Santiago Escobar (Ed.)

# Reduction Strategies in Rewriting and Programming

10th International Workshop, WRS 2011

Novi Sad, Serbia, 29 May 2011.

Informal Proceedings

# Preface

This report contains the informal proceedings of the *10th International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2011)*, held at Novi Sad, Serbia, on May 29th, 2011. Previous editions of the workshop were held in Utrecht (2001), Copenhagen (2002), Valencia (2003), Aachen (2004), Nara (2005), Seattle (2006), Paris (2007), Hagenberg (2008), Brasilia (2009), and Edinburgh (2010); the last one as a joint workshop with the STRATEGIES workshop.

This workshop promotes research and collaboration in the area of reduction strategies. It encourages the presentation of new directions, developments, and results as well as surveys and tutorials on existing knowledge in this area. Reduction strategies define which (sub)expression(s) should be selected for evaluation and which rule(s) should be applied. These choices affect fundamental properties of computations such as laziness, strictness, completeness, and efficiency, to name a few. For this reason programming languages such as Elan, Maude, OBJ, Stratego, and TOM allow the explicit definition of the evaluation strategy, whereas languages such as Clean, Curry, and Haskell allow its modification. In addition to strategies in rewriting and programming, WRS also covers the use of strategies and tactics in other areas such as theorem and termination proving.

The Program Committee of WRS 2011 collected four reviews for each paper and held an electronic discussion during March-April 2011. The Program Committee selected 7 regular papers for presentation at the workshop. In addition to the selected papers, the scientific program includes two invited lectures by Narciso Martí-Oliet from the Universidad Complutense de Madrid, Spain, and Detlef Plump from the University of York, UK. We would like to thank them for having accepted our invitation.

We would also like to thank all the members of the Program Committee and all the referees for their careful work in the review and selection process. Many thanks to all authors who submitted papers and to all conference participants. Finally, we express our gratitude to all members of the local organization of the Federated Conference on Rewriting, Deduction, and Programming (RDP 2011).

Valencia, Spain, Santiago Escobar
April 2011 WRS 2011 Editor

# Organization

## Program Committee

| | |
|---|---|
| Dan Dougherty | Worcester Polytechnic Institute, USA |
| Santiago Escobar | Universidad Politécnica de Valencia, Spain |
| Maribel Fernández | King's College London, UK |
| Jrgen Giesl | RWTH Aachen, Germany |
| Bernhard Gramlich | Technische Universität Wien, Austria |
| Hélène Kirchner | Centre de Recherche INRIA Bordeaux, France |
| Francisco Javier López Fraguas | Universidad Complutense de Madrid, Spain |
| Salvador Lucas | Universidad Politécnica de Valencia, Spain |
| Aart Middeldorp | University of Innsbruck, Austria |
| Jaco van de Pol | University of Twente, The Netherlands |
| Masahiko Sakai | Nagoya University, Japan |
| Manfred Schmidt-Schauss | Johann Wolfgang Goethe-Universität, Germany |

## Sponsoring Institutions

Departamento de Sistemas Informáticos y Computación (DSIC)
Universidad Politécnica de Valencia (UPV)

# Table of Contents

The Maude strategy language and some of its applications . . . . . 1
*Narciso Martí-Oliet*

Graph Programs: Semantics, Verification and Implementation . . . . . 3
*Detlef Plump*

Closed reduction strategies in a linear lambda calculus with recursion . . . . . 5
*Sandra Alves, Maribel Fernandez, Mario Florido and Ian Mackie*

Constrained beta reduction . . . . . 11
*Sandra Alves and Ian Mackie*

Lazy Term Rewriting Modulo Associativity and Commutativity . . . . . 17
*Walid Belkhir and Alain Giorgetti*

Strategies for Decreasingly Confluent Rewrite Systems . . . . . 23
*Nao Hirokawa and Aart Middeldorp*

Application of monadic substitution to recursive type containment . . . . . 29
*Vladimir Komendantsky*

Productivity of Non-Orthogonal Term Rewrite Systems . . . . . 35
*Matthias Raffelsieper*

Strategy Independent Reduction Lengths in Rewriting and Binary Arithmetic . . . . . 41
*Hans Zantema*

# The Maude strategy language and some of its applications

Narciso Martí-Oliet
Facultad de Informática, Universidad Complutense de Madrid
narciso@esi.ucm.es

**Abstract**

Maude is a declarative specification and programming language based on rewriting logic. A Maude rewrite theory essentially consists of a signature, a set of equations, and a set of (conditional) rewrite rules. While the set of equations is assumed to be confluent and terminating, so that every term has a normal form with respect to equational reduction, rewrite rules can be nonconfluent or nonterminating.

A strategy is described as an operation that, when applied to a given term, produces a set of terms as a result, given that the process is nondeterministic in general. The basic strategies consist of the application of a rule to a given term, but rules can be conditional with respect to some rewrite conditions which in turn can also be controlled by means of strategies. Those basic strategies are combined by means of several combinators, including: regular expression constructions (concatenation, union, and iteration), if-then-else, combinators to control the way subterms of a given term are rewritten, and recursion. Furthermore, strategies can also be generic.

This strategy language has been endowed with both a simple set-theoretic semantics and a rewriting semantics, which are sound and complete with respect to the controlled rewrites. Moreover, the rewriting semantics also enjoys properties of monotonicity and persistence.

The first applications of the strategy language included the operational semantics of process algebras such as CCS and the ambient calculus. The same ideas were later applied to the two-level operational semantics of the parallel functional programming language Eden and to modular structural operational semantics. It has also been used in the implementation of basic Knuth-Bendix completion algorithms and of congruence closure algorithms. Other applications include a sudoku solver, neural networks, membrane systems, hierarchical design graphs, and a representation of BPMN.

# Graph Programs: Semantics, Verification and Implementation

Detlef Plump
The University of York
det@cs.york.ac.uk

**Abstract**

GP is a rule-based, nondeterministic programming language for high-level problem solving in the domain of graphs, freeing programmers from handling low-level data structures. The language has a simple syntax and semantics, to facilitate formal reasoning about programs. In this talk, I will introduce GP by a number of example programs and discuss its structural operational semantics. A special feature of this semantics is the use of failing computations to define branching and iteration. I will then present proof rules for Hoare-style verification of graph programs, and demonstrate their use by verifying a simple colouring program. Finally, I will briefly describe GP's current implementation.

# Closed reduction strategies in a linear lambda calculus with recursion

Sandra Alves*
DCC-FCUP & LIACC
University of Porto

Maribel Fernández
Department of Informatics
King's College London

Mário Florido
DCC-FCUP & LIACC
University of Porto

Ian Mackie
LIX, CNRS UMR 7161
École Polytechnique

**Abstract**

We define linear versions of the $\lambda$-calculus with bounded and unbounded recursion. In order to preserve the linearity of the calculus, two different approaches can be used to define the reduction rules for recursors: either the function to be iterated is closed when the recursor is built, or it is closed at the time of reduction. We analyse the power of linear calculi with closed-construction and closed-reduction. Although in the case of linear calculi with bounded recursors closed reduction and closed construction capture different classes of functions, we show that both closed construction and closed reduction yield Turing-complete systems in the presence of unbounded recursion.

## 1 Introduction

Over the last years, several linear calculi have been proposed with the aim of performing program analyses: for instance, strictness analysis, pointer analysis, resource analysis (see, e.g., [6, 8, 9, 22, 20, 21, 19, 16]). Linearity is a property exploited by compilers to optimise code, and it has applications in other domains, such as in the area of quantum computation, concurrency and communication [17, 15, 18, 23], and hardware compilation [12].

In this paper we focus on functional computations. We study two linear versions of the $\lambda$-calculus defined in previous work: in one of the systems, called System $\mathcal{L}$ [3], the linear $\lambda$-calculus is extended with a bounded recursor, and in the other, called System $\mathcal{L}_{\mathsf{rec}}$ [4], with an unbounded recursor. In order to preserve the linearity of the calculus, two different approaches can be used to define the reduction rules for recursors: either the function to be iterated should be closed when the recursor is built (this is called closed construction), or it should be closed at the time of reduction (this is called closed reduction). Closed reductions have interesting properties: in addition to offering efficient reduction strategies, applications such as new calculi of explicit substitution were obtained; see [11] for more details. It was shown in [3] that System $\mathcal{L}$ has all the computation power of Gödel's System $\mathcal{T}$, despite being a syntactically linear system. However, if a closed-construction approach is used in a linear $\lambda$-calculus with bounded recursion then the resulting system is less powerful: only primitive recursive functions can be defined (see [7, 2]). Thus, in the presence of linearity and bounded recursion, closed construction and closed reduction yield radically different computational power.

System $\mathcal{L}_{\mathsf{rec}}$ [4] has an unbounded recursor with a closed reduction strategy. Although linear, it is Turing complete. The main contribution of this paper is a comparison of the closed construction and closed reduction approaches for linear calculi with unbounded recursion. We define a version of System $\mathcal{L}_{\mathsf{rec}}$ with closed construction. Surprisingly, although in the case of System $\mathcal{L}$ closed-reduction and closed-construction capture different classes of functions, for System $\mathcal{L}_{\mathsf{rec}}$ both closed-construction and closed-reduction yield Turing-complete systems.

*Partially supported by funds granted to *LIACC* through the *Programa de Financiamento Plurianual, Fundação para a Ciência e Tecnologia* and *FEDER/POSI*.

## 2 Background: System $\mathcal{L}$ and System $\mathcal{L}_{\mathsf{rec}}$

We assume the reader is familiar with the $\lambda$-calculus [5]. In this section we recall the definition of System $\mathcal{L}$ [3], a linear version of Gödel's System $\mathcal{T}$ (for details on the latter see [14]), and System $\mathcal{L}_{\mathsf{rec}}$, an extension of the linear $\lambda$-calculus with unbounded recursion [4].

The terms of System $\mathcal{L}$ are obtained by extending the terms of the linear $\lambda$-calculus [1] with numbers, pairs, and an iterator, whereas in System $\mathcal{L}_{\mathsf{rec}}$ instead of an iterator we use an unbounded recursor.

Linear $\lambda$-terms $t, u, \ldots$ are inductively defined by: $x \in \Lambda$, $\lambda x.t \in \Lambda$ if $x \in \mathsf{fv}(t)$, and $tu \in \Lambda$ if $\mathsf{fv}(t) \cap \mathsf{fv}(u) = \varnothing$. Note that $x$ is used at least once in the body of the abstraction, and the condition on the application ensures that all variables are used at most once. Thus these conditions ensure syntactic linearity (variables occur exactly once). In System $\mathcal{L}$ we also have numbers generated by 0 and $\mathsf{S}$, with an iterator and pairs:

$$\mathsf{iter}\ t\ u\ v \quad \text{if } \mathsf{fv}(t) \cap \mathsf{fv}(u) = \mathsf{fv}(u) \cap \mathsf{fv}(v) = \mathsf{fv}(v) \cap \mathsf{fv}(t) = \varnothing$$

$$\begin{array}{ll} \langle t, u \rangle & \text{if } \mathsf{fv}(t) \cap \mathsf{fv}(u) = \varnothing \\ \mathtt{let}\ \langle x, y \rangle = t\ \mathtt{in}\ u & \text{if } x, y \in \mathsf{fv}(u) \text{ and} \\ & \mathsf{fv}(t) \cap (\mathsf{fv}(u) - \{x, y\}) = \varnothing \end{array}$$

Since $\lambda$ and `let` are binders, terms are defined modulo $\alpha$-equivalence as usual.

System $\mathcal{L}_{\mathsf{rec}}$ is also an extension of the linear $\lambda$-calculus, with numbers and pairs, but instead of an iterator working on numbers, $\mathcal{L}_{\mathsf{rec}}$ has a recursor:

$$\mathsf{rec}\ t_1\ t_2\ t_3\ t_4 \qquad \text{if } \mathsf{fv}(t_i) \cap \mathsf{fv}(t_j) = \varnothing, \text{for } i \neq j$$

**Closed reduction.** Both System $\mathcal{L}$ and System $\mathcal{L}_{\mathsf{rec}}$ use a closed reduction strategy, which waits for arguments to become closed before firing redexes. Closed-reduction is a strategy for cut-elimination for linear logic, introduced by Jean-Yves Girard in [13]. Recasting the ideas to the $\lambda$-calculus was done in [10], and the result is a naturally efficient reduction strategy. In the $\lambda$-calculus, closed reduction avoids $\alpha$-conversion while allowing reductions inside abstractions (in contrast with standard weak strategies), thus achieving more sharing of computation. When applied to System $\mathcal{L}$ and System $\mathcal{L}_{\mathsf{rec}}$, it imposes certain conditions on reduction rules; in particular iterated functions should be closed. The intuition here is that we should only copy closed terms because then all the resources are there. In linear logic words, we can promote a term that is closed.

The reduction rules for System $\mathcal{L}$ and System $\mathcal{L}_{\mathsf{rec}}$ are given in Table 1: the first four rules define reduction for System $\mathcal{L}$, and in $\mathcal{L}_{\mathsf{rec}}$ the rules for *Iter* are replaced by the rules for *Rec*. Substitution is a meta-operation defined as usual, and reductions can take place in any context.

The rules for the recursor $Rec_1$ and $Rec_2$ allow us to simulate a bounded iterator. Note that the *Rec* rules pattern-match on a pair of numbers (the usual bounded recursor works on a single number). This feature is used to represent both bounded and unbounded recursion with the same operator (as the examples below illustrate). An alternative would be to have an extra parameter of type $\mathsf{N}$ in the recursor.

Both System $\mathcal{L}$ and System $\mathcal{L}_{\mathsf{rec}}$ are confluent and closed by reduction, but the untyped calculi are not strongly normalising: it is possible to encode linearly the infinite computation given by $\Omega \equiv (\lambda x.xx)(\lambda x.xx)$ (e.g., encode $\lambda x.xx$ as $\lambda x.\mathsf{iter}\ S^2 0\ (\lambda xy.xy)\ (\lambda y.yx)$). A type system for System $\mathcal{L}$ was given in [3], and the typed version of the calculus proved to be strongly normalising. Types are also preserved by reduction.

| Name | Reduction | | | Condition |
|---|---|---|---|---|
| *Beta* | $(\lambda x.t)v$ | $\to$ | $t[v/x]$ | $\mathsf{fv}(v) = \varnothing$ |
| *Let* | $\mathtt{let}\ \langle x, y\rangle = \langle t, u\rangle\ \mathtt{in}\ v$ | $\to$ | $(v[t/x])[u/y]$ | $\mathsf{fv}(t) = \mathsf{fv}(u) = \varnothing$ |
| $Iter_1$ | $\mathsf{iter}\ (\mathsf{S}\ t)\ u\ v$ | $\to$ | $v(\mathsf{iter}\ t\ u\ v)$ | $\mathsf{fv}(v) = \varnothing$ |
| $Iter_2$ | $\mathsf{iter}\ 0\ u\ v$ | $\to$ | $u$ | $\mathsf{fv}(v) = \varnothing$ |
| $Rec_1$ | $\mathsf{rec}\ \langle 0, t'\rangle\ u\ v\ w$ | $\to$ | $u$ | $\mathsf{fv}(t'vw) = \varnothing$ |
| $Rec_2$ | $\mathsf{rec}\ \langle \mathsf{S}\ t, t'\rangle\ u\ v\ w$ | $\to$ | $v(\mathsf{rec}\ (w\langle t, t'\rangle)\ u\ v\ w)$ | $\mathsf{fv}(vw) = \varnothing$ |

Table 1: Closed reduction rules

In the same spirit, one can define types for $\mathcal{L}_{\mathsf{rec}}$, however, since System $\mathcal{L}_{\mathsf{rec}}$ is a Turing complete system, even typed terms can be non-normalising. For example:
$\mathsf{rec}\ \langle \mathsf{S}\ 0, 0\rangle\ I\ I\ \lambda x.\mathtt{let}\ \langle y, z\rangle = x\ \mathtt{in}\ \langle \mathsf{S}y, z\rangle$ does not have a normal form.

**Closed construction.** The closed reduction strategy waits, to reduce an iterator/recursor term, until the iterated functions are closed. One can ask a stronger constraint on the construction of terms, that is, to constrain iterators/recursors to be closed on construction (i.e., we have a syntactical constraint that only terms without free variables are used in this context). For System $\mathcal{L}$, one imposes an extra condition on the iterated function $v$, when defining iterators:

$$\mathsf{iter}\ t\ u\ v \qquad \text{if } \mathsf{fv}(t) \cap \mathsf{fv}(u) = \varnothing \text{ and } \mathsf{fv}(v) = \varnothing$$

For System $\mathcal{L}_{\mathsf{rec}}$, one imposes an extra condition on $v$ and $w$:

$$\mathsf{rec}\ t\ u\ v\ w, \qquad \text{if } \mathsf{fv}(t) \cap \mathsf{fv}(u) = \varnothing \text{ and } \mathsf{fv}(vw) = \varnothing$$

In [7], Dal Lago defines a linear $\lambda$-calculus with bounded iteration following the closed construction approach, and proves that the system encodes exactly the set of primitive recursive functions. A similar system allowing iterators to be open at construction, but imposing a closed condition on reduction, allows to encode more than the primitive recursive functions, and in particular allows the encoding of the Ackermann function, as shown in [2].

## 3 The computational power of System $\mathcal{L}$ and System $\mathcal{L}_{\mathsf{rec}}$

Both System $\mathcal{L}$ and System $\mathcal{L}_{\mathsf{rec}}$ follow the closed-reduction approach. In [3], several results on the computational power of System $\mathcal{L}$ were stated and proved, in particular, System $\mathcal{L}$ can encode the primitive recursive functions (as it includes Dal Lago's system [7]). For example, consider the second projection function $\pi_2$ ($\pi_1$ is defined similarly) and a function $C$ to copy numbers:

$$\pi_2 \quad = \quad \lambda x.\mathtt{let}\ \langle a, b\rangle = x\ \mathtt{in}\ \mathsf{rec}\ \langle a, 0\rangle\ b\ I\ I$$

$$C \quad = \quad \lambda x.\mathsf{rec}\ \langle x, 0\rangle\ \langle 0, 0\rangle\ (\lambda x.\mathtt{let}\ \langle a, b\rangle = x\ \mathtt{in}\ \langle \mathsf{S}a, \mathsf{S}b\rangle)\ I$$

In fact, in System $\mathcal{L}$ it is possible to encode a much more powerful class of functions: Gödel's System $\mathcal{T}$.

