$.

Now we recall Post's correspondence problem (PCP), which is known to be undecidable.

Definition 2.1 *An instance of PCP is a finite set $P \subseteq \mathcal{A}^* \times \mathcal{A}^*$ of finite pairs of non-empty strings over an alphabet $\mathcal{A}$ with at least two symbols. A solution of P is a string w such that*

$$w = u_1 \cdots u_k = v_1 \cdots v_k$$

for some $(u_i, v_i) \in P$. The Post's correspondence problem (PCP) is a problem to decide whether such a solution exists or not.

Example 2.2 *The set $P = \{(ab, a), (a, ba)\}$ is an instance of PCP over $\{a, b\}$. It has a solution $aba = u_1u_2 = v_1v_2$ with $(u_1, v_1) = (ab, a)$, $(u_2, v_2) = (a, ba)$.*

Theorem 2.3 ([15]) *PCP is undecidable.*

3 Length preserving SRSs and undecidability of their termination

Definition 3.1 *An SRS $\mathcal{R}$ is said to be length preserving if $|l| = |r|$ for every rule $l \to r$ in $\mathcal{R}$.*

Since there is a finite number of rules, the number of different symbols appearing in the rules is finite, and fixed for them. Hence the number of strings with a given length is also finite. Thus the decidability of the following problems for length preserving SRSs trivially follows.

(i) *Reachability problem*: problem to decide $s \xrightarrow[\mathcal{R}]{*} t$ for given strings s and t and a SRS $\mathcal{R}$.

(ii) *String-confluence problem*: problem to decide confluence of s for a given string s and a SRS $\mathcal{R}$.

(iii) *String-termination problem*: problem to decide termination of s for a given string s and a SRS $\mathcal{R}$.

In this section we argue about the undecidability of termination, right-most termination and left-most termination for length preserving SRSs. As stated in the introduction, Caron showed the undecidability in [2]. Moreover the proof works also for right-most termination and left-most termination because there is only one redex in each string that corresponds to a correct automata configuration. Nevertheless we give an alternative proof from the following reasons:

- Caron's proof composed of two stages; the first state gives an algorithm that reduce PCP into the unform halting problem for linear-bounded automata and the second stage gives an algorithm reducing the uniform halting problem into the termination problem for length preserving SRSs. On the other hand, we give a proof by reducing the Post's correspondence problem into termination problem of SRSs directly.
- The SRS $\mathcal{T}_P$ given in this section is rather straightforward and easy to understand. This helps the understanding of SRS $\mathcal{C}_P$ given in the next section, which is more difficult although it is just a variant of $\mathcal{T}_P$.

As a preparation for giving the transformation, we introduce a kind of null symbol $-$ and an equal length representation of each pair in instances of PCP. Let $P = \{(u_1, v_1), \ldots, (u_n, v_n)\}$ be an instance of PCP over $\mathcal{A}$.

$$\overline{P} = \{(u, v\underbrace{-\cdots-}_{|m|}) \mid (u,v) \in P \text{ and } |u| - |v| = m \geq 0\}$$
$$\cup \{(u\underbrace{-\cdots-}_{|m|}, v) \mid (u,v) \in P \text{ and } |u| - |v| = m < 0\}$$

We write $\overline{\mathcal{A}}$ for $\mathcal{A} \cup \{-\}$. We define an equivalence relation $\sim \subseteq (\overline{\mathcal{A}})^* \times (\overline{\mathcal{A}})^*$ as identity relation with ignoring all null symbols $-$, that is $u \sim v$ if and only if $\hat{u} = \hat{v}$ where $\hat{u}$

and $\hat{v}$ denote the strings obtained from u and v by removing all $-$s respectively.

Example 3.2 *For an instance $P = \{(ab, a), (a, ba)\}$ of PCP, we have $\overline{P} = \{(ab, a-), (a-, ba)\}$. The solution corresponds to $u_1u_2 = ab\ a- \sim a-\ ba = v_1v_2$ for $(u_1, v_1), (u_2, v_2) \in \overline{P}$.*

We use symbols like $0^{\ a}_{b\ a'}_{\ b'}$, where 0 is called the *tag* of the symbol and a is called the *first subscript* of the symbol, b the *second*, a' the *third* and b' the *fourth*. We code the solution of the previous example into $\tilde{0}^{\ a}_{a\ a}\tilde{0}^{\ b}_{-\ b}\tilde{0}^{\ a}_{b\ a}\tilde{0}^{\ -}_{-\ a}$.

For an easy handling of strings that consist of such symbols, we introduce a notation defined as

$$(X_1 \cdots X_k)^{a_1 \cdots a_k}_{b_1 \cdots b_k\ a'_1 \cdots a'_k\ b'_1 \cdots b'_k} = X_1{}^{a_1}_{b_1\ a'_1\ b'_1} \cdots X_n{}^{a_k}_{b_k\ a'_k\ b'_k}.$$

For example the above solution is denoted by $(\tilde{0}0)^{ab}_{a-\ ab\ a-}(\tilde{0}0)^{a-}_{ba\ a-\ ba}$ or $(\tilde{0}0\tilde{0}0)^{aba-}_{a-ba\ aba-\ a-ba}$. Note that the lengths of the strings in those subscripts are the same whenever we use this notation.

The first and the second subscripts keep a candidate of solutions of P in equal length representation and will never be changed by reductions. The third and the fourth subscripts are used as working area for checking whether the candidate is a solution or not.

We relate a solution of the given instance of PCP with a loop in an infinite reduction sequence:

$$\Xi_0(\hat{0}0 \cdots 0)^{u_1}_{v_1\ u_1\ v_1} \cdots (\hat{0}0 \cdots 0)^{u_k}_{v_k\ u_k\ v_k}\Psi_0 \xrightarrow[\mathcal{T}_P]{*} \Xi_2(\hat{2}2 \cdots 2)^{u_1}_{v_1\ w_1\ w_1} \cdots (\hat{2}2 \cdots 2)^{u_k}_{v_k\ w_k\ w_k}\Psi_2$$
$$\xrightarrow[\mathcal{T}_P]{*} \Xi_0(\hat{0}0 \cdots 0)^{u_1}_{v_1\ u_1\ v_1} \cdots (\hat{0}0 \cdots 0)^{u_k}_{v_k\ u_k\ v_k}\Psi_0.$$

(i) The former part checks whether $u_1 \cdots u_k \sim v_1 \cdots v_k$ by using the third and the fourth subscripts as working area.

(ii) The latter part checks whether $(u_1, v_1), \ldots, (u_n, v_n) \in \overline{P}$ and initializes the working area.

Definition 3.3 *Let P be an instance of PCP over $\mathcal{A}$. The SRS $\mathcal{T}_P$ over Σ obtained from P is defined as follows, where individual rules are shown in Figure 1.*

$$\Sigma = \{\Xi_i, \Psi_i \mid i \in \{0, 1, 2\}\} \cup \Sigma_c$$
$$\Sigma_c = \left\{ n^{x_1}_{x_2\ x_3\ x_4}, \hat{n}^{x_1}_{x_2\ x_3\ x_4}, \underline{n}^{x_1}_{x_2\ x_3\ x_4}, \underline{\hat{n}}^{x_1}_{x_2\ x_3\ x_4} \;\middle|\; x_i \in \overline{\mathcal{A}}, n \in \{0, 1, 2\} \right\}$$
$$\mathcal{T}_P = \alpha_1 \cup \beta_1 \cup \gamma_1 \cup \alpha_2 \cup \beta_2 \cup \gamma_2 \cup \delta_2$$

Example 3.4 *Consider the instance $P = \{(a, ba), (ab, a)\}$ of PCP. Rules α_1, β_1 depend on P and the other rules depend only on the alphabet $\mathcal{A}$.*

$$\alpha_1 = \left\{ (\tilde{\underline{1}}2)^{a\ -}_{b\ a}{}^{\ }_{x_1\ y_1}{}^{\ }_{x_2\ y_2}\Psi_2 \to (\tilde{0}0)^{a-}_{ba\ a-\ ba}\Psi_0,\ (\tilde{\underline{1}}2)^{a\ b}_{a\ -}{}^{\ }_{x_1\ y_1}{}^{\ }_{x_2\ y_2}\Psi_2 \to (\tilde{0}0)^{a\ b}_{a-\ a\ b\ a-}\Psi_0 \;\middle|\; x_i, y_i \in \overline{\mathcal{A}} \right\}$$
$$\beta_1 = \left\{ (\tilde{\underline{1}}2)^{a\ -}_{b\ a}{}^{\ }_{x_1\ y_1}{}^{\ }_{x_2\ y_2}\tilde{2}^{z_1}_{z_2\ z_3\ z_4} \to (\tilde{0}0)^{a-}_{ba\ a-\ ba}\tilde{\underline{1}}^{z_1}_{z_2\ z_3\ z_4},\ (\tilde{\underline{1}}2)^{a\ b}_{a\ -}{}^{\ }_{x_1\ y_1}{}^{\ }_{x_2\ y_2}\tilde{2}^{z_1}_{z_2\ z_3\ z_4} \to (\tilde{0}0)^{a\ b}_{a-\ a\ b\ a-}\tilde{\underline{1}}^{z_1}_{z_2\ z_3\ z_4} \;\middle|\; x_i, y_i, z_i \in \overline{\mathcal{A}} \right\}$$

$$
\begin{aligned}
\alpha_1 &= \left\{ (\underline{\tilde{1}}2\cdots 2)^{\ u}_{v\ u'}_{\ \ v'}\Psi_2 \to (\tilde{0}0\cdots 0)^{\ u}_{v\ u}_{\ \ v}\Psi_0 \;\middle|\; (u,v)\in\overline{P}, u',v'\in(\overline{\mathcal{A}})^* \right\} \\
\beta_1 &= \left\{ (\underline{\tilde{1}}2\cdots 2)^{\ u}_{v\ u'}_{\ \ v'}\tilde{2}^{\ x_1}_{x_2\ x_4}_{\ x_3} \to (\tilde{0}0\cdots 0)^{\ u}_{v\ u}_{\ \ v}(\underline{\tilde{1}})^{\ x_1}_{x_2\ x_4}_{\ x_3} \;\middle|\; (u,v)\in\overline{P}, u',v'\in(\overline{\mathcal{A}})^*, x_j\in\overline{\mathcal{A}} \right\} \\
\gamma_1 &= \left\{ \Xi_2\tilde{2}^{\ x_1}_{x_2\ x_4}_{\ x_3} \to \Xi_0\underline{\tilde{1}}^{\ x_1}_{x_2\ x_4}_{\ x_3} \;\middle|\; x_j\in\overline{\mathcal{A}} \right\} \\
\alpha_2 &= \left\{ 0^{\ x_1}_{x_2\ x_3}_{\ x_3}\Psi_0 \to \underline{2}^{\ x_1}_{x_2\ x_3}_{\ x_3}\Psi_2 \;\middle|\; x_j\in\overline{\mathcal{A}} \right\} \\
\beta_2 &= \left\{ 0^{\ x_1\ y_1}_{x_2\ y_2}_{\ x_3\ y_3} \to \underline{2}^{\ x_1\ y_1}_{x_2\ y_2}_{\ x_3\ y_3},\ \tilde{0}^{\ x_1\ y_1}_{x_2\ y_2}_{\ x_3\ y_3} \to \underline{\tilde{2}}^{\ x_1\ y_1}_{x_2\ y_2}_{\ x_3\ y_3},\ 0^{\ x_1\ y_1}_{x_2\ y_2}_{\ x_3\ y_3} \to \underline{\tilde{2}}^{\ x_1\ y_1}_{x_2\ y_2}_{\ x_3\ y_3} \;\middle|\; x_j,y_j\in\overline{\mathcal{A}} \right\} \\
\gamma_2 &= \left\{ \Xi_0\underline{\tilde{2}}^{\ x_1}_{x_2}_{\ x_3} \to \Xi_2\tilde{2}^{\ x_1}_{x_2}_{\ x_3} \;\middle|\; x_j\in\overline{\mathcal{A}} \right\} \\
\delta_2 &= \left\{ X^{\ x_1\ y_1}_{\ -\ \ y_2}_{\ x_4\ y_4} Y^{\ }_{z} \to X^{\ x_1\ y_1}_{\ z\ \ y_2}_{\ x_4\ y_4} Y^{\ }_{-},\ X^{\ x_1\ y_1}_{\ x_2\ \ -}_{\ x_3\ y_3} Y^{\ }_{z} \to X^{\ x_1\ y_1}_{\ x_2\ \ z}_{\ x_3\ y_3} Y^{\ }_{-} \;\middle|\; x_j,y_j\in\overline{\mathcal{A}}, z\in\mathcal{A}, X,Y\in\{0,\tilde{0}\} \right\}
\end{aligned}
$$

Fig. 1. Rules in $\mathcal{T}_P$

$\mathcal{T}_P$ is not terminating since we can construct an infinite reduction sequence. We start with a string $\Xi_0(\tilde{0}0)^{\ ab\ a-}_{a-\ ab}(\tilde{0}0)^{\ a-}_{ba\ a-}\Psi_0$. Rules in δ_2 move null symbols in the third or the fourth subscripts into the tail:

$$
\Xi_0(\tilde{0}0)^{\ ab\ a-}_{a-\ ab}(\tilde{0}0)^{\ a-}_{ba\ a-}\Psi_0 \underset{\delta_2}{\to} \Xi_0(\tilde{0}0)^{\ ab\ a-}_{a-\ ab}(\tilde{0}0)^{\ a-}_{-a\ ab}\Psi_0 \underset{\delta_2}{\to} \Xi_0(\tilde{0}0)^{\ ab\ a-}_{a-\ ab}(\tilde{0}0)^{\ a-}_{a-\ a-}\Psi_0.
$$

Rules in $\alpha_2\cup\beta_2\cup\gamma_2$ check in right-to-left order that the third and the fourth subscripts are the same:

$$
\begin{aligned}
&\Xi_0(\tilde{0}0)^{\ ab\ a-}_{a-\ ab}(\tilde{0}0)^{\ a-}_{ba\ a-}\Psi_0 \underset{\alpha_2}{\to} \Xi_0(\tilde{0}0)^{\ ab\ a-}_{a-\ ab}(\tilde{0}\underline{2})^{\ a-}_{ba\ a-}\Psi_2 \underset{\beta_2}{\to} \Xi_0(\tilde{0}0)^{\ ab\ a-}_{a-\ ab}(\underline{\tilde{2}}2)^{\ a-}_{ba\ a-}\Psi_2 \\
&\underset{\beta_2}{\to} \Xi_0(\tilde{0}\underline{2})^{\ ab\ a-}_{a-\ ab}(\tilde{2}2)^{\ a-}_{ba\ a-}\Psi_2 \underset{\beta_2}{\to} \Xi_0(\underline{\tilde{2}}2)^{\ ab\ a-}_{a-\ ab}(\tilde{2}2)^{\ a-}_{ba\ a-}\Psi_2 \underset{\gamma_2}{\to} \Xi_2(\tilde{2}2)^{\ ab\ a-}_{a-\ ab}(\tilde{2}2)^{\ a-}_{ba\ a-}\Psi_2.
\end{aligned}
$$

Rules in $\gamma_1\cup\beta_1\cup\alpha_1$ check in left-to-right order that the first and the second subscripts consist of pairs in $\overline{P}$ and copy the first subscript to the third and the second to the fourth respectively:

$$
\begin{aligned}
&\Xi_2(\tilde{2}2)^{\ ab\ a-}_{a-\ ab}(\tilde{2}2)^{\ a-}_{ba\ a-}\Psi_2 \underset{\gamma_1}{\to} \Xi_0(\underline{\tilde{1}}2)^{\ ab\ a-}_{a-\ ab}(\tilde{2}2)^{\ a-}_{ba\ a-}\Psi_2 \underset{\beta_1}{\to} \Xi_0(\tilde{0}0)^{\ ab\ a-}_{a-\ ab}(\underline{\tilde{1}}2)^{\ a-}_{ba\ a-}\Psi_2 \\
&\underset{\alpha_1}{\to} \Xi_0(\tilde{0}0)^{\ ab\ a-}_{a-\ ab}(\tilde{0}0)^{\ a-}_{ba\ a-}\Psi_0.
\end{aligned}
$$

Obviously $\mathcal{T}_P$ is length preserving. The proof of the following lemma is found in Section 5.

Lemma 3.5 *Let P be an instance of PCP. Then the following properties are equivalent:*

(i) *P has a solution.*

(ii) *$\mathcal{T}_P$ is not right-most terminating.*

(iii) *$\mathcal{T}_P$ is not left-most terminating.*

(iv) *$\mathcal{T}_P$ is not terminating.*

Theorem 3.6 *Termination, right-most termination and left-most termination are undecidable properties for length preserving SRSs.*

Proof. We assume that termination (right-most termination, left-most termination) of length preserving SRSs is decidable. Then it follows from Lemma 3.5 that PCP is decidable, which contradicts to Theorem 2.3. □

$$
\begin{aligned}
\alpha'_1 &= \left\{ (\tilde{0}0\cdots0)^{\ u}_{v\ u}\Psi_0 \to (\underline{\tilde{1}}1\cdots1)^{\ u}_{u\ v}\Psi_1 \;\middle|\; (u,v)\in\overline{P} \right\} \\
\beta'_1 &= \left\{ (\tilde{0}0\cdots0)^{\ u\ \ x_1}_{v\ u\ x_2}\underline{\tilde{1}}^{\ x_1}_{x_1\ x_2} \to (\underline{\tilde{1}}1\cdots1)^{\ u\ \ x_1}_{u\ v\ x_2}\tilde{1}^{\ x_1}_{x_1\ x_2} \;\middle|\; (u,v)\in\overline{P}, x_i\in\overline{\mathcal{A}} \right\} \\
\gamma'_1 &= \left\{ \Xi_0\underline{\tilde{1}}^{\ x_1}_{x_1\ x_2} \to \Xi_1\tilde{1}^{\ x_1}_{x_1\ x_2} \;\middle|\; x_i\in\overline{\mathcal{A}} \right\} \\
\epsilon_2 &= \left\{ X^{\ x_1\ \ y_1}_{-\ \ x_4}Y^{\ y_1}_{y_2\ y_4} \to X^{\ x_1}_{x_2\ z\ x_4}Y^{\ y_1}_{-\ y_4},\ X^{\ x_1\ \ y_1}_{x_2\ x_3\ -}Y^{\ y_1}_{y_2\ y_3\ z} \to X^{\ x_1\ \ y_1}_{x_2\ x_3\ z}Y^{\ y_1}_{y_2\ y_3\ -} \;\middle|\; x_j,y_j\in\overline{\mathcal{A}}, z\in\mathcal{A}, X,Y\in\{2,\tilde{2}\} \right\}
\end{aligned}
$$

Fig. 2. Rules in $\mathcal{C}_P$

4 Undecidability of confluence for length preserving SRSs

We modify the construction of SRS in the last section. In contrast to the SRS $\mathcal{T}_P$, which works sequentially, the SRS $\mathcal{C}_P$ works in parallel, that is, a solution of a given instance of PCP is related with the following two reduction sequences

$$
\begin{aligned}
&\Xi_0(\tilde{0}0\cdots0)^{\ u_1\ \ u_k}_{v_1\ v_1\ \ v_k}\cdots(\tilde{0}0\cdots0)^{\ u_k}_{u_k\ u_k}\Psi_0 \xrightarrow[\mathcal{C}_P]{*} \Xi_2(\tilde{2}2\cdots2)^{\ u_1\ \ u_k}_{v_1\ w_1\ \ w_k}\cdots(\tilde{2}2\cdots2)^{\ u_k}_{w_k\ w_k}\Psi_2 \\
&\Xi_0(\tilde{0}0\cdots0)^{\ u_1\ \ u_k}_{v_1\ u_1\ \ v_k}\cdots(\tilde{0}0\cdots0)^{\ v_k}_{u_k\ v_k}\Psi_0 \xrightarrow[\mathcal{C}_P]{*} \Xi_1(\tilde{1}1\cdots1)^{\ u_1\ \ u_k}_{v_1\ u_1\ \ v_k}\cdots(\tilde{1}1\cdots1)^{\ v_k}_{u_k\ v_k}\Psi_1
\end{aligned}
$$

that demonstrate its non-confluence.

(i) The former reduction checks whether $u_1\cdots u_k \sim v_1\cdots v_k$ by using the third and the fourth subscripts as working area.

(ii) The latter reduction checks whether $(u_1,v_1),\ldots,(u_n,v_n)\in\overline{P}$ and checks the working area is correctly initialized.

In case of P has no solution, $\mathcal{C}_P$ must be confluent, which makes the design of $\mathcal{C}_P$ difficult.

Definition 4.1 *Let P be an instance of PCP over $\mathcal{A}$. The SRS $\mathcal{C}_P$ over Σ obtained from P is defined as follows:*

$$
\begin{aligned}
\mathcal{C}_P &= \Theta\cup\Phi, \\
\Theta &= \Theta_1\cup\Theta_2, \\
\Theta_1 &= \alpha'_1\cup\beta'_1\cup(\alpha'_1\cup\beta'_1)^{-1}, \\
\Theta_2 &= \alpha_2\cup\beta_2\cup\delta_2\cup\epsilon_2\cup(\alpha_2\cup\beta_2\cup\delta_2\cup\epsilon_2)^{-1}, \\
\Phi &= \gamma'_1\cup\gamma_2
\end{aligned}
$$

where rules α_2, β_2, δ_2 and γ_2 are shown in Figure 1 and the other rules are shown in Figure 2.

Remark that the reductions by Θ-rules are symmetric, that is to say, $s\underset{\Theta}{\to}t$ if and only if $t\underset{\Theta}{\to}s$, which plays an important role to make $\mathcal{C}_P$ confluent if P has no solution.

Example 4.2 *Let $P=\{(a,ba),(ab,a)\}$ be an instance of PCP. Rules α'_1, β'_1 de-*

pends on P and the other rules depend only on the alphabet $\mathcal{A}$.

$$\alpha'_1 = \left\{ (\tilde{0}0)^{a-}_{a-}{}^{ab}_{ba}\Psi_0 \to (\tilde{\underline{1}}1)^{a-}_{a-}{}^{ab}_{ba}\Psi_1,\ (\tilde{0}0)^{ab}_{a-}{}^{ab}_{a-}\Psi_0 \to (\tilde{\underline{1}}1)^{ab}_{a-}{}^{ab}_{a-}\Psi_1 \right\}$$

$$\beta'_1 = \left\{ (\tilde{0}0)^{a-}_{a-}{}^{ba}_{ba}\tilde{\underline{1}} \to (\tilde{\underline{1}}1)^{a-}_{a-}{}^{ba}_{ba}\tilde{1}^{x_1}_{x_2},\ (\tilde{0}0)^{ab}_{a-}{}^{ab}_{a-}\tilde{\underline{1}} \to (\tilde{\underline{1}}1)^{ab}_{a-}{}^{ab}_{a-}\tilde{1}^{x_1}_{x_2} \;\middle|\; x_i \in \overline{\mathcal{A}} \right\}$$

We can show that $\mathcal{C}_P$ is not confluent since we have non-joinable branches.

$$\Xi_0(\tilde{0}0)^{ab}_{a-}{}^{ab}_{a-}(\tilde{0}0)^{a-}_{a-}{}^{ba}_{ba}\Psi_0 \underset{\alpha'_1}{\to} \Xi_0(\tilde{0}0)^{ab}_{a-}{}^{ab}_{a-}(\tilde{\underline{1}}1)^{a-}_{a-}{}^{ba}_{ba}\Psi_1 \underset{\beta'_1}{\to} \Xi_0(\tilde{\underline{1}}1)^{ab}_{a-}{}^{ab}_{a-}(\tilde{1}1)^{a-}_{a-}{}^{ba}_{ba}\Psi_1$$
$$\underset{\gamma'_1}{\to} \Xi_1(\tilde{1}1)^{ab}_{a-}{}^{ab}_{a-}(\tilde{1}1)^{a-}_{a-}{}^{ba}_{ba}\Psi_1 \ ,$$
$$\Xi_0(\tilde{0}0)^{ab}_{a-}{}^{ab}_{a-}(\tilde{0}0)^{a-}_{a-}{}^{ba}_{ba}\Psi_0 \underset{\delta_2\cup\alpha_2\cup\beta_2\cup\gamma_2}{\overset{*}{\to}} \Xi_2(\tilde{2}2)^{ab}_{a-}{}^{ab}_{ab}(\tilde{2}2)^{a-}_{a-}{}^{ba}_{a-}\Psi_2.$$

Note that the detail of the latter sequence is found in Example 3.4.

Obviously $\mathcal{C}_P$ is length preserving. The proof of the following main lemma is found in the next section.

Lemma 4.3 *Let P be an instance of PCP. Then, P has a solution if and only if $\mathcal{C}_P$ is not confluent.*

Theorem 4.4 *Confluence of length preserving SRSs is an undecidable property.*

Proof. We assume that the problem is decidable. Then it follows from Lemma 4.3 that PCP is decidable, which contradicts to Theorem 2.3. □

5 Proofs

Every occurrence of the symbols Ξ_1, Ξ_2 and Ξ_3 in rules are right-most positions in both-hand sides. Moreover, for every rule, Ξ_i appears in the left-hand side if and only if Ξ_j appears in the right-hand side. Hence we can separate any reduction sequence having a symbol Ξ_i into two reduction sequences by cutting each string at the Ξ_j occurrence. Symbols Ψ_i also have the similar property. Therefore the following proposision holds.

Proposition 5.1 *Let $\mathcal{R}$ be $\mathcal{T}_P$ or $\mathcal{C}_P$ obtained from an instance P of PCP. For any $i \in \{0,1,2\}$ and $S_1, S_2, S \in \Sigma^*$, the followings hold:*

(a) If $S_1\Xi_i S_2 \underset{\mathcal{R}}{\to} S$, then $(S = S'_1\Xi_i S_2) \wedge (S_1 \underset{\mathcal{R}}{\to} S'_1)$ or $(S = S_1\Xi_j S'_2) \wedge (\Xi_i S_2 \underset{\mathcal{R}}{\to} \Xi_j S'_2)$ for some $S'_1, S'_2 \in \Sigma^*$ and $j \in \{0,1,2\}$.

(b) If $S_1\Xi_i S_2 \underset{\mathcal{R}}{\overset{*}{\to}} S$, then $S = S'_1S'_2$, $S_1 \underset{\mathcal{R}}{\overset{*}{\to}} S'_1$ and $\Xi_i S_2 \underset{\mathcal{R}}{\overset{*}{\to}} S'_2$ for some $S'_1 \in \Sigma^*$ and non-empty $S'_2 \in \Sigma^*$.

(c) If $S_1\Psi_i S_2 \underset{\mathcal{R}}{\to} S$, then $(S = S'_1\Psi_j S_2) \wedge (S_1\Psi_i \underset{\mathcal{R}}{\to} S'_1\Psi_j)$ or $(S = S_1\Psi_i S'_2) \wedge (S_2 \underset{\mathcal{R}}{\to} S'_2)$ for some $S'_1, S'_2 \in \Sigma^*$ and $j \in \{0,1,2\}$.

(d) If $S_1\Psi_i S_2 \underset{\mathcal{R}}{\overset{*}{\to}} S$, then $S = S'_1S'_2$, $S_1\Psi_i \underset{\mathcal{R}}{\overset{*}{\to}} S'_1$ and $S_2 \underset{\mathcal{R}}{\overset{*}{\to}} S'_2$ for some $S'_2 \in \Sigma^*$ and non-empty $S'_1 \in \Sigma^*$.

Proof. We prove (a). Let $S_1 \Xi_i S_2 \underset{\mathcal{R}}{\rightarrow} S$. The only interesting case is that the redex in the rewrite step contains the displayed symbol Ξ_i. Then one of γ_1-rules, γ_2-rules or γ_1'-rules is applied. From the construction of the rules, we have $S = S_1 \Xi_j S_2'$ and $\Xi_i S_2 \underset{\mathcal{R}}{\rightarrow} \Xi_j S_2'$ for some $S_2' \in \Sigma^*$ and $j \in \{0, 1, 2\}$.

The claim (b) is easily proved by induction on the number k of the rewrite steps in $S_1 \Xi_i S_2 \overset{*}{\underset{\mathcal{R}}{\rightarrow}} S$. For (c) and (d), the proofs are similar to (a) and (b) respectively.□

We say a string over Σ is *normal* if it is in one of the following three forms:

(p1) $\Xi_i \chi$, (p2) $\chi \Psi_j$, (p3) $\Xi_i \chi \Psi_j$,

where $\chi \in (\Sigma_c)^*$, $i, j \in \{0, 1, 2\}$.

We prepare a measure for the proof of the next lemma. For a non-empty string $X_1 \cdots X_n$ over Σ, we define $||X_1 \cdots X_n||$ by the summation of the number of occurrences of Ξ_i symbols in $X_2 \cdots X_n$ and the number of occurrences of Ψ_i symbols in $X_1 \cdots X_{n-1}$.

Lemma 5.2 *Let $\mathcal{R}$ be $\mathcal{T}_P$ or $\mathcal{C}_P$ over Σ obtained from an instance P of PCP. Then $\mathcal{R}$ is confluent (resp. terminating, right-most terminating, left-most terminating) if and only if w is confluent (resp. terminating, right-most terminating, left-most terminating) for every normal $w \in \Sigma^*$.*

Proof. Firstly we prove the termination part of the lemma. Since $\Rightarrow$-direction is trivial, consider $\Leftarrow$-direction.

Let $S_1 \underset{\mathcal{R}}{\rightarrow} S_2 \underset{\mathcal{R}}{\rightarrow} \cdots$ be an infinite reduction sequence such that $||S_1||$ is minimal. We show a contradiction assuming $||S_1|| > 0$. We have two cases that $S_1 = w \Xi_i S'$ and $S_1 = S' \Psi_i w$ for some normal w and some $S' \in \Sigma^*$.

- In the former case that $S_1 = w \Xi_i S'$, we can construct an infinite reduction sequence starting from at least one of w or $\Xi_i S'$ by applying Proposition 5.1(a) infinitely many times, which contradicts to the minimality of S_1.
- In the latter case, we can show a contradiction in similar to the former case by using Proposition 5.1(c).

Secondly we prove the confluence part of the lemma. Since $\Rightarrow$-direction is trivial, consider $\Leftarrow$-direction. We show that every $S_1 \in \Sigma^*$ is confluent by induction on $||S_1||$. If $||S_1|| = 0$, then S_1 is normal and it is confluent from the assumption. If $||S_1|| > 0$, then we have two cases that $S_1 = w_1 \Xi_i S_1'$ and $S_1 = S_1' \Psi_i w_1$ for some normal w_1 and some $S_1' \in \Sigma^*$.

- In the former case, let $S_2 \overset{*}{\underset{\mathcal{R}}{\leftarrow}} w_1 \Xi_i S_1' \overset{*}{\underset{\mathcal{R}}{\rightarrow}} S_3$. By Proposition 5.1(b), we have $S_2 = w_2 S_2'$, $S_3 = w_2 S_3'$, $w_2 \overset{*}{\underset{\mathcal{R}}{\leftarrow}} w_1 \overset{*}{\underset{\mathcal{R}}{\rightarrow}} w_3$ and $S_2' \overset{*}{\underset{\mathcal{R}}{\leftarrow}} \Xi_i S_1' \overset{*}{\underset{\mathcal{R}}{\rightarrow}} S_3'$. Since w_1 is confluent from the assumption, we have $w_2 \downarrow_{\mathcal{R}} w_3$. Since $\Xi_i S_1'$ is confluent from the induction hypothesis, we have $S_2' \downarrow_{\mathcal{R}} S_3'$. Therefore we have $S_2 = w_2 S_2' \downarrow_{\mathcal{R}} w_3 S_3' = S_3$.
- In the latter case, we can show it by using Proposition 5.1(d) in similar to the former case.

□

Note that this lemma is provable more elegantly by using a notion of persistency [17] in similar way to [4,5]. However we gave the above proof to make the paper self-contained.

5.1 Termination analysis of $\mathcal{T}_P$

In the sequel, we analyze the termination property for $\mathcal{T}_P$.

We use a notation $n^\diamond$ to represent either n or $\tilde{n}$ for $n \in \{0,1,2\}$. We use a notation $\vec{u}$ for $u_1 \cdots u_k$.

Lemma 5.3 *Let P be an instance of PCP.*

(a) If $u_1 \cdots u_k \sim v_1 \cdots v_k$ for some $(u_i, v_i) \in \overline{P}$, then $w \xrightarrow[\mathcal{T}_P]{+} w$ where $w = \Xi_0(\tilde{0}0 \cdots 0)^{u_1}_{v_1}{}^{u_1}_{v_1}{}_{v_1} \cdots (\tilde{0}0 \cdots 0)^{u_k}_{v_k}{}^{u_k}_{v_k}{}_{v_k}\Psi_0$. Moreover, both of right-most reduction and left-most reduction are possible.

(b) If $\Xi_0\chi\Psi_0 \xrightarrow[\mathcal{T}_P]{+} \Xi_0\chi\Psi_0$ for some $\chi \in (\Sigma_c)^*$, then P has a solution.

Proof. (a): We have a reduction sequence $\Xi_0(\tilde{0}0^\diamond \cdots 0^\diamond)^{\vec{u}}_{\vec{v}}{}^{\vec{u}}_{\vec{v}}\Psi_0 \xrightarrow[\delta_2]{*} \Xi_0(\tilde{0}0^\diamond \cdots 0^\diamond)^{\vec{u}}_{\vec{v}}{}^{\vec{u}}_{\vec{v}}\Psi_0 \xrightarrow[\alpha_2\cup\beta_2\cup\gamma_2]{+} \Xi_2(\tilde{2}2^\diamond \cdots 2^\diamond)^{\vec{u}}_{\vec{v}}{}^{\vec{u}}_{\vec{v}}\Psi_2$. Here left-most reduction is possible. The right-most reduction exists by applying rules δ_2 as lazily as possible. Since $(u_i, v_i) \in \overline{P}$, we have a reduction sequence $\Xi_2(\tilde{2}2^\diamond \cdots 2^\diamond)^{\vec{u}}_{\vec{v}}{}^{\vec{u}}_{\vec{v}}\Psi_2 \xrightarrow[\gamma_1\cup\beta_1\cup\alpha_1]{+} \Xi_0(\tilde{0}0^\diamond \cdots 0^\diamond)^{\vec{u}}_{\vec{v}}{}^{\vec{u}}_{\vec{v}}\Psi_0$.

(b): Let $\Xi_0\chi\Psi_0 \xrightarrow[\mathcal{T}_P]{+} \Xi_0\chi\Psi_0$. From the construction of $\mathcal{T}_P$, a string $\Xi_2\chi'\Psi_2$ must appear in this reduction sequence. From the reduction sequence $\Xi_0\chi\Psi_0 \xrightarrow[\delta_2\cup\alpha_2\cup\beta_2\cup\gamma_2]{+} \Xi_2\chi'\Psi_2$, the string χ must be in forms of $(\tilde{0}0 \cdots 0)^{u_1}_{v_1}{}^{u_1}_{u'_1}{}_{v'_1} \cdots (\tilde{0}0 \cdots 0)^{u_k}_{v_k}{}^{u_k}_{u'_k}{}_{v'_k}$ and χ' must be in forms of $(\tilde{2}2 \cdots 2)^{u_1}_{v_1}{}^{u_1}_{w_1}{}_{w_1} \cdots (\tilde{2}2 \cdots 2)^{u_k}_{v_k}{}^{u_k}_{w_k}{}_{w_k}$ where $\vec{u'} \sim \vec{v'}$. From the reduction sequence $\Xi_2\chi'\Psi_2 = \Xi_2(\tilde{2}2^\diamond \cdots 2^\diamond)^{\vec{u}}_{\vec{v}}{}^{\vec{u}}_{\vec{v}}\Psi_2 \xrightarrow[\gamma_1\cup\beta_1\cup\alpha_1]{+} \Xi_0(\tilde{0}0^\diamond \cdots 0^\diamond)^{\vec{u}}_{\vec{v}}{}^{\vec{u}}_{\vec{u'}}{}_{\vec{v'}}\Psi_0 = \Xi_0\chi\Psi_0$, we have $(u_i, v_i) \in \overline{P}$ for every i. Since $\vec{u'}$ and $\vec{v'}$ are copied from $\vec{u}$ and $\vec{v}$ respectively in the latter reduction sequence by β_1-rules, we have $\vec{u'} = \vec{u}$ and $\vec{v'} = \vec{v}$. Thus we conclude $\vec{u} \sim \vec{v}$, which means that P has a solution. □

Proof for Lemma 3.5

((i)⇒(ii)∧(iii)): By Lemma 5.3(a).
((ii)∨(iii)⇒(iv)): Trivial.
((iv)⇒(i)): Let $\mathcal{T}_P$ is not terminating. From Lemma 5.2, there is a non-terminating and normal string w. Infinite reduction sequences starting from w must contain a string starting with Ξ_0 and ending with Ψ_0 by the construction of $\mathcal{T}_P$. Thus the lemma follows from Lemma 5.3(b). □

5.2 Confluence analysis of $\mathcal{C}_P$

In the sequel, we analyze the confluence property for $\mathcal{C}_P$.

The following propositions on the working area obtained from the construction of rules.

Proposition 5.4 *If* $(\cdots)^{u\ u}_{v\ u'}_{\ v'} \xrightarrow[\mathcal{C}_P]{*} (\cdots)^{u\ u}_{v\ u''}_{\ v''}$, *then* $u' \sim u''$ *and* $v' \sim u''$.

Proposition 5.5 $\xleftarrow[\Theta]{*} = \xleftrightarrow[\Theta]{*} = \xrightarrow[\Theta]{*}$.

The following lemma shows that strings in a specific form are closed under reductions by Θ-rules. For example, $\{(\underline{2}22)^{u\ u}_{v\ u_1}_{\ v_1}, (0\underline{2}2)^{u\ u}_{v\ u_2}_{\ v_2}, (00\underline{2})^{u\ u}_{v\ u_3}_{\ v_3}\}$ is closed under the reductions.

Lemma 5.6 *Let* $\chi = (\underbrace{0^\diamond \cdots 0^\diamond}_{n} \underline{p^\diamond} \underbrace{p^\diamond \cdots p^\diamond}_{m})^{u\ u}_{v\ u'}_{\ v'} \xrightarrow[\Theta]{*} \chi'$ *where* $m, n \geq 0$ *and* $p \in \{1, 2\}$. *Then* $\chi' = (\underbrace{0^\diamond \cdots 0^\diamond}_{n'} \underline{p^\diamond} \underbrace{p^\diamond \cdots p^\diamond}_{m'})^{u\ u}_{v\ u''}_{\ v''}$ *for some* $m', n' \geq 0$.

Proof. For any string in forms of χ for $p = 1$ (resp. $p = 2$), the only Θ_1-rules (resp. Θ_2-rules) are applicable, which produce a string in forms of χ'. □

We state some properties on Θ_1-rules.

Lemma 5.7 *Consider the following strings for* $i \leq j$:

$$\chi = (\tilde{0}0\cdots 0)^{u_1}_{v_1\ u'_1}_{\ v'_1} \cdots\ (\tilde{0}0\cdots 0)^{u_{i-1}}_{v_{i-1}\ u'_{i-1}}_{\ v'_{i-1}} (\tilde{\underline{1}}1\cdots 1)^{u_i}_{v_i\ u'_i}_{\ v'_i} (\tilde{1}1\cdots 1)^{u_{i+1}}_{v_{i+1}\ u'_{i+1}}_{\ v'_{i+1}} \cdots (\tilde{1}1\cdots 1)^{u_k}_{v_k\ u'_k}_{\ v'_k},$$

$$\chi' = (\tilde{0}0\cdots 0)^{u_1}_{v_1\ u''_1}_{\ v''_1} \cdots\ (\tilde{0}0\cdots 0)^{u_{j-1}}_{v_{j-1}\ u''_{j-1}}_{\ v''_{j-1}} (\tilde{\underline{1}}1\cdots 1)^{u_j}_{v_j\ u''_j}_{\ v''_j} (\tilde{1}1\cdots 1)^{u_{j+1}}_{v_{j+1}\ u''_{j+1}}_{\ v''_{j+1}} \cdots (\tilde{1}1\cdots 1)^{u_k}_{v_k\ u''_k}_{\ v''_k}.$$

If $\chi \xleftrightarrow[\Theta]{*} \chi'$ *then* $u_l = u'_l$, $v_l = v'_l$ *and* $(u_l, u'_l) \in \overline{P}$ *for all* $i \leq l < j$ *and* $u'_l = u''_l$ *and* $v'_l = v''_l$ *for all* $j \leq l$.

Proof. We have $\chi \xleftrightarrow[\Theta]{*} \chi'$ by Proposition 5.5. The lemma is proved by induction on the number of the rewrite steps. □

Next we state some properties on Θ_2-rules.

Lemma 5.8 *Let* $\chi = (\underline{2}^\diamond 2^\diamond \cdots 2^\diamond)^{u\ u}_{v\ u'}_{\ v'} \xrightarrow[\Theta]{*} (0^\diamond \cdots 0^\diamond \underline{2}^\diamond)^{u\ u}_{v\ u''}_{\ v''}$. *Then* $u'' \sim u' \sim v' \sim v''$.

Proof. We can prove, by induction on n, the claim that $\chi \xrightarrow[\Theta]{n} (0^\diamond \cdots 0^\diamond \underline{2}^\diamond)^{u_1}_{v_1\ u'_1}_{\ v'_1}$ $(2^\diamond \cdots 2^\diamond)^{u_2}_{v_2\ u'_2}_{\ v'_2}$ implies $u'_1 \sim v'_1$. From this claim we have $u' \sim v'$. Hence the lemma follows from Proposition 5.4. □

Lemma 5.9 *If* $w = \Xi_0 (0^\diamond \cdots 0^\diamond)^{u\ u}_{v\ u'}_{\ v'} \Psi_0 \xrightarrow[\mathcal{C}_P]{*} \Xi_0 \tilde{\underline{2}}^{x_1}_{x_2\ x_3} \chi \Psi_2 = w'$ *for some* $\chi \in (\Sigma_c)^*$, *then* $u' \sim v'$.

Proof. Prove by induction on the number of rewrite steps in the reduction sequence. In the case that the first step is a reduction by α_1'-rules, we have $w \underset{\alpha_1'}{\to} \Xi_0\chi'\Psi_1 \overset{*}{\underset{\Theta_1}{\to}} \Xi_0\chi''\Psi_1 \underset{(\alpha_1')^{-1}}{\to} \Xi_0(0^\diamond\cdots 0^\diamond)^{u}_{u''_{v''}}\Psi_0 \overset{*}{\underset{\mathcal{C}_P}{\to}} w'$. The claim follows since $u' \sim u''$ and $v' \sim v''$ by Proposition 5.4 and $u'' \sim v''$ by the induction hypothesis.

Consider the case that the first step is a reduction by α_2-rules. We have $w \underset{\alpha_2}{\to} \Xi_0(0^\diamond\cdots 0^\diamond\underline{2})^{u}_{u'_{v'}} \overset{*}{\underset{\mathcal{C}_P}{\to}} w'$. If $(\alpha_2)^{-1}$-rules are applied in the sequence, it is similar to the above case. Hence assume that $(\alpha_2)^{-1}$-rules are not applied. Then, $w' = \Xi_0(\tilde{2}2^\diamond\cdots 2^\diamond)^{u}_{u''_{v''}}\Psi_2$ by Lemma 5.6. Thus $u' \sim v'$ follows from Proposition 5.5 and Lemma 5.8.

Consider the case that the first step is a reduction by δ_2-rules. We have $w \underset{\delta_2}{\to} \Xi_0(0^\diamond\cdots 0^\diamond)^{u}_{u''_{v''}}\Psi_0 \overset{*}{\underset{\mathcal{C}_P}{\to}} w'$. The claim follows since $u' \sim u''$ and $v' \sim v''$ from Proposition 5.4 and $u'' \sim v''$ from the induction hypothesis. □

Lemma 5.10 *If* $w = \Xi_0(\tilde{0}0\cdots 0)^{u_1}_{u'_1}\cdots(\tilde{0}0\cdots 0)^{u_k}_{u'_k}\Psi_0 \overset{*}{\underset{\mathcal{C}_P}{\to}} \Xi_0\underline{\tilde{1}}^{x_1}_{x_2}\chi\Psi_1 = w'$ *for some* $\chi \in (\Sigma_c)^*$, *then* $u_1\cdots u_k \sim u'_1\cdots u'_k$, $v_1\cdots v_k \sim v'_1\cdots v'_k$ *and* $(u_i, v_i) \in \overline{P}$ *for every* i.

Proof. Prove by induction on the number of rewrite steps in the reduction sequence. Consider the case that the first step is a reduction by α_1'-rules and $(\alpha_1')^{-1}$-rules are not applied in the reduction. We have $w \underset{\alpha_1'}{\to} w'' \overset{*}{\underset{\mathcal{C}_P}{\to}} w'$, $u_k = u'_k$ and $v_k = v'_k$, where $w'' = \Xi_0(\tilde{0}0^\diamond\cdots 0^\diamond)^{u_1\cdots u_{k-1}}_{u'_1\cdots u'_{k-1}}(\underline{\tilde{1}}1\cdots 1)^{u_k}_{u'_k}\Psi_1$. Hence $w' = \Xi_0(\tilde{1}1\cdots 1)^{u_1}_{u''_1}\cdots(\tilde{1}1\cdots 1)^{u_k}_{u''_k}\Psi_2$ by Lemma 5.6. By applying Lemma 5.7 with $i = 0$ and $j = k$ we obtain $u_l = u''_l$ and $v_l = v''_l$ for all $1 \le l < k$ and $u''_k = u'_k$ and $v''_k = v'_k$. Hence we have $\vec{u} = \vec{u''}$ and $\vec{u} = \vec{u''}$. Since $\vec{u'} \sim \vec{u''}$ and $\vec{u'} \sim \vec{u''}$ by Proposition 5.4, $\vec{u} \sim \vec{u'}$ and $\vec{v} \sim \vec{v'}$ follow.

In the other cases, the proof is similar to that of Lemma 5.9. □

Lemma 5.11 *Let* P *be an instance of PCP. If* $w = \Xi_0\underline{\tilde{1}}^{x_1}_{x_2}\chi\Psi_1 \overset{*}{\underset{\mathcal{C}_P}{\leftrightarrow}} \Xi_0\underline{\tilde{2}}^{x_1}_{x_3}\chi'\Psi_2 = w'$ *for some* $\chi, \chi' \in (\Sigma_c)^*$, *then* P *has a solution.*

Proof.

Let $w \overset{*}{\underset{\mathcal{C}_P}{\leftrightarrow}} w'$. Then a string $\Xi_0\chi''\Psi_0$ must appear in this reduction and no underlined tag appears in χ'' from the construction of rules. Thus χ'' must be in forms of $\Xi_0(\tilde{0}\cdots 0)^{u_1}_{v_1}\cdots(\tilde{0}\cdots 0)^{u_k}_{v_k}\Psi_0$; otherwise the underlined tag displayed in w do not move to next symbol of Ψ_i by Lemma 5.6 and the construction of rules. By Lemma 5.9 and Lemma 5.10, we have $\vec{u} \sim \vec{v}$ and $(u_i, v_i) \in \overline{P}$, which means P has a solution.

□

We need some more lemma in order to guarantee the confluence of $\mathcal{C}_P$ when P has no solution.

Lemma 5.12 *Let w_1 and w_2 be normal strings over Σ^*. Then,*

(a) $w_1 \underset{\mathcal{C}_P\setminus\gamma_1'}{\overset{*}{\leftrightarrow}} w_2$ implies $w_1 \downarrow_{\mathcal{C}_P} w_2$, and

(b) $w_1 \underset{\mathcal{C}_P\setminus\gamma_2}{\overset{*}{\leftrightarrow}} w_2$ implies $w_1 \downarrow_{\mathcal{C}_P} w_2$.

Proof. Before proving (a), we show the claim $(*)$ that $w_1 \underset{\gamma_2}{\leftarrow} w_2 \underset{\Theta}{\overset{*}{\rightarrow}} w_3 \underset{\gamma_2}{\rightarrow} w_4$ implies $w_1 \underset{\Theta}{\overset{*}{\rightarrow}} w_4$ by induction on the number of rewrite steps. First of all w_2 must begin with $\Xi_0(\underline{\tilde{2}})^{x_1}_{x_3}$ since it has a redex of γ_2. Hence we can represent that $w_1 = \Xi_2(\tilde{2}X_1\cdots X_n)^{u}_{u'}S'$, $w_2 = \Xi_0(\underline{\tilde{2}}X_1\cdots X_n)^{u}_{u'}S'$, $w_3 = \Xi_0(\underline{\tilde{2}}X_1\cdots X_n)^{u}_{u''}S''$ and $w_4 = \Xi_2(\tilde{2}X_1\cdots X_n)^{u}_{u''}S''$ for $n \geq 0$, $X_i \in \{2, \tilde{2}\}$ and $S', S'' \in \Sigma^*$, where each tag of left-most symbol of S' and S'' is not 2 or $\tilde{2}$.

In the case that $S' = S'' = \Psi_2$, since $u' \sim u''$ and $v' \sim v''$ by Proposition 5.4, we have $w_1 \underset{\epsilon\cup\epsilon^{-1}}{\overset{*}{\rightarrow}} w_4$. In the other cases, we can separate the reduction, from the construction of rules, into $S' \underset{\Theta}{\overset{*}{\rightarrow}} S''$ and $w_1' = \Xi_2(\tilde{2}X_1\cdots X_n)^{u}_{u'} \underset{\gamma_2}{\leftarrow} \Xi_0(\underline{\tilde{2}}X_1\cdots X_n)^{u}_{u'} \underset{\Theta}{\overset{*}{\rightarrow}} \Xi_0(\underline{\tilde{2}}X_1\cdots X_n)^{u}_{u''} \underset{\gamma_2}{\rightarrow} \Xi_2(\tilde{2}X_1\cdots X_n)^{u}_{u''} = w_4'$. For the latter sequence, we have $w_1' \underset{\epsilon\cup\epsilon^{-1}}{\overset{*}{\rightarrow}} w_4'$ since $u' \sim u''$ and $v' \sim v''$ by Proposition 5.4. Therefore $w_1 \underset{\Theta}{\overset{*}{\rightarrow}} w_4$.

Now we prove the lemma (a) by induction on the number k of reduction steps by γ_2-rules in $w_1 \underset{\mathcal{C}_P\setminus\gamma_1'}{\overset{*}{\leftrightarrow}} w_2$.

- $(k = 0)$: It follows from Proposition 5.5.
- $(k = 1)$: The reduction sequence can be represented as $w_1 \underset{\Theta}{\overset{*}{\leftrightarrow}} w_3 \underset{\gamma_2}{\leftrightarrow} w_4 \underset{\Theta}{\overset{*}{\leftrightarrow}} w_2$. Then $w_1 \downarrow_{\mathcal{C}_P} w_2$ follows from Proposition 5.5.
- $(k > 1)$: The reduction sequence can be represented as $w_1 \underset{\Theta}{\overset{*}{\leftrightarrow}} w_3 \underset{\gamma_2}{\leftrightarrow} w_4 \underset{\mathcal{C}_P\setminus\gamma_1'}{\overset{*}{\leftrightarrow}} w_2$. If $w_3 \underset{\gamma_2}{\rightarrow} w_4$ we have done by Proposition 5.5 and the induction hypothesis. Otherwise $w_1 \underset{\Theta}{\overset{*}{\leftrightarrow}} w_3 \underset{\gamma_2}{\leftarrow} w_4 \underset{\Theta}{\overset{*}{\leftrightarrow}} w_4' \underset{\gamma_2}{\rightarrow} w_2' \underset{\mathcal{C}_P\setminus\gamma_1'}{\overset{*}{\leftrightarrow}} w_2$. Then $w_1 \downarrow_{\mathcal{C}_P} w_2$ by induction hypothesis since $w_1 \underset{\Theta}{\overset{*}{\leftrightarrow}} w_3 \underset{\Theta}{\overset{*}{\leftrightarrow}} w_2' \underset{\mathcal{C}_P\setminus\gamma_1'}{\overset{*}{\leftrightarrow}} w_2$ by the claim $(*)$ above.

Before proving (b), we show the claim $(**)$ that $w_1 \underset{\gamma_1'}{\leftarrow} w_2 \underset{\Theta}{\overset{*}{\rightarrow}} w_3 \underset{\gamma_1'}{\rightarrow} w_4$ implies $w_1 \underset{\Theta}{\overset{*}{\rightarrow}} w_4$ by induction on the number of rewrite steps. First of all w_2 must begin with $\Xi_0(\underline{\tilde{1}})^{x_1}_{x_2}$ since it has a redex of γ_1'. Hence we can represent that $w_1 = \Xi_1(\tilde{1}X_1\cdots X_n)^{u}_{u'}S'$, $w_2 = \Xi_0(\underline{\tilde{1}}X_1\cdots X_n)^{u}_{u'}S'$, $w_3 = \Xi_0(\underline{\tilde{1}}X_1\cdots X_n)^{u}_{u''}S''$ and $w_4 = \Xi_1(\tilde{1}X_1\cdots X_n)^{u}_{u''}S''$ for $n \geq 0$, $X_i \in \{1, \tilde{1}\}$ and $S', S'' \in \Sigma^*$, where each tag of left-most symbol of S' and S'' is not 1 or $\tilde{1}$.

In the case that $S' = S'' = \Psi_1$, we have $u' = u''$ and $v' = v''$ by applying Lemma 5.7 with $i = j = 1$. Thus $w_1 = w_4$ follows. In the other cases, we can separate the reduction, from the construction of rules, into $S' \underset{\Theta}{\overset{*}{\to}} S''$ and $w'_1 = \Xi_1(\tilde{1}X_1 \cdots X_n)^{u}_{v}{}^{u}_{u'}{}^{}_{v'} \underset{\gamma'_1}{\leftarrow} \Xi_0(\underline{\tilde{1}}X_1 \cdots X_n)^{u}_{v}{}^{u}_{u'}{}^{}_{v'} \underset{\Theta}{\overset{*}{\to}} \Xi_0(\underline{\tilde{1}}X_1 \cdots X_n)^{u}_{v}{}^{u}_{u''}{}^{}_{v''} \underset{\gamma'_1}{\to} \Xi_1(\tilde{1}X_1 \cdots X_n)^{u}_{v}{}^{u}_{u''}{}^{}_{v''} = w'_4$. For the latter sequence, we have $w'_1 = w'_4$ since $u' \sim u''$ and $v' \sim v''$ by Lemma 5.7. Therefore $w_1 \underset{\Theta}{\overset{*}{\to}} w_4$.

By using the claim $(**)$, the lemma (b) can be shown in similar to (a). □

Proof for Lemma 4.3

Since $\Rightarrow$-direction is easy from the observation of Example 4.2, we show $\Leftarrow$-direction.

Assuming that P has no solution, let's show that $\mathcal{C}_P$ is confluent. From Lemma 5.2, it is enough to consider $w_1 \underset{\mathcal{C}_P}{\overset{*}{\leftarrow}} w_0 \underset{\mathcal{C}_P}{\overset{*}{\to}} w_2$ for a normal string w_0.

- Consider the case that w_0 starts with Ξ_0 and ends with Ψ_i for some $i \in \{0, 1, 2\}$. Assume that both of γ'_1 and γ_2 are applied in the reduction sequence. Then P must have a solution by Lemma 5.11, which is a contradiction. Hence at least one of γ'_1 or γ_2 rules cannot be applied in the reduction sequence.
- In either of following cases:
  · w_0 ends with Ψ_i for some $i \in \{0, 1, 2\}$ and all other symbols are of Σ_c,
  · w_0 starts with Ξ_1 or Ξ_2, and
  · w_0 starts with Ξ_0 and all other symbols are of Σ_c,
  It is easy to see that at least one of γ'_1 or γ_2 rules cannot be applied in the reduction sequence.

In any of the above cases, we have $w_1 \downarrow_{R_P} w_2$ by Lemma 5.12. □

6 Length-two SRSs

Length-two SRSs are SRSs that consist of rules with length two, that is, $|l| = |r| = 2$ for every rule $l \to r$. In this section we give a transformation of a length preserving SRS over Σ_0 into a length-two SRS over Δ that preserves confluent property and termination property.

Let $\Sigma = \Sigma_0 \cup \{\text{–}\}$ and $m + 1 (\geq 3)$ be the maximum length of rules in $\mathcal{R}$. Let $\Delta_0 = (\Sigma_0)^m$ and $\Delta = \Delta_0 \cup \{wv \mid w \in (\Sigma_0)^k, v \in \{\text{–}\}^{m-k}, 1 \leq k \leq m-1\}$.

The natural mapping $\phi : \Delta \to \Sigma^m$ is defined as $\phi(w) = w$. This mapping is naturally extended to $\phi : \Delta^* \to \Sigma^*$.

Example 6.1 *Let* $\Sigma_0 = \{a, b\}$ *and* $m = 2$. *Then* $\Delta_0 = \{aa, ab, ba, bb\}$, $\Delta = \Delta_0 \cup \{a\text{–}, b\text{–}\}$ *and* $\phi(ab\ bb\ a\text{–}) = abbba\text{–}$.

We give a transformation of a length preserving SRS $\mathcal{R}$ into a length-two SRS $tw(\mathcal{R})$ over Δ.

$$tw(\mathcal{R}) = \{w_1 w_2 \to w_3 w_4 \mid w_i \in \Delta,\ \phi(w_1 w_2) \underset{\mathcal{R}}{\to} \phi(w_3 w_4)\}$$

Example 6.2 *Let* $\mathcal{R} = \{bbb \to aaa\}$ *over* $\Sigma_0 = \{a, b\}$. *Then* $tw(\mathcal{R})$ *is the following*

length-two SRS over Δ*, where* Δ *is displayed in Example 6.1.*

$$tw(\mathcal{R}) = \begin{cases} bb\ b\text{–} \to aa\ a\text{–}, & bb\ ba \to aa\ aa, \quad bb\ bb \to aa\ ab, \\ ab\ bb \to aa\ aa, & bb\ bb \to ba\ aa \end{cases}$$

We say a string $w_1 \cdots w_n$ over Δ^* is *normal* if $w_1, \ldots, w_{n-1} \in \Delta_0$. From the construction of $tw(\mathcal{R})$, all reachable strings from a normal string are also normal.

We define a mapping $\psi : \Delta^* \to (\Sigma_0)^*$ as $\psi(\alpha) = w$ where w is a string obtained from $\phi(\alpha)$ by removing all –'s. We define a mapping $\psi^{-1} : (\Sigma_0)^* \to \Delta^*$ as $\psi^{-1}(w) = \alpha$ where $\psi(\alpha) = w$ and α is normal. For example $\psi(ab\ bb\ a\text{–}) = abbba$ and $\psi^{-1}(abbba) = ab\ bb\ a\text{–}$. Trivially we have $\psi^{-1}(\psi(\alpha)) = \alpha$ for normal $\alpha \in \Delta^*$ and $\psi(\psi^{-1}(w)) = w$ for $w \in (\Sigma_0)^*$.

Proposition 6.3 *(a) For a normal* $\alpha_1 \in \Delta^*$*, if* $\alpha_1 \underset{tw(\mathcal{R})}{\to} \alpha_2$ *then* $\psi(\alpha_1) \underset{\mathcal{R}}{\to} \psi(\alpha_2)$

(b) For $w_1 \in (\Sigma_0)^*$*, if* $w_1 \underset{\mathcal{R}}{\to} w_2$ *then* $\psi^{-1}(w_1) \underset{tw(\mathcal{R})}{\to} \psi^{-1}(w_2)$

Proof. From the construction of $tw(\mathcal{R})$. □

Lemma 6.4 *Let* $\mathcal{R}$ *an SRS. The SRS* $tw(\mathcal{R})$ *is confluent (resp. terminating, right-most terminating, left-most terminating) if and only if* α *is confluent (resp. terminating, right-most terminating, left-most terminating) for every normal* $\alpha \in \Delta^*$*.*

Proof. We can prove it in similar to the proof of Lemma 5.2. Here $\Delta \backslash \Delta_0$ symbols play the same roles as Ψ_i symbols. □

Lemma 6.5 *Let* $\mathcal{R}$ *be an length preserving SRS.* $\mathcal{R}$ *is terminating (resp. left-most terminating, right-most terminating) if and only if* $tw(\mathcal{R})$ *is terminating (resp. left-most terminating, right-most terminating).*

Proof. ($\Rightarrow$): Let $tw(\mathcal{R})$ be non-terminating. By Lemma 6.4 we have an infinite reduction sequence for $tw(\mathcal{R})$ starting from a normal string. This direction follows from Proposition 6.3(a).
($\Leftarrow$): Let $\mathcal{R}$ be non-terminating. Then we have an infinite reduction sequence. By Proposition 6.3(b) we have an infinite reduction sequence for $tw(\mathcal{R})$.

This proof also works on either left-most case or right-most case. □

Lemma 6.6 *Let* $\mathcal{R}$ *be an length preserving SRS.* $\mathcal{R}$ *is confluent if and only if* $tw(\mathcal{R})$ *is confluent.*

Proof. ($\Rightarrow$): Let $\beta_1 \underset{tw(\mathcal{R})}{\overset{*}{\leftarrow}} \alpha \underset{tw(\mathcal{R})}{\overset{*}{\to}} \beta_2$. We can assume that α is normal by Lemma 6.4. We have $\psi(\beta_1) \underset{\mathcal{R}}{\overset{*}{\leftarrow}} \psi(\alpha) \underset{\mathcal{R}}{\overset{*}{\to}} \psi(\beta_2)$ by Proposition 6.3(a). Since $\mathcal{R}$ is confluent, there exists a string $w \in \Sigma_0^*$ such that $\psi(\beta_1) \underset{\mathcal{R}}{\overset{*}{\to}} w \underset{\mathcal{R}}{\overset{*}{\leftarrow}} \psi(\beta_2)$. Therefore we have $\beta_1 = \psi^{-1}(\psi(\beta_1)) \underset{tw(\mathcal{R})}{\overset{*}{\to}} \psi^{-1}(w) \underset{tw(\mathcal{R})}{\overset{*}{\leftarrow}} \psi^{-1}(\psi(\beta_2)) = \beta_2$ by Proposition 6.3(b).