Regarding System $\mathcal{L}_{\mathsf{rec}}$'s computational power, note that the term $\mathsf{rec}\ \langle t, 0\rangle\ u\ v\ (\lambda x.x)$ encodes System $\mathcal{L}$'s iterator $\mathsf{iter}\ t\ u\ v$: it has the same type and has the same normal form. Therefore, System $\mathcal{L}_{\mathsf{rec}}$ also encodes the full System $\mathcal{T}$. Furthermore, we can also encode the minimisation

operator $\mu_f$ used to define partial recursive functions. Recall that if $f : \mathbb{N} \to \mathbb{N}$ is a total function on natural numbers, $\mu_f = \min\{x \in \mathbb{N} \mid f(x) = 0\}$. If we consider $\overline{f}$ to be a closed $\lambda$-term in $\mathcal{L}_{\mathsf{rec}}$ representing a total function $f$ on natural numbers, then

$$M = \mathsf{rec}\ \langle \overline{f}0, 0\rangle\ 0\ (\lambda x.\mathsf{S}(x))\ F$$

where $F = \lambda x.\mathtt{let}\ \langle y, z\rangle = C(\pi_2 x)\ \mathtt{in}\ \langle \overline{f}(\mathsf{S}y), \mathsf{S}z\rangle$, is an encoding of $\mu_f$. Recall that System $\mathcal{L}$ encodes all the primitive recursive functions, therefore so does System $\mathcal{L}_{\mathsf{rec}}$. If we add to that the functions obtained from the primitive recursive functions by minimisation, we obtain a Turing complete system.

## 4 Closed reduction/closed construction and iteration

Imposing a closed-at-construction restriction on iterators clearly has an impact in the presence of linearity. For Dal Lago's system [7], it makes the difference from encoding exactly the set of primitive recursive functions, or being able to encode more powerful functions, such as the Ackermann function.

For Gödel's System $\mathcal{T}$, the fact that we do not allow iterators to be open at construction, does not affect the set of definable functions. If we define $v = \lambda xy.y(xy)$, then each iterator term $\mathsf{iter}\ n\ b\ f$ in System $\mathcal{T}$, where $f$ may be an open term, can be translated into the typable term $(\mathsf{iter}\ n\ (\lambda x.b)\ v)f$, where $x \notin \mathsf{fv}(b)$. It is easy to see that $\mathsf{iter}\ n\ b\ f$ and $(\mathsf{iter}\ n\ (\lambda x.b)\ v)f$ have the same normal form $f(\ldots(fb))$. It is worth remarking that we rely on a non-linear term $v$ to get this result. Indeed, iterating $v$ is essentially equivalent to constructing a Church numeral.

For a linear system with iteration such as System $\mathcal{L}$, although some functions are naturally defined using an open function, for example: $\mathsf{mult} = \lambda mn.\mathsf{iter}\ m\ 0\ (\mathsf{add}\ n)$, one can encode them using a closed-at-construction iteration. In general, an iterator with an open function where the free variables are of type $\mathsf{N}$ can be encoded using a closed-at-construction iterator, as follows. Consider $\mathsf{iter}\ t\ u\ v$, where $v$ is open, for free variables $x_1, \ldots, x_n$ of type $\mathsf{N}$. Then let
$F \equiv \mathtt{let}\ Cx'_1 = \langle x_1, x''_1\rangle\ \mathtt{in} \ldots \mathtt{let}\ Cx_k = \langle x_k, x''_k\rangle\ \mathtt{in}\ \langle vx_0, x''_1, x''_k\rangle$ and
$W \equiv \lambda x.\mathtt{let}\ x = \langle x_0, x'_1, x'_k\rangle\ \mathtt{in}\ F$.
Then we simulate $\mathsf{iter}\ t\ u\ v$ using a closed iterator as follows: $\pi_1(\mathsf{iter}\ t\ \langle u, x_1, x_k\rangle\ W)$.

This technique can also be applied to open functions where the free variables are of type $\tau$, for $\tau$ generated by the following grammar: $\tau ::= \mathsf{N} \mid \tau \otimes \tau$. More generally, open functions where the free variables have base type can be encoded when we consider iteration closed-at-construction.

## 5 Closed construction and unbounded recursion

We now consider what happens when we use the closed-at-construction approach in a linear system with unbounded recursion such as System $\mathcal{L}_{\mathsf{rec}}$.

Notice that the encoding of $\mu_f$ in $\mathcal{L}_{\mathsf{rec}}$ given above is a term closed-at-construction. Since all the primitive recursive functions are definable using closed-at-construction iterators, which are trivially encoded using closed $\mathcal{L}_{\mathsf{rec}}$ recursors, we conclude that imposing a closed-at-construction condition on System $\mathcal{L}_{\mathsf{rec}}$ still gives a Turing complete system.

Note however that, although System $\mathcal{L}_{\mathsf{rec}}$ can encode all the computable functions, that does not mean one can encode all the computational behaviours. For example for any closed function $f$, one can encode in $\mathcal{L}_{\mathsf{rec}}$ a term $Y$, such that, $Yf \to f(Yf)$. However, this relies on the fact that one can copy any closed function $f$, which can be done both in System $\mathcal{L}$ and System $\mathcal{L}_{\mathsf{rec}}$ with closed reduction, but so far there is no known encoding when one imposes a closed-at-construction condition.

# References


[1] S. Abramsky. Computational Interpretations of Linear Logic. *Theoretical Computer Science*, 111:3–57, 1993.

[2] S. Alves, M. Fernández, M. Florido, and I. Mackie. The power of closed reduction strategies. *ENTCS*, 174(10):57–74, 2007.

[3] S. Alves, M. Fernández, M. Florido, and I. Mackie. Gödel's system $\mathcal{T}$ revisited. *Theor. Comput. Sci.*, 411(11-13):1484–1500, 2010.

[4] S. Alves, M. Fernández, M. Florido, and I. Mackie. Linearity and recursion in a typed $\lambda$-calculus. Submitted, 2011.

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

[6] G. Boudol, P.-L. Curien, and C. Lavatelli. A semantics for lambda calculi with resources. *MSCS*, 9(4):437–482, 1999.

[7] U. Dal Lago. The geometry of linear higher-order recursion. In *Proc. Logic in Computer Science (LICS'05)*, pages 366–375, June 2005.

[8] T. Ehrhard and L. Regnier. The differential lambda-calculus. *Theor. Comput. Sci.*, 309(1-3):1–41, 2003.

[9] T. Ehrhard and L. Regnier. Uniformity and the Taylor expansion of ordinary lambda-terms. *Theor. Comput. Sci.*, 403(2-3):347–372, 2008.

[10] M. Fernández and I. Mackie. Closed reduction in the $\lambda$-calculus. In *Proceedings of Computer Science Logic (CSL'99)*, volume 1683 of *LNCS*, pages 220–234. Springer-Verlag, September 1999.

[11] M. Fernández, I. Mackie, and F.-R. Sinot. Closed reduction: explicit substitutions without alpha conversion. *MSCS*, 15(2):343–381, 2005.

[12] D. R. Ghica. Geometry of synthesis: a structured approach to VLSI design. In *POPL*, pages 363–375, 2007.

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

[14] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge Tracts in Theor. Comp. Sci. Cambridge University Press, 1989.

[15] M. Giunti and V. T. Vasconcelos. A linear account of session types in the pi calculus. In *CONCUR*, pages 432–446, 2010.

[16] M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *POPL*, pages 185–197, 2003.

[17] K. Honda. Types for dyadic interaction. In *CONCUR'93*, volume 715 of *LNCS*, pages 509–523. Springer, 1993.

[18] N. Kobayashi. A partially deadlock-free typed process calculus. *ACM Trans. Program. Lang. Syst.*, 20(2):436–482, 1998.

[19] E. Nöcker, J. Smetsers, M. van Eekelen, and M. Plasmeijer. Concurrent clean. In *PARLE'91*, volume 506 of *LNCS*, pages 202–219. Springer, 1991.

[20] P. Wadler. Linear types can change the world! In *IFIP TC 2 Conf. on Progr. Concepts and Methods*, pages 347–359. North Holland, 1990.

[21] D. Walker. Substructural type systems. In *Adv. Topics in Types and Progr. Languages*, chapter 1, pages 3–43. MIT Press, Cambridge, 2005.

[22] K. Wansbrough and S. P. Jones. Simple usage polymorphism. In *Proc. ACM SIGPLAN Workshop on Types in Compilation*. ACM Press, 2000.

[23] N. Yoshida, K. Honda, and M. Berger. Linearity and bisimulation. In *FoSSaCS*, LNCS, pages 417–434. Springer-Verlag, 2002.

# Constrained beta reduction

Sandra Alves
DCC-FCUP & LIACC
University of Porto

Ian Mackie
LIX, CNRS UMR 7161
École Polytechnique

**Abstract**

In this paper we investigate reduction strategies for the lambda calculus by placing conditions on the usual beta rule. This provides a framework that can be instantiated to a number of interesting cases. We begin our study with the notion of typed reduction, in which reduction can only take place if some typing constraint is verified on the redex, irrespective of whether the whole term is typable or not.

## 1 Introduction

The $\lambda$-calculus is defined as a set $\Lambda$ containing terms from the grammar:

$$t, u ::= x \mid tu \mid \lambda x.t$$

together with the $\beta$-reduction rule:

$$(\lambda x.t)u \to_\beta t[u/x]$$

For any term $t$, there may be a number of redexes (possible $\beta$-reductions) and the order in which these are reduced dictates a strategy for the reduction. Typical examples include always reducing the left-most or always reducing the inner-most redex. Strategies may also be weak, meaning that reduction is not permitted in certain contexts. Usually this means that reduction is forbidden under an abstraction, which gives the lazy lambda calculus [1].

In this paper we take an alternative point of view, which is orthogonal to the usual notion of strategy: we make the $\beta$ rule conditional. This still leaves notions of strategy possible (left-most, inner-most, etc.). However, in this paper we just look at what happens when we perform this restriction. We also investigate constrained reduction on subsets of the $\Lambda$. Specifically, let $(\Lambda, \to_\beta)$ be the usual set of $\lambda$-terms together with the usual $\beta$-reduction rule. We consider variants of this reduction system by:

- restricting the set $\Lambda$ so that we consider subsets $\Gamma \subseteq \Lambda$, for example simply typable terms, $\lambda I$ terms, linear terms, intersection typable terms, etc.
- making $\to_\beta$ conditional, by asking for a local property to be true of the redex.

We say that a term $t \in \Gamma \subseteq \Lambda$ is in normal form when no rule can apply. Note that normal forms here are not necessarily the usual normal forms of the $\lambda$-calculus. We will refer to the usual normal forms of $\lambda$-calculus, as the $\beta$ normal forms. We assume familiarity with these and other notions related to $\lambda$-calculi and types.

By a reduction strategy we simply mean a way of ordering or constraining reductions in the $\lambda$-calculus [4, 6, 7]. Consequently, we include weak reduction, normal reduction, perpetual reduction [15], optimal reduction [14], etc. Moreover, we may ask that these strategies work on different subsets of the $\lambda$-calculus, for instance typed terms, linear terms, affine terms, etc. In this paper we propose to look again at strategies in the $\lambda$-calculus by asking for some predicate to be verified in the redex:

$$(\lambda x.t)u \to_\beta t[u/x] \text{ if } \mathcal{P}((\lambda x.t)u)$$

A great many strategies can be defined in this way by using different predicates. For example, for $\mathcal{P}((\lambda x.t)u)$ we can have:

1. $u$ is in normal form.
2. $x \in FV(t)$.
3. $(\lambda x.t)$ is closed.
4. $u$ is closed.
5. $(\lambda x.t)u$ is closed.
6. $(\lambda x.t)$ is typable, with respect to a particular type system. In this paper we will consider the Simple Types System for the $\lambda$-calculus [9].

Properties 3 and 4 correspond to closed function (**cf**) and closed argument (**ca**) strategies respectively [11, 10]. The closed-argument reduction strategy was used in [3] to define an extension of the linear $\lambda$-calculus [2] with natural numbers, pairs, and a linear iterator. In that calculus, the use of a closed-argument strategy to maintain the linear constraints on variables, as opposed to forcing terms to be closed-by-construction, was a key aspect in obtaining a system as powerful as Gödel's System $\mathcal{T}$. Relating to property 5, Çağman and Hindley [8] observed that $\alpha$-conversion can be avoided if $\beta$-redexes are closed. However, this is a very strong restriction, and the resulting calculus is very weak.

We are only interested in the local properties of the redex and not the context where the redex is. We allow reductions therefore to take place in any context, so there may be many redexes that satisfy the predicate: our "strategy" does not stipulate which redex should be reduced first.

The examples above are subtle. For instance: $u$ is closed means that the argument must be closed before the redex can reduce, but if it is not closed, then another reduction (which does have a closed argument) may close the term $u$ so that it can be reduced later. Thus properties can become true under reduction. To further illustrate this point, the last example asks for the function to be typable—however, we do not ask for the whole term (or even the whole redex) to be typable. An example would be $(\lambda xy.y)((\lambda x.xx)(\lambda x.xx))$, where only one redex can reduce (the subterm $(\lambda x.xx)(\lambda x.xx)$ although is a redex under the usual $\beta$-rule, is not a redex for this constrained $\beta$-rule).

While looking at a framework for constrained $\beta$-reduction, a number of interesting cases evolved for typable terms. This is the emphasis that we wish to talk about in the current paper, but a full development of the general ideas will be in the longer version of this paper.

## 2 Typed $\beta$-reduction

In this section we define a series of reduction strategies based on properties on typability, more particularly typability with simple types (see [5] for more details on type systems for the $\lambda$-calculus).

### 2.1 Typed redex

We start by considering a strategy where a redex is only fired if it is typable. Thus $\beta$-reduction is defined as:

$$(\lambda x.t)u \to_\beta t[u/x] \text{ if } (\lambda x.t)u \text{ is simply typable}$$

The simply typed $\lambda$-calculus has a subject reduction property which means that all terms that are typed, will be typed during any reduction, so for the simply typed $\lambda$-terms the predicate

in the above reduction relation is redundant. Therefore, for the simply typed $\lambda$-calculus, the normal forms under the above reduction rule, will correspond to the usual $\beta$ normal forms.

On the other hand, if the initial terms were not in the simply typed $\lambda$-calculus, then terms would only be reducible up to a certain point, but not to the full $\beta$ normal form. For example: if $I \equiv (\lambda x.x)$ is the $\lambda$-term representing the identity function, then the term $((II)I)(\lambda x.xx)$ reduces to $I(\lambda x.xx)$, but not to its $\beta$ normal form $(\lambda x.xx)$. This leads to the following question: for which classes of terms, do the $\beta$ constrained normal forms, correspond exactly to the usual (full) $\beta$ normal forms?

Note that, in the $\lambda$-calculus, there are three ways of creating new redexes (see [13]):

1. $(\lambda xy.t)uv \to (\lambda y.t[u/x])v$
2. $(\lambda x.x)(\lambda y.t)u \to (\lambda y.t)u$
3. $(\lambda x.C[xu])(\lambda y.t) \to C'[(\lambda y.t)u']$, where $C$ and $C'$ are contexts, and $C'[\,]$ and $u'$, are obtained from $C[\,]$ and $u$, by replacing all the free occurrences of $x$ by $(\lambda y.t)$.

In every case, if the new (created) redex is typed, then part of the initial term (the one necessary to build the new redex), has to be also typed. This in not sufficient to show that the set of terms for which we can reach $\beta$ normal forms, under this typed reduction, corresponds exactly to the simply typed $\lambda$-terms. For example $\lambda x.xx$ is a $\beta$ normal form and and it is not simply typed. However, if a term reduces to a $\beta$ normal form, by the $\beta$ typed-redex reduction, and the normal is has a simple type, then the initial term is also simply typed.

**Proposition 1.** *If $t \to^* u$, by typed-redex reduction, where $u$ is a simply typable $\beta$ normal form, then $t$ was also a simply typable term.*

Note that a simply typed redex does not duplicate or discard untyped terms. Therefore, any untyped term would have to be in the $\beta$ normal form, which does not happen since the normal form is simply typed.

## 2.2 Typed function (tf)/Typed argument (ta)

We now consider two more reduction relations, based on whether the function, or the argument are typed. The $\beta$-reduction rule in each case is respectively defined as:

$$\begin{array}{lrcll} (\mathbf{tf}): & (\lambda x.t)u & \to & t[u/x] & \text{if } \lambda x.t \text{ is simply typable} \\ (\mathbf{ta}): & (\lambda x.t)u & \to & t[u/x] & \text{if } u \text{ is simply typable} \end{array}$$

For example $(\lambda xy.y)((\lambda x.xx)(\lambda x.xx))$ reduces to the usual *beta* normal form in (**tf**) but not in (**ta**). On the other hand, $(\lambda x.xx)I$ reduces to the usual $\beta$-normal form in (**ta**) but not in (**tf**).

Intuitively, asking for the function to be typed (but not necessarily the argument), in a sense, means that we do not care if the argument is "ill-behaved", because, since the function is "well-behaved", it will either discard or move around the "ill-behaved" argument. On the other hand, if the argument is typed, then an "ill-behaved" function, will just move around "well-behaved" arguments.

Note that, applying usual $\beta$-reduction on the $\lambda$-term $(\lambda xy.y)((\lambda x.xx)(\lambda x.xx))$, may yield an infinite reduction sequence, if one follows a call-by-value evaluation strategy. In either (**ta**) or (**tf**) that does not happen: in (**ta**) the term does not reduce, and in (**tf**) it corresponds to call-by-name strategy in the usual $\lambda$-calculus.

The above observations lead us to the following conjecture:

**Conjecture 1.** ***tf*** *(resp.* ***ta****) is terminating.*

Note that, although we believe that **tf** (resp. **ta**) is terminating, the normal forms do not always coincide with the full $\beta$-normal forms. The term $(\lambda xy.x)((\lambda x.xx)(\lambda x.xx))$ is an example for both **tf** and **ta**.

### 2.3 Typed function or argument (tf + ta)

We now combine the strategies defined in the previous section. We begin with an example. Consider the term $\lambda x.xx$ which is well-known to be typable in intersection type systems, but not simple types.

- $(\lambda x.xx)I$ reduces after one redex to a simply typable term, and is $\beta$ strongly normalising. We observe that the argument of the redex is simply typable (it corresponds to (**ta**)).
- $I(\lambda x.xx)$ reduces after one redex to an intersection typable term, and is $\beta$ strongly normalising. We observe that the function of the redex is simply typable (it corresponds to (**tf**)).

Asking for just the function or the argument to be simply typable is not enough to reduce $I(\lambda x.xx)I$ to its full $\beta$ normal form, however, asking for either function or argument allows this term to reduce to full normal form.

We call this strategy (**tf + ta**) and define it as

$$(\lambda x.t)u \to t[u/x], \text{ if } \lambda x.t \text{ or } u \text{ are simply typable}$$

Note that asking for either function or argument to be simply typable is enough to reduce to $\beta$ normal form all the terms reducible in either (**tf**) or (**ta**). If we recall the example from the previous section $(\lambda xy.y)((\lambda x.xx)(\lambda x.xx))$, one notices that using these strategies, allows us to follow a normalising path of reduction. These observations lead us to the following:

**Conjecture 2.** ***tf + ta*** *reduction is terminating.*

Similarly to what happens in **tf** (resp. **ta**), the normal forms do not coincide with the full $\beta$-normal forms. Again, $(\lambda xy.x)((\lambda x.xx)(\lambda x.xx))$ is an example of such a term. However, for **tf + ta**, we believe they do coincide for terms which are normalising (that is, terms for which is possible to reach a normal form).

## 3 Dynamic type-inference

Having defined properties on the typability of the redexes (or part of the redexes), implies that some kind of mechanism needs to be implemented, to check that the property holds.

If one imposes a constraint that reduction can only take place if the redex itself is typable, then if the initial term is typable then the term can be fully ($\beta$) normalised.

However, for the other systems, where we consider weaker constraints, some redexes which are not simply typable, may become typable after a (**tf**) or (**ta**) reduction. Recall the example, $(\lambda x.xx)I$, which becomes simply typed, after a (**ta**) reduction. This poses some interesting questions regarding the kind of type inference mechanism one can define for these systems. Note that, one can always use principal type algorithms for the simply typed $\lambda$-calculus [16, 12], but perhaps more interesting dynamic algorithms can be defined.

# 4 Conclusion

In this short paper we have outlined a framework for studying constraints on $\beta$-reduction in the $\lambda$-calculus, and began the study of one particular instance based on types. Current and future work include constructing a general framework to study all variants, and in particular, building links with existing strategies and variants of the $\lambda$-calculus.

# References


[1] S. Abramsky. The lazy $\lambda$-calculus. In D. A. Turner, editor, *Research Topics in Functional Programming*, chapter 4, pages 65–117. Addison Wesley, 1990.

[2] S. Abramsky. Computational Interpretations of Linear Logic. *Theoretical Computer Science*, 111:3–57, 1993.

[3] S. Alves, M. Fernández, M. Florido, and I. Mackie. Gödel's system $\mathcal{T}$ revisited. *Theor. Comput. Sci.*, 411(11-13):1484–1500, 2010.

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

[5] H. P. Barendregt. Lambda Calculi with Types. In S. Abramsky, D. M. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2. Clarendon Press, Oxford, 1992.

[6] J. A. Bergstra and J. W. Klop. Church-Rosser Strategies in the Lambda Calculus. *Theor. Comput. Sci.*, 9:27–38, 1979.

[7] J. A. Bergstra and J. W. Klop. Strong normalization and perpetual reductions in the lambda calculus. *Elektronische Informationsverarbeitung und Kybernetik*, 18(7/8):403–417, 1982.

[8] N. Çağman and J. R. Hindley. Combinatory weak reduction in lambda calculus. *Theoretical Computer Science*, 198(1–2):239–249, 1998.

[9] H. B. Curry. Functionality in Combinatory Logic. In *Proceedings of the National Academy of Science, U.S.A*, volume 20, pages 584–590. National Academy of Sciences, 1934.

[10] M. Fernández and I. Mackie. Closed reduction in the $\lambda$-calculus. In *Proceedings of Computer Science Logic (CSL'99)*, volume 1683 of *LNCS*, pages 220–234. Springer-Verlag, September 1999.

[11] M. Fernández, I. Mackie, and F.-R. Sinot. Closed reduction: explicit substitutions without alpha conversion. *MSCS*, 15(2):343–381, 2005.

[12] J. R. Hindley. *Basic Simple Type Theory*, volume 42 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1997.

[13] J. Lévy. *Réductions correctes et optimales dans le lambda calcul*. PhD thesis, Université Paris VII, 1978.

[14] 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.

[15] F. van Raamsdonk, P. Severi, M. H. Sørensen, and H. Xi. Perpetual reductions in lambda-calculus. *Inf. Comput.*, 149(2):173–225, 1999.

[16] M. Wand. A Simple Algorithm and Proof for Type Inference. *Fundamenta Infomaticae*, 10:115–121, 1987.

# Lazy Term Rewriting Modulo Associativity and Commutativity*

Walid Belkhir[1] and Alain Giorgetti[1,2]
[1] LIFC, University of Franche-Comté, France
[1,2] INRIA Nancy - Grand Est, CASSIS project, 54600 Villers-lès-Nancy, France
{walid.belkhir,alain.giorgetti}@lifc.univ-fcomte.fr

**Abstract**

We suggest a lazy evaluation semantics for first-order term rewriting with associative and commutative function symbols, known as AC-rewriting. The AC-matching produces a lazy list of solutions: a first substitution besides a non-evaluated object that encodes the remaining computations. We extend lazy rule application to usual traversal strategies.

## 1 Introduction

Term rewriting modulo associativity and commutativity of some function symbols, known as AC-rewriting, is a key operation in many programming languages, theorem provers and computer algebra systems. Examples of AC-symbols are $+$ and $*$ for addition and multiplication in arithmetical expressions, $\vee$ and $\wedge$ for disjunction and conjunction in Boolean expressions, etc. AC-rewriting performance mainly relies on that of its AC-matching algorithm. On the one hand, the problem of deciding whether an AC-matching problem has a solution is NP-complete [1]. On the other hand, the number of solutions to a given AC-matching problem can be exponential in the size of its pattern. Thus many works propose optimizations for AC-matching. One can divide optimized algorithms in two classes, depending on what they are designed for. In the first class some structural properties are imposed on the terms, and the pattern falls into one of several forms for which efficient algorithms can be designed. Examples are the depth-bounded patterns in the many-to-one matching algorithm used by Elan [11] and greedy matching techniques adopted in Maude [4]. In the second class there is no restriction on the structural properties of the terms. Algorithms in this class are search-based, and uses several techniques to collapse the search space, such as constraint propagation on non linear variables [8], recursive decomposition via bipartite graphs [5], ordering matching subproblems based on constraint propagation [6] and Diophantine techniques [7].

Formal semantics proposed so far for AC-rewriting enumerate all the possible solutions of each matching problem. More precisely, the application modulo AC of a rewrite rule $l \to r$ to a given term $t$ usually proceeds in two steps. Firstly, all the solutions (i.e. substitutions) $\sigma_1$, ..., $\sigma_n$ ($n \geq 0$) of the AC-matching problem whether the term $t$ matches the pattern $l$ are computed and stored in a structure, say a set $\{\sigma_1, \cdots, \sigma_n\}$. Secondly, this set is applied to $r$ and the result is the set $\{\sigma_1(r), \cdots, \sigma_n(r)\}$. Other structures such as multisets or lists can alternatively by used for other applications of the calculus. Directly implementing this *eager* semantics is clearly less efficient than a lazy mechanism that only computes a first result of the application of a rewrite rule and allows the computation by need of the remaining results. The potential benefits are clear: performances would be increased by avoiding unnecessary calculations and the same specification would also hold for more general matching problems with potential infinite sets of solutions. As far as we know no work defines the AC-matching in a lazy way and integrates it in a rewriting semantics.

This paper presents a lazy semantics for the first-order fragment of the AC-rewriting calculus [3]. In this fragment first-order rewrite rules are applied to first-order terms. Firstly, we describe a lazy AC-matching algorithm. Secondly, we integrate the lazy AC-matching in a strategy language and we define

*This work is supported by the project PPF MIDi of the University of Franche-Comté.

the semantics of rule and strategy functional application. Our goal is to specify the lazy AC-matching and the strategy language semantics for an implementation in a strict rule-based language, e.g. Maude. We reach this goal by representing lazy lists by means of explicit objects.

## 2 Notations and preliminaries

Familiarity with the usual first-order notions of signature, (ground) term, arity, position and substitution is assumed. Let $X$ be a countable set of variables, $F$ a countable set of function symbols, and $F_{AC} \subseteq F$ the set of associative-commutative function symbols. Let $T(F,X)$ denote the set of terms built out of the symbols in $F$ and the variables in $X$. The type of terms is denoted by $\mathcal{T}$. Let $\mathcal{S}$ denote the set and the type of substitutions. A substitution is a set of assignments $\{x_1 \mapsto t_1, \cdots, x_n \mapsto t_n\}$ with distinct variables $x_1, \cdots, x_n$ in $X$ and terms $t_1, \cdots, t_n$ in $T(F,X)$. If $t$ is a term and $\sigma$ is a substitution then $\sigma(t)$ denotes the term that results from the application of $\sigma$ to $t$. Given a position $p$, the subterm of $t$ at position $p$ is denoted by $t_{|p}$. A term $t$ in $T(F,X)$ is *flat* if no *AC* symbol occurs as the child of the same symbol in the term.

**$\mathbb{T}$-Matching.** For an equational theory $\mathbb{T}$ and two terms $t$ and $u$ in $T(F,X)$, a substitution $\sigma$ in $\mathcal{S}$ is a *solution* to the $\mathbb{T}$-matching problem $t \ll_{\mathbb{T}} u$ iff $\mathbb{T} \models (\sigma(t) = u)$ and all the variables substituted by $\sigma$ are in $t$. We say that $t$ matches $u$ modulo $\mathbb{T}$ iff the $\mathbb{T}$-matching problem $t \ll_{\mathbb{T}} u$ admits at least one solution. In this paper $\mathbb{T}$ is fixed and axiomatizes the associativity and commutativity of symbols in $F_{AC}$, i.e. it is the union of the sets of axioms $\{t_1 + t_2 = t_2 + t_1,\ (t_1 + t_2) + t_3 = t_1 + (t_2 + t_3)\}$ when $+$ ranges over $F_{AC}$.

**Strategies.** Primitive strategies are rewrite rules $l \to r$ and the id and fail strategies that respectively always and never succeed. By default, a rewrite rule only applies at the top. This behavior can be modified by strategy constructors. For sake of simplicity we only consider here the four usual traversal strategies *leftmost-outermost*, *leftmost-innermost*, *parallel-outermost* and *parallel-innermost* [13, Definition 4.9.5]. They select redexes according to their position. The first two only hold when the root of the rule left-hand side is not an AC symbol. They select at most one redex, namely the leftmost one, respectively at minimal and maximal positions. The last two respectively select all the redexes at these minimal and maximal positions. Let $v$ be one of these four strategy constructors. The annotation of the rewrite rule $l \to r$ by $v$ is a strategy denoted by $v(l \to r)$. The sequential composition of two strategies $u$ and $w$ is denoted by $u;w$. The application of a strategy $u$ to a term $t$ is denoted by $[u] \cdot t$.

**Specification of laziness.** We specify lazy semantics and lazy algorithms with the same formalism of lazy lists. A lazy AC-matching algorithm produces a lazy list of substitutions and a strategy application produces a lazy list of terms. Both types of lazy lists are instances of the same type $LList(\alpha)$ of polymorphic lazy lists with elements of type $\alpha$ [2, Chapter 13]. This type is defined by the constructor $\perp_\alpha$ of the empty list and by the constructor :: of the lazy list $b :: \beta$ where $b$ is of type $\alpha$ and $\beta$ is also of type $LList(\alpha)$. The *concatenation* of two lazy lists $\beta_1$ and $\beta_2$ of type $LList(\alpha)$ is denoted by $\beta_1 \cdot\cdot \beta_2$.

## 3 Lazy AC-matching

In this section we justify the existence of a lazy AC-matching algorithm that computes a first solution to any AC-matching problem $t \ll_{AC} u$ and makes it possible to get the remaining solutions by need.

**Proposition 1.** *There is a lazy algorithm that computes the solutions of a given AC-matching problem.*

**Justification.** Such an algorithm can be designed as follows. It is first specified by a rewriting system, named ACM (specifying matching algorithms with rewriting systems is a well-known approach, see e.g. [13, Section 2.1.4]). The system ACM rewrites any AC-matching problem $t \ll_{AC} u$ into $\perp_{\mathbb{S}}$ or a non-empty lazy list $s :: \sigma$, where $s$ is a first solution of the matching problem and $\sigma$ is an explicit object representing the delayed computations. In order to get the remaining solutions by need, we introduce in ACM a fresh symbol *Next* and a set of reduction rules for terms of the form $Next(\sigma)$. These rules are expected to rewrite $Next(\sigma)$ into a lazy list, whose first element, if it exists, is the next solution to the input problem and whose tail represents the remaining solutions. This specification may be proved to be sound wrt. the definition of AC-matching given in Section 2.

Then the system ACM is shown to be terminating and confluent. This guarantees that its encoding in a rule-based programming language is an algorithm. It is lazy since it returns only a first solution, and an explicit object $\sigma$ representing the delayed computations. To get another solution, the same algorithm should be invoked on the term $Next(\sigma)$.

The first algorithm input is assumed to be an AC-matching problem $t \ll_{AC} u$ where the two terms $t$ and $u$ are flat (an algorithm to flatten terms can be found e.g. in [9]). The system ACM can be designed by combining the syntactic matching and a lazy production of surjections supporting the associativity and commutativity axioms as follows. We denote by $S_{n,k}$, where $k, n \in \mathbb{N}^*$, $k \leq n$, the set of all the surjective functions from $\{1, \cdots, n\}$ to $\{1, \cdots, k\}$. The lazy production of the elements of $S_{n,k}$ is made possible by composing two iterators: a first one [10] which iterates over the set of all the partitions[1] of size $k$ over $\{1, \cdots, n\}$ and a second one [12] which iterates over the set of all the permutations of the set $\{1, \cdots, k\}$. Let $L_{n,k}$ denote a lazy list composed of all the elements of $S_{n,k}$. The main reduction rule in ACM is:

$$\underbrace{+(t_1, \cdots, t_k)}_{t} \ll_{AC} \underbrace{+(u_1, \cdots, u_n)}_{u} \rightsquigarrow \{t_1 \ll_{AC} \alpha_1, \cdots, t_k \ll_{AC} \alpha_k\} \cdot\cdot (t, u, \mathfrak{l}) \text{ if } + \in F_{AC},\ k \leq n \text{ and } L_{n,k} = \mathfrak{s} :: \mathfrak{l}.$$

The expression $\{t_1 \ll_{AC} \alpha_1, \cdots, t_k \ll_{AC} \alpha_k\}$ is a set of matching problems. It will be reduced by ACM to a lazy list of substitutions. It corresponds to the choice of the first surjection $\mathfrak{s}$ in the lazy list $L_{n,k}$. The expression $(t, u, \mathfrak{l})$ is an explicit object representing the delayed choice of another surjection in $L_{n,k}$. The sequence $\alpha_1, \cdots, \alpha_k$ is an arrangement of the sequence $u_1, \cdots, u_n$, determined by the surjection $\mathfrak{s}$. For instance, consider $\mathfrak{s} \in S_{4,3}$ s.t. $\mathfrak{s}(1) = 2, \mathfrak{s}(2) = 1, \mathfrak{s}(3) = 3, \mathfrak{s}(4) = 2$. Then the sequence $b_2, +(b_1, b_4), b_3$ is an arrangement of $b_1, b_2, b_3, b_4$ completely determined by $\mathfrak{s}$, and the substitution $\{x_1 \mapsto b_2, x_2 \mapsto +(b_1, b_4)\}$ is a solution of the AC-matching problem $+(x_1, x_2, b_3) \ll_{AC} +(b_1, b_2, b_3, b_4)$ where $x_1$ and $x_2$ are variables and the other symbols are constants.

# 4 Lazy AC-rewriting with strategies

In this section we integrate the lazy AC-matching with strategy application. We define the semantics of strategy application by labeled rewrite rules of the form `rule_label:` $\cdots \rightsquigarrow \cdots$. The rewrite relation $\rightsquigarrow$ should not be confused with the relation $\rightarrow$ of the strategy language. To avoid this confusion, this semantics is not called a "rewriting semantics" but the *operational semantics* of the strategy language.