($\Leftarrow$): Let $u_1 \underset{\mathcal{R}}{\overset{*}{\leftarrow}} w \underset{\mathcal{R}}{\overset{*}{\to}} u_2$. We have $\psi^{-1}(u_1) \underset{tw(\mathcal{R})}{\overset{*}{\leftarrow}} \psi^{-1}(w) \underset{tw(\mathcal{R})}{\overset{*}{\to}} \psi^{-1}(u_2)$ by Proposition 6.3(b). Since $\mathcal{R}$ is confluent, there exists a string $\alpha \in \Delta^*$ such that $\psi^{-1}(u_1) \underset{tw(\mathcal{R})}{\overset{*}{\to}} \alpha \underset{tw(\mathcal{R})}{\overset{*}{\leftarrow}} \psi^{-1}(u_2)$. Since α is normal, we have $u_1 =$

$\psi(\psi^{-1}(u_1)) \xrightarrow[\mathcal{R}]{*} \psi(\alpha) \xleftarrow[\mathcal{R}]{*} \psi(\psi^{-1}(u_2)) = u_2$ by Proposition 6.3(a). □

Theorem 6.7 *Confluence (termination, left-most termination, right-most termination) is an undecidable property for length-two SRSs.*

Proof. Directly obtained from Theorem 4.4 and Lemma 6.6 (Lemma 6.5). □

Acknowledgment

We would like to thank the anonymous referees for their helpful comments and remarks. This work is partly supported by MEXT.KAKENHI #18500011 and #16300005.

References

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

[2] A.-C. Caron. *Linear Bounded Automata and Rewrite Systems: Influence of Initial Configuration on Decision Properties*, Proc. of the Colloquium on Trees in Algebra and Programming(CAAP 91), LNCS, *493*, pp.74–89, 1991.

[3] N. Derchowitz. *Termination of Linear Rewriting Systems*, Proc. of the 8th International Colloquium on Automata, Languages and Programming(ICALP 18), LNCS, *115*, pp.448–458, 1981.

[4] A. Geser, A. Middeldorp, E. Ohlebusch, H. Zantema. *Relative Undecidability in Term Rewriting(Part 1: The Termination Hierarchy)*, Information and Computation, *178(1)*, pp.101–131, 2002.

[5] A. Geser, A. Middeldorp, E. Ohlebusch, H. Zantema. *Relative Undecidability in Term Rewriting (Part 2: The confluence Hierarchy)*, Information and Computation, *178(1)*, pp.132–148, 2002.

[6] G. Godoy, A. Tiwari, R. Verma. *On the Confluence of Linear Shallow Term Rewriting Systems*, Proc. of 20th Intl. Symposium on Theoretical Aspects of Computer Science (STACS 2003), LNCS, *2507*, pp.85–96, 2003.

[7] G. Godoy, A. Tiwari. *Confluence of Shallow Right-Linear Rewrite Systems*, Proc. of 14th Annual Conference on Computer Science Logic (CSL 2005), LNCS, *3634*, pp.541–556, 2005.

[8] G. Godoy, A. Tiwari. *Termination of Rewrite Systems with Shallow Right-Linear, Collapsing, and Right-Ground Rules*, Proc. of 20th International Conference on Automated Deduction (CADE 2005), LNCS, *3632*, pp.164–176, 2005.

[9] G. Huet, D. S. Lankford. *On the Uniform Halting Problem for Term Rewriting Systems*, Technical Report of INRIA, *283*, 1978.

[10] F. Jacquemard. *Reachability and Confluence are Undecidable for Flat Term Rewriting Systems*, Information Processing Letters *87(5)*, pp.265–270, 2003.

[11] K. E. Knuth, P. B. Bendix. *Computational Problems in Abstract Algebra*, Pergamon Press, Oxford, pp.263–297, 1970.

[12] I. Mitsuhashi, M. Oyamguchi, Y. Ohta, and T. Yamada. *The Joinability and Related Decision Problems for Confluent Semi-Constructor TRSs*, Transactions of Information Processing Society of Japan, *47(5)*, pp.1502-1514, 2006.

[13] I. Mitsuhashi, M. Oyamaguchi and F. Jacquemard. *The Confluence Problem for Flat TRSs*, Proc. of 8th Intl. Conf. on Artificial Intelligence and Symbolic Computation (AISC'06), LNAI *4120*, pp.68-81, 2006.

[14] M. Oyamaguchi, *The Church-Rosser Property for ground term rewriting systems is Decidable*, Theoretical Computer Science, *49*, pp.43–79, 1987.

[15] E. Post. *A Variant of a Recursively Unsolvable Problem*. Bulletin of the American Mathematical Society, *52*, pp.264–268, 1946.

[16] Y. Wang, M. Sakai, *Decidability of Termination for Semi-Constructor TRSs, Left-Linear Shallow TRSs and Related Systems*, Proc. of 17th International Conference on Term Rewriting and Applications (RTA 2006), LNCS, *4098*, pp.343–356, 2006.

[17] H. Zantema, *Termination of Term Rewriting: Interpretation and type elimination*, Journal of Symbolic Computation, *17*, pp.23–50, 1994.

Rules and Strategies in Java

Pierre-Etienne Moreau and Antoine Reilles

INRIA & INPL & Loria
(moreau|reilles)@loria.fr

Abstract

In this paper we present the essential feature we have considered when designing a new language based on rules and strategies. Relying on the implementation of Tom, we explain how these ingredients can be implemented and integrated in a Java environment.

1 Introduction

The notion of rewrite rule is an abstraction that can be used to model various processes. It has been used intensively to model, study, and analyze various parts of a complex system, from algorithms to running software. On one side it can be used to describe the behavior of a transition system for instance. On the other side, it provides a theoretical framework useful to certify and prove properties such as termination or confluence.

Besides its straightforward interpretation, the notion of rewrite rule can be used to produce efficient implementations. It has been successfully used in theorem provers and proof assistants such as RRL, Otter, C*i*ME, Coq, as well as the main execution mechanism of rule based and functional languages such as ASF+SDF [7], Clean, Caml, ELAN [4], Maude [3], and Stratego [10] for instance.

Programming with rewrite rules is apparently easy: a complex transformation can be decomposed into elementary transformations, encoded using a rewrite rule, then, we rely on the rule engine to fire a rule whenever it is possible. Most of the time, we are interested in getting a result whose computation is deterministic. In other words, the result should be reproducible, the set of rules should be terminating and confluent. However, things are rarely confluent by nature. One solution could be to use the Knuth-Bendix completion, but this is not realistic on large programs. In practice, starting from an initial signature and a simple set of rewrite rules, the programmer often modifies the signature and the rules in order to encode some control. From a software engineering point of view, this annihilates the elegance of term rewriting and makes the system much more complex and difficult to maintain.

A first solution to this problem is to assign a priority to each rule and to consider an execution mechanism that encodes a fixed order of reduction, such as *innermost* or *outermost*, also called *call by value* or *call by name* in functional programming languages. Another solution is to separate the control from the rules. Instead of encoding the control into the rules themselves, it is described in a distinct expression or language. The expressions that specifies how the rules should be applied are called *strategies*. The design of such a strategy language is not easy and several attempts and proposals have been made.

OBJ is one of the first languages that introduced an explicit form of strategy, called *evaluation strategy*. To each operator a list of integers can be attached to specify in which order the arguments should be evaluated.

Maude followed this approach and added the notion of *meta-level*. In this setting, a rewrite rule has a name, considered as a constant, and can be explicitly applied via the `meta-apply` operator. The application of a set of rules can be controlled by another program, expressed by rewriting and using `meta-apply`. This new program can also be controlled by another program from the meta-meta-level. This tower of reflexivity is very elegant and expressive, but a bit difficult to use.

ELAN has its origins in OBJ, but followed another approach. Instead of having a meta-level, ELAN was the first language to introduce an explicit *strategy language* to control the application of rules. Each rule has a name which corresponds to an elementary strategy. A strategy can then be combined with another one using operators such as `;` (sequence), `repeat`, `dont-care`, and `dont-know` for instance. This strategy language was both very expressive and easy to use.

Stratego has been inspired by ELAN, Maude, and the functional programming style. It introduces a quite elegant and simple strategy language. Similarly to ELAN, a rule is an elementary strategy that can be combined with strategy operators such as `;` (sequence), `<+` (left-choice), *etc.* The main contribution comes from the introduction of a recursion operator and two generic congruence operators `All` and `One`, that can be used to describe higher-level strategies such as *top-down*, *innermost* or *outermost*. In this setting, we have $TopDown(s) = \mu x.s\ ;\ All(x)$, which applies s to a term t, and then recursively applies the $TopDown(s)$ strategy to the immediate subterms of t.

ASF+SDF has also a strategy language, in the same spirit as the OBJ's one. To each operator an annotation can be attached, that specifies its behavior. The combination of `traversal` and `bottom-up` indicates that a given set of rules should be applied in a bottom-up way for example. This approach is of course less general than the previous ones, but it is an interesting trade-of between expressiveness and simplicity to use.

During the last decade we have accumulated an important experience in both implementing and using rule based languages. In this paper we try to isolate the essential constructs and features that have to be considered when designing a new rule and strategy based language. In a second part, we explain how those features can be smoothly integrated and efficiently implemented into an object oriented programming language such as Java.

2 Our wish list

Starting from the ELAN experience, we have tried to design a new language based on the same concepts. This process leads us to analyse what were the good points, and what were the points that could be improved. The ELAN language clearly has many interesting constructs, in particular the notion of rules, strategies, and equational matching. Unfortunately, its cohabitation with largely used programming languages such as C or Java needs some improvement. In particular, it could be very interesting to use the constructs provided by ELAN in those programming languages themselves. Starting from that situation, our goal was to design a new language, called Tom, with a comparable expressiveness, but in a more accessible programming environment. In this section we present what are the requirements and the essential features that have been considered when designing the Tom programming language.

Terms

A first requirement was to be able to describe and manipulate tree shaped structures, or terms. The ELAN experience showed that it is more convenient to manipulate many sorted terms, especially since this simple typing of terms does help catching many programming errors. Languages such as ASF+SDF, ELAN or Maude do mix the term algebra used to express the rules and the concrete input syntax of the specifications. This feature is very convenient to prototype transformations, but makes the implementation of the language much more complex. In order to keep the language and the implementation simple, we wanted to keep the syntax of rules in a prefix notation, like in many functional programming languages.

Describing elementary transformations.

When designing complex applications, it is important to be able to decompose the various transformations into different rewrite systems or elementary transformations, and then decide which rule apply to which term. A first step towards this goal is to let the user define labeled rules:

$$[\ell]\ l \longrightarrow r$$

A same label can be given to several rules to define a rewrite system. The application of such system has to be explicitly specified by the user. Given a term t, the application of a labeled rule performs a single step of reduction at the root position, if t is matched by a left-hand side. Otherwise, the application fails.

Computing canonical forms.

Another important feature inherited from ELAN is the ability to automatically maintain terms in normal form with respect to a rewrite system. This is done via the definition of unlabeled rules which are applied until getting an irreducible term. The considered rewrite system has of course to be confluent and terminating.

Using unlabeled rules, it is possible to specify a canonical form for the term data structures of the application, such as representing logical formulas in disjunc-

tive normal form, or enforcing constant expression evaluation in a programming language. For instance, the following rule set can be used to express how to build boolean formulas in disjunctive normal form:

```
Not(And(l,r))  -> Or(Not(l),Not(r))
Not(Or(l,r))   -> And(Not(l),Not(r))
Not(True())    -> False()
Not(False())   -> True()
Not(Not(x))    -> x
And(x,Or(y,z)) -> Or(And(x,y),And(x,z))
And(Or(y,z),x) -> Or(And(y,x),And(z,x))
```

Maintaining a data structure in canonical form is an essential feature that improves the quality of software. The programmer does no longer have to take care of this maintenance by calling normalisation functions. In addition, this makes the writing of elementary transformations simpler since only canonical forms have to be considered.

Controlling the rewriting.

In our case, the situation is a bit particular since in addition to the notion of rule, there is an underlying Turing complete language, namely Java. To control the application of rules, an attracting possibility would be to use Java directly. This language natively offers the composition (`;`), the repetition (`while`), and many other control statements. However, the more we use Java, the more it becomes complex to reason about programs and to perform proofs. How to show that a rewrite system is terminating under a given strategy when this strategy is expressed in Java? Therefore, the problem is to find a good strategy language which is both expressive and simple to use.

As mentioned in the introduction, we have studied and experimented the expressiveness of several existing strategy languages. The one proposed by Stratego [10] has several interesting properties. It is atomic, being composed of less than 10 elementary combinators. It is expressive, allowing the definition of various traversal strategies. Initially not tailored to support non-deterministic searches, we will see that this limitation can be removed in a simple and elegant way, using context information.

We thus decided to base our strategy language on elementary combinators, that are combined with labeled rules to build more complex strategies. Given a set of rules, there are two fundamental operators for combining rules: the choice operator $Choice(s_1, s_2)$ which applies s_2 only if the application of s_1 fails. The sequence operator $Sequence(s_1, s_2)$ which applies s_1, and then, if that succeeds, applies s_2. The different combinators are described in Figure 1.

Recursive strategy definitions are essential to describe the common rewriting strategies such as *top-down*, *bottom-up* and *leftmost-innermost*. For instance, *bottom-up* is defined as $BottomUp(v) = Sequence(All(BottomUp(v)), v)$

The basic strategy combinators and the strategies that are created by composing them are generic, and will perform equally on any data structure, when the labeled rules are specific to the data structure (since they perform pattern matching and

Combinator	Semantics
`Identity()`	Does nothing, and returns the original term
`Fail()`	Always fails
`All(`v`)`	Applies v to all direct subterms in sequence
`Choice(`v_1`,`v_2`)`	Applies v_1, then v_2 if v_1 failed
`Sequence(`v_1`,`v_2`)`	Applies v_1, then v_2. Fail if one of them fail
`Not(`v`)`	Applies v. Fail if v is applicable, returns identity otherwise
`Omega(`i`,`v`)`	Applies v to the i-th sub-term if it exists, fails otherwise
`One(`v`)`	Applies v to all sub-terms in order, until a success
`IfThenElse(`c`,`v_1`,`v_2`)`	Applies c, then if c success, apply v_1, otherwise v_2

Fig. 1. Elementary strategy combinators

term construction). In order to build elegantly some transformations, we need data structure dependent strategy combinators, such as congruence strategies. They are used to decompose terms and to apply strategies to subterms. For example, the congruence strategy for the `And` constructor, noted `_And` has two arguments, which are strategies to be applied respectively to the left and right subterms of an `And`. `_And(s1,s2)` applied to the term `And(x,y)` will apply `s1` to `x` and `s2` to `y`, and will fail if applied to a term that is not rooted by `And`.

Knowing the context.

Usually, a rule application, even under strategies, is context free. When a rule or a strategy is applied, the result only depends on the term to which the strategy is applied. However, it is common to require some knowledge about the context when applying a strategy. For example, we may want to know if a particular subterm is in positive or negative position in a formula.

For that, context information such as the list of terms that were traversed by the strategy before accessing the current subterm, or the position of the current subterm in the traversed term should be accessible when evaluating the right hand side of a labeled rule. The explicit representation of the notion of position also corresponds to classical operations found in the literature, such as the access to a given subterm ($t_{|\omega}$) or the replacement ($t[u]_\omega$) for instance.

Exploring a search space.

Many applications of rule based systems do require the manipulation of non deterministic rewrite systems. For instance, cryptographic protocol verification can be treated as a reachability problem in a particular non deterministic rewrite system that models the protocol. We thus require the strategy language to be able to support non determinism, by letting the user compute the set of successors of the application of a set of rules, and also the successors of the application of rules to

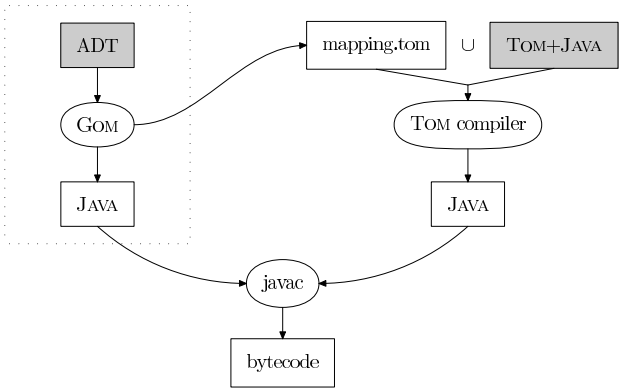


Fig. 2. Given a signature (ADT), a Java implementation is generated. In addition, a *mapping* that makes the implementation a parameter of Tom is generated. The algebraic data structure can be directly manipulated in a Tom program. The compilation process translates the data structure and the rule based program into Java classes, that can be combined with other libraries.

different redexes. In ELAN, only the first case can be easily implemented.

Such a control should be flexible enough to support the exploration of a search space in a *depth-first* or *breadth-first* manner.

Integration into Java.

Considered separately, each feature mentioned previously is not a revolution and already appears in a form or another in an existing rule based language, except the explicit representation of position information which is a real contribution. The main difficulty of our approach is to *integrate everything into* Java. This is also the main contribution of this work.

3 Our approach

As presented in [5,1], the main component of our system is the Tom compiler. It mainly introduces two constructs: a `‘<term>` to build an algebraic term, and a `%match` construct, similar to `switch/case` which executes an action (a Java statement) when a pattern matches a given subject. This system does not impose a specific term data structure: the implementation of the term data structure becomes a parameter of the Tom compiler, using a *mapping* mechanism. An advantage of this approach is to make the `%match` construct usable on any kind of data structure. The counterpart is that the user has to provide an implementation.

Terms

To help the programmer, we have designed a tool called Gom [6] that takes an algebraic signature as input and generates a Java implementation of this signature. This work is an extension of [8]. The interaction between Tom and Gom is illustrated in Figure 2. In this formalism, a specification for boolean expressions can be the following:

```
%gom {
  module Bool
```

```
  imports String
  abstract syntax
  Bool = True()
       | False()
       | Not(b:Bool)
       | And(l:Bool,r:Bool)
       | Or(l:Bool,r:Bool)
       | Var(name:String)
}
```

The language provides modularity with the use of the `imports` keyword. Java classes implementing this structure are then generated, providing a typed interface as well as an efficient implementation based on maximal sharing. Following the *factory* design pattern, these classes provide static construction functions.

Computing canonical forms.

The implementation of canonization functions can be done by encoding a rewrite system into functions: to each defined symbol a function is associated. These functions are called when a term has to be built, and the corresponding normal form is returned. The main drawback of this approach is that the user (a Java programmer) can *forget* to call these normalisation functions and directly use the factory generated by Gom instead. This would result in terms which are no longer in normal form. To avoid this problem, we consider that the notion of unlabeled rule is strongly tied to the term data structure that is used in the application. We thus propose to define the set of unlabeled rules in conjunction to the definition of the data structure, in the Gom formalism. The code that implements the normalisation function will then be generated in the construction functions provided by the factory. An important consequence of this design is the impossibility to build a term which is not in normal form. Therefore, the boolean formulae that should only be manipulated in disjunctive normal form can be defined as follows:

```
%gom {
  module Bool
  imports String
  abstract syntax
  Bool = True()
       | False()
       | Not(b:Bool)
       | And(l:Bool,r:Bool)
       | Or(l:Bool,r:Bool)
       | Var(name:String)
  rules() {
    Not(And(l,r))  -> Or(Not(l),Not(r))
    Not(Or(l,r))   -> And(Not(l),Not(r))
    Not(True())    -> False()
    Not(False())   -> True()
    Not(Not(x))    -> x
```

```
    And(x,Or(y,z)) -> Or(And(x,y),And(x,z))
    And(Or(y,z),x) -> Or(And(y,x),And(z,x))
  }
}
```

This construct ensures that there is no way to obtain a term that is not normal with respect to the rewrite system. It is also possible to use conditional rewrite rules in the ruleset, which lets the definition of ordered or balanced trees more convenient.

Describing elementary transformations.

The notion of labeled rule cannot be implemented with the same approach. A labeled rule corresponds to an elementary strategy that has to be manipulated as an object. Therefore, its implementation cannot be a *function*: this is not a first order object in Java. In our setting, an elementary strategy is implemented by a *class* which has an `apply` function. The implementation of such a class is generated automatically when using the `%strategy` keyword:

```
%strategy swap() {
  visit Bool {
    Or(Var(x),Var(y)) -> {
      if(x.compareTo(y)) {
        return `Or(Var(y),Var(x));
      }
    }
  }
}
```

The construction `%strategy swap()` defines a labeled rule whose name is `swap`. Similarly to the `%match` construct, the right-hand side of a rule can be any Java statement. In this example, `return `Or(Var(y),Var(x))` is used to return an object of sort `Bool`. Such a rule can then be applied to a term using the `apply` function. The following instructions applies the `swap` strategy to a simple term, storing the result in the `res` variable:

```
Bool b = `Or(Var("b"),Var("a"));
Bool res = `swap().apply(b);
```

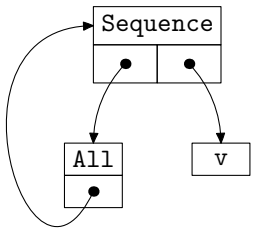
Controlling the rewriting.

In [11], J. Visser introduced a new pattern to implement elementary combinators. This work, implemented in JJTraveler, is very important since it makes available in Java the combinator that exists in the Stratego language. A minor drawback of the approach is the implementation of the elementary strategies (the labeled rules), which requires some extra effort from the user. In particular, a quite complex interface has to be instantiated for each labeled rule. This code is now automatically generated by the `%strategy` keyword. A simple solution to implement our strategy language would have been to reuse JJTraveler in combination with the `%strategy` construct. This was our first attempt.

When programming with rules and strategies, as in Stratego [9,10] and ELAN [2],

the definition of recursive composed strategies is very frequent. The definition of common rewriting strategies such as *top-down*, *bottom-up*, and *leftmost-innermost* is done via the use of an explicit recursion operator μ. This operator, like the `let rec` construct in functional languages, is essential. However, in JJTraveler this μ operator does not exist. Thus, the graph that represents the recursive strategy has to be encoded directly in Java. This is not easy and error prone. For example, the construction of the *BottomUp* strategy $\mu x\,.\,All(x)\ ;\ s$ is encoded as follows:

```
package jjtraveler;
public class BottomUp extends Sequence {
  public BottomUp(Visitor v) {
    super(null,v);
    first = new All(this);
  }
}
```



This graph is obtained by first allocating a `Sequence` with a dummy pointer set to `null`. Then the correct graph is built. This is clearly not the abstraction level we want to provide. In our framework, which is an extension of JJTraveler, we allow the definition of strategies that explicitly use the μ recursion operator. In Java, the *bottom-up* strategy can be implemented as follows:

```
Strategy BottomUp(Strategy s) {
  return `mu(x, Sequence(All(x),s));
}
```

This strategy can then be used to apply a strategy `s` to each node of a term `u` with `t = `BottomUp(s).visit(u)`.

Context and position

When analysing source code, context free information is not enough. For instance, when verifying a particular part of a program, sometimes we require to know what are the variables that are defined in the context, as well as their type. Such information can be obtained by letting user strategies to use a mutable state, that will be altered during the tree traversal. When finding a variable declaration for instance, the variable is stored, and further on, when finding a variable, it can be compared to what has been previously stored:

```
%strategy VarCheck(Stack bag) {
  visit Instruction {
    Assign(var,expression,body) -> {
      bag.push(`var);
      this.visit(`body);
      bag.pop();
    }
  }
  visit Expression {
    Variable(var) -> {
      if(!bag.contains(`var)) {
```

```
        throw new Exception("Undefined variable " + `var);
      }
    }
  }
}
```

Another way to take the strategy application context into account is to use the notion of *position* in a term. The idea is to come back to the textbook notion of rewriting, resorting to the notion of position to identify a particular subterm or redex ($t_{|\omega}$ or $t[u]_\omega$ for example). The novelty here is to give access to the current position in a term while traversing it. This position can be manipulated as a first class object in the programming language.

To implement such a feature, each strategy combinator has to maintain the path from the root to the currently traversed term. This information is automatically maintained by the elementary combinators provided by the library: `Sequence`, `All`, `One`, *etc.* From a user perspective, the current position is returned by the function `getPosition()`. The object representing the position is not mutable and can be stored to be reused later. It will not be further modified when the strategy continues its traversal.

The expressive power provided by the explicit representation of positions is very high. For example, this feature is essential to implemented non deterministic exploration. Given a rewrite system implemented by a strategy, to compute the set of all possible successors, a two step algorithm can be used. First, a top-down traversal is performed to identify all possible redexes. This set of redexes can be represented by a list of pairs (t, ω_i) where t is the term, and each ω_i correspond to the position of a redex. In a second step, the list is iterated, and the successors of $t_{|\omega_i}$ are computed and collected.

4 Conclusion

The design of a language that integrates the notions of rules and strategies in a mainstream programming language such as Java is not an easy task. We described the constructs and features that are required, and we showed how they can be integrated into the Java language.

This results in a language in which formally defining algorithms using terms, rules, and strategies is easy and elegant, without losing the flexibility and versatility of the underlying host language. It makes possible the mixture between the formal definition of a logic system, using Tom, with user code that provides a fully fledged interface implemented using low level code. The integration of formal aspects into a classical programming language eases a gradual integration of formal methods into existing projects. In practice, this integration becomes natural since the use of algebraic constructs, as those provided by Tom, makes the code both smaller and more readable.

References

[1] Emilie Balland, Paul Brauner, Radu Kopetz, Pierre-Etienne Moreau, and Antoine Reilles. Tom: Piggybacking rewriting on java. In *Proceedings of the 18th Conference on Rewriting Techniques and Applications*, Lecture Notes in Computer Science. Springer-Verlag, 2007.

[2] Peter Borovanský. *Le contrôle de la réécriture: étude et implantation d'un formalisme de stratégies.* Thèse de Doctorat d'Université, Université Henri Poincaré - Nancy I, October 1998.

[3] Manuel Clavel and José Meseguer. Reflection and strategies in rewriting logic. In J. Meseguer, editor, *Electronic Notes in Theoretical Computer Science*, volume 4. Elsevier Science Publishers, 2000.

[4] Pierre-Etienne Moreau and Hélène Kirchner. A compiler for rewrite programs in associative-commutative theories. In *Principles of Declarative Programming*, number 1490 in Lecture Notes in Computer Science, pages 230–249. Springer-Verlag, September 1998.

[5] Pierre-Etienne Moreau, Christophe Ringeissen, and Marian Vittek. A Pattern Matching Compiler for Multiple Target Languages. In G. Hedin, editor, *12th Conference on Compiler Construction, Warsaw (Poland)*, volume 2622 of *LNCS*, pages 61–76. Springer-Verlag, May 2003.

[6] Antoine Reilles. Canonical abstract syntax trees. In *Proceedings of the 6th International Workshop on Rewriting Logic and its Applications*. Electronic Notes in Theoretical Computer Science, 2006. to appear.

[7] Mark van den Brand, Jan Heering, Paul Klint, and Pieter Olivier. Compiling language definitions: The ASF+SDF compiler. *ACM Transactions on Programming Languages and Systems*, 24(4):334–368, 2002.

[8] Mark van den Brand, Pierre-Etienne Moreau, and Jurgen Vinju. A generator of efficient strongly typed abstract syntax trees in java. *IEE Proceedings - Software Engineering*, 152(2):70–78, December 2005.

[9] Eelco Visser and Zine-el-Abidine Benaissa. A core language for rewriting. In Claude Kirchner and Hélène Kirchner, editors, *Proceedings of the 2nd International Workshop on Rewriting Logic and its Applications (WRLA'98)*, volume 15, Pont-à-Mousson (France), September 1998. Electronic Notes in Theoretical Computer Science.

[10] Eelco Visser, Zine-el-Abidine Benaissa, and Andrew Tolmach. Building program optimizers with rewriting strategies. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 13–26. ACM Press, September 1998.

[11] Joost Visser. Visitor combination and traversal control. In *Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications (OOPSLA'01)*, pages 270–282, New York, NY, USA, 2001. ACM Press.

Regular Strategies as Proof Tactics for CIRC

Dorel Lucanu[1,2,4]

Faculty of Computer Science
Alexandru Ioan Cuza University
Iaşi, Romania

Grigore Roşu[3,5]

Department of Computer Science
University of Illinois
Urbana-Champaign, USA

Gheorghe Grigoraş[2,6]

Faculty of Computer Science
Alexandru Ioan Cuza University
Iaşi, Romania

Abstract

CIRC is an automated circular coinductive prover implemented as an extension of Maude. The main engine of CIRC consists of a set of rewriting rules implementing the circularity principle. The power of the prover can be increased by adding new capabilities implemented also by rewriting rules. In this paper we prove the correctness of the coinductive prover and show how rewriting strategies, expressed as regular expressions, can be used for specifying proof tactics for CIRC. We illustrate the strength of the method by defining a proof tactic combining the circular coinduction with a particular form of simplification for proving the equivalence of context-free processes.

Keywords:

1 Introduction

A behavioral algebraic specification is an algebraic specification where the sorts are split into *visible* (or *observational*) for data and *hidden* for states, and the equality is behavioral. Two states are *behaviorally equivalent* if and only if they

[1] This work is partially supported by the CEEX project 47/2005.
[2] This work is partially supported by the CNCSIS project 1162/2007.
[3] Partially supported by NSF grants CCF-0448501 and CNS-0509321.
[4] Email: dlucanu@info.uaic.ro
[5] Email: grosu@cs.uiuc.edu
[6] Email: grigoras@info.uaic.ro

appear to be the same under any visible *experiment*. The experiments are derived operations of visible result sort, and defined only with *behavioral operations* (*derivatives*). A canonical example is that of (infinite) streams. The sort *Stream* of streams is hidden, the sort of elements, e.g. *Int*, is visible, and the behavioral operations are the head `hd(*:Stream)` and the tail `tl(*:Stream)`. The special variable `*:Stream` marks the place of the state parameter. Examples of experiments are `hd(*:Stream)`, `hd(tl(*:Stream))`, `hd(tl(tl(*:Stream)))`, and so on. The behavioral equivalence (equality) over streams is given by $S \equiv S'$ iff $\texttt{hd}(S) = \texttt{hd}(S')$ and $\texttt{tl}(S) \equiv \texttt{tl}(S')$. If this is the case, then all the above experiments return the same visible data value for S and S', respectively. The behavioral equivalence coincides with the equality over the visible data.

The main issue in behavioral specification theory is how to prove that two states are behavioral equivalent. The coalgebraic *bisimulation* (see, e.g., Jacobs and Rutten [10]) as well as Hennicker's *context induction* [7] are both sound proof techniques for behavioral equivalence. Unfortunately, both of them need human intervention: coinduction to pick a "good" bisimulation relation, and context induction to device and prove auxiliary lemmas. *Circular coinduction* [4,14] is an automatic proof technique for behavioral equivalence, supported in BOBJ. By circular coinduction one can prove, for instance, the equality $zip(zeros, ones) = blink$ on streams as follows (*zeros* is the stream 0^ω, *ones* is 1^ω, *blink* is $(01)^\omega$, zip merges two streams):

(i) check that the two streams have the same head, 0;

(ii) take the tail of the two streams and generate the new goal $zip(ones, zeros) = 1\ blink$; this becomes the next task;

(iii) check that the two new streams have the same head, 1;

(iv) take the tail of the two new streams; after simplification one gets the new goal $zip(zeros, ones) = blink$, which is nothing but the original proof task;

(v) conclude that $zip(zeros, ones) = blink$ holds.

The intuition for the above "proof" is that the two streams have been exhaustively tried to be distinguished by iteratively checking their heads and taking their tails. Ending up in circles (we obtained the same new proof task as the original one) means that the two streams are indistinguishable, i.e., they are equal.

Circular coinduction can be explained and proved to be correct by reducing it to either bisimulation or context induction: it iteratively constructs a bisimulation, but it also discovers all lemmas needed by a context induction proof. Since the behavioral equivalence problem is Π^0_2-complete [14] (it is so, even in the context of just streams [15]), there is *no* algorithm or proof system that is complete for behavioral equality in general, as well as *no* algorithm or proof system that is complete for inequality of streams. Therefore, the best we can do is to focus our efforts on exploring heuristics or deduction rules to prove or disprove equalities of streams that *work well on examples of interest* rather than in general.

BOBJ [4,14] was the first system supporting circular coinduction. Hausmann, Mossakowski, and Schröder [6] also developed circular coinductive techniques and tactics in the context of CoCASL. CIRC [12,11] is an automated circular coinductive prover implemented in Full Maude [3] as a behavioral extension of the Maude

system [2], making heavy use of meta-level and reflection capabilities of rewriting logic. Maude is by now a very mature system, with many uses, a high-performance rewrite engine and a broad spectrum of analysis tools. Maude's current meta-level capabilities where not available when we developed the BOBJ system; consequently, BOBJ was a heavy system, with rather poor parsing and performance. By allowing the entire Maude system visible to the user, CIRC inherits all Maude's uses, performance and analysis tools. CIRC implements the *circularity principle*, which generalizes circular coinductive deduction [5] and can be expressed as follows. Assume that each equation of interest (to be proved) e admits a *frozen form* $fr(e)$ and a set of derived equations, its *derivatives*, $Der(e)$. The circularity principle says that if from hypotheses $\mathcal{H}$ together with $fr(e)$ we can deduce $Der(e)$, then e is a consequence of $\mathcal{H}$. When $fr(e)$ freezes the equation at the top, as in [5], the circularity principle becomes circular coinduction. Interestingly, when the equation is frozen at the bottom on a variable, then it becomes a structural induction (on that variable) derivation rule. This way, CIRC supports both coinduction and induction as projections of a more general principle. This paper makes two main contributions. First, we prove the correctness of CIRC's coinductive capabilities. Second, we define and implement a strategy language for CIRC.

Version 1.2 of CIRC [11] provides automatic proving support for both coinduction and induction, but not for combinations of them. A combination of the two techniques is possible only in an assisted way. In practice, there are many cases when the two must be combined with other techniques. CIRC's proof capabilities are implemented using rewriting rules. A proof tactic constraints the application of rules according to a given aim. To express the proof tactics, we use a strategy language based on regular expressions over rule labels. Regular expressions can be behaviorally specified; CIRC includes a copy of this specification to handle the proof tactics. This new feature is included CIRC v1.3. We illustrate it by defining a proof tactic able to prove the equivalence of context-free processes.

The paper is structured as follows. Section 2 introduces behavioral specification and presents the coinductive capabilities of the CIRC, and shows the correctness of implementation of the circular coinduction as rewriting rules. We use the regular expressions as an example of behavioral specification and we show that CIRC together with this specification supplies a fully automatic decision procedure for the equivalence of regular expressions. Section 3 introduces the regular strategies and defines proof tactics in terms of regular strategies. A proof tactic that combines coinduction with simplification of goals is presented. Section 4 shows how context-free processes can be behaviorally specified and presents a proof technique, based on coinduction and simplification, for proving the equivalence of context-free processes. The paper ends with concluding remarks.

2 Behavioral Algebraic Specifications

Let Σ be an algebraic signature consisting of a set $Sorts(\Sigma)$ of *sorts* and an $S^* \times S$-indexed set $Op(\Sigma) = (Op(\Sigma)_{w,s} \mid w \in S^*, s \in S)$ of *operations*. We assume $Sorts(\Sigma) = V \cup H$, where V is a subset of *visible sorts*, H is a subset of *hidden sorts*, and $V \cap H = \emptyset$. For each $f \in Op(\Sigma)_{w,s}$, we use the following notations:

arity$(f) = w$, *sort*$(f) = s$, and *type*$(f) = (w, s)$ (also written $w \to s$). Let $\mathcal{X}$ be a fixed S-indexed set of *variables*. $T_\Sigma(\mathcal{X})$ is the Σ-algebra of terms with variables in $\mathcal{X}$. Let *Var*(t) denote the set of variables occurring in term t. A Σ-*behavioral operation* for hidden sort $h \in H$, also called a *derivative*, is a term $\delta \in T_\Sigma(\mathcal{X} \cup \{*{:}h\})$, where $*{:}h$ is a special variable of sort h. The sort of δ is called δ's *result sort*. A Σ-*equation* is a sentence $(\forall X)\ t = t'$ `if` c, where t and t' are Σ-terms over variables $X \subseteq \mathcal{X}$ having the same result sort, and c is the *condition* of the equation consisting of a finite set of pairs (t_i, t_i') of terms over variables X and with *visible* result sorts; we only consider equations with finitely many visible conditions in this paper. A condition c is also written as $t_1 = t_1' \wedge \cdots \wedge t_n = t_n'$. If the sorts of t and t' are hidden, then the equation is called *behavioral*. If the condition c is empty, then we get an unconditional Σ-equation and write it as $(\forall X)\ t = t'$.

A *behavioral specification* is a triple $\mathcal{B} = (\Sigma, \Delta, E)$, where Σ is an algebraic signature, Δ is a set of behavioral operations, and E is a set of Σ-equations. Let $=_E$ denote the standard equational derivability congruence over $T_\Sigma(\mathcal{X})$ with the equations E (the E-equality), and let $T_{\Sigma,E}(\mathcal{X})$ denote the quotient $T_\Sigma(\mathcal{X})/{=_E}$. A Δ-*experiment* for the hidden sort $h \in H$ is inductively defined as follows: each behavioral operation for the hidden sort $h \in H$ with visible result sort is a Δ-experiment for h; if γ is a Δ-experiment for h' and δ a behavioral operation for h with result sort h', then $\gamma[\delta/*{:}h']$ is a Δ-experiment for h.

The notion of *behavioral equivalence* is an inherently semantic one: there is a behavioral equivalence relation on each model which can be defined as "indistinguishability under experiments". For technical simplicity, we here prefer to avoid introducing models, so we give an alternative, proof theoretic definition. The Δ-*behavioral equivalence* $\equiv_\Delta$ over $T_\Sigma(\mathcal{X})$ is the E-equality (closure under equational deduction with E) generated by the following: for each visible sort $v \in V$, $\equiv_{\Delta,v}$ is $=_{E,v}$; if $h \in H$ and $t, t' \in T_\Sigma(\mathcal{X})_h$ then $t \equiv_\Delta t'$ iff $\gamma[t/*{:}h] =_E \gamma[t'/*{:}h]$ for each Δ-experiment γ for h. (We here therefore assume that operations are behaviorally congruent, i.e., $f(t_1, \ldots, t_n) \equiv_\Delta f(t_1', \ldots, t_n')$ whenever $t_i \equiv_\Delta t_i'$ for $i = 1, \ldots, n$.) We often write $\gamma[t]$ for $\gamma[t/*{:}h]$. Note that the relation $\equiv_\Delta$ is not algorithmic, because one needs an infinite number of experiments to decide it; the basis for the Π_2^0 result in [14] is the observation that *for each* experiment *there is* some proof (thanks to the completeness of equational deduction). Moreover, the E-equality is undecidable for the general case. $\mathcal{B}$ *behaviorally entails* the behavioral Σ-equation e, written $\mathcal{B} \models\!\!\models e$, iff

(i) either e is of the form $(\forall X)\ t = t'$ (the condition is empty) and $t \equiv_\Delta t'$,

(ii) or e is of the form $(\forall X)\ t = t'$ `if` c with c consisting of $t_1 = t_1' \wedge \cdots \wedge t_n = t_n'$ and $\mathcal{B}(c) \models\!\!\models (\forall X)\ t = t'$, where $\mathcal{B}(c)$ is the behavioral specification $(\Sigma \cup X, \Delta, E \cup \{(\forall \emptyset) t_i = t_i' \mid i = 1, \ldots, n\})$ (the variables in X are added as constants, then the condition is added as hypothesis).

The *equational entailment* is defined in a similar way: $\mathcal{B} \models e$, iff

(i) either e is of the form $(\forall X)\ t = t'$ and $t =_E t'$,

(ii) or e is of the form $(\forall X)\ t = t'$ `if` c and $\mathcal{B}(c) \models (\forall \emptyset)\ t = t'$.

Example: Regular Expressions.

Regular expressions (RE) are finite presentations for possibly infinite languages. The languages denoted by REs are inductively defined using the operations union, concatenation, and Kleene closure. Formally, let *Alph* be an alphabet; the set of *regular expressions* over *Alph* is given by the grammar

$$R ::= \varepsilon \mid \emptyset \mid A \mid R_1 + R_2 \mid R_1 \# R_2 \mid R^*$$

where A ranges over *Alph*. The *language* denoted by a RE R is defined as follows: $\mathcal{L}(\varepsilon) = \{\varepsilon\}$, $\mathcal{L}(\emptyset) = \emptyset$, $\mathcal{L}(A) = \{A\}$, $\mathcal{L}(R_1 + R_2) = \mathcal{L}(R_1) \cup \mathcal{L}(R_2)$, $\mathcal{L}(R_1 \# R_2) = \{ww' \mid w \in \mathcal{L}(R_1), w' \in \mathcal{L}(R_2)\}$, and $\mathcal{L}(R^*) = (\mathcal{L}(R))^*$. Two REs are *equivalent* iff they denote the same language. In [16,17] a behavioral specification for (Extended) REs is given, where the behavioral equivalence coincides with the RE equivalence. The behavioral operations (derivatives) defining behavioral equivalence are `epsIn_` (testing the membership of ε to a RE) and `_{_}`, which takes a RE R and a letter a and returns an expression characterized by $L(R\{a\}) = \{w \mid aw \in L(R)\}$. $R\{a\}$ is semantically equivalent to an RE because regular languages are closed under left-quotient by an arbitrary language. Here is a Maude description of REs:

```
(th RE is including BOOL .
  sort Ere .                                  --- extended regular expressions
  sort Alph .                                 --- the alphabet
  ops a b : -> Alph .                         --- the letters of the alphabet

  vars R R1 R2 : Ere .    vars A B : Alph .

  op  _`{_`} : Ere Alph -> Ere .              --- the letters derivatives
  op epsIn_ : Ere -> Bool .                   --- epsilon membership derivative

  subsort Alph < Ere .                        --- a letter
  eq epsIn A = false .
  eq B { A } = if A == B then epsilon else empty fi .

  op epsilon : -> Ere .                       --- the empty word
  eq epsilon { A } = empty .
  eq epsIn epsilon = true .

  op empty : -> Ere .                         --- the empty language
  eq empty { A } = empty .
  eq epsIn empty = false .

  op _#_ : Ere Ere -> Ere [assoc] .           --- concatenation
 ceq ( R1 # R2 ){ A } = ((R1 { A }) # R2) + (R2 { A })
     if epsIn R1 = true .
 ceq ( R1 # R2 ){ A } = (R1 { A }) # R2
     if epsIn R1 = false .
  eq epsIn ( R1 # R2 ) =  epsIn R1 and  epsIn R2 .

  op _+_ : Ere Ere -> Ere [assoc comm] .      --- union
  eq ( R1 + R2 ){ A } = (R1 { A }) + (R2 { A }) .
  eq epsIn ( R1 + R2 ) = epsIn R1 or  epsIn R2 .

  op _* : Ere -> Ere .                        --- star operator
  eq R * { A } = (R { A }) # (R *) .
  eq epsIn R * = true .

  --- simplifying equations
  eq empty + R = R .   eq empty # R = empty .   eq epsilon # R = R .   eq R + R = R .
endth)
```

2.1 Circular Coinduction

As mentioned in the introduction, CIRC implements the principle of circularity, which generalizes both structural induction and circular coinduction; we will discuss this principle in depth elsewhere. We here focus on its coinductive instance. Circular coinduction [4,5,14], is a sound proof calculus for $\models$, which can be defined

as an instance of the circularity principle as follows: the "frozen" form of equation "$(\forall X)\ t = t'$ `if` c" is "$(\forall X) fr(t) = fr(t')$ `if` c", where $fr : sort(t) \to$ `new` is a new operation and `new` is a new sort. The set $Der_\Delta(e)$ is

$$\{(\forall X)\ fr(\delta[t/*{:}h]) = fr(\delta[t'/*{:}h])\ \texttt{if}\ c \mid \delta \text{ behavioral for } h = sort(t)\}.$$

The frozen operator ensures the sound use of the coinduction hypotheses. We take the liberty to also call $fr(e)$ *visible* when e is visible.

In CIRC we use the standard rewriting-based semi-decision procedure to derive equations "$(\forall X)\ t = t'$ `if` c": add the variables X as constants, then add the conditions in c to the set of equations, and then reduce t, t' to normal forms orienting all the equations into rewrite rules. In what follows we let $\mathcal{E} \vdash e$ denote the fact that e can be deduced from $\mathcal{E}$ using this standard approach ($\mathcal{E}$ is any set of equations). The inference relation $\vdash$ is sound for $\models$, i.e., $\mathcal{E} \vdash e$ implies $\mathcal{E} \models e$. We currently do not interfere with the rewriting procedure: if rewriting does not terminate during a proof session then CIRC does not terminate either. If we write $\mathcal{E} \nvdash e$ then we mean "knowingly incapable of proving it", that is, that the rewrite engine reduced the two terms to normal forms, but those are not equal. Obviously, this does not necessarily mean that the equation is not true.

CIRC implements circular coinduction as a nondeterministic procedure aiming at reducing a pair $(E, fr(e))$ to a pair $(\mathcal{E}, \emptyset)$, where E is the original set of equations in $\mathcal{B}$ and e is the equation to prove. If that is the case, then $\mathcal{B} \models e$. While trying to do so, the procedure can also fail, in which case we conclude that it could not prove $\mathcal{B} \models e$, or it can run forever. Here are the reduction rules:

[EqRed] :
$(\mathcal{E}, \mathcal{G} \cup \{fr(e)\}) \Rightarrow (\mathcal{E}, \mathcal{G})$ if $\mathcal{E} \vdash fr(e)$
[CoindFail] :
$(\mathcal{E}, \mathcal{G} \cup \{fr(e)\}) \Rightarrow \mathit{failure}$ if $\mathcal{E} \nvdash fr(e)$ and e is visible
[CCStep] :
$(\mathcal{E}, \mathcal{G} \cup \{fr(e)\}) \Rightarrow (\mathcal{E} \cup \{nf(fr(e))\}, \mathcal{G} \cup Der_\Delta(e))$ if $\mathcal{E} \nvdash fr(e)$ and e is hidden.

Let $nf(e)$ denote the equation e where the left-hand and right-hand sides are reduced to normal forms. [EqRed] removes a goal if it can be proved using ordinary equational reduction. [CoindFail] says that the procedure fails whenever it finds a visible goal which cannot be proved using ordinary equational reduction. Finally, [CCStep] implements the circularity principle: when a behavioral equation cannot be proved using ordinary equational reduction, its frozen form (or an equivalent variant of its frozen form, such as its normal form) is added to the specification and its derivatives are added to the set of goals.

Theorem 2.1 *Let $\mathcal{B} = (\Sigma, \Delta, E)$ be a behavioral specification and let e be a Σ-equation such that $(E, fr(e)) \Rightarrow^\star (\mathcal{E}, \emptyset)$ using the procedure above. Then $\mathcal{B} \models e$.*

The proof of Theorem 2.1 is based on the following two lemmas.

Lemma 2.2 *If $(\Sigma \cup \{fr\}, \Delta, E) \models fr(e)$, then $(\Sigma, \Delta, E) \models e$.*

Proof. We assume without lose the generality that e is an unconditional equation $(\forall X)t = t'$. If γ is a Δ-experiment, then we have $fr(\gamma[t]) =_E fr(\gamma[t'])$ iff $\gamma[t] =_E \gamma[t']$ by the definition of $=_E$ and by the fact that E does not include equations involving fr. □

Lemma 2.3 *If* $(\Sigma, \Delta, \mathcal{E} \cup \{fr(e)\}) \models Der_\Delta(e)$, *then* $(\Sigma, \Delta, \mathcal{E}) \models\!\models e$.

Proof. We assume again that e is an unconditional equation $(\forall X)t = t'$. Let $<$ be the order over the experiments given by their depth. We show by Noetherian induction on $<$ that $\gamma[u] =_E \gamma[v]$, where u (resp. v) is either t (resp. t') or $\theta(t)$ (resp. $\theta(t')$), where θ is any substitution. Let γ be an appropriate experiment for t (and t'). If $\gamma[t] = \gamma[t']$ is in $Der_\Delta(e)$, then we get $\gamma[t] =_E \gamma[t']$ by the hypothesis of the lemma ($fr(e)$ cannot be used in any derivation of $\gamma[t] = \gamma[t']$). Otherwise, there is a derivative $\delta \in \Delta$ and an experiment γ' such that $\gamma = \gamma'[\delta]$ and $\delta[t] = \delta[t']$ in $Der_\Delta(e)$. If $fr(\delta[t]) =_E fr(\delta[t'])$ ($fr(e)$ is not used in this derivation), then $fr(\gamma'[\delta[t]]) =_E fr(\gamma'[\delta[t']])$ and hence $\gamma[t] =_E \gamma[t']$. If $fr(e)$ is used in the derivation of $fr(\delta[t]) = fr(\delta[t'])$, then there is a substitution θ such that $fr(\delta[t]) =_E fr(\theta(t))$ and $fr(\delta[t']) =_E fr(\theta(t'))$. Since $\gamma' < \gamma$, we have $\gamma'[\theta(t)] =_E \gamma'[\theta(t')]$. Hence $\gamma[t] =_E \gamma[t']$. □

Corollary 2.4 *If* $(\Sigma, \Delta, \mathcal{E} \cup \{fr(e)\}) \models \mathcal{G} \cup Der_\Delta(e)$, *then* $(\Sigma, \Delta, \mathcal{E}) \models\!\models \mathcal{G} \cup \{e\}$.

Proof of Theorem 2.1.

We proceed by induction on the number of applications of [CCStep]. If this rule is not applied, then $(\Sigma, \Delta, \mathcal{E} \cup \{fr(e)\}) \models Der_\Delta(e)$ and the conclusion of the theorem follows by Lemma 2.3. We assume that

$$(E, fr(e)) \Rightarrow^\star (\mathcal{E}', \mathcal{G}' \cup \{fr(e')\}) \Rightarrow (\mathcal{E}' \cup \{nf(fr(e'))\}, \mathcal{G}' \cup Der_\Delta(e')) \Rightarrow^\star (\mathcal{E}, \emptyset)$$

where the last reductions are given using only [EqRed]. Hence $(\Sigma, \Delta, \mathcal{E}' \cup \{fr(e')\}) \models \mathcal{G}' \cup Der_\Delta(e')$. It follows that $(\Sigma, \Delta, \mathcal{E}') \models\!\models \mathcal{G}' \cup \{e'\}$ by the corollary of Lemma 2.3. The conclusion of Theorem 2.1 follows now applying the induction hypothesis. □

The *successful termination* of the CIRC procedure above, i.e., reaching of a configuration of the form $(\mathcal{E}', \emptyset)$, is not guaranteed. Let us consider, for instance, the addition of streams, behaviorally defined by $hd(S + S') = hd(S) + hd(S')$ and $tl(S + S') = tl(S) + tl(S')$, and the (convolution) product of streams, behaviorally defined by $hd(S \times S') = hd(S) \times hd(S')$ and $tl(S \times S') = tl(S) \times S' + [hd(S)] \times tl(S')$, where $[x]$ denotes the stream $x0^\omega$. The execution of CIRC procedure for the input goal $[0] \times [0] = [0]$ produces an infinite process. First [CCStep] is applied, which replace the initial goal $fr([0] \times [0]) = fr([0])$ with $fr(hd([0] \times [0])) = fr(hd([0]))$ and $fr(tl([0] \times [0])) = fr(tl([0]))$. The former is solved by [EqRed] and the latter is reduced by [EqRed] to $fr([0] \times [0] + [0] \times [0]) = fr([0])$. A new application of [CCStep] followed by [EqRed] (twice) will generate the goal $fr([0] \times [0] + [0] \times [0] + [0] \times [0] + [0] \times [0]) = fr([0])$. The process infinitely continues generating larger and larger goals. In Section 3 we extend CIRC with proof tactics able to handle such cases.

Since the behavioral entailment problem is Π^0_2-complete [14,15], we know that there can be no procedure to decide behavioral equalities or inequalities in general.

The reaching of the configuration *failure* means a *failing termination*.

For regular expressions, we are in the happy case when CIRC together with the specification `RE` yield a fully automatic decision procedure for their equivalence:

Proposition 2.5 *If one defines the behavioral operators to be the two derivatives and the test for epsilon membership, then CIRC becomes a fully automatic decision procedure for the equivalence of REs:*

(i) *Regular expressions R_1 and R_2 are equivalent iff CIRC successfully terminates for the initial configuration $(Eqns(\texttt{RE}), \{R_1 = R_2\})$;*

(ii) *Regular expressions R_1 and R_2 are not equivalent iff CIRC fails for the initial configuration $(Eqns(\texttt{RE}), \{R_1 = R_2\})$.*

Therefore, no case analysis or other assisted tactics are needed to prove the equivalence of REs. Two REs are equivalent iff CIRC returns `Proof succeeded`, and are not equivalent iff CIRC returns `failed during coinduction`.

Here we show CIRC at work presenting how it proves the equivalence of two regular expressions. CIRC extends Full-Maude [3] with a set of declarations and commands which allow the user to introduce information regarding behavioral specifications and commands to assist the prover. First we have to introduce the behavioral operations. These are included in a `cmod...endcm` module:

```
(cmod B-RE is importing RE .
    derivative epsIn(*:Ere) .
    derivative *:Ere { a } .
    derivative *:Ere { b } .
endcm)
```

After the modules `RE` and `B-RE` are loaded, we introduce the goal we want to prove:

```
Maude> in re
Introduced theory RE
Introduced beh spec B-RE
Maude> (add goal (a + b)*  = ((a *)#(b *))*  .)
Goal (a + b)*  = (a * # b *)*  added.
```

The circular coinduction algorithm is triggered with the command:

```
Maude> (coinduction .)
Proof succeeded.
```

3 Regular Strategies as Proof Tactics

A successful reduction of the circular coinduction algorithm is of the form

$$(\mathcal{E}_0, \mathcal{G}_0) \xrightarrow{\text{CCStep}} (\mathcal{E}_1, \mathcal{G}_1) \ldots \xrightarrow{\text{EqRed}} (\mathcal{E}_i, \mathcal{G}_i) \ldots \xrightarrow{\text{CCStep}} (\mathcal{E}_j, \mathcal{G}_j) \ldots \xrightarrow{\text{EqRed}} (\mathcal{E}_n, \emptyset)$$

and a failing reduction is of the form

$$(\mathcal{E}_0, \mathcal{G}_0) \xrightarrow{\text{CCStep}} (\mathcal{E}_1, \mathcal{G}_1) \ldots \xrightarrow{\text{EqRed}} (\mathcal{E}_i, \mathcal{G}_i) \ldots \xrightarrow{\text{CCStep}} (\mathcal{E}_j, \mathcal{G}_j) \ldots \xrightarrow{\text{CoindFail}} \mathit{failure}$$

If we consider only the labels of the rules applied in the reduction, then the former one is described by a word in the language given by the regular expression $R_{succ} = \text{CCStep}(\text{CCStep} + \text{EqRed})^*\text{EqRed}$, and the later one by a word given by the regular expression $R_{fail} = \text{CCStep}(\text{CCStep} + \text{EqRed})^*\text{CoindFail}$. We say that the regular expression $R_{succ} + R_{fail}$ describes the circular coinduction proof tactic.

CIRC can be extended with new proof capabilities implemented as rewriting rules and new proof tactics described by regular expressions. We assume that the prover is given by a set of rewriting rules of the form $\ell : C \to C'$ `if` *cond*, where ℓ is the label of the rule, C, C' are configurations, and *cond* is the condition of the rule. Let $\mathcal{L}$ denote the set of the rules labels. The same label may be shared by more rules. This is, e.g., the case of two rules whose conditions are complementary, i.e., if one condition is true, then the other one is false. The description of the proof could become simpler if we use the same label for the two rules. We say that a word $w \in \mathcal{L}^*$ describes a reduction $C \xrightarrow{*} C'$ iff $w = \ell_1 \dots \ell_n$ and the reduction is given by $C = C_0 \xrightarrow{\ell_1} C_1 \dots \xrightarrow{\ell_n} C_n = C'$, where $C_{i-1} \xrightarrow{\ell_i} C_i$ means that the configuration C_i is obtained from C_{i-1} by applying the rule ℓ_i. We write $C \xrightarrow{w} C'$.

A *regular strategy term* is a regular expression over $\mathcal{L}$. The *semantics* of a strategy term R consists of all the reductions $C \xrightarrow{w} C'$ with $w \in L(R)$.

We extend the definition of configurations by adding a new component given by a regular strategy term. A rewriting rule

$$\ell : C \to C' \texttt{ if } cond$$

is replaced by

$$\ell : (C, R) \to (C, R') \texttt{ if } cond \wedge nf(R\{\ell\}) = R' \wedge R' \neq empty.$$

where $nf(R\{\ell\}) = R' \wedge R' \neq empty$ means that there is a word starting with ℓ in $L(R)$. We have $(C, R) \xrightarrow{w} (C', \varepsilon)$ if and only if $w \in L(R)$.

Let $\mathcal{B} = (\Sigma, \Delta, \mathcal{E})$ be a behavioral specification. We say that a regular strategy term R is a *proof tactic* for $\mathcal{B}$ if $\mathcal{B} \models e$ whenever $(E, \{fr(e)\}, R) \xrightarrow{w} (\mathcal{E}, \emptyset, \varepsilon)$.

3.1 Simplification

Since circular coinduction uses frozen forms of the goals, the coinductive hypotheses added by [CCStep] can be applied only at the top. There are cases when it is sound to apply these hypotheses under the top. For instance, we assume that we have to show the following equality over streams: $S \times [0] = [0]$. [CCStep] will add the hypothesis $fr(S \times [0]) = fr([0])$, and replace the above goal with $fr(hd(S \times [0])) = fr(hd([0]))$ and $fr(tl(S \times [0])) = fr(tl([0]))$. The former is solved by [EqRed] and the last is reduced to $fr(tl(S) \times [0] + [hd(S)] \times [0]) = fr([0])$. It is easy to see that the circular coinduction algorithm produces a infinite set of new goals in this case. For streams it is sound to simplify the goals with the following rule:

$$\frac{S_1 + S_2 = S'}{S_1 = S'} \quad \text{if } S_2 = [0]$$

Then the above goal can be simplified to $fr(tl(S) \times [0]) = fr([0])$ because we get $fr([hd(S)] \times [0]) = fr([0])$ applying the coinductive hypothesis. The remaining goal is proved in the same way.

The following rule handles the general case:

[Simpl] :
$(\mathcal{E}, \mathcal{G} \cup \{fr(e)\}) \Rightarrow (\mathcal{E}, \mathcal{G}')$ if *scond*
where

$$\mathcal{G}' = \begin{cases} \mathcal{G} \cup \{t_i = t'_i \texttt{ if } c \mid i = 1, \ldots, n\} & \text{if } e := f(t_1, \ldots, t_n) = f(t'_1, \ldots, t'_n) \texttt{ if } c \\ \mathcal{G} \cup \{fr(e)\} & \text{otherwise} \end{cases}$$

where *scond* is a condition which constraints the application of the rule (in the above example, $S_2 = [0]$ is such a condition). [Simpl] replaces a goal with the corresponding set of subgoals if the left hand side and right hand side of the equation e have in top the operator f, and let the set of goals unchanged in the other cases.

The rule [Simpl] must be used only in a controlled way, otherwise it could trigger infinite reductions, due to the "otherwise" case.

Theorem 3.1 *Let $\mathcal{B} = (\Sigma, \Delta, \mathcal{E})$ be a behavioral specification, and let $f \in \Sigma$ be an operation such that the proving of the goal $f(t_1, \ldots, t_n) = f(t'_1, \ldots, t'_n)$ $\texttt{if } c$ can be simplified to the proving of the subgoals $\{t_i = t'_i \texttt{ if } c \mid i = 1, \ldots, n\}$ provided that the simplification condition scond holds. Then R_{cosi} = CCStep(Simpl^n # CCStep + Simpl^n # EqRed)*EqRed is a proof tactic for $\mathcal{B}$.*

Proof. Let e be an equation such that $(E, \{e\}, R_{cosi}) \xrightarrow{w} (\mathcal{E}, \emptyset, \varepsilon)$ with $w \in L(R_{cosi})$. We proceed by induction on the number representing how many times [Simpl] is applied. If [Simpl] is not applied, then $w \in L(R_{succ})$ and we apply Theorem 2.1. We point out one application of [Simpl] which replaces a goal e' of the form $f(t_1, \ldots, t_n) = f(t'_1, \ldots, t'_n)$ $\texttt{if } c$ with $\{t_i = t'_i \texttt{ if } c \mid i = 1, \ldots, n\}$. We have $\mathcal{B} \models t_i = t'_i$ $\texttt{if } c$ by the inductive hypothesis, for $i = 1, \ldots n$. Hence $\mathcal{B} \models e'$ by the assumption on $\mathcal{B}$ and f. □

[Simpl] is applied successively more than once when a goal of the form $f(f(\ldots)) = f(f(\ldots))$ $\texttt{if } c$ is reached. The choice of n could be a difficult task. A bigger value for n leads to a larger class of goals which can be proved by this tactic but to a less computationally efficient tactic; a smaller value leads to a more computationally efficient tactic but with a more restricted applicability. We may have both if we implement [Simpl] as follows:

[Simpl] :
$(\mathcal{E}, \mathcal{G} \cup \{fr(e)\}, R) \Rightarrow (\mathcal{E}, \mathcal{G}', R')$ $\texttt{if } nf(R\{\text{Simpl}\}) \neq empty \wedge scond$
where

$$(\mathcal{G}', R') = \begin{cases} (\mathcal{G} \cup \{t_i = t'_i \texttt{ if } c \mid i = 1, \ldots, n\}, R) & \text{if } e \text{ is } f(t_1, \ldots, t_n) = f(t'_1, \ldots, t'_n) \texttt{ if } c \\ (\mathcal{G}, nf(R\{\text{Simpl}\})) & \text{otherwise} \end{cases}$$

The above rule applies [Simpl] as long as it changes the configuration. We may consider now $n = 1$ in Theorem 3.1. In terms of strategy languages [13,18], the new

rule [Simpl] is equivalent to [Simpl'] `orelse id`, where [Simpl'] is similar to [Simpl] but it is applied only if it simplifies a goal, and `id` is the identity strategy.