The operational semantics of the strategies id and fail and of strategy composition are defined in Figure 1(a) for any lazy list of terms $\tau$ and any two strategies $u$ and $v$. The operational semantics of top rewriting is defined in Figure 1(b). Let LTR be the system of these four rules. The rules `rule1` and `rule2` reduce the application of a rewrite rule at the top of terms in a lazy list in $LList(\mathcal{T})$. The rules `subs1` and `subs2` reduce the application of a substitution on the same lazy lists. The expression $l \ll_{AC} t$ is reduced to its normal form by the system ACM of lazy AC-matching. The application of the system LTR

[1] A *partition* of a set $A$ is a set of nonempty subsets of $A$ such that every element $a \in A$ is in exactly one of these subsets.

| | |
|---|---|
| `id:` | $[\mathsf{id}]\cdot\tau \rightsquigarrow \tau$ |
| `fail:` | $[\mathsf{fail}]\cdot\tau \rightsquigarrow \bot_{\mathcal{T}}$ |
| `compose:` | $[u;v]\cdot\tau \rightsquigarrow [u]\cdot([v]\cdot\tau)$ |

(a) id, fail and composition rules

| | |
|---|---|
| `rule1:` | $[l\to r]\cdot\bot_{\mathcal{T}} \rightsquigarrow \bot_{\mathcal{T}}$ |
| `rule2:` | $[l\to r]\cdot(t::\tau) \rightsquigarrow (l\ll_{AC} t)(r)\cdot\cdot([l\to r]\cdot\tau)$ |
| `subs1:` | $\bot_{\mathcal{S}}(t) \rightsquigarrow \bot_{\mathcal{T}}$ |
| `subs2:` | $(s::\sigma)(t) \rightsquigarrow s(t)::\sigma(t)$ |

(b) Top rewriting

Figure 1: AC-rewriting operational semantics

to the expression $[l\to r]\cdot(t::\bot_{\mathcal{T}})$ produces either $\bot_{\mathcal{T}}$ or a lazy list $u_1::\tau_1$, where $u_1$ is the first result of the application of the rewrite rule $l\to r$ at the top of the term $t$ and $\tau_1$ is (a syntactic object denoting) the lazy list of the other results. When applying the rule

| | |
|---|---|
| `NeedNext:` | $\sigma \rightsquigarrow \mathit{Next}(\sigma)$ if $\sigma$ neither match $s::\sigma'$ nor $\bot_{\mathcal{S}}$ |

on $\tau_1$, the system `ACM` and then the system `LTR`, we get again either $\bot_{\mathcal{T}}$ or a lazy list $u_2::\tau_2$, where $u_2$ is the second result of the application of the rewrite rule $l\to r$ at the top of the term $t$ and $\tau_2$ is again (a syntactic object denoting) a lazy list of terms that represents the remaining results, and so on. If the rule `NeedNext` is implemented in a language which does not support anti-patterns, the negative conditions on the structure of $\sigma$ in the rule can be replaced by a matching with the explicit patterns of $\sigma$ distinct from $s::\sigma'$ and $\bot_{\mathcal{S}}$.

The application of a traversal strategy on a lazy list of terms in $\mathit{LList}(\mathcal{T})$ is defined as follows:

| | |
|---|---|
| `traversal1:` | $[v(l\to r)]\cdot\bot_{\mathcal{T}} \rightsquigarrow \bot_{\mathcal{T}}$ |
| `traversal2:` | $[v(l\to r)]\cdot(t::\tau) \rightsquigarrow [v(l\to r)]\cdot t\ \cdot\cdot\ [v(l\to r)]\cdot\tau$ |

The rules

$$[v(u)]\cdot t \rightsquigarrow \begin{cases} [u]\cdot t & \text{if } [u]\cdot t \neq \bot_{\mathcal{T}} \text{ or } t\in X \\ \uparrow f([v(u)]\cdot t_1,\cdots,[v(u)]\cdot t_n) & \text{if } [u]\cdot t = \bot_{\mathcal{T}} \text{ and } t = f(t_1,\cdots,t_n) \end{cases}$$

define the application of the traversal strategy $v(u)$ on the term $t$ for the rewrite rule $u$ and the *parallel-outermost* strategy constructor $v$. The other traversal strategies can be handled similarly. The application of a rewrite rule at the top of a term yields a lazy list of terms in $\mathit{LList}(\mathcal{T})$. Here the application of a rule to a term at arbitrary depth, via a traversal strategy, yields a *decorated* term, which is a term where some subterms are replaced by a lazy list of terms, abusively called *lazy subterm*, with the property that lazy subterms are not nested. In other words, the positions of two lazy subterms are not comparable, for the standard prefix partial order over the set of term positions. The operator $\uparrow$ is assumed to reduce a decorated term to a lazy list of terms. We justify its behavior as follows. A decorated term can be encoded by a tuple $(t,k,p,\delta)$ where $t$ is the term before strategy application, $k$ is the number of decorated positions, $p$ is a function from $\{1,\ldots,k\}$ to the domain of $t$ (i.e. its set of positions) which defines the decorated positions, such that $p(i)$ and $p(j)$ are not comparable if $i\neq j$, and $\delta$ is the function from $\{1,\ldots,k\}$ to $\mathit{LList}(\mathcal{T})$ such that $\delta(i)$ is the lazy list at position $p(i)$, $1\leq i\leq k$. It is easy to construct an iterator over the $k$-tuples of positive integers (for instance in lexicographical order), and to derive from it an iterator over tuples of terms $(s_1,\ldots,s_k)$ with $s_i$ in the list $\delta(i)$ for $1\leq i\leq k$. From this iterator and function $p$ we derive an iterator producing the lazy list $\uparrow t$ by replacing each subterm $t_{|p(i)}$ by $s_i$.

# 5 Conclusion

We presented a lazy evaluation semantics for AC-rewriting and described a lazy AC-matching algorithm underlying it. The semantics is designed to be implemented in a strict rule-based language by repre-

senting lazy lists of substitutions and terms by explicit objects. We also described a common principle for lazy traversal strategies. The potential benefits are clear: performances are dramatically increased by avoiding unnecessary computations. However our approach is not efficient when the number of solutions of the AC-matching problem is small. In particular we do not claim for efficiency for the search of the first solution by the AC-matching algorithm. We plan to extend the approach to other strategy combinators and to work on an implementation.

**Acknowledgements** The authors are grateful to the anonymous referees for helpful comments and suggestions.

# References

[1] D. Benanav, D. Kapur, and P. Narendran. Complexity of matching problems. In *Proc. of the 1st Int. Conf. on Rewriting Techniques and Applications*, pages 417–429. Springer, 1985.

[2] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development, Coq'Art:the Calculus of Inductive Constructions*. Springer, 2004.

[3] 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.

[4] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *LNCS*. Springer, 2007.

[5] S. Eker. Associative-commutative matching via bipartite graph matching. *Comput. J.*, 38(5):381–399, 1995.

[6] S. Eker. Fast matching in combinations of regular equational theories. *Electr. Notes Theor. Comput. Sci.*, 4, 1996.

[7] S. Eker. Single elementary associative-commutative matching. *J. Autom. Reasoning*, 28(1):35–51, 2002.

[8] B. Gramlich. Efficient AC-matching using constraint propagation. In *Proc. 2nd Int. Workshop on Unification, Internal Report 89 R 38, CRIN*, Val d'Ajol, France, July 1988.

[9] J.-P. Jouannaud. Associative-commutative rewriting via flattening, 2008. Unpublished manuscript. `http://www.lix.polytechnique.fr/~jouannaud/articles/acrvf.pdf`.

[10] S.-I. Kawano and S.-I. Nakano. Constant time generation of set partitions. *IEICE Transactions*, 88-A(4):930–934, 2005.

[11] H. Kirchner and P.-E. Moreau. Promoting rewriting to a programming language: a compiler for non-deterministic rewrite programs in AC-theories. *J. Funct. Program.*, 11:207–251, March 2001.

[12] T. Knapen, D. Abrahams, R. Richter, and J. Siek. Permutation iterator, 2006. `http://www.boost.org/doc/libs/1_46_0/libs/iterator/doc/permutation_iterator.html`.

[13] Terese. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.

# Strategies for Decreasingly Confluent Rewrite Systems*

Nao Hirokawa
School of Information Science
Japan Advanced Institute of Science and Technology, Japan

Aart Middeldorp
Institute of Computer Science
University of Innsbruck, Austria

**Abstract**

In this note we present two normalising strategies for the class of decreasingly confluent term rewrite systems introduced by (Hirokawa and Middeldorp, 2010).

## 1 Introduction

In this note we are concerned with normalising strategies for first-order term rewrite systems (TRSs for short). For the class of orthogonal TRSs several normalisation results are known (see [5, Chapters 4 and 9] for an overview). We extend two of these results to the larger class of confluent systems introduced in [2]. This class consists of all left-linear and locally confluent TRSs $\mathcal{R}$ for which the critical pair steps are relatively terminating with respect to $\mathcal{R}$. Such TRSs will be called *decreasingly confluent* in this note.

**Example 1.1.** Consider the left-linear and locally confluent TRS $\mathcal{R}$

$$\begin{array}{lll} \mathsf{nats} \to \mathsf{0} : \mathsf{inc}(\mathsf{nats}) & \mathsf{inc}(x : y) \to \mathsf{s}(x) : \mathsf{inc}(y) & \mathsf{hd}(x : y) \to x \\ & \mathsf{inc}(\mathsf{tl}(\mathsf{nats})) \to \mathsf{tl}(\mathsf{inc}(\mathsf{nats})) & \mathsf{tl}(x : y) \to y \end{array}$$

of [1]. The critical pair steps $\mathsf{inc}(\mathsf{tl}(\mathsf{nats})) \to \mathsf{tl}(\mathsf{inc}(\mathsf{nats}))$ and $\mathsf{inc}(\mathsf{tl}(\mathsf{nats})) \to \mathsf{inc}(\mathsf{tl}(\mathsf{0} : \mathsf{inc}(\mathsf{nats})))$ are relatively terminating with respect to $\mathcal{R}$ [2]. Hence $\mathcal{R}$ is decreasingly confluent.

In the next section we give a simple inductive definition of a strategy which coincides with the full-substitution strategy when restricted to orthogonal TRSs. We prove that this strategy is *cofinal* and hence normalising for decreasingly confluent TRSs. In Section 3 we prove that the parallel-outermost strategy is normalising for decreasingly confluent TRSs. In the remainder of this section we recall some definitions and fix some notations.

A *strategy* for a TRS $\mathcal{R}$ is a relation $\rightsquigarrow$ such that $\rightsquigarrow \subseteq \to_{\mathcal{R}}^*$ and $\mathsf{NF}(\rightsquigarrow) = \mathsf{NF}(\to_{\mathcal{R}})$. For technical reasons we use the following variation. A *strategy* for a TRS $\mathcal{R}$ is a relation $\rightsquigarrow$ such that $\rightsquigarrow \subseteq \to_{\mathcal{R}}^*$ and $\mathsf{NF}(\rightsquigarrow) = \varnothing$. One can obtain strategies from total strategies by restricting steps to reducible terms of $\mathcal{R}$. A total strategy $\rightsquigarrow$ is *normalising* if every infinite $\rightsquigarrow$ sequence starting from a weakly normalising (with respect to $\to_{\mathcal{R}}$) term contains a normal form. A total strategy $\rightsquigarrow$ is *cofinal* if for every infinite sequence $s_0 \rightsquigarrow s_1 \rightsquigarrow s_2 \rightsquigarrow \cdots$ and for every term $t$ with $s_0 \to^* t$ there is a $k \geqslant 0$ with $t \to^* s_k$. Cofinal strategies are normalising [5, Proposition 4.9.3].

An overlap $(\ell_1 \to r_1, p, \ell_2 \to r_2)_\mu$ of a TRS $\mathcal{R}$ consists of variants $\ell_1 \to r_1$ and $\ell_2 \to r_2$ of rules of $\mathcal{R}$ without common variables, a non-variable position $p$ in $\ell_2$, and a most general unifier $\mu$ of $\ell_1$ and $\ell_2|_p$. If $p = \varepsilon$ then we require that $\ell_1 \to r_1$ and $\ell_2 \to r_2$ are not variants of each other. The set of critical pair steps $\{\ell_2\mu \to \ell_2\mu[r_1\mu]_p, \ell_2\mu \to r_2\mu \mid (\ell_1 \to r_1, p, \ell_2 \to r_2)_\mu \text{ is an overlap of } \mathcal{R}\}$ is denoted by $\mathsf{CPS}(\mathcal{R})$. Given two TRSs $\mathcal{R}$ and $\mathcal{S}$ we write $\to_{\mathcal{R}/\mathcal{S}}$ for $\to_{\mathcal{S}}^* \cdot \to_{\mathcal{R}} \cdot \to_{\mathcal{S}}^*$. We say that $\mathcal{R}$ is relatively terminating with respect to $\mathcal{S}$ or simply that $\mathcal{R}/\mathcal{S}$ is terminating whenever the relation $\to_{\mathcal{R}/\mathcal{S}}$ is well-founded. With any TRS $\mathcal{R}$ we associate the rewrite orders $> := \to^+_{\mathsf{CPS}(\mathcal{R})/\mathcal{R}}$ and $\gtrsim := \to_{\mathcal{R}}^*$. Note that $> \subseteq \gtrsim$. For decreasingly confluent TRSs $>$ is well-founded and in the proofs in the subsequent sections we employ well-founded induction with respect to $>$.

*This research is partially supported by FWF (Austrian Science Fund) project P22467 and the Grant-in-Aid for Young Scientists (B) 22700009 of the Japan Society for the Promotion of Science.

## 2 Maximal Strategy

In this section we present a cofinal strategy for decreasingly confluent TRSs. We start by recalling the multi-step relation.

**Definition 2.1.** Let $\mathcal{R}$ be a TRS. The *multi-step* relation $\multimap_{\mathcal{R}}$ (or simply $\multimap$) is inductively defined as follows: (1) $x \multimap x$ for all variables $x$, (2) $f(s_1,\dots,s_n) \multimap f(t_1,\dots,t_n)$ if $s_i \multimap t_i$ for all $1 \leqslant i \leqslant n$, and (3) $\ell\sigma \multimap r\tau$ if $\ell \to r \in \mathcal{R}$ and $\sigma \multimap \tau$, where $\sigma \multimap \tau$ if $x\sigma \multimap x\tau$ for all variables $x$.

By restricting case (2) in Definition 2.1 to non-redexes, we obtain an easily computable strategy. Due to the choice of the rule $\ell \to r$ in case (3), the strategy is non-deterministic. For orthogonal TRSs $\mathcal{R}$ the choice disappears and it can be shown that $\multimap\!\bullet_{\mathcal{R}}$ executes one step of the full-substitution strategy.

**Definition 2.2.** Let $\mathcal{R}$ be a TRS. The *maximal* strategy $\multimap\!\bullet_{\mathcal{R}}$ (or simply $\multimap\!\bullet$ if $\mathcal{R}$ can be inferred from the context) is defined as follows:

(1) $x \multimap\!\bullet x$ for all variables $x$,

(2) $f(s_1,\dots,s_n) \multimap\!\bullet f(t_1,\dots,t_n)$ if $s_i \multimap\!\bullet t_i$ for all $1 \leqslant i \leqslant n$ and $f(s_1,\dots,s_n)$ is not a redex,

(3) $\ell\sigma \multimap\!\bullet r\tau$ if $\ell \to r \in \mathcal{R}$ and $\sigma \multimap\!\bullet \tau$.

The next lemma is the key for the main theorem of this section.

**Lemma 2.3.** *Let $\mathcal{R}$ be a left-linear TRS. If $t \mathrel{\multimap\!\circ\!\!-} s \multimap\!\bullet u$ then $t < s > u$ or $t \multimap u$.*

*Proof.* Let $t \mathrel{\multimap\!\circ\!\!-} s \multimap\!\bullet u$. We perform structural induction on $s$. The base case is trivial. For the induction step, we consider four cases, depending on which of the two clauses (2) and (3) was used to derive $t \mathrel{\multimap\!\circ\!\!-} s \multimap\!\bullet u$.

- Suppose both $t \mathrel{\multimap\!\circ\!\!-} s$ and $s \multimap\!\bullet u$ were obtained using clause (2). We have $s = f(s_1,\dots,s_n)$, $t = f(t_1,\dots,t_n)$, and $u = f(u_1,\dots,u_n)$ such that $t_i \mathrel{\multimap\!\circ\!\!-} s_i \multimap\!\bullet u_i$ for all $1 \leqslant i \leqslant n$. From the induction hypothesis we learn that $t_i < s_i > u_i$ or $t_i \multimap u_i$ for all $1 \leqslant i \leqslant n$. If $t_i < s_i > u_i$ for some $i$ then we obtain $t < s > u$. If $t_i \multimap u_i$ for all $1 \leqslant i \leqslant n$ then $t \multimap u$ by clause (2) of the definition.

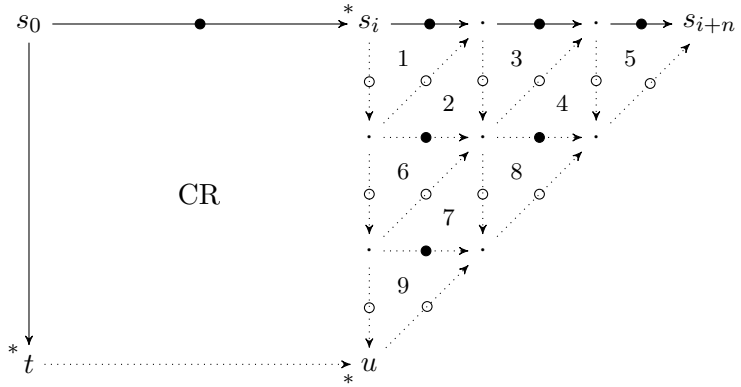
- Suppose both $t \mathrel{\multimap\!\circ\!\!-} s$ and $s \multimap\!\bullet u$ were obtained using clause (3). If different rewrite rules were used then $s$ contains overlapping redexes and we easily obtain $t < s > u$. So suppose both $t \mathrel{\multimap\!\circ\!\!-} s$ and $s \multimap\!\bullet u$ were obtained using the rewrite rule $\ell \to r$. Then there are substitutions $\sigma$, $\tau$, and $\rho$ with $\sigma \multimap \tau$ and $\sigma \multimap\!\bullet \rho$ such that $s = \ell\sigma$, $t = r\tau$, and $u = r\rho$. From the induction hypothesis we learn that $x\tau < x\sigma > x\rho$ or $x\tau \multimap x\rho$ for all variables $x \in \mathcal{V}\mathsf{ar}(\ell)$. If $x\tau < x\sigma > x\rho$ for some variable $x \in \mathcal{V}\mathsf{ar}(\ell)$ then $\ell\sigma > \ell\tau \gtrsim r\tau$ and $\ell\sigma > \ell\rho \gtrsim r\rho$. Thus we obtain $t < s > u$. If $x\tau \multimap x\rho$ for all variables $x \in \mathcal{V}\mathsf{ar}(\ell)$ then $\tau \multimap \rho$ and thus $t \multimap u$ by clause (3) of the definition.