[Simpl] is implemented in CIRC (version 1.3) by extending the configuration with attribute *proofStatus* having as values regular strategies R. A copy of the specification `RE` is included and used to handle the regular strategies.

4 Equivalence of Context Free Processes

In this section we illustrate the usefulness of the regular strategies showing how the result of Proposition 2.5 can be extended to context-free grammars. We have to notice that we cannot obtain a fully automatic decision procedure for context-free grammars because the equivalence problem for these grammars is undecidable [8]. Baeten, Bergstra, and Klop [1] have shown that the equivalence problem is decidable for normalized context free processes presented in Greibach normal forms. These processes are equivalent to simple (iredundant) context-free grammars. However the proof of decidability and the presentation of the algorithms are not easy and many researchers looked for simpler solutions, see , e.g., [9]. In this section we show that an adequate behavioral specification together with two simplifying rules and the proof tactic given in Theorem 3.1 supply a fully automatic semi-decision procedure for context free-processes. We prefer the formalism of context-free processes to that of context-free grammars because their equivalence is expressed as a bisimulation relation, which is more suitable for the behavioral approach.

BPA (Basic Process Algebra) *expressions* are defined by the following grammar:

$$E ::= a \mid X \mid E_1 + E_2 \mid E_1 E_2$$

where a ranges over an alphabet $Alph$, X over variables, and E, E_1, E_2 over BPA expressions. A *process* is defined by a finite set of equations of the form $X_i \stackrel{\text{def}}{=} E_i$. The operational semantics of BPA process is given by the following rules:

$$\frac{E_1 \stackrel{a}{\to} E_1'}{E_1 + E_2 \stackrel{a}{\to} E_1'} \qquad \frac{E_2 \stackrel{a}{\to} E_2'}{E_1 + E_2 \stackrel{a}{\to} E_2'} \qquad \frac{E_1 \stackrel{a}{\to} E_1'}{E_1 E_2 \stackrel{a}{\to} E_1' E_2}$$

$$a \stackrel{a}{\to} \varepsilon \quad \text{if } a \in Alph \qquad \frac{E \stackrel{a}{\to} E'}{X \stackrel{a}{\to} E'} \quad \text{if } X \stackrel{\text{def}}{=} E$$

Note that ε is not a process; it is a configuration used to mark the end of a transition process. The transition relation is extended to words w in $Alph^*$ as follows: $p \stackrel{aw}{\longrightarrow} q$ if $p \stackrel{a}{\to} p'$ and $p' \stackrel{w}{\to} q$. The *norm* $|E|$ of an expression E is defined by structural induction over BPA expressions: $|a| = 1$ if $a \in Alph$, $|E_1 + E_2| = \min(|E_1|, |E_2|)$, $|E_1 E_2| = |E_1| + |E_2|$, and $|X| = |E|$ if $X \stackrel{\text{def}}{=} E$. The norm has the following nice property: $|X| = \min\{length(w) \mid X \stackrel{w}{\to} \varepsilon\}$. A process is *normed* if $|X_i|$ is finite for each equation $X_i \stackrel{\text{def}}{=} E_i$ from its definition. A relation R over processes is a *bisimulation* if it satisfies: $p \, R \, q$ implies

(i) if $p \stackrel{a}{\to} p'$ then there is q' such that $q \stackrel{a}{\to} q'$ and $p' \, R \, q'$;

(ii) if $q \stackrel{a}{\to} q'$ then there is p' such that $p \stackrel{a}{\to} p'$ and $p' \, R \, q'$.

Two processes are *bisimilar* $p \sim q$ if there is a bisimulation R such that $p\,R\,q$. The language defined by a variable X is $L(X) = \{w \mid X \xrightarrow{w} \varepsilon\}$. If $X \sim Y$, then $L(X) = L(Y)$.

Here is the behavioral specification we associate to BPA:

```
(th BPA is inc BOOL + INT .
  sort Alph .          --- the alphabet
  sort Pexp .          --- process expressions
  sort Pid .           --- process ids
  sort Peq .           --- process equations
  sort Proc .          --- processes (sets of process equations)
  op  pmain : -> Proc .

  vars E E1 E2 : Pexp .  vars X Y X1 X2 : Pid .  vars P P1 P2 Q : Proc .  vars A B : Alph .

  op _=def_ : Pid Pexp -> Peq .

  subsort Peq < Proc .
  op _`,_ : Proc Proc -> Proc [assoc comm] .
  eq (P , P) = P .

  op  _`{_`}  : Pexp Alph -> Pexp .  --- the letters derivatives
  op |_| : Pexp -> Int .             --- the norm derivative

  sort PidSet .                      --- process variables
  subsort Pid < PidSet .             --- sets of process variables
  op none : -> PidSet .
  op _`,_ : PidSet PidSet -> PidSet [assoc comm id: none] .
  vars PS PS' : PidSet .
  eq (X, X) = X .

  op f : PidSet Pexp -> Int? .       --- auxiliary function
  eq f((X, PS), X) = infty .
  ceq f((X, PS), Y) = f((X, Y, PS), E)
      if ( (Y =def E), P ) := pmain /\ X =/= Y .
  eq f(PS, A ) = 1 .
  eq f(PS, (E1 + E2)) = min(f(PS, E1), f(PS, E2)) .
  eq f(PS, (E1 # E2)) = f(PS, E1) + f(PS, E2) .

  subsort Alph < Pexp .              --- a letter
  subsort Pid < Pexp .               --- a process id
  eq | A | = 1 .

  op epsilon : -> Pexp .
  eq epsilon { A } = deadlock .

  op deadlock : -> Pexp .
  eq deadlock { A } = deadlock .

  eq B { A } = if A == B then epsilon else deadlock fi .

  op _+_ : Pexp Pexp -> Pexp [prec 33 assoc comm] .  --- union
  eq ( E1 + E2 ){ A } = (E1 { A }) + (E2 { A }) .
  eq | E1 + E2 | = min(| E1 |, | E2 |) .

  op _#_ : Pexp Pexp -> Pexp [prec 32 assoc] .       --- concatenation
  eq ( E1 # E2 ){ A } = (E1 { A }) # E2 .
  eq | E1 # E2 | = | E1 | + | E2 | .

  ceq X { A } = E { A } if ( (X =def E), P ) := pmain .
  ceq | X | = f(X, E) if ( (X =def E), P ) := pmain .

--- simplifying equations
  eq deadlock + E = E .
  eq deadlock # E = deadlock .
  eq epsilon # E = E .
  eq E + E = E .
  eq (E1 + E2) # E = (E1 # E) + (E2 # E) .
 endth)
```

The behavioral operators are the letter derivatives, defined in a similar way to those of regular expressions, and the norm. The sort for process variables is `Pid`. The definition of the norm requires an auxiliary function $f(X, E)$, which returns the norm of the variable X with the definition E.

The following two simplification rules are sound for normed processes [9]:

$$\frac{E_1E_2 \sim E_1'E_2}{E_1 \sim E_1'} \qquad \frac{E_1 + E_2 \sim E_1' + E_2'}{E_1 \sim E_1',\ E_2 \sim E_2'} \text{ if } |E_1| = |E_1'| \wedge |E_2| = |E_2'|$$

We present here how CIRC [7] together with the proof tactic R_{cosi} given by Theorem 3.1 are used for proving the equivalence $X \sim A$, where $X \stackrel{\text{def}}{=} aXb + ab$, $A \stackrel{\text{def}}{=} aB + aY$, $B \stackrel{\text{def}}{=} Ab$, and $Y \stackrel{\text{def}}{=} b$. Note that processes are not required to be in Greibach normal form. Here is a Maude specification of the above process:

```
(th BPA-EX is including BPA .
  ops a b : -> Alph .   ops X Y A B : -> Pid .
  eq pmain = ( X =def ( a # X  # b ) + a # b ),
             ( A =def (a # B) + (a # Y) ),
             ( B =def A # b ),
             ( Y =def b ) .
endth)
```

The simplification rules are specified together with the derivatives in the `cmod` module:

```
(cmod B-BPA-EX is importing BPA-EX .
    derivative *:Pexp { a } .
    derivative *:Pexp { b } .
    derivative | *:Pexp | .
    simplify
      < E1:Pexp + E2:Pexp = E1':Pexp + E2':Pexp >
    by
      < E1:Pexp = E1':Pexp > /\
      < E2:Pexp = E2':Pexp >
    if
      | E1:Pexp | = | E1':Pexp | /\
      | E2:Pexp | = | E2':Pexp | .
    simplify
      < E1:Pexp # E2:Pexp = E1':Pexp # E2':Pexp >
    by
      < E1:Pexp = E1':Pexp >
    if
      E2:Pexp = E2':Pexp .
endcm)
```

We introduce the goal:

```
Maude> (add goal X  = A  .)
Goal X  =  A  added.
```

and we specify the proof tactic R_{cosi} by the following command:

```
Maude> (coinduction and simplification .)
Proof succeeded.
```

We can see the proof steps applied by the prover using "`show history .`" command:

```
Maude> (show history .)
Introduced beh spec B-BPA-EX
Goal X  =  A  added.
Hypothesis X  =  A  added.
Goal X{a} =  A{a} reduced to b + X # b  =  B + Y
Goal b + X # b  =  B + Y  simplified to
1 .  X # b  =  B
2 .  b  =  Y
Hypothesis X # b  =  B  added.
Goal b  =  Y  reduced to b  =  Y
Hypothesis b  =  Y  added.
Goal X{b} =  A{b} proved by reduction.
Goal | X |  =  | A |  proved by reduction.
Goal X # b{a} =  B{a} reduced to
b # b + X # b # b  =  B # b + Y # b
Goal b # b + X # b # b  =  B # b + Y # b  simplified to
1 .  X # b # b  =  B # b
2 .  b # b  =  Y # b
```

[7] Regular strategies are included in the version 1.3 of CIRC.

```
Goal X # b # b  =  B # b  simplified to X # b  =  B
Goal X # b  =  B  proved by reduction.
Goal b # b  =  Y # b  simplified to b  =  Y
Goal b  =  Y  proved by reduction.
Goal X # b{b} =  B{b} proved by reduction.
Goal | X # b |  =  | B |  proved by reduction.
Goal b{a} =  Y{a} proved by reduction.
Goal b{b} =  Y{b} proved by reduction.
Goal | b |  =  | Y |  proved by reduction.
Proof succeeded.
```

Note the use of the simplification rules during the proving process:

- $b + Xb = B + Y$ is simplified to $Xb = B$ and $b = Y$,
- $bb + Xbb = Bb + Yb$ is simplified to $Xbb = Bb$ and $bb = Yb$;
- $Xbb = Bb$ is simplified to $Xb = B$.

Without regular strategies, which allow to combine the circular coinduction steps with the simplification rule, the class of context-free processes for which CIRC is able to automatically prove the equivalence is much smaller. Regular strategies effectively enlarge the class of problems which can be automatically proved with CIRC. The problem of finding the complete subclass of context-free processes for which the equivalence can be automatically proved with CIRC remains open.

5 Conclusion

We presented CIRC, an automated prover supporting the principle of circularity and in particular circular coinduction. CIRC is implemented as an extension of Maude using its metalevel programming capabilities. Two novel contributions have been made in this paper. First, we showed the correctness of the circular coinductive proof strategy. Second, we showed how CIRC can be extended using regular strategies. The method is exemplified by adding a simplification rule and defining a proof tactic (as a regular strategy) used for proving the equivalence of context-free processes. The use of regular strategies as proof tactics enlarge the class of problems which can be automatically solved. CIRC implements also the circularity principle for proving properties by induction. Regular strategies make the use of proof tactics that combine coinduction with induction possible. This can be done now only in an assisted way.

References

[1] Baeten, J. C. M., J. A. Bergstra and J. W. Klop, *Decidability of bisimulation equivalence for process generating context-free languages*, Journal of the ACM **40** (1993), pp. 653 – 682.

[2] Clavel et al., M., *Maude: Specification and Programming in Rewriting Logic*, J. of TCS **285** (2002), pp. 187–243.

[3] Clavel, M., F. Durán, S. Eker and J. Meseguer, *Building Equational Proving Tools by Reflection in Rewriting Logic*, in: *Cafe: An Industrial-Strength Algebraic Formal Method*, Élsevier, 2000 .

[4] Goguen, J., K. Lin and G. Roşu, *Circular coinductive rewriting*, in: *Proceedings of Automated Software Engineering 2000* (2000), pp. 123–131.

[5] Goguen, J., K. Lin and G. Roşu, *Conditional Circular Coinductive Rewriting with Case Analysis*, in: *WADT'02*, LNCS **2755** (2003), pp. 216–232.

[6] Hausmann, D., T. Mossakowski and L. Schröder, *Iterative Circular Coinduction for CoCASL in Isabelle/HOL*, in: *Fundamental Approaches to Software Engineering 2005*, LNCS **3442** (2005), pp. 341–356.

[7] Hennicker, R., *Context induction: a proof principle for behavioral abstractions*, Formal Aspects of Computing **3** (1991), pp. 326–345.

[8] Hopcroft, J. and J.D.Ullman, "Introduction to automata theory, languages, and computation," Addison–Wesley, 1979.

[9] Hüttel, H. and C. Stirling, *Actions Speak Louder Than Words: Proving Bisimilarity for Context-Free Processes.*, J. Log. Comput. **8** (1998), pp. 485–509.

[10] Jacobs, B. and J. Rutten, *A tutorial on (co)algebras and (co)induction*, Bulletin of the European Association for Theoretical Computer Science **62** (1997), pp. 222–259.

[11] Lucanu, D. and G. Roşu, *The CIRC Prover*, `http://fsl.cs.uiuc.edu/index.php/Circ`.

[12] Lucanu, D. and G. Roşu, *Circ: A Circular Coinductive Prover*, in: *2nd Conference on Algebra and Coalgebra in Computer Science (CALCO 2007)*, Bergen, Norway, 2007, to appear in Lecture Notes in Computer Science.

[13] Marti-Oliet, N., J. Meseguer and A. Verdejo, *Towards a Strategy Language for Maude*, Electronic Notes of Theoretical Computer Science **117** (2005), pp. 417–441.

[14] Roşu, G., "Hidden Logic," Ph.D. thesis, University of California at San Diego (2000).

[15] Roşu, G., *Equality of Streams is a Π_2^0-Complete Problem*, in: *the 11th ACM SIGPLAN Int. Conf. on Functional Programming (ICFP'06)* (2006).

[16] Roşu, G. and M. Viswanathan, *Testing Extended Regular Language Membership Incrementally by Rewriting*, in: *RTA'03*, LNCS **2706** (2003.), pp. 499–514.

[17] Sen, K. and G. Rosu, *Generating Optimal Monitors for Extended Regular Expressions.*, Electr. Notes Theor. Comput. Sci. **89** (2003).

[18] Visser, E., Z. e. A. Benaissa and A. Tolmach, *Building program optimizers with rewriting strategies*, ACM SIGPLAN Notices **34** (1999), pp. 13–26.

Formalizing Constructions of Abstract Machines for Functional Languages in Coq

Małgorzata Biernacka[1]

LRI, Univ Paris-Sud, CNRS, Orsay F-91405
INRIA Futurs, ProVal, Parc Orsay Université, F-91893

Dariusz Biernacki[2]

INRIA Futurs, ProVal, Parc Orsay Université, F-91893
LRI, Univ Paris-Sud, CNRS, Orsay, F-91405

Abstract

This note reports on preliminary work on formalizing and mechanizing derivations of abstract machines from reduction semantics of functional languages in Coq, based on the method of refocusing due to Danvy and Nielsen. The ultimate goal is to develop a general framework for representing programming languages and their various semantic specifications inside Coq, and to automatize derivations between these specifications while ensuring their correctness.
In this note, we describe two case studies of derivations of abstract machines from reduction semantics (i.e., machines realizing the reduction strategies given by the corresponding reduction semantics): we consider the standard call-by-value lambda calculus, for which the derived machine is the substitution-based CK machine of Felleisen and Friedman, and the call-by-name variant of the calculus of closures which leads to the environment-based Krivine machine. These correspondences have been discovered and reported in previous work by the first author and Danvy in the setting of functional programs and their transformations, whereas here we adopt a relational approach and we formalize the derivation steps and prove them correct within Coq's type theory. We also discuss some technical issues of our formalization and a future construction of the general framework for refocusing in Coq.

Keywords: abstract machine, reduction semantics, refocusing, Coq

1 Introduction

Typically, a programming language is given several different semantic specifications, each designed for a particular purpose and each preferred by either the language designer, implementor or programmer. In order for all these different specifications to define the same behaviour, we need to provide proofs of their equivalence. With some practice, this task can be performed by hand and indeed, it usually is done—whenever the need arises—in an ad hoc manner.

[1] Email: mbiernac@lri.fr
[2] Email: dabi@lri.fr

We argue that it need not be so, and in many cases we could benefit from considering systematic and to some point automatic derivation methods that would allow one to arrive from one semantic description taken as standard, to another one that is more convenient in a given application, while the correctness of the outcome is guaranteed by the derivation method. This work not only saves one the tedious task of proving similar kinds of theorems whenever one introduces even a mild variation into a language, but it also may help connect the same computational features as they occur in different contexts.

There has been some work aiming at connecting various forms of operational semantics of functional languages. In particular, Ager et al. have discovered the functional correspondence between higher-order evaluators and abstract machines [1,2,3] when these semantic descriptions are implemented as ML programs: one can then interderive them by means of standard program transformation techniques (CPS transformation, defunctionalization, closure conversion). In another line of work, connections between reduction semantics and abstract machines have been studied, and it has been discovered how the refocusing method of Danvy and Nielsen provides a way to systematically derive the latter from the former [5,6,9]. The work of the first author and Danvy studies the correspondence not only for the standard λ-calculus but also for a wide class of its extensions with a non-standard notion of reduction, thus accommodating such non-local computational tools and effects as, e.g., control operators, assignment, state, or lazy evaluation [6]. In particular, the method has been applied to calculi of closures in order to materialize the slogan that calculi of explicit substitutions correspond to environment machines, and to context-sensitive reduction semantics that give rise to store-based machines. In addition, the systematic method shows how an abstract machine realizes a particular reduction strategy specified by the underlying reduction semantics. As with the functional correspondence, implementations of semantic descriptions in a functional programming language have been considered and the correctness of the derived machines relies on the correctness of standard program transformation techniques.

Since we argue for the generality and the uniformity of these correspondences, it is natural to try to automate them in an appropriate tool. In the present work, we report on an implementation of two case studies and we give some ideas on how a proof assistant can be used to aid in the tasks outlined above. Developing a general framework for expressing arbitrary languages and semantic descriptions is left as future work.

In this note, we look at the refocusing method from another point of view—we recast the development in the language of Coq [7] and we then give formal proofs of correctness of all refocusing steps, applied to two prototypical applicative programming languages: in order to show the plain version of the method, we apply it to the standard call-by-value λ-calculus leading to the CK abstract machine [11], and to illustrate one of its extensions, the second case study focuses on the call-by-name version of the extended Curien's calculus of closures [8] as introduced by the first author and Danvy [5]. We suggest a logical characterization of what it means for an abstract machine to realize a particular reduction strategy, and we identify the general properties of each of the intermediate derivation steps and we hope to be able to use them in a general framework. In this work, we consider only abstract

machines, i.e., transition systems operating on source code, as opposed to virtual machines, operating on compiled code. [3]

The previous articles on refocusing and the present one look at computation in the λ-calculus from the rewriting point of view. However, we must warn the reader that in this particular context, where the focus is on evaluation (as found in typical functional languages) rather than general normalization, it is often convenient to deviate from the canonical understanding of some term-rewriting concepts, in order to adapt them to this restricted area of application. Here, we provide definitions of the concepts used in a non-standard way.

2 Refocusing in reduction semantics

The starting point of refocusing is a specification of a reduction semantics. Let us briefly recall its ingredients.

2.1 Reduction semantics

Reduction semantics is a form of small-step operational semantics based on term rewriting. In our approach, we assume it is given by the syntactic categories of terms, values (non-reducible terms taken as correct results of evaluation), redexes (or, more precisely: potential redexes possibly including "smallest" stuck terms, if ill-formed terms are allowed in the syntax), evaluation contexts and two functions: `contract` – performing a basic computation step (i.e., rewriting a redex) and `plug` – giving meaning to the evaluation contexts by assigning a term to each pair of a term and a context. (If we look at contexts as "terms with a hole", then the `plug` function simply fills the hole with a given term.) Evaluation contexts together with the `plug` function determine the reduction strategy: the next redex to reduce in a given term is found by decomposing the term into a redex and the surrounding context. We say that a term t decomposes into another term t_0 and an evaluation context c, if $\texttt{plug}\,(t_0, c) = t$, where $=$ stands for syntactic equality. [4] We use the notation with an explicit call to the function `plug` rather than the usual notation $c[t_0]$ whenever we want to emphasize the fact that we mean the actual implementation of this operation (here, in Coq). We further say that a decomposition (t, c) is trivial if c is the empty context or if t is a value. Hence, it is convenient to define potential redexes as non-value terms whose all decompositions are trivial. We furthermore assume that a value can only be decomposed into itself and the empty context.

The (naive) process of evaluation then consists in repeatedly decomposing a given term into a redex and an evaluation context (if it is not already a value), contracting the redex (if it is not a stuck term) and plugging the contractum into the context. In general, evaluation may be nonterminating or stuck.

For the refocusing method to apply, we need to ensure the unique-decomposition property of the reduction semantics; in effect, we only allow sequential computation.

The unique-decomposition property can be formulated as follows:

[3] The distinction between abstract and virtual machines has been introduced by Ager et al. [1].

[4] We assume that no variable capture occurs when plugging, which indeed is the case for evaluation.

Definition 2.1 We say that the reduction semantics has the unique-decomposition property if for each term t, t is either a value or there exists a unique potential redex r and a unique context c such that $\texttt{plug}\,(r, c) = t$ (i.e., for all redexes r, r' and evaluation contexts c, c', if $\texttt{plug}\,(r, c) = t$ and $\texttt{plug}\,(r', c') = t$, then $r = r'$ and $c = c'$).

Providing these ingredients is sufficient to precisely define a reduction semantics for a language, even though we leave the question of how to decompose a term unspecified. Let us now consider this operation in more detail. Clearly, when we want to implement the naive evaluation procedure according to a given reduction semantics, we need to write a decomposing function which, for a given non-value term, will return a pair of a potential redex and a context. Furthermore, we assume that for a value, the decomposing function returns this value and the empty context. We say that a function `decompose` from terms to decompositions is correct with respect to a given reduction semantics if and only if it is the right inverse of the `plug` function, i.e., if $\texttt{plug} \circ \texttt{decompose} = \texttt{id}$. Now, the consequence of the unique decomposition is the fact that any correct decomposing function will return the same decomposition for a given term. Hence the implementation of the evaluation process remains correct regardless of the particular implementation of the decomposing function, as long as this function is correct in the above sense.

However, we note that an alternative characterization of the unique-decomposition property can be given, if we already have a decomposing function that satisfies the conditions of the following lemma:

Lemma 2.2 (Unique decomposition) *Assume we have total functions* `plug` *and* `decompose`. *If the decomposition function* `decompose` *is correct with respect to the function* `plug`*, i.e., if* $\texttt{plug} \circ \texttt{decompose} = \texttt{id}$*, and moreover, if* $\texttt{decompose}(\texttt{plug}\,(\texttt{r}, \texttt{c})) = (\texttt{r}, \texttt{c})$ *for any redex* r *and any context* c*, then the reduction semantics has the unique-decomposition property.*

The lemma states that it is enough to write a decomposition function and prove that it satisfies the two conditions in order to verify the unique-decomposition property of the reduction semantics, and in fact in our implementation this approach turned out to be more convenient, since for applying refocusing we need to write a decomposition function and prove these properties anyway. (The correctness condition is needed to ensure existence of a decomposition for all terms, and the second condition is necessary for proving uniqueness.) There are several natural ways of implementing the decomposing function: either naively by recursive descent over terms, or derived from the plug function, roughly by inverting its defining clauses.

2.2 Refocusing

Refocusing is a derivation method used to obtain abstract machines from specifications of reduction semantics. By an abstract machine we mean a transition system simulating evaluation of terms, with an explicit representation of control stacks, stores, etc., as found in real-life implementations, but still at a sufficiently abstract level to facilitate reasoning about execution without drowning in implementation details. The virtue of refocusing is to show how certain features of a higher-level

specification of a language become concrete in the corresponding abstract machine, and in which sense an abstract machine realizes a particular evaluation strategy for a given language.

Originally, refocusing has been presented as an algorithm for constructing an efficient evaluation function that avoids reconstructing intermediate terms in the decompose-contract-plug loop described in the previous section [10]. Only subsequently, it has been observed that, in effect, refocusing yields abstract machines. Moreover, refocusing has been extended to account for calculi of explicit substitutions that yield environment-based machines, and to context-sensitive reduction semantics, leading to abstract machines with global store [5]. In the latter work, the method has been implemented, presented and studied in the setting of functional programming languages, and the correctness of the derivation steps have been established as a consequence of the correctness of the applied program transformations.

In this work, we shift to a more abstract view: we represent all the intermediate steps (i.e., various functions computing the value of a term according to the same strategy; all extensionally equivalent, but becoming closer to a transition system) as relations (inductive predicates in Coq), and we provide uniform induction principles for proving correctness of the derivation steps that can be made into Coq tactics. Let us now discuss some implementation features.

2.3 The Coq proof assistant

Almost all the ingredients of a reduction semantics as described in Section 2.1 can be represented in Coq in an intuitive way. The only problem arises when representing evaluation—it cannot be written as a function, since it is possibly nonterminating. Therefore, we choose to represent all the functions (except for `plug` and `contract`) as relations, and in fact, this representation is quite natural and it gives rise to simple and uniform correctness proofs of intermediate steps: the Coq-generated induction principles are used to prove the simulations by induction on derivations (which corresponds to proofs by cases in the functional specification of these relations).

Below we present the main features of our implementation, common to both languages that we consider, and therefore suggesting a solution to be applied in a general framework. [5]

- We represent terms, values and redexes as different sets (datatypes) and we provide injection functions for values and redexes into terms, together with their injectivity proofs (the functions `value_to_term` and `redex_to_term` provide coercions from the set of values and from the set of redexes, respectively, to the set of terms; as a consequence, we can simplify the presentation and use values or redexes where objects of type terms are expected).

[5] In the remainder of the article, we include fragments of the Coq development, but with the syntax transformed to a more readable form and accessible for readers not familiar with Coq. The complete Coq development can be found at `http://www.lri.fr/~mbiernac/coq/refocus/`.

Parameters `term value redex` : **Set**.
Parameter `value_to_term` : `value` → `term`.
Parameter `redex_to_term` : `redex` → `term`.
Axiom `value_to_term_injective` :
$\forall\, v, v'.\, \texttt{value\_to\_term}\ \texttt{v} = \texttt{value\_to\_term}\ \texttt{v}' \to \texttt{v} = \texttt{v}'$.
Axiom `redex_to_term_injective` :
$\forall\, r, r'.\, \texttt{redex\_to\_term}\ \texttt{r} = \texttt{redex\_to\_term}\ \texttt{r}' \to \texttt{r} = \texttt{r}'$.

- The reduction strategy is given by the grammar of reduction contexts and the plug function:

  Parameter `context` : **Set**.
  Parameter `plug` : `term` → `context` → `term`.

- The notion of a potential redex allows us to prove the totality of decomposition, i.e., for every non-value term there exists a decomposition of that term into a potential redex and a context:

  Axiom `decomposition` :
  $\forall\, t : \texttt{term}.\, (\exists\, v : \texttt{value}.\, t = v) \vee (\exists\, r : \texttt{redex}, c : \texttt{context}.\, t = \texttt{plug}\,(r,\, c))$.

- Now we need to provide a total decomposition function. The particular definition, obtained by inverting the plugging function is naturally defined by two mutually recursive functions: one descending over the structure of the currently decomposed term, and an auxiliary function descending over the surrounding evaluation context when a value is encountered. This algorithm cannot be easily represented in Coq as a function, and so we choose to represent it as a relation. (It is possible to define this function in Coq, using the `Function` command with an appropriate measure function that takes into account both the term and the surrounding evaluation context. Such solution, however, does not seem to be worth the effort, since in effect it is defined using a well-founded relation that coincides with our relation, and the computational overhead is significant.)

  Parameter `decomp` : **Set**.
  Parameter `dec` : `term` → `context` → `decomp` → **Prop**.
  Parameter `decctx` : `context` → `value` → `decomp` → **Prop**.

- To complete the definition of reduction semantics, we need to ensure the unique-decomposition property. As mentioned in Section 2.1, we choose to do it by proving the following property of the function `dec`, which implies the condition required in Lemma 2.2:

  Axiom `dec_plug` : $\forall\, c, c'$: `context`, t : `term`, d : `decomp`.
  $\texttt{dec}\,(\texttt{plug}\,(t,\, c))\, c'\, d \leftrightarrow \texttt{dec}\, t\, (c \circ c')\, d$.
  where $c \circ c'$ denotes the standard concatenation of contexts (implemented by the function `compose`).

- All the intermediate functions are represented as relations and we prove that each step is correct in the sense that the evaluation relations defined on the basis of each of them are all equivalent. Proofs of correctness are done by induction on derivations using the induction principles generated by the `Scheme` command (mutual induction in cases where there are mutually recursive relations).

3 From a reduction semantics to an abstract machine for the λ-calculus

In this section we detail the implementations of the derivation of abstract machines for two languages: the standard call-by-value λ-calculus, leading to the substitution-based CK machine, and the call-by-name extended calculus of closures, leading to the environment-based Krivine machine.

3.1 The call-by-value lambda-calculus

We consider the call-by-value λ-calculus with names drawn from a countable set `var_name`:[6]

$$\frac{x : \texttt{var\_name}}{\texttt{var}\,x : \texttt{term}} \qquad \frac{x : \texttt{var\_name} \quad t : \texttt{term}}{\texttt{lam}\,(x,\, t) : \texttt{term}} \qquad \frac{t_0 : \texttt{term} \quad t_1 : \texttt{term}}{\texttt{app}\,(t_0,\, t_1) : \texttt{term}}$$

3.1.1 A reduction semantics

A value is either a variable or a λ-abstraction, and a potential redex is an application of a value to a value (i.e., by the definition of Section 2.1, a non-value term with only trivial decompositions):

$$\frac{x : \texttt{var\_name}}{\texttt{v\_var}\,x : \texttt{value}} \qquad \frac{x : \texttt{var\_name} \quad t : \texttt{term}}{\texttt{v\_lam}\,(x,\, t) : \texttt{value}} \qquad \frac{v_0 : \texttt{value} \quad v_1 : \texttt{value}}{\texttt{beta}\,(v_0,\, v_1) : \texttt{redex}}$$

In the presence of ill-formed (stuck) terms, `contract` is a partial function, undefined for an application of a variable to a value:

$$\begin{aligned}\texttt{contract}\,(\texttt{beta}\,(v_0,\, v_1)) = \ & \textbf{match}\, v_0\, \textbf{with}\\ & \mid \texttt{v\_var}\,x \Rightarrow \texttt{None}\\ & \mid \texttt{v\_lam}\,(x,\, t) \Rightarrow \texttt{Some}\,(\texttt{subst}\,(t,\, x,\, v_1))\\ & \textbf{end}\end{aligned}$$

where $\texttt{subst} : \texttt{term} \to \texttt{var\_name} \to \texttt{term} \to \texttt{term}$ is a function performing the standard capture-avoiding substitution.

The following two definitions encode the call-by-value, left-to-right reduction strategy: first we give the structure of evaluation contexts, and below we give them a meaning by defining the function `plug`:

$$\frac{}{\texttt{mt} : \texttt{context}} \qquad \frac{v : \texttt{value} \quad c : \texttt{context}}{\texttt{ap\_l}\,(v,\, t) : \texttt{context}} \qquad \frac{t : \texttt{term} \quad c : \texttt{context}}{\texttt{ap\_r}\,(t,\, c) : \texttt{context}}$$

$$\begin{aligned}\texttt{plug}\,(t,\, c) = \ & \textbf{match}\, c\, \textbf{with}\\ & \mid \texttt{mt} \Rightarrow t\\ & \mid \texttt{ap\_l}\,(v,\, c') \Rightarrow \texttt{plug}\,(\texttt{app}\,(v,\, t),\, c')\\ & \mid \texttt{ap\_r}\,(t',\, c') \Rightarrow \texttt{plug}\,(\texttt{app}\,(t,\, t'),\, c')\\ & \textbf{end}\end{aligned}$$

Evaluation contexts are represented "inside out" (this interpretation is defined by the function `plug`) which makes them suitable for performing evaluation by an

[6] In the sequel, we use inference rules to represent inductive relations in Coq as well as inductive datatypes (sets)—the latter choice is made in order to emphasize the role of types. To avoid confusion we use the same constructor names as in the Coq development.

abstract machine, where contexts become stacks: pushing a term on the stack corresponds to the construction of the context, and popping a term from the stack corresponds to the destruction of the context.

A decomposition is either a pair of a redex and a context, or a value in the empty context, representing a single value:

$$\frac{v : \mathtt{value}}{\mathtt{d\_val}\, v : \mathtt{decomposition}} \qquad \frac{r : \mathtt{redex} \quad c : \mathtt{context}}{\mathtt{d\_red}\,(r,\, c) : \mathtt{decomposition}}$$

Evaluation is then implemented as a relation representing the decompose-contract-plug loop, where the decomposing function `dec` is defined as a relation as follows:

$$\frac{\mathtt{decctx}\,(c,\, v,\, d)}{\mathtt{dec}\,(v,\, c,\, d)}\ (\mathtt{val\_ctx}) \qquad \frac{\mathtt{dec}\,(t_0,\, \mathtt{ap\_r}\,(t_1,\, c),\, d)}{\mathtt{dec}\,(\mathtt{app}\,(t_0,\, t_1),\, c,\, d)}\ (\mathtt{app\_ctx})$$

$$\frac{}{\mathtt{decctx}\,(\mathtt{mt},\, v,\, \mathtt{d\_val}\, v)}\ (\mathtt{mt\_dec}) \qquad \frac{\mathtt{dec}\,(t,\, \mathtt{ap\_l}\,(v,\, c),\, d)}{\mathtt{decctx}\,(\mathtt{ap\_r}\,(t,\, c),\, v,\, d)}\ (\mathtt{ap\_r\_dec})$$

$$\frac{}{\mathtt{decctx}\,(\mathtt{ap\_l}\,(v_0,\, c),\, v_1,\, \mathtt{d\_red}\,(\mathtt{beta}\,(v_0,\, v_1),\, c))}\ (\mathtt{ap\_l\_dec})$$

$$\frac{\mathtt{dec}\,(t,\, \mathtt{mt},\, d)}{\mathtt{decmt}\,(t,\, d)}\ (\mathtt{d\_intro}) \qquad \frac{}{\mathtt{iter}\,(\mathtt{d\_val}\, v,\, v)}\ (\mathtt{i\_val})$$

$$\frac{\mathtt{contract}\,(r) = \mathtt{Some}\, t \quad \mathtt{decmt}\,(\mathtt{plug}\,(t,\, c),\, d) \quad \mathtt{iter}\,(d,\, v)}{\mathtt{iter}\,(\mathtt{d\_red}\,(r,\, c),\, v)}\ (\mathtt{i\_red})$$

$$\frac{\mathtt{decmt}\,(t,\, d) \quad \mathtt{iter}\,(d,\, v)}{\mathtt{eval}\,(t,\, v)}\ (\mathtt{e\_intro})$$

It is straightforward to prove that `decmt`, `iter` and `eval` all define functions. `decmt` is a total function, returning a decomposition for all terms, but `iter` and `eval` are not total, since they are not defined for stuck terms. (If needed, they could be made total by introducing an additional constructor handling stuck terms in the relation `iter`.)

3.1.2 A pre-abstract machine

If the decomposing function satisfies the following property:

$$\forall\, t,\, c,\, d.\, \mathtt{decmt}\,(\mathtt{plug}\,(t,\, c),\, d) \leftrightarrow \mathtt{dec}\,(t,\, c,\, d)$$

then in the relations of Section 3.1.1 we can replace the calls to `decmt` by calls to `dec`, and we obtain the following definitions of $\mathtt{iter}_0$ and $\mathtt{eval}_0$:

$$\frac{}{\mathtt{iter}_0\,(\mathtt{d\_val}\, v,\, v)}\ (\mathtt{i\_val}_0) \qquad \frac{\mathtt{contract}\,(r) = \mathtt{Some}\, t \quad \mathtt{dec}\,(t,\, c,\, d) \quad \mathtt{iter}_0\,(d,\, v)}{\mathtt{iter}_0\,(\mathtt{d\_red}\,(r,\, c),\, v)}\ (\mathtt{i\_red}_0)$$

$$\frac{\mathtt{dec}\,(t,\, \mathtt{mt},\, d) \quad \mathtt{iter}_0\,(d,\, v)}{\mathtt{eval}_0\,(t,\, v)}\ (\mathtt{e\_intro}_0)$$

The two relations `iter` and $\mathtt{iter}_0$ are equivalent, and we prove it by induction on derivations.

The resulting definition is called a pre-abstract machine because it already optimizes the decompose-contract-plug loop by avoiding the reconstruction of intermediate terms, and directly proceeding to decomposing the contractum in the given evaluation context.

3.1.3 A staged abstract machine

Next, we observe that whenever we reach a decomposition, it will immediately be consumed by $\mathtt{iter}_0$, therefore we can distribute the calls to $\mathtt{iter}_0$ in the definition of `decctx`. This way, the new relations $\mathtt{dec}_1$ and $\mathtt{decctx}_1$ implement compatibility rules of the semantics (traversing terms and contexts in search of a redex) and $\mathtt{iter}_1$ implements contraction and immediately performs decomposition on the contractum:

$$\frac{\mathtt{decctx}_1\,(c,\,v,\,v')}{\mathtt{dec}_1\,(v,\,c,\,v')}\,(\mathtt{val\_ctx}_1) \qquad \frac{\mathtt{dec}_1\,(t_0,\,\mathtt{ap\_r}\,(t_1,\,c),\,v)}{\mathtt{dec}_1\,(\mathtt{app}\,(t_0,\,t_1),\,c,\,v)}\,(\mathtt{app\_ctx}_1)$$

$$\frac{\mathtt{iter}_1\,(\mathtt{d\_val}\,v,\,v')}{\mathtt{decctx}_1\,(\mathtt{mt},\,v,\,v')}\,(\mathtt{mt\_dec}_1) \qquad \frac{\mathtt{dec}_1\,(t,\,\mathtt{ap\_l}\,(v,\,c),\,v')}{\mathtt{decctx}_1\,(\mathtt{ap\_r}\,(t,\,c),\,v,\,v')}\,(\mathtt{ap\_r\_dec}_1)$$

$$\frac{\mathtt{iter}_1\,(\mathtt{d\_red}\,(\mathtt{beta}\,(v_0,\,v_1),\,c),\,v)}{\mathtt{decctx}_1\,(\mathtt{ap\_l}\,(v_0,\,c),\,v_1,\,v)}\,(\mathtt{ap\_l\_dec}_1)$$

$$\frac{}{\mathtt{iter}_1\,(\mathtt{d\_val}\,v,\,v)}\,(\mathtt{i\_val}_1) \qquad \frac{\mathtt{contract}\,(r) = \mathtt{Some}\,t \quad \mathtt{dec}_1\,(t,\,c,\,v)}{\mathtt{iter}_1\,(\mathtt{d\_red}\,(r,\,c),\,v)}\,(\mathtt{i\_red}_1)$$

$$\frac{\mathtt{dec}_1\,(\mathtt{mt},\,t,\,v)}{\mathtt{eval}_1\,(t,\,v)}\,(\mathtt{e\_intro}_1)$$

Theorem 3.1 $\forall\, t,\, v.\, \mathtt{eval}_0\,(t,\,v) \leftrightarrow \mathtt{eval}_1\,(t,\,v)$.

The proof of equivalence is done by induction on derivations and relies on the totality of `dec` and on the following auxiliary properties, capturing the nature of the transition from the pre-abstract machine to the staged abstract machine:

(i) $\forall\, t,\, c,\, d.\, \mathtt{dec}\,(t,\,c,\,d) \Rightarrow \forall\, v.\, \mathtt{iter}_1\,(d,\,v) \Rightarrow \mathtt{dec}_1\,(t,\,c,\,v)$

(ii) $\forall\, c,\, v,\, d.\, \mathtt{decctx}_0\,(c,\,v,\,d) \Rightarrow \forall\, v'.\, \mathtt{iter}_1\,(d,\,v') \Rightarrow \mathtt{decctx}_1\,(c,\,v,\,v')$

(iii) $\forall\, t,\, c,\, v.\, \mathtt{dec}_1\,(t,\,c,\,v) \Rightarrow \forall\, d.\, \mathtt{dec}_0\,(t,\,c,\,d) \Rightarrow \mathtt{iter}_0\,(d,\,v)$

(iv) $\forall\, c,\, v,\, v'.\, \mathtt{decctx}_1\,(c,\,v,\,v') \Rightarrow \forall\, d.\, \mathtt{decctx}_0\,(c,\,v,\,d) \Rightarrow \mathtt{iter}_0\,(d,\,v')$

3.1.4 An eval/apply abstract machine

The next step consists in inlining the iterating function and obtaining a proper abstract machine, i.e., a transition system without a trampoline function triggering the loop. This stage makes us "forget" the sites where we found redexes and performed reductions, and that is why it is not always trivial to unwind abstract machines of this form to read the underlying reduction semantics [4,11,12]. The eval/apply abstract machine consists of two kinds of transitions: one that analyzes terms and one that analyzes reduction contexts:

$$\frac{\mathtt{decctx_2}\,(c,\,v,\,v')}{\mathtt{dec_2}\,(v,\,c,\,v')}\,(\mathtt{val\_ctx_2}) \qquad \frac{\mathtt{dec_2}\,(t_0,\,\mathtt{ap\_r}\,(t_1,\,c),\,v)}{\mathtt{dec_2}\,(\mathtt{app}\,(t_0,\,t_1),\,c,\,v)}\,(\mathtt{app\_ctx_2})$$

$$\frac{}{\mathtt{decctx_2}\,(\mathtt{mt},\,v,\,v)}\,(\mathtt{mt\_dec_2}) \qquad \frac{\mathtt{dec_2}\,(t,\,\mathtt{ap\_l}\,(v,\,c),\,v')}{\mathtt{decctx_2}\,(\mathtt{ap\_r}\,(t,\,c),\,v,\,v')}\,(\mathtt{ap\_r\_dec_2})$$

$$\frac{\mathtt{contract}\,(\mathtt{beta}\,(v_0,\,v_1)) = \mathtt{Some}\,t \quad \mathtt{dec_2}\,(t,\,c,\,v)}{\mathtt{decctx_2}\,(\mathtt{ap\_l}\,(v_0,\,c),\,v_1,\,v)}\,(\mathtt{ap\_l\_dec_2})$$

$$\frac{\mathtt{dec_2}\,(t,\,\mathtt{mt},\,v)}{\mathtt{eval_2}\,(t,\,v)}\,(\mathtt{e\_intro_2})$$

Again, the equivalence of the two evaluation relations is proved in each direction by induction on derivations of the corresponding decomposing function. Moreover, for the left-to-right direction we need the auxiliary property:

$$\begin{aligned}\forall\, d,\, v.\, \mathtt{iter_1}\,(d,\,v) \rightarrow\ & \mathbf{match}\, d\, \mathbf{with}\\ & \mid \mathtt{d\_val}\, v' \Rightarrow \mathtt{decctx_2}\,(\mathtt{mt},\,v',\,v)\\ & \mid \mathtt{d\_red}\,(r,\,c) \Rightarrow \mathtt{dec_2}\,(r,\,c,\,v)\\ & \mathbf{end}\end{aligned}$$

3.1.5 The CK abstract machine

The definition from Section 3.1.4 is a relational presentation of an abstract machine. Another, more traditional presentation is shown below: we arrive at it simply transposing each inference rule described in $\mathtt{dec_2}$ and $\mathtt{decctx_2}$ into a new relation between the premise and the conclusion of each rule, giving rise to two kinds of configurations corresponding to these relations. We also add the initial and final configurations (and their corresponding transitions), corresponding to loading the machine with a term to evaluate, and recovering the value, respectively. (These transitions are read off the $\mathtt{eval_2}$ relation.)

$$\frac{t : \mathtt{term}}{\mathtt{c\_init}\,t : \mathtt{configuration}} \qquad \frac{t : \mathtt{term} \quad c : \mathtt{context}}{\mathtt{c\_eval}\,(t,\,c) : \mathtt{configuration}}$$

$$\frac{c : \mathtt{context} \quad v : \mathtt{value}}{\mathtt{c\_apply}\,(c,\,v) : \mathtt{configuration}} \qquad \frac{v : \mathtt{value}}{\mathtt{c\_final}\,v : \mathtt{configuration}}$$

$$\begin{aligned}\mathtt{c\_init}\,t &\rhd \mathtt{c\_eval}\,(t,\,\mathtt{mt})\\ \mathtt{c\_eval}\,(v,\,c) &\rhd \mathtt{c\_apply}\,(c,\,v)\\ \mathtt{c\_eval}\,(\mathtt{app}\,(t_0,\,t_1),\,c) &\rhd \mathtt{c\_eval}\,(t_0,\,\mathtt{ap\_r}\,(t_1,\,c))\\ \mathtt{c\_apply}\,(\mathtt{mt},\,v) &\rhd \mathtt{c\_final}\,v\\ \mathtt{c\_apply}\,(\mathtt{ap\_l}\,(v_0,\,c),\,v_1) &\rhd \mathtt{c\_eval}\,(t,\,c) \ \text{ if } \mathtt{contract}\,(\mathtt{beta}\,(v_0,\,v_1)) = \mathtt{Some}\,t\\ \mathtt{c\_apply}\,(\mathtt{ap\_r}\,(t,\,c),\,v) &\rhd \mathtt{c\_eval}\,(t,\,\mathtt{ap\_l}\,(v,\,c))\end{aligned}$$

$$\frac{\mathtt{c\_init}\,t \rhd^{+} \mathtt{c\_final}\,v}{\mathtt{eval_3}\,(t,\,v)}\,(\mathtt{e\_intro_3})$$

The transitive closure of transitions $\rhd^{+}$ is defined in a standard way and is represented by an inductive predicate $\mathtt{transitive\_closure} : \mathtt{configuration} \rightarrow \mathtt{configuration} \rightarrow \mathtt{Prop}$.

The equivalences are proved as before with an auxiliary property:

$$\begin{array}{l}\forall\, C_0, C_1.\, C_0 \rhd^+ C_1 \rightarrow \textbf{match}\, C_0, C_1\, \textbf{with}\\ \quad \mid \mathtt{c\_eval}\,(t,\, c), \mathtt{c\_final}\, v \;\Rightarrow\; \mathtt{dec}_2\,(t,\, c,\, v)\\ \quad \mid \mathtt{c\_apply}\,(c,\, v'), \mathtt{c\_final}\, v \;\Rightarrow\; \mathtt{decctx}_2\,(c,\, v',\, v)\\ \quad \mid \_, \_ \;\Rightarrow\; \textbf{True}\\ \quad \textbf{end}\end{array}$$

The resulting abstract machine coincides with Felleisen and Friedman's canonical substitution-based abstract machine for evaluating λ-terms under call-by-value, the CK machine [11].

3.2 The call-by-name λ-calculus with explicit substitutions

In this section we discuss the call-by-name λ-calculus with de Bruijn indices and with closures (a weak version of explicit substitutions). Due to lack of space, we only outline the most important differences with respect to the previous section.

3.2.1 A reduction semantics

Let us start with the specification of the language and its reduction semantics. The terms are defined similarly as for the previous case (see p. 90), but with de Bruijn indices to represent variables, and we introduce a new syntactic category of closures, representing terms with delayed substitutions:

$$\frac{t : \mathtt{term} \quad s : \mathtt{list\ closure}}{\mathtt{pair}\,(t,\, s) : \mathtt{closure}} \qquad \frac{cl_0, cl_1 : \mathtt{closure}}{\mathtt{comp}\,(cl_0,\, cl_1) : \mathtt{closure}}$$

A value is a closure representing a λ-abstraction paired with an arbitrary delayed substitution (the type of `substitution` is defined as a list of closures) and there are 3 forms of a redex:

$$\frac{t : \mathtt{term} \quad s : \mathtt{substitution}}{\mathtt{v\_val}\,(t,\, s) : \mathtt{value}} \qquad \frac{v : \mathtt{value} \quad cl : \mathtt{closure}}{\mathtt{r\_beta}\,(v,\, s) : \mathtt{redex}}$$

$$\frac{i : \mathtt{nat} \quad s : \mathtt{substitution}}{\mathtt{r\_get}\,(i,\, s) : \mathtt{redex}} \qquad \frac{t_0, t_1 : \mathtt{term} \quad s : \mathtt{substitution}}{\mathtt{r\_app}\,(t_0,\, t_1,\, s) : \mathtt{redex}}$$

All reduction takes place at the level of closures, and not at the level of terms:

$$\begin{array}{l}\mathtt{contract}\, r = \textbf{match}\, r\, \textbf{with}\\ \quad \mid \mathtt{r\_beta}\,(\mathtt{v\_val}\,(t,\, s),\, cl) \;\Rightarrow\; \mathtt{Some}\,(\mathtt{pair}\,(t,\, cl :: s))\\ \quad \mid \mathtt{r\_get}\,(i,\, s) \;\Rightarrow\; \mathtt{nth\_option}\,(s,\, i)\\ \quad \mid \mathtt{r\_app}\,(t,\, s,\, cl) \;\Rightarrow\; \mathtt{Some}\,(\mathtt{comp}\,(\mathtt{pair}\,(t,\, cl),\, \mathtt{pair}\,(s,\, cl)))\\ \quad \textbf{end}\end{array}.$$

We define the left-to-right call-by-name reduction strategy as follows, by giving the grammar of contexts and the function `plug`:

$$\frac{}{\mathtt{mt} : \mathtt{context}} \qquad \frac{cl : \mathtt{closure} \quad c : \mathtt{context}}{\mathtt{ap\_r}\,(cl,\, c) : \mathtt{context}}$$

$$\begin{array}{l}\mathtt{plug}\,(cl,\,c) = \textbf{match}\,c\,\textbf{with}\\ \quad |\,\mathtt{mt} \Rightarrow cl\\ \quad |\,\mathtt{ap\_r}\,(cl',\,c') \Rightarrow \mathtt{plug}\,(\mathtt{comp}\,(cl,\,cl'),\,c')\\ \quad \textbf{end}\end{array}$$

The decomposition relation is defined as follows:

$$\frac{\mathtt{decctx}\,(c,\,v,\,d)}{\mathtt{dec}\,(v,\,c,\,d)}\,(\mathtt{val\_ctx}) \qquad \frac{\mathtt{dec}\,(cl_0,\,\mathtt{ap\_r}\,(cl_1,\,c),\,d)}{\mathtt{dec}\,(\mathtt{comp}\,(cl_0,\,cl_1),\,c,\,d)}\,(\mathtt{comp\_ctx})$$

$$\frac{}{\mathtt{dec}\,(\mathtt{pair}\,(n,\,s),\,c,\,\mathtt{d\_red}\,(\mathtt{r\_get}\,(n,\,s),\,c))}\,(\mathtt{var\_ctx})$$

$$\frac{}{\mathtt{dec}\,(\mathtt{pair}\,(\mathtt{app}\,(t_0,\,t_1),\,s),\,c,\,\mathtt{d\_red}\,(\mathtt{r\_app}\,(t_0,\,t_1,\,s),\,c))}\,(\mathtt{app\_ctx})$$

$$\frac{}{\mathtt{decctx}\,(\mathtt{mt},\,v,\,\mathtt{d\_val}\,v)}\,(\mathtt{mt\_dec})$$

$$\frac{}{\mathtt{decctx}\,(\mathtt{ap\_r}\,(cl,\,c),\,v_1,\,\mathtt{d\_red}\,(\mathtt{r\_beta}\,(v,\,cl),\,c))}\,(\mathtt{ap\_r\_dec})$$

$$\frac{\mathtt{dec}\,((t,\,\mathtt{mt}),\,\mathtt{mt},\,d)}{\mathtt{decmt}\,(t,\,d)}\,(\mathtt{d\_intro})$$

Next, we define the decompose-contract-plug loop just as in the case of the call-by-value λ-calculus described in Section 3.1 except that the role of terms is now played by closures.

All the derivation steps up to the eval/apply machine are performed analogously to the development in Section 3.1.

3.2.2 A push/enter abstract machine

However, in the case of the calculus of closures our goal is to show how we can arrive at the Krivine machine, the canonical abstract machine for the call-by-name evaluation in the λ-calculus that uses environments. First of all, the Krivine machine is a push/enter machine which means that it has only one kind of configurations. We obtain a push/enter machine from an eval/apply machine by inlining the definition of $\mathtt{decctx}_2$ in the definition of $\mathtt{dec}_2$, and thus obtaining the following specification:

$$\frac{}{\mathtt{dec}_3\,(v,\,\mathtt{mt},\,\mathtt{d\_val}\,v)}\,(\mathtt{val\_ctx\_mt}_3) \qquad \frac{\mathtt{dec}_3\,(cl_0,\,\mathtt{ap\_r}\,(cl_1,\,c),\,d)}{\mathtt{dec}_3\,(\mathtt{comp}\,(cl_0,\,cl_1),\,c,\,d)}\,(\mathtt{comp\_ctx}_3)$$

$$\frac{\mathtt{dec}_3\,(\mathtt{pair}\,(t,\,cl :: s),\,c,\,v)}{\mathtt{dec}_3\,(\mathtt{pair}\,(\mathtt{lam}\,t,\,s),\,\mathtt{ap\_r}\,(cl,\,c),\,v)}\,(\mathtt{val\_ctx\_ap}_3)$$

$$\frac{\mathtt{contract}\,((\mathtt{r\_get}\,(n,\,s))) = \mathtt{Some}\,cl \quad \mathtt{dec}_3\,(cl,\,c,\,v)}{\mathtt{dec}_3\,(\mathtt{pair}\,(n,\,s),\,c,\,v)}\,(\mathtt{var\_ctx}_3)$$

$$\frac{\mathtt{dec}_3\,(\mathtt{comp}\,(\mathtt{pair}\,(t_0,\,s),\,\mathtt{pair}\,(t_1,\,s)),\,c,\,v)}{\mathtt{dec}_3\,(\mathtt{pair}\,(\mathtt{app}\,(t_0,\,t_1),\,s),\,c,\,v)}\,(\mathtt{app\_ctx}_3)$$

$$\frac{\mathtt{dec}_3\,(\mathtt{pair}\,(t,\,nil),\,\mathtt{mt},\,v)}{\mathtt{eval}_3\,(t,\,v)}\,(\mathtt{e\_intro}_3)$$

The correctness of this step is stated in the expected way and proved by induction on the derivation $\mathtt{dec}_2$ with the auxiliary property:

$$\begin{array}{l}\forall\, c, v, v'.\, \mathtt{decctx}_2\,(c,\, v,\, v') \to \textbf{match}\, c\, \textbf{with}\\ \qquad\qquad\qquad\qquad\quad \mid \mathtt{mt} \Rightarrow \mathtt{dec}_3\,(v,\, \mathtt{mt},\, v')\\ \qquad\qquad\qquad\qquad\quad \mid \mathtt{ap\_r}\,(cl,\, c) \Rightarrow \mathtt{dec}_3\,(\mathtt{r\_beta}\,(v,\, s),\, c,\, v')\\ \qquad\qquad\qquad\qquad\quad \textbf{end}\end{array}$$

3.2.3 An environment abstract machine

The final two steps are performed to get from a machine operating on closures to a machine operating on terms and substitutions (which become environments). To this end, we first observe that if we start evaluating a closure which is a pair of a term and a substitution, then we can bypass the step where the `comp` constructor is used, i.e., whenever $\mathtt{app\_ctx}_3$ is used in the derivation, then the next step will necessary be the application of $\mathtt{comp\_ctx}_3$. Therefore, we can merge the two steps into one, and in this way we obtain a relation that only operates on closures formed with the `pair` constructor. Moreover, all substitutions and contexts occurring in this new relation also contain only the restricted form of closures. This sublanguage coincides with Curien's calculus of closures [8]. We represent it using the datatypes `closureC`, `substitutionC` and `contextC`, together with their injection functions into the unrestricted language (treated as coercions whenever possible). The final relation, obtained by splitting components of the restricted closure is the following:

$$\frac{}{\mathtt{dec}_5\,(\mathtt{lam}\, t,\, s,\, \mathtt{mtC},\, \mathtt{v\_val}\,(t,\, s))}\ (\mathtt{val\_ctx\_mt}_5)$$

$$\frac{\mathtt{dec}_5\,(t,\, cl :: s,\, c,\, v)}{\mathtt{dec}_5\,(\mathtt{lam}\, t,\, s,\, \mathtt{apC\_r}\,(cl,\, c),\, v)}\ (\mathtt{val\_ctx\_ap}_5)$$

$$\frac{\mathtt{contract}\,((\mathtt{r\_get}\,(n,\, s))) = \mathtt{Some}\,(\mathtt{pairC}\,(cl,\, s'))\quad \mathtt{dec}_5\,(cl,\, s',\, c,\, v)}{\mathtt{dec}_5\,(n,\, s,\, c,\, v)}\ (\mathtt{var\_ctx}_5)$$

$$\frac{\mathtt{dec}_5\,(t_0,\, s,\, \mathtt{apC\_r}\,(\mathtt{pairC}\,(t_1,\, s),\, c),\, v)}{\mathtt{dec}_5\,(\mathtt{app}\,(t_0,\, t_1),\, s,\, c,\, v)}\ (\mathtt{app\_ctx}_5)$$

$$\frac{\mathtt{dec}_5\,(t,\, nil,\, \mathtt{mtC},\, v)}{\mathtt{eval}_5\,(t,\, v)}\ (\mathtt{e\_intro}_5)$$

The equivalence between the two evaluation relations holds for restricted closures only.

3.2.4 The Krivine abstract machine

Transforming the relation from Section 3.2.3 into the usual transition systems yields the following result, the canonical Krivine machine:

$$\frac{t : \mathtt{term}}{\mathtt{c\_init}\, t : \mathtt{configuration}}\qquad \frac{t : \mathtt{term}\quad s : \mathtt{substitutionC}\quad c : \mathtt{contextC}}{\mathtt{c\_eval}\,(t,\, s,\, c) : \mathtt{configuration}}$$

$$\frac{v : \mathtt{value}}{\mathtt{c\_final}\, v : \mathtt{configuration}}$$

$$\begin{aligned}
\texttt{c_init}\,t &\rhd \texttt{c_eval}\,(t,\,\texttt{nil},\,\texttt{mtC})\\
\texttt{c_eval}\,(\texttt{lam}\,t,\,s,\,\texttt{mtC}) &\rhd \texttt{c_final}\,(\texttt{v_val}\,(t,\,s))\\
\texttt{c_eval}\,(\texttt{lam}\,t,\,s,\,\texttt{apC_r}\,(cl,\,c)) &\rhd \texttt{c_eval}\,(t,\,cl :: s,\,c)\\
\texttt{c_eval}\,(\texttt{var}\,i,\,s,\,c) &\rhd \texttt{c_eval}\,(t,\,s',\,c)\\
&\qquad \text{if } \texttt{nth_option}\,(s,\,i) = \texttt{Some}\,(\texttt{pairC}\,(t,\,s'))\\
\texttt{c_eval}\,(\texttt{app}\,(t_0,\,t_1),\,s,\,c) &\rhd \texttt{c_eval}\,(t_0,\,s,\,\texttt{apC_r}\,(\texttt{pairC}\,(t_1,\,s),\,c))
\end{aligned}$$

$$\frac{\texttt{c\_init}\,t \rhd^{+} \texttt{c\_final}\,v}{\texttt{eval}_6\,(t,\,v)}\;(\texttt{e\_intro}_6)$$

4 Towards a generic framework

The refocusing method allows one to split the process of writing an abstract machine into smaller, simpler stages and intuitive proofs of their equivalence, as outlined in previous sections. These intermediate proofs can be automated: we can write Coq tactics for proving the equivalence of the pre-abstract machine with the staged abstract machine, etc., as long as these machines are specified in the way described in Section 3.1. Furthermore, if the user can provide a decomposition function together with the proof of the Property 2.1 for this function, we can automate the proof of unique decomposition for the specified reduction semantics. It seems plausible to make the development modular, with the use of Coq module types for describing signatures of reduction semantics and the intermediate steps.