- Suppose $t \mathrel{\multimap\!\circ\!\!-} s$ was obtained using clause (2) and $s \multimap\!\bullet u$ was obtained using clause (3). Let $\ell \to r$ be the rewrite rule that was used to obtain $s \multimap\!\bullet u$. We have $s = f(s_1,\dots,s_n) = \ell\sigma$, $t = f(t_1,\dots,t_n)$ with $s_i \multimap t_i$ for all $1 \leqslant i \leqslant n$, and $u = r\tau$ with $\sigma \multimap\!\bullet \tau$. If one of the redexes contracted in $s \multimap t$ overlaps $\ell$ then $t < s > u$. Otherwise $t = \ell\rho$ for some substitution $\rho$ with $\sigma \multimap \rho$. We obtain $t \multimap u$ by clause (3) exactly as in the preceding case.

- Suppose $t \mathrel{\leftarrow\!\!\circ} s$ was obtained using clause (3) and $s \mathrel{\bullet\!\!\rightarrow} u$ was obtained using clause (2). This case is impossible because $s$ is a redex. □

**Theorem 2.4.** *The maximal strategy is cofinal for decreasingly confluent TRSs.*

*Proof.* Let $\mathcal{R}$ be a decreasingly confluent TRS. Suppose $t \mathrel{^*\!\leftarrow} s_0 \mathrel{\bullet\!\!\rightarrow} s_1 \mathrel{\bullet\!\!\rightarrow} s_2 \mathrel{\bullet\!\!\rightarrow} \cdots$ We have to show that $t \to^* s_k$ for some $k \geqslant 0$. Because $>$ is well-founded, there is an $i \geqslant 0$ such that $s_i \not> s_m$ for all $m \geqslant i$. Confluence yields a term $u$ such that $s_i \to^* u \mathrel{^*\!\leftarrow} t$. Let $n$ be the length of $s_i \to^* u$. We have $s_i \mathrel{\bullet\!\!\rightarrow}^n u$. By using Lemma 2.3 repeatedly, the following diagram is obtained (where we assume that $n = 3$):




The numbers in the triangles indicate the order in which the diagram is filled. Note that when applying Lemma 2.3 to $t' \mathrel{\leftarrow\!\!\circ} s' \mathrel{\bullet\!\!\rightarrow} u'$, we never obtain $t' < s' > u'$ because that would imply $s_i > s_{i+n}$. □

# 3 Maximal-Outermost Strategy

The strategy defined below is better known under the name *parallel-outermost*. We prefer to reserve the latter for the relation in which any number of outermost redexes is contracted.

**Definition 3.1.** Let $\mathcal{R}$ be a TRS. The *maximal-outermost* strategy $\mathrel{+\!\!\!\Rightarrow}_{\mathsf{mo}}$ is defined as follows:

(1) $x \mathrel{+\!\!\!\Rightarrow}_{\mathsf{mo}} x$ for all variables $x$,

(2) $f(s_1, \ldots, s_n) \mathrel{+\!\!\!\Rightarrow}_{\mathsf{mo}} f(t_1, \ldots, t_n)$ if $s_i \mathrel{+\!\!\!\Rightarrow}_{\mathsf{mo}} t_i$ for all $1 \leqslant i \leqslant n$ and $f(s_1, \ldots, s_n)$ is not a redex,

(3) $\ell\sigma \mathrel{+\!\!\!\Rightarrow}_{\mathsf{mo}} r\sigma$ if $\ell \to r \in \mathcal{R}$.

Unlike the maximal strategy, the maximal-outermost strategy is not cofinal, already for orthogonal TRSs.

**Example 3.2.** Consider the orthogonal TRS consisting of the rules $\mathsf{f}(x) \to \mathsf{f}(x)$ and $\mathsf{a} \to \mathsf{b}$. We have $\mathsf{f}(\mathsf{a}) \mathrel{+\!\!\!\Rightarrow}_{\mathsf{mo}} \mathsf{f}(\mathsf{a}) \mathrel{+\!\!\!\Rightarrow}_{\mathsf{mo}} \cdots$ and $\mathsf{f}(\mathsf{a}) \to \mathsf{f}(\mathsf{b})$ but $\mathsf{f}(\mathsf{b}) \to^* \mathsf{f}(\mathsf{a})$ does not hold.

Despite the lack of cofinality, the maximal-outermost strategy is normalising for decreasingly confluent TRSs. To prove this result, we classify parallel steps $\mathrel{+\!\!\!\Rightarrow}$. We write $\mathrel{+\!\!\!\Rightarrow}_{\mathsf{po}}$ for the (partial) parallel-outermost relation. An *inside* redex is a redex below an outermost redex and which does not overlap with an outermost redex. Parallel steps in which only inside redexes are contracted are denoted by $\mathrel{+\!\!\!\Rightarrow}_{\mathsf{pi}}$. The following lemma presents some properties of these relations for left-linear TRSs. For space reasons we refrain from giving the easy proofs.

**Lemma 3.3.** *Let $\mathcal{R}$ be a left-linear TRS.*

*(a) If $s \mathrel{\mathsf{pi}}\!\!\Rightarrow t \mathrel{\mathsf{po}}\!\!\Rightarrow u$ then $s \mathrel{\mathsf{po}}\!\!\Rightarrow \cdot \Rightarrow u$.*

*(b) If $s \mathrel{\mathsf{pi}}\!\!\Rightarrow t \in \mathsf{NF}(\mathcal{R})$ then $s = t$.*

*(c) If $t \mathrel{\mathsf{po}}\!\!\Leftarrow s \mathrel{\mathsf{mo}}\!\!\Rightarrow u$ and $s \not> u$ then $t \Rightarrow u$.*

*(d) If $t \mathrel{\mathsf{po}}\!\!\Leftarrow s \Rightarrow u$ and $s \not> u$ then $t \Rightarrow \cdot \mathrel{\mathsf{po}}\!\!\Leftarrow u$.* □

A crucial fact is that the parallel-outermost relation $\mathrel{\mathsf{po}}\!\!\Rightarrow$ suffices to compute normal forms.

**Lemma 3.4.** *Let $\mathcal{R}$ be a decreasingly confluent TRS. If $s \to^* u \in \mathsf{NF}(\mathcal{R})$ then $s \mathrel{\mathsf{po}}\!\!\Rightarrow^* u$.*

*Proof.* First we show that $s \mathrel{\mathsf{po}}\!\!\Rightarrow^* u$ when $s \Rightarrow t \mathrel{\mathsf{po}}\!\!\Rightarrow^n u \in \mathsf{NF}(\mathcal{R})$. We perform induction on $(s, n)$ with respect to the lexicographic product of $>$ and the standard order on $\mathbb{N}$. Suppose $s \Rightarrow t \mathrel{\mathsf{po}}\!\!\Rightarrow^n u \in \mathsf{NF}(\mathcal{R})$. We distinguish four cases.

- If $s \mathrel{\mathsf{po}}\!\!\Rightarrow t$ then $s \mathrel{\mathsf{po}}\!\!\Rightarrow^{n+1} u$.
- If $s \mathrel{\mathsf{pi}}\!\!\Rightarrow t$ and $n = 0$ then $s = u$ follows from Lemma 3.3(b).
- If $s \mathrel{\mathsf{pi}}\!\!\Rightarrow t$ and $n > 0$ then Lemma 3.3(a) yields $s \mathrel{\mathsf{po}}\!\!\Rightarrow \cdot \Rightarrow \cdot \mathrel{\mathsf{po}}\!\!\Rightarrow^{n-1} u$ and we obtain $s \mathrel{\mathsf{po}}\!\!\Rightarrow^* u$ from the induction hypothesis.
- In the remaining case the parallel step $s \Rightarrow t$ contracts at least one redex that is neither an outermost nor an inside redex. This can only happen if that redex overlaps with an outermost redex in $s$. So there is a term $t'$ with $s \mathrel{\mathsf{po}}\!\!\Rightarrow t'$ and $s \to_{\mathsf{CPS}(\mathcal{R})} t'$. Confluence yields $t' \to^* u$. Since $s > t'$ we can apply the induction hypothesis to obtain $t' \mathrel{\mathsf{po}}\!\!\Rightarrow^* u$. Hence also $s \mathrel{\mathsf{po}}\!\!\Rightarrow^* u$.

The lemma is obtained from the above statement by induction on the length of $s \to^* u$. □

Parallel steps on incomparable terms behave like those in orthogonal TRSs. The first part of the following lemma states a kind of triangle property for $\mathrel{\mathsf{mo}}\!\!\Rightarrow$. The left diagram in Figure 3 illustrates the proof for $n = 3$. The right diagram illustrates the proof of the second part of the lemma for $n = 3$.

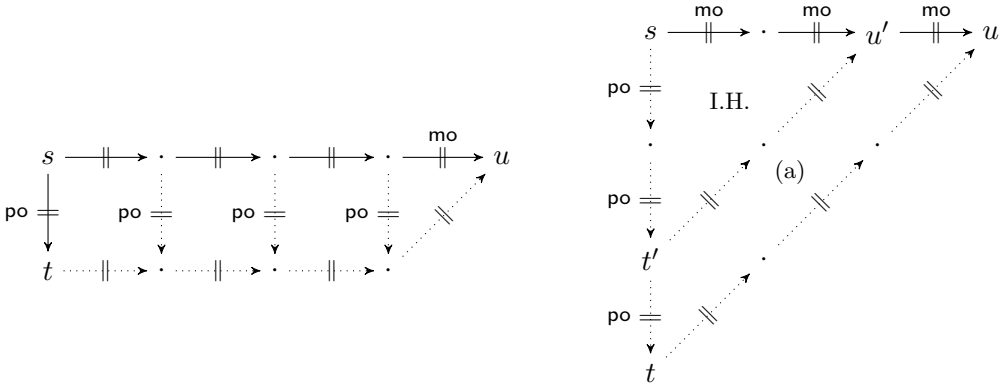



Figure 1: The proof of Lemma 3.5.

**Lemma 3.5.** *Let $\mathcal{R}$ be a left-linear TRS.*

*(a) If $t \ {}_{\mathsf{po}}\!\!\Leftarrow\!\!\Vdash s \Vdash^n \cdot \Vdash_{\mathsf{mo}} u$ and $s \not> u$ then $t \Vdash^{n+1} u$.*

*(b) If $t \ {}^{n}_{\mathsf{po}}\!\!\Leftarrow\!\!\Vdash s \Vdash^n_{\mathsf{mo}} u$ and $s \not> u$ then $t \Vdash^n u$.*

*Proof.* The first claim is obtained by induction on $n$ using Lemma 3.3(c,d). For the second claim we also use induction on $n$. If $n = 0$ then $s = t = u$. If $n > 0$ then $s \Vdash^{n-1}_{\mathsf{po}} t' \Vdash_{\mathsf{po}} t$ and $s \Vdash^{n-1}_{\mathsf{mo}} u' \Vdash_{\mathsf{mo}} u$ for some terms $t'$ and $u'$. Because $s \to^* u' \to^* u$ and $s \not> u$, also $s \not> u'$ and $u' \not> u$. The induction hypothesis yields $t' \Vdash^{n-1} u'$. Again, we deduce $t' \not> u$ from $s \to^* t' \to^* u' \to^* u$ and $s \not> u$. Because $t \ {}_{\mathsf{po}}\!\!\Leftarrow\!\!\Vdash t' \Vdash^{n-1} u' \Vdash_{\mathsf{mo}} u$ and $t' \not> u$, we conclude $t \Vdash^n u$ by part (a). □

We arrive at the main theorem.

**Theorem 3.6.** *The maximal-outermost strategy is normalizing for decreasingly confluent TRSs.*

*Proof.* Suppose $s_0 \Vdash_{\mathsf{mo}} s_1 \Vdash_{\mathsf{mo}} s_2 \Vdash_{\mathsf{mo}} \cdots$ and $s_0 \to^* t \in \mathsf{NF}(\mathcal{R})$. Due to the well-foundedness of $>$ there exists a $k \geqslant 0$ such that $s_k \not> s_m$ for all $m \geqslant k$. Confluence of $\mathcal{R}$ yields $s_k \to^* t \in \mathsf{NF}(\mathcal{R})$. Using Lemma 3.4 we have $s_k \Vdash^n_{\mathsf{po}} t$ for some $n \geqslant 0$. Since $s_k \Vdash^n_{\mathsf{mo}} s_{k+n}$ and $s_k \not> s_{k+n}$, Lemma 3.5(b) yields $t \Vdash^n s_{k+n}$. Because $t$ is a normal form, we conclude $s_{k+n} = t$. □

# 4 Conclusion

In this note we showed that the maximal strategy is cofinal (and thus normalising) for decreasingly confluent TRSs and that the maximal-outermost strategy is normalising for the same class of TRSs. The investigation whether other normalising strategies for orthogonal TRSs remain normalising for decreasingly confluent TRSs is left for future work. The same holds for root-normalisation [3] and infinitary normalisation [4].

**Acknowledgements** We are grateful to the anonymous referees for valuable comments.

# References

[1] B. Gramlich and S. Lucas. Generalizing Newman's lemma for left-linear rewrite systems. In *Proc. 17th RTA*, volume 4098 of *LNCS*, pages 66–80, 2006.

[2] N. Hirokawa and A. Middeldorp. Decreasing diagrams and relative termination. In *Proc. 5th IJCAR*, volume 6173 of *LNAI*, pages 487–501, 2010.

[3] A. Middeldorp. Call by need computations to root-stable form. In *Proc. 24th POPL*, pages 94–105. ACM Press, 1997.

[4] V. van Oostrom. Normalisation in weakly orthogonal rewriting. In *Proc. 10th RTA*, volume 1631 of *LNCS*, pages 60–74, 1999.

[5] Terese. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.

# Application of monadic substitution to recursive type containment

Vladimir Komendantsky
School of Computer Science
University of St Andrews
St Andrews, KY16 9SX, UK
vk10@st-andrews.ac.uk

**Abstract**

In this paper, we present a computer-checked, constructive soundness and completeness result for prototypic recursive type containment with respect to containment of non-wellfounded (finite or infinite) trees. The central role is played by formalisation of substitution of recursive types as a monad, with a traverse function implementing a strategy for potentially infinite recursive unfolding. In the rigorous setting of constructive type theory, the very definition of subtyping should be scrutinised to allow a prover to accept it. As a solution, we adapt the method of weak bisimilarity by mixed induction-coinduction recently introduced to Coq by Nakata and Uustalu. However, our setting is different in that we work with a notion of weak similarity that corresponds to subtyping. We accomplish the task of representing infinitary subtyping in the Calculus of (Co-)Inductive Constructions. Our technique does not require extensions of the Calculus and therefore can be ported to other dependently typed languages.

## 1 Introduction

Paper-and-pen proofs of soundness and completeness of recursive type containment (subtyping) with respect to models based on tree structures have been known since the early 1990's result of Amadio and Cardelli [2]. Over the years, their technique has been gradually refined and slightly simplified in presentation [3, 5]. Recently there has been a renewed interest in subtyping [4] which was also stimulated by progress in the area of dependently typed languages such as Agda or Coq. It becomes possible to capture subtyping in such languages, although traditional proofs are not necessarily straightforward to implement. Implementations rather take into account what is practically possible in type theory and what is not. Especially, this concerns notions of rewriting required for possibly infinite unfolding of recursive types.

Here we show that a prototypic subtype relation that can neither be defined as a least fixed point nor as a greatest fixed point alone can nevertheless be defined with inductive and coinductive types and an impredicative universe of propositions. One of the most interesting points is that the definition proceeds alike a fold in functional programming, although a rather unusual one, that is not applied to any starting object. This is an illustration of a very general programming pattern allowing to alternate least and greatest fixed points. The soundness and completeness result stated in the paper, apart from guaranteeing correctness of our definition of subtyping, has a technical significance of providing a mechanism of weak head normal forms that seems indispensable in situations when we need to "evaluate" a subtyping statement.

One possible application for subtyping in a language of $\mu$-types that motivates our work is coercive containment of regular expressions [6], a paradigm based on understanding regular expressions as types and concerned with a proof-theoretic interpretation of containment of languages denoted by regular expressions. This also motivates the choice of monadic presentations of $\mu$-types. Monadic presentations [1, 8, 7] are usually employed to study type-preserving substitutions. In the case of coercions, regular expressions themselves are modified, but words in the

underlying language of the source of the coercion are preserved in the language of the target of the coercion. So, in fact, we might speak of type-coercing rather than type-preserving substitutions. In this paper though, coercions are not yet spelt out in the object language, and we have to study how exactly we can extract the coercive content from proofs of subtyping statements.

Throughout the paper, we use a human-oriented type theoretic notation for the meta-language in which we define recursive types (the object language): $\star$ denotes the universe of types (which is predicative), $\star'$ denotes the universe of propositions (which is impredicative), and inductive and coinductive definitions are displayed in natural-deduction style with single and, respectively, double lines.

## 2 Recursive types

Below is the *traditional* definition of the set of *recursive types*. It is commonly defined by the following grammar [3, 2]:

$$E, F ::= \bot \mid \top \mid X \mid E \twoheadrightarrow F \mid \mu X.\ E \twoheadrightarrow F$$

where $X$ is a symbolic variable taken from a set of variables. The least fixed point operator $\mu$ binds free occurrences of the variable $X$ in $E \twoheadrightarrow F$. This traditional omission of products and sums that are needed for practical programming is due to the fact that their treatment is very similar to that of $\twoheadrightarrow$, see, e.g., [2]. The only true problem, however, is concerned with the $\mu$ operator.

For the purpose of type-theoretic encoding, we choose another, equivalent and yet more tangible definition of recursive types, where named variables are replaced by nameless de Brujin variables. We represent recursive types in Coq [10], by induction as follows:

$$\mathsf{ty} : \mathbb{N} \to \star$$

$$\frac{}{\bot : \mathsf{ty}\ n} \qquad \frac{}{\top : \mathsf{ty}\ n} \qquad \frac{i : \mathcal{I}_n}{X_i : \mathsf{ty}\ n} \qquad \frac{E : \mathsf{ty}\ n \qquad F : \mathsf{ty}\ n}{E \twoheadrightarrow F : \mathsf{ty}\ n} \qquad \frac{E : \mathsf{ty}\ (1+n) \qquad F : \mathsf{ty}\ (1+n)}{\mu\ E \twoheadrightarrow F : \mathsf{ty}\ n}$$

where the constructors are, respectively, the empty type $\bot$, the unit type $\top$, a variable, the function type constructor $\to$ and the least fixed point arrow type constructor $\mu\ \_ \to \_$. The dependent type $\mathcal{I}_n$ represents the first $n$ natural numbers, and therefore an object $i$ of type $\mathcal{I}_n$ is a pair consisting of a natural number $m$ and a proof of $m < n$. An interesting point is, if we compare $\mathcal{I}_n$ with the algebraic type of finite number employed by McBride in [8], that the relation $<$ on natural numbers can be expressed as a boolean relation, which allows to encode the whole proof of $m < n$ as a boolean value, and is very handy in case analysis.

Recursive types have a correspondence with non-wellfounded (finite or infinite) trees with the following definition by coinduction:

$$\mathsf{tree} : \mathbb{N} \to \star$$

$$\frac{}{\bot^\infty : \mathsf{tree}\ n} \qquad \frac{}{\top^\infty : \mathsf{tree}\ n} \qquad \frac{i : \mathcal{I}_n}{X_i^\infty : \mathsf{tree}\ n} \qquad \frac{t : \mathsf{tree}\ n \qquad u : \mathsf{tree}\ n}{t \twoheadrightarrow^\infty u : \mathsf{tree}\ n}$$

Intuitively, trees are views of $\mu$-types unfolded *ad infinitum*. We denote the tree corresponding to a type $E$ by $[\![E]\!]$. The straightforward subtree relation $\mathsf{tle}\ n$ on $\mathsf{tree}\ n$ is denoted by $\leq^\infty$ (by noting that we can infer the implicit argument $n$):

$$\mathsf{tle}\ n : \mathsf{tree}\ n \to \mathsf{tree}\ n \to \star'$$

$$\frac{}{\bot^\infty \leq^\infty t} \quad \frac{}{t \leq^\infty \top^\infty} \quad \frac{i : \mathcal{I}_n}{X_i^\infty \leq^\infty X_i^\infty} \quad \frac{u_1 \leq^\infty t_1 \qquad t_2 \leq^\infty u_2}{(t_1 \rightarrowtriangle^\infty t_2) \leq^\infty (u_1 \rightarrowtriangle^\infty u_2)}$$

Note that the subtree relation is contravariant in the first argument of $\rightarrowtriangle^\infty$ and covariant in the second. Two recursive types are in the subtype relation when their potentially infinite unfoldings are in the subtree relation. This is where monadic term presentations come into play. Instead of trying to define inductive limits of sequences of approximations of unfoldings of recursive types (as traditionally done in paper-and-pen proofs, e.g., in [2]), we encapsulate unfolding *ad infinitum* by relying on expressivity of dependent types of the CIC which allow to define this powerful monadic presentation structure. The point here, similar to a remark made by Amadio and Cardelli in [2], is that unfoldings of recursive types are *regular* trees, which we treat using a mix of induction and coinduction.

## 3 Monadic substitution

We implemented in Coq a generic notion of symbolic substitution introduced in [1] for untyped lambda terms. It is based on the notion of *universe of types*, that is, a function space $A \to \star$ where $A$ can be any given type and $\star$ is the polymorphic type of all types. $A$ is said to *index* the type $\star$. For effective indexing, the index type should be countable, and for that, it suffices to consider the type $\mathbb{N}$ of natural numbers. Following McBride [8], we call the resulting type of universe *stuff*:

$$\mathsf{stuff} : \star \qquad\qquad \mathsf{stuff} = \mathbb{N} \to \star$$

For a given $n$, the intended meaning of $\mathsf{stuff}\ n$ is *stuff with $n$ variables*.

In the foundation of the method, there is a type of monadic structure called *kit* [8, 7]:

$$\mathsf{kit} : \mathsf{stuff} \to \mathsf{stuff} \to \star$$

$$\frac{\mathit{var} : \forall\ n.\ \mathcal{I}_n \to U\ n \qquad \mathit{lift} : \forall\ n.\ U\ n \to T\ n \qquad \mathit{wk} : \forall\ n.\ U\ n \to U\ (1 + n)}{\mathsf{Kit}\ \mathit{var}\ \mathit{lift}\ \mathit{wk} : \mathsf{kit}\ U\ T}$$

A *substitution* of type $\mathsf{sub}\ T\ m\ n$ is such that it applies to stuff with at most $m$ variables and yields stuff with at most $n$ variables. Hence a substitution is essentially an $m$-tuple of $T\ n$, that is,

$$\mathsf{sub} : \mathsf{stuff} \to \mathbb{N} \to \mathbb{N} \to \star \qquad\qquad \mathsf{sub}\ T\ m\ n = m\text{-}\mathsf{tuple}\ (T\ n)$$

In order to establish compositionality on substitutions, we define applicative structure on substitutions which is called subApp:

$$\mathsf{subApp} : \mathsf{stuff} \to \star$$

$$\frac{\mathit{var} : \forall\ n.\ \mathcal{I}_n \to T\ n \qquad \mathit{app} : \forall\ U\ m\ n.\ \mathsf{kit}\ U\ T \to T\ m \to \mathsf{sub}\ U\ m\ n \to T\ n}{\mathsf{SubApp}\ \mathit{var}\ \mathit{app} : \mathsf{subApp}\ T}$$

A straightforward substitution strategy is implemented by the function trav below that traverses a term $E$ and applies a given substitution $s$. Note that, since $s$ is a tuple, $s_i$ is a consistent notation for the $i$-th element of $s$.

$$\begin{array}{lcl}
\mathsf{trav} & : & \forall\ T\ m\ n.\ \mathsf{kit}\ T\ \mathsf{ty} \to \mathsf{ty}\ m \to \mathsf{sub}\ T\ m\ n \to \mathsf{ty}\ n \\
\mathsf{trav}\ K\ \bot\ s & = & \bot \\
\mathsf{trav}\ K\ \top\ s & = & \top \\
\mathsf{trav}\ K\ X_i\ s & = & \mathsf{let}\ \mathsf{Kit}\ _\ li\ _ = K\ \mathsf{in}\ li\ _\ s_i \\
\mathsf{trav}\ K\ (F \rightarrowtriangle G)\ s & = & (\mathsf{trav}\ K\ F\ s) \rightarrowtriangle (\mathsf{trav}\ K\ G\ s) \\
\mathsf{trav}\ K\ (\mu\ F \rightarrowtriangle G)\ s & = & \mu\ (\mathsf{trav}\ K\ F\ (\mathsf{lift_sub}\ K\ s)) \rightarrowtriangle (\mathsf{trav}\ K\ G\ (\mathsf{lift_sub}\ K\ s))
\end{array}$$

Here, lift_sub is a function that lifts a substitution to the next order, that is, it shifts indices of the active variables in the substitution by +1. This function has type

$$\forall\ (T\ U : \mathsf{stuff})\ (K : \mathsf{kit}\ T\ U)\ m\ n.\ \mathsf{sub}\ T\ m\ n \to \mathsf{sub}\ T\ (1+m)\ (1+n)$$

What is the concrete structure of substitution on ty and why kit is indeed a monad will be explained in an extended version of this paper. A definition from the concrete structure of substitution, the function unfld that unfolds a $\mu$-redex, is used in the next section.

## 4 Weak similarity by mixed induction-coinduction

We define the weak similarity relation $\mathsf{tyle}\ n \subseteq \mathsf{ty}\ n \times \mathsf{ty}\ n$ by folding the inductive part of the definition into the coinductive one. Our technique is an illustration of a generic method for folding one relation in another. We use the impredicative universe of propositions that we denote by $\star'$, which is needed for the proof of soundness and completeness. First, we define the inductive part $\mathsf{tylei}\ n\ R$ of the subtyping relation (denoting $\mathsf{tylei}\ n\ R\ E\ F$ by $E \leq_R F$, suppressing the implicit argument $n$):

$$\mathsf{tylei} : \forall\ n.\ (\mathsf{ty}\ n \to \mathsf{ty}\ n \to \star') \to \mathsf{ty}\ n \to \mathsf{ty}\ n \to \star'$$

$$\frac{}{\bot \leq_R E} \quad \frac{}{E \leq_R \top} \quad \frac{R\ E\ F \qquad R\ G\ H}{F \twoheadrightarrow G \leq_R E \twoheadrightarrow H} \quad \frac{}{\mu\ E \twoheadrightarrow F \leq_R \mathsf{unfld}\ E\ F}$$

$$\frac{}{\mathsf{unfld}\ E\ F \leq_R \mu\ E \twoheadrightarrow F} \quad \frac{}{E \leq_R E} \quad \frac{E \leq_R F \qquad F \leq_R G}{E \leq_R G}$$

Having the rules for reflexivity and transitivity in the inductive part of the subtype relation is essential for this construction. Indeed, by having these rules explicitly, we are able to compare elements of pairs in the domain of the subtype relation in a *finite* number of steps possibly *infinitely*. Leaving transitivity out of the definition would collapse finite and infinite transitivity chains to infinite ones only, and the soundness argument would not work.

Next, we fold the inductive relation $\leq_R$ in a coinductive relation $\mathsf{tyle}\ n$ and produce a weak similarity (denoting $\mathsf{tyle}\ n\ E\ F$ by $E \leq F$):

$$\mathsf{tyle}\ n : \mathsf{ty}\ n \to \mathsf{ty}\ n \to \star'$$

$$\frac{\forall\ E\ F.\ R\ E\ F \to E \leq F \qquad E \leq_R F}{E \leq F}$$

The only introduction rule for $\leq$ has two hypotheses, namely, that $R$ is a subrelation of $\leq$, and that $F$ is $\leq_R$-accessible from $E$ in finitely many steps (since $\leq_R$ is an inductive relation).

## 5 Soundness and completeness

**Main Theorem** (Soundness and completeness)**.**

$$\forall\ (n : \mathbb{N})\ (E\ F : \mathsf{ty}\ n).\ E \leq F \leftrightarrow [\![E]\!] \leq^{\infty} [\![F]\!]$$

The "only if" direction (completeness) follows by a straightforward application of the coinduction principle. For the "if" direction (soundness), we define the weak head normal form (WHNF) of the relation $\leq^{\infty}$ and solve the problem via WHNFs, which is a common workaround helping with ensuring syntactic guardedness of the proof [4, 9].

## 6 Conclusions

We showed how monadic substitution can be used to define a rather simple case of subtyping relation: $\mu$-types without products or sums. Our study is closely related to the work of Altenkirch and Danielsson [4] who define subtyping using a suspension computation monad inspired by semantics of programming languages. The method with the suspension monad turns out not to be immediately applicable outside the special setting of [4] because of the way the termination checker works in Coq at the moment. Namely, there is no united checker for recursive and corecursive functions. Instead, there appear to be two independent mechanisms, one for checking structurality of recursive functions, and another for checking guardedness of corecursive ones. This leads to difficulties with mixed induction-coinduction. In fact, it would have been very beneficial for the termination checker of Coq if the suspension monad was implemented in it since that would require unifying the two parts together at last.

Here, we resolve these difficulties on top of the prover, following a very general method that allows to encode infinitary subtyping by folding the inductive relation $\leq_R$ into the coinductive $\leq$. Also, we remark that the presented approach of weak similarity is a natural solution to problems arising from declaring closure properties such as transitivity in coinductive relations that were discussed in [5]. Indeed, with our definitions, infinite transitivity chains do not arise.

**Acknowledgements.** I would like to thank Keiko Nakata and Niels Anders Danielsson for their help and advice regarding theorem proving and provers. The research is supported by the research fellowship EU FP7 Marie Curie IEF 253162 'SImPL'.

## References

[1] T. Altenkirch and B. Reus. Monadic presentations of lambda-terms using generalized inductive types. In *Computer Science Logic '99*, LNCS 1683, pages 453–468. Springer, 1999.

[2] R. M. Amadio and L. Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993.

[3] M. Brandt and F. Henglein. Coinductive axiomatization of recursive type equality and subtyping. In R. Hindley, editor, *Proc. 3rd Int. Conf. on Typed Lambda Calculi and Applications (TLCA)*, volume 1210 of *LNCS*, pages 63–81, Nancy, France, 1997. Springer-Verlag.

[4] N. A. Danielsson and T. Altenkirch. Subtyping, declaratively. In C. Bolduc, J. Desharnais, and B. Ktari, editors, *Mathematics of Program Construction*, volume 6120 of *Lecture Notes in Computer Science*, pages 100–118. Springer Berlin / Heidelberg, 2010. doi:10.1007/978-3-642-13321-3_8.

[5] V. Gapeyev, M. Y. Levin, and B. Pierce. Recursive subtyping revealed. *J. Fun. Prog.*, 12(6):511–548, 2002.

[6] F. Henglein and L. Nielsen. Regular Expression Containment: Coinductive Axiomatization and Computational Interpretation. In *Proc. 38th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, January, 2011.

[7] C. Keller and T. Altenkirch. Hereditary substitutions for simple types, formalized. In *Proceedings of the third ACM SIGPLAN workshop on Mathematically structured functional programming*, MSFP '10, pages 3–10. ACM, 2010.

[8] C. McBride. Type-preserving renaming and substitution, 2005. Manuscript.

[9] K. Nakata and T. Uustalu. Resumptions, weak bisimilarity and big-step semantics for While with interactive I/O: An exercise in mixed induction-coinduction. In *Proc. Structural Operational Semantics (SOS) 2010*, 2010.

[10] The Coq development team. The Coq proof assistant reference manual. `http://coq.inria.fr/refman/`.

# Productivity of Non-Orthogonal Term Rewrite Systems

Matthias Raffelsieper
Department of Computer Science, TU Eindhoven
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
eMail: M.Raffelsieper@tue.nl

**Abstract**

Productivity is the property that finite prefixes of an infinite constructor term can be computed using a given term rewrite system. Hitherto, productivity has only been considered for orthogonal systems, where non-determinism is not allowed. This paper presents techniques to also prove productivity of non-orthogonal term rewrite systems. For such systems, it is desired that one does not have to guess the reduction steps to perform, instead any outermost-fair reduction should compute an infinite constructor term in the limit. As a main result, it is shown that for possibly non-orthogonal term rewrite systems this kind of productivity can be concluded from context-sensitive termination.

## 1 Introduction

Productivity is the property that a given set of computation rules computes a desired infinite object. This has been studied mostly in the setting of *streams*, the simplest infinite objects. However, as already observed in [9], productivity is also of interest for other infinite structures, for example infinite trees, or mixtures of finite and infinite structures. A prominent example of the latter are lists in the programming language Haskell [7], which can be finite (by ending with a sentinel "[]") or which can go on forever.

Existing approaches for automatically checking productivity, e.g., [2, 3, 9], are restricted to *orthogonal* systems. The main reason for this restriction is that it disallows non-determinism. A complete computer program (i.e., a program and all possible input sequences, neglecting sources of true randomness) always behaves deterministically, as the steps of computation are precisely determined. However, often a complete program is not available, too large to be studied, or its inputs are provided by the user or they are not specified completely. In this case, non-determinism can be used to abstract from certain parts by describing a number of possible behaviors. In such a setting, the restriction to orthogonal systems, which is even far stronger than only disallowing non-determinism, should be removed. An example of such a setting are hardware components, describing streams of output values which are depending on the streams of input values. To analyze such a component in isolation, all possible input streams have to be considered.

This paper presents an extension of the techniques in [9] to analyze productivity of specifications that may contain non-determinism. These specifications are assumed to be given as *term rewrite systems (TRS)* [1], where the terms are considered to have two possible sorts. The first sort $d$ is for *data*. Terms of this sort represent the elements in an infinite structure, but which are not infinite terms by themselves. An example for data are the Booleans false and true, or the natural numbers represented in Peano form by the two constructors 0 and succ. The second sort is the sort $s$ for *structure*. Terms of this sort are to represent the intended structure containing the data and therefore are allowed to be infinite.

We still have to impose some restrictions on specifications to make our approach work. These restrictions are given below in the definition of *proper* specifications.

**Definition 1.** A *proper specification* is a tuple $\mathcal{S} = (\Sigma_d, \Sigma_s, \mathcal{C}, \mathcal{R}_d, \mathcal{R}_s)$, where $\Sigma_d$ is the signature of data symbols, each of type $d^m \to d$ (then the data arity of such a symbol $g$ is defined to be $\mathsf{ar}_d(g) = m$), $\Sigma_s$ is the signature of structure symbols $f$, which have types of the shape $d^m \times s^n \to s$ (and data arity $\mathsf{ar}_d(f) = m$, structure arity $\mathsf{ar}_s(f) = n$), $\mathcal{C} \subseteq \Sigma_s$ is a set of *constructors*, $\mathcal{R}_d$ is a terminating TRS over the signature $\Sigma_d$, and $\mathcal{R}_s$ is a TRS containing rules $f(u_1, \ldots, u_m, t_1, \ldots, t_n) \to t$ satisfying the following properties:

- $f \in \Sigma_s \setminus \mathcal{C}$ with $\mathsf{ar}_d(f) = m$, $\mathsf{ar}_s(f) = n$,
- $f(u_1,\ldots,u_m,t_1,\ldots,t_n)$ is a well-sorted linear term, $t$ is a well-sorted term of sort $s$, and
- for all $1 \leq i \leq n$ and for all $p \in \mathrm{Pos}(t_i)$ such that $t_i|_p$ is not a variable and $\mathrm{root}(t_i|_p) \in \Sigma_s$, it holds that $\mathrm{root}(t_i|_{p'}) \notin \mathcal{C}$ for all $p' < p$ (i.e., no structure symbol is below a constructor).