In particular, we can imagine the following signature of a reduction semantics:

Parameters `term value redex context` : **Set**.
Parameter `plug` : `term` → `context` → `term`.

Then we can define an eval/apply abstract machine realizing the reduction strategy specified by this semantics as a transition system, i.e., the transitive closure of a transition function:

Parameter `configuration` : **Set**.
Parameter `transition` : `configuration` → `configuration` → **Prop**.
Parameter `transitive_closure` : `configuration` → `configuration` → **Prop**.

and such that the following properties hold:

Axiom `search_redex` :
$\forall\, t, c, c', r.\, \texttt{plug}\,(t,\,c) = \texttt{plug}\,(r,\,c') \rightarrow \texttt{c\_eval}\,(t,\,c) \rhd^{+} \texttt{c\_eval}\,(r,\,c')$.
Axiom `search_value` :
$\forall\, t, c, v.\, \texttt{plug}\,(t,\,c) = \texttt{plug}\,(v,\,\texttt{mt}) \rightarrow \texttt{c\_eval}\,(t,\,c) \rhd^{+} \texttt{c\_final}\,v$.
Axiom `decompose_redex` :
$\forall\, r, t, c.\, \texttt{contract}\,(r) = \texttt{Some}\,t \rightarrow \texttt{c\_eval}\,(r,\,c) \rhd^{+} \texttt{c\_eval}\,(t,\,c)$.

These properties characterize the correspondence between a reduction semantics and an abstract machine extensionally. If we can prove that these properties hold (regardless of the particular implementation of the decomposing function), then we ensure that the abstract machine—represented by transitive_closure—realizes the reduction strategy of the given reduction semantics. This issue has been studied also by Hardin et al. [12] who considered the converse direction and identified the reduc-

tion strategies "wired" in several virtual machines for the λ-calculus, by establishing a bisimulation between the machines and the underlying reduction semantics (of λ-calculi with explicit substitutions), and thus justifying each machine transition with respect to the underlying reduction semantics. It would also be interesting to connect our extensional approach to reduction strategies with the intentional, transition-system-based approach.

Looking at other examples of refocusing, we are able to identify tactics also for closure unfolding (leading to environment machines) and store unfolding (leading to store machines), and to extend the method to work for context-sensitive reduction semantics where contraction rules take into account not only the redex but also its surrounding context (as found in languages with global effects).

Taking this idea further, we would like to be able to automatically generate the abstract machine corresponding to a given reduction semantics, together with the proof of its correctness. To this end, we need to develop a meta-language for representing such abstract machines and manipulate them as Coq objects.

5 Conclusion

We have described a formalization in Coq of the refocusing method applied to two prototypical programming languages: the call-by-value λ-calculus, and the call-by-name calculus of closures. These case studies are to serve as a basis for a general framework for representing programming languages and their operational semantics in Coq, and to automate their interderivations, while ensuring their correctness.

A generalized refocusing method has been shown to apply to languages with a non-standard (context-sensitive) notion of reduction, e.g., languages with control operators, state, lazy evaluation, etc., as studied by Biernacka and Danvy [6]. Such languages are in common use, therefore our intended framework seems of significant practical importance.

References

[1] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. In Dale Miller, editor, *Proceedings of the Fifth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'03)*, pages 8–19, Uppsala, Sweden, August 2003. ACM Press.

[2] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. A functional correspondence between call-by-need evaluators and lazy abstract machines. *Information Processing Letters*, 90(5):223–232, 2004. Extended version available as the research report BRICS RS-04-3.

[3] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. A functional correspondence between monadic evaluators and abstract machines for languages with computational effects. *Theoretical Computer Science*, 342(1):149–172, 2005. Extended version available as the research report BRICS RS-04-28.

[4] Małgorzata Biernacka, Dariusz Biernacki, and Olivier Danvy. An operational foundation for delimited continuations in the CPS hierarchy. *Logical Methods in Computer Science*, 1(2:5):1–39, November 2005. A preliminary version was presented at the Fourth ACM SIGPLAN Workshop on Continuations (CW'04).

[5] Małgorzata Biernacka and Olivier Danvy. A concrete framework for environment machines. *ACM Transactions on Computational Logic*, 2006. To appear. Available as the research report BRICS RS-06-3.

[6] Małgorzata Biernacka and Olivier Danvy. A syntactic correspondence between context-sensitive calculi and abstract machines. *Theoretical Computer Science*, 375:76–108, 2007. Extended version available as the research report BRICS RS-06-18.

[7] The Coq Development Team. *The Coq Proof Assistant Reference Manual, Version 8.1*, 2006. http://coq.inria.fr.

[8] Pierre-Louis Curien. An abstract framework for environment machines. *Theoretical Computer Science*, 82:389–402, 1991.

[9] Olivier Danvy. From reduction-based to reduction-free normalization. Research Report BRICS RS-04-30, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, December 2004. Invited talk at the 4th International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2004), Aachen, Germany, June 2, 2004. To appear in ENTCS.

[10] Olivier Danvy and Lasse R. Nielsen. Syntactic theories in practice. In Mark van den Brand and Rakesh M. Verma, editors, *Informal proceedings of the Second International Workshop on Rule-Based Programming (RULE 2001)*, volume 59.4 of *Electronic Notes in Theoretical Computer Science*, Firenze, Italy, September 2001.

[11] Matthias Felleisen and Daniel P. Friedman. Control operators, the SECD machine, and the λ-calculus. In Martin Wirsing, editor, *Formal Description of Programming Concepts III*, pages 193–217. Elsevier Science Publishers B.V. (North-Holland), Amsterdam, 1986.

[12] Thérèse Hardin, Luc Maranget, and Bruno Pagano. Functional runtime systems within the lambda-sigma calculus. *Journal of Functional Programming*, 8(2):131–172, 1998.

On Term-Graph Rewrite Strategies

Rachid Echahed[1,2]

CNRS, LIG Laboratory
46, avenue Felix Viallet
38000 Grenoble
France

Keywords: Data-structures transformation, Graph Rewrite Systems, Rewrite Strategies

Graph rewriting is being investigated in various areas nowadays (see for instance [19,11,12]). In this talk, we consider term-graph rewriting as the underlying operational semantics of rule-based (functional or logic) programming languages (e.g. [17,8,9]). There are many reasons that motivate the use of term-graphs. They actually allow sharing of subexpressions which leads to efficient computations [3,2]. They also permit to go beyond the processing of first-order terms by handling efficiently real-world data-structures represented by cyclic graphs such as doubly-linked list or circular lists, e.g. [6,9].

Using a term-graph rewrite system (tGRS) is not an easy task. Indeed, the classical properties of term rewriting systems (TRS) cannot be lifted without caution to tGRSs. One of these properties is confluence. Let us consider the rule $F(a, a, x) \rightarrow x$ where a is a constant and x is a variable. This rule, which constitutes an orthogonal TRS, generates a confluent rewrite relation over (finite or infinite) terms whereas it generates a non confluent rewrite relation over term-graphs (see, e.g., [14]). It is well-known that this source of non-confluence of tGRSs comes from the so-called "collapsing rules" in orthogonal tGRSs. A rewrite rule is collapsing if its right-hand side is a variable. However, collapsing rules are very often used in programming and thus cannot be prohibited in any programming discipline. Most of access functions are defined by means of collapsing rules such as :

```
car (cons (x,u)) -> x
cdr (cons (x,u)) -> u
left-tree (bin-tree (l,x,r)) -> l
```

[1] Email:Rachid.Echahed@imag.fr

[2] This work has been partly funded by the project ARROWS of the French *Agence Nationale de la Recherche*.

```
right-tree (bin-tree (l,x,r)) -> r
```

The term-graphs we consider are supposed to represent data-structures and are the ones introduced in [4]. We define below such graphs in a mono-sorted setting. Lifting the presented results to the many-sorted case is straightforward.

Let Ω be a set of operation symbols such that each symbol in Ω, say f, is provided with a natural number, n, representing its *arity*. A *term-graph* G over Ω is defined by:

- a set of *nodes* $\mathcal{N}_G$,
- a subset of *labeled nodes* $\mathcal{N}_G^{\Omega} \subseteq \mathcal{N}_G$,
- a *labeling function* $\mathcal{L}_G : \mathcal{N}_G^{\Omega} \to \Omega$,
- a *successor function* $\mathcal{S}_G : \mathcal{N}_G^{\Omega} \to \mathcal{N}_G^*$,

such that for each labeled node n, the length of the string $\mathcal{S}_G(n)$ is the arity of the operation $\mathcal{L}_G(n)$. A rooted term-graph is a term-graph with a distinguished node: its root.

There are many ways in the literature to define term-graph rewrite systems (see,e.g., [18,4]). In this talk we will investigate orthogonal constructor-based tGRSs consisting of rewrite rules having the following shape :

$$[L \to R \mid \phi]$$

where R is a sequence of actions, ϕ is a node constraint and L is a rooted term-graph. A *node constraint* is a (possibly empty) conjunction of disequations between nodes: $\bigwedge_{i=1}^{n}(\alpha_i \neq \beta_i)$. The sequential execution of actions in R aims to rewrite a term-graph in a stepwise fashion. An action is one of the following forms :

- a **node definition** or **node labeling** $\alpha : f(\alpha_1, \ldots, \alpha_n)$ where $\alpha, \alpha_1, \ldots, \alpha_n$ are nodes and f is a label of arity n. This means that α is (re)labeled by f.
- an **edge redirection** or **local redirection** $\alpha \gg_i \beta$ where α, β are nodes and $i \in \{1, \ldots, \mathrm{ar}(\mathcal{L}(\alpha))\}$. This is an edge redirection and means that the target of the "ith" edge (argument) outgoing α is redirected to point β.
- a **global redirection** $\alpha \gg \beta$ where α and β are nodes. This means that all edges pointing α are redirected to point β.

The result of applying an action a to a term-graph t is denoted by $a[t]$ and is defined as the following term-graph s:

- If $a = \alpha : f(\alpha_1, \ldots, \alpha_n)$ then $\mathcal{N}_s = \mathcal{N}_t \cup \{\alpha, \alpha_1, \ldots, \alpha_n\}$, $\mathcal{L}_s(\alpha) = f$, $\mathcal{L}_s(\beta) = \mathcal{L}_t(\beta)$ if $\beta \neq \alpha$, and $\mathcal{S}_s(\alpha) = \alpha_1 \ldots \alpha_n$, $\mathcal{S}_s(\beta) = \mathcal{S}_t(\beta)$ if $\beta \neq \alpha$. $\cup$ denotes classical union.
- If $a = \alpha \gg_i \beta$ then $\mathcal{N}_s = \mathcal{N}_t$, $\mathcal{L}_s = \mathcal{L}_t$, and if $\mathcal{S}_t(\alpha) = \alpha_1 \ldots \alpha_i \ldots \alpha_n$ then $\mathcal{S}_s(\alpha) = \alpha_1 \ldots \alpha_{i-1}\ \beta\ \alpha_{i+1} \ldots \alpha_n$ and for any node γ we have $\mathcal{S}_s(\gamma) = \mathcal{S}_t(\gamma)$ iff $\gamma \neq \alpha$.
- If $a = \alpha \gg \beta$ then $\mathcal{N}_s = \mathcal{N}_t$, $\mathcal{L}_s = \mathcal{L}_t$ and for all nodes δ such that α occurs in $\mathcal{S}_t(\delta)$, *i.e.*, $\mathcal{S}_t(\delta) = \alpha_1 \ldots \alpha_i \ldots \alpha_n$ then $\mathcal{S}_s(\delta) = \alpha_1 \ldots \alpha_{i-1}\ \beta\ \alpha_{i+1} \ldots \alpha_n$ and for any node γ we have $\mathcal{S}_s(\gamma) = \mathcal{S}_t(\gamma)$ iff α does not occur in $\mathcal{S}_t(\gamma)$

As an example of term-graph rewrite systems, we define below two functions

length and *reverse*. The first function computes the length of a circular list, whereas *reverse* performs the so-called in-situ list reversal.

Length of a circular list :
$r : length(p) \to r : length'(p, p)$
$r : length'(p_1 : cons(n, p_2), p_2) \to r : s(0)$
$[r : length'(p_1 : cons(n, p_2), p_3) \mid p_2 \not\approx p_3] \to r : s(q); q : length'(p_2, p_3)$

In-situ list reversal :
$o : reverse(p) \to o : reverse'(p, q : nil)$
$o : reverse'(p_1 : cons(n, q : nil), p_2) \to p_1 \gg_2 p_2; o \gg p_1$
$o : reverse'(p_1 : cons(n, p_2 : cons(m, p_3), p_4) \to p_1 \gg_2 p_4; o \gg_1 p_2; o \gg_2 p_1$

Let $\rho : [L \to R \mid \phi]$ be a rewrite rule. A *ρ-matcher for* a term-graph t (*at* a node $\alpha \in \mathcal{N}_t$) is a homomorphisme σ from L to t satisfying the following conditions :

- σ is a solution of ϕ.
- $\alpha = \sigma(\mathcal{R}\text{oot}_L)$.

If σ is a ρ-matcher for t at α then t rewrites to the term-graph s s.t. $s = \sigma(R)[t]$. we write $t \to_{\rho,\sigma} \sigma(R)[t]$ or $t \to \sigma(R)[t]$.

As said earlier, rewrite relations induced by term-graph rewrite systems are not confluent in general. We shall discuss this issue and define classes of graphs and rewrite systems for which we can show the confluence of the rewrite relation. The confluence of a rewrite relation allows one to to evaluate expressions in a deterministic and efficient way by using rewrite strategies. Such strategies have been well investigated in the setting of finite and infinite orthogonal TRSs (e.g., [16,13,15]). In [1], a strategy that computes outermost needed redexes based on definitional trees has been designed in the framework of orthogonal constructor-based TRSs. We will show how Antoy's strategy can be extended to orthogonal constructor-based term-graph rewrite systems with the same nice properties. We particularly prove that the resulting strategy is c-hyper-normalizing on the class of admissible graphs and develops shortest derivations (see, e.g., [7,8]).

In recent work [5,10], term-graph rewrite strategies based on orderings have been proposed. The aim of such orderings is to ensure the computation of unique normal forms. These strategies will be discussed as well as future research directions.

References

[1] S. Antoy. Definitional trees. In *Proc. of the 4th Intl. Conf. on Algebraic and Logic programming*, pages 143–157. Springer Verlag LNCS 632, 1992.

[2] S. Antoy and B. Brassel. Computing with subspaces. In *Proc. of the 9th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP07)*. to appear, 2007.

[3] S. Antoy, D. W. Brown, and S.-H. Chiang. Lazy context cloning for non-deterministic graph rewriting. *Electr. Notes Theor. Comput. Sci.*, 176(1):3–23, 2007.

[4] H. Barendregt, M. van Eekelen, J. Glauert, R. Kenneway, M. J. Plasmeijer, and M. Sleep. Term graph rewriting. In *PARLE'87*, pages 141–158. Springer Verlag LNCS 259, 1987.

[5] R. Caferra, R. Echahed, and N. Peltier. Rewriting term-graphs with priority. In *Proc. of the 8th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP06)*, pages 109–120, 2006.

[6] D. Duval, R. Echahed, and F. Prost. Modeling pointer redirection as cyclic term-graph rewriting. *Electr. Notes Theor. Comput. Sci.*, 176(1):65–84, 2007.

[7] R. Echahed and J. Janodet. On constructor-based graph rewriting systems. Technical report 985-1, IMAG, 1997.

[8] R. Echahed and J. C. Janodet. Admissible graph rewriting and narrowing. In *Proc. of Joint International Conference and Symposium on Logic Programming (JICSLP'98)*, pages 325–340. MIT Press, June 1998.

[9] R. Echahed and N. Peltier. Narrowing data-structures with pointers. In *Procs of the 3rd International Conference on Graph Transformation (ICGT 2006)*, pages 92–106, 2006.

[10] R. Echahed and N. Peltier. Non strict confluent rewrite systems for data-structures with pointers. In *RTA*, page to appear, 2007.

[11] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 2: Applications, Languages and Tools*. World Scientific, 1999.

[12] H. Ehrig, H.-J. Kreowski, U. Montanari, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 3: Concurrency, Parallelism and Distribution*. World Scientific, 1999.

[13] G. Huet and J.-J. Lévy. Computations in orthogonal term rewriting systems. In J.-L. Lassez and G. Plotkin, editors, *Computational logic: essays in honour of Alan Robinson*. MIT Press, Cambridge, MA, 1991. Previous version: Call by need computations in non-ambiguous linear term rewriting systems, Technical Report 359, INRIA, Le Chesnay, France, 1979.

[14] J. R. Kennaway, J. K. Klop, M. R. Sleep, and F. J. D. Vries. On the adequacy of graph rewriting for simulating term rewriting. *ACM Transactions on Programming Languages and Systems*, 16(3):493–523, 1994. Previous version: Technical Report CS-R9204, CWI, Amsterdam, 1992.

[15] J. R. Kennaway, J. K. Klop, M. R. Sleep, and F. J. D. Vries. Transfinite reduction in orthogonal term rewriting systems. *Information and Computation*, pages 18–38, 1995.

[16] M. J. O'Donnell. *Computing in Systems Described by Equations*. Springer Verlag LNCS 58, 1977.

[17] R. Plasmeijer and M. van Eekelen. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley, 1993.

[18] D. Plump. Term graph rewriting. In H. Ehrig, G. Engels, H. J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2, pages 3–61. World Scientific, 1999.

[19] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997.

Towards a sharing strategy for the graph rewriting calculus

P. Baldan[a] and C. Bertolissi[b] and H. Cirstea[c] and C. Kirchner[d]

[a] *Dipartimento di Informatica, Università Ca' Foscari di Venezia, Italy*

[b] *LIF-Université de Provence, Marseille, France*

[c] *LORIA-UHP Nancy 1, France*

[d] *INRIA-LORIA, Nancy, France*

Abstract

The graph rewriting calculus is an extension of the ρ-calculus, handling graph like structures rather than simple terms. The calculus over terms is naturally generalized by using unification constraints in addition to the standard ρ-calculus matching constraints. The transformations are performed by explicit application of rewrite rules as first class entities. The possibility of expressing sharing and cycles allows one to represent and compute over regular infinite entities.
We propose in this paper a reduction strategy for the graph rewriting calculus which aims at maintaining the sharing information as long as possible in the terms. The corresponding reduction relation is shown to be confluent and complete *w.r.t.* the small-step semantics of the graph rewriting calculus.

1 Introduction

Main interest for term rewriting stem from functional and rewrite based languages as well as from theorem proving. In particular, we can describe the behaviour of a functional or rewrite based program by analyzing some properties of the associated term rewriting system. In this framework, terms are often seen as trees but in order to improve the efficiency of the implementation of such languages, it is of fundamental interest to think and implement terms as graphs [BvEG+87]. In this case, the possibility of sharing subterms allows to save space (by using multiple pointers to the same subterm instead of duplicating the subterm) and to save time (a redex appearing in a shared subterm will be reduced at most once and equality tests can be done in constant time when the sharing is maximal).

Graph rewriting is a useful technique for the optimization of functional and declarative languages implementations [PJ87]. Moreover, the possibility to define cycles leads to an increased expressive power that allows one to represent easily regular infinite data structures. Cyclic term graph rewriting has been widely studied, both from an operational [BvEG+87,AK96] and from a categorical/logical point of view [CG99] (see [SPvE93] for a survey on term graph rewriting).

The graph rewriting calculus, or ρ_g-calculus, introduced in [Ber05], is a common generalization of the cyclic λ-calculus [AK97] and the ρ-calculus [CK01], providing a framework where pattern matching, graphical structures and higher-order capabilities are primitive. The ρ_g-calculus deals with cyclic terms with bound variables and can express vertical sharing as well as horizontal sharing by means of a list of recursion equations. In the ρ_g-calculus computations related to the matching are made explicit and performed at the object-level. The calculus, under suitable linearity constraints for patterns, has been shown to be confluent [BBCK07] and expressive enough for simulating cyclic λ-calculus and term-graph rewriting.

In view of a future implementation, we are interested in improving the efficiency of the ρ_g-calculus. To this aim we present a reduction strategy aimed at keeping the sharing information as long as possible in ρ_g-calculus terms. In the ρ_g-calculus the loss of sharing is caused by the application of the substitution rules, which allow to create copies of (sub)terms of a ρ_g-calculus term. Indeed, during the computation, some loss of sharing is unavoidable, for example for making a rule application explicit or for solving a matching constraint. However, a strategy which suitably restricts the application of the substitution rules can avoid some useless loss of sharing, leading to more compact normal forms. The strategy should allow to perform essentially the same reductions to normal form as in the unconstrained calculus, in the sense that the normal form of a term with respect to the strategy (when it exists) should be the same as in the original calculus, up to sharing.

Indeed, we will show that, under suitable linearity constraints, the proposed strategy is correct and complete with respect to the reduction relation of the ρ_g-calculus. Additionally, we will show that the reduction relation of the ρ_g-calculus induced by such strategy is confluent.

The paper is organized as follows. In the first section we review the graph rewriting calculus. Section 3 describes the reduction strategy *SharingStrat* proposed for preserving sharing in ρ_g-calculus terms. In Section 4 we show that *SharingStrat* is sound and complete with respect to the small step semantics of the ρ_g-calculus. Moreover, along the lines of the proof of confluence for the ρ_g-calculus, we show that the ρ_g-calculus with *SharingStrat* is confluent. We conclude in Section 5 by presenting some perspectives of future work.

2 The graph rewriting calculus

The syntax of the ρ_g-calculus is presented in Fig. 1. As in the plain ρ-calculus, λ-abstraction is generalized by a rule abstraction $P \twoheadrightarrow G$, where P is in general an arbitrary term. There are two different application operators: the functional application operator, denoted simply by concatenation, and the constraint application operator, denoted by "$\_\ [\_]$". Terms can be grouped together into *structures* built using the operator "$\_ \wr \_$". This operator is useful for representing the (non-deterministic) application of a set of rewrite rules and consequently, the non-deterministic results. For example, the non-deterministic application of one of the rules in $\{a \twoheadrightarrow b,\ a \twoheadrightarrow c\}$ to the term a can be written $(a \twoheadrightarrow b \wr a \twoheadrightarrow c)\ a$. This term, as it will become clearer after the formal definition of the semantics of the calculus (see Figure 2), reduces to $(a \twoheadrightarrow b)\ a \wr (a \twoheadrightarrow c)\ a$ and then to $b \wr c$. Note

Terms			**Constraints**		
$\mathcal{G},\mathcal{P}$	$::= \mathcal{X}$	(Variables)	$\mathcal{C}$	$::= \epsilon$	(Empty constraint)
	$\mid \mathcal{K}$	(Constants)		$\mid \mathcal{X} = \mathcal{G}$	(Recursion equation)
	$\mid \mathcal{P} \twoheadrightarrow \mathcal{G}$	(Abstraction)		$\mid \mathcal{P} \ll \mathcal{G}$	(Match equation)
	$\mid \mathcal{G}\,\mathcal{G}$	(Functional application)		$\mid \mathcal{C},\mathcal{C}$	(Conjunction)
	$\mid \mathcal{G} \wr \mathcal{G}$	(Structure)			
	$\mid \mathcal{G}\,[\mathcal{C}]$	(Constraint application)			

Fig. 1. Syntax of the ρ_{g}-calculus

that the calculus is untyped, but type systems, in the style of those introduced for the ρ-calculus in [BCKL03,Wac05], would be conceivable.

In the ρ_{g}-calculus constraints are conjunctions (built using the operator "$\_,\_$") of match equations of the form $\mathcal{P} \ll \mathcal{G}$ and recursion equations of the form $\mathcal{X} = \mathcal{G}$. The empty constraint is denoted by ϵ. The operator "$\_,\_$" is supposed to be associative, commutative, with ϵ as neutral element.

We assume that the application operator associates to the left, while the other operators associate to the right. To simplify the syntax, operators have different priorities. Here are the operators ordered from higher to lower priority: "$\_\ \_$", "$\_ \twoheadrightarrow \_$", "$\_ \wr \_$", "$\_\,[\_]$" , "$\_ \ll \_$", "$\_ = \_$" and "$\_,\_$".

The symbols $G, H, P \ldots$ range over the set $\mathcal{G}$ of terms, $x, y, \ldots$ range over the set $\mathcal{X}$ of variables, $a, b, \ldots$ range over a set $\mathcal{K}$ of constants. The symbols $E, F, \ldots$ range over the set $\mathcal{C}$ of constraints.

We call *algebraic* the terms of the form $(((f\ \ G_1)\ \ G_2)\ldots)\ \ G_n$, with $f \in \mathcal{K}$, $G_i \in \mathcal{X} \cup \mathcal{K}$ or G_i algebraic for $i = 1 \ldots n$, and we usually denote them by $f(G_1, G_2, \ldots, G_n)$.

We denote by $\bullet$ (black hole) a constant, already introduced in [AK96] using the equational approach and also in [Cor93] using the categorical approach, to give a name to "undefined" terms that correspond to the expression $x\ [x = x]$ (self-loop). The notation $x =_\circ x$ is an abbreviation for the sequence $x = x_1, \ldots, x_n = x$. We use the symbol $\mathsf{Ctx}[\_]$ for a context with exactly one hole $\_$. We say that a ρ_{g}-term is *acyclic* if it contains no sequence of constraints of the form $\mathsf{Ctx}_0[x_0] \ll \ll \mathsf{Ctx}_1[x_1], \mathsf{Ctx}_2[x_1] \lll \mathsf{Ctx}_3[x_2], \ldots, \mathsf{Ctx}_m[x_n] \lll \mathsf{Ctx}_{m+1}[x_0]$, with $n, m \in \mathbb{N}$ and $\lll \in \{=, \ll\}$. A sequence of this kind is called a *cycle*.

For the purposes of this paper we restrict to left-hand sides of abstractions and match equations that are *acyclic*, *algebraic terms* in *normal form*. The set of all these terms, called *patterns*, is denoted by $\mathcal{P}$. For instance, the ρ_{g}-term $f(y)\ [y = g(y)] \twoheadrightarrow a$ is not allowed since the abstraction has a cyclic left-hand side.

A ρ_{g}-term is called *well-formed* if each variable occurs at most once as left-hand side of a recursion equation. All the ρ_{g}-terms considered in the sequel will be implicitly assumed to be well-formed.

The notions of free and bound variables of ρ_{g}-terms take into account the three binders of the calculus: abstraction, recursion and match. Intuitively, variables on the left hand-side of any of these operators are bound by the operator. As usual, we work modulo α-*conversion*. The set of free variables of a ρ_{g}-term G is denoted by $\mathcal{FV}(G)$. A variable in a term G is called *active*, or in *active position*, if it appears free in the left-hand side of an application occurring in G. Moreover, given a constraint

$\mathcal{C}$ we will refer to the set $\mathcal{DV}(\mathcal{C})$, of variables "defined" in $\mathcal{C}$. This set includes, for any recursion equation $x = G$ in $\mathcal{C}$, the variable x and for any match $P \ll G$ in $\mathcal{C}$, the set of free variables of P. For a formal definition, see [BBCK05].

Finally, in order to ensure the confluence of the calculus, we will assume all patterns to be linear. Roughly, a pattern is called linear if each variable occurs free at most once in the pattern.

Definition 2.1 (Linear ρ_{g}-calculus) *The class of (algebraic)* linear patterns *is defined as follows:*

$$\mathcal{L} \ ::= \ \mathcal{X} \ \mid \ \mathcal{K} \ \mid \ (((\mathcal{K}\,\mathcal{L}_0)\,\mathcal{L}_1)\ldots)\,\mathcal{L}_n \ \mid \ \mathcal{L}_0\,[\mathcal{X}_1 = \mathcal{L}_1, \ldots, \mathcal{X}_n = \mathcal{L}_n]$$

where we assume that $\mathcal{FV}(\mathcal{L}_i) \cap \mathcal{FV}(\mathcal{L}_j) = \emptyset$ *for* $i \neq j$. *A constraint* $[L_1 \ll \ll G_1, \ldots, L_n \lll G_n]$, *where* $\lll \in \{=, \ll\}$, *is* linear *if all patterns* $L_1, \ldots, L_n$ *are linear and* $\mathcal{FV}(L_i) \cap \mathcal{FV}(L_j) = \emptyset$, $i \neq j$. *The* linear ρ_{g}-calculus *is the calculus where all the patterns in the left-hand side of abstractions and all constraints are linear.*

In this paper we will focus on the linear ρ_{g}-calculus, hence the qualification "linear" will be often omitted, and the involved patterns and constraints will be assumed to be linear unless stated otherwise.

We define next an order over variables bound by a match or an equation. This order will be later used in the definition of the substitution rule of the calculus, which will allow one only "upward" substitutions, a constraint which is essential for the confluence of the calculus (see [BBCK07]). We denote by $\leq$ the least pre-order on recursion variables such that $x \geq y$ if $x = \mathsf{Ctx}[y]$ appears in the list of constraints for some context $\mathsf{Ctx}[\_]$. The equivalence induced by the pre-order is denoted $\equiv$ and we say that x and y are cyclically equivalent ($x \equiv y$) if $x \geq y \geq x$ (they lie on a common cycle). We write $x > y$ if $x \geq y$ and $x \not\equiv y$.

Example 2.2 [Some ρ_{g}-terms]

(i) In the rule $(2 * f(x)) \twoheadrightarrow ((y + y)\ [y = f(x)])$ the sharing in the right-hand side avoids the copying of the object instantiating $f(x)$, when the rule is applied to a ρ_{g}-term.

(ii) The ρ_{g}-term $x\ [x = cons(0, x)]$ represents an infinite list of zeros.

(iii) The ρ_{g}-term $f(x, y)\ [x = g(y), y = g(x)]$ is an example of twisted sharing that can be expressed using mutually recursive constraints (to be read as a `letrec` construct). We have that $x \geq y$ and $y \geq x$, hence $x \equiv y$.

The complete set of evaluation rules of the ρ_{g}-calculus is presented in Fig. 2. As in the plain ρ-calculus, in the ρ_{g}-calculus the application of a rewrite rule to a term is represented as the application of an abstraction. A redex can be activated using the ρ rule in the Basic rules, which creates the corresponding matching constraint. The computation of the substitution which solves the matching is then performed explicitly by the Matching rules and, if the computation is successful, the result is a recursion equation added to the list of constraints of the term. This means that the substitution is not applied immediately to the term but it is kept in the environment for a delayed application or for deletion if useless, as expressed by the Graph rules.

More precisely, the first two rules ρ and δ come from the ρ-calculus. For each of

Basic rules:

$$
\begin{array}{lll}
(\rho) & (P \twoheadrightarrow G_2)\, G_3 & \to_\rho\ G_2\ [P \ll G_3] \\
 & (P \twoheadrightarrow G_2)\ [E]\, G_3 & \to_\rho\ G_2\ [P \ll G_3, E] \\
(\delta) & (G_1 \wr G_2)\, G_3 & \to_\delta\ G_1\, G_3 \wr G_2\, G_3 \\
 & (G_1 \wr G_2)\ [E]\, G_3 & \to_\delta\ (G_1\, G_3 \wr G_2\, G_3)\ [E]
\end{array}
$$

Matching rules:

$$
\begin{array}{lll}
(\mathit{propagate}) & P \ll (G\ [E]) & \to_p\ P \ll G, E \quad \mathit{if}\ P \neq x \\
(\mathit{decompose}) & K(G_1, \ldots, G_n) \ll K(G'_1, \ldots, G'_n) & \to_{dk}\ G_1 \ll G'_1, \ldots, G_n \ll G'_n \\
 & & \quad \mathit{with}\ n \geq 0 \\
(\mathit{solved}) & x \ll G, E & \to_s\ x = G, E \quad \mathit{if}\ x \notin \mathcal{DV}(E)
\end{array}
$$

Graph rules:

$$
\begin{array}{lll}
(\mathit{external\ sub}) & \mathsf{Ctx}[y]\ [y = G, E] & \to_{es}\ \mathsf{Ctx}[G]\ [y = G, E] \\
(\mathit{acyclic\ sub}) & G\ [P \lll \mathsf{Ctx}[y], y = G_1, E] & \to_{ac}\ G\ [P \lll \mathsf{Ctx}[G_1], y = G_1, E] \\
 & & \quad \mathit{if}\ G_1\ \text{is a variable or}\ (x > y,\ \forall x \in \mathcal{FV}(P)) \\
 & & \quad \mathit{where} \lll \in \{=, \ll\} \\
(\mathit{garbage}) & G\ [E, x = G'] & \to_{gc}\ G\ [E] \\
 & & \quad \mathit{if}\ x \notin \mathcal{FV}(E) \cup \mathcal{FV}(G) \\
 & G\ [\epsilon] & \to_{gc}\ G \\
(\mathit{black\ hole}) & \mathsf{Ctx}[x]\ [x =_\circ x, E] & \to_{bh}\ \mathsf{Ctx}[\bullet]\ [x =_\circ x, E] \\
 & G\ [P \lll \mathsf{Ctx}[y], y =_\circ y, E] & \to_{bh}\ G\ [P \lll \mathsf{Ctx}[\bullet], y =_\circ y, E] \\
 & & \quad \mathit{if}\ \ x > y,\ \forall x \in \mathcal{FV}(P)
\end{array}
$$

Fig. 2. Small-step semantics of the ρ_{g}-calculus

these rules, an additional rule dealing with the presence of constraints is considered.

The Matching rules and in particular the rule *decompose* are strongly related to the theory modulo which we want to compute the solutions of the matching. In this paper we consider the syntactical matching, which is known to be decidable, but extensions to more elaborated theories are possible.

The Graph rules are inherited from the cyclic λ-calculus [AK97]. The first two rules make a copy of a ρ_{g}-term associated to a recursion variable into a term that is inside the scope of the corresponding constraint. As already mentioned, the substitution rule allows one to make the copies only upwards *w.r.t.* the order defined on the variables of ρ_{g}-terms. Recall that "$\_, \_$" is assumed to be associative, commutative and with ϵ as neutral element, and thus evaluation steps are performed modulo the corresponding theory.

We denote by $\mapsto_{\not\mathsf{g}}$ ($\mapsto_{\mathcal{M}}$) and $\mapsto\!\!\!\!\to_{\not\mathsf{g}}$ ($\mapsto\!\!\!\!\to_{\mathcal{M}}$) the relations induced by the set of rules of Fig. 2 and by the subset of Matching rules, respectively. For any two rules r and s belonging to this set, we will write $\mapsto\!\!\!\!\to_{r,s}$ to express the two steps $\mapsto_r \mapsto_s$.

As mentioned above, the (linear) ρ_{g}-calculus, with the rewrite relation $\mapsto_{\not\mathsf{g}}$, has been shown to be confluent [BBCK07]. A term G is in *normal form* if no one of the reduction rules of Fig. 2 can be applied to G. A reduction of a term H into its normal form G, when it exists, is denoted by $H \mapsto\!\!\!\!\to_{\not\mathsf{g}}^{!} G$.

Example 2.3 [A simple reduction]

$$
\begin{array}{ll}
 & (f(a,a) \twoheadrightarrow a)\ (f(y,y)\ [y = a]) \\
\mapsto_p & a\ [f(a,a) \ll f(y,y)\ [y = a]]\ \mapsto_p\ a\ [f(a,a) \ll f(y,y), y = a] \\
\mapsto_{dk} & a\ [a \ll y, a \ll y, y = a]\ =\ a\ [a \ll y, y = a] \quad \textit{(by idempotency)} \\
\mapsto_{ac} & a\ [a \ll a, y = a]\ \mapsto_{dk}\ a\ [y = a]\ \mapsto\!\!\!\!\to_{gc}\ a
\end{array}
$$

Example 2.4 [Reduction to the normal form]

Consider the term $G = f(y, y)\ [y = z\ f(a), z = f(x) \twoheadrightarrow x]$. We show one of the possible reductions of G to its normal form.

$$
\begin{aligned}
& f(y,y)\ [y = z\ f(a), z = f(x) \twoheadrightarrow x] \\
\mapsto_{ac}\ & f(y,y)\ [y = (f(x) \twoheadrightarrow x)\ f(a), z = f(x) \twoheadrightarrow x] \\
\mapsto_{p}\ & f(y,y)\ [y = x\ [f(x) \ll f(a)], z = f(x) \twoheadrightarrow x] \\
\mapsto_{dk}\ & f(y,y)\ [y = x\ [x \ll a], z = f(x) \twoheadrightarrow x] \\
\mapsto_{s}\ & f(y,y)\ [y = x\ [x = a], z = f(x) \twoheadrightarrow x] \\
\mapsto_{es}\ & f(y,y)\ [y = a\ [x = a], z = f(x) \twoheadrightarrow x] \\
\mapsto_{gc}\ & f(y,y)\ [y = a, z = f(x) \twoheadrightarrow x] \\
\mapsto_{es}\ & f(a,a)\ [y = a, z = f(x) \twoheadrightarrow x] \mapsto_{gc} f(a,a)
\end{aligned}
$$

Example 2.5 [Encoding of the Peano addition]

We suppose given the constants $0, S, add$ and rec. We define the following ρ-term that computes the addition over Peano integers.

$$
plus \triangleq (rec\, z) \twoheadrightarrow \left(\begin{array}{l} (add\, 0\, y) \twoheadrightarrow y \\ \wr(add\, (S\, x)\, y) \twoheadrightarrow S\ (z\, (rec\, z)\, (add\, x\, y)) \end{array} \right)
$$

The variable z will contain a copy of $plus$ to allow "recursive calls". If we use the notations $\overline{m}$, $\overline{m+n}$ and $\overline{m-n}$ for the terms $S(\ldots(S\, 0)\ldots)$ with the right number of S symbols, then the term $plus\, (rec\ plus)\ (add\, \overline{n}\, \overline{m})$ reduces to $\overline{m+n}$. Actually, to obtain this result we also need a way of getting rid of some stuck subterms, in which matching definitively fails (see [CLW03,CHW06]).

3 A sharing strategy for ρ_{g}-calculus

In view of a future efficient implementation of the calculus, we are interested in studying suitable strategies that aim at keeping the sharing information as long as possible in ρ_{g}-terms.

Intuitively, the strategy should delay as much as possible the application of the substitution rules, (*external sub*) and (*acyclic sub*), which can break the sharing by duplicating terms. For instance, consider the reduction

$$
\begin{aligned}
& f(x,x)\ [x = (a \twoheadrightarrow g(b))a] \\
\mapsto_{es}\ & f((a \twoheadrightarrow g(b))a, x)\ [x = (a \twoheadrightarrow g(b))a] \\
\mapsto_{es}\ & f((a \twoheadrightarrow g(b))a, (a \twoheadrightarrow g(b))a)\ [x = (a \twoheadrightarrow g(b))a] \\
\mapsto_{p}\ & f(g(b)\ [a \ll a], g(b)\ [a \ll a])\ [x = (a \twoheadrightarrow g(b))a] \\
\mapsto_{p}\ & f(g(b)\ [\epsilon], g(b)\ [\epsilon])\ [x = (a \twoheadrightarrow g(b))a] \\
\mapsto_{gc}\ & f(g(b), g(b))
\end{aligned}
$$

The uncontrolled use of substitution induces useless and expensive (both in terms of time and space) duplications of terms. For instance, in the case above, the following reduction would be preferable

$$\begin{aligned} & f(x,x)\ [x = (a \twoheadrightarrow g(b))a] \\ \mapsto_{\rho}\ & f(x,x)\ [x = g(b)\ [a \ll a]] \\ \mapsto_{p}\ & f(x,x)\ [x = g(b)\ [\epsilon]] \\ \mapsto\!\!\!\!\!\rightarrow_{gc}\ & f(x,x)\ [x = g(b)] \end{aligned}$$

The idea underlying the proposed strategy is to constrain substitution rules to be applied only if they are needed for generating new redexes for the basic or matching rules. Note that, in particular, substitutions which do not contribute to generating new basic or matching redexes will never be applied. Hence the strategy will enlarge the class of terms which are in normal form.

For instance, we allow the application of the (*external sub*) rule to the terms $x\ a\ [x = f(x) \twoheadrightarrow x]$ or $x\ a\ [x = a \wr (a \twoheadrightarrow b)]$, since this is useful for creating, respectively, a new (ρ) redex and a new (δ) redex. Instead, (*external sub*) cannot be applied to the terms $f(x,x)\ [x = g(x)]$ or $x\ [x = f(x)]$ which are actually considered in normal form. Note, however, that capturing the notion of "substitution needed for generating a new redex" is not straightforward since more than one substitution step can be needed to generate a new redex for the basic or matching rules as it happens below, where the generated redex is underlined:

$$y\ [y = x\ f(a), x = f(z) \twoheadrightarrow y] \mapsto_{es} x\ f(a)\ [y = x\ f(a), x = f(z) \twoheadrightarrow y]$$

$$\mapsto_{es} \underline{(f(z) \twoheadrightarrow y)\ f(a)}\ [y = x\ f(a), x = f(z) \twoheadrightarrow z]$$

Note that a single step would suffice to generate the redex if we removed the acyclicity constraint for substitutions, allowing the reduction

$$y\ [y = x\ f(a), x = f(z) \twoheadrightarrow y] \mapsto y\ [y = (f(z) \twoheadrightarrow y)\ f(a), x = f(z) \twoheadrightarrow y]$$

The definition of the strategy will rely on the fact, formally proved later, that the above phenomenon is an instance of a completely general case.

There is one more situation in which we want to apply the substitution rules, that is when we have trivial recursion equations of the kind $x = y$ where both sides are single variables, like in $x * y + x\ [x = z, y = z, z = 1]$. In this case, we may want to simplify the term to $(z * z + z)\ [z = 1]$ in which useless names have been eliminated by garbage collection.

Hereafter, we call *basic redex* any term which has one of the shapes $(P \twoheadrightarrow G_2)\ G_3$, $(P \twoheadrightarrow G_2)\ [E]\ G_3$, $(G_1 \wr G_2)\ G_3$ or $(G_1 \wr G_2)\ [E]\ G_3$, which can be reduced using the *Basic rules* in Fig. 2. Similarly, a term of the form $P \ll G$ is called a *matching* redex if it can be reduced by one of the Matching rules.

We define next the reduction strategy we can adopt in the ρ_{g}-calculus to maintain the sharing information during the reduction as long as possible.

Definition 3.1 [Sharing Strategy] The evaluation strategy *SharingStrat* is defined as follows.

(i) All the evaluation rules but (*external sub*) and (*acyclic sub*) are applicable without any restriction.

(ii) The rules (*external sub*) and (*acyclic sub*) are applied to a term G only if no other rule is applicable and if
  (a) their application replaces a variable by a variable (renaming), or
  (b) their application creates (in one step) a basic or a matching redex, or
  (c) the term G has the form $G'\ [x = \mathsf{Ctx}[y], y = \mathsf{Ctx}'[z], E]$, with $x \equiv y$ and $\mathsf{Ctx}[\mathsf{Ctx}'[x]]$ includes a basic or a matching redex.

In other words the rules (*external sub*) and (*acyclic sub*) are applied when their application leads to

- the instantiation of a variable by a variable (condition *(ii)a*);
- the instantiation of an active variable by an abstraction or a structure, which produces a *Basic redex* (condition *(ii)b*);
- the instantiation of a variable in a stuck match equation, which produces a *Matching redex*, i.e., which enables a decomposition or constraint propagation *w.r.t.* the match equation (condition *(ii)b*).

Additionally, condition *(ii)c* captures the fact that, given a term $G\ [E]$ if a cyclic substitution in E would generate a redex, then one is allowed to apply some external substitutions in order to reproduce the same redex in G.

Example 3.2 [Multiplication]

Let us use an infix notation for the constant "$*$". The following ρ_g-term corresponds to the application of the rewrite rule $\mathcal{R} = x * s(y) \twoheadrightarrow (x * y + x)$ to the term $1 * s(1)$ where the constant 1 is shared.

$$
\begin{aligned}
& (x * s(y) \twoheadrightarrow (x * y + x))\ (z * s(z)\ [z = 1]) \\
\mapsto_{\rho}\ & x * y + x\ [x * s(y) \ll (z * s(z)\ [z = 1])] \\
\mapsto_{p}\ & x * y + x\ [x * s(y) \ll z * s(z), z = 1] \\
\mapsto\!\!\!\!\rightarrow_{dk}\ & x * y + x\ [x \ll z, y \ll z, z = 1] \\
\mapsto\!\!\!\!\rightarrow_{s}\ & x * y + x\ [x = z, y = z, z = 1] \\
\mapsto\!\!\!\!\rightarrow_{es}\ & (z * z + z)\ [x = z, y = z, z = 1] \quad \textit{(allowed by Def. 3.1(ii) a))} \\
\mapsto\!\!\!\!\rightarrow_{gc}\ & (z * z + z)\ [z = 1]
\end{aligned}
$$

Notice that the term $(z * z + z)\ [z = 1]$ is in normal form *w.r.t.* the strategy *SharingStrat* but can be reduced to $(1 * 1 + 1)$ if no evaluation strategy is used.

Example 3.3 [Reduction to normal form]

We consider the term G of Example 2.4 and we reduce it following the strategy *SharingStrat*. We obtain:

$$
\begin{aligned}
& f(y, y)\ [y = z\ f(a), z = f(x) \twoheadrightarrow x] \\
\mapsto_{ac}\ & f(y, y)\ [y = (f(x) \twoheadrightarrow x)\ f(a), z = f(x) \twoheadrightarrow x] \quad \textit{(by Def. 3.1(ii) b))} \\
\mapsto_{\rho}\ & f(y, y)\ [y = x\ [f(x) \ll f(a)], z = f(x) \twoheadrightarrow x]
\end{aligned}
$$

$\mapsto_{dk}\ f(y,y)\ [y = x\ [x \ll a], z = f(x) \twoheadrightarrow x]$

$\mapsto\!\!\!\!\!\rightarrow_{s}\ \ f(y,y)\ [y = x\ [x = a], z = f(x) \twoheadrightarrow x]$

$\mapsto_{gc}\ f(y,y)\ [y = x\ [x = a]]$

Note that the normal form with respect to *SharingStrat*, i.e., the term $f(y,y)\ [y = x\ [x = a]]$, represents a graph where the arguments of f are shared. Instead, as shown in Example 2.4, the reduction in the ρ_{g}-calculus with no evaluation strategy leads to the term $f(a,a)$ where the arguments of f are duplicated.

Example 3.4 Consider the ρ_{g}-term $G = f(y)[y = x\ a, x = y \wr b]$. Notice that $x \equiv y$, thus the (*acyclic sub*) rule cannot be applied. We have instead the reduction:

$f(y)[y = x\ a, x = y \wr b] \mapsto_{es} f(x\ a)[y = x\ a, x = y \wr b] \mapsto_{es}$

$f((y \wr b)\ a)[y = x\ a, x = y \wr b] \mapsto_{\delta} f((y\ a \wr b\ a))[y = x\ a, x = y \wr b]$

This derivation is a valid derivation using the strategy *SharingStrat*. Indeed, there exists a cyclic substitution step which transforms the pre-redex $x\ a$ into a basic redex $(y \wr b)\ a$. Hence, the first (*external sub*) rule step can be performed following Def. 3.1(ii) c). The second (*external sub*) rule step is needed to create the basic redex $(y \wr b)\ a$, thus it is allowed for Def. 3.1(ii) b).

4 Properties of the sharing strategy

In this section we will show some basic properties of the ρ_{g}-calculus with the evaluation strategy *SharingStrat*. First, we will show the soundness and completeness of the strategy *SharingStrat w.r.t.* ρ_{g}-calculus (normalising) derivations. In the second part, we will adapt the proof of confluence described in [BBCK07] in order to prove that the ρ_{g}-calculus with the strategy *SharingStrat* is still confluent.

4.1 Soundness and completeness

Here we prove that the reduction strategy proposed for the ρ_{g}-calculus is sound and complete with respect to the one step semantics of the ρ_{g}-calculus as defined in Section 2. Actually, while soundness is immediate, completeness will be proved only for normalising reductions.

Proposition 4.1 (Soundness) *Given two ρ_{g}-terms G and G_n, if $G \mapsto\!\!\!\!\!\rightarrow_{\not\rho\mathsf{g}} G_n$ in the ρ_{g}-calculus with the strategy* SharingStrat*, then $G \mapsto\!\!\!\!\!\rightarrow_{\not\rho\mathsf{g}} G_n$ in the ρ_{g}-calculus.*

Proof. Trivial. □

The completeness result relies on a couple of technical lemmata. In the proofs, it will be convenient to refer to a notion of *cyclic substitution*, which consists of the application of the rule *(ac)* without any restriction on the order of variables

$$G\ [P \lll \mathsf{Ctx}[y], y = G_1, E] \to_c G\ [P \lll \mathsf{Ctx}[G_1], y = G_1, E]$$

We will denote by $\mapsto_s$ any application of the substitution rules, *i.e.* (*external sub*) or (*acyclic sub*), possibly cyclic, if this is specified.

We remark that these cyclic substitutions are not allowed in the calculus, but they are just used as a technically convenient tool in proofs. In particular we will use the following simple fact.

Proposition 4.2 *Let G be $G'\ [x = \mathsf{Ctx}[y], y = \mathsf{Ctx}'[z], E]$ with $\{x, y\} \cap \mathcal{FV}(G') \neq \emptyset$. If $G \mapsto_c G'\ [x = \mathsf{Ctx}[\mathsf{Ctx}'[z]], y = \mathsf{Ctx}'[z], E]$ and $\mathsf{Ctx}[\mathsf{Ctx}'[z]]$ includes a basic or matching redex, then G is not in normal form with respect to* SharingStrat.

Proof. If $x \equiv y$, then by Definition 3.1*(ii)c* we can apply an external substitution, replacing in G' x or y with their definition. Otherwise, the considered step is a valid application of rule (ac). □

Lemma 4.3 *Let $\mathsf{Ctx}[\_], \mathsf{Ctx}_1[\_]$ be two ρ-contexts such that $\mathsf{Ctx}_1[x]$ is neither a free variable nor a free constrained variable (i.e. neither x nor $x\ [E]$ with $\mathcal{DV}(E) \cap \{x\} = \emptyset$). Let $\mathsf{Ctx}[\mathsf{Ctx}_1[y]]$ be a ρ-term without redexes and $\mathsf{Ctx}[\mathsf{Ctx}_1[G]]$ be a ρ-term containing a* Basic *or* Matching *redex. Then, $\mathsf{Ctx}_1[G]$ contains a* Basic *or* Matching *redex.*

Proof. [Sketch.] By induction on the form of $\mathsf{Ctx}[\_]$. The interesting cases are the followings:

- $\mathsf{Ctx}[\_]$ *is an application.* In this case $\mathsf{Ctx}[\_] = \mathsf{Ctx}'[\_]\ G'$ or $\mathsf{Ctx}[\_] = G'\ \mathsf{Ctx}'[\_]$. If $\mathsf{Ctx}'[x]$ is not a free (constrained) variable, then we conclude by inductive hypothesis. Hence, either $\mathsf{Ctx}[\_] = \_\ G'$ or $\mathsf{Ctx}[\_] = G'\ \_$.

  If $\mathsf{Ctx}[\_] = \_\ G'$, note that $\mathsf{Ctx}_1[y]$ cannot be a variable and it cannot be an abstraction or a structure (otherwise $\mathsf{Ctx}[\mathsf{Ctx}_1[y]]$ would contain a redex). If $\mathsf{Ctx}_1[y]$ is an application the property clearly holds since no new redexes can be created by instantiation. If $\mathsf{Ctx}_1[y]$ is a constraint application of the form $H\ [E]$ then, again, H cannot be an abstraction or a structure (otherwise $\mathsf{Ctx}[\mathsf{Ctx}_1[y]]$ would contain a redex). If $H = y$ then it can be instantiated by G and create a new redex only if $y \notin \mathcal{DV}(E)$ but this contradicts the hypothesis.

  If instead, $\mathsf{Ctx}[\_] = G'\ \_$ the property trivially holds.
- $\mathsf{Ctx}[\_]$ is a matching equation. Then $\mathsf{Ctx}[\_] = G' \ll \mathsf{Ctx}'[\_]$, and as above, when $\mathsf{Ctx}'[\_]$ is non-empty we conclude by inductive hypothesis. Thus, let $\mathsf{Ctx}[\_] = \_ G'$. Since the term $f(G_1, G_2, \ldots, G_n) \ll \mathsf{Ctx}_1[y]$ contains redexes then $\mathsf{Ctx}_1[y]$ is not of the form $f(H_1, H_2, \ldots, H_n)$ or $H\ [E]$. The term $f(G_1, G_2, \ldots, G_n) \ll \mathsf{Ctx}_1[G]$ is a redex only if $\mathsf{Ctx}_1[G]$ has the form $f(H_1, H_2, \ldots, H_n)$ or $H\ [E]$ and this is possible only if $\mathsf{Ctx}_1[x]$ is a free (constrained) variable (which contradicts the hypothesis).

□

A key point is that it is not possible to create a Basic or Matching redex by further reducing a term that is in normal form *w.r.t.* the reduction strategy. To prove this result, we use the following lemma.

Lemma 4.4 *Let G, G_1, G_2 be ρ_{g}-terms not containing trivial recursion equations, i.e. equations of the form $x = y$. Let $G \mapsto_s G_1 \mapsto_s G_2$ be two (possibly cyclic) substitution steps such that G and G_1 do not contain a* Basic *or* Matching *redex and G_2 does. Then, there exists a (possibly cyclic) substitution step $G \mapsto_s G_2'$, such that the* Basic *or* Matching *redex is present in G_2'.*

Proof.

- Consider the two-steps *external sub* reduction $\mathsf{Ctx}[y]\ [y = \mathsf{Ctx}_1[z], z = H, E] \mapsto_{es} \mathsf{Ctx}[\mathsf{Ctx}_1[z]]\ [y = \mathsf{Ctx}_1[z], z = H, E] \mapsto_{es} \mathsf{Ctx}[\mathsf{Ctx}_1[H]]\ [y = \mathsf{Ctx}_1[z], z = H, E]$ where only the last term contains a Basic or Matching redex. Since by hypothesis $\mathsf{Ctx}_1[\_]$ is not empty, by Lemma 4.3, we know that the redex is in the term $\mathsf{Ctx}_1[H]$. Hence we can build the following one-step reduction: $\mathsf{Ctx}[y]\ [y = \mathsf{Ctx}_1[z], z = H, E] \mapsto_{ac} \mathsf{Ctx}[y]\ [y = \mathsf{Ctx}_1[H], z = H, E]$ Notice that this substitution step may be cyclic, if y and z are cyclically equivalent.
- For the two-steps *acyclic sub* reduction $x\ [x = \mathsf{Ctx}[y], y = \mathsf{Ctx}_1[z], z = H, E] \mapsto_{ac} x\ [x = \mathsf{Ctx}[\mathsf{Ctx}_1[z]], y = \mathsf{Ctx}_1[z], z = H, E] \mapsto_{es} x\ [x = \mathsf{Ctx}[\mathsf{Ctx}_1[H]], y = \mathsf{Ctx}_1[z], z = H, E]$ we proceed similarly as in the previous case.
- Consider the two-steps reduction $\mathsf{Ctx}[y]\ [y = \mathsf{Ctx}_1[z], z = H, E] \mapsto_{ac} \mathsf{Ctx}[y]\ [y = \mathsf{Ctx}_1[H], z = H, E] \mapsto_{es} \mathsf{Ctx}[\mathsf{Ctx}_1[H]]\ [y = \mathsf{Ctx}_1[H], z = H, E]$ Since by hypothesis $\mathsf{Ctx}_1[\_]$ is not empty, the first *acyclic sub* step instantiates a variable in the term $\mathsf{Ctx}_1[z]$ without changing its structure. Thus, to create a redex, it is sufficient to perform the one-step reduction $\mathsf{Ctx}[y]\ [y = \mathsf{Ctx}_1[z], z = H, E] \mapsto_{es} \mathsf{Ctx}[\mathsf{Ctx}_1[z]]\ [y = \mathsf{Ctx}_1[z], z = H, E]$.
- Consider the two-steps reduction $\mathsf{Ctx}[y]\ [y = \mathsf{Ctx}_1[z], z = H, E] \mapsto_{es} \mathsf{Ctx}[\mathsf{Ctx}_1[z]]\ [y = \mathsf{Ctx}_1[z], z = H, E] \mapsto_{ac} \mathsf{Ctx}[\mathsf{Ctx}_1[z]]\ [y = \mathsf{Ctx}_1[H], z = H, E]$ The redex is created in $\mathsf{Ctx}_1[H]$. The first *external sub* step copies a sub-term in the graph but is without effect *w.r.t.* redex creation. We thus can build the one-step reduction $\mathsf{Ctx}[y]\ [y = \mathsf{Ctx}_1[z], z = H, E] \mapsto_{ac} \mathsf{Ctx}[y]\ [y = \mathsf{Ctx}_1[H], z = H, E]$.

□

Corollary 4.5 *Let G be a ρ_{g}-term with no trivial recursion equations, and let $G \mapsto\!\!\!\!\twoheadrightarrow_{\mathsf{s}} G_n$ such that G_n contains a* Basic *or* Matching *redex and G does not, then there exists a a (possibly cyclic) substitution step $G \mapsto_s G'_n$, such that the* Basic *or* Matching *redex is present in G'_n.*

Proof. By induction, using Lemma 4.4. □

Using the above result and exploiting Proposition 4.2 we easily prove the result below, from which completeness follows.

Corollary 4.6 *If a ρ_{g}-term G is in normal form with respect to the strategy* SharingStrat *and $G \mapsto\!\!\!\!\twoheadrightarrow_{es,ac} G_n$, then G_n is in normal form with respect to the strategy* SharingStrat.

Theorem 4.7 (Completeness) *Given a normalising ρ_{g}-term G, if $G \mapsto\!\!\!\!\twoheadrightarrow_{\not\rho\mathsf{g}}^{!} G_n$ in the ρ_{g}-calculus, then there exists a ρ_{g}-term G_m such that $G \mapsto\!\!\!\!\twoheadrightarrow_{\not\rho\mathsf{g}}^{!} G_m$ in the ρ_{g}-calculus with the strategy* SharingStrat *and $G_m \mapsto\!\!\!\!\twoheadrightarrow_{es,ac} G_n$.*

Proof. First, notice that in the ρ_{g}-calculus with the strategy *SharingStrat*, the reduction $G \mapsto\!\!\!\!\twoheadrightarrow_{\not\rho\mathsf{g}} \ldots$ cannot be infinite, otherwise we would have an infinite reduction also in the ρ_{g}-calculus. Thus we have $G \mapsto\!\!\!\!\twoheadrightarrow_{\not\rho\mathsf{g}}^{!} G_m$. Moreover, we have $G_m \mapsto\!\!\!\!\twoheadrightarrow_{\not\rho\mathsf{g}}^{!} G_n$ in the ρ_{g}-calculus, since the calculus is confluent. In order to conclude we have to prove that $G_m \mapsto\!\!\!\!\twoheadrightarrow_{es,ac} G_n$ (using only substitution steps).

This follows immediately from Corollary 4.6. In fact, by contradiction, if there

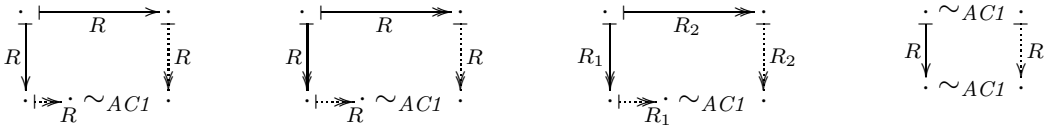


Fig. 3. Properties of rewriting modulo *AC1*

were a reduction $G_n \mapsto\!\!\!\!\!\rightarrow_{es,ac} G'_n \mapsto_M G''_n$, then, by Corollary 4.6, G_n would not be in normal form *w.r.t.* the strategy *SharingStrat*. □

Notice that we cannot expect completeness to hold in general, since "useless" unsharing followed by the reduction of some basic or matching redexes cannot be simulated while obeying *SharingStrat*. For instance, the result in Theorem 4.7 would not hold for the derivation

$$\begin{aligned} & f(x,x)\ [x = (a \twoheadrightarrow b)\ a] \\ & \quad \mapsto_{es} f((a \twoheadrightarrow b)\ a, x)\ [x = (a \twoheadrightarrow b)\ a] \\ & \quad \mapsto\!\!\!\!\!\rightarrow_{p,dk} f(b,x)\ [x = (a \twoheadrightarrow b)\ a] \end{aligned}$$

4.2 Confluence

We will next show that the (linear) ρ_{g}-calculus calculus with the evaluation strategy *SharingStrat* is confluent. The proof is obtained by adapting the confluence proof for the ρ_{g}-calculus [BBCK07].

As already mentioned, in the ρ_{g}-calculus, rewriting can be thought of as acting over equivalence classes of ρ_{g}-graphs with respect to the congruence relation, denoted by $\sim_{AC1}$ or simply $AC1$, generated by the associativity, commutativity and neutral element axioms for the "$\_,\_$" operator. The relation induced over $AC1$-equivalence classes is written $\mapsto_{\rho_g/AC1}$. Concretely, in the proof, the notion of rewriting modulo $AC1$ [PS81], denoted $\mapsto_{\rho_g,AC1}$, is used.

Figure 4.2 provides a graphical representation of some properties of a relation R modulo the congruence relation $\sim_{AC1}$. A formal definition can be found in [Ohl98].

A key property in the confluence proof is the compatibility of the reduction relation with respect to the equivalence on terms, that holds for any subset of rules of the ρ_{g}-calculus. This property ensures that the rewrite relation is particularly well-behaved *w.r.t.* the congruence relation $AC1$.

Clearly, according to the strategy *SharingStrat*, the order in which constraints are listed does not influence the applicability of substitution rules. Therefore compatibility for the ρ_{g}-calculus with strategy *SharingStrat* can be proved exactly as for in the unrestricted calculus.

Lemma 4.8 (Compatibility of ρ_g) *Compatibility with $AC1$ holds for any rule r of the ρ_{g}-calculus with strategy* SharingStrat.

$$\mapsto_{r,AC1} \cdot \sim_{AC1} \quad \subseteq \quad \sim_{AC1} \cdot \mapsto_{r,AC1}$$

Following the proof in [BBCK07], the evaluation rules of the ρ_g-calculus are split into two subsets for which confluence is first proved independently. Then, this intermediate result, together with a commutation lemma, is used for proving the confluence of the union of the two subsets. Here this proof is adapted to the ρ_g-calculus calculus with the strategy *SharingStrat*, under the same assumptions of linearity for patterns and constraints.

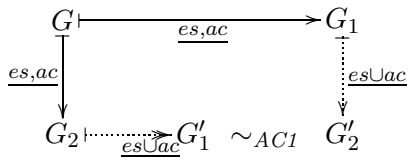
The first subset of rules, called τ, includes (ρ), (*propagate*), (*decompose*), (*solved*), (*garbage*) and (*black hole*), and the second one, called Σ, consists of the remaining rules, *i.e.* (*external sub*), (*acyclic sub*) and the (δ). Since only the latter set includes rules whose application is constrained by the strategy *SharingStrat*, only the proofs concerning the relation induced by this set should be adapted *w.r.t.* to the unrestricted version of the calculus. In what follows we detail the corresponding proofs, while the properties that can be inherited from the unrestricted calculus [BBCK07] are just stated.

Proposition 4.9 *The relation τ is confluent modulo AC1 under the strategy* SharingStrat.

The relation induced by the set of rules Σ is shown to be confluent adapting the *complete development method* defined for the λ-calculus: a terminating version of the relation (the development), denoted $\underline{\Sigma}$, can be defined and its properties are used for deducing the confluence of the original rewrite relation. Due to space constraints, the development relation cannot be defined in full details here. Roughly, a step $G \to_{\underline{\Sigma}} H$ of such relation consists of the complete development, with respect to Σ, of a set of redexes selected in the starting term G. The notation $\underline{\Sigma}$ arises from the fact that the selection is done by underlining the redexes.

First of all, notice that since the strategy *SharingStrat* affects the rewrite relation by restraining the application of the substitution rules, the relation $\underline{\Sigma}$ is clearly still normalising under this strategy. Hence, for proving its confluence, we simply need to verify its local confluence [Ohl98].

Lemma 4.10 *The relation $(\underline{es \cup ac})$ is locally confluent modulo AC1 under the strategy* SharingStrat*:*



Proof. By analysis of the critical pairs. If the terms duplicated by the substitutions are simply variables, then local confluence follows from the corresponding result for the ρ_g-calculus. A non trivial critical pair arises when one of the substitution steps occurs in the term duplicated by the other substitution. For example, consider the term $G = G_0\ [y = \underline{x}\ a, x = f(\underline{z}\ a) \wr b, z = a \twoheadrightarrow b]$ and the two (ac)-steps leading respectively to $G_1 = G_0\ [y = \underline{x}\ a, x = f((a \twoheadrightarrow b)\ a) \wr b, z = a \twoheadrightarrow b]$ and $G_2 = G_0\ [y = (f(\underline{z}\ a) \wr b)\ a, x = f(\underline{z}\ a) \wr b, z = a \twoheadrightarrow b]$. We can close the critical pair since there exist two reductions $G_1 \mapsto_{ac} G_3$ and $G_2 \mapsto\!\!\!\!\twoheadrightarrow_{ac} G_3$ such that

$G_3 = G_0\ [y = (f((a \twoheadrightarrow b)\ a) \wr b)\ a, x = f((a \twoheadrightarrow b)\ a) \wr b, z = a \twoheadrightarrow b].$

□

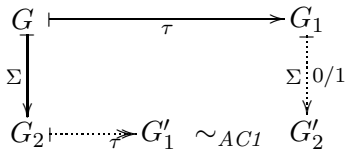
The same arguments as for the unrestricted version of the calculus can be used to show that the relations $(\underline{es \cup ac})$ and $(\underline{\delta})$ commute. Using this result and the compatibility of the two relations we obtain the confluence of the $\underline{\Sigma}$ relation and then of the Σ relation.

Proposition 4.11 *The relation Σ is confluent modulo AC1 under the strategy* SharingStrat.

Once having proved the confluence of the two rewrite relation independently, we prove general confluence of the $(\rho_g, AC1)$ relation by showing the (strong) commutation of the two subsets of rules [Ohl98].

Lemma 4.12 *The relations τ and Σ commute modulo AC1.*

Proof. Since the relations τ and Σ are compatible with $AC1$, it is enough to show strong commutation between the two relations instead of general commutation:



If the applied Σ-rule is the (δ) rule, the diagram can be closed as described in [BBCK07]. We are interested here in the cases where the applied Σ-rule is a substitution rule. We proceed by analysing the critical pairs. The diagram can always be closed under the strategy *SharingStrat*, since the τ-rules do not interfere with the creation of basic redexes. For example :

$$\begin{array}{ccc} P \ll (\mathsf{Ctx}[y]\ [y = H, E]) & \xrightarrow[p]{} & P \ll \mathsf{Ctx}[y], y = H, E \\ \Big\downarrow{\scriptstyle es} & & \Big\downarrow{\scriptstyle ac} \\ P \ll (\mathsf{Ctx}[H]\ [y = H, E]) & \xrightarrow[p]{} & P \ll \mathsf{Ctx}[H], y = H, E \end{array}$$

The basic redex $\mathsf{Ctx}[H]$ can be created before or after the propagation of the list of constraints. We can reason similarly for the application of the other τ-rules, like the (*decompose*) or the (*garbage*) rule (in this case we may have zero Σ steps for closing the diagram). □

Theorem 4.13 (Confluence of $\rho_{\mathsf{g}}, AC1$) *The rewrite relation $\rho_{\mathsf{g}}, AC1$ is confluent modulo AC1 under the strategy* SharingStrat.

Finally, the main theorem states the confluence of the $\rho_g/AC1$ relation, by deducing it from the confluence of the $(\rho_g, AC1)$ relation.

Theorem 4.14 (Confluence) *The linear ρ_{g}-calculus with the strategy* SharingStrat *is confluent modulo AC1.*

5 Conclusions

In this paper we have proposed a reduction strategy *SharingStrat* for the $\rho_{\mathtt{g}}$-calculus, an extension of the ρ-calculus able to deal with graph like structures. The strategy *SharingStrat* aims at maintaining the sharing information as long as possible in the $\rho_{\mathtt{g}}$-terms and is shown to be sound and complete (for normalising terms). Moreover, the $\rho_{\mathtt{g}}$-calculus with the strategy *SharingStrat* is shown to be confluent, under some restrictions of linearity on patterns.

There are several interesting directions for future research. We intend to investigate the issue of optimality for the reduction strategy, where the notion of "optimal" has to be formally defined, for example in terms of time, space or sharing conservation. In this case a natural reference to compare with would be the work on optimal reduction for lambda calculus. Another matter for future work is to model the rewrite strategy not at the meta level, but in the calculus itself. Taking inspiration from analogous work in the ρ-calculus [CKLW03], we would like to have rewrite rules as primal strategies and iterate rewritings on a $\rho_{\mathtt{g}}$-term adding a fix-point operator to the calculus. Moreover, to detect failures of rewrite rule application at some occurrences, we need also to define an exception handling mechanism.

References

[AK96] Z. M. Ariola and J. W. Klop. Equational term graph rewriting. *Foundamenta Informaticae*, 26(3-4):207–240, 1996.

[AK97] Z. M. Ariola and J. W. Klop. Lambda calculus with explicit recursion. *Information and Computation*, 139(2):154–233, 1997.

[BBCK05] C. Bertolissi, P. Baldan, H. Cirstea, and C. Kirchner. A rewriting calculus for cyclic higher-order term graphs. In M. Fernandez, editor, *2nd International Workshop on Term Graph Rewriting*, volume 127, pages 21–41, Roma, Italy, September 2005. Electronic Notes in Theoretical Computer Science.

[BBCK07] C. Bertolissi, P. Baldan, H. Cirstea, and C. Kirchner. A rewriting calculus for cyclic higher-order term graphs. To appear in *Mathematical Structures in Computer Science*, 2007.

[BCKL03] G. Barthe, H. Cirstea, C. Kirchner, and L. Liquori. Pure Patterns Type Systems. In *Proceedings of POPL'03: Principles of Programming Languages, New Orleans, USA*, volume 38, pages 250–261. ACM, 2003.

[Ber05] C. Bertolissi. *The graph rewriting calculus: properties and expressive capabilities.* Thèse de Doctorat d'Université, Institut National Polytechnique de Lorraine, Nancy, France, Octobre 2005.

[BvEG+87] H. P. Barendregt, M. C. J. D. van Eekelen, J. R. W. Glauert, J. R. Kennaway, M. J. Plasmeijer, and M. R. Sleep. Term graph rewriting. In *Proceedings of PARLE'87, Parallel Architectures and Languages Europe*, volume 259 of *Lecture Notes in Computer Science*, pages 141–158, Eindhoven, 1987. Springer-Verlag.

[CG99] A. Corradini and F. Gadducci. Rewriting on cyclic structures: Equivalence of operational and categorical descriptions. *Theoretical Informatics and Applications*, 33:467–493, 1999.

[CHW06] Horatiu Cirstea, Clement Houtmann, and Benjamin Wack. Distributive rho-calculus. In *6th International Workshop on Rewriting Logic and its Applications*, Electronic Notes in Theoretical Computer Science. Elsevier, 2006. to appear.

[CK01] H. Cirstea and C. Kirchner. The rewriting calculus — Part I *and* II. *Logic Journal of the Interest Group in Pure and Applied Logics*, 9(3):427–498, May 2001.

[CKLW03] H. Cirstea, C. Kirchner, L. Liquori, and B. Wack. Rewrite strategies in the rewriting calculus. In Bernhard Gramlich and Salvador Lucas, editors, *Third International Workshop on Reduction Strategies in Rewriting and Programming* , Valencia, Spain, June 2003. Electronic Notes in Theoretical Computer Science.

[CLW03] H. Cirstea, L. Liquori, and B. Wack. Rewriting calculus with fixpoints: Untyped and first-order systems. In Stefano Berardi, Mario Coppo, and Ferruccio Damian, editors, *Types for Proofs and Programs (TYPES)*, volume 3085 of *Lecture Notes in Computer Science*, pages 147–171, Torino (Italy), May 2003.

[Cor93] A. Corradini. Term rewriting in CT_Σ. In M.-C. Gaudel and J.-P. Jouannaud, editors, *Proceedings of TAPSOFT'93, Theory and Practice of Software Development—4th International Joint Conference CAAP/FASE*, pages 468–484. Springer, Berlin, Heidelberg, 1993.

[Ohl98] E. Ohlebusch. Church-Rosser Theorems for Abstract Reduction Modulo an Equivalence Relation. In T. Nipkow, editor, *Proceedings of the 9th International Conference on Rewriting Techniques and Applications (RTA-98)*, volume 1379 of *Lecture Notes in Computer Science*, pages 17–31. Springer, 1998.

[PJ87] S. Peyton-Jones. *The implementation of functional programming languages*. Prentice Hall, Inc., 1987.

[PS81] G. Peterson and M. E. Stickel. Complete sets of reductions for some equational theories. *Journal of the ACM*, 28:233–264, 1981.

[SPvE93] M. R. Sleep, M. J. Plasmeijer, and M. C. J. D. van Eekelen, editors. *Term graph rewriting: theory and practice*. Wiley, London, 1993.

[Wac05] B. Wack. *Typage et déduction dans le calcul de réécriture*. Thèse de doctorat, Université Henri Poincaré - Nancy I, October, 7- 2005.

Complete Laziness

François-Régis Sinot

Universidade do Porto (DCC & LIACC)
Rua do Campo Alegre 1021–1051
4169–007 Porto, Portugal

Abstract

Lazy evaluation (or call-by-need) is widely used and well understood, partly thanks to a clear operational semantics given by Launchbury. However, modern non-strict functional languages do not use plain call-by-need evaluation: they also use optimisations like fully lazy λ-lifting or partial evaluation. To ease reasoning, it would be nice to have all these features in a uniform setting. In this paper, we generalise Launchbury's semantics in order to capture "complete laziness", as coined by Holst and Gomard in 1991, which is slightly more than fully lazy sharing, and closer to on-the-fly needed partial evaluation. This gives a clear, formal and implementation-independent operational semantics to completely lazy evaluation, in a natural (or big-step) style similar to Launchbury's. Surprisingly, this requires sharing not only terms, but also contexts, a property which was thought to characterise optimal reduction.

1 Introduction

Lazy evaluation (also known as call-by-need) is an evaluation strategy for functional languages providing some notion of sharing. The idea behind lazy evaluation is intuitive: a subterm should be evaluated only if it is needed, and if so, it should be evaluated only once. Since its introduction by Wadsworth [23], there have been several efforts, on one hand to improve its concrete implementation, e.g. [19], and on the other, to improve its abstract formalisation: big-step operational semantics of call-by-need have been given independently in [15] and [20]; small-step presentations based on contexts have been given in [2,18]. While all these works have their own merits, Launchbury's natural semantics [15] certainly gives one of the clearest account of the process of lazy evaluation.

Yet, lazy evaluation captures only the sharing of *values*. For example, evaluation of the term $(\lambda f.fI(fI))(\lambda w.(\underline{II})\,w)$ where $I = \lambda x.x$ will reduce the underlined redex II twice, because the subterm $\lambda w.(II)\,w$ will be shared, then copied as a whole when necessary, since it is already a value (the redex II is under a λ-abstraction). This is indeed what happens in standard implementations of call-by-need [19,11].

This is not usually considered as a problem, because this term can also be transformed into $(\lambda f.fI(fI))((\lambda z.(\lambda w.z\,w))(II))$ in which the redex II will be shared by a lazy interpreter, and evaluated only once, because it is no longer under a λ-

abstraction. This transformation is called *fully lazy λ-lifting* and is used at compile-time in compilers for non-strict languages [11,19].

Implementations allowing to share this kind of redexes are called *fully lazy*. Wadsworth was the first to define this notion: he noticed that the redex II should not be copied since no occurrence of the bound variable w occurs in it [23]. But still, the resulting redex $I\,w$ will be evaluated twice by a fully lazy implementation, while its evaluation could have been shared using partial evaluation [12]. In other words, there is a notion of laziness, beyond full laziness, with the same sharing power as partial evaluation. This notion has been coined "complete laziness" in [10] (and later "ultimate sharing" in [1]), but seems to be otherwise unstudied, and in particular lacks a suitable operational semantics.

Moreover, some recent works [21,22] are likely to implement completely lazy evaluation, which is left as an open problem in [10], but there is no hope of proving (or even stating) this formally without a proper operational semantics. This present work thus also goes one step further in this direction.

In this paper, we define a clear and implementation-independent operational semantics for completely lazy evaluation. It is a natural (or big-step) semantics, in a style similar to Launchbury's for lazy evaluation. This is both a formal and effective definition of completely lazy evaluation, and a step towards a better understanding of sharing in the λ-calculus.

2 Launchbury's Semantics

We first briefly review Launchbury's semantics for lazy evaluation, as we will follow the same approach for completely lazy evaluation in Section 3. It is defined on expressions of a λ-calculus enriched with recursive *lets*. As pointed out by Launchbury, *lets* are useful in the input language as they allow to build explicitly cyclic structures. Without them, this would be more difficult and some sharing could be lost. This is in particular one of Launchbury's criticisms against the semantics of [20]. *Lets* are also useful in the intermediate language, as they play an important role in the definition of the semantics.

Launchbury splits the presentation of the semantics in two distinct stages: a static transformation into simpler expression (called *normalisation*), and a dynamic semantics defined only on normalised expressions.

2.1 Normalising terms

First, every expression e is transformed into an expression $\hat{e}$ in which all bound variables are renamed to completely fresh variables. This amounts to performing enough α-conversions, so that expressions respect Barendregt's convention. Expressions are then normalised to obey the following syntax, where arguments of applications are restricted to variables in order to share arguments with a *let* construct.

$$\begin{aligned} t, u &::= x \mid \lambda x.t \mid t\ x \mid \mathit{let}\ x_1 = u_1, \ldots, x_n = u_n\ \mathit{in}\ t \\ v, w &::= \lambda x.t \end{aligned}$$

Values $v, w, \ldots$ are not used in this section, but will allow us to characterise

precisely the result of evaluation in Section 2.2. Launchbury's semantics is only defined on closed terms (or more precisely, on closed pairs of an environment and a term), and the dynamic rules of Section 2.2 preserve closedness (that is, if the left-hand side of the conclusion of a rule is closed, the left-hand side of all the premises of this rule are closed as well). That is why, in Launchbury's semantics, values are always λ-abstractions (and never of the form $x\,t_1 \dots t_n$).

The restriction on application means that arguments are already explicitly named closures, ready to be shared. This normalisation stage thus already contributes to capture sharing. It is defined precisely by the following function $(\cdot)^*$ from standard, unconstrained λ-terms with recursive *lets* to terms t, u obeying the syntax above.

$$
\begin{aligned}
x^* &= x \\
(\lambda x.t)^* &= \lambda x.(t^*) \\
(t\ u)^* &= \begin{cases} (t^*)\ u & \text{if } u \text{ is a variable,} \\ let\ x = u^*\ in\ (t^*)\ x & \text{otherwise } (x \text{ is a fresh variable).} \end{cases} \\
(let\ x_1 = u_1, \dots, x_n = u_n\ in\ t)^* &= let\ x_1 = u_1^*, \dots, x_n = u_n^*\ in\ t^*
\end{aligned}
$$

2.2 Dynamic semantics

The semantics is not defined on terms alone; some data must be added to actually represent sharing. Launchbury's choice is to use heaps or environments (written Γ, Δ, Θ), which are defined either as finite mappings from variables to terms (or equivalently as unordered association lists binding distinct variable names to values).

Evaluation is only defined on closed pairs $\Gamma : t$, meaning that the free variables in t have to be bound in the environment Γ. Evaluation judgements are of the form $\Gamma : t \Downarrow_L \Delta : v$, to be read "the term t in the environment Γ reduces to the value v together with the new environment Δ", and are defined by the set of deduction rules in Figure 1.

The only rule where sharing is indeed captured is Var_L. To evaluate a variable x, the heap must contain a binding $x \mapsto t$, otherwise x has a direct dependency on itself and evaluation should fail. Then t is evaluated to a value v, in a heap where the binding for x has been removed, in order to avoid direct dependencies. The binding for x in the environment is then updated with the value v, in order to avoid a possible recomputation if x is needed several times, and the evaluation returns $\hat{v}$, i.e. v with fresh names for all its bound variables. It is the only rule where renaming occurs and this is sufficient to avoid all unwanted name capture [15]. An example is given in Figure 4 (A_n is defined in Section 4).

3 Modelling Complete Laziness

In lazy evaluation, only closed terms are shared; e.g., in $(\lambda f.fI(fI))(\lambda w.(II)\,w)$, lazy evaluation will share $(\lambda w.(II)\,w)$, but will reduce II twice. To obtain complete laziness (and reduce II only once), we need to share the body $(II)\,w$ as well. In other words, to realise complete laziness, open terms need to be shared as well. More precisely, in an abstraction $\lambda x.t$, we do not want to share t as a whole, because, when x would be instantiated, the shared representation of t would be updated,

$$\frac{}{\Gamma : \lambda x.t \Downarrow_L \Gamma : \lambda x.t}\ Lam_L$$

$$\frac{\Gamma : t \Downarrow_L \Delta : \lambda y.t' \quad \Delta : t'\{y := x\} \Downarrow_L \Theta : v}{\Gamma : t\ x \Downarrow_L \Theta : v}\ App_L$$

$$\frac{\Gamma : t \Downarrow_L \Delta : v}{(\Gamma, x \mapsto t) : x \Downarrow_L (\Delta, x \mapsto v) : \hat{v}}\ Var_L$$

$$\frac{(\Gamma, x_1 \mapsto u_1, \ldots, x_n \mapsto u_n) : t \Downarrow_L \Delta : v}{\Gamma : let\ x_1 = u_1, \ldots, x_n = u_n\ in\ t \Downarrow_L \Delta : v}\ Let_L$$

Fig. 1. Launchbury's semantics

thus preventing x from being instantiated by another argument. In fact, we exactly want to share the part of t that does not depend on x. In other words, if we write $t = C[x]$ where x does not appear in the context $C[\,]$ (possibly with several instances of the same hole), then $C[\,]$ is exactly what should be shared. In the example above, what should be shared is indeed $(II)\,[\,]$. The comparison with contexts is helpful to emphasise that the free variables are not part of what should be shared, but is otherwise misleading: there may be several occurrences of the same free variable (hence the notion of hole is not adequate), and normal capture-avoiding term-substitution should be used (instead of context-substitution). It is really more adequate to say that we need to share open terms.

We thus need variables to represent open subterms. Since we may have to deal with several distinguished variables in these terms, it is just as simple to use the concept and notation of metavariables taken from Combinatory Reduction Systems [13]. In other words, we will write for instance $Z(x, y)$ (and we will call it a *metavariable*) for a variable representing an open term in which x and y denote the free variables. Just any term t can be substituted for $Z(x, y)$, but if x and y appear in t (perhaps even several times), then the rules will be able to treat them in a special way. It should also be noted that α-equivalence is extended in the obvious way, with for instance $\lambda x.Z(x) =_\alpha \lambda y.Z(y)$. There is no need for α-equivalence on metavariables.

We follow Launchbury's approach and present the semantics in two phases: a static transformation into simpler expression (again called *normalisation*), and a dynamic semantics for normalised expressions.

3.1 Normalising terms

The normalisation stage has two purposes. The first one is to avoid name capture, by renaming all (λ- and *let*-) bound variables to fresh variables. The second one is to name explicitly with a *let*-construct any subterm that may need to be shared. For lazy evaluation, it is enough to do this for arguments in applications. Here, for completely lazy evaluation, we also need to do this for bodies of abstractions. We will thus assume that expressions $t, u, \ldots$ belong to a new set, defined as follows, where we write $Z(\vec{x})$ for $Z(x_1, \ldots, x_n)$. We also define *values* $v, w, \ldots$ in this context,

which will be used in Section 3.2 to characterise precisely the result of evaluation.

$$\begin{aligned} b &::= x \mid Z(\vec{x}) \\ t, u &::= b \mid \lambda x.b \mid t\ b \mid let\ b_1 = u_1, \ldots, b_n = u_n\ in\ t \\ v, w &::= \lambda x.b \mid x\ b_1\ \ldots\ b_n \end{aligned}$$

Similarly to Launchbury's semantics, the completely lazy semantics will also be defined only on closed terms. However, in the course of evaluation, we may have to evaluate an open term (this happens in the first premise of rule *MVar* in Figure 2, which will be explained later), and this evaluated open term will be used to update the binding of a metavariable. In other words, it is important to allow open terms of the form $x\,b_1 \ldots b_n$ as values, contrarily to Section 2.1. However, terms of the form $Z(\vec{x})\,b_1 \ldots b_n$ are not values, for the same reason as $x\,t_1 \ldots t_n$ was not a value in Section 2.1: because the completely lazy semantics is only defined on pairs environments/terms which are meta-closed, i.e. in which all metavariables are bound (either by *lets* or by the environment), and this property is preserved by the rules. The situation is really reminiscent of what happens in Combinatory Reduction Systems, where metavariables essentially play the same role as variables in first-order systems.

Standard λ-expressions with *lets* can be translated into this form by the following normalisation function, which takes an auxiliary list of variables as an extra argument (written as a subscript). The semantics is only defined on closed terms and this list should initially be empty. The normalisation function takes terms from an unconstrained λ-calculus with recursive *lets* and without metavariables to terms t, u obeying the syntax above.

$$\begin{aligned} (x)^*_{\vec{z}} &= x \\ (\lambda x.t)^*_{\vec{z}} &= \begin{cases} \lambda x.t & \text{if } t \text{ is a variable,} \\ let\ Z(\vec{z}, x) = (t)^*_{\vec{z},x}\ in\ \lambda y.Z(\vec{z}, y) & \text{otherwise.} \end{cases} \\ (t\ u)^*_{\vec{z}} &= \begin{cases} (t)^*_{\vec{z}}\ u & \text{if } u \text{ is a variable,} \\ let\ Z(\vec{z}) = (u)^*_{\vec{z}}\ in\ (t)^*_{\vec{z}}\ Z(\vec{z}) & \text{otherwise.} \end{cases} \\ (let\ x_1 &= u_1, \ldots, x_n = u_n\ in\ t)^*_{\vec{z}} \\ &= let\ Z_1(\vec{z}) = (u_1)^*_{\vec{z}}, \ldots, Z_n(\vec{z}) = (u_n)^*_{\vec{z}}, \\ &\qquad x_1 = Z_1(\vec{z}), \ldots, x_n = Z_n(\vec{z})\ in\ (t)^*_{\vec{z}} \end{aligned}$$

All variable and metavariable names created by the function $(\cdot)^*$ are assumed to be fresh. The purpose of the auxiliary list $\vec{z}$ is to remember which variables are bound by outer λ's (and not by *let* constructs), because these are exactly the variables that could be instantiated by different terms in different copies. The normalisation function seems to introduce many indirections, but this is necessary in order to preserve sharing. For instance, in the case for *let* expressions, a new binding with a metavariable $Z_i(\vec{z})$ is introduced to share the evaluation of u_i when the variables $\vec{z}$ are free (that is, when it is considered as an open term), but it is still necessary to have a binding for x_i (which may appear in t or any u_j), in order to share the evaluation of u_i when the variables of $\vec{z}$ are bound to some expressions. When $\vec{z}$ is

empty, nothing special happens, although we may want to simply write Z instead of $Z()$. It is not safe in general to replace such metavariables by normal variables. This is discussed on an example in Section 4.3. The procedure could be refined to save some indirections and minimise the number of variables bound by the new metavariables, however the present formulation suffices for our purpose.

3.2 Dynamic semantics

As in Launchbury's semantics, we use heaps to model sharing. Now heaps specify bindings from distinct variable or metavariable names to terms. Again, evaluation is only defined for meta-closed (see above) pairs $\Gamma : t$ in which all bound variables are distinct, and it is specified by the deduction rules in Figure 2. We observe that the result of evaluation is a pair $\Delta : v$ where v is a value (a term in weak head normal form, i.e. of the form $\lambda x.b$ or $x\, b_1 \dots b_n$).

The first four rules are exactly those of Launchbury's semantics. The *MVar* rule is called when it is needed to evaluate a shared, possibly open subterm. Completely lazy sharing is obtained here: t is evaluated, and $Z(\vec{x})$ is updated with the result. There would be a risk that t would be evaluated too much, if the variables in $\vec{x}$ were instantiated. This does not happen, because of the normalisation procedure, which ensures that variables bound by λ-abstractions are fresh and do not appear in *let*-bindings for metavariables (this property is preserved during evaluation, cf. Proposition 5.1). Then, the evaluation goes on with the right variable names, thanks to the substitution of x_1 by y_1, ..., x_n by y_n, written $\{\vec{x} := \vec{y}\}$, in $\hat{v}$ (that is, v with all its bound variables renamed to fresh variables). However, $Z(\vec{x})$ should not be further updated, since the variables in $\vec{y}$ are likely to be bound in the environment. In this second phase, we keep the binding for $Z(\vec{x})$ in the environment so that this shared open term can be used with different instantiations of its free variables. The last two rules just deal with open terms in a natural way. The same rules would make sense in Launchbury's semantics to deal with open terms or constants.

Evaluation may fail in rule *MVar* only, if there is no binding for $Z(\vec{x})$ in the environment. This happens for example if $Z(\vec{x})$ has a direct dependency on itself. This allows us to detect some non-terminating programs (but of course not all of them). The same happens in Launchbury's semantics in the *Var* rule (here, for a variable, Var_1 or Var_2 is always applicable).

4 Examples

In this section, we illustrate the behaviour of the semantics in order to give some concrete evidence that it indeed captures completely lazy sharing. To make our point more concrete, we assume given additional rules for the evaluation of arithmetical expressions, as found in [15], for instance:

$$\frac{}{\Gamma : n \Downarrow \Gamma : n} \qquad \frac{\Gamma : t_1 \Downarrow \Delta : n_1 \qquad \Delta : t_2 \Downarrow \Theta : n_2}{\Gamma : t_1 + t_2 \Downarrow \Theta : n_1 + n_2}$$

We will also omit some inessential details, for instance some superfluous bindings, which could be avoided with a more clever normalisation procedure.

$$\frac{}{\Gamma : \lambda x.b \Downarrow \Gamma : \lambda x.b}\ Lam$$

$$\frac{\Gamma : t \Downarrow \Delta : \lambda y.b' \quad (\Delta, y \mapsto b) : b' \Downarrow \Theta : v}{\Gamma : t\ b \Downarrow \Theta : v}\ App_1$$

$$\frac{\Gamma : t \Downarrow \Delta : v}{(\Gamma, x \mapsto t) : x \Downarrow (\Delta, x \mapsto v) : \hat{v}}\ Var_1$$

$$\frac{(\Gamma, b_1 \mapsto u_1, \ldots, b_n \mapsto u_n) : t \Downarrow \Delta : v}{\Gamma : let\ b_1 = u_1, \ldots, b_n = u_n\ in\ t \Downarrow \Delta : v}\ Let$$

$$\frac{\Gamma : t \Downarrow \Delta : v \quad (\Delta, Z(\vec{x}) \mapsto v) : \hat{v}\{\vec{x} := \vec{y}\} \Downarrow \Theta : w}{(\Gamma, Z(\vec{x}) \mapsto t) : Z(\vec{y}) \Downarrow \Theta : w}\ MVar$$

$$\frac{x \notin \Gamma}{\Gamma : x \Downarrow \Gamma : x}\ Var_2$$

$$\frac{\Gamma : t \Downarrow \Delta : x\ b_1 \ldots b_n}{\Gamma : t\ b \Downarrow \Delta : x\ b_1 \ldots b_n\ b}\ App_2$$

Fig. 2. Completely lazy semantics

Following the tradition initiated by [15], we lay proofs out vertically, so as to stress the sequential nature of evaluation. If $\Gamma : t \Downarrow \Delta : v$, we write:

$\Gamma : t$

[*a sub-proof*

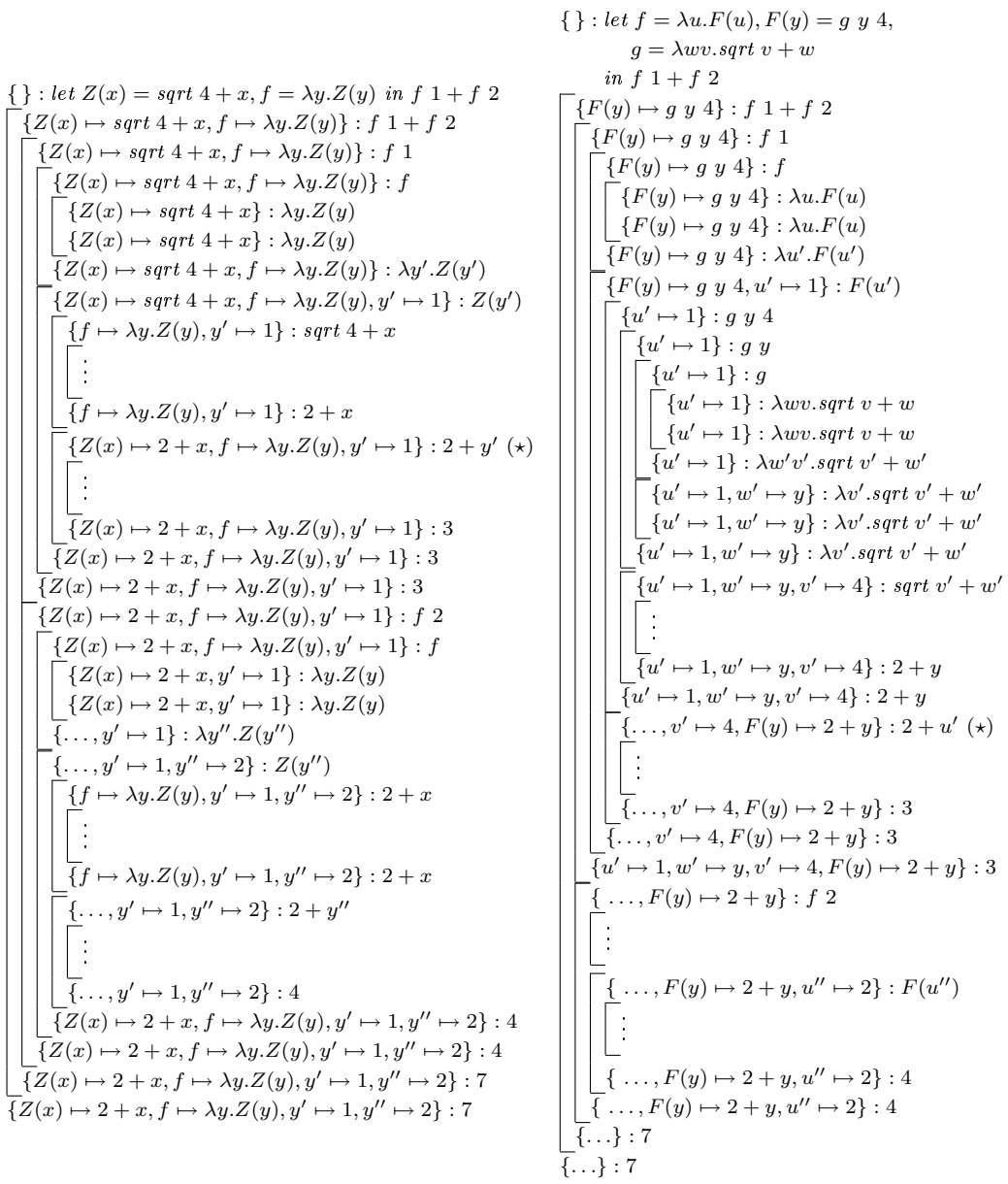
[*another sub-proof*

$\Delta : v$

4.1 Simple examples

Let us begin with an example taken from [19]: $let\ f = \lambda x.sqrt\,4 + x\ in\ f\ 1 + f\ 2$. This first example illustrates the sharing of a constant expression inside a λ-abstraction, which would be achieved by fully lazy λ-lifting, but not by standard lazy evaluation. For simplicity, let us omit some indirections and assume that it is normalised as: $let\ Z(x) = sqrt\ 4 + x, f = \lambda y.Z(y)\ in\ f\ 1 + f\ 2$. The evaluation derivation of this example is sketched on Figure 3(a). We can observe on line ($\star$) that $sqrt\ 4$ is indeed evaluated only once, and that $Z(x)$ is indeed updated with $2 + x$ (in particular, we evaluate $sqrt\ 4 + x$ first, rather than $sqrt\ 4 + y'$).

However, such constant subexpressions may also be created dynamically, as in the following program, taken from [19] as well (the translation is again simplified).



(a) Sharing of a constant subexpression

(b) Partial application

Fig. 3. Simple examples

$$\left(\begin{array}{l} let\ f = \lambda y.g\ y\ 4, \\ \quad g = \lambda yx.sqrt\ x + y \\ \quad in\ f\ 1 + f\ 2 \end{array}\right)^* \quad = \quad \begin{array}{l} let\ f = \lambda u.F(u), F(y) = g\ y\ 4, \\ \quad g = \lambda wv.sqrt\ v + w \\ \quad in\ f\ 1 + f\ 2 \end{array}$$

In the evaluation, the bindings for f and g will not be modified, since they are already bound to values, we thus omit them for conciseness: all environments implicitly contain $f \mapsto \lambda u.F(u)$ and/or $g \mapsto \lambda wv.sqrt\ v + w$. In this example again,

shown in Figure 3(b), *sqrt* 4 is evaluated only once, even though this redex is only generated on the fly by a partial application. This can be seen from the fact that $F(y)$ is updated with $2+y$ on line ($\star$). This example is further discussed in Section 6.

4.2 Efficiency

We can also give a striking example, adapted from [9,8,1], to demonstrate that completely lazy reduction can perform exponentially better than lazy evaluation. Consider the family of terms:

$$A_0 = \lambda x.I$$
$$A_n = (\lambda h.(\lambda w.w\,h\,(w\,w))\,A_{n-1}) \equiv let\ Z(h) = (\lambda w.w\,h\,(w\,w))\,A_{n-1}\ in\ \lambda h'.Z(h')$$

A_n has exactly one redex $(\lambda w.\ldots)\,A_{n-1}$, which is under a λ-abstraction, hence will not be shared by lazy evaluation. Consequently, evaluation of $A_n\,I$ using call-by-need requires a number of steps in $O(2^n)$ [9,8]. In Launchbury's semantics, this can be seen on the evaluation sketch in Figure 4, where $T(n)$ denotes the number of steps necessary to evaluate $A_n\,x$ (this is indeed independent from x). Overall, $T(n) = O(2^n)$ with standard lazy evaluation. The A_i's are shared, but no significant update will ever happen since they already are weak head normal forms.

Now, with the completely lazy semantics, reduction will proceed as shown in Figure 5. The $O(T(n-1))$ first steps in this example are similar to the evaluation using call-by-need, except that not only $w, w', \ldots$ are updated, but also $Z(h), Z'(h), \ldots$ corresponding to the body under the outermost λ in $w, w', \ldots$ Then, in the second phase, almost no computation has to be performed since $Z'(h)$ is already bound to the identity (independently of h). Completely lazy evaluation of A_n is linear in n.

This example shows that, although some bookkeeping (indirections essentially) is added, completely lazy evaluation may be exponentially better than lazy evaluation, which is a very strong statement. As a matter of fact, the same improvement can be obtained by fully lazy λ-lifting on this example, but Section 6 will make clear that complete laziness has strictly more "sharing power" than full laziness. Note that all steps are taken into account: the bookkeeping due to the indirections is linear in n in this example. The exact details of implementations are fortunately not part of the semantics, but this means that however bad the implementation is, it will still perform better than any cutting-edge lazy interpreter on certain terms. This contrasts with optimal reduction, where the cost of bookkeeping ruins the benefits of optimality [16].

This means that completely lazy evaluation, hence the semantics we are putting forward, should be considered as a promising basis for an implementation: it achieves much better sharing than call-by-need, yet does not fall into the well-known problems of optimal reduction.

4.3 Recursion

Finally, with respect to recursion, the situation is very similar to that in Launchbury's semantics. For instance, $let\ x = x\ in\ x$ is normalised to $let\ Z = x, x = Z\ in\ x$. Evaluation of this programs fails as shown on Figure 6. This illustrates why there

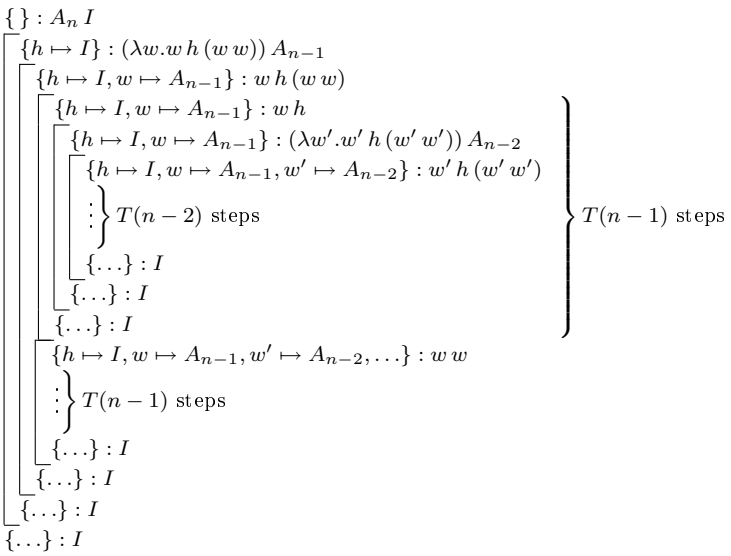


Fig. 4. Call-by-need evaluation of $A_n\, I$

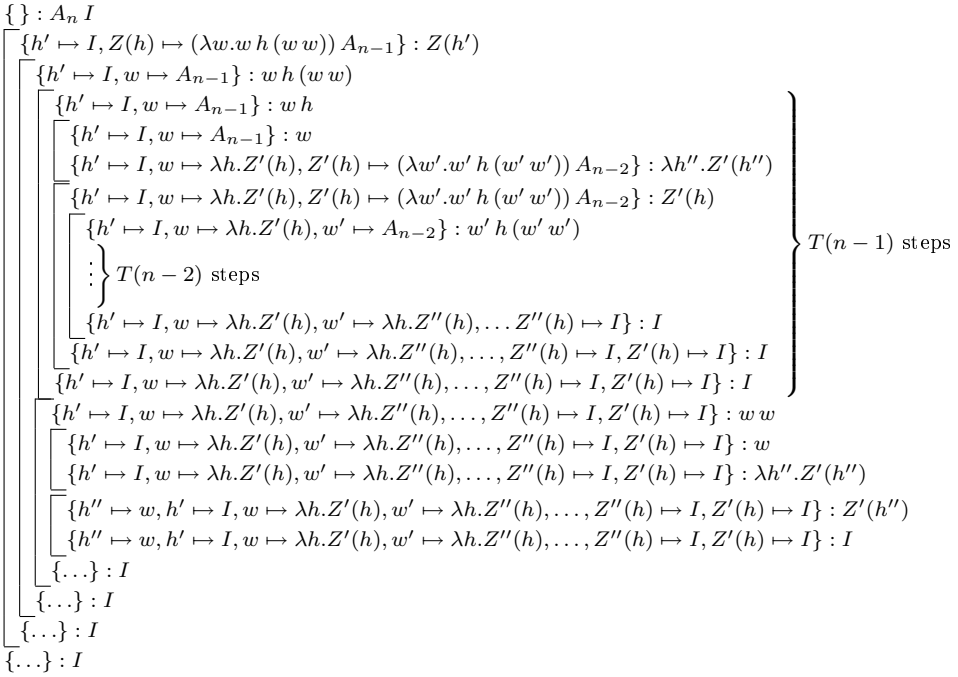


Fig. 5. Completely lazy evaluation of $A_n\, I$

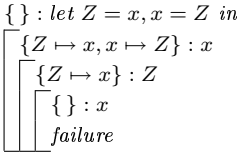


Fig. 6. Recursion with a direct dependency

is an extra indirection compared to the same program in Launchbury's framework: evaluation should not fail on a variable (because in completely lazy evaluation we need to perform reductions on open terms); it may only fail on a metavariable.

If we directly feed this example, without $(\cdot)^*$-translation, into the completely lazy semantics, we obtain: $\{\} : let\ x = x\ in\ x \Downarrow \{x \mapsto x\} : x$. In other words, we obtain a meaningless value, whereas the right behaviour is to fail. This illustrates that it is unsafe in general to replace metavariables (even without arguments) by normal variables. The converse is also unsafe: imagine we want to normalise the term $let\, x = \lambda y.x\ in\ x$ by replacing the *let*-bound variable x by a metavariable. The problem is that x appears both in a context where it is a closed term, and could be represented by Z, and in a context where it is potentially open, and should be represented by $Z(y)$.

5 Properties

5.1 Well-formedness

The first important property to check is that the semantics is indeed well defined. Since it is defined only on terms of a particular form, as produced by the normalisation procedure of Section 3.1, we should check that the result of evaluation has the correct form as well. The property that arguments of applications and bodies of abstractions are variables or metavariables is clearly preserved, since we only ever substitute variables for variables. The naming property is also preserved, as we will now show.

Following [15], we say that $\Gamma : t$ is *distinctly named* if all bound variables and metavariables are distinct. There are three standard types of binding: by a *let* construct, by a λ-abstraction, by a top-level binding in the heap. However, there is a last type of binding here: if $Z(\vec{x}) \mapsto t$ is a binding (for Z) in Γ, we also consider that it is a binding for the variables in $\vec{x}$. In particular, it is crucial that these variables are distinct from other bound variables in rule *MVar*.

Proposition 5.1 *If $\Gamma : t \Downarrow \Delta : v$ and $\Gamma : t$ is distinctly named, then every heap/term pair in the evaluation proof tree is also distinctly named.*

Proof. In general, the rules preserve bound variables. Var_1 and *MVar* copy a term, which may contain binders, but, one of the copies is renamed with fresh variables.□

In other words, it is sufficient to perform α-conversion in rules Var_1 and *MVar* alone to keep all bound variables distinct. In the remainder of this paper, pairs $\Gamma : t$ are always assumed to be distinctly named.

5.2 Correctness

Now that we know that the semantics does nothing wrong syntactically, we should also prove that it does nothing wrong semantically; in other words that evaluation preserves the denotational semantics of terms.

We define a readback function $(\cdot)^\circ$ from pairs $\Gamma : t$ to λ-terms (in fact, to potentially infinite λ-terms in case of cycles, but this is not really important) that

removes the shared variables and metavariables. For every binding $Z(\vec{x}) \mapsto u$ or *let* $Z(\vec{x}) = u$, the readback substitutes every metavariable $Z(\vec{y})$ by $\hat{u}\{\vec{x} := \vec{y}\}$, and then removes the binding for $Z(\vec{x})$, and similarly for the bindings for variables. This is possible thanks to the distinct naming.

Lemma 5.2 (i) *If* $\Gamma : t \Downarrow \Delta : u$, *then* $(\Gamma : t)^\circ \to_\beta^* (\Delta : u)^\circ$.

(ii) *When a* β*-reduction is performed during evaluation, (a copy of) the corresponding redex after readback is the leftmost outermost.*

Proof sketch. The rule App_1 is the only one where β-reduction is performed. The other rules do not have any effect after readback, in the sense that, for every rule (except *MVar*), the readback of the left-hand side of each premise is exactly the readback of the left-hand side of the conclusion. For *MVar*, $((\Gamma, Z(\vec{x}) \mapsto t) : Z(\vec{y}))^\circ = ((\Gamma, Z(\vec{x}) \mapsto t) : t\{\vec{x} := \vec{y}\})^\circ$ and since the variables in $\vec{x}$ are not bound in the environment (thanks to Proposition 5.1), the redexes in the readback of the left-hand side of both premises of the *MVar* rule are already present in the left hand-side of its conclusion. Now for rule App_1, let us take the notations of Figure 2. The first premise of rule App_1 focuses on the left subterm of an application while outermost β-reduction is performed in the second one: $(\Delta : (\lambda y.b')\, b)^\circ \to_\beta ((\Delta, y \mapsto b) : b')^\circ$.□

The previous lemma gives an idea of what happens during evaluation. In particular evaluation will always terminate on terms which have a weak head normal form. However the semantics does not exactly coincide with call-by-name ($\Downarrow_{CBN}$): a redex shared in our semantics may correspond to two different β-redexes, one evaluated by call-by-name, and the other not. In a more realistic functional languages with types and constants, *programs* are closed terms of base type (e.g. integers). The semantics coincide on these "basic observables':

Theorem 5.3 *If* t *is a program, then* $\Gamma : t \Downarrow \Delta : u$ *iff* $(\Gamma : t)^\circ \Downarrow_{CBN} (\Delta : u)^\circ$.

5.3 Sharing

Now that we know that the operational semantics given in Section 3 is correct with respect to its result, we should also give some evidence that it captures the sharing expected from complete laziness, which is defined in [10, Section 3.1] as follows:

Definition 5.4 An evaluation is completely lazy if all needed redexes are evaluated exactly once.

This sounds very much like optimal reduction, but it is weaker: optimal reduction also requires that *potential* redexes [17,14] are evaluated at most once. For instance, in the term $(\lambda x.x\, I)(\lambda y.\Delta\, (y\, I))$, the subterm $y\, I$ is not an actual redex, but it is a potential one since it may (and will) become an actual redex after substitution of y by I. We think that most of the conceptual and practical difficulty of optimal reduction comes from the requirement to share such subterms, which justifies the interest of complete laziness as a framework with as much sharing as possible, but excluding potential redexes. This is discussed further in Section 6.

Theorem 5.5 *Let* r *be a* β*-redex in* t. *Then in the derivation of* $\Gamma : t \Downarrow \Delta : v$, r *is reduced at most once.*

Proof sketch. The normalisation binds every non-trivial subexpression to a meta-variable. There is thus a subterm of t of the form $let\ Z(\vec{x}) = r^*\ in\ t'$. If r is reduced, rules *Let* and *MVar* must have been used, and $Z(\vec{x})$ is indeed updated with a value where r has been fired. No occurrence of r thus remains in the expression, hence r cannot be reduced more than once. □

The proposed semantics thus captures completely lazy sharing, in a more direct and operational way than in [10], where complete laziness is formalised as a meta-interpreter implemented in a fully lazy language.

6 Related Work

In the λ-calculus, there is a tension between reduction of the leftmost outermost redex (which is the only normalising choice in general), and reduction of other redexes, which may endanger termination, but may also lead to shorter reduction paths. In this last family of strategies, reduction of the outermost argument (call-by-value) is the most traditional, but some have also studied the impact of performing certain reductions under λ-abstractions [8,6]. The situation is nicely summed up in [7]:

> There is evidently a subtle interplay among the issues of efficiency, normalizability, and redex sharing. The quandary is then to find a way to edge closer to the brink of optimality without plunging into the abyss of non-normalizability.

This apparent tension can be resolved by sharing mechanisms: call-by-need resolves the tension between call-by-name and call-by-value by providing a way to share the evaluation of arguments. The framework we propose here generalises the approach, and resolves the tension between call-by-name and strategies which may reduce under λ's. We thus feel that this present work is a step forward in realising Field's programme.

There have been some works concerned with formal ways to express more than usually lazy reduction. One notable attempt is [2, Section 6], where a fully lazy calculus is given. This calculus can be viewed as a small step semantics for fully lazy evaluation, where reduction is restricted to some cleverly designed classes of contexts. However, the semantics performs on-the-fly λ-lifting, with one of the axioms involving costly conditions about maximal free expressions. It thus seems reasonable to say that this semantics is not effective, unless more details are given about how to implement these conditions at a reasonable cost. In contrast, our semantics does not perform any λ-lifting, it just has the right notion of sharing.

Fully lazy evaluation shares only the so-called maximal free expressions (MFE). This leads to cumbersome situations, as pointed out in [19, pages 398–400]. For instance, in the program $let\ g = \lambda y.\lambda x.y + sqrt\ x, f = \lambda y.g\ y\ 4\ in\ (f\ 1) + (f\ 2)$, the computation of $sqrt\ 4$ is performed twice, because $sqrt\ x$ is not an MFE of any λ-expression. This problem can be avoided with a different ordering of the parameters of g, but there are terms in which no ordering of the parameters is right. For instance, if the binding for g was in fact $g = \lambda x.\lambda y.sqrt\ x + sqrt\ y$, then some sharing will be lost with any order of the arguments. We think that this should be taken as a hint that full laziness is a too syntactic notion.

Our semantics indeed allows to share expressions of this kind, as demonstrated by the second example of Section 4, and thus captures completely, rather than fully, lazy evaluation. In the case $g = \lambda x.\lambda y.sqrt\ x + sqrt\ y$, our semantics would share a partial application indifferently on the first or the second argument of g.

We do believe that complete laziness is the rational way to capture the spirit of full laziness, abstracted away from syntactical consideration. Moreover, some implementations [21,22] are likely to follow our semantics more faithfully than fully lazy evaluation. In any case, the present work provides a formal tool to reason more precisely about fine issues concerning sharing, which was missing until now.

Another theme highly related to this present work is of course optimality theory, defined in [17] and implemented in [14]. In the introduction of [3], one may read:

> Lamping's breakthrough was a technique to share contexts [...]

This is of course true of optimal reduction, but what we learn here is that it is also true already for completely lazy reduction, which comes as a surprise. In other words, optimal reduction needs yet something more than the ability to share contexts. A simple example to show that the present semantics is not optimal is the term $(\lambda x.x\,I)(\lambda y.\Delta\,(y\,I))$ where $I = \lambda w.w$ and $\Delta = \lambda z.z\,z$. The semantics will perform the reduction $\Delta\,(y\,I) \to (y\,I)\,(y\,I)$, while the optimal choice is to share the potential redex $y\,I$ and reduce it only once when y is instantiated. This present work thus also paves the way to a better understanding of optimal reduction.

7 Conclusion

In this paper we have presented a natural semantics to model completely lazy evaluation. In contrast with Launchbury's work, this is not just a formalisation of a well-known and commonly implemented evaluation strategy. It is rather one of the first attempts to effectively define completely lazy evaluation.

The semantics is not meant to provide a direct specification for an abstract machine, but rather to be a general framework to reason about laziness and study various implementations. Since the framework is very simple compared to more concrete ones, it is also a good basis to study different extensions and properties.

Besides a better understanding of the theoretical issues of sharing and efficiency in functional, and more generally declarative programming languages, this work aims at being used as a foundational basis for implementations. Of course, the legitimacy of (various degrees of) laziness has been decreasing along the years [16,4] and it may seem that our work is primarily of theoretical interest. We do not believe this.

First, laziness is not always useless and there are techniques to combine the advantages of strictness and laziness, such as static analyses [5] and optimistic evaluation [4]. There is no reason to think that these techniques cannot be adapted to our framework. Moreover, laziness may not be useless in all declarative languages, in particular in proof assistants, where proof terms are built interactively and may have a very unusual and intricate shape, for which highly lazy strategies may be well-suited. We believe that the emergence of these new paradigms, with their specific problems, is the occasion to take a fresh look at the theory and practice of the implementation of programming languages.

References

[1] Amtoft, T., "Sharing of Computations," Ph.D. thesis, University of Aarhus (1993).

[2] Ariola, Z. M. and M. Felleisen, *The call-by-need lambda calculus*, Journal of Functional Programming **7** (1997), pp. 265–301.

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

[4] Ennals, R. and S. Peyton Jones, *Optimistic evaluation: an adaptive evaluation strategy for non-strict programs*, in: *Proceedings of International Conference on Functional Programming (ICFP'03)*, ACM Sigplan Notices **38, 9** (2003), pp. 287–298.

[5] Faxén, K.-F., *Cheap eagerness: speculative evaluation in a lazy functional language.*, in: *Proceedings of International Conference on Functional Programming (ICFP'00)*, ACM Sigplan Notices **35, 9** (2000), pp. 150–161.

[6] Fernández, M., I. Mackie and F.-R. Sinot, *Closed reduction: Explicit substitutions without α-conversion*, Mathematical Structures in Computer Science **15** (2005), pp. 343–381.

[7] Field, J., *On laziness and optimality in lambda interpreters: Tools for specification and analysis*, in: *Proceedings of Principles of Programming Languages (POPL'90)* (1990), pp. 1–15.

[8] Fradet, P., *Compilation of head and strong reduction*, in: D. Sannella, editor, *5th European Symposium on Programming*, Lecture Notes in Computer Science **788** (1994), pp. 211–224.

[9] Frandsen, G. S. and C. Sturtivant, *What is an efficient implementation of the λ-calculus ?*, in: J. Hughes, editor, *Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science **523** (1991), pp. 289–312.

[10] Holst, C. K. and C. K. Gomard, *Partial evaluation is fuller laziness*, Sigplan Notices **26** (1991), pp. 223–233.

[11] Hughes, R. J. M., "The Design and Implementation of Programming Languages," Ph.D. thesis, Programming Research Group, Oxford University (1983).

[12] Jones, N. D., P. Sestoft and H. Søndergaard, *An experiment in partial evaluation: The generation of a compiler generator*, in: *Rewriting Techniques and Applications*, Lecture Notes in Computer Science **202** (1985), pp. 125–140.

[13] Klop, J. W., "Combinatory Reduction Systems," Mathematical centre tracts, Centre for Mathematics and Computer Science, Amsterdam (1980).

[14] Lamping, J., *An algorithm for optimal lambda calculus reduction*, in: *Proceedings of the 17th ACM Symposium on Principles of Programming Languages (POPL'90)* (1990), pp. 16–30.

[15] Launchbury, J., *A natural semantics for lazy evaluation*, in: *Proceedings of Principles of Programming Languages (POPL'93)*, 1993, pp. 144–154.

[16] Lawall, J. L. and H. G. Mairson, *Optimality and inefficiency: What isn't a cost model of the lambda calculus?*, in: *International Conference on Functional Programming*, 1996, pp. 92–101.

[17] Lévy, J.-J., *Optimal reductions in the lambda calculus*, in: J. P. Hindley and J. R. Seldin, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, Academic Press, 1980 pp. 159–191.

[18] Maraist, J., M. Odersky and P. Wadler, *The call-by-need lambda calculus*, Journal of Functional Programming **8** (1998), pp. 275–317.

[19] Peyton Jones, S., "The Implementation of Functional Programming Languages," Prentice Hall International, 1987.

[20] Seaman, J. and S. P. Iyer, *An operational semantics of sharing in lazy evaluation*, Science of Computer Programming **27** (1996), pp. 289–322.

[21] Shivers, O. and M. Wand, *Bottom-up beta-reduction: uplinks and lambda-DAGs*, in: *Proceedings of European Symposium on Programming (ESOP'05)*, Lecture Notes in Computer Science **3444** (2005), pp. 217–232.

[22] Sinot, F.-R., *Call-by-need in token-passing nets*, Mathematical Structures in Computer Science **16** (2006), pp. 639–666.

[23] Wadsworth, C. P., "Semantics and Pragmatics of the Lambda-Calculus," Ph.D. thesis, Oxford University (1971).

Stack-based Strategic Control

Victor L. Winter[1]

Department of Computer Science
University of Nebraska at Omaha
USA

Abstract

In a strategic framework, combinators provide a fundamental mechanism for exercising control over rewriting. This type of control is based on the observation of the success or failure of strategy application. This paper describes a framework where information relating to the outcome of strategy application is stored in two internally maintained stacks. These stacks represent an implicit state which is used to control the rewriting process.

Keywords: program transformation, strategic programming, TL, HATS

1 Overview

In a strategic framework, combinators provide a fundamental mechanism for exercising control over rewriting. Such control is based on *observing* the outcome of strategy application (i.e., its success or failure). In this paper we describe how rewriting can be controlled in the strategic language TL [11][9]. TL distinguishes itself from standard strategic languages in four important ways: First, the behavior of unsuccessful strategy application (i.e., application failure) is an identity on terms. That is, failure is not communicated at the term level. As a result, we say that TL is an *identity-based* system. Second, TL provides a unique set of combinators for controlling strategy application. Third, in TL it is possible for strategies themselves to change during application through a mechanism known as *strategic reduction*. Thus, the result of applying a strategy s to a term t is a tuple (s', t') denoting the resultant strategy and term. Fourth, TL is higher-order in the sense that it is possible to dynamically create strategies which, in turn, can then be applied to terms.

[1] Email: vwinter@mail.unomaha.edu

1.1 Contribution

There are two primary contributions made in this paper: First, a novel framework is developed where the operation of strategy application, when viewed abstractly from the perspective of being either successful or unsuccessful, is stored in a structure called a *control stack*. Second, it is shown how control stacks can be used to give a formal big-step style semantics to a core set of TL combinators. Included in the core are the combinators

$$\{\ \textit{transient},\ \textit{opaque},\ \textit{raise},\ \textit{hide},\ \textit{lift}\ \}$$

which are unique to TL. Though informally mentioned in other articles, this article is the first place where the semantics of { *opaque*, *raise*, *hide*, *lift* } are formally defined. Two previously unpublished examples are then presented showing how these non-standard combinators can be used to realize practical transformation objectives.

The rest of this paper is organized as follows: Section 2 gives a brief overview of the basic rewriting framework of TL followed by a big-step style semantics for a set of core TL combinators. Section 3 gives two examples of how the combinators of TL can be used to realize solutions to instances of a class of problems having practical application. Section 4 discusses related work and Section 5 concludes.

2 The Semantics of TL

TL is a language that has been developed exclusively for describing transformation-based computation [11,8]. The principle artifacts manipulated during the execution of a TL program are *parse trees*, which we refer to as *terms*. TL provides a notation for describing parse tree structures relative to a given (assumed) grammar G. Trees expressed using this notation are referred to as *patterns*.

A pattern is either a subscripted nonterminal (e.g., B_1) or an expression of the form $B[\![\alpha']\!]$. An expression $B[\![\alpha']\!]$ is well-formed if and only if the derivation $B \stackrel{+}{\Rightarrow} \alpha$ is possible according to the underlying grammar and α' is obtained from α by subscripting all nonterminals occurring in α.

Transformation is accomplished through the application of rewrite rules to terms. In TL, a first-order rewrite rule has the following syntactic structure:

$$\mathit{lhs} \rightarrow \mathit{rhs}\ [\ \textit{if}\ \mathit{condition}\ ]$$

where [and] are syntactic meta symbols indicating that the enclosed section (i.e., the conditional portion) of a rule is optional. In order for a first-order rule to be well-formed it is necessary that *lhs* be a *pattern*, that *rhs* be a *strategic expression* (i.e., an expression whose evaluation yields a term), and that *condition* be an expression consisting of one or more *match expressions* combined using the Boolean connectives: *and*, *or*, *not*.

A *match expression* is a first-order match between two patterns. Let t_1 denote a pattern, possibly non-ground, and let t_2 denote a ground pattern. The expression $t_1 \ll t_2$ denotes a match expression and evaluates to *true* if and only if a substitution σ can be constructed so that $\sigma(t_1) = t_2$.