Furthermore, $\mathcal{R}_s$ is required to be *exhaustive*, meaning that for every $f \in \Sigma_s \setminus \mathcal{C}$ with $\mathsf{ar}_d(f) = m$, $\mathsf{ar}_s(f) = n$, ground normal forms $u_1,\ldots,u_m \in \mathrm{NF}_{\mathrm{gnd}}(\mathcal{R}_d)$, and terms $t_1,\ldots,t_n \in \mathcal{T}(\Sigma_d \cup \Sigma_s)$ such that for every $1 \leq i \leq n$, $t_i = c_i(u'_1,\ldots,u'_k,t'_1,\ldots,t'_l)$ with $u'_j \in \mathrm{NF}_{\mathrm{gnd}}(\mathcal{R}_d)$ for $1 \leq j \leq k = \mathsf{ar}_d(c_i)$ and $c_i \in \mathcal{C}$, there exists at least one rule $\ell \to r \in \mathcal{R}_s$ such that $\ell$ matches the term $f(u_1,\ldots,u_m,t_1,\ldots,t_n)$.

A proper specification $\mathcal{S}$ is called *orthogonal*, if $\mathcal{R}_d \cup \mathcal{R}_s$ is orthogonal, otherwise it is called *non-orthogonal*.

Nesting of structure symbols inside constructors on left-hand sides of $\mathcal{R}_s$ is disallowed for technical reasons, however it is not a severe restriction in practice. Often, it can be achieved by unfolding the specification, as was presented in [4, 8].

The above definition coincides with the definition of proper specifications given in [9] for orthogonal proper specifications.[1] For such orthogonal proper specifications, productivity is the property that every ground term $t$ of sort $s$ can, in the limit, be rewritten to a possibly infinite term consisting only of constructors. This is equivalent to stating that for every prefix depth $d \in \mathbb{N}$, the term $t$ can be rewritten to another term $t'$ having only constructor symbols on positions of depth $d$ or less. It was already observed in [2] that productivity of orthogonal specifications is equivalent to the existence of an *outermost-fair* reduction computing a constructor prefix for any given depth. Below, we give a general definition of outermost-fair reductions, as they will also be used in the non-orthogonal setting.

**Definition 2.**

- A *redex* is a subterm $t|_p$ of a term $t$ at position $p \in \mathrm{Pos}(t)$ such that a rule $\ell \to r$ and a substitution $\sigma$ exist with $t|_p = \ell\sigma$. The redex $t|_p$ is said to be *matched* by the rule $\ell \to r$.
- A redex is called *outermost* iff it is not a strict subterm of another redex.
- A redex $t|_p = \ell\sigma$ is said to *survive* a reduction step $t \to_{\ell' \to r', q} t'$ if $p \parallel q$, or if $p < q$ and $t' = t[\ell\sigma']_p$ for some substitution $\sigma'$ (i.e., the same rule can still be applied at $p$).
- A rewrite sequence (reduction) is called *outermost-fair*, iff there is no outermost redex that survives as an outermost redex infinitely long.
- A rewrite sequence (reduction) is called *maximal*, iff it is infinite or ends in a *normal form* (a term that cannot be rewritten further).

For non-orthogonal proper specifications, requiring just the existence of a reduction to a constructor prefix of arbitrary depth does not guarantee an outermost-fair rewrite sequence to reach it, due to the possible non-deterministic choices.

**Example 3.** Consider a proper specification with the TRS $\mathcal{R}_s$ consisting of the following rules:

$$\begin{array}{lcl@{\qquad}lcl} \mathsf{maybe} & \to & 0 : \mathsf{maybe} & \mathsf{random} & \to & 0 : \mathsf{random} \\ \mathsf{maybe} & \to & \mathsf{maybe} & \mathsf{random} & \to & 1 : \mathsf{random} \end{array}$$

This specification is not orthogonal, since the rules for maybe and those for random overlap. We do not want to call this specification productive, since it admits the infinite outermost-fair reduction $\mathsf{maybe} \to \mathsf{maybe} \to \ldots$ that never produces any constructors. However, there exists an infinite reduction producing infinitely many constructors starting in the term maybe, namely $\mathsf{maybe} \to 0 : \mathsf{maybe} \to 0 : 0 : \mathsf{maybe} \to \ldots$. When only considering the rules for random then we want to call the resulting specification productive, since no matter what rule of random we choose, an element of the stream is created.

[1] To see this, observe that a defined symbol cannot occur on a non-root position of a left-hand side. Otherwise, such a subterm would unify with some left-hand side due to exhaustiveness, which would contradict orthogonality.

Requiring just the existence of a constructor normal form is called *weak productivity* in [2]. We already stated above that this is not the notion of productivity we are interested in, since it requires to "guess" the reduction steps leading to a constructor term. The notion we are interested in is *strong productivity*, which requires all outermost-fair reductions to reach a constructor term. This kind of productivity was also defined in [2].

**Definition 4.** A proper specification $\mathcal{S}$ is called *strongly productive* iff for every ground term $t$ of sort $s$ all maximal outermost-fair rewrite sequences starting in $t$ end in (i.e., have as limit for infinite sequences) a constructor normal form.

Thus, the term maybe in Example 3 is not strongly productive, whereas the term random in that example is strongly productive. It was shown in [2] that weak and strong productivity coincide for orthogonal proper specifications.

## 2 Criteria for Strong Productivity

An orthogonal proper specification is productive if and only if a reduction exists that creates a constructor at the top, as was shown in [9]. This is also the case for non-orthogonal proper specifications. However, here we have to consider all maximal outermost-fair reductions, instead of just requiring the existence of such a reduction. Hence, the characterization for strong productivity used in this paper is the following.

**Proposition 5.** *Let $\mathcal{S}$ be a proper specification. Then $\mathcal{S}$ is* strongly productive *iff for every maximal outermost-fair reduction $t_0 \to_{\mathcal{R}_d \cup \mathcal{R}_s} t_1 \to_{\mathcal{R}_d \cup \mathcal{R}_s} \dots$ with $t_0$ being of sort $s$ there exists $k \in \mathbb{N}$ such that* $\mathrm{root}(t_k) \in \mathcal{C}$.

This characterization of productivity leads to a first technique to prove strong productivity of proper specifications. It is a simple syntactic check that determines whether every right-hand side of sort $s$ starts with a constructor. For orthogonal proper specifications, this was already observed in [9].

**Theorem 6.** *Let $\mathcal{S}$ be a proper specification. If for all rules $\ell \to r \in \mathcal{R}_s$ we have* $\mathrm{root}(r) \in \mathcal{C}$*, then $\mathcal{S}$ is strongly productive.*

The above criterion is sufficient to prove strong productivity of the proper specification consisting of the two rules for random in Example 3, since both have right-hand sides with the constructor : at the root. However, it is easy to create examples which are strongly productive, but do not satisfy the syntactic requirements of Theorem 6.

**Example 7.** Consider the proper specification with the following TRS $\mathcal{R}_s$:

$$\begin{array}{rcl@{\qquad}rcl@{\qquad}rcl} \mathsf{ones} & \to & 1 : \mathsf{ones} & \mathsf{finZeroes} & \to & 0:0:\mathsf{ones} & \mathsf{f}(0:xs) & \to & \mathsf{f}(xs) \\ \mathsf{finZeroes} & \to & 0 : \mathsf{ones} & \mathsf{finZeroes} & \to & 0:0:0:\mathsf{ones} & \mathsf{f}(1:xs) & \to & 1:\mathsf{f}(xs) \end{array}$$

The constant finZeroes produces non-deterministically a stream that starts with one, two, or three zeroes followed by an infinite stream of ones. Function f takes a binary stream as argument and filters out all occurrences of zeroes. Thus, productivity of this example proves that only a finite number of zeroes can be produced. This however cannot be proven with the technique of Theorem 6, since the right-hand side of the rule $\mathsf{f}(0:xs) \to \mathsf{f}(xs)$ does not start with the constructor :.

Another technique presented in [9] to show productivity of orthogonal proper specifications is based on context-sensitive termination. The idea is to disallow rewriting in structure arguments of constructors, thus context-sensitive termination implies that for every ground term of sort $s$, a term starting with a constructor can be reached (due to the exhaustiveness requirement). As was observed by Endrullis and Hendriks recently in [4], this set of blocked positions can be enlarged, making the approach even stronger.

Below, the technique for proving productivity by showing termination of a corresponding context-sensitive TRS is extended to also be applicable in the case of our more general proper specifications. This version already includes an adaption of the improvement mentioned above.

**Definition 8.** Let $\mathcal{S}$ be a proper specification. The replacement map $\mu_{\mathcal{S}} : \Sigma_d \cup \Sigma_s \to 2^{\mathbb{N}}$ is defined as follows: [2]

- $\mu_{\mathcal{S}}(f) = \{1, \ldots, \mathsf{ar}_d(f)\}$, if $f \in \Sigma_d \cup \mathcal{C}$
- $\mu_{\mathcal{S}}(f) = \{1, \ldots, \mathsf{ar}_d(f) + \mathsf{ar}_s(f)\} \setminus \{1 \leq i \leq \mathsf{ar}_d(f) + \mathsf{ar}_s(f) \mid t|_i \text{ is a variable for all } \ell \to r \in \mathcal{R}_s \text{ and all non-variable subterms } t \text{ of } \ell \text{ with } \mathrm{root}(t) = f\}$,[3] otherwise

In the remainder, we leave out the subscript $\mathcal{S}$ if the specification is clear from the context. The replacement map $\mu$ is used to define the set of *allowed* positions of a non-variable term $t$ as $\mathrm{Pos}_\mu(t) = \{\varepsilon\} \cup \{i.p \mid i \in \mu(\mathrm{root}(t)), p \in \mathrm{Pos}_\mu(t|_i)\}$. This replacement map is *canonical* [6] for the left-linear TRS $\mathcal{R}_s$, guaranteeing that non-variable positions of left-hand sides are allowed. We extend it to the non-left-linear TRS $\mathcal{R}_d \cup \mathcal{R}_s$ by allowing all arguments of symbols from $\Sigma_d$. Context-sensitive rewriting is the restriction of the rewrite relation to those redexes on positions from $\mathrm{Pos}_\mu$. Formally, $t \xrightarrow{\mu}_{\ell \to r, p} t'$ iff $t \to_{\ell \to r, p} t'$ and $p \in \mathrm{Pos}_\mu(t)$ and we say a TRS $\mathcal{R}$ is *$\mu$-terminating* iff no infinite $\xrightarrow{\mu}_{\mathcal{R}}$-chain exists.

Our main result of this paper is that also for possibly non-orthogonal proper specifications, $\mu$-termination implies productivity.

**Theorem 9.** *A proper specification $\mathcal{S} = (\Sigma_d, \Sigma_s, \mathcal{C}, \mathcal{R}_d, \mathcal{R}_s)$ is strongly productive, if $\mathcal{R}_d \cup \mathcal{R}_s$ is $\mu_{\mathcal{S}}$-terminating.*

It can be shown that Theorem 9 subsumes Theorem 6, since for a TRS $\mathcal{R}_s$ with $\mathrm{root}(r) \in \mathcal{C}$ for all $\ell \to r \in \mathcal{R}_s$, the only allowed positions on right-hand sides are of sort $d$ and $\mathcal{R}_d$ is terminating. The technique of Theorem 9, i.e., proving $\mu$-termination of the corresponding context-sensitive TRS, is able to prove strong productivity of Example 7. This can for example be seen by typing the corresponding context-sensitive TRS into a modern termination tool such as AProVE [5].

Checking productivity in this way, i.e., by checking context-sensitive termination, can only prove productivity but not disprove it. This is illustrated in the next example.

**Example 10.** Consider the proper specification with the following rules in $\mathcal{R}_s$:

$$\mathsf{a} \to \mathsf{f}(\mathsf{a}) \qquad \mathsf{f}(x : xs) \to x : \mathsf{f}(xs) \qquad \mathsf{f}(\mathsf{f}(xs)) \to 1 : xs$$

Starting in the term $\mathsf{a}$, we observe that an infinite $\mu$-reduction starting with $\mathsf{a} \to \mathsf{f}(\underline{\mathsf{a}})$ exists, which can be continued by reducing the underlined redex repeatedly, since $\mu(\mathsf{f}) = \{1\}$. Thus, the example is not $\mu$-terminating. However, the specification is productive, as can be shown by a case analysis based on the root symbol of some arbitrary ground term $t$. In case $\mathrm{root}(t) = :$, then nothing has to be done, according to Proposition 5. Otherwise, if $\mathrm{root}(t) = \mathsf{a}$, then any maximal outermost-fair reduction must start with $t = \mathsf{a} \to \mathsf{f}(\mathsf{a})$, thus we can reduce our analysis to the final case, where $\mathrm{root}(t) = \mathsf{f}$. In this last case, $t = \mathsf{f}(t')$. Due to the rules for the symbol $\mathsf{f}$, another case analysis is performed for $t'$. If $\mathrm{root}(t') = :$, then this

[2] Note that in [4], Endrullis and Hendriks consider orthogonal TRSs and also block arguments of symbols in $\Sigma_d$ that only contain variables on left-hand sides of $\mathcal{R}_d$. This however is problematic when allowing non-left-linear data rules. Example:

$$\begin{array}{lrcl\quad lrcl\quad lrcl} \mathcal{R}_s: & \mathsf{f}(1) & \to & \mathsf{f}(\mathsf{d}(0,\mathsf{d}(1,0))) & \mathsf{f}(0) & \to & \mathsf{c} \in \mathcal{C} & \\ \mathcal{R}_d: & \mathsf{d}(x,x) & \to & 1 & \mathsf{d}(0,x) & \to & 0 & \mathsf{d}(1,x) \to 0 \end{array}$$

Here, the term $\mathsf{f}(\mathsf{d}(0,\mathsf{d}(1,0)))$ can only be $\mu$-rewritten to the term $\mathsf{f}(0)$ (which then in turn has to be rewritten to $\mathsf{c}$) if defining $\mu(\mathsf{d}) = \{1\}$, since the subterm $\mathsf{d}(1,0)$ can never be rewritten to $0$. However, the example is not strongly productive, as reducing in this way gives rise to an infinite outermost-fair reduction $\mathsf{f}(\mathsf{d}(0,\mathsf{d}(1,0))) \to \mathsf{f}(\mathsf{d}(0,0)) \to \mathsf{f}(1) \to \ldots$. Blocking arguments of data symbols can only be done when $\mathcal{R}_d$ is left-linear.

[3] The requirement of $t$ not being a variable ensures that $\mathrm{root}(t)$ is defined.

constructor cannot be reduced further. Hence, in any maximal outermost-fair reduction sequence a redex of the form $\mathsf{f}(\tilde{u} : \tilde{t})$ exists at the root, until it is eventually reduced using the second rule which results in a term with the constructor : at the root. For $\mathrm{root}(t') = a$ we again must reduce $t = \mathsf{f}(\mathsf{a}) \to \mathsf{f}(\mathsf{f}(\mathsf{a}))$. Finally, in case $t = \mathsf{f}(t') = \mathsf{f}(\mathsf{f}(t''))$, we have two possibilities. The first one occurs when the term $t'$ is eventually reduced at the root. Since $\mathrm{root}(t') = \mathsf{f}$, this has to happen with either of the $\mathsf{f}$-rules, creating a constructor : which, as we already observed, must eventually result in the term $t$ also being reduced to a term with the constructor : at the root. Otherwise, in the second possible scenario, the term $t'$ is never reduced at the root. Then however, an outermost redex of the shape $\mathsf{f}(\mathsf{f}(\tilde{t}))$ exists in all terms that $t$ can be rewritten to in this way, thus it has to be reduced eventually with the third rule. This again creates a term with constructor : at the root. Combining all these observations, we see that in every maximal outermost-fair reduction there exists a term with the constructor : as root symbol, which proves productivity due to Proposition 5.

## 3 Conclusions and Future Work

We have presented a generalization of the productivity checking techniques in [9] (including an improvement observed in [4]) to non-orthogonal specifications, which are able to represent non-deterministic systems. These naturally arise when abstracting away certain details of an implementation, such as for example concrete sequences of input values. A main difference to the orthogonal setting is that in the non-orthogonal case, a single reduction to a constructor term is not sufficient, instead all outermost-fair reductions must be considered.

Our non-orthogonal setting still imposes certain restrictions on the specifications that can be treated. The most severe restriction is the requirement of left-linear rules in $\mathcal{R}_s$. Dropping this requirement however would make Theorem 9 unsound. Similarly, also the requirement that structure arguments of constructors must be variables cannot be dropped without losing soundness of Theorem 9. This requirement however is not that severe in practice, since many specifications can be unfolded by introducing fresh symbols [4, 8].

In the future, it would be interesting to investigate whether transformations of non-orthogonal proper specifications, similar to those in [9], can be defined. It is clear that rewriting of right-hand sides for example is not productivity-preserving for non-orthogonal specifications, since it only considers one possible reduction. However, it would be interesting to investigate whether for example narrowing of right-hand sides is productivity preserving, as it considers all possible reductions.

## References

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

[2] J. Endrullis. *Termination and Productivity*. PhD thesis, Vrije Universiteit Amsterdam, 2010.

[3] J. Endrullis, C. Grabmayer, and D. Hendriks. Data-Oblivious Stream Productivity. In *Proceedings of LPAR'08*, volume 5330 of *Lecture Notes in Computer Science*, pages 79–96. Springer-Verlag, 2008.

[4] J. Endrullis and D. Hendriks. Lazy Productivity via Termination. *Theoretical Computer Science*, 2011. doi:10.1016/j.tcs.2011.03.024.