A basic conditionless n-order rule has the form:

$$lhs_1 \rightarrow lhs_2 \rightarrow \cdots \rightarrow lhs_n \rightarrow rhs$$

The current implementation of TL makes a syntactic distinction between first-order strategy application and higher-order strategy application. Let s^1 and s^2 respectively denote a first and second-order strategy and let t denote a term. In TL, the first-order application is written $s^1(t)$ and the second-order application is written $s^2[t]$.

TL provides a rich framework for defining traversals. TL also provides a predefined library of traversals that includes both first-order as well as higher-order generic traversals. The TL traversal library contains common first-order generic traversals such as bottom-up left-to-right (BUL), top-down left-to-right (TDL) and so on. The expression $BUL\{s\}$ denotes a strategy that when applied to a term t will traverse t in a bottom-up left-to-right fashion and apply the first-order strategy s to every term encountered.

TL also provides a predefined top-down (TD) traversal in addition to the top-down left-to-right (TDL) traversal. The traversal TD is distinctly different from TDL. In particular, the expression $TD\{s\}$ denotes a strategy that when applied to a term t will traverse t in a top-down fashion with the following behavior: Let s' denote the strategy that results from applying s to t. Each direct sub-term of t will be given its own copy of s' at which point the TD traversal continues. See [8] for a more detailed discussion of TD.

2.1 The Semantics of the TL Core Combinator Set

There are two fundamental combinator types in TL: those that control *strategy application*, and those that control *strategic reduction*. Examples of the former include $\{<+, <;, hide, lift\}$. Examples of the latter include the unary combinators $\{transient, opaque, raise\}$.

In TL, the behavior of all combinators can be formally described using two internally maintained stacks: $\mathcal{A}$ and $\mathcal{R}$. *Strategy application* is controlled by the stack $\mathcal{A}$. *Strategic reduction* is controlled by the stack $\mathcal{R}$.

In order to abstract away some low-level operational details from our semantics, a stack is modelled here as an infinite structure whose initial entries are all set to the Boolean value *false*. Initial stacks corresponding to $\mathcal{A}$ and $\mathcal{R}$ are respectively denoted $\bot_{\mathcal{A}}$ and $\bot_{\mathcal{R}}$. An element can be pushed onto the top of a stack using a "dot" constructor. For example, let $\mathcal{A}$ denote a stack and let x denote a Boolean value, then $x.\mathcal{A}$ denotes the stack $\mathcal{A}$ with the value x pushed on to it. Elements can be extracted from the top of a stack using pattern matching. For example, in the pattern $x.\mathcal{A}$, the symbol x denotes the top of the stack and $\mathcal{A}$ denotes the rest of the stack. Similarly, in the pattern $x_1.x_2.\mathcal{R}$ the symbol x_1 denotes the top of the stack, the symbol x_2 denotes the second symbol on the stack and $\mathcal{R}$ denotes the rest of the stack.

Lastly, we extend Boolean operations (e.g., $\vee$) to stacks. For example, let $\mathcal{A}_1$ and $\mathcal{A}_2$ denote two (infinite) stacks. The Boolean expression $\mathcal{A}_1 \vee \mathcal{A}_2$ denotes the stack whose elements are obtained from the element-wise disjunction of $\mathcal{A}_1$ and $\mathcal{A}_2$.

2.2 Basic Strategy Application

Figure 1 gives a big-step style semantics describing the basic forms of strategy application. In syntax used, the application of a strategy s to a term t is denoted $s \cdot \langle t \rangle$. The rules *E-success* and *E-failure* in Figure 1 make use of a predicate *applies* which we do not formally define here. Informally, the predicate $applies(r, t)$ evaluates to *true* if the application of the rewrite rule r to the term t is successful; otherwise it evaluates to *false*. Recall, that TL models application failure as an identity on terms. Thus, the term-level value *FAIL* is not used to communicate application failure. In the interests of space and to sidestep a digression tangential to the focus of this paper, we assume that the semantics of the predicate $applies(r, t)$ is understood.

$$\frac{applies(r,t) \quad t \xrightarrow{r} t'}{r \cdot \langle t \rangle \Downarrow \langle true.\bot_{\mathcal{A}}, true.\bot_{\mathcal{R}}, r, t' \rangle}\text{E-success}$$

$$\frac{\neg applies(r,t)}{r \cdot \langle t \rangle \Downarrow \langle \bot_{\mathcal{A}}, \bot_{\mathcal{R}}, r, t \rangle}\text{E-failure}$$

$$\frac{}{SKIP \cdot \langle t \rangle \Downarrow \langle \bot_{\mathcal{A}}, \bot_{\mathcal{R}}, SKIP, t \rangle}\text{E-skip}$$

$$\frac{}{ID \cdot \langle t \rangle \Downarrow \langle true.\bot_{\mathcal{A}}, true.\bot_{\mathcal{R}}, ID, t \rangle}\text{E-id}$$

Fig. 1. Atomic rule application and the strategic constants *SKIP* and *ID*

The result of evaluating an application of a strategy s to a term t is a tuple of the form $\langle \mathcal{A}, \mathcal{R}, s', t' \rangle$ where (1) $\mathcal{A}$ and $\mathcal{R}$ respectively denote the control stack restricting strategic application and the control stack enabling strategic reduction, and (2) s' and t' respectively denote the strategy and term resulting from the application. One thing to notice is that the result of applying the strategic constant *SKIP* to a term t produces the same result as application failure.

2.3 The Strategic Reduction Stack $\mathcal{R}$

Figure 2 gives the semantics for the combinators $\{transient, opaque, raise\}$. Informally stated, a strategy $transient(s)$ will reduce to the strategic constant *SKIP* after application if it can be observed (via the topmost element of the stack $\mathcal{R}$) that the application of the strategy s to a term t has been successful. Note that in this case, the top of the stack $\mathcal{R}$ is popped. As a result, the reduction caused by a transient is not cascading. To see what we mean by this consider the application of the strategy $transient(transient(s_1) <+ s_2)$ to a term t under an assumption Γ that the application of s_1 is successful. Furthermore Γ assumes that after the application of s_1, the strategic reduction stack has the configuration $true.\bot_{\mathcal{R}}$. In this case, the assumption Γ would entail the strategic reduction shown in Equation

$$\frac{s \cdot \langle t \rangle \Downarrow \langle \mathcal{A}, true.\mathcal{R}, s', t' \rangle}{transient(s) \cdot \langle t \rangle \Downarrow \langle \mathcal{A}, \mathcal{R}, SKIP, t' \rangle}\text{E-transient1}$$

$$\frac{s \cdot \langle t \rangle \Downarrow \langle \mathcal{A}, \bot_{\mathcal{R}}, s', t' \rangle}{transient(s) \cdot \langle t \rangle \Downarrow \langle \mathcal{A}, \mathcal{R}, transient(s'), t' \rangle}\text{E-transient2}$$

$$\frac{s \cdot \langle t \rangle \Downarrow \langle \mathcal{A}, y.\mathcal{R}, s', t' \rangle}{opaque(s) \cdot \langle t \rangle \Downarrow \langle \mathcal{A}, \mathcal{R}, opaque(s'), t' \rangle}\text{E-opaque}$$

$$\frac{s \cdot \langle t \rangle \Downarrow \langle \mathcal{A}, y.\mathcal{R}, s', t' \rangle}{raise(s) \cdot \langle t \rangle \Downarrow \langle \mathcal{A}, y.y.\mathcal{R}, raise(s'), t' \rangle}\text{E-raise}$$

Fig. 2. The combinators effecting the control stack $\mathcal{R}$

1.

$$\Gamma \vdash transient(transient(s_1) <+ s_2) \xrightarrow{reduce} transient(SKIP <+ s_2) \quad (1)$$

The *opaque* combinator restricts (in a limited fashion) the ability of the *transient* combinator to observe that its contents has been successfully applied. This is accomplished by popping the stack $\mathcal{R}$. Consider the application of the strategy $transient(opaque(s_1) <+ s_2)$ to a term t under an assumption Γ that the application of s_1 is successful. Furthermore, Γ assumes that after the application of s_1, the strategic reduction stack has the configuration $true.\bot_{\mathcal{R}}$ and the strategic value resulting from the application is s_1'. In this case, the assumption Γ would entail the strategic (identity) reduction shown in Equation 2:

$$\Gamma \vdash transient(opaque(s_1) <+ s_2) \xrightarrow{reduce} transient(opaque(s_1') <+ s_2) \quad (2)$$

In contrast, the *raise* combinator enables strategic reduction to propagate up one level (i.e., to the second enclosing *transient* if it exists). This is accomplished by duplicating the top element of the stack $\mathcal{R}$. Consider the application of the strategy $transient(transient(raise(s_1)) <+ s_2)$ to a term t under an assumption Γ that the application of s_1 is successful. Furthermore, Γ assumes that after the application of s_1, the strategic reduction stack has the configuration $true.\bot_{\mathcal{R}}$. In this case, the assumption Γ would entail the strategic reduction shown in Equation 3.

$$\Gamma \vdash transient(transient(raise(s_1)) <+ s_2) \xrightarrow{reduce} SKIP \quad (3)$$

2.4 The Strategy Application Stack $\mathcal{A}$

Figure 3 gives the semantics of the combinators $\{<+, <;, hide, lift\}$. Notice that in the rule *E-choice1*, the observation of the successful application of s_1 propagates

$$\frac{s_1 \cdot \langle t\rangle \Downarrow \langle true.\mathcal{A}, \mathcal{R}, s_1', t'\rangle}{(s_1 <+\ s_2) \cdot \langle t\rangle \Downarrow \langle true.\mathcal{A}, \mathcal{R}, s_1' <+\ s_2, t'\rangle}\text{E-choice1}$$

$$\frac{s_1 \cdot \langle t\rangle \Downarrow \langle \bot_{\mathcal{A}}, \mathcal{R}_1, s_1', t'\rangle \qquad s_2 \cdot \langle t'\rangle \Downarrow \langle \mathcal{A}_2, \mathcal{R}_2, s_2', t''\rangle}{(s_1 <+\ s_2) \cdot \langle t\rangle \Downarrow \langle \mathcal{A}_2, \mathcal{R}_1 \vee \mathcal{R}_2, s_1' <+\ s_2', t''\rangle}\text{E-choice2}$$

$$\frac{s_1 \cdot \langle t\rangle \Downarrow \langle \mathcal{A}_1, \mathcal{R}_1, s_1', t'\rangle \qquad s_2 \cdot \langle t'\rangle \Downarrow \langle \mathcal{A}_2, \mathcal{R}_2, s_2', t''\rangle}{(s_1 <;\ s_2) \cdot \langle t\rangle \Downarrow \langle (\mathcal{A}_1 \vee \mathcal{A}_2), (\mathcal{R}_1 \vee \mathcal{R}_2), s_1' <;\ s_2', t''\rangle}\text{E-seq}$$

$$\frac{s \cdot \langle t\rangle \Downarrow \langle x.\mathcal{A}, \mathcal{R}, s', t'\rangle}{hide(s) \cdot \langle t\rangle \Downarrow \langle \mathcal{A}, \mathcal{R}, hide(s'), t'\rangle}\text{E-hide}$$

$$\frac{s \cdot \langle t\rangle \Downarrow \langle x.\mathcal{A}, \mathcal{R}, s', t'\rangle}{lift(s) \cdot \langle t\rangle \Downarrow \langle x.x.\mathcal{A}, \mathcal{R}, lift(s'), t'\rangle}\text{E-lift}$$

Fig. 3. The combinators effecting the stack $\mathcal{A}$

(i.e., $true.\mathcal{A}$). More importantly notice that in the rule *E-choice2*, the value (s_1', t') propagates! This is not as disruptive to the overall semantics as first meets the eye. Furthermore, it is precisely these kinds of propagations that gives this identity-based strategic system its power.

The rule *E-seq* gives the semantics of left-to-right sequential composition. Notice that in this case the stack information in the resultant tuple corresponds to the disjunction of the stack information obtained from the application of s_1 and the application of s_2. We would like to point out that in a failure-based strategic framework a conjunction of these two stacks would be more appropriate. TL has a combinator $(<*)$ that has precisely such a semantics, but its description has been omitted from this discussion.

The semantics of the *hide* combinator is given by the rule *E-hide*. Informally speaking, *hide* restricts the ability of a combinator like $<+$ to observe whether a strategy has been successfully applied. This is accomplished by popping the stack $\mathcal{A}$. Consider the application of the strategy $hide(s_1) <+\ s_2$ to a term t under an assumption Γ that application of s_1 is successful. Furthermore, Γ assumes that, after the application of s_1, the strategy application stack has the configuration $true.\bot_{\mathcal{A}}$. In this case, the assumption Γ would entail the behavioral equivalence (from the perspective of strategic application) shown in Equation 4.

$$\Gamma \vdash hide(s_1) <+\ s_2 \equiv s_1 <; s_2 \tag{4}$$

The equivalence shown in Equation 4 generally raises an initial reaction prompting several questions: (1) Does *hide* do anything new? The answer is yes. (2) Is the *hide* combinator simply a derived form? The answer is no. We kindly ask the reader to suspend judgement on these issues until the end of Section 3. (See [9][10]

for an additional example of a nontrivial use of *hide*.)

The *lift* combinator is given by the rule *E-lift*. The *lift* combinator enables the observation of successful application to propagate past an enclosing *hide*. This is accomplished by duplicating the top element of the stack $\mathcal{A}$. Consider the application of the strategy $hide(lift(s_1)) <+ s_2$ under an assumption Γ where the application of s_1 is successful. Furthermore, Γ assumes that, after the application of s_1, the strategy application stack has the form $true.\bot_{\mathcal{A}}$. In this case, the assumption Γ would entail the behaviorial equivalence shown in Equation 5.

$$\Gamma \vdash hide(lift(s_1)) <+ s_2 \equiv s_1 \tag{5}$$

2.5 Iterators

The strategic system presented so far defines the semantics of *strategy application* to a single term: $s \cdot \langle t \rangle$. Iterators can be introduced into this framework to extend the notion of strategy application to term sequences.

In this paper, we are interested in iterators only from the perspective of their interaction with control stacks. This focus suggests viewing an iterator abstractly (in a non-constructive fashion) as a stream-like entity from which terms can be extracted in an incremental fashion. We will use the symbol Φ, in subscripted form, to abstractly denote the term stream produced by an iterator.

Term streams can be deconstructed using standard pattern matching. We write $\Phi = t_i.\Phi_{i+1}$ to denote a match expression that succeeds if the stream Φ_i can be deconstruted into the (next) term t_i and the rest of the term sequence Φ_{i+1}; otherwise the match fails. Finite term streams are terminated by the special symbol *end*. For example, a term stream consisting of a single term has the form: $t_1.end$.

Figure 4 gives the semantics of strategy application over term sequences.

$$\frac{}{s \cdot \langle end \rangle \Downarrow \langle \bot_{\mathcal{A}}, \bot_{\mathcal{R}}, s, end \rangle}\text{E-iterator1}$$

$$\frac{s \cdot \langle t_i \rangle \Downarrow \langle \mathcal{A}_1, \mathcal{R}_1, s', t_i' \rangle \qquad s' \cdot \Phi_{i+1} \Downarrow \langle \mathcal{A}_2, \mathcal{R}_2, s'', \Phi_{i+1}' \rangle}{s \cdot \langle t_i.\Phi_{i+1} \rangle \Downarrow \langle (\mathcal{A}_1 \vee \mathcal{A}_2), (\mathcal{R}_1 \vee \mathcal{R}_2), s'', t_i'.\Phi_{i+1}' \rangle}\text{E-iterator2}$$

Fig. 4. The generalization of application to term sequences

In should be noted that in practice, the term sequences produced by iterators are typically intimately linked to some initial term or set of terms. For example, a bottom-up traversal will visit all sub-terms of the initial term to which it is applied. Because of the dependencies that typically exist between the elements in term sequences, it is helpful to view a term as a mutable value. This helps to understand the global change brought about by the application of an iterator to a term.

3 Applications

This section takes a look at two examples in which the interaction between various combinators can be used to achieve specific transformational objectives. Additional examples can be found in [11][8][10][9][7].

3.1 Let-Block Optimization

This example shows how the combinators *hide*, *transient*, and *lift* can be used together with generic traversal to optimize ML-style let-blocks. An important capability highlighted by this example is that *the notion of conditional control is extended over the domain of generic traversals*. Conceptually speaking, the heart of this example is a strategy of the form: $BUL\{s_1\} <+ s_2$. In some circumstances, the application of the traversal $BUL\{s_1\}$ is seen to succeed preventing the application of s_2. In other circumstances the application of $BUL\{s_1\}$ is seen to fail thereby allowing the application of s_2.

The goal of let-block optimization, as described here, is to in-line the expression bound to the variable declared in a let-block, but only if the declared variable occurs no more than once in the body of the let-block. The strategy described in this example assumes that a let-block is an expression having the following form:

$$\textit{let val id = expr in expr end} \tag{6}$$

The strategy also assumes that all identifiers declared in a let-block are unique. Transformations establishing these two assumptions for arbitrary let-blocks are straightforward and well known. Therefore, these assumptions do not represent a loss-of generality. A grammar fragment formalizing the structures of interest, with respect to this example, is given in Figure 5. A strategic program, written in TL, realizing let-block optimization is shown in Figure 6, and a concrete example showing the results of let-block optimization is given in Figure 7.

eval_list	::=	(dec [";"] | expr ";") eval_list | ϵ
dec	::=	"val" id "=" expr | ...
expr	::=	id | let_block | ...
let_block	::=	"let" dec "in" expr "end"
id	::=	*identifier*

Fig. 5. An extended-BNF grammar fragment describing a subset of ML

In this example, the strategy *optimize_let_blocks* performs overall let-block optimization. The strategy *optimize_let_blocks* traverses the term to which it is applied in a bottom-up left-to-right (BUL) fashion and applies the strategy *simplify_let* <; *cleanup* to each term encountered during the traversal. The strategy *simplify_let* <; *cleanup* consists of the sequential (left-to-right) composition (<;) of the two strategies *simplify_let* and *cleanup*.

The strategy *simplify_let* performs the actual optimization of an individual let-block and the strategy *cleanup* removes unnecessary top-level parenthesis that may

optimize_let_blocks: $BUL\{simplify\_let<;cleanup\}$

simplify_let: $expr_0 \rightarrow (BUL\{check[id_1]\} <+ unfold)(expr_0)$
if $expr_0 \gg expr[\![let\ val\ id_1 = expr_1\ in\ expr_2\ end]\!]$

---

identity: $id_1 \rightarrow expr[\![id_1]\!] \rightarrow expr[\![id_1]\!]$

check: $id_1 \rightarrow hide(transient(identity[id_1]) <+ lift(identity[id_1]))$

unfold: $expr[\![let\ val\ id_1 = expr_1\ in\ expr_2\ end]\!]$
$\rightarrow$
$BUL\{expr[\![id_1]\!] \rightarrow expr[\![(expr_1)]\!]\}(expr_2)$

---

cleanup: ...

Fig. 6. A strategic program for optimizing let-blocks

be introduced during the optimization process. From an operational perspective, the strategy *simplify_let* is a conditional strategy whose condition assures that *simplify_let* will be applied to an expression $expr_0$ only if $expr_0$ is a let-block [2]. When a let-block is encountered, an instance of the strategy $BUL\{check[id_1]\} <+ unfold$ is dynamically created (for a specific value of id_1) and this strategy is then applied to the let-block (i.e., to $expr_0$). The strategy $BUL\{check[id_1]\} <+ unfold$ consists of the left-to-right conditional composition ($<+$) of the strategy $BUL\{check[id_1]\}$ and *unfold*.

When applied to an expression that is a let-block, the strategy *unfold* will remove the let-block and return an expression corresponding to an instance of the body of the let-block in which the expression ($expr_1$) bound to the declared variable (id_1) is substituted for all occurrences of the declared variable. This substitution is accomplished by traversing the body of the let-block ($expr_2$) and suitably rewriting all occurrences of the declared variable.

The strategy $BUL\{check[id_1]\}$ serves as a filter that, due to its conditional composition with *unfold*, only passes control to *unfold* (i.e., only permits the application of *unfold*) under suitable circumstances. Specifically, the application of *unfold* is permitted only to let-blocks whose bodies contain no more than one occurrence of their declared variable.

When applied to a specific identifier id_1, the higher-order strategy *check* will return the following strategy:

$$hide(transient(identity[id_1]) <+ lift(identity[id_1])) \tag{7}$$

Here, the strategic expression $identity[id_1]$ has been added for readability and evaluates to the first-order identity strategy [3] on the expression id_1.

[2] The conditional portion of this strategy is added only for readability. Technically speaking, the pattern $expr[\![let\ val\ id_1 = expr_1\ in\ expr_2\ end]\!]$ could have been in-lined for all occurrences of $expr_0$ in which case the conditional portion of the strategy could be removed.

[3] It should be noted that the pattern $expr[\![id_1]\!]$ will not match with left-hand side of a val declaration,

When viewed externally (e.g., from the context of the enclosing BUL traversal), the strategy *hide*(*transient*(*identity*[id_1]) <+ *lift*(*identity*[id_1])) is seen to successfully apply if and only if the strategy *lift*(*identity*[id_1]) is successfully applied. In this case, since the *transient* and *lift* combinators are applied to the same strategy, it follows that the strategy enclosed by *lift* can only be applied after the strategy enclosed by *transient* has been applied [4]. Furthermore, since these strategies are conditionally composed, we can infer that the term to which the *lift* strategy is applied must have a position that is different, in the term sequence resulting from the *BUL* traversal, than the position of term to which the *transient* strategy is applied. Specifically, this implies that there are two or more occurrences of terms matching *expr*$[\![id_1]\!]$. When this arises, control is not passed to the *unfold* strategy; otherwise control is passed to the *unfold* strategy.

Figure 7 is a concrete example showing let-block optimization. Note that this example contains instances of the three cases that must be accounted for by this optimization. Case 1 arises when there are no occurrences of the declared variable in the body of the let-block. Case 2 arises when there is exactly one occurrence of the declared variable in the body of the let-block. Case 3 arises when there are two (or more) occurrences of the declared variable in the body of the let-block.

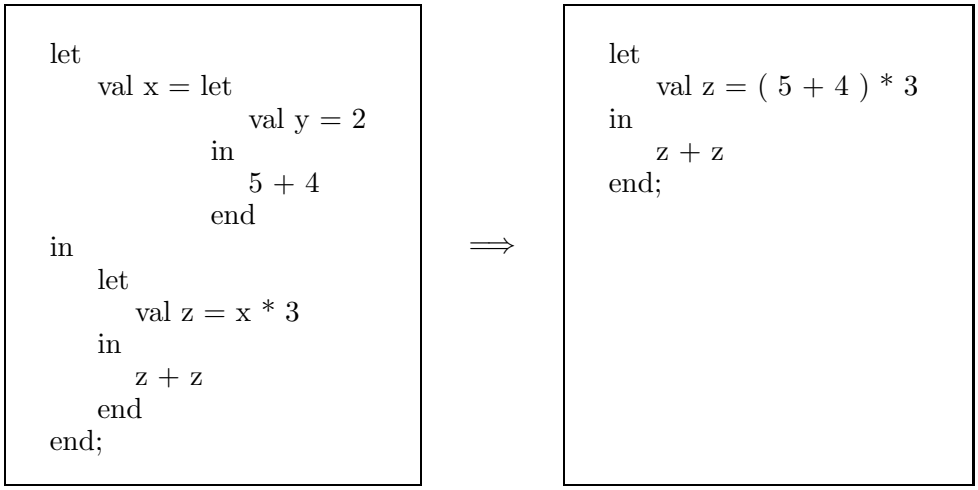


Fig. 7. A concrete example of let-block optimization

3.2 Contextual Renaming

In this example, we take a look at how the combinators *transient*, *raise*, and *opaque* can be combined to control renaming within a block-structured class hierarchy. The goal here is to implement a renaming policy that only renames variables within classes having specific structural properties. We consider two policies: (1) rename all occurrences of a specific variable within a particular (sub)class, and (2) rename all occurrences of a specific variable in all classes that are subclasses of a given class.

It should be noted that the kind of renaming policies considered here can be adapted to a variety of weaving tasks that often arise within an aspect-oriented

which is precisely what is needed.

[4] Recall that a transient enclosed strategy reduces to skip after its successful application.

framework. For example, the transformation in this example can be (and has been) adapted to implement the semantics of a variety of pointcut expressions that describe static contexts.

It is important to note that the renaming contexts in this example are "hardwired" into the strategies themselves. This is done to allow the discussion to focus on control issues. Since TL supports higher-order strategies, only a minor modification is needed to remove such hardwiring and instead parameterize renaming contexts with respect to expressions arising within the source program itself (e.g., pointcuts associated with advice).

class_list	::=	class_def class_list | ϵ
class_def	::=	"class" id "{" dec_list "}"
dec_list	::=	dec [";"] dec_list | ϵ
dec	::=	class_def | field_def | ...
field_def	::=	type id ["=" expr]
expr	::=	id | ...

Fig. 8. An extended-BNF grammar fragment for contextual renaming

A grammar fragment formalizing the structures of interest, with respect to this example, is given in Figure 8. A strategic program, written in TL, realizing contextual renaming is shown in Figure 9, and a concrete example showing the results of contextual renaming is given in Figure 10.

In this example, the strategy *contextual_rename* performs the overall renaming. This is accomplished by applying, to the input term, the strategy *rename_henceforth* <; *rename_here* using the traversal *TD*. The strategy *rename_henceforth* <; *rename_here* is the sequential left-to-right composition (<;) of the strategy *rename_henceforth* and *rename_here*.

The strategy *rename_henceforth* implements a hardwired renaming policy in which the rewrite rule *r1* is used to rewrite occurrences of the term $expr[\![henceforth]\!]$ to $expr[\![HENCEFORTH]\!]$. Control is only passed to *r1* provided the subject term occurs (1) in the context of a *class B* which is the (immediate) inner class of *class A*, or (2) in any inner class extending the context *class B* <: *class A*, where <: denotes the standard subtype relation; otherwise the application of *r1* is prevented.

Within the strategy *rename_henceforth* control over the application of *r1* is exercised by the strategic expression $filter[id[\![A]\!]] <+ filter[id[\![B]\!]]$. When evaluated, *rename_henceforth* has the form

$$transient(\varepsilon_1 <+ \varepsilon_2 <+ r1) \tag{8}$$

where ε_1 has the form:

contextual_rename: $TD\{rename\_henceforth <; rename\_here\}$

---

r1: $expr[\![henceforth]\!] \rightarrow expr[\![HENCEFORTH]\!]$

rename_henceforth: $transient(filter[id[\![A]\!]] <+ filter[id[\![B]\!]] <+ r1)$

r2: $expr[\![here]\!] \rightarrow expr[\![HERE]\!]$

rename_here:

$transient(filter[id[\![A]\!]] <+ filter[id[\![B]\!]] <+ bad\_context <+ r2)$

---

bad_context:

$class\_def[\![class\ id_1\{dec\_list_1\}]\!] \rightarrow class\_def[\![class\ id_1\{dec\_list_1\}]\!]$

filter:
id_1
$\rightarrow$
$transient($
$class\_def[\![class\ id_1\{dec\_list_1\}]\!]$
$\rightarrow$
$class\_def[\![class\ id_1\{dec\_list_1\}]\!]$
$<+$
$raise(bad\_context)$
$<+$
$opaque(ID)$
$)$

Fig. 9. A strategic program for contextual renaming

```
transient(
    (* — Case 1: Positive Match — *)
    class_def[[class A {dec_list1}]] → class_def[[class A {dec_list1}]]
    <+
    (* — Case 2: Negative Match — *)
    raise(bad_context)
    <+
    (* — Case 3: Short-circuit — *)
    opaque(ID)
)
```

It should be noted that the strategy ε_1 is a transient enclosed strategy whose body, as the comments suggest, consists of three cases. **Case 1** arises when a term t corresponding to *class A* is encountered. When this occurs the rewrite associated with case 1 applies and, due to the fact that the body of ε_1 is enclosed in a *transient*, the entire strategy ε_1 is reduced to the strategy *SKIP*.

Case 2 arises when a term t corresponding to a class other than *class A* is

encountered (i.e., we are in the wrong context). When this occurs the strategy associated with case 2 applies. Since this strategy involves the combinator *raise*, its successful application to t will cause the contents of the two enclosing transients to reduce to *SKIP*. In other words, the strategy *rename_henceforth* which has the form $transient(\varepsilon_1 <+ \varepsilon_2 <+ r1)$ will be reduced to *SKIP*. Thus, no renaming of any sub-term of t, visited by the traversal *TD*, will take place.

Case 3 applies to all terms t that do not correspond to a class definition (e.g., expr, field_def, etc.). In this case, we would like our strategy ε_1 to ignore such terms since they are of the wrong type. However, we also simultaneously want to prevent the application of rewrite *r1* with which $\varepsilon_1 <+ \varepsilon_2$ is conditionally composed. To prevent an application of *r1*, the $<+$ combinator must perceive the application of ε_1 as succeeding. This is accomplished by the strategic constant *ID*. In contrast, the *transient* combinator must perceive the application of ε_1 as failing. This is accomplished by applying the *opaque* combinator to the strategic constant *ID*.

The description of the strategy ε_2 is similar to ε_1. The only difference is that ε_2 involves the class id B whereas ε_1 involves the class id A.

The behavior of the strategy *rename_here* is similar to that of the strategy *rename_henceforth*. The only difference is that, after evaluation, the strategy *rename_here* will have the form:

$$transient(\varepsilon_1 <+ \varepsilon_2 <+ bad\_context <+ r1) \tag{9}$$

The addition of of the strategy *bad_context* assures that renaming will only occur within a designated class. In particular, renaming will not continue on into the inner classes of the designated class.

Figure 10 is a concrete example showing the results of contextual renaming. Notice that there are two occurrences of *class B* but the rewrite $here \rightarrow HERE$ only occurs in the *class B* which is an (immediate) inner class of *class A*. Furthermore, note that the strategy *rename_henceforth* also avoids rewriting the occurrence of the identifier *henceforth* in the occurrence of *class B* which is an inner class of *class D*.

4 Related Work

In TL, the $raise/opaque$ and $hide/lift$ combinators provide an exception-like mechanism for communicating information pertaining to strategy application between combinators within a strategy. Similarly, virtually all programming languages offer some mechanisms to describe nonstandard control flows that can be used to escape from nested computations. Basic mechanisms found in early programming languages include *goto*, *break*, *continue*, and *return*. Modern programming languages support more sophisticated abstractions that enable nonstandard flow of control via the throwing and catching of exceptions. Scheme takes this idea further and offers exotic control abstractions such as *call-with-current-continuation* (call/cc) and *dynamic-wind*.

Stratego [6] is a first-order strategic programming language providing a rich set of primitives for controlling the application of rewrite rules. In Stratego, *dynamic rules* can be created at run-time [2]. Such rules inherit information from the context

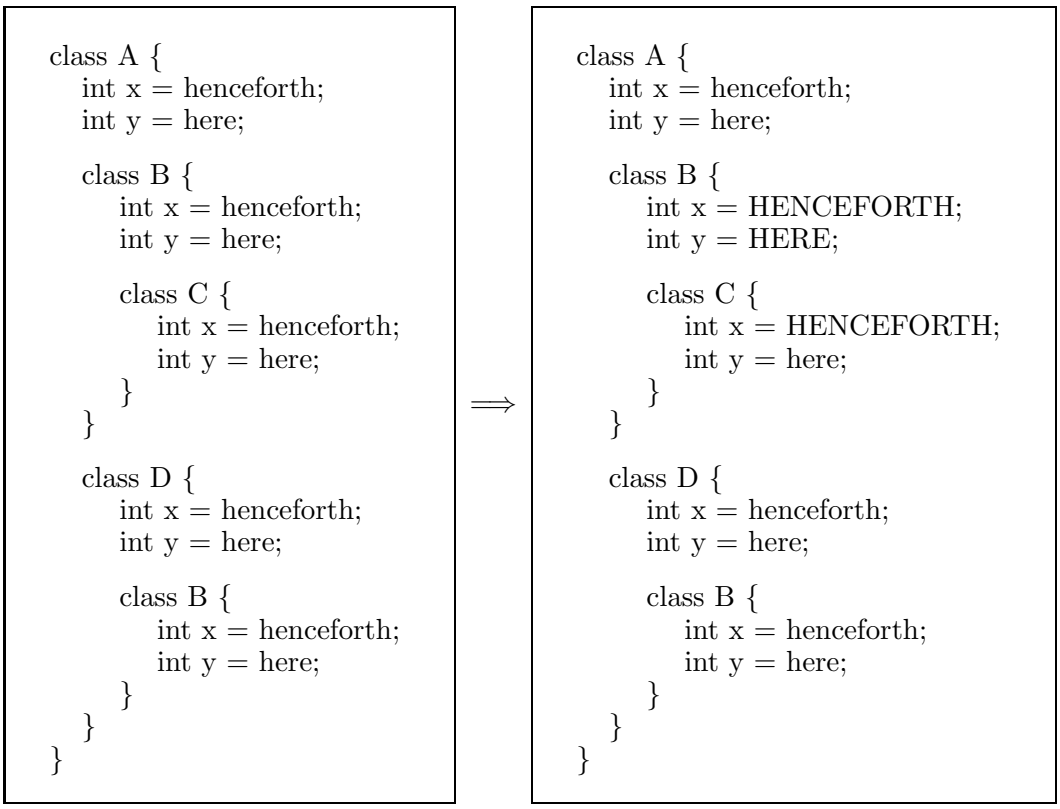


Fig. 10. A concrete example of contextual renaming

in which they are created and are analogous to the higher-order rules of TL. Stratego also provides a transient-like ability supporting "the application of dynamic rules only once" [2].

TOM [1] is a system that extends an imperative language (e.g., Java) with matching primitives. TOM supports a standard set of strategic constructs including *Choice*, *Sequence*, *All*, and *One*. In TOM, all strategies are seen as an extension of either the Identity strategy or the Fail strategy. In [1], it is shown how TOM's strategic primitives together with a generalized recursion operator μ can be used to express computational tree logic (CTL) formulae.

In [4], *conditional transformations* are defined within a logic-based framework. Transformations can be composed into OR-sequences as well as AND-sequences. The behavior of OR-sequences has similarities with the identity-based semantics of TL.

In [5], an extension to the first-order (identity-based) rewriting system ASF+SDF is described in which one can combine parameterized rewrite rules with a fixed set of generic traversals.

The ρ-calculus [3] provides a fully general (higher-order) framework in which strategies can be applied to other strategies and yield strategy sets as their results.

5 Conclusion

TL provides a rich environment in which the interplay between dynamic strategy creation and strategic reduction (via the *transient* combinator) are brought together in an identity-based framework. The identity-based nature of TL enables the notion of conditional application to be seamlessly extended over the domain of iterators. The more exotic combinators of TL (*hide*, *opaque*, etc.) greatly enhance the control that can be embedded within strategies. This kind of control is especially useful in the context of dynamic strategy generation.

References

[1] E. Balland, P.-E. Moreau, and A. Reilles. Bytecode rewriting in Tom. In *Second Workshop on Bytecode Semantics, Verification, Analysis and Transformation (Bytecode 07)*, Braga/Portugal, 2007.

[2] M. Bravenboer, A. van Dam, K. Olmos, and E. Visser. Program transformation with scoped dynamic rewrite rules. *Fundamenta Informaticae*, 69:1–56, 2005.

[3] Cirstea, Horatiu and Kirchner, Claude. An introduction to the rewriting calculus. Research Report RR-3818, INRIA, Dec. 1999.

[4] G. Kniesel. A Logic Foundation for Conditional Program Transformations. Technical Report IAI-TR-2006-1, Computer Science Department III, University of Bonn, January 2006. ISSN 0944-8535.

[5] M. G. J. van den Brand, P. Klint, and J. J. Vinju. Term rewriting with traversal functions. *ACM Trans. Softw. Eng. Methodol.*, 12(2):152–190, 2003.

[6] E. Visser, Z. el Abidine Benaissa, and A. Tolmach. Building program optimizers with rewriting strategies. In *ICFP '98: Proc. of the third ACM SIGPLAN international conference on Functional programming*, pages 13–26. ACM Press, 1998.

[7] V. Winter. Model-driven Transformation-based Generation of Java Stress Tests. *Electronic Notes in Theoretical Computer Science (ENTCS)*.

[8] V. Winter. Strategy Construction in the Higher-Order Framework of TL. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 124, 2004.

[9] V. Winter and J. Beranek. Program Transformation Using HATS 1.84. In R. Lämmel, J. Saraiva, and J. Visser, editors, *Generative and Transformational Techniques in Software Engineering (GTTSE)*, volume 4143 of *LNCS*, pages 378–396, 2006.

[10] V. Winter, J. Beranek, F. Fraij, S. Roach, and G. Wickstrom. A Transformational Perspective into the Core of an Abstract Class Loader for the SSP. *ACM Trans. on Embedded Computing Sys.*, 5(4):0–0, 2007.

[11] V. Winter and M. Subramaniam. The Transient Combinator, Higher-order Strategies, and the Distributed Data Problem. *Science of Computer Programming (Special Issue on Program Transformation)*, 52:165–212, 2004.

Computational Soundness of a Call by Name Calculus of Recursively-scoped Records

Elena Machkasova [1]

Division of Science and Mathematics
University of Minnesota, Morris
Morris, MN, U.S.

Abstract

The paper presents a calculus of recursively-scoped records: a two-level calculus with a traditional call-by-name λ-calculus at a lower level and unordered collections of labeled λ-calculus terms at a higher level. Terms in records may reference each other, possibly in a mutually recursive manner, by means of labels. We define two relations: a rewriting relation that models program transformations and an evaluation relation that defines a small-step operational semantics of records. Both relations follow a call-by-name strategy. We use a special symbol called a black hole to model cyclic dependencies that lead to infinite substitution. Computational soundness is a property of a calculus that connects the rewriting relation and the evaluation relation: it states that any sequence of rewriting steps (in either direction) preserves the meaning of a record as defined by the evaluation relation. The computational soundness property implies that any program transformation that can be represented as a sequence of forward and backward rewriting steps preserves the meaning of a record as defined by the small step operational semantics.
In this paper we describe the computational soundness framework and prove computational soundness of the calculus. The proof is based on a novel inductive context-based argument for meaning preservation of substituting one component into another.

Keywords: Calculus, call-by-name, computational soundness, recursively-scoped records

1 Introduction

In this work we present an untyped call-by-name calculus of *recursively-scoped records*. Recursively-scoped records (called *records* for the remainder of the paper) are unordered collections of labeled components that may reference each other, possibly in a mutually recursive manner. Representation of mutual dependencies arises in many calculi that model separate compilation, modules and linking, e.g. [1,15], dynamic code manipulation, e.g. [6], `letrec`, e.g. [13]. While our system has a much more modest set of features, it captures the essence of mutual dependencies – substitution with a possibility of cyclic dependencies.

We use a common approach pioneered by G. Plotkin in [11] of defining two relations in a system: a rewriting relation that represents program transformations and

[1] Email: elenam@morris.umn.edu

an evaluation relation that defines the meaning of a term via small-step operational semantics. Computational soundness connects the two relations: it implies that any two terms that are equivalent with respect to the rewriting relation have the same meaning as defined by the evaluation relation. Thus any program transformation that can be constructed as a sequence of rewriting steps (forward and/or backward) preserves the meaning of a term.

In our work both the rewriting relation and the evaluation follow the call-by-name strategy. Since this strategy allows β-reduction and substitution of unevaluated terms, the repertoire of transformations represented by the rewriting relation is greatly expanded to include unrestricted common subexpression elimination within a record, specialization, etc. The computational soundness result implies that all such transformations on mutually dependent components are meaning preserving, and thus can be used in a variety of systems modeling modules and linking, mutual dependencies in a `letrec` binding, etc. Our future plan is to investigate whether the meaning preservation result holds if the evaluation is restricted to a more efficient call-by-value strategy, while transformations follow a more liberal call-by-name one.

As demonstrated in section 5.2, our calculus fails to satisfy properties required for some previously known proof methods for computational soundness: it lacks confluence of the rewriting relation required for Plotkin's original proof method and it fails to satisfy *lift* and *project* properties required for the approach in [10]. We use a novel context-based approach to complete the proof. It is an open question whether the proof can also be completed using an alternative diagram-based approach in [14].

The main contribution of the paper is the computational soundness proof of a call-by-name calculus of mutually dependent components in the framework of term meaning defined via a small-step operational semantics. For more detailed presentation of this work, including proofs omitted here, see [8].

2 Related Work

G. Plotkin in [11] has proven a property equivalent to computational soundness for the call-by-name and the call-by-value term calculi. Z. M. Ariola and J. W. Klop studied issues of confluence and meaning preservation in similar systems of mutually dependent components. The straightforward definition of such a system breaks confluence (see [3]). In [4], in order to achieve confluence, substitution on cycles is disallowed. In [2] Z. M. Ariola and S. Blom show that unrestricted cyclic substitution is meaning preserving up to infinite unwindings of terms; their proof uses an approach that they call "skew-confluence".

In our earlier work [10,7], we proved computational soundness of a non-confluent call-by-value calculus of records similar to the one considered here. We developed and used a diagram-based proof method based on properties that we called *lift* and *project*. This approach has been further extended and generalized to a collection of abstract diagram-based proof methods in [14]. However, the system considered here does not meet the requirements of the lift and project method (see Section 5.2) and it is unclear whether it can be handled using diagram-based methods presented in [14]. Nevertheless, the novel inductive context-based method presented here allows us to prove computational soundness of the call-by-name system.

A recent independent work [12] by M. Schmidt-Schauß presents a proof of correctness of a *copy* rule (analogous to our substitution rule) in a call-by-need and a call-by-name settings. The proof approach uses the machinery of infinite trees and is significantly different from our context-based approach. The relation between the applicability of two proof methods is a subject of future research.

3 Call-by-Name Calculus of Records

Records are unordered collections of labeled terms. Terms are elements of the traditional call-by-name λ-calculus [5], extended with constants, operations, and special symbols that represent interdependencies between terms. Each term in a record is marked by its unique label. The system can be viewed as a two-level calculus, with regular terms at the lower level and records at the upper level.

The term level of the calculus is defined below. We use prefix T for sets at the term level (such as $TTerm$), R is used at the level of records.

Definition 3.1 (Term-Level Calculus Syntax)

$$
\begin{array}{lll}
M, N \in TTerm & ::= & c \mid x \mid l \mid \bullet \mid \lambda x.M \mid M_1 \mathbin{@} M_2 \mid M_1 \ + \ M_2 \\
\mathbb{C} \in TContext & ::= & \Box \mid \lambda x.\mathbb{C} \mid \mathbb{C} \mathbin{@} M \mid M \mathbin{@} \mathbb{C} \mid \mathbb{C} \ + \ M \mid M \ + \ \mathbb{C} \\
\mathbb{E} \in TEvalContext & ::= & \Box \mid \mathbb{E} \mathbin{@} M \mid \mathbb{E} \ + \ M \mid c \ + \ \mathbb{E} \\
\mathbb{N} \in TNonEvalCntxt & & \mathbb{N} \in TContext, \mathbb{N} \notin TEvalContext
\end{array}
$$

M, N denote terms, c stands for constants (such as numbers 1, 2, etc.), x, y, z are variables (distinct from constants), l stands for labels (distinct from variables and constants), $\bullet$ is a special symbol that denotes a black hole, i.e. a cyclic dependency of a record component on itself, $\lambda x.M$ is a lambda abstraction, $M_1 \mathbin{@} M_2$ is an application, $+$ is a binary operation on terms. For simplicity we use only addition in our examples, but other operations can be added. The scope of a lambda binding extends as far to the right as possible, unless limited by parentheses. It is straightforward to extend the calculus with booleans, conditionals, and other features, but for simplicity they are not considered here.

The set $FV(M)$ of free variables of a term M is defined as usual. Labels are distinct from variables, and are not included in $FV(M)$. Syntactic equivalence of terms is defined in [8] and follows the standard approach (see [5]). We use $=$ to denote equality up to α-renaming.

Contexts are used as a way of specifying a particular subterm in a term. We use $\mathbb{C}$ as a metavariable for a term context, $\mathbb{E}$ as a metavariable for a subset of general contexts called *evaluation contexts*, and $\mathbb{N}$ for the complement of this subset called *non-evaluation contexts*. The symbol $\Box$ denotes a context hole. As an example, in the term $2 + \lambda x.3$ the subterm 2 appears in the context $\Box + \lambda x.3$ (an evaluation context) and 3 appears in $2 + \lambda x.\Box$ (a non-evaluation context, since $\Box$ is under a λ). Definition 3.3 uses evaluation contexts as means to specify a subterm to be evaluated according to the evaluation relation. If a subterm appears in a non-evaluation context, it will not be reduced by evaluation.

$\mathbb{C}\{M\}$ denotes the result of filling the hole in the context $\mathbb{C}$ with the term M (we use the notation $\mathbb{C}\{M\}$ instead of the traditional $\mathbb{C}[M]$ to avoid confusion with record delimiters). For instance, if $\mathbb{C} = \lambda x.\Box$ and $M = x+2$ then $\mathbb{C}\{M\} = \lambda x.x+2$. The notations for filling an evaluation context $\mathbb{E}$ and a non-evaluation context $\mathbb{N}$ are analogous. We can also fill a hole in a context with another context (denoted as $\mathbb{C}_1\{\mathbb{C}_2\}$), the result is a context. Note that it is possible to capture free variables of M when filling a context hole. Thus we do not introduce α-renaming of contexts. Definition 5.6 introduces contexts with multiple holes.

Definition 3.2 (Record-Level Calculus)

$$
\begin{aligned}
D \in \mathit{RTerm} \quad & ::= [l_1 \mapsto M_1, ..., l_n \mapsto M_n],\ l_i \neq l_j \text{ for } i \neq j \\
\mathbb{D} \in \mathit{RContext} \quad & ::= [l \mapsto \mathbb{C}, l_1 \mapsto M_1, \ldots, l_n \mapsto M_n], \mathbb{C} \in \mathit{TContext} \\
\mathbb{G} \in \mathit{REvalContext} \quad & ::= [l \mapsto \mathbb{E}, l_1 \mapsto M_1, \ldots, l_n \mapsto M_n], \mathbb{E} \in \mathit{TEvalContext}
\end{aligned}
$$

D denotes a record with bindings of the form $l_i \mapsto M_i$. If $l \mapsto M$ occurs in a record, we say the term M is bound to the label l. We use notation $l \mapsto M \in D$ to indicate that the binding $l \mapsto M$ occurs in D. $L(D)$ denotes the set of all labels of D. We assume that all terms in a record are closed, i.e. for any record $D = [l_1 \mapsto M_1, ..., l_n \mapsto M_n]$ we have $\cup_{i=1}^{n} FV(M_i) = \emptyset$. Recall that labels are separate from variables and are not included in $FV(M)$.

The following is an example of a record: $[l_1 \mapsto 2+3, l_2 \mapsto \lambda x.x, l_3 \mapsto l_2 \mathbin{@} l_1]$. It has three components, labeled by l_1, l_2, and l_3, respectively. The term $2+3$ is bound to l_1, $\lambda x.x$ is bound to l_2, and the component bound to l_3 references the first two by applying one to the other.

Definition 3.2 also introduces two record-level contexts: a general record context $\mathbb{D}$ and record evaluation context $\mathbb{G}$. For instance, $[l_1 \mapsto 2 + \Box, l_2 \mapsto \lambda x.x]$ is a record-level evaluation context (and also a general record context since evaluation contexts are a subset of general contexts). Record-level contexts are filled with terms, not with records. For instance, one may fill the above context with a term 3 obtaining the record $[l_1 \mapsto 2+3, l_2 \mapsto \lambda x.x]$.

Record components are unordered, i.e two records that differ only in the order of their components are considered equivalent. We define α-renaming of records as α-renaming of bound variables in their components (recall that records consist of closed terms). Since records are intended to be embedded in larger systems, such as program modules, record components may be referenced from outside of a record. Thus there is no label renaming analogous to α-renaming of terms [2].

3.1 Calculus Relations

Both levels of the calculus follow the call-by-name reduction strategy. We define a *rewriting relation* $\rightarrow$ (which we also call *reduction*) and evaluation relation $\Rightarrow$ at the two levels of the calculus in definitions 3.3 and 3.5 respectively. Intuitively,

[2] It is possible to add hidden components to records that cannot be referenced from outside of a record (see [10]). Records then are identified up to renaming of hidden labels. However, here we focus on computational soundness of mutually recursive components which is independent from the issue of hidden labels.

the reduction relation represents transformations (i.e. "optimizations") of terms and records, and the evaluation relation represents the way records are evaluated at run-time by an evaluation engine (such as an interpreter). As discussed in section 5.1, the meaning of a record is defined by the result (called the *outcome*) of its evaluation.

At a more technical level, the difference between the two relations is that the rewriting relation reduces a redex in any context, while the evaluation reduces a redex in an evaluation context.

Definition 3.3 (Relations at the Term Level) *The rewriting relation* $\rightarrow$ *and the evaluation relation* $\Rightarrow$ *at the term level are defined as follows:*

$$
\begin{array}{lll}
(\lambda x.M)\ @\ N & \rightsquigarrow M[x := N] & (\beta) \\
c_1\ +\ c_2 & \rightsquigarrow c_3 \text{ where } c_3 \text{ is the result of the operation } + & (op) \\
\mathbb{E}\{R\} & \Rightarrow \mathbb{E}\{Q\} \text{ where } R \rightsquigarrow Q & \\
\mathbb{C}\{R\} & \rightarrow \mathbb{C}\{Q\} \text{ where } R \rightsquigarrow Q &
\end{array}
$$

The $\rightsquigarrow$ arrow denotes the "elementary" reduction, i.e. the basic operations at the term level of the calculus: a call-by-name β-reduction ($M[x := N]$ stands for the result of the capture-free substitution of N for x in M) and an operation (op) that replaces an operation on two constants by their result, also a constant. The rewriting relation $\rightarrow$ can perform an elementary reduction in any context $\mathbb{C}$, i.e. anywhere inside a term. The evaluation step $\Rightarrow$ performs the same operations but only in an evaluation context. $TEvalContext \subseteq TContext$ implies $\Rightarrow \subseteq \rightarrow$.

The term that is α-equivalent to the left-hand side of an elementary reduction rule is called *a term redex*. R denotes redexes. Intuitively, a redex is the subterm that gets reduced by the reduction. The redex is enclosed in a context that remains unchanged by the reduction [3]. As an example, in the reduction $\lambda x.2+3 \rightarrow \lambda x.5$ the redex is $2+3$ and the context is $\lambda x.\Box$. In the evaluation step $1+(\lambda x.x)\ @\ 3 \Rightarrow 1+3$ the redex is $(\lambda x.x)\ @\ 3$ and the context is $1+\Box$.

When writing $\mathbb{C}_1\{M_1\} = \mathbb{C}_2\{M_2\}$ we assume that we chose syntactically equivalent representatives of the α-equivalence classes of $\mathbb{C}_1\{M_1\}$ and $\mathbb{C}_2\{M_2\}$. See [8] for details.

Lemma 3.4 states that a term may have at most one redex in an evaluation context or at most one label in such a context, but not both.

Lemma 3.4 *If* $\mathbb{E}_1\{R_1\} = \mathbb{E}_2\{R_2\}$*, where* R_1, R_2 *are redexes, then* $\mathbb{E}_1 = \mathbb{E}_2$ *and* $R_1 = R_2$*. If* $M = \mathbb{E}_1\{l_1\} = \mathbb{E}_2\{l_2\}$ *then* $\mathbb{E}_1 = \mathbb{E}_2$ *and* $l_1 = l_2$ *and* $M \neq \mathbb{E}\{R\}$ *for any* $\mathbb{E}$ *and* R*.*

3.1.1 Relations at the level of records.

Following the call-by-name strategy, both a reduction of a record component and a substitution from one component into another one may copy an unevaluated term.

[3] More precisely, it is possible to find such representatives M, N in the two respective α-equivalence classes that $M \rightarrow N$ by reducing the given redex in the given context, and the context remains unchanged.

Definition 3.5 (Relations at the Level of Records)

$$\begin{array}{lll}
\mathbb{D}\{R\} & \to \mathbb{D}\{Q\} \text{ where } R \rightsquigarrow Q & (T) \\
\mathbb{D}\{l\} & \to \mathbb{D}\{N\} \text{ where } l \mapsto N \in \mathbb{D}\{l\},\ \mathbb{D} \neq [l \mapsto \mathbb{E}, \dots] & (S) \\
\mathbb{G}\{R\} & \Rightarrow \mathbb{G}\{Q\} \text{ where } R \rightsquigarrow Q & (TE) \\
\mathbb{G}\{l\} & \Rightarrow \mathbb{G}\{N\} \text{ where } l \mapsto N \in \mathbb{G}\{l\},\ \mathbb{G} \neq [l \mapsto \mathbb{E}, \dots] & (SE) \\
[l_1 \mapsto \mathbb{E}\{l_1\}, ...] & \Rightarrow [l_1 \mapsto \bullet, ...] & (B1) \\
[l_1 \mapsto \mathbb{E}\{\bullet\}, ...] & \Rightarrow [l_1 \mapsto \bullet, ...] & (B2)
\end{array}$$

Definition 3.5 gives three kinds of reductions on records, two of which have an evaluation and a rewriting version. A *term reduction* simply reduces a term redex in one of the record's components. It is a rewriting step (see rule T) when it happens in a general context and an evaluation step (rule TE) when it is in an evaluation context. For example, $[l_1 \mapsto \lambda x.2 + 3] \to [l_1 \mapsto \lambda x.5]$ is a rewriting step, but not an evaluation step. Such steps are called *non-evaluation* steps (see Definition 3.6).

Substitution replaces a label occurring in a component of a record by the term bound to that label in the record. Analogously to the term reduction, the substitution is a rewriting step (rule S) if the label occurs in a general context, and an evaluation step (rule SE) if it occurs in an evaluation context.

For example, $[l_1 \mapsto 2 + 3, l_2 \mapsto l_1 + 1] \Rightarrow [l_1 \mapsto 2 + 3, l_2 \mapsto (2 + 3) + 1]$ is an evaluation step since $\square + 1$ is an evaluation context. The following substitution is a reduction, but not an evaluation step, since l_1 appears under a lambda: $[l_1 \mapsto 2 + 3, l_2 \mapsto \lambda x.l_1] \to [l_1 \mapsto 2 + 3, l_2 \mapsto \lambda x.(2 + 3)]$. Note that, just like a term reduction, the substitution is call-by-name: the term that gets substituted does not have to be evaluated first.

The side conditions $\mathbb{D}, \mathbb{G} \neq [l \mapsto \mathbb{E}, \dots]$ eliminate an ambiguity between substitution and the black hole rule (B1) by preventing a substitution into a label that directly depends on itself in an evaluation context. For instance, the following substitution is not allowed: $[l_1 \mapsto l_1 + 1] \Rightarrow [l_1 \mapsto l_1 + 1 + 1]$, the rule (B1) is applied instead (see below).

A *black hole symbol* $\bullet$ denotes apparent infinite substitution cycles that cannot be meaningfully evaluated. The rule (B1) introduces a black hole to replace a label that depends on itself in an evaluation context. For instance, $[l_1 \mapsto l_1 + 1] \Rightarrow [l_1 \mapsto \bullet]$ instead of an infinite substitution $[l_1 \mapsto l_1 + 1] \Rightarrow [l_1 \mapsto l_1 + 1 + 1] \Rightarrow \dots$ The notion of a black hole was first introduced in [3]. In this work it is essential for confluence of $\Rightarrow$ on records.

The rule (B2) turns a component that depends on a black hole into a black hole: $[l_1 \mapsto \bullet, l_2 \mapsto l_1 + 1] \overset{S}{\Longrightarrow} [l_1 \mapsto \bullet, l_2 \mapsto \bullet + 1] \overset{B2}{\Longrightarrow} [l_1 \mapsto \bullet, l_2 \mapsto \bullet]$.

The black hole rules do not have analogous non-evaluation rules since a self-dependency in a non-evaluation context may be a legitimate recursion and does not always lead to infinite substitution cycle or it may be eliminated during evaluation.

Definition 3.6 (Non-evaluation Relation and Closures) *The following notations are used at both the term and the record level:*

(i) *A non-evaluation relation* $\hookrightarrow$ *is defined as* $\hookrightarrow = \rightarrow \setminus \Rightarrow$.

(ii) $\longrightarrow^*$, $\Longrightarrow^*$, $\hookrightarrow^*$ *denote reflexive transitive closures of the respective relations;* $\leftrightarrow$, $\overset{n}{\leftrightarrow}$ *denote the reflexive symmetric transitive closures of* $\rightarrow$ *and* $\hookrightarrow$, *respectively.*

The non-evaluation relation $\hookrightarrow$ can be equivalently defined as a reduction in a non-evaluation context.

A normal form of a term with respect to a relation R is an object that cannot be further reduced by R. The definition is applicable to both terms and records.

Definition 3.7 (Normal Form) *Given a relation* R *on a set of terms, a normal form with respect to (w.r.t.)* R *is a term* M *for which there is no* M' *such that* MRM'. *The predicate* $nf_R(M)$ *is true if* M *is a normal form w.r.t.* R *and false otherwise. A term* N *is an* R*-normal form of* M *if* MR^*N *and* $nf_R(N)$.

4 Confluence of Evaluation

It follows from Lemma 3.4 that there is at most one evaluation step in any record component. For instance, if a component is of the form $\mathbb{E}\{R\}$, i.e. it has a term evaluation redex, it may not have a label in an evaluation context.

However, there is no ordering on components in a record, so any component that has a term or a substitution redex may be evaluated. Thus it is possible to have multiple evaluation steps originating at the same record:

$$[l_1 \mapsto 2+3, l_2 \mapsto l_1+1] \Rightarrow [l_1 \mapsto 5, l_2 \mapsto l_1+1]$$
$$[l_1 \mapsto 2+3, l_2 \mapsto l_1+1] \Rightarrow [l_1 \mapsto 2+3, l_2 \mapsto 2+3+1]$$

This flexibility opens a way for modeling separate compilation and evaluation of modules: evaluation of known components may start before the entire record becomes available.

Lemma 4.1 (Confluence of Evaluation) $\Rightarrow$ *is confluent on records.*

Proof. Case analysis on pairs of evaluation redexes shows that evaluation satisfies the strip lemma (see [5], Ch. 11) which implies confluence. See [9] for details [4]. □

The presence of a black hole in the calculus is essential for confluence of evaluation. Consider the following record: $[l_1 \mapsto 2 + l_2, l_2 \mapsto l_1 + 1]$. Note that both labels are in evaluation contexts in both components. Without a black hole the substitution into the first component would yield $[l_1 \mapsto 2 + l_1 + 1, l_2 \mapsto l_1 + 1]$, substitution into the second component gives $[l_1 \mapsto 2 + l_2, l_2 \mapsto 2 + l_2 + 1]$. In the first resulting record both components reference l_1, in the second one they both reference l_2, and any subsequent substitutions preserve these properties. This is a variation of a famous non-confluence example introduced in [3].

However, a black hole allows us to bring these two records together by a sequence of evaluation steps since both labels appear in an evaluation context, and thus

[4] The black hole rules in the system in [9] differ slightly from the rules presented here. However, the difference does not affect the essence of the proof.

represent an infinite cycle of substitutions:

$$\begin{aligned}
&[l_1 \mapsto 2 + l_1 + 1, l_2 \mapsto l_1 + 1] \Rightarrow \\
&[l_1 \mapsto \bullet, l_2 \mapsto l_1 + 1] \Rightarrow \\
&[l_1 \mapsto \bullet, l_2 \mapsto \bullet + 1] \Rightarrow \\
&[l_1 \mapsto \bullet, l_2 \mapsto \bullet]
\end{aligned}$$

The record $[l_1 \mapsto 2 + l_2, l_2 \mapsto 2 + l_2 + 1]$ also evaluates to $[l_1 \mapsto \bullet, l_2 \mapsto \bullet]$.

Confluence of evaluation guarantees uniqueness of a normal form w.r.t. $\Rightarrow$ if a term has one. We also prove that a record may not have a normal form and diverge at the same time (this property, also known as *uniform normalization*, is not automatically implied by confluence).

Lemma 4.2 *If $D \Longrightarrow^* D'$, $nf_{\Rightarrow}(D')$, and no component in D' is bound to $\bullet$, then there is no infinite sequence $D \Rightarrow D_1 \Rightarrow D_2 \ldots$.*

4.1 An Efficient Evaluation Strategy

Confluence of evaluation guarantees that no matter what path an evaluation of a record takes, all of the resulting records can be evaluated to the same record. We do not want to fix the order of evaluating components since we would like to leave the flexibility of modeling systems where progress can be made on evaluating a record before all of its components become available or where components may be evaluated in parallel.

However, for proving properties of our calculus it is convenient to impose a particular order of evaluation that we call *efficient evaluation strategy*. Intuitively, this strategy requires that if a component bound to l_1 needs a component bound to l_2 (i.e. the component bound to l_1 is of the form $\mathbb{E}\{l_2\}$) then the term bound to l_2 must be completely evaluated (i.e. not have either term redexes or substitution redexes) before the substitution into the component bound to l_1 is made. This strategy imposes a partial order on components. The process stops if it discovers a cycle of mutual evaluation dependencies.

The formal definition depends on the partial function $\mathrm{next}(D, l)$ that defines the label of the component in which the next evaluation step takes place in order to make progress on evaluation of the component bound to l in D.

Definition 4.3 (Next Component To Be Evaluated) *Let $l \mapsto M \in D$. A function $next(D, l) : RTerm \times L(D) \to L(D) \cup \{\bullet\}$ is defined as follows:*

(i) *If $M = \mathbb{E}\{R\}$ then $next(D, l) = l$,*

(ii) *If $M = \mathbb{E}\{\bullet\}$ or $M = \mathbb{E}\{l\}$ then $next(D, l) = \bullet$,*

(iii) *If $M = \mathbb{E}\{l'\}$ then:*
   (a) *If $next(D, l')$ is undefined, $next(D, l) = l$,*
   (b) *If $next(D, l') = \bullet$ or l' is bound to $\mathbb{E}'\{l\}$ or there is a sequence of labels $l_1, \ldots, l_n \in L(D)$, $n \geq 1$, such that D is of the form*

$$[l \mapsto \mathbb{E}\{l'\}, l' \mapsto \mathbb{E}_1\{l_1\}, \ldots, l_i \mapsto \mathbb{E}_i\{l_{i+1}\}, \ldots, l_n \mapsto \mathbb{E}_n\{l\}, \ldots]$$

then $next(D, l) = \bullet$,

(c) *Otherwise* $next(D, l) = next(D, l')$.

(iv) *Otherwise* $next(D, l)$ *is undefined.*

If $\text{next}(D, l)$ is undefined then the component bound to l is fully evaluated.

Let $\mathcal{L}$ to denote an ordered sequence of distinct labels; $\mathcal{L}_1 \preceq \mathcal{L}_2$ means that $\mathcal{L}_1$ is a prefix of $\mathcal{L}_2$ or $\mathcal{L}_1 = \mathcal{L}_2$. An efficient evaluation strategy follows the sequence of labels in $\mathcal{L}$ as a sequence of "goals".

Definition 4.4 (Efficient Evaluation Strategy) *Given a record D and a label l, an efficient evaluation strategy starting at l is a sequence of evaluation steps $D_1 \Rightarrow D_2 \Rightarrow \ldots \Rightarrow D_n$ s.t. for all $i < n$ $next(D_i, l)$ is defined and not equal to $\bullet$ and an evaluation step $D_i \Rightarrow D_{i+1}$ evaluates the component bound to $next(D_i, l)$. We denote this sequence as $D \underset{e}{\overset{l}{\Longrightarrow}}{}^* D_n$.*

Given a sequence $\mathcal{L} = l_1, l_2, \ldots l_n$ s.t. $l_i \in L(D)$ for all i, an efficient strategy w.r.t. $\mathcal{L}$ is a sequence $D \underset{e}{\overset{l_1}{\Longrightarrow}}{}^* D_1 \underset{e}{\overset{l_2}{\Longrightarrow}}{}^* \ldots \underset{e}{\overset{l_n}{\Longrightarrow}}{}^* D_n$ s.t. $next(D_i, l_j)$ is undefined for all $j < i$ for $1 \leq i \leq n$ (i.e. each component l_j in $\mathcal{L}$ is fully evaluated before evaluation of l_i starts). Note that it is possible that $next(D_n, l_n)$ is not undefined. An efficient strategy w.r.t. $\mathcal{L}$ is denoted $\underset{e}{\overset{\mathcal{L}}{\Longrightarrow}}{}^*$.

The efficient evaluation strategy stops if it discovers that a record component evaluates to a black hole since such records represent divergence (see Definition 5.2). Thus, if $\text{next}(D, l) = \bullet$, no evaluation takes place.

The strategy is called "efficient" because it evaluates a component only once - the first time it is needed. Since no unevaluated components are copied, no computation is duplicated. This is similar to a call-by-value or a call-by-need strategy. However, unlike the call-by-value strategy, it does not require that a component evaluates to a value before it can be substituted (traditionally only constants, variables, and λ-abstractions are considered values), only to a substitution-free normal form. If a record D evaluates to a normal form D' then efficient evaluation strategy with any choice of $\mathcal{L}$ that includes all labels in $L(D)$ reaches D'. The strategy detects cycles of substitution as early as possible since the evaluation follows component dependencies as far as possible before evaluating any of them.

Below is an evaluation sequence that follows the efficient strategy w.r.t. l_1. On the left on line i we show the value of $\text{next}(D_i, l_1)$. For simplicity we write just $\text{next}(l)$ instead of $\text{next}(D, l)$ since the record on each line is obvious.

$$
\begin{array}{lll}
\text{next}(l_1) = l_3 & [l_1 \mapsto l_2, l_2 \mapsto l_3 + 2, l_3 \mapsto 1 + 3] & \Rightarrow \\
\text{next}(l_1) = l_2 & [l_1 \mapsto l_2, l_2 \mapsto l_3 + 2, l_3 \mapsto 4] & \Rightarrow \\
\text{next}(l_1) = l_2 & [l_1 \mapsto l_2, l_2 \mapsto 4 + 2, l_3 \mapsto 4] & \Rightarrow \\
\text{next}(l_1) = l_1 & [l_1 \mapsto l_2, l_2 \mapsto 6, l_3 \mapsto 4] & \Rightarrow \\
\text{next}(l_1) \text{ is undefined} & [l_1 \mapsto 6, l_2 \mapsto 6, l_3 \mapsto 4] &
\end{array}
$$

In contrast the step below does not follow the efficient strategy: the redex $l_3 + 2$ is

duplicated so it will have to be evaluated twice, possibly duplicating evaluation of $1 + 3$ as well. Note, however, that the record eventually evaluates to the same one as in the efficient evaluation sequence above.

$$[l_1 \mapsto l_2, l_2 \mapsto l_3 + 2, l_3 \mapsto 1 + 3] \quad \Rightarrow$$
$$[l_1 \mapsto l_3 + 2, l_2 \mapsto l_3 + 2, l_3 \mapsto 1 + 3] \Rightarrow \ldots$$

If a record D has a normal form without black-hole-bound components, it is possible to reach that normal form using the efficient evaluation strategy with any sequence $\mathcal{L}$ that includes all labels in $L(D)$ (see [8] for the proof).

5 Computational Soundness of the Calculus

5.1 Definition of Computational Soundness

Computational soundness states that rewriting relation in the calculus preserves the meaning of terms. A term's meaning is given by its normal form w.r.t. evaluation relation if such a normal form exists, otherwise the "meaning" is divergence. The notion of *outcome* in Definition 5.2 formalizes this idea. Some normal forms may be syntactically different, but have the same "meaning". The classification function (Definition 5.1) groups terms based on their "meaning".

5.1.1 Classification function.

In order to define a term's meaning, we partition all terms into *equivalence classes*. The function that assigns a class to a term is called a *classification* (the term first introduced in [7]). Two elements of the same class have the same meaning (however, they may be further distinguished by supplying a context that uses them). For instance, at the term level it is reasonable to make constants 2 and 3 be in different classes since their meaning is clearly different. However, it is common to group all lambda abstractions in the same class since a function by itself is not distinguishable from any other function until it is applied.

It is possible to define different classification functions for the same calculus. For instance, one may wish to distinguish between different types of errors by further subdividing the **error** class. On the other hand, if one is only concerned with proving termination equivalence then all normal forms may be placed into one class [5]. A calculus may be computationally sound for one choice of classification and unsound for another.

The classification function used in this work is defined in Definition 5.1. For simplicity we use the same notation Cl for the classification function at both levels of the calculus. This function is very similar to the one used in [7], except for the inclusion of a black hole. Since record components contain only closed terms, we do not have term-level classes for variables: a label bound to a variable would be considered an error. The function is well-defined on α-equivalence classes both at the term and at the record level.

[5] Records with at least one component bound to a black hole should be classified as diverging

Definition 5.1 (Classification) *The classification function* $Cl : TTerm \cup RTerm \to S$*, where* S *is a set of equivalence classes, is defined as follows:*

- $Cl(M) = \mathbf{eval}$ *if* $M = \mathbb{E}\{R\}$*,* R *is a redex. Such terms are called evaluatable.*
- $Cl(c) = \mathbf{const}(c)$*, where* $\mathbf{const}(c_1) = \mathbf{const}(c_2)$ *if and only if* $c_1 = c_2$
- $Cl(\bullet) = \bullet$
- $Cl(\lambda x.N) = \mathbf{abs}$
- $Cl(\mathbb{E}\{l\}) = \mathbf{stuck}(l)$*, where* $\mathbf{stuck}(l_1) = \mathbf{stuck}(l_2)$ *if and only if* $l_1 = l_2$
- $Cl(M) = \mathbf{error}$ *if* M *does not belong to any of the above categories*
- $Cl([l_1 \mapsto M_1, \ldots l_n \mapsto M_n]) = [l_1 \mapsto Cl(M_1), \ldots l_n \mapsto Cl(M_n)]$ *if* $Cl(M_i) \neq \bullet$ *for all* i *s.t.* $1 \leq i \leq n$
- $Cl([\ldots, l_i \mapsto \bullet, \ldots]) = \bot$

An equivalence class of a record D with no label bound to a black hole is an unordered collection of labeled term-level classes corresponding to components of D. For instance, $Cl([l_1 \mapsto \lambda x.x, l_2 \mapsto l_1 \ @ \ 1]) = [l_1 \mapsto \mathbf{abs}, l_2 \mapsto \mathbf{stuck}(l_1)]$.

Since a black hole represents an infinite substitution, the class of a record with a black-hole-bound component is $\bot$. Note that a record with a black hole in a non-evaluation context does not necessarily diverge, and thus is not classified as $\bot$: consider $[l \mapsto (\lambda x.1) \ @ \ \bullet] \Rightarrow [l \mapsto 1]$, the latter record is a normal form.

The following property, called *class preservation*, is important for proving computational soundness: if $D_1 \hookrightarrow D_2$ (recall Definition 3.6) then $Cl(D_1) = Cl(D_2)$.

5.1.2 Outcome and Computational Soundness.

Classification characterizes a record at a given moment, while *outcome* characterizes its "ultimate fate" - what happens to it if it gets evaluated as far as possible.

Definition 5.2 (Outcome) *The outcome of a record* D*, denoted* $Outcome(D)$*, is* $Cl(D')$ *where* D' *is the normal form of* D *w.r.t.* $\Rightarrow$ *if* D *has a normal form or a symbol* $\bot$ *if evaluation of* D *diverges.*

Lemmas 4.1 and 4.2 guarantee that the outcome is well-defined since every record either has a unique normal form or diverges on all evaluation paths (we identify a label bound to a black hole with divergence). The outcome formalizes the notion that the meaning of a term is the result of its evaluation.

Definition 5.3 (Meaning Preservation and Computational Soundness) *A relation* R *is meaning preserving if* MRN *implies that* $Outcome(M) = Outcome(N)$*. A calculus is computationally sound if* $\leftrightarrow$ *is meaning preserving.*

The evaluation $\Rightarrow$ is meaning preserving since it is confluent (see Lemma 4.1).

The above classification groups all abstractions in one class. However, this does not mean that replacing an abstraction by any other one may be considered meaning preserving. One can always distinguish two semantically different abstractions by considering them in a record with a term that applies the abstraction to an argument. A transformation is provably meaning preserving if its results are the same no matter what other components appear in a record. Since we can assume that any abstraction bound to a label is applied to arbitrary terms in other components,

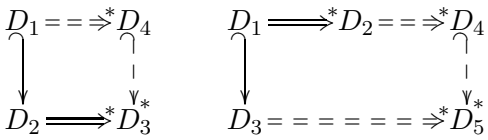


Fig. 1. Lift and Project properties.

transformations must preserve the actual behavior of abstractions. [7] formalizes this notion via record contexts which we do not present here due to lack of space.

5.2 Proof Methods and Their Applicability

Historically various methods have been used for proving computational soundness. Plotkin's method in [11] requires *confluence of the rewriting relation* in the calculus. However, many recently developed calculi model such inherently non-confluent features of programming languages as mutually dependent components. The repertoire of proof methods has been expanded to relax requirements on the calculus. In this section we review some of these proof methods and show that our calculus fails to satisfy their requirements.

Failure of Confluence and Standardization Method. The traditional method for computational soundness proofs has three requirements: confluence of the rewriting relation, standardization (a property that relates the rewriting relation and the evaluation relation), and the class preservation property defined in Section 5.1.1 (see [7] for detailed discussion). However, in our system $\rightarrow$ is non-confluent. The non-confluence example below is based on that in [3]. It also appears in the call-by-value version of our calculus described in [7,10]. Recall that confluence is preserved when both reductions are evaluation steps, see section 4.

Example 5.4 Consider a record $[l_1 \mapsto \lambda x.l_2, l_2 \mapsto \lambda y.l_1]$. By reducing each of the two redexes we obtain these two records: $[l_1 \mapsto \lambda x.\lambda y.l_1, l_2 \mapsto \lambda y.l_1]$ and $[l_1 \mapsto \lambda x.l_2, l_2 \mapsto \lambda y.\lambda x.l_2]$. The only reductions that can originate from these records are substitutions. No matter what substitutions we perform on the records, they cannot be reduced to a common one since in the first one both components always reference l_1, and in the second record they reference l_2.

Failure of Lift and Project Method. In [7,10] we use an approach based on three properties of the calculus: the *lift*, and *project* properties defined in Definition 5.5, and the class preservation property in Section 5.1.1 to prove computational soundness of a call-by-value calculus of recursively-scoped records. The project property "projects" a given evaluation sequence down, and the lift property "lifts" a given sequence up; see Figure 1 [6].

Definition 5.5 (Lift and Project) *A calculus has the lift property if, given* $D_1 \hookrightarrow D_2 \Longrightarrow^* D_3$, *there exists* D_4 *s.t.* $D_1 \Longrightarrow^* D_4 \hookrightarrow^* D_3$. *A calculus has the project property if, given* $D_1 \Longrightarrow^* D_2$ *and* $D_1 \hookrightarrow D_3$, *there exist* D_4, D_5 *s.t.* $D_2 \Longrightarrow^* D_4 \hookrightarrow^* D_5$ *and* $D_3 \Longrightarrow^* D_5$.

[6] In diagrams double arrows represent $\Rightarrow$, single arrows $\rightarrow$, arrows with a hook are $\hookrightarrow$. Solid arrows are the given relations, dashed arrows are those claimed to exist by the property. See Definition 3.6 for closure notations.

Even though the current system is very similar to the one considered in [7,10], the call-by-name nature of substitution breaks the lift and the project properties, as shown by the following counterexample. The right hand side non-evaluation arrow pointing up contradicts the properties.

$$\begin{array}{ccc} [l_1 \mapsto 2+3, l_2 \mapsto \lambda x.l_1] & \Longrightarrow & [l_1 \mapsto 5, l_2 \mapsto \lambda x.l_1] \\ \downarrow & & \downarrow \\ [l_1 \mapsto 2+3, l_2 \mapsto \lambda x.2+3] & \Longrightarrow & [l_1 \mapsto 5, l_2 \mapsto \lambda x.2+3] \end{array}$$

$$]l_1 \mapsto 5, l_2 \mapsto \lambda x.5]$$

Applicability of Other Diagram-Based Methods. The lift and project method has been extended and generalized in [14]. While it is possible that a form of the approach presented there, known as *lift/project when terminating* (or LPT), is applicable, we have not been able to construct such a proof.

A black hole, which is technically a normal form, may require a modification of the LPT approach. In our system a non-evaluation step may convert a record with a component evaluating to black hole to a diverging record, as shown below. Diagram-based methods generally do not equate diverging terms with normal forms. Note that the outcome of both records is $\bot$ so the meaning is preserved.

$$\begin{array}{lcll} [l_1 \mapsto l_2 \mathbin{@} 2, l_2 \mapsto \lambda x.l_1] & \Rightarrow & [l_1 \mapsto (\lambda x.l_1) \mathbin{@} 2, l_2 \mapsto \lambda x.l_1] & \Rightarrow \\ [l_1 \mapsto l_1, l_2 \mapsto \lambda x.l_1] & \Longrightarrow^* & [l_1 \mapsto \bullet, l_2 \mapsto \lambda x.\bullet] & \\ [l_1 \mapsto l_2 \mathbin{@} 2, l_2 \mapsto \lambda x.l_2 \mathbin{@} 2] & \Rightarrow & [l_1 \mapsto (\lambda x.l_2 \mathbin{@} 2) \mathbin{@} 2, l_2 \mapsto \lambda x.l_2 \mathbin{@} 2] & \Rightarrow \\ [l_1 \mapsto l_2 \mathbin{@} 2, l_2 \mapsto \lambda x.l_2 \mathbin{@} 2] & \Rightarrow & \ldots & \end{array}$$

5.3 Context-Based Proof of Computational Soundness

Meaning Preservation of the Term Reduction. The meaning preservation property of a term reduction can be proven using the lift and project approach with the machinery of marked redexes and residuals. The proof is similar to that for the call-by-value calculus in [7]. See [8] for details.

Meaning Preservation of Substitution. We show that substitution preserves the outcome of a record. A key idea of the proof is to use the efficient evaluation strategy (see Definition 4.4) to guarantee that each component is only evaluated once, the first time it is needed.

Definition 5.6 (Multi-hole contexts) *A multi-hole context* $\mathbb{M}$ *is defined as*

$$\mathbb{M} ::= \Box \mid M \mid \lambda x.\mathbb{M} \mid \mathbb{M} + \mathbb{M} \mid \mathbb{M} \mathbin{@} \mathbb{M}$$

Contexts $\mathbb{M}$ are filled with terms in the same manner as single-hole contexts.

Multi-hole contexts allow us to formalize the notion that two records differ only by replacing some occurrences of a term M_1 by M_2.

Definition 5.7 *A record* D_1 *is called* (M_1, M_2)*-similar to a record* D_2 *(denoted*

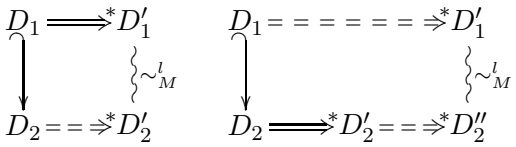


Fig. 2. Lemma 5.8: (l, M)-similarity (denoted by a wave-like line) preserved by $\Rightarrow$

$D_1 \sim_{M_2}^{M_1} D_2$) *if there exist multi-hole contexts* $\mathbb{M}_1, \ldots, \mathbb{M}_n$ *s.t.*

$$D_1 = [l_1 \mapsto \mathbb{M}_1\{M_1, \ldots, M_1\}, \ldots, l_n \mapsto \mathbb{M}_n\{M_1, \ldots, M_1\}],$$
$$D_2 = [l_1 \mapsto \mathbb{M}_1\{M_2, \ldots, M_2\}, \ldots, l_n \mapsto \mathbb{M}_n\{M_2, \ldots, M_2\}].$$

Lemma 5.8 *(see Figure 2) Let* $D_1 = [l \mapsto M, l' \mapsto \mathbb{N}\{l\}, \ldots] \overset{S}{\hookrightarrow} [l \mapsto M, l' \mapsto \mathbb{N}\{M\}, \ldots] = D_2$ *and* $D_1 \Longrightarrow^* D_1'$ *(recall that* $\mathbb{N}$ *is a non-evaluation context) and let* $\mathcal{L} \preceq l, l', l_1, \ldots, l_n$*, where* $l_1 \ldots l_n$ *is a sequence of labels in* $L(D_1)$*,* $n \geq 0$*, and* $l \neq l_i$*,* $l' \neq l_i$ *for all* $1 \leq i \leq n$*. It is possible that* $l = l'$*. Then*

- *If* $D_1 \underset{e}{\overset{\mathcal{L}}{\Longrightarrow}}{}^* D_1'$ *then there exists* D_2' *s.t.* $D_2 \underset{e}{\overset{\mathcal{L}}{\Longrightarrow}}{}^* D_2'$*,* $D_1' \sim_M^l D_2'$*, and* $Outcome(D_1') = \bot$ *if and only if* $Outcome(D_2') = \bot$*.*
- *If* $D_2 \underset{e}{\overset{\mathcal{L}}{\Longrightarrow}}{}^* D_2'$ *then there exist* D_1', D_2'' *s.t.* $D_1 \underset{e}{\overset{\mathcal{L}}{\Longrightarrow}}{}^* D_1'$ *and* $D_2' \underset{e}{\overset{l_n}{\Longrightarrow}}{}^* D_2''$*,* $D_2'' \sim_M^l D_1'$*, and* $Outcome(D_1') = \bot$ *if and only if* $Outcome(D_2') = \bot$*.*

Lemma 5.8 states the key property for meaning preservation of a substitution step: the original and the transformed records remain (l, M)-similar after any number of steps in the evaluation sequence that follows the efficient strategy with the sequence of labels given in the lemma.

The efficient evaluation strategy guarantees that every component gets evaluated to a normal form *before* it gets substituted into any other component. The strategy attempts to evaluate the term M (bound to l) in both records. By Lemma 5.8 if such an evaluation terminates with a black-hole-free term in one record, it does so in the other. In this case all components needed for evaluating M have been evaluated as well, so future evaluation of M gives the same result. In an example below the initial substitution occurs in the second component, and evaluating l requires evaluating l''; the corresponding records in the two sequences are $(l, l'' + 2)$-similar:

$$[l \mapsto l'' + 2, l' \mapsto (\lambda x.l)\ @\ 1, l'' \mapsto 3 + 1] \Longrightarrow^* [l \mapsto 6, l' \mapsto (\lambda x.l)\ @\ 1, l'' \mapsto 4]$$
$$[l \mapsto l'' + 2, l' \mapsto (\lambda x.l'' + 2)\ @\ 1, l'' \mapsto 3 + 1] \Longrightarrow^* [l \mapsto 6, l' \mapsto (\lambda x.l'' + 2)\ @\ 1, l'' \mapsto 4]$$

The sequences continue with $(l, l'' + 2)$-similar records until both arrive at an identical result $[l \mapsto 6, l' \mapsto 6, l'' \mapsto 4]$. See [8] for other cases of component dependencies.

We show that if two normal forms D_1, D_2 are (l, M)-similar then $Cl(D_1) = Cl(D_2)$. Thus non-evaluation substitution preserves the outcome.

Evaluation steps preserve the outcome since $\Rightarrow$ is confluent. We have shown that both a term reduction non-evaluation step and a non-evaluation substitution preserve the outcome. Thus we have the desired computational soundness result:

Theorem 5.9 *If $D_1 \leftrightarrow D_2$ then $Outcome(D_1) = Outcome(D_2)$.*

6 Conclusions and Future Work

We have proven that the call-by-name calculus of recursively-scoped records is computationally sound. Our system captures the essential features of mutually recursive components. We plan to study applicability of our proof method to more complex systems with possible cyclic dependencies, such as `letrec` calculi and more sophisticated systems that model modules and linking. We will also investigate how the context method compares to other methods of proving computational soundness.

Acknowledgement

Many thanks to Emily Christiansen who participated in the early stages of this research, to Dr. Manfred Schmidt-Schauß for helpful discussions, to Dr. Franklyn Turbak and Dr. Nicholas McPhee for detailed feedback on drafts of this paper, and to anonymous WRS reviewers for many helpful comments.

References

[1] Davide Ancona and Elena Zucca: A calculus of module systems. Vol. 12, 2002, pp. 91-132.

[2] Z. M. Ariola, Stefan Blom: Skew confluence and the lambda calculus with letrec. Annals of pure and applied logic 117/1-3, 97-170, 2002

[3] Z. M. Ariola and J. W. Klop: Equational Term Graph Rewriting. Fundamentae Informaticae, Vol. 26, Nrs. 3,4, June 1996. p. 207-240.

[4] Z. M. Ariola, J. W. Klop: Lambda calculus with explicit recursion. Journal of Information and Computation, Vol. 139 (2): 154-233, 1997.

[5] H. P. Barendregt: The Lambda Calculus, its Syntax and semantics. Studies in Logic, volume 103, Elsevier Science Publishers, 1984.

[6] Sonia Fagorzi and Elena Zucca: A Calculus for Reconfiguration: (Extended abstract). Electr. Notes Theor. Comput. Sci., Vol. 135, N. 3, 2006, pp. 49-59.

[7] E. Machkasova: Computational Soundness of Non-Confluent Calculi with Applications to Modules and Linking, Ph.D. dissertation, April 2002, Boston University

[8] E. Machkasova: Computational Soundness of a Call by Name Calculus of Recursively-scoped Records. Working Papers Series, University of Minnesota, Morris, Volume 2 Number 3, 2007. Available at http://cda.morris.umn.edu/~elenam/

[9] E. Machkasova, E. Christiansen: Call-by-name Calculus of Records and its Basic Properties. Working Papers Series, University of Minnesota, Morris, Volume 2 Number 2, 2006 (updated 2007). Available at http://cda.morris.umn.edu/~elenam/

[10] E. Machkasova, F. Turbak: A calculus for link-time compilation. In Programming Languages & Systems, 9th European Symp. Programming, volume 1782 of LNCS, pages 260-274 Springer-Verlag, 2000

[11] G. D. Plotkin: Call-by-name, call-by-value and the lambda calculus. Theoret. Comput. Sci., 1, 1975.

[12] M. Schmidt-Schauß: Correctness of copy in calculi with letrec, case and constructors. Frank report 28, Institut für Informatik. Fachbereich Informatik und Mathematik. J. W. Goethe-Universität Frankfurt am Main, February 2007.

[13] Manfred Schmidt-Schauß and Michael Huber: A lambda-calculus with letrec, case, constructors and non-determinism. In First International Workshop on Rule-Based Programming, 2000.

[14] J. B. Wells, Detlef Plump, and Fairouz Kamareddine: Diagrams for meaning preservation. In Rewriting Techniques & Applications, 14th Int'l Conf., RTA 2003, volume 2706 of LNCS, pp. 88-106. Springer-Verlag, 2003

[15] J. B. Wells and René Vestergaard: Equational reasoning for linking with first-class primitive modules. In Programming Languages & Systems, 9th European Symp. Programming, volume 1782 of LNCS, pages 412-428. Springer-Verlag, 2000.

Minimality in a Linear Calculus with Iteration

Sandra Alves[a] Mário Florido[a] Ian Mackie[b]
François-Régis Sinot[a]

[a] *Universidade do Porto, DCC/LIACC*
Rua do Campo Alegre 1021–1051
Porto, Portugal

[b] *LIX, CNRS UMR 7161*
École Polytechnique
91128 Palaiseau, France

Abstract

System $\mathcal{L}$ is a linear version of Gödel's System $\mathcal{T}$, where the λ-calculus is replaced with a linear calculus; or alternatively a linear λ-calculus enriched with some constructs including an iterator. There is thus at the same time in this system a lot of freedom in reduction and a lot of information about resources, which makes it an ideal framework to start a fresh attempt at studying reduction strategies in λ-calculi. In particular, we show that call-by-need, the standard strategy of functional languages, can be defined directly and effectively in System $\mathcal{L}$, and can be shown minimal among weak strategies.

1 Introduction

Gödel's System $\mathcal{T}$ is an extremely powerful calculus: essentially anything that we want to compute can be expressed [8]. A *linear* variant of this well-known calculus, called System $\mathcal{L}$, was introduced in [1], and shown to be every bit as expressive as System $\mathcal{T}$. The novelty of System $\mathcal{L}$ is that it is based on the linear λ-calculus, and all duplication and erasing can be done through an encoding using the iterator.

There are many well-known, and well-understood, strategies for reduction in the (pure) λ-calculus. When investigating deeper into the structure of terms, we get a deeper understanding of reduction. For instance, calculi with explicit resource management or explicit substitution allow a finer control over reduction. In a similar way, System $\mathcal{L}$ splits the usual λ in two different constructs: a binder, able to generate a substitution, and an iterator able to erase or copy its argument. This entails a finer control of these fundamentally different issues, which are intertwined in the λ-calculus. Having a calculus which offers at the same time a lot of freedom in reduction and a lot of information about resources makes it an ideal framework to start a fresh attempt at studying reduction strategies in λ-calculi.

This paper is a first step towards a thorough study of reduction strategies for System $\mathcal{L}$. The main contributions of this paper are:

(i) we present, and compare, different ways of writing the reduction rules associated to iterators;

(ii) we define a weak reduction relation for System $\mathcal{L}$ (we call this new system *weak System* $\mathcal{L}$) similar to weak reduction used in the implementation of functional programming languages, where reduction is forbidden inside abstractions;

(iii) we present reduction strategies for the weak reduction relation: call-by-name, call-by-value, and call-by-need (emphasising this last one), proving that they are indeed strategies in a technical sense. Since neededness is usually undecidable, extra features (like sharing graphs, environments, explicit substitutions) are generally added to actually implement call-by-need. In contrast, for System $\mathcal{L}$, we can define call-by-need *within the calculus* in an *effective* way.

(iv) we give a proof of minimality of the call-by-need strategy. It is well-known that there exists no computable minimal strategy for the λ-calculus [3]. One of the main contributions in this paper is a (family of) computable minimal strategies for weak System $\mathcal{L}$.

The rest of this paper is structured as follows. In the next section we present some related work. In Section 3, we recall some background on rewriting and System $\mathcal{L}$. In Section 4 we discuss the issues about strategies and choices. In Section 5 we define weak reduction in System $\mathcal{L}$, and study different weak strategies in Section 6. Finally, Section 7 concludes the paper.

2 Related Work

In [3, Chapter 13], Barendregt defined the notion of *L-1-optimal* (or *minimal*, in this paper) strategy for the λ-calculus as a normalising strategy, minimal with respect to the length of paths in the terms reduction graph. In [3] it was shown that there exists no computable optimal L-1-strategy for the λ-calculus. One of the main contributions in this paper is a set of computable minimal strategies for weak System $\mathcal{L}$ (a version of System $\mathcal{L}$ where reduction is forbidden inside abstractions).

Weak System $\mathcal{L}$ is very much inspired in weak λ-calculi [11,12,2], weak reductions for functional programming languages [14] and lazy evaluation models [9,19]. In all these works reduction is forbidden inside abstractions and lazy evaluation is achieved by enlarging the calculus with extra syntax (graph reductions [19], explicit let bindings [2,11] or explicit heap [9]) to express sharing of subterm evaluation. In this paper we present a set of minimal strategies for weak System $\mathcal{L}$ with the same features as lazy evaluation: there is no loss of sharing except inside abstractions, and we only reduce terms that are actually used, but we insist that our definition is effective, within the calculus. This is possible due to the finer control of linear substitution and copying using the iterator, as opposed to the λ-calculus where these different issues are mixed up.

The notion of reduction in System $\mathcal{L}$, called closed reduction, is already weak in the sense that it imposes strong constraints on the application of reduction rules (see [7]). In this paper we define a weaker form of reduction: closed reduction without reduction inside abstractions. The main motivation for this further constraint is to define a simple computable minimal strategy for weak System $\mathcal{L}$.

Some previous works studied the relation between recursion and iteration in System $\mathcal{T}$ [13,4,15] showing that, in many cases, recursion is more efficient than iteration. We choose to use iteration in System $\mathcal{L}$ because it avoids the duplication of a variable, and it is then more suitable within a linear setting, such as System $\mathcal{L}$. Thus, in this paper, efficiency should be understood in this setting: a linear calculus with iteration. Another reason why we insist on using a linear discipline (and thus an iterator instead of a recursor) is that efficiency in a linear calculus, such as System $\mathcal{L}$, can be measured by the number of steps to normalise terms, because each reduction either decreases the size of the term (by β-reduction) or increases it by adding the size of the iterated function (applying the iteration reduction rule). This is no longer true in a non-linear setting, such as System $\mathcal{T}$, where the number of reduction steps cannot be used as a measure of efficiency (see [5] for a detailed discussion about the problems of naively using reduction steps as a measure of efficiency in a non-linear calculus).

In general, call-by-need (and even minimal strategies) may copy expressions in some situations (for example inside abstractions). Sharing of subterms across different instantiations of bound variables is addressed by optimal reduction strategies [10]. Although this line of research applied to System $\mathcal{L}$ is a promising one, optimal reduction in this sense is not an issue in this paper: here we follow the weak reduction approach, as is standard in the implementation of functional languages [14].

3 Background

3.1 Rewriting

Here we briefly recall some definitions, and refer the reader to [16] for more details.

Definition 3.1 An *abstract reduction system* (ARS) is a pair $(A, \to)$ where A is a set and $\to$ a binary relation on A. We write $t \to u$ if $(t, u) \in \to$. The reflexive transitive closure of $\to$ is $\to^*$, and $\leftarrow$ is the inverse relation of $\to$. A normal form is a $t \in A$ such that there exists no u such that $t \to u$. We also note $t \to^n u$ if $t \underbrace{\to \cdots \to}_{n} u$.

Definition 3.2 $\to$ is said to have the *diamond property* [1] if, whenever $u_1 \leftarrow t \to u_2$ with $u_1 \neq u_2$, there exists a v such that $u_1 \to v \leftarrow u_2$. $\to$ is said to be *confluent* if $\to^*$ has the diamond property. $\to$ is said to be *strongly normalising* if there is no object admitting an infinite $\to$-reduction path.

Definition 3.3 A *strategy* for an ARS $(A, \to)$ is a sub-ARS $(A, \to_\triangleright)$ of $(A, \to)$ (i.e. such that $\to_\triangleright \subseteq \to$) with the same normal forms.

Note that this definition is more liberal than others (e.g. [3]), in the sense that a strategy is not required to be deterministic.

Definition 3.4 A $\to$-strategy $\to_\triangleright$ is *normalising* if all $\to_\triangleright$-reduction paths starting from an object, which admits a finite $\to$-reduction path to normal form, are finite.

[1] This property is called CR[1] in [16, Ex. 1.3.18], where "diamond property" means something else.

It is *minimal* [2] if the length of any $\twoheadrightarrow$-reduction from an object a to a normal form b is minimal among all possible $\rightarrow$-reductions from a to b.

Definition 3.5 A redex is *needed* if some residual of it must be fired in any reduction to normal form.

In [18], van Oostrom gives a method to reduce the global problem of proving that a strategy is minimal (or maximal), to a verification of certain properties of local reduction diagrams. To avoid recalling all that work here, we combine some parts of Theorems 1 and 2 of [18], as the following theorem, which will be used to show the minimality of *call-by-need* among weak strategies (Theorem 6.8).

Theorem 3.6 *Let $\twoheadrightarrow$ be a $\rightarrow$-strategy. If, whenever $s \leftarrow t \rightarrow u$, either u admits an infinite $\twoheadrightarrow$-reduction or there exists an r such that $s \rightarrow^n r \leftarrow^m u$ with $n \leq m$, then $\twoheadrightarrow$ is normalising and minimal.*

3.2 System $\mathcal{L}$

In this section we recall the syntax and reduction rules of System $\mathcal{L}$ [1]. Table 1 gives the syntax of System $\mathcal{L}$. The set of linear λ-terms is built from: variables $x, y, \ldots$; linear abstraction $\lambda x.t$, where $x \in \mathsf{fv}(t)$; and application $t\,u$, where $\mathsf{fv}(t) \cap \mathsf{fv}(u) = \varnothing$. Here $\mathsf{fv}(t)$ denotes the set of free variables of t. These conditions ensure that terms are syntactically linear (variables occur exactly once in each term).

Since we are in a linear calculus, we cannot have the usual notion of pairs and projections; instead, we have pairs and splitters which use both projections, as shown on Table 1. A simple example is the swapping function (see below).

Finally, we have booleans true and false, with a linear conditional; and numbers (built from $\mathsf{0}$ and S), with a linear iterator. $\mathsf{S}^n\mathsf{0}$ denotes n applications of S to $\mathsf{0}$. [3]

The dynamics of the system is given by the set of conditional reduction rules in Table 2 (which can be seen as a higher-order membership conditional rewrite system, see [17,20]). The conditions on the rewrite rules ensure that *Beta* only applies to redexes where the argument is a closed term (which implies that α-conversion is not needed to implement substitution), and only closed functions are iterated. Table 2 gives the reduction rules for System $\mathcal{L}$, substitution is a meta-operation defined as usual. Reductions can take place in any context where the conditions are satisfied.

We give some examples to illustrate the system:

- Swapping: $\mathsf{swap} = \lambda x.\mathsf{let}\ \langle y, z\rangle = x\ \mathsf{in}\ \langle z, y\rangle$.
- Erasing numbers: although we are in a linear system, we can erase numbers by using them in iterators.

$$\begin{aligned}\mathsf{fst} &= \lambda x.\mathsf{let}\ \langle t, u\rangle = x\ \mathsf{in}\ \mathsf{iter}\ u\ t\ (\lambda z.z)\\ \mathsf{snd} &= \lambda x.\mathsf{let}\ \langle t, u\rangle = x\ \mathsf{in}\ \mathsf{iter}\ t\ u\ (\lambda z.z)\end{aligned}$$

- Copying numbers: $C = \lambda x.\mathsf{iter}\ x\ \langle \mathsf{0}, \mathsf{0}\rangle\ (\lambda x.\mathsf{let}\ \langle a, b\rangle = x\ \mathsf{in}\ \langle \mathsf{S}\ a, \mathsf{S}\ b\rangle)$ takes a number n and returns a pair $\langle n, n\rangle$.

[2] Minimality is called *L-1-optimality* in [3] and simply *optimality* in [21].

[3] Throughout this paper we use the notation $t^n u$ to denote $\underbrace{(t \cdots (t}_{n}\ u) \cdots)$.

Construction	**Variable Constraint**	**Free Variables** (fv)
$\mathsf{0}, \mathsf{true}, \mathsf{false}$	$-$	$\varnothing$
$\mathsf{S}\ t$	$-$	$\mathsf{fv}(t)$
$\mathsf{iter}\ t\ u\ w$	$\mathsf{fv}(t) \cap \mathsf{fv}(u) = \mathsf{fv}(u) \cap \mathsf{fv}(w) = \varnothing$ $\mathsf{fv}(t) \cap \mathsf{fv}(w) = \varnothing$	$\mathsf{fv}(t) \cup \mathsf{fv}(u) \cup \mathsf{fv}(w)$
x	$-$	$\{x\}$
tu	$\mathsf{fv}(t) \cap \mathsf{fv}(u) = \varnothing$	$\mathsf{fv}(t) \cup \mathsf{fv}(u)$
$\lambda x.t$	$x \in \mathsf{fv}(t)$	$\mathsf{fv}(t) \setminus \{x\}$
$\langle t, u\rangle$	$\mathsf{fv}(t) \cap \mathsf{fv}(u) = \varnothing$	$\mathsf{fv}(t) \cup \mathsf{fv}(u)$
$\mathsf{let}\ \langle x, y\rangle = t\ \mathsf{in}\ u$	$\mathsf{fv}(t) \cap \mathsf{fv}(u) = \varnothing,\ x, y \in \mathsf{fv}(u)$	$\mathsf{fv}(t) \cup (\mathsf{fv}(u) \setminus \{x, y\})$
$\mathsf{cond}\ t\ u\ w$	$\mathsf{fv}(u) = \mathsf{fv}(w), \mathsf{fv}(t) \cap \mathsf{fv}(u) = \varnothing$	$\mathsf{fv}(t) \cup \mathsf{fv}(u)$

Table 1
Terms

Name	Reduction		Condition
Beta	$(\lambda x.t)\, u$	$\longrightarrow t[u/x]$	$\mathsf{fv}(u) = \varnothing$
Let	$\mathsf{let}\ \langle x, y\rangle = \langle t, u\rangle\ \mathsf{in}\ w$	$\longrightarrow (w[t/x])[u/y]$	$\mathsf{fv}(t) = \mathsf{fv}(u) = \varnothing$
Cond	$\mathsf{cond}\ \mathsf{true}\ u\ w$	$\longrightarrow u$	
Cond	$\mathsf{cond}\ \mathsf{false}\ u\ w$	$\longrightarrow w$	
Iter	$\mathsf{iter}\ \mathsf{0}\ u\ w$	$\longrightarrow u$	$\mathsf{fv}(w) = \varnothing$
Iter	$\mathsf{iter}\ (\mathsf{S}\ t)\ u\ w$	$\longrightarrow w(\mathsf{iter}\ t\ u\ w)$	$\mathsf{fv}(t) = \mathsf{fv}(w) = \varnothing$

Table 2
Closed reduction

- Addition: $\mathsf{add} = \lambda mn.\mathsf{iter}\ m\ n\ (\lambda x.\mathsf{S}\ x)$
- Multiplication: $\lambda mn.\mathsf{iter}\ m\ \mathsf{0}\ (\mathsf{add}\ n)$
- Predecessor: $\lambda n.\mathsf{fst}(\mathsf{iter}\ n\ \langle \mathsf{0}, \mathsf{0}\rangle\ (\lambda x.\mathsf{let}\ \langle t, u\rangle = C(\mathsf{snd}\ x)\ \mathsf{in}\ \langle t, \mathsf{S}\ u\rangle))$
- Ackermann: $ack(m, n) = (\mathsf{iter}\ m\ (\lambda x.\mathsf{S}\ x)\ (\lambda gu.\mathsf{iter}\ (\mathsf{S}\ u)\ (\mathsf{S}\ \mathsf{0})\ g))\ n$

System $\mathcal{L}$ is essentially a typed calculus, and most of the properties stated in the remainder of this paper rely on this in a crucial way, although some properties are also valid in the untyped calculus (this will always be stated explicitly). We write $\Gamma \vdash_{\mathcal{L}} t : A$ if the term t has type A in the environment Γ, where A is a *linear type*: $A, B ::= \mathsf{Nat} \mid \mathsf{Bool} \mid A \multimap B \mid A \otimes B$ where Nat and Bool are the types of numbers and booleans. The full details of the type system are not essential for the remainder of this paper and are thus omitted. We recall from [1] that System $\mathcal{L}$ is confluent, reduction preserves types, typable terms are strongly normalising, and:

Theorem 3.7 (Adequacy) *If t is closed and typable, then one of the following holds:*

- $\vdash_{\mathcal{L}} t : \mathsf{Nat}$ *and* $t \to^* \mathsf{S}^n\ \mathsf{0}$ *for some integer n;*
- $\vdash_{\mathcal{L}} t : \mathsf{Bool}$ *and either* $t \to^* \mathsf{true}$ *or* $t \to^* \mathsf{false}$*;*
- $\vdash_{\mathcal{L}} t : A \multimap B$ *and* $t \to^* \lambda x.u$ *for some term u;*
- $\vdash_{\mathcal{L}} t : A \otimes B$ *and* $t \to^* \langle u, w\rangle$ *for some terms u, w.*

4 Intuitions and choices

Here we emphasise what the exact choices are when defining reduction strategies in System $\mathcal{L}$, in particular, from an efficiency point of view.

Efficiency.
Semantically, we have: $\mathsf{iter}\ (\mathsf{S}^n\mathsf{0})\ u\ w = w^n(u)$ (i.e. n copies of w applied to u). However as shown in the given rewrite rules, we actually make use of $n+1$ occurrences of w, and then throw one away. To circumvent this defect we change the definition in order to stop at $\mathsf{S}\,\mathsf{0}$ rather than $\mathsf{0}$:

$$\begin{array}{lll} \mathsf{iter}\ \mathsf{0}\ u\ w & \to u & \mathsf{fv}(w) = \varnothing \\ \mathsf{iter}\ (\mathsf{S}\,\mathsf{0})\ u\ w & \to w\,u & \mathsf{fv}(w) = \varnothing \\ \mathsf{iter}\ (\mathsf{S}(\mathsf{S}\,t))\ u\ w & \to w(\mathsf{iter}\ (\mathsf{S}\,t)\ u\ w) & \mathsf{fv}(t) = \mathsf{fv}(w) = \varnothing \end{array}$$

There is no strong motivation behind the condition on the second rule, except to ensure the conservativity of the new rules with respect to System $\mathcal{L}$. It is clear that the last two rules split the previous one, and because there are only two cases to consider in the pattern matching (S and $\mathsf{0}$) then this will not have any consequences on any of the results of System $\mathcal{L}$, which can be stated as follows:

Proposition 4.1 *Let us call* $\xrightarrow{\mathsf{old}}$ *the reduction relation defined in Section 3.2 and* $\xrightarrow{\mathsf{new}}$ *the reduction relation with the modified rules for* iter *above. Then:*

(i) *if* $t \xrightarrow{\mathsf{new}} u$, *then* $t \xrightarrow{\mathsf{old}}{}^n u$ *with* $n = 1$ *or* $n = 2$;

(ii) *if* $t \xrightarrow{\mathsf{old}} u$, *then there exists a* w *such that* $t \xrightarrow{\mathsf{old}}{}^* \xrightarrow{\mathsf{new}} w$ *and* $u \xrightarrow{\mathsf{old}}{}^* w$;

(iii) v *is a* $\xrightarrow{\mathsf{new}}$*-normal form if and only if* v *is a* $\xrightarrow{\mathsf{old}}$*-normal form;*

(iv) $\xrightarrow{\mathsf{new}}$ *is strongly normalising;*

(v) *if* v *is a normal form (for* $\xrightarrow{\mathsf{old}}$ *and* $\xrightarrow{\mathsf{new}}$*), then* $t \xrightarrow{\mathsf{new}}{}^* v$ *if and only if* $t \xrightarrow{\mathsf{old}}{}^* v$;

(vi) $\xrightarrow{\mathsf{new}}$ *is confluent.*

Proof.

(i) Straightforward.

(ii) The problem in this case is that a $\xrightarrow{\mathsf{old}}$-redex is not necessarily a $\xrightarrow{\mathsf{new}}$-redex: if we have $\mathsf{iter}\ (\mathsf{S}t)\ u\ w \xrightarrow{\mathsf{old}} w(\mathsf{iter}\ t\ u\ w)$, we know that t is closed, and by adequacy (Theorem 3.7), $t \xrightarrow{\mathsf{old}}{}^* \mathsf{S}^n\,\mathsf{0}$ for some $n \geq 0$, so that, in the case $n \geq 1$, $\mathsf{iter}\ (\mathsf{S}t)\ u\ w \xrightarrow{\mathsf{old}}{}^* \mathsf{iter}\ (\mathsf{S}^{n+1}\,\mathsf{0})\ u\ w \xrightarrow{\mathsf{new}} w(\mathsf{iter}\ (\mathsf{S}^n\,\mathsf{0})\ u\ w)$ and $w(\mathsf{iter}\ t\ u\ w) \xrightarrow{\mathsf{old}}{}^* w(\mathsf{iter}\ (\mathsf{S}^n\,\mathsf{0})\ u\ w)$, and similarly in the case $n = 0$.

(iii) Consequence of Points i and ii.

(iv) Consequence of Point i and strong normalisation: suppose we have an infinite $\xrightarrow{\mathsf{new}}$-reduction, then we obtain an infinite $\xrightarrow{\mathsf{old}}$-reduction.

(v) The "only if" part is a consequence of Point i. For the "if" part, assume $t \xrightarrow{\mathsf{old}}{}^* v$ with v a normal form. Using Point iv, consider w such that $t \xrightarrow{\mathsf{new}}{}^* w$ and w is a normal form. By the "only if" part of this point, we know that $t \xrightarrow{\mathsf{old}}{}^* w$, thus $v = w$ by the unicity of normal forms in System $\mathcal{L}$ (consequence

of the confluence of System $\mathcal{L}$).

(vi) Assume $t \xrightarrow{\mathsf{new}}{}^* u_1$ and $t \xrightarrow{\mathsf{new}}{}^* u_2$. By Point i, we also have $t \xrightarrow{\mathsf{old}}{}^* u_1$ and $t \xrightarrow{\mathsf{old}}{}^* u_2$. By confluence, there is a w such that $u_1 \xrightarrow{\mathsf{old}}{}^* w$ and $u_2 \xrightarrow{\mathsf{old}}{}^* w$. Using strong normalisation, let v be the $\xrightarrow{\mathsf{old}}$-normal form of w. Then, using Point v, $u_1 \xrightarrow{\mathsf{new}}{}^* v$ and $u_2 \xrightarrow{\mathsf{new}}{}^* v$.

□

Of course, if we are considering an untyped calculus, where non-terminating computations can be represented, then $\xrightarrow{\mathsf{old}}$ and $\xrightarrow{\mathsf{new}}$ are not equivalent as we now require forcing more evaluation to complete the pattern matching: let Δ be the term $(\lambda x.\mathsf{iter}\ (\mathsf{S}^2\mathsf{0})\ (\lambda x_1 x_2.x_1 x_2)\ (\lambda z.zx))$ and Ω be the non-terminating (untyped) term $\Delta\Delta$. Then $\mathsf{iter}\ (\mathsf{S}\ \Omega)\ I\ (\lambda x.\mathsf{iter}\ \mathsf{0}\ I\ x)$ will terminate with the old system but not with the new. In other words, iter is now more strict in its first argument. We use this version, because efficiency is now an issue. From now on, $\rightarrow$ means $\xrightarrow{\mathsf{new}}$.

Alternative iteration.
There are two ways of writing the rules for an iterator. The one given above (both $\xrightarrow{\mathsf{old}}$ and $\xrightarrow{\mathsf{new}}$) which we shall call *outer-iter* (and denote $\rightarrow_{\mathsf{out}}$) and also this one, which we shall call *inner-iter*:

$$\begin{array}{lll}
\mathsf{iter}\ \mathsf{0}\ u\ w & \rightarrow_{\mathsf{in}} u & \mathsf{fv}(w) = \varnothing \\
\mathsf{iter}\ (\mathsf{S}\,\mathsf{0})\ u\ w & \rightarrow_{\mathsf{in}} wu & \mathsf{fv}(w) = \varnothing \\
\mathsf{iter}\ (\mathsf{S}(\mathsf{S}\,t))\ u\ w & \rightarrow_{\mathsf{in}} \mathsf{iter}\ (\mathsf{S}\,t)\ (wu)\ w & \mathsf{fv}(t) = \mathsf{fv}(w) = \varnothing
\end{array}$$

We remark the relation with *fold right* and *fold left* for lists in functional programming. These operators encapsulate recursion patterns on lists, in the same way as an iterator on numbers encapsulates recursion patterns on numbers. The difference between *foldl* and *foldr* is simply the order in which the elements of the lists are accessed: left-to-right, or right-to-left. A left-to-right approach can start working on elements of lists, even infinite lists, whereas the right-to-left approach works well in the finite case (i.e. it is strict in the list). The same reasoning applies to our iterator. Of course, the origins of these operators on lists are indeed iterators on numbers (primitive recursive schemes).

With the inner-iter reduction policy, iter is strict in its first argument. For example, in an untyped calculus, if the number is not terminating, then neither is the iter (irrespectively of the evaluation order). This will not be a problem in System $\mathcal{L}$ because it is a strongly normalising calculus.

Now we have a whole collection of strategies to look at: leftmost and outermost with each of the alternatives gives different strategies. For instance, if we use inner-iter with leftmost reduction, then we get iter evaluated first. If we have outermost with outer-iter, then we compute the applications first, etc. And of course, we are interested in finding the "best" combination.

The next results show on one hand, that extending System $\mathcal{L}$ with this new form of iteration does not change the calculus itself (although it gives one more way to reduce iterators), and on the other, that both ways of reducing iterators essentially use the same number of steps when the number of iterations is known.

Lemma 4.2 *For any number $n \geq 1$, any term u and any closed term w, we have:* $\mathsf{iter}\ (\mathsf{S}^n\,\mathsf{0})\ u\ w \xrightarrow{\mathsf{out}}{}^{n}\ w^n(u) \ {}^{n}\!\xleftarrow{\mathsf{in}}\ \mathsf{iter}\ (\mathsf{S}^n\,\mathsf{0})\ u\ w$.

Proof. Straightforward by induction on n. □

Theorem 4.3 *If we add the rules corresponding to inner-iteration ($\to_{\mathsf{in}}$) to reduction rules of System $\mathcal{L}$ ($\to_{\mathsf{out}}$), we get a new system ($\to_{\mathsf{i+o}} = \to_{\mathsf{out}} \cup \to_{\mathsf{in}}$) with the following properties:*

(i) *subject reduction;*

(ii) *strong normalisation;*

(iii) *confluence;*

(iv) *the normal form of a term is the same using $\to_{\mathsf{out}}$, $\to_{\mathsf{in}}$ or $\to_{\mathsf{i+o}}$.*

Proof.

(i) *Subject reduction:* Straightforward.

(ii) *Strong normalisation:* Adapt the proof for System $\mathcal{T}$ based on reducibility [8]. Let $\nu(t)$ bound the length of every normalisation sequence beginning with t, and let $l(t)$ be the number of symbols in the normal form of t. We prove that if t, u and w are reducible, then $\mathsf{iter}\ t\ u\ w$ is reducible, by induction on $\nu(t) + \nu(w^n(u)) + \nu(w) + l(t)$, where $\mathsf{S}^n(\mathsf{0})$ is the normal form of t.

(iii) *Confluence:* Let us first note that, because of inner-iter, we lose confluence of the untyped calculus. For example

$$w(\mathsf{iter}\ \mathsf{true}\ u\ w) \xleftarrow{\mathsf{out}} \mathsf{iter}\ \mathsf{S}(\mathsf{true})\ u\ w \xrightarrow{\mathsf{in}} \mathsf{iter}\ \mathsf{true}\ (wu)\ w.$$

Now for typed terms (System $\mathcal{L}$), let us consider the only critical pair:

$$w(\mathsf{iter}\ t\ u\ w) \xleftarrow{\mathsf{out}} \mathsf{iter}\ \mathsf{S}(t)\ u\ w \xrightarrow{\mathsf{in}} \mathsf{iter}\ t\ (wu)\ v$$

Since t is closed and typable, then $t \to^* (\mathsf{S}^n\mathsf{0})$, therefore

$$\begin{array}{ccc} w(\mathsf{iter}\ t\ u\ w) & & \mathsf{iter}\ t\ (wu)\ w \\ \ast\downarrow & & \ast\downarrow \\ w(\mathsf{iter}\ (\mathsf{S}^n\mathsf{0})\ u\ w) & & \mathsf{iter}\ (\mathsf{S}^n\mathsf{0})\ (wu)\ w \\ \ast\downarrow & & \ast\downarrow \\ w(w^n(u)) & = & w^n(wu) \end{array}$$

The result follows using Newman's Lemma.

(iv) *Normal forms:* $\to_{\mathsf{out}}$ and $\to_{\mathsf{in}}$ can be seen as strategies of $\to_{\mathsf{i+o}}$, i.e. the notion of normal form is the same for the three reductions. For instance, consider a term t, v a $\to_{\mathsf{i+o}}$-normal form of t and w a $\to_{\mathsf{in}}$-normal form of t (both exist because $\to_{\mathsf{i+o}}$ is strongly normalising and $\to_{\mathsf{in}} \subset \to_{\mathsf{i+o}}$). But w is also a $\to_{\mathsf{i+o}}$-normal form of t, hence $v = w$ since $\to_{\mathsf{i+o}}$ is confluent (unicity of normal forms).

□

Iteration vs. β-reduction.
In the linear λ-calculus, it is known that all computation is useful and is used exactly once. In System $\mathcal{L}$, this is true at the level of abstraction, but we have the power of copying and erasing at the level of the iterators. We therefore claim that the choice at the level of β-reduction is inessential; what only matters is the choice in the iterator.

Here we present several reduction strategies for iterators, following their counterpart definitions for β-reductions in the λ-calculus and functional programming languages. These reduction strategies are defined for System $\mathcal{L}$ (where every term is linear), thus the only problematic reductions are in the iterator case.

Basically, we have the choice to reduce as much as possible inside iterators before firing them, or not. But we also have the choice to give the preference to outer-iter or to inner-iter. In fact, since the inner-iter reduction policy makes iter strict, it makes a lot more sense to use on one hand call-by-name and outer-iter together, and on the other call-by-value and inner-iter.

Below, we present the reduction for the iterator rule, assuming that it is properly lifted to any context.

Outer iteration by name.
Iteration by name reduces the leftmost outermost iterator first. It is closely related to Engelfriet and Schmidt's outside-in derivation for context-free grammars or first-order recursion equations [6].

$$\begin{array}{lll} \mathsf{iter}\ \mathsf{0}\ u\ w & \to u & \mathsf{fv}(w) = \varnothing \\ \mathsf{iter}\ (\mathsf{S}\,\mathsf{0})\ u\ w & \to wu & \mathsf{fv}(w) = \varnothing \\ \mathsf{iter}\ (\mathsf{S}(\mathsf{S}\,t))\ u\ w & \to w(\mathsf{iter}\ (\mathsf{S}\,t)\ u\ w) & \mathsf{fv}(t) = \mathsf{fv}(w) = \varnothing \end{array}$$

There is no syntactical constraint on w, so that outermost reduction is possible.

Inner iteration by value.
Iteration by value reduces leftmost innermost iterators first. It is closely related to Engelfriet and Schmidt's inside-out derivations.

$$\begin{array}{lll} \mathsf{iter}\ \mathsf{0}\ u\ v & \to u & \mathsf{fv}(v) = \varnothing \\ \mathsf{iter}\ (\mathsf{S}\,\mathsf{0})\ u\ v & \to vu & \mathsf{fv}(v) = \varnothing \\ \mathsf{iter}\ (\mathsf{S}(\mathsf{S}\,t))\ u\ v & \to \mathsf{iter}\ (\mathsf{S}\,t)\ (vu)\ v & \mathsf{fv}(t) = \mathsf{fv}(v) = \varnothing \end{array}$$

v is some notion of normal form (in the sequel, this notion will be that of value).

5 The Weak System $\mathcal{L}$

5.1 Weakness of System $\mathcal{L}$ reduction

Although reduction in System $\mathcal{L}$ is allowed in any context, in particular under λ-abstractions, it is already somehow weaker than usual strong reduction for the λ-calculus, due to the use of closed reduction (free variable conditions on the rules).

In particular, normal forms may still contain iterators. Note that we can however always compute the weak head normal forms of closed terms (see [1]).

We note the following:

- Normal forms of closed terms of functional type may contain iterators. For instance, $T = \lambda x.\mathsf{iter}\ (\mathsf{S}^2 0)\ I\ x$ is a normal form.
- We also remark that T could be an argument to a function, and thus values are not the normal forms we could think of, even if we allow reduction under an abstraction. In other words, there is no strategy that will always allow us to avoid copying an iterator. For instance, in $\mathsf{iter}\ (\mathsf{S}^2 0)\ (\lambda x.x)\ (\lambda x.\mathsf{iter}\ (\mathsf{S}^2 0)\ I\ x)$, the argument $\lambda x.\mathsf{iter}\ (\mathsf{S}^2 0)\ I\ x$ is a normal form, so it will be copied by the other iterator *no matter which strategy we are using.*

5.2 Weak System $\mathcal{L}$

In the λ-calculus, two views of the notion of function coexist. One of them is that functions are ordinary syntactic objects, on which we can compute. The other sees functions as abstract objects inside which it is not sensible to compute; as pieces of programs which have to wait for their argument before executing. This opposition can be seen in the following rule:

$$\frac{t \to v}{\lambda x.t \to \lambda x.v}\ (\xi)$$

This rule is part of the λ-calculus, but a strategy of the λ-calculus is free to contain it or not: in the first case, the strategy is said to be *strong*, in the second, it is *weak*. In general, weak strategies cannot reduce beyond weak head normal form, thus they are not strategies of the λ-calculus in the sense of Definition 3.3.

Weak strategies are the reductions used in functional programming languages [14]. In fact, it is more convenient to see weak strategies of the λ-calculus as strategies of a weak λ-calculus, along the lines presented in [12].

Here we present a weak version of System $\mathcal{L}$, which we call *weak System $\mathcal{L}$* with the same restriction as in ordinary weak reduction: do not reduce under abstractions, i.e. we remove the (ξ) rule. For the same reason, reduction in the second argument of let constructs should also be prohibited. We also forbid reduction inside pairs, so as to avoid computations in the first argument of let constructs that are not needed in order to reach a pair. We do allow reduction under a S, though, as well as under cond and iter. The new calculus is defined as:

Definition 5.1 The weak System $\mathcal{L}$ is the calculus with reduction $\to_w$, defined by allowing System $\mathcal{L}$ reduction $\to$ in any weak evaluation context $W[\,]$, defined as follows:

$$\begin{aligned} W[\,] ::= {} & [\,] \mid W[\,]\,t \mid t\,W[\,] \mid \mathsf{S}\,W[\,] \mid \mathsf{let}\ \langle x, y\rangle = W[\,]\ \mathsf{in}\ t \\ & \mid \mathsf{cond}\ W[\,]\ u\ w \mid \mathsf{cond}\ t\ W[\,]\ w \mid \mathsf{cond}\ t\ u\ W[\,] \\ & \mid \mathsf{iter}\ W[\,]\ u\ w \mid \mathsf{iter}\ t\ W[\,]\ w \mid \mathsf{iter}\ t\ u\ W[\,] \end{aligned}$$

There is still a lot of freedom to define strategies, in particular in the iter case.

5.3 Confluence

In the λ-calculus, it is well-known that removing the (ξ) rule leads to a non-confluent calculus, as evidenced by the following diverging pair (see e.g. [12]):

$$\lambda y.y\,(I\,I) \leftarrow (\lambda xy.y\,x)\,(I\,I) \rightarrow (\lambda xy.y\,x)\,I \rightarrow \lambda y.y\,I$$

where $I = \lambda x.x$. This has led to the introduction or use of frameworks such as supercombinators or explicit substitutions [12], which is not completely satisfactory either. This is a true problem, and we do have the same problem here. The notion of function is the same in System $\mathcal{L}$ as in the λ-calculus, hence we also obtain a non-confluent weak calculus. Thus non-confluence of weak System $\mathcal{L}$ is expected. However weak System $\mathcal{L}$ is confluent for programs: closed terms of base type.

Definition 5.2 A *program* is a closed System $\mathcal{L}$ term of type Nat or Bool.

Definition 5.3 We call *values* the normal forms for $\rightarrow_w$.

Proposition 5.4 *Closed values are the closed terms of this form:*

$$v ::= \mathsf{S}^n\,\mathsf{0} \mid \mathsf{true} \mid \mathsf{false} \mid \langle t, u\rangle \mid \lambda x.t$$

Proof. By adapting the proof of adequacy (again, this result is not valid in the untyped calculus). □

In the following, a term denoted by v will always be assumed to be a value.