[5] J. Giesl, P. Schneider-Kamp, and R. Thiemann. AProVE 1.2: Automatic Termination Proofs in the Dependency Pair Framework. In *Proceedings of IJCAR'06*, volume 4130 of *Lecture Notes in Computer Science*, pages 281–286. Springer-Verlag, 2006. Web interface: `http://aprove.informatik.rwth-aachen.de`.

[6] S. Lucas. Context-Sensitive Rewrite Strategies. *Information and Computation*, 178(1):294–343, 2002.

[7] S. Peyton Jones. Haskell 98 *Language and Libraries: The revised report*. Cambridge University Press, 2003.

[8] H. Zantema. Well-Definedness of Streams by Termination. In *Proceedings of RTA'09*, volume 5595 of *Lecture Notes in Computer Science*, pages 164–178. Springer-Verlag, 2009.

[9] H. Zantema and M. Raffelsieper. Proving Productivity in Infinite Data Structures. In *Proceedings of RTA'10*, volume 6 of *LIPIcs*, pages 401–416. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2010.

# Strategy Independent Reduction Lengths in Rewriting and Binary Arithmetic

Hans Zantema
University of Technology, Eindhoven, The Netherlands
Radboud University, Nijmegen, The Netherlands
h.zantema@tue.nl

**Abstract**

In this paper we give a criterion by which one can conclude that every reduction of a basic term to normal form has the same length. As a consequence, the number of steps to reach the normal form is independent of the chosen strategy. In particular this holds for TRSs computing addition and multiplication of natural numbers, both in unary and binary notation.

## 1 Introduction

For many term rewriting systems (TRSs) the number of rewrite steps to reach a normal form strongly depends on the chosen strategy. For instance, in using the standard if-then-else-rules

$$\mathsf{if}(\mathsf{true}, x, y) \to x \quad \mathsf{if}(\mathsf{false}, x, y) \to y$$

it is a good strategy to first rewrite the leftmost (boolean) argument of the symbol $\mathsf{if}$ until this argument is rewritten to $\mathsf{false}$ or $\mathsf{true}$ and then apply the corresponding rewrite rule for $\mathsf{if}$. In this way redundant reductions in the third argument are avoided in case the condition rewrites to $\mathsf{true}$, and redundant reductions in the second argument are avoided in case the condition rewrites to $\mathsf{false}$. More general, choosing a good reduction strategy is essential for doing efficient computation by rewriting. Roughly speaking, for erasing rules, that is, some variable in the left-hand side does not appear in the right-hand side, it seems a good strategy to postpone rewriting this possibly erasing argument, as is the case in the above if-then-else example. Conversely, in case of duplicating rules, that is, a variable occurs more often in the right-hand side than in the left-hand side, it seems a good strategy to first rewrite the corresponding argument before duplicating it.

Surprisingly however, there are practical examples of TRSs including both such erasing and duplicating rules, where the strategy has no influence at all on the number steps required to reach a normal form. In this paper we investigate criteria for this phenomenon. As an example, using the results of this paper, we will show that for multiplying two natural numbers by the standard TRS

$$\begin{array}{rcl@{\qquad}rcl} \mathsf{plus}(\mathsf{0}, x) & \to & x & \mathsf{mult}(\mathsf{0}, x) & \to & \mathsf{0} \\ \mathsf{plus}(\mathsf{s}(x), y) & \to & \mathsf{s}(\mathsf{plus}(x, y)) & \mathsf{mult}(\mathsf{s}(x), y) & \to & \mathsf{plus}(\mathsf{mult}(x, y), y) \end{array}$$

the number of steps to reach the resulting normal form is independent of the chosen strategy. Note that the third rule of this TRS is erasing, and the last rule is duplicating.

In this TRS ground terms reduce to ground normal forms of the shape $\mathsf{s}^n(\mathsf{0})$ for any natural number $n$, representing the corresponding number $n$ in unary notation. For large numbers $n$ this is a quite inefficient representation. Much more efficient is the binary representation. In this paper we also give a TRS for doing addition and multiplication in binary representation. Here for every addition or multiplication of two positive integer numbers we show by the main theorem

of this paper that every reduction to normal form has the same number of steps. Moreover, this number has the same complexity as the standard binary algorithms: linear for addition and quadratic for multiplication.

Through the paper we assume familiarity with standard notions in rewriting like orthogonality and normal forms as they are introduced in e.g. [3]. For instance, a *reduction* is a sequence of rewrite steps, and a *reduct* of a term $t$ is a term $u$ for which there is a reduction from $t$ to $u$.

In Section 2 we investigate the diamond property and present our main result Theorem 3: a criterion by which for every basic term all reductions to normal form have the same length. Here a term is called basic if only its root is a defined symbol. We apply this to unary arithmetic. In Section 3 it is shown how our theorem applies to binary arithmetic. We conclude in Section 4.

## 2 The diamond property and the main result

We say that a binary relation $\to$ satisfies the *diamond property*, if for every three elements $s, t, u$ satisfying $s \to t$ and $s \to u$ and $t \neq u$, there exists an element $w$ such that $t \to w$ and $u \to w$.

This notion is slightly different from other variants of the diamond property, for instance introduced in [3] where it is typically used for reflexive relations. However, as we are interested in the exact number of steps to reach a normal form, our present version is the most natural. It is an instance of the *balanced weak Church-Rosser property* from [4]. The following lemma is an immediate consequence of Lemma 1 from [4].

**Lemma 1.** *Let $\to$ be a relation satisfying the diamond property. Then every element has at most one normal form, and for every element having a normal form it holds that every reduction of this element to its normal form has the same number of steps.*

An orthogonal TRS is said to be *variable preserving*[1] if for every rule $\ell \to r$ every variable occurring in $\ell$ occurs exactly once in $r$. For instance, the TRS consisting of the two rules for $\mathsf{plus}$ as given in the introduction, is variable preserving.

**Lemma 2.** *Let $R$ be a variable preserving orthogonal TRS. Then the relation $\to_R$ on the set of all terms satisfies the diamond property.*

*Proof.* (sketch)

This follows by analyzing all cases as they occur in the proof of the critical pair lemma ([3], Lemma 2.7.15): for the case of disjoint redexes it is immediate, for the case of nestingness it follows from variable preservation, and the third case of overlap does not occur due to orthogonality., for the case of nestingness it follows from variable preservation, and the third case of overlap does not occur due to orthogonality. □

As a direct consequence of Lemmas 1 and 2 we conclude that for all terms with respect to the two $\mathsf{plus}$ rules from the introduction the number of steps to reach the normal form is independent of the strategy. This does not hold any more for the full system also containing the rules for $\mathsf{mult}$. For instance, the term $\mathsf{mult}(\mathsf{0}, \mathsf{plus}(\mathsf{0}, \mathsf{0}))$ admits the following two reductions to normal form having lengths one and two, respectively:

$$\mathsf{mult}(\mathsf{0}, \mathsf{plus}(\mathsf{0}, \mathsf{0})) \to_R \mathsf{0}, \quad \mathsf{mult}(\mathsf{0}, \mathsf{plus}(\mathsf{0}, \mathsf{0})) \to_R \mathsf{mult}(\mathsf{0}, \mathsf{0}) \to_R \mathsf{0}.$$

[1] Some texts have a weaker notion of *variable preserving*, but our version is more suitable for investigating reduction lengths.

However, here the starting term $\mathsf{mult}(\mathsf{0},\mathsf{plus}(\mathsf{0},\mathsf{0}))$ is not *basic*: it contains more than one *defined symbol*. A symbol is called a *defined symbol* if it occurs as the root of the left-hand side of a rule. Similar as in texts on runtime complexity like [1] we define:

> A term is defined to be *basic* if the root is a defined symbol, and it is the only defined symbol occurring in the term.

We will prove that for basic terms like terms of the shape $\mathsf{mult}(\mathsf{s}^m(\mathsf{0}),\mathsf{s}^n(\mathsf{0}))$ every reduction to its normal form $\mathsf{s}^{mn}(\mathsf{0})$ has the same length.

**Theorem 3.** *Let $R$ be an orthogonal TRS over $\Sigma$, and $\Sigma' \subseteq \Sigma$, such that every rule $\ell \to r$ of $R$ is of one of the following shapes:*

- *$\ell \to r$ is variable preserving, and neither $\ell$ nor $r$ contain symbols from $\Sigma'$,*
- *the root of $\ell$ is in $\Sigma'$, and for every symbol in $r$ from $\Sigma'$ the arguments of this symbol do not contain defined symbols.*

*Then any two reductions of a basic term to normal form have the same length.*

*Proof.* (sketch)
Due to the shape of the rules, starting from a basic term the following property remains invariant during rewriting:

> For every symbol from $\Sigma'$ in the term, the arguments of this symbol do not contain defined symbols.

So every reduct of a basic term satisfies this invariant. Next we show that the relation $\to_R$ restricted to terms satisfying this invariant, satisfies the diamond property. To prove this, let such a term $s$ both rewrite to $t$ and to $u$, $t \neq u$. In case the redexes are parallel, then the diamond property is easily concluded. In the remaining case one redex is above the other. Due to the invariant and the shape of the rules, one of the following cases holds:

- The roots of both redexes are not in $\Sigma'$. Then both reduction steps are with respect to variable preserving rules, and the diamond property follows from Lemma 2.
- The root of the innermost (of the two) redexes is in $\Sigma'$, but the root of the outermost redex is not. Then the reduction with respect to the outermost redex is variable preserving, by which the diamond property can be concluded.

In all cases the diamond property can be concluded. Now the theorem follows from Lemma 1. □

Indeed now by Theorem 3 we can conclude that with respect to the TRS with $\mathsf{plus}$ and $\mathsf{mult}$ as given in the introduction for every basic term of the shape $\mathsf{mult}(\mathsf{s}^m(\mathsf{0}),\mathsf{s}^n(\mathsf{0}))$ every reduction to its normal form $\mathsf{s}^{mn}(\mathsf{0})$ has the same length: let $\Sigma'$ consist of the single symbol $\mathsf{mult}$. Then the two rules for $\mathsf{plus}$ satisfy the condition for rules of the first type, and the two rules for $\mathsf{mult}$ satisfy the condition for rules of the second type, by which the claim follows from Theorem 3.

As another example consider the Fibonacci function $\mathsf{fib}$ defined by the rules

$$\begin{array}{rcl@{\qquad}rcl}
\mathsf{plus}(\mathsf{0},x) & \to & x & \mathsf{fib}(\mathsf{0}) & \to & \mathsf{0}\\
\mathsf{plus}(\mathsf{s}(x),y) & \to & \mathsf{s}(\mathsf{plus}(x,y)) & \mathsf{fib}(\mathsf{s}(\mathsf{0})) & \to & \mathsf{s}(\mathsf{0})\\
 & & & \mathsf{fib}(\mathsf{s}(\mathsf{s}(x))) & \to & \mathsf{plus}(\mathsf{fib}(x),\mathsf{fib}(\mathsf{s}(x))).
\end{array}$$

Choosing $\Sigma'$ to consist only of the symbol $\mathsf{fib}$, all requirements of Theorem 3 hold, so we conclude that for every $k$ every reduction of $\mathsf{fib}(\mathsf{s}^k(\mathsf{0}))$ to normal form has the same length.

The converse of Theorem 3 does not hold, not even for terminating orthogonal constructor systems. For instance, for the TRS consisting of the three rules

$$\mathsf{f}(\mathsf{0}) \to \mathsf{0}, \quad \mathsf{f}(\mathsf{s}(x)) \to \mathsf{s}(\mathsf{0}), \quad \mathsf{g}(x) \to \mathsf{f}(\mathsf{f}(x))$$

it is easily proved that any two reductions of a basic term to normal form have the same length, while the conditions of Theorem 3 do not hold.

## 3 Binary arithmetic

For more efficient computation of numbers it is natural to exploit binary notation, in which the size of the representation is logarithmic rather than linear in the value of the number. In standard binary notation positive integers can be seen to be uniquely composed from a constant 1 representing value 1, and two unary operators .0 and .1, where .0 means putting a 0 behind the number, by which its value is duplicated, and .1 means putting a 1 behind the number, by which its value $x$ is replaced by $2x + 1$.

Every positive integer has a unique representation as a ground term over these three symbols 1, .0 and .1, corresponding to the usual binary notation in which a postfix notation for .0 and .1 is used. For instance, the number 29 is 11101 in binary notation, and is written as 1.1.1.0.1 as a postfix ground term. Since we use prefix notation as the standard, instead for 29 we write .1(.0(.1(.1(1)))). Introducing a constant 0 wouls violate unicity, that's why we restrict to positive integers.

In order to express addition and multiplication in this notation the successor succ is needed as an additional operator, having rewrite rules

$$\begin{array}{rcl} \mathsf{succ}(\mathsf{1}) & \to & .\mathsf{0}(\mathsf{1}) \\ \mathsf{succ}(.\mathsf{0}(x_p)) & \to & .\mathsf{1}(x_p) \\ \mathsf{succ}(.\mathsf{1}(x_p)) & \to & .\mathsf{0}(\mathsf{succ}(x_p)) \end{array}$$

Now we can express addition:

$$\begin{array}{rcllrcl} \mathsf{plus}(\mathsf{1},x) & \to & \mathsf{succ}(x) & \mathsf{plus}(.\mathsf{0}(x),.\mathsf{0}(y)) & \to & .\mathsf{0}(\mathsf{plus}(x,y)) \\ \mathsf{plus}(.\mathsf{0}(x),\mathsf{1}) & \to & \mathsf{succ}(.\mathsf{0}(x)) & \mathsf{plus}(.\mathsf{0}(x),.\mathsf{1}(y)) & \to & .\mathsf{1}(\mathsf{plus}(x,y)) \\ \mathsf{plus}(.\mathsf{1}(x),\mathsf{1}) & \to & .\mathsf{0}(\mathsf{succ}(x)) & \mathsf{plus}(.\mathsf{1}(x),.\mathsf{0}(y)) & \to & .\mathsf{1}(\mathsf{plus}(x,y)) \\ & & & \mathsf{plus}(.\mathsf{1}(x),.\mathsf{1}(y)) & \to & .\mathsf{0}(\mathsf{succ}(\mathsf{plus}(x,y))) \end{array}$$

Indeed now for every ground term composed from $\mathsf{1}, .\mathsf{0}, .\mathsf{1}, \mathsf{succ}, \mathsf{plus}$ containing at least one symbol succ or plus a rule is applicable. So the ground normal forms are the ground terms composed from $\mathsf{1}, .\mathsf{0}, .\mathsf{1}$, exactly being the binary representations of positive integers. Since the TRS is easily proved to be terminating, e.g., by recursive path order, every ground term will reduce to such a ground normal form, being the binary representations of a positive integer. As by every rule the numeric value of the term is preserved, this TRS serves for computing the binary value of any ground term considered so far.

Surprisingly, it is very simple to extend this system to multiplication, by using the fresh symbol mult for multiplication and introducing the following rules

$$\begin{array}{rcl} \mathsf{mult}(\mathsf{1},x) & \to & x \\ \mathsf{mult}(.\mathsf{0}(x),y) & \to & .\mathsf{0}(\mathsf{mult}(x,y)) \\ \mathsf{mult}(.\mathsf{1}(x),y) & \to & \mathsf{plus}(.\mathsf{0}(\mathsf{mult}(x,y)),y) \end{array}$$

The above observations are also easily checked for the extended TRS consisting of all rules presented so far: all rules preserve values, and for every ground term composed from $1, .0, .1, \mathsf{succ}, \mathsf{plus}, \mathsf{mult}$ containing at least one symbol $\mathsf{succ}$ or $\mathsf{plus}$ or $\mathsf{mult}$ a rule is applicable. So the ground normal forms are the ground terms composed from $1, .0, .1$, exactly being the binary representations of positive integers. Also the extended TRS is easily proved to be terminating, e.g., by recursive path order, so every ground term will reduce to such a ground normal form, being the binary representations of a positive integer. So the extended TRS also serves for executing multiplication.

Choosing $\Sigma'$ to consist of the single symbol $\mathsf{mult}$ it is easily checked that all conditions of Theorem 3 are satisfied. So by Theorem 3 we conclude that for using this TRS for computing the addition or multiplication of two binary numbers, every reduction to normal form has the same length.

In [5] it is proved that for addition this reduction length is linear in the size of the arguments, and for multiplication it is quadratic in the size of the arguments, so having the same complexity as the standard algorithms for binary addition and multiplication. The TRSs as presented in [5] have been the basis of the implementation of integer number computation in the mCRL2 tool set [2].

## 4 Conclusion

Basic terms are typical terms to be rewritten: a defined symbol on top and constructor terms as arguments. We gave a criterion for orthogonal TRSs by which all reductions of a basic term to normal form have the same length, showing that the reduction length is independent of the chosen strategy. This applies to both addition and multiplication, both in unary and binary notation.

In unary notation the Fibonacci function still satisfies our criteria. However, for more complicated user defined functions experiments based on a simple implementation show that reduction lengths of basic terms are typically not strategy independent any more.

**Acknowledgment**. We want to thank Evans Kaijage for fruitful discussions on this topic and for doing some experiments.

## References

[1] Martin Avanzini and Georg Moser. Closing the gap between runtime complexity and polytime computability. In Christopher Lynch, editor, *Proceedings of the 21st International Conference on Rewriting Techniques and Applications*, volume 6 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 33–48, Dagstuhl, Germany, 2010. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[2] J. F. Groote et al. The mCRL2 tool set. `http://mcrl2.org/mcrl2/wiki/index.php/Home`.

[3] Terese. *Term Rewriting Systems*. Cambridge University Press, 2003.

[4] Y. Toyama. Reduction strategies for left-linear term rewriting systems. In A. Middeldorp, V. van Oostrom, F. van Raamsdonk, and R. de Vrijer, editors, *Processes, Terms and Cycles: Steps on the Road to Infinity:Essays Dedicated to Jan Willem Klop on the Occasion of His 60th Birthday*, volume 3838 of *Lecture Notes in Computer Science*, pages 198–223. Springer, 2005.

[5] H. Zantema. Basic arithmetic by rewriting and its complexity. Available at `http://www.win.tue.nl/~hzantema/aritm.pdf`, 2003.