Proposition 5.5 *$\rightarrow_w$ is confluent on programs.*

Proof. Assume $t \rightarrow_w^* u_1$ and $t \rightarrow_w^* u_2$, where t is of base type. Consider v_1 and v_2 the $\rightarrow_w$-normal forms of u_1 and u_2 respectively. v_1 and v_2 are values of base types, hence are also normal forms for $\rightarrow$ (using Proposition 5.4). But $\rightarrow$ is confluent, thus has the property of unicity of normal forms. We conclude $v_1 = v_2$. □

6 Weak Strategies

We are now in a position to define reduction strategies for the weak System $\mathcal{L}$ similar to known strategies for the weak λ-calculus. In this section, all strategies are weak: they perform no reduction under abstraction, and, consistently, they are defined only on closed terms. We essentially just mention call-by-name and call-by-value, while we will give more details on call-by-need, which can interestingly be defined directly in the calculus, in an operational way.

6.1 Call-by-name and call-by-value

Definition 6.1 *Call-by-name* reduction is leftmost outermost weak reduction. In particular, iteration is by name. *Call-by-value* differs from call-by-name by reducing the argument of an application $(t\ u)$ before contracting the redex and by using iteration by value instead of iteration by name.

Proposition 6.2 *Call-by-name and call-by-value are strategies of weak System $\mathcal{L}$. Moreover, call-by-name is normalising (in the untyped weak System $\mathcal{L}$).*

Proof. They are clearly strategies. In weak System $\mathcal{L}$, the leftmost outermost redex is always needed. Call-by-name reduces only needed redexes, thus it is normalising. □

Remark 6.3 Call-by-value is not normalising in the untyped calculus: recall Ω, a (untypable) term without weak head normal form. Then $\mathsf{iter}\ \mathsf{0}\ (\lambda x.x)\ \Omega$ starts an infinite reduction although the term has normal form $\lambda x.x$.

6.2 Call-by-need

Under *call-by-need* (or *lazy evaluation*), an iterated term, not in normal form, is evaluated at most once, regardless of how many times the term is iterated. Thus such an iterated term may not be duplicated (by another iterator) before it has been reduced and may be reduced only if actually used.

The standard, non operational, definition of call-by-need is: reduce the argument first (i.e. use call-by-value) if it will be needed, do not reduce it otherwise (i.e. use call-by-name). In general, it is difficult to decide if an argument will be needed or not in the syntax of the λ-calculus, and extra features are added to actually implement call-by-need (sharing graphs, environments, explicit substitutions). Here, the interesting point is that we can characterise call-by-need *within the calculus*.

Definition 6.4 Call-by-need is defined by the weak strategy (still with the liberal meaning) $\to_l$. See Table 3.

Lazy evaluation contexts:

$$\begin{aligned} L[] ::= {} & [] \mid L[]\,t \mid \mathsf{S}\,L[] \mid \mathsf{let}\ \langle x, y\rangle = L[]\ \mathsf{in}\ t \\ & \mid \mathsf{cond}\ L[]\ u\ w \mid \mathsf{cond\ true}\ L[]\ w \mid \mathsf{cond\ false}\ u\ L[] \\ & \mid \mathsf{iter}\ L[]\ u\ w \mid \mathsf{iter\ 0}\ L[]\ w \mid \mathsf{iter}\ (\mathsf{S}\,t)\ u\ L[] \end{aligned}$$

Base cases:

$$\begin{array}{rl} (\lambda x.t)\,u \to_l t[u/x] & \mathsf{fv}(u) = \varnothing \\ \mathsf{let}\ \langle x, y\rangle = \langle t, t'\rangle\ \mathsf{in}\ u \to_l u[t/x][t'/y] & \mathsf{fv}(t) = \mathsf{fv}(t') = \varnothing \\ \mathsf{cond\ true}\ u\ w \to_l u & \\ \mathsf{cond\ false}\ u\ w \to_l w & \\ \mathsf{iter\ 0}\ u\ w \to_l u & \mathsf{fv}(w) = \varnothing \\ \mathsf{iter}\ (\mathsf{S\,0})\ u\ w \to_l w\ u & \mathsf{fv}(w) = \varnothing \\ \mathsf{iter}\ (\mathsf{S}(\mathsf{S}\,t))\ u\ v \to_l v\,(\mathsf{iter}\ (\mathsf{S}\,t)\ u\ v) & \mathsf{fv}(t) = \mathsf{fv}(v) = \varnothing, v \text{ is a value} \end{array}$$

Context rule: $\dfrac{t \to_l v}{L[t] \to_l L[v]}$

Table 3
Call-by-need

Proposition 6.5 *$\to_l$ is a strategy for $\to_w$.*

Proof. It is clear that $\to_l \subset \to_w$. Moreover, the normal forms for $\to_w$ are values in the sense of Proposition 5.4, as we can replace $\to_w$ by $\to_l$ in the proof of Adequacy (Theorem 3.7). □

Proposition 6.6 *$\to_l$ reduces only needed redexes. Hence $\to_l$ is normalising and it has the same normal forms as $\to_w$ (see Proposition 5.4).*

Proof. Say that a position is needed if it is the position of a needed redex or if it is above a needed position. By induction, it is easy to see that $L[\,]$ only defines contexts where the hole $[\,]$ is in a needed position. □

Proposition 6.7 *$\to_l$ has the diamond property.*

Proof. In this proof, we simply write $\to$ for $\to_l$, and we assume that there are diverging reductions $t_1 \leftarrow_p t \to_q t_2$ at positions p and q respectively. If p and q are disjoint, the pair is joined in one step on each side by applying the other rule at the corresponding position. Otherwise, one of the position is the outermost, let's say p and write $q = p \cdot q'$. We look at all possible cases for the subterm t' at position p.

- $t' = uw$: by definition of $L[\,]$, $u \to_{q'} u'$ so $u \neq \lambda x.s$ (by Proposition 5.4), and no rule is applicable at the root of t'; this case thus does not happen.
- Similar argument for $t' = \mathsf{let}\ \langle x, y\rangle = w\ \mathsf{in}\ u$.
- $t' = \mathsf{cond}\ \mathsf{true}\ u\ w$: $u \leftarrow t' \to_{q'} \mathsf{cond}\ \mathsf{true}\ u'\ w$, then $u \to u'$, $\mathsf{cond}\ \mathsf{true}\ u'\ f \to u'$.
- Similar argument for $t' = \mathsf{cond}\ \mathsf{false}\ u\ w$ and $t' = \mathsf{iter}\ \mathsf{0}\ u\ w$.
- $t' = \mathsf{iter}\ (\mathsf{S}\,\mathsf{0})\ u\ w$: straightforward.
- $t' = \mathsf{iter}\ (\mathsf{S}(\mathsf{S}\,s))\ u\ w$: reduction at the root is allowed only when w is a value, thus the only possible innermost reduction is in s, and it is straightforward to conclude.

□

6.3 Minimality

Efficiency is a very pragmatic notion. In many cases, there is no better argument to demonstrate the efficiency of a strategy than a benchmark. On the contrary, here, System $\mathcal{L}$ gives us enough grip to actually *give a proof* of the efficiency of call-by-need. To measure efficiency, we just count the number of reduction steps; hence minimality (Definition 3.4) corresponds to the most efficient strategy. This is a more realistic notion here than in the λ-calculus, because when substitution is used, it is always linearly.

Theorem 6.8 (Minimality) *$\to_l$ is minimal, i.e. if t is a closed term, $t \to_w^m v$ and $t \to_l^n v$ where v is a value, then $n \leq m$.*

Proof. We use Theorem 3.6 (see [18] for more details on these techniques). Throughout this proof, we write $\twoheadrightarrow$ instead of $\to_l$ and $\to$ instead of $\to_w$ to improve readability. Assume $t_1 \twoheadleftarrow_p t \to_q t_2$. We want to show that there exists t_3 such that $t_1 \to^m t_3 \twoheadleftarrow^n t_2$ with $m \leq n$. If the reductions are disjoint, this is easy because redexes are preserved by disjoint reductions.

If q is above p, then the $\to_q$ step is also a $\twoheadrightarrow_q$ step (definition of $L[\,]$ and $\twoheadrightarrow$) and we may use the diamond property for $\twoheadrightarrow$, except in the case $\mathsf{iter}\ (\mathsf{S}(\mathsf{S}\,t))\ u\ w' \leftarrow \mathsf{iter}\ (\mathsf{S}(\mathsf{S}\,t))\ u\ w \to w\,(\mathsf{iter}\ (\mathsf{S}\,t)\ u\ w)$ where t and w are closed and w is not a value.

This case requires some more work. First $\mathsf{iter}\ (\mathsf{S}(\mathsf{S}\,t))\ u\ w' \to w'\,(\mathsf{iter}\ (\mathsf{S}\,t)\ u\ w')$ and $w\,(\mathsf{iter}\ (\mathsf{S}\,t)\ u\ w) \twoheadrightarrow w'\,(\mathsf{iter}\ (\mathsf{S}\,t)\ u\ w)$. Using Propositions 6.6 and 5.4, and the fact that w' is closed, we have $w' \twoheadrightarrow^k \lambda x.w''$, hence $w'\,(\mathsf{iter}\ (\mathsf{S}\,t)\ u\ w') \to^{k+1} w''[\mathsf{iter}\ (\mathsf{S}\,t)\ u\ w'/x]$ and $w'\,(\mathsf{iter}\ (\mathsf{S}\,t)\ u\ w) \twoheadrightarrow^{k+1} w''[\mathsf{iter}\ (\mathsf{S}\,t)\ u\ w/x]$. Again, $w''[\mathsf{iter}\ (\mathsf{S}\,t)\ u\ w/x] \twoheadrightarrow^n v$, where v is a value. If w is not at a needed position in $w''[\mathsf{iter}\ (\mathsf{S}\,t)\ u\ w/x]$, then the same reduction can be mimicked on $w''[\mathsf{iter}\ (\mathsf{S}\,t)\ u\ w'/x]$. Otherwise, this reduction can be decomposed as $w''[\mathsf{iter}\ (\mathsf{S}\,t)\ u\ w/x] \twoheadrightarrow^{n_1} C[w] \twoheadrightarrow C[w'] \twoheadrightarrow^{n_2} v$ with $n = n_1 + n_2 + 1$, for some context $C[\,]$, and we can mimick this reduction on $w''[\mathsf{iter}\ (\mathsf{S}\,t)\ u\ w'/x]$, omitting one step: $w''[\mathsf{iter}\ (\mathsf{S}\,t)\ u\ w'/x] \twoheadrightarrow^{n_1} C[w'] \twoheadrightarrow^{n_2} v$. In both cases, we indeed have $w''[\mathsf{iter}\ (\mathsf{S}\,t)\ u\ w'/x] \to^m v \twoheadleftarrow^n w''[\mathsf{iter}\ (\mathsf{S}\,t)\ u\ w/x]$ with $m \leq n$. This concludes this case.

If q is below p and the $\to_q$ step is not also a $\twoheadrightarrow_q$ step (otherwise, use Proposition 6.7), we look at all possible cases. There are three (by looking at the definition of $L[\,]$ and $\twoheadrightarrow$). For instance, $u \leftarrow \mathsf{cond}\ \mathsf{true}\ u\ w \to \mathsf{cond}\ \mathsf{true}\ u\ w' \twoheadrightarrow u$. The cases for $\mathsf{cond}\ \mathsf{false}\ u\ w$ and $\mathsf{iter}\ \mathsf{0}\ u\ w$ are similar. The important point is that $w\,(\mathsf{iter}\ t\ u\ v) \leftarrow \mathsf{iter}\ (\mathsf{S}\,t)\ u\ v \to \mathsf{iter}\ (\mathsf{S}\,t)\ u\ v'$ is not a case to consider (v is a normal form for $\to_w$). □

Hence, thanks to Proposition 6.7, any sub-strategy (in particular any deterministic one) of $\to_l$ will also be minimal. It is already known that call-by-need is optimal in a large class of rewrite systems, including weak λ-calculi [12]. However, our present statement is much stronger because the notion of optimality in [12] takes into account parallel reduction of family of redexes. In other words, it is assumed that there is some adequate sharing mechanism that will allow all redexes of the same family to be reduced at the same time. We should also mention that this proof is a nice illustration of using the techniques of [18]. The call-by-need strategy presented here is an effective approximation of the internal needed strategy (whose minimality for orthogonal TRSs is reproved in [18]), which retains minimality (in our system; there is no hope of a similar result in general for orthogonal TRSs).

Each iterated term is evaluated at most once and it is reduced only if actually used. It is remarkable that call-by-need is easily implementable without any syntactic extension to the calculus. Note that this does not happen with standard call-by-need, which is not expressible within the syntax of the λ-calculus: one has to extend it with some explicit binding syntax (Wadsworth graph reductions, explicitly let bindings or explicit heap) to express sharing of subterm evaluation.

7 Conclusion

System $\mathcal{L}$ is a calculus that isolates the linear and non-linear components of a computation. We have used this calculus to make a study of evaluation strategies in this context, where it is precisely the non-linear aspects of the computation that we need to control. This leads to a simple description of strategies and to a definition of minimal strategies within the calculus. Moreover, We anticipate that we can make heavy use of these results in current implementation work based around System $\mathcal{L}$.

References

[1] S. Alves, M. Fernández, M. Florido, and I. Mackie. The power of linear functions. In Z. Ésik, editor, *Proceedings of the 15th EACSL Conference on Computer Science Logic (CSL'06)*, volume 4207 of *Lecture Notes in Computer Science*, pages 119–134. Springer-Verlag, 2006.

[2] Z. M. Ariola and M. Felleisen. The call-by-need lambda calculus. *Journal of Functional Programming*, 7(3):265–301, May 1997.

[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] L. Colson and D. Fredholm. System T, call-by-value and the minimum problem. *Theor. Comput. Sci.*, 206(1-2):301–315, 1998.

[5] U. Dal Lago and S. Martini. An invariant cost model for the lambda calculus. In A. Beckmann, U. Berger, B. Löwe, and J. V. Tucker, editors, *Logical Approaches to Computational Barriers, Second Conference on Computability in Europe, CiE 2006, Proceedings*, volume 3988 of *Lecture Notes in Computer Science*, pages 105–114. Springer, 2006.

[6] J. Engelfriet and E. Schmidt. IO and OI. *Journal of Computer and Systems Sciences*, 15:328–353, 1997.

[7] M. Fernández, I. Mackie, and F.-R. Sinot. Closed reduction: explicit substitutions without alpha conversion. *Mathematical Structures in Computer Science*, 15(2):343–381, 2005.

[8] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.

[9] J. Launchbury. A natural semantics for lazy evaluation. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 144–154, Charleston, South Carolina, Jan. 1993.

[10] J.-J. Lévy. Optimal reductions in the lambda-calculus. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda-Calculus, and Formalism*, pages 159–191. Academic Press, Inc., New York, NY, 1980.

[11] J. Maraist, M. Odersky, and P. Wadler. The call-by-need lambda calculus. *Journal of Functional Programming*, 8(3):275–317, May 1998.

[12] L. Maranget. Optimal derivations in orthogonal term rerwiting systems and in weak lambda calculi. In *Proc. of the 1991 conference on Principles of Programming Languages*. ACM Press, 1991.

[13] M. Parigot. On the representation of data in lambda-calculus. In *CSL '89: Proceedings of the third workshop on Computer science logic*, pages 309–321, New York, NY, USA, 1989. Springer-Verlag New York, Inc.

[14] S. L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall International, 1987.

[15] Z. Spawski and P. Urzyczyn. Type fixpoints: iteration vs. recursion. In *ICFP '99: Proceedings of the fourth ACM SIGPLAN international conference on Functional programming*, pages 102–113, New York, NY, USA, 1999. ACM Press.

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

[17] Y. Toyama. Confluent term rewriting systems with membership. In *Proceedings of the 1st International Workshop on Conditional Term Rewriting Systems, CTRS'87, Orsay, France*, volume 308 of *Lecture Notes in Computer Science*, pages 228–241. Springer-Verlag, 1988.

[18] V. van Oostrom. Random descent. In *Proceedings of Rewriting Techniques and Applications, (RTA'07)*, Lecture Notes in Computer Science. Springer-Verlag, 2007.

[19] C. P. Wadsworth. *Semantics and Pragmatics of the Lambda-Calculus*. PhD thesis, Oxford University, 1971.

[20] J. Yamada. Confluence of terminating membership conditional TRS. In *Proceedings of the 3rd International Workshop on Conditional Term Rewriting Systems, CTRS'92, Pont-à-Mousson, France*, volume 656 of *LNCS*, pages 378–392. Springer-Verlag, 1993.

[21] N. Yoshida. Optimal reduction in weak lambda-calculus with shared environments. *Journal of Computer Software*, 11(6):3–18, Nov. 1994.

Token-passing Nets for Functional Languages

José Bacelar Almeida, Jorge Sousa Pinto, Miguel Vilaça [1]

CCTC / Departamento de Informática
Universidade do Minho
4710-057 Braga, Portugal

Abstract

Token-passing nets were proposed by Sinot as a simple mechanism for encoding evaluation strategies for the λ-calculus in interaction nets. This work extends token-passing nets to cover a typed functional language equipped with structured types and unrestricted recursion. The resulting interaction system is derived systematically from the chosen big-step operational semantics.
Along the way, we actually characterize and discuss several design decisions of token-passing nets and extend them in order to achieve simpler interaction net systems with a higher degree of embedded parallelism.

Keywords: Interaction nets, reduction strategies, λ-calculus, recursion.

1 Introduction

Interaction nets [7] constitute a Turing-complete computational paradigm, where computation is purely local, and thus (strong) confluence is obtained for free.

The linear λ-calculus can be very naturally encoded in interaction nets with just two symbols. When one drops the linearity restriction however, things become more subtle: since variable substitution is implemented within the formalism, and not as an external meta-operation, copying and erasure of terms must be dealt with explicitly. Many encodings have been studied (e.g. [9,10,8]), some of which allow for a great degree of sharing of computations.

Token-passing nets [11] were proposed as a simple mechanism for encoding λ-calculus evaluation strategies in the interaction net framework. One of the most attractive features is their simplicity, allowing computations to be easily traced in the term syntax. This makes them particularly well suited for debugging or educational purposes.

The purpose of this paper is on one hand to attempt to characterize token-passing nets as a specific class of interaction nets, and on the other hand to study their relation with the big-step operational semantics of the λ-calculus. This study

[1] Emails: {jba,jsp,jmvilaca}@di.uminho.pt

will then allows us to extend the token-passing encoding to a typed functional language with structured types and arbitrary recursion.

The paper is structured as follows: Section 2 reviews basic notions of interaction nets and encodings of λ-calculi. Section 3 is devoted to the characterization of token-passing nets. We proceed to give in Section 4 an encoding of a small functional language, and conclude with some remarks and pointers to further work.

2 Interaction Nets and λ-Calculi Encodings

An interaction net system [7] is specified by giving a set Σ of symbols, and a set $\mathcal{R}$ of interaction rules. Each symbol $\alpha \in \Sigma$ has an associated (fixed) *arity*. An occurrence of a symbol $\alpha \in \Sigma$ will be called an *agent*. If the arity of α is n, then the agent has $n+1$ *ports*: a distinguished one called the *principal port*, and n *auxiliary ports* labelled $x_1, \ldots, x_n$.

A net built on Σ is a graph (not necessarily connected) where the nodes are agents. The edges between nodes of the graph are connected to *ports* in the agents, such that there is at most one edge connected to every port in the net. Edges may be connected to two ports of the same agent. Principal ports of agents are depicted by an arrow.

The ports of agents where there is no edge connected are called the *free ports* of the net. The set of free ports define the *interface* of the net.

There are two special instances of a net: a wiring (a net containing no agents, only edges between free ports), and the empty net (containing no agents and no edges). The dynamics of Interaction Nets are based on the notion of *active pair*: any pair of agents (α, β) in a net, with an edge connecting together their principal ports. An *interaction rule* $((\alpha, \beta) \to N) \in \mathcal{R}$ replaces an occurrence of the active pair (α, β) by the net N. Rules must satisfy two conditions: the interfaces of the left-hand side and right-hand side are equal (this implies that the free ports are preserved during reduction), and there is at most one rule for each pair of agents, so there is no ambiguity regarding which rule to apply.

If a net does not contain any active pairs then we say that it is in normal form. We use the notation $\to$ for one-step reduction and $\to^*$ for its transitive reflexive closure. Additionally, we write $N \twoheadrightarrow N'$ if there is a sequence of interaction steps $N \to^* N'$, such that N' is a net in normal form. The strong constraints on the definition of interaction rules imply that reduction is strongly commutative (the one-step diamond property holds), and thus confluence is easily obtained. Consequently, any normalizing interaction net is strongly normalizing.

λ-calculi encodings

The linear λ-calculus possesses a natural encoding in interaction systems. We consider two symbols λ and @ with arity 2. Principal ports for these agents are chosen in such a way that a β-redex will correspond to an active pair in the interaction system. Figure 1 presents the term to net translation of the encoding. In short, variables become wires: a bound variable is connected to the corresponding binder (backpointer in the syntax tree), and free variables are left dangling as free ports of

Fig. 1. Linear λ-calculus: Encoding and reduction rule

the net. The reduction rule between these two agents corresponds to β-reduction (or substitution). It is captured by a proper rewiring of the free ports of the rule (also shown in Figure 1).

The simplicity of the encoding is certainly attractive and intuitive. However, when we leave the linearity assumption, things become considerably more subtle. In fact, interaction nets force an explicit treatment of the copying and erasing operations required by the non-linear use of variables. This is often accomplished by using specific agent symbols (usually denoted δ and ϵ) that implement these operations through sequences of local interactions. It is not however trivial to regulate this process in order to prevent different copying processes from interfering with each other in an unsound way.

The above mentioned difficulty motivated the proposal of complex encodings of λ-calculi in interaction nets (e.g. [1,10]). These encodings typically provide a setting were it is possible to delimit the scope of δs – the so called *boxes* (due to their close relationship to exponential boxes from Linear Logic proof-nets [4]).

3 Token-based Evaluation

It is fairly obvious how to define an interaction net system capable of performing duplication of (syntactical representations of) λ-terms. The problem is that, in such a system, redexes no longer correspond to active pairs, since syntactical representations now correspond to trees of agents (principal ports on the roots), which does not match the definition of the application agent given above. Thus the system does not capture β-reduction as interaction.

Token passing nets were introduced by Sinot [11] as a simple mechanism to encode strategies in interaction nets. The main insight behind them is that the above mentioned difficulties may be overcome as long as duplication is restricted to nets that correspond to syntactical representations of terms. Reduction strategies are encoded through the use of an *evaluation token* that traverses the syntactical representation of the term under consideration, in order to trigger computational steps in a controlled manner.

In order to make the preceding sentence meaningful, we must characterize:

(i) A well-behaved class of nets that act as "syntactic entities";

(ii) The notion of "computational step" that evaluation will trigger;

(iii) The traversal strategy for the evaluation token.

In what follows, we will provide our own characterization of token-passing nets, and

treat each of the above points in turn (to some extent, we take the opposite route from Sinot, who avoids giving an explicit characterization of this class of interaction nets). We then extend our characterization to cover different language constructs and evaluation strategies. Even though our presentation diverges in several aspects from Sinot's, we believe we remain faithful to the underlying intuitions.

Let us start by putting forward some observations:

- Token-passing nets usually restrict evaluation to closed terms. This means that evaluation never occurs under a binder (λ, for the simple λ-calculus). Note that it is easier to give an appropriate definition of syntactic-nets under the closed evaluation assumption. In this paper we restrict our attention to closed evaluation.
- Sinot imposes a linearity restriction on the evaluation token, which forces evaluation to proceed sequentially (but note that copying and erasing are not restricted in this way). We will not take sequentiality as a defining attribute of token-passing nets, since we believe that relaxing the sequentiality constraint gives rise to more natural (and also considerably simpler) encodings for the strategies considered.

3.1 Syntactical Nets

We start with a characterization of syntactical nets, used to represent the syntax of a language of terms with binders. Syntactical nets are essentially syntax trees with backpointers that associate bound variables with their corresponding binding constructors. In the following, this class of nets will be defined as the image of a translation function from terms to interaction nets.

For concreteness, we adopt a term syntax inspired by the Combinatory Reduction Systems (CRS) [6], which introduces an abstraction $[x]t$ denoting the binding of variable x in t. Taking the pure lambda-calculus as an example, we consider a binary constructor @ that makes no binding in its arguments, and a unary constructor λ that binds a variable in its argument (in higher-order abstract syntax, this constructor has type `(Term->Term)->Term`). For instance, the lambda-term $\lambda x.xx$ will be represented as $\lambda([x]@(x,x))$.

We assume the standard definitions for free and bound variables, α-equivalence and substitution. Namely (assuming Barendregt's variable convention [2]):

$$\begin{aligned} FV(c_i(t_1,\ldots,t_n)) &= FV(t_1)\cup\ldots\cup FV(t_n)\\ FV([x]t) &= FV(t)\setminus\{x\}\\ c_i(t_1,\ldots,t_n)\{u/x\} &= c_i(t_1\{u/x\},\ldots,t_n\{u/x\})\\ ([y]t)\{u/x\} &= [y](t\{u/x\}). \end{aligned}$$

The translation from terms to interaction nets is fairly obvious. To each term constructor c_i with arity n we associate an interaction net symbol with arity $n+b$, where b is the number of variables bound by c_i. The constructor c_i in the term

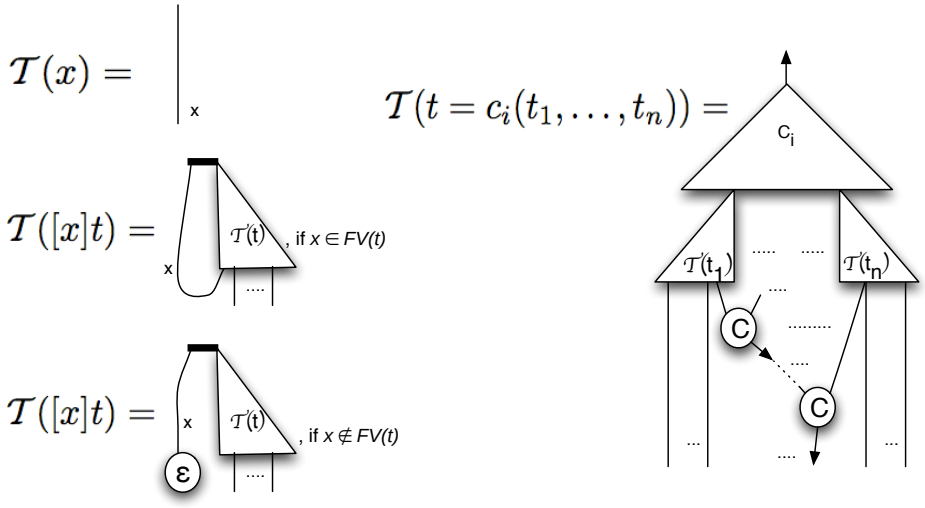
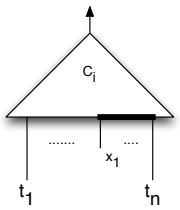


Fig. 2. Term translation

$c_i(t_1, \ldots, [x_1, \ldots]t_n)$ will be represented graphically as



where the root port (shown at the top) is the principal port of the agent, and variables bound in a term are explicitly identified graphically. This will be useful in the definition of the translation, shown in Figure 2.

Some remarks follow.

- Agents ϵ and c are introduced to account for non-linearity in terms.
- Free variables shared between arguments of term constructors are glued together with spines of c agents. Note that principal ports of these agents are oriented downwards.
- All principal ports are connected to auxiliary ports. That means that syntactical nets are in normal form.

Definition 3.1 A *syntactical net* is a net N such that $N = \mathcal{T}(t)$ for some term t.

The dynamics of agents ϵ and c is given in Figure 3, corresponding to management operations on syntactical nets: ϵ is used for erasing and c for copying. δ is a variant of c used for copying inside binders. The following properties justify the claim that syntactical nets are amenable to being manipulated by ϵ, δ and c.

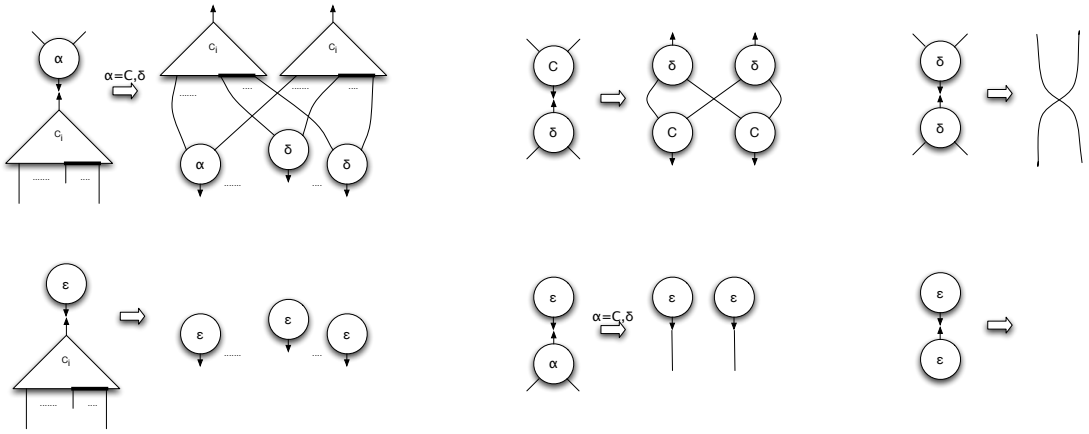
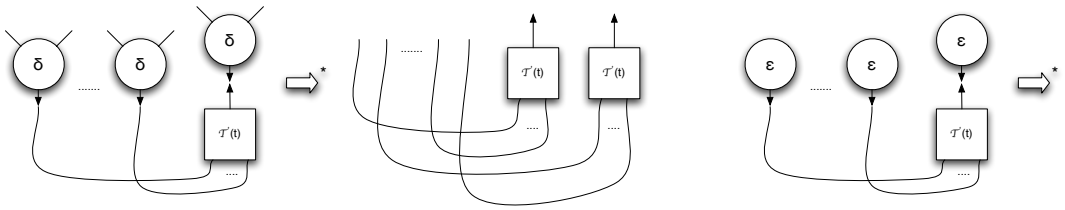


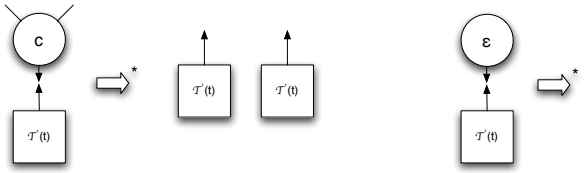
Fig. 3. Rules for agents ϵ, c and δ

Lemma 3.2 *For every open term t, the following reduction sequence exists:*



Proof. By induction on the structure of t. □

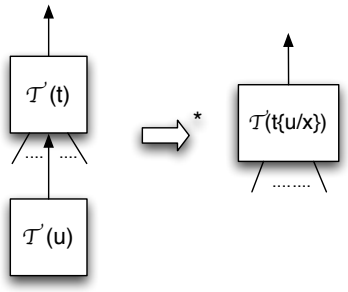
Proposition 3.3 *For every closed term t, the following reduction sequence exists:*



Proof. By induction on the structure of t and the previous result. □

As a consequence of these properties, we establish the justification of the "substitution as rewiring" slogan.

Corollary 3.4 *For every term t and closed term u we have the following.*



Proof. Direct from the definition of substitution and the previous result. □

Note that the previous proposition does not hold if u is an open term, due to a mismatch between δ and c agents that occurs when open abstractions are copied. This is a manifestation of the above mentioned interference between sharing and copying.

3.2 Computational Steps

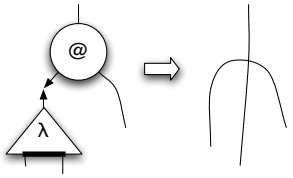
Redexes in the term language should correspond to active pairs between two term constructor agents at the level of nets .

Clearly, in order for such active pairs to occur, syntactical nets must be abandoned and a different set of agents needs to be considered for the *external* term constructors in the redexes.

Let us take the pure λ-calculus again as an example. The rewriting rule associated with the redex is (again adopting the syntax of CRS meta-terms):

$$@(\lambda([x]Z(x)), u) \Rightarrow Z(u)$$

The appropriate reduction rule is of course the standard encoding of β-reduction in interaction nets, which uses an agent for @ that can be seen as a computational, or *activated*, version of its syntactical counterpart, in the sense that it is ready for reduction.



Note that we choose to graphically represent the syntactical and the computational variants as agents with different shapes.

An important feature of the activated agents is that they are *ephemeral*, in the sense that they vanish after a single reduction (i.e. the right-hand side of these computational rules does not introduce activated agents).

3.3 Evaluation Strategies

We now turn our attention to the evaluation process. This will be triggered by interaction with a special agent called a *token*, denoted as $\Downarrow$, with arity 1. The token has its principal port oriented downwards (ready to interact with syntactical nets). The interaction rules involving the token $\Downarrow$ will determine how evaluation is performed through the syntax.

To specify an evaluation order at the term level, the so-called big-step style of operational semantics (originally proposed under the name of natural semantics [5]) is particularly well-suited, since it is essentially syntax-directed.

Let us illustrate with the call-by-value reduction strategy for the λ-calculus. The corresponding big-step rules are (where the symbol $\Rightarrow$ denotes the *evaluates-to*

relation):

$$\frac{}{\lambda x.t \Rightarrow \lambda x.t} \qquad \frac{u \Rightarrow \lambda x.t \qquad v \Rightarrow v' \qquad t\{v'/x\} \Rightarrow z}{u\ v \Rightarrow z}$$

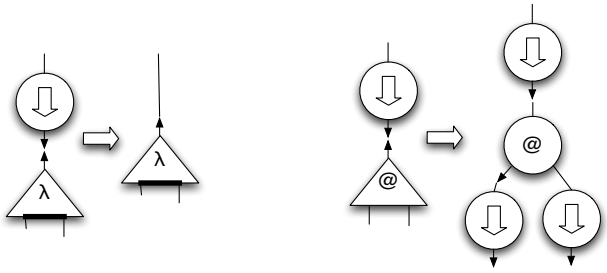
The first rule tells us that abstractions are canonical forms. For applications, the rule becomes more interesting. We should perform evaluation on both arguments; apply a β-reduction step on these results; and finally evaluate the outcome. It is instructive to make the contraction of the β-redex $(u'\,v')$ explicit in the rule:

$$\frac{u \Rightarrow u' \qquad v \Rightarrow v' \qquad u'\ v' \xrightarrow{\beta} t \qquad t \Rightarrow z}{u\ v \Rightarrow z}$$

It becomes clear that the rule actually performs two different actions: recursive calls to the evaluation relation, and a β-reduction step. As β-reduction is treated by separate rules, we only need to handle the recursion pattern of the rule. The evaluation semantics can also be expressed as term rewriting rules by introducing in our syntax a new unary constructor $\Downarrow(\cdot)$ (for evaluation) and binary constructor $\mathsf{a}(\cdot,\cdot)$ (for 'syntactical' application):

$$\begin{aligned} \Downarrow(\lambda([x]Z(x))) &\rightarrow \lambda([x]Z(x)) \\ \Downarrow(\mathsf{a}(u,v)) &\rightarrow \Downarrow(@(\Downarrow(u), \Downarrow(v))) \end{aligned}$$

These, in turn, translate directly to interaction rules. We call these *token-passing* or *evaluation* rules:



Some comments are in order here:

- Evaluation is triggered by connecting the principal port of the token $\Downarrow$ to the root of the translation of a closed term. We will denote this net by $\Downarrow(\mathcal{T}(t))$.
- Evaluation rules do not perform computational steps on their own. They just activate the syntactic agent in order to make these steps possible, and send a new $\Downarrow$ token along the principal port of the activated agent. This guarantees that a computational active pair will eventually be created.
- There should exist an evaluation rule for each term constructor. This guarantees that normal forms are syntactical nets.
- $\Downarrow$ agents are never introduced under a binding constructor, following the assumption that these terms are always canonical forms (we are thus considering weak notions of reduction).

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \qquad \frac{\Gamma, x : \sigma \vdash t : \tau}{\Gamma \vdash \lambda x.t : \sigma \to \tau} \qquad \frac{\Gamma \vdash t : \sigma \to \tau \quad \Gamma \vdash u : \sigma}{\Gamma \vdash t\ u : \tau}$$

$$\frac{}{\Gamma \vdash \mathsf{true} : \mathbb{B}} \qquad \frac{}{\Gamma \vdash \mathsf{false} : \mathbb{B}} \qquad \frac{\Gamma \vdash b : \mathbb{B} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \tau}$$

$$\frac{}{\Gamma \vdash [] : [\tau]} \qquad \frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : [\tau]}{\Gamma \vdash (e_1 :: e_2) : [\tau]} \qquad \frac{\Gamma \vdash t : [\tau] \quad \Gamma \vdash e_1 : \sigma \quad \Gamma, x : \tau, y : [\tau] \vdash e2 : \sigma}{\Gamma \vdash (\mathsf{Case}\ t\ \mathsf{of}\ [] \to e_1\ ;\ (x :: y) \to e_2) : \sigma}$$

$$\frac{\Gamma, y : \tau \vdash t : \tau}{\Gamma \vdash (\mathsf{Rec}\ y.t) : \tau}$$

Fig. 4. Typing rules for BLR

$$\frac{}{V \Rightarrow V}\ V \in \{\mathsf{true}, \mathsf{false}, [], \lambda x.t\} \qquad \frac{e_1 \Rightarrow w \quad e_2 \Rightarrow z}{e_1 :: e_2 \Rightarrow w :: z}$$

$$\frac{t \Rightarrow \mathsf{true} \quad e_1 \Rightarrow z}{\mathsf{if}\ t\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \Rightarrow z} \qquad \frac{t \Rightarrow \mathsf{false} \quad e_2 \Rightarrow z}{\mathsf{if}\ t\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \Rightarrow z}$$

$$\frac{l \Rightarrow [] \quad e_1 \Rightarrow z}{\mathsf{Case}\ l\ \mathsf{of}\ [] \to e_1\ ;\ (h :: t) \to e_2 \Rightarrow z} \qquad \frac{l \Rightarrow w :: y \quad e_2\{w/h,\ y/t\} \Rightarrow z}{\mathsf{Case}\ l\ \mathsf{of}\ [] \to e_1\ ;\ (h :: t) \to e_2 \Rightarrow z}$$

$$\frac{u \Rightarrow \lambda x.t \quad v \Rightarrow z \quad t\{z/x\} \Rightarrow w}{t\ u \Rightarrow w} \qquad \frac{f\{\mathsf{Rec}\ y.f/y\} \Rightarrow z}{\mathsf{Rec}\ y.f \Rightarrow z}$$

Fig. 5. Evaluation rules for BLR

4 Encoding of a Functional Language

We present now the encoding of a complete typed functional language. For the sake of simplicity, we choose to keep the language rather minimal – it includes types for booleans and lists, pattern-matching and general recursion (hence its name BLR). However, it should be fairly evident that the proposed encoding extends smoothly to additional types or type constructors.

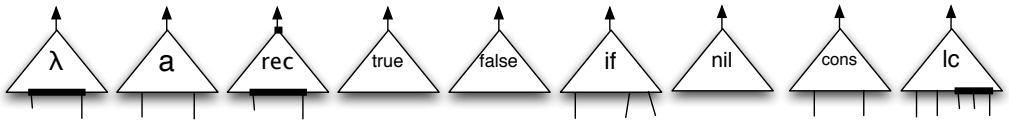
Terms and types for the language are given by the following grammar:

$$\begin{aligned}
\mathcal{T} = {} & \mathcal{V} \mid \lambda\mathcal{V}.\mathcal{T} \mid \mathcal{T}\ \mathcal{T} \mid \\
& \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ \mathcal{T}\ \mathsf{then}\ \mathcal{T}\ \mathsf{else}\ \mathcal{T} \mid \\
& [] \mid \mathcal{T} :: \mathcal{T} \mid \mathsf{Case}\ \mathcal{T}\ \mathsf{of}\ [] \to \mathcal{T}\ ;\ (\mathcal{V} :: \mathcal{V}) \to \mathcal{T} \mid \\
& \mathsf{Rec}\ \mathcal{V}.\mathcal{T} \\
\mathrm{Tp} = {} & \mathbb{B} \mid [\mathrm{Tp}] \mid \mathrm{Tp} \to \mathrm{Tp}
\end{aligned}$$

where $\mathcal{V}$ denotes a set of variables. Type assignment and evaluation rules for the language are given in Figures 4 and 5. The evaluation strategy shown is *call by value*, as visible in the application or the list-cons rule. Note however that pattern matching (*if-then-else* and *list-case* expressions) has as usual a *call by name* flavour to allow for an effective use of recursion (see, e.g. [14]).

We will now design a token-passing system for this language by addressing each of the points identified in the previous section. To encode the syntax, we shall

consider an agent for each term constructor of the language. Observe that the language includes additional binding constructors besides λ. In fact, the typing rules clearly state that the third argument of the *list-case* constructor binds two variables, and the single argument of the *rec* constructor binds one variable. Taking this into account, we are led to the following choice of syntactical agents:



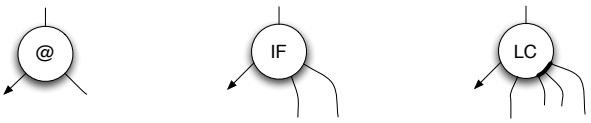
Turning now to the dynamics of the system, we start by writing a set of term-rewriting rules corresponding to the evaluation semantics given.

The evaluation rules for lists are directly translated as

$$\begin{aligned} \Downarrow(\mathsf{nil}) &\to \mathsf{nil} \\ \Downarrow(\mathsf{cons}(x,y)) &\to \mathsf{cons}(\Downarrow(x),\Downarrow(y)) \\ \Downarrow(\mathsf{lc}(l,e_1,e_2)) &\to \Downarrow(\mathsf{LC}(\Downarrow(l),e_1,e_2)) \\ \mathsf{LC}(\mathsf{nil},x,[a,b]Z(a,b)) &\to x \\ \mathsf{LC}(\mathsf{cons}(x,y),z,[a,b]Z(a,b)) &\to Z(x,y) \end{aligned}$$

Note that, in conformance with the discussion in the previous section on the application rule, the agents lc and LC are two variants of the *list-case* constructor: the former is the syntactical term constructor agent and the latter is its activated computational version. The same applies to the encoding of *if-then-else* (although this is simpler since no binding happens).

The full set of activated computational agents is thus:



The recursion rule translates directly to a rewriting rule as

$$\Downarrow(\mathsf{rec}([y]Z(y))) \to \Downarrow(Z(\mathsf{rec}([y]Z(y))))$$

The presentation of these rules in the interaction net formalism is given in Figure 6. Observe that non-linear uses of variables or meta-variables in the rules are explicitly handled in the interaction net rules by the use of the agents ϵ, c and δ. More precisely, erasing and copying of variables are handled by ϵ and c agents respectively (Lemma 3.3), and of meta-variables by ϵ and δ agents (Lemma 3.2).

The correctness of the encoding is stated in the following proposition.

Proposition 4.1 *For every closed term t,*

$$t \Rightarrow z \qquad \textit{iff} \qquad \Downarrow(\mathcal{T}(t)) \;\to^*\; \mathcal{T}(z)$$

Proof.

($\Rightarrow$) By induction on the length of the derivation of $t \Rightarrow z$:

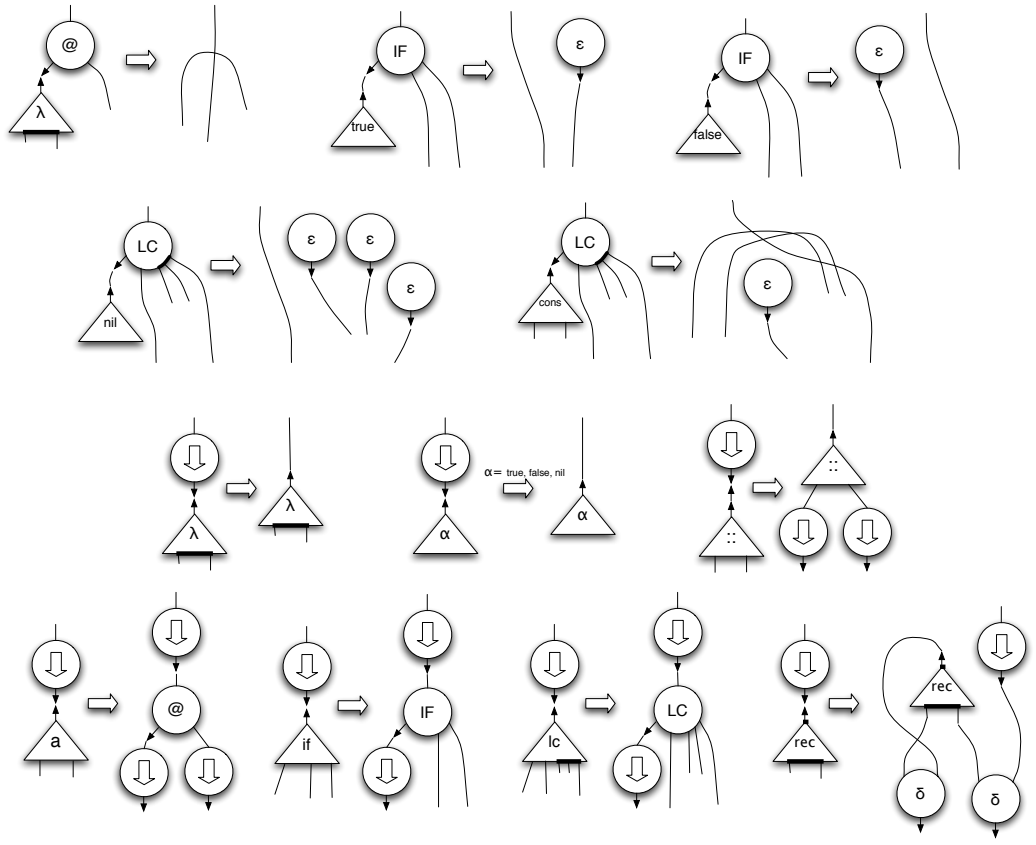


Fig. 6. Reduction rules for BLR

- If $t \in \{\lambda x.t', \mathsf{true}, \mathsf{false}, []\}$, then $t \Rightarrow t$. On the other hand, $\Downarrow(\mathcal{T}(t)) = \mathcal{T}(t)$ (by the corresponding interaction rules).
- If $t = h :: u$ then $t \Rightarrow w :: v$ where $h \Rightarrow w$ and $u \Rightarrow v$. On the other hand, $\Downarrow(\mathcal{T}(h :: u)) \rightarrow (\Downarrow(\mathcal{T}(h)) :: \Downarrow(\mathcal{T}(u)))$ by $(\Downarrow, \mathsf{cons})$-rule which reduces to $(\mathcal{T}(w) :: \mathcal{T}(v)) = \mathcal{T}(h : u)$ by induction hypothesis.
- If $t = (u\ v)$ then $t \Rightarrow z$ where $u \Rightarrow \lambda x.t'$, $v \Rightarrow v'$ and $t'\{v'/x\} \Rightarrow z$. By $(\Downarrow, \mathsf{a})$-rule, we have that $\Downarrow(\mathcal{T}(t)) \rightarrow \Downarrow(@(\Downarrow(u), \Downarrow(v)))$ which, by induction hypothesis, reduces to $\Downarrow(@(\lambda x.t', v')$. Applying $(@, \lambda)$-rule and Corollary 3.4, it reduces to $\Downarrow(t'\{v'/x\})$ that reduces to $\mathcal{T}(z)$ (again by induction hypothesis).
- If t is a pattern-matching construct: we illustrate for the case where $t = \mathsf{Case}\ l\ \mathsf{of}\ [] \rightarrow e_1\ ;\ (x::y) \rightarrow e_2$ and $l \Rightarrow u::v$ (other cases are similar). We have $t \Rightarrow z'$ where $e_2\{u/x,\ v/y\} \Rightarrow z$. Applying $(\Downarrow, \mathsf{lc})$-rule and induction hypothesis we get $\Downarrow(\mathcal{T}(t)) \rightarrow \Downarrow(\mathsf{LC}(\Downarrow(l), \mathcal{T}(e_1), [x, y]\mathcal{T}(e_2))) \rightarrow^* \Downarrow(\mathsf{LC}(\mathsf{cons}(\mathcal{T}(u), \mathcal{T}(v)), \mathcal{T}(e_1), [x, y]\mathcal{T}(e_2))$. By $(\mathsf{LC}, \mathsf{cons})$-rule and Corollary 3.4 it reduces to $\Downarrow(e_2\{u/x,\ u/y\})$ which, by induction hypothesis reduces to $\mathcal{T}(z)$.
- If $t = \mathsf{Rec}\ y.f$ then $t \Rightarrow z$ where $f\{\mathsf{Rec}\ y.f/y\} \Rightarrow z$. Applying $(\Downarrow, \mathsf{rec})$-rule, Lemma 3.2 and Corollary 3.4 it reduces to $\Downarrow(\mathcal{T}(f\{\mathsf{Rec}\ y.f/y\})$ which, by induction hypothesis, reduces to $\mathcal{T}(z)$.

($\Leftarrow$) We have that $\Downarrow(\mathcal{T}(t)) \rightarrow^* \mathcal{T}(z)$. Let us start assuming that the rules involving agents @, IF and LC) are applied only when these agents have their arguments in normal form. Such a reduction sequence always exists since we end up with

a normal form. The result for an arbitrary reduction sequence follows from the confluence property of interaction nets.

The proof proceeds by induction on the length of the reduction sequence and case analysis:

- If $t \in \{\lambda x.t', \mathsf{true}, \mathsf{false}, []\}$ then $\Downarrow(\mathcal{T}(t)) = \mathcal{T}(t)$. On the other hand $t \Rightarrow t$.
- If $t = h :: u$ then $\Downarrow(\mathcal{T}(t)) \rightarrow (\Downarrow(\mathcal{T}(h)) :: \Downarrow(\mathcal{T}(u)))$. Since we end up with a normal form, we must have that $\Downarrow(\mathcal{T}(h)) \rightarrow^* w$ and $\Downarrow(\mathcal{T}(u)) \rightarrow^* v$. By induction hypothesis and applying the appropriate evaluation rule we finally get that $\mathcal{T}(t) \Rightarrow z$.
- If $t = (u\ v)$ then $\Downarrow(\mathcal{T}(t)) \rightarrow (@(\Downarrow(\mathcal{T}(u)), \Downarrow(\mathcal{T}(u)))$. By assumption, we get $\Downarrow(\mathcal{T}(u)) \rightarrow^* \mathcal{T}(u')$ and $\Downarrow(\mathcal{T}(v)) \rightarrow^* \mathcal{T}(v')$. By induction hypothesis $\mathcal{T}(u) \Rightarrow u'$ and $\mathcal{T}(v) \Rightarrow (v')$. Well-typedness of t allow us to conclude that $z' = \lambda x.t'$ and, by $(@, \lambda)$-rule and Corollary 3.4 we get $\Downarrow(\mathcal{T}(t')\{\mathcal{T}(v')/x\}) \rightarrow^* \mathcal{T}(z)$ which, again by induction hypothesis and evaluation rule, allow us to conclude that $\mathcal{T}(t) \Rightarrow \mathcal{T}(z)$.
- If t is a pattern-matching construct: we illustrate for the case where $t = \mathsf{Case}\ l\ \mathsf{of}\ [] \rightarrow e_1\ ;\ (x::y) \rightarrow e_2$ and $l \Rightarrow u::v$ (the other cases are similar). We have that $\Downarrow((\mathsf{lc}(\mathcal{T}(l), \mathcal{T}(e_1), [x, y]\mathcal{T}(e_2)))) \rightarrow \Downarrow(\mathsf{LC}(\Downarrow(\mathcal{T}(l)), \mathcal{T}(e_1), [x, y]\mathcal{T}(e_2)))$. By assumption, $\Downarrow(\mathcal{T}(l)) \rightarrow^* \mathcal{T}(l')$ and by induction hypothesis $\mathcal{T}(l) \Rightarrow \mathcal{T}(l')$ (by well-typedness we know that l' must by either $[]$ or $u::v$ – we are now considering the second alternative). By $(\mathsf{LC}, \mathsf{cons})$-rule, Lemma 3.3 and Corollary 3.4 we get $\Downarrow(\mathcal{T}(e_2)\{\mathcal{T}(u)/x,\ \mathcal{T}(v)/y\}) \rightarrow^* \mathcal{T}(z)$ which, again by induction hypothesis and the corresponding evaluation rule give $\mathcal{T}(t) \Rightarrow \mathcal{T}(z)$.
- If $t = \mathsf{Rec}\ y.\mathcal{T}(f)$, then by $(\Downarrow, \mathsf{rec})$-rule, Lemma 3.2 and Corollary 3.4 we have that $\Downarrow(\mathcal{T}(t)) \rightarrow \Downarrow(\mathcal{T}(f)\{\mathcal{T}(\mathsf{Rec}\ y.f)/y\})$. By induction hypothesis and applying the appropriate evaluation rule we get $\mathcal{T}(t) \Rightarrow \mathcal{T}(z)$.

□

5 Extensions and Variations

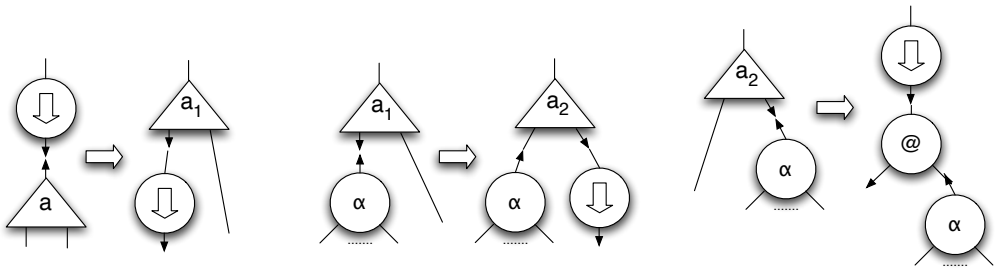
5.1 Sequentiality

As mentioned, the original presentation of token-passing nets [11] imposes sequential evaluation, with a linearity restriction on the appearance of $\Downarrow$ in the interaction rules. Let us illustrate how this approach might be incorporated in our framework.

Consider the evaluation rule for application:

$$\frac{u \Rightarrow \lambda x.t \qquad v \Rightarrow v' \qquad t\{v'/x\} \Rightarrow z}{u\ v \Rightarrow z}$$

We have seen that the corresponding interaction rule introduces three evaluation agents $\Downarrow$. In order to satisfy the linearity restriction, one should decompose the rule in several, each introducing a $\Downarrow$ in turn.



In these rules, α denotes an arbitrary agent. Note the need for additional agents (a_1 and a_2) that are analogous to a but with a different choice of principal port. These agents are responsible for scheduling the application of the several steps.

5.2 Other Evaluation Strategies

In our presentation we adopt the *call-by-value* strategy. It should be emphasized that this setting equally applies if the choice was a *call-by-name* strategy. In fact, it would be slightly simpler, as its evaluation rules are also simpler.

More refined evaluation strategies such as *call-by-need* do require additional effort. The problem lies on the fact that the corresponding big-step rules are no longer purely syntactic oriented (e.g. they incorporate evaluation contexts or side conditions). In order to capture these extended rules, we might either extend the language syntax to internalize these features, or adapt the "internals" of the interaction system to capture the intended semantics. In what follows, we show an instance of the second approach: we describe how Sinot's approach for handling call by need might be considered in our setting.

In [13], Sinot extends the token-passing approach to handle the call by need evaluation strategy. We may summarize his approach as follows:

- The translation of terms replaces the agent c by a new agent s, used to signal sharing.
- This new agent s is not a pure interaction-net agent – it possesses two principal ports and only one auxiliary port.
- The reduction rules for the new agent are given in Figure 7.

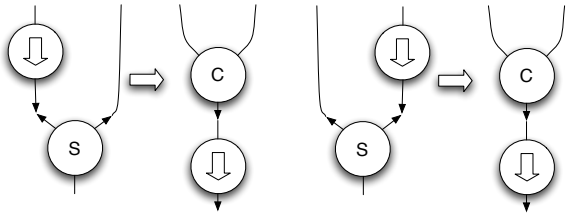


Fig. 7. Evaluation rules for the call by need strategy

Once again, the presented system is considerably simpler than the one presented in [13] due to the omission of the linearity restriction on the evaluation token. We omit a detailed proof of the correctness of this encoding (and some technicalities involving the interaction of s with δ) due to space restrictions. However, a reader familiar with [13] should have no difficulty in adapting it to this setting.

As noted in [13], the departure from the interaction net formalism does have an associated cost – confluence is no longer immediate from the locality/independence of interaction. In fact, the system actually fails to be confluent, which is harmless since non-confluence is limited to non-normalizing terms interacting with ϵ.

At this point we should also mention that the present work suggests alternatives to the handling of sharing that do not force to leave the interaction net formalism. One might replace the sharing agent s by a proper syntax constructor dup that duplicates a free occurrence of a variable x by two fresh ones y, z. Figure 8 shows an example of use of this agent and the corresponding evaluation rule.

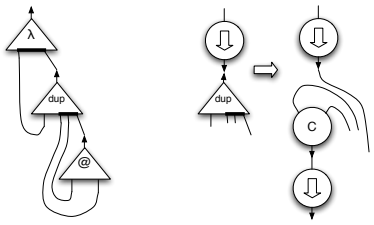


Fig. 8. Encoding of $\lambda x.x\ x$ with dup and its evaluation rule

Of course, evaluation is triggered by duplication, and we do not have any guarantees that the result will actually be required by the computation. But even if the resultant strategy is "more eager" than call by need, the more structured nature of syntactical nets might lead to greater flexibility in the encoding of complex strategies. It would be interesting to explore the applicability of this idea to encode closed reduction as defined by Fernández et al. [3].

6 Discussion and Further Work

In this paper we have studied a new system for token-passing by extending the encoding of the λ–calculus in this paradigm to a richer language, much more suited for real world programming. Although only a small fragment is presented in this paper, several other extensions have already been coded, and their great simplicity and usefulness have been proved.

The approach to token-passing nets differs in several aspects from their original presentation. Namely,

- We relax the sequential impositions of the original presentation, corresponding to a linearity constraint on the evaluation token. Sequentiality forces the existence of agents that are small variations on other agents. Removing this constraint we are led to considerably smaller and simpler interaction systems, which do without the extra agents required by the initial formulation. Since the bureaucracy of the system is removed, this formulation is also more intuitive.
- This approach also minimizes the differences between strategies; Call-by-Value and Call-by-Name systems are easily obtained from each other just by changing a few rules.
- From the methodological point of view, we avoid the "small step semantics" used in [11,13] as an intermediate step in the conversion from big-step rules to token-

passing nets, since it did not allow us to reason about management (copying and erasing) operations.

- In [12], the author deliberately avoids a characterization of token-passing nets. Here we have attempted precisely to give a characterization of these nets.

Additional features for the language and/or different evaluation strategies may be incorporated in a fairly structured manner, as shown for the case of sequential evaluation or the call-by-need strategy.

Acknowledgment

The work of the 3rd. author was funded by FCT grant SFRH / BD / 18874 / 2004.

References

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

[2] Henk P. Barendregt. *The Lambda Calculus. Its Syntax and Semantics*. Studies in Logic and the Foundations of Mathematics. Elsevier Science Publishers B.V., 1984. BAR h 84:1 1.Ex.

[3] Maribel Fernández, Ian Mackie, and François-Régis Sinot. Closed reduction: explicit substitutions without alpha-conversion. *Mathematical Structures in Computer Science*, 15(2):343–381, 2005.

[4] Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987.

[5] Gilles Kahn. Natural semantics. In *STACS 87, Proceedings of the 4th Annual Symposium on Theoretical Aspects of Computer Science, Passau, Germany, February 1987*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39. Springer, 1987.

[6] Jan Willem Klop, Vincent van Oostrom, and Femke van Raamsdonk. Combinatory reduction systems: Introduction and survey. *Theor. Comput. Sci.*, 121(1&2):279–308, 1993.

[7] Yves Lafont. Interaction nets. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages (POPL'90)*, pages 95–108. ACM Press, January 1990.

[8] Sylvain Lippi. Encoding left reduction in the lambda-calculus with interaction nets. *Mathematical Structures in Computer Science*, 12(6):797–822, 2002.

[9] Ian Mackie. YALE: Yet another lambda evaluator based on interaction nets. In *Proceedings of the 3rd International Conference on Functional Programming (ICFP'98)*, pages 117–128. ACM Press, 1998.

[10] Ian Mackie. Efficient λ-evaluation with interaction nets. In V. van Oostrom, editor, *Proceedings of the 15th International Conference on Rewriting Techniques and Applications (RTA'04)*, volume 3091 of *Lecture Notes in Computer Science*, pages 155–169. Springer-Verlag, June 2004.

[11] François-Régis Sinot. Call-by-name and call-by-value as token-passing interaction nets. In Pawel Urzyczyn, editor, *TLCA*, volume 3461 of *Lecture Notes in Computer Science*, pages 386–400. Springer, 2005.

[12] François-Régis Sinot. Call-by-need in token-passing nets. *Mathematical Structures in Computer Science*, 16(4):639–666, 2006.

[13] François-Régis Sinot. Token-passing nets: Call-by-need for free. *Electr. Notes Theor. Comput. Sci.*, 135(3):129–139, 2006.

[14] Glynn Winskel. *The Formal Semantics of Programming Languages*. Foundations of Computing. The MIT Press, 1993.