Technical Report 06-04 Computer Science Department Portland State University

The 2006 Federated Logic Conference

The Seattle Sheraton Hotel and Towers

Seattle, Washington

August 10 - 22, 2006

RTA'06 Workshop

WRS'06
The Sixth International Workshop on Reduction Strategies in Rewriting and Programming

August 11th, 2006

Proceedings

Editor:

Sergio Antoy

Technical Report 06-04 Computer Science Department Portland State University

Preface

This volume contains the pre-workshop proceedings of the **Sixth International Workshop on Reduction Strategies in Rewriting and Programming (WRS'06)** which was held on August 11th, 2006, in Seattle, Washington, USA, as part of the Federated Logic Conference, FLoC 2006. The workshop is the sixth edition in a series of events intended to provide a forum for presenting and discussing cutting-edge ideas and new results, recent developments, research directions and surveys on existing knowledge in the area of reduction strategies. The previous WRS editions were: WRS 2001 (Utrecht, The Netherlands), WRS 2002 (Copenhagen, Denmark), WRS 2003 (Valencia, Spain), WRS 2004 (Aachen, Germany) and WRS 2005 (Nara, Japan).

Reduction strategies study which subexpression(s) of an expression should be selected for evaluation and which rule(s) should be applied. These choices affect fundamental properties of a computation such as laziness, strictness, completeness and need to name a few. For this reason some programming languages, e.g., Elan, Maude, *OBJ* and Stratego, allow the explicit definition of the evaluation strategy, whereas other languages, e.g., Clean, Curry, and Haskell, allow its modification. Strategies pose challenging theoretical problems and play an important role in practical tools such as theorem provers, model checkers and programming languages. In implementations of languages, strategies bridge the gap between operational principles, e.g., graph and term rewriting, narrowing and lambda-calculus, and semantics, e.g., normalization, computation of values and head-normalization.

In addition to regular paper sessions, the workshop hosted two invited talks, one by Richard Kieburtz, OHSU/OGI School of Science & Engineering, and the other by Claude Kirchner, INRIA and LORIA.

I would like to thank the program committee and the additional reviewers for providing four timely and accurate reviews for each submitted paper, the invited speakers for accepting the program committee invitation, the authors for their contributions, the FLoC organizers, and Portland State University for logistic and financial support.

Portland, Oregon
July 13, 2006

Sergio Antoy
Program Committee Chair

Program Committee

Sergio Antoy	Portland State University
Santiago Escobar	Universidad Politécnica de Valencia
Jürgen Giesl	RWTH Aachen
Bernhard Gramlich	Technische Universität Wien
Ralf Lämmel	Microsoft Corporation
Salvador Lucas	Universidad Politécnica de Valencia
Narciso Marti-Oliet	Universidad Complutense de Madrid
Mizuhito Ogawa	Japan Advanced Institute of Science and Technology
Jaco van de Pol	Centrum voor Wiskunde en Informatica
Manfred Schmidt-Schauß	Johann Wolfgang Goethe-Universität

Additional Reviewers

Bernd Braßel
Rita Loogen
Grigore Rosu
David Sabel
Traian-Florin Serbanuta
Tim Sheard
Andrew Tolmach
Alicia Villanueva

Table of Contents

Programmed Strategies for Program Verification

(invited talk)

Richard B. Kieburtz

OGI School of Science & Engineering, OHSU
Portland, Oregon

Abstract

Plover is an automated property-verifier for Haskell programs that has been under development for the past three years as a component of the Programatica project. In Programatica, predicate definitions and property assertions written in P-logic, a programming logic for Haskell, can be embedded in the text of a Haskell program module. P-logic properties refine the type system of Haskell but cannot be verified by type-checking alone; a more powerful logical verifier is needed. Plover codes the proof rules of P-logic, and additionally, embeds strategies and decision procedures for their application and discharge. It integrates a reduction system that implements a rewriting semantics for Haskell terms with a congruence-closure algorithm that supports reasoning with equality. It can employ splitting strategies to explore alternative valuations of expressions of type Bool or other finite data types, but these strategies lead to exponential growth of terms and must be employed cautiously. Plover itself is written in Stratego, which has proven to be a powerful language in which to write a verifier. This talk will explain the design and implementation of some of the strategies that enable Plover to comprehend Haskell and to discharge some valid property assertions.

Rewriting (your) Calculus

(invited talk)

Claude Kirchner

INRIA and LORIA
Nancy, France

Abstract

Rewriting is clearly established as a general paradigm which agility eases to express and reason about computation and deduction. The rewriting calculus, generalizing the lambda calculus and the rewriting relation, provides us with a theoretical and uniform foundation for this expressivity. Introduced in the late nineties, the framework and its meta-properties are now better understood. We will show why the calculus is well-suited to represent computation as well as deduction, and therefore deduction modulo.

The rewriting calculus is therefore a good candidate to backup proof assistants where the user can adapt the computation mechanism to its needs. But furthermore, we want the next generation of proof assistants to provide the user with the possibility to adapt also the deduction system to its needs. We will see how this could be designed and how it can be used to get better higher-level logical representation of user- defined theories.

Reduction and conversion strategies for the Calculus of (co)Inductive Constructions: Part I

Claudio Sacerdoti Coen

Department of Computer Science, University of Bologna, Italy, `sacerdot@cs.unibo.it`

Abstract. We compare several reduction and conversion strategies for the Calculus of (co)Inductive Constructions by running benchmarks on the library of the Coq proof assistant. All the strategies have been implemented in an independent verifier for the proof objects of Coq that is part of the Matita proof assistant.

1 Introduction

According to the Poincaré principle simple facts that can be automatically verified by means of computation should not be proved explicitly with deduction steps. In theorem provers based on the Curry-Howard correspondence the Poincaré principle is implemented by the conversion typing rule: every proof term for P is also a proof term for Q whenever P is convertible with Q ($P \simeq Q$). The *conversion* relation must be a decidable equivalence relation, and it is usually defined as the symmetric reflexive closure of a transitive and contextual *reduction* relation. Usually the latter includes at least β-reduction and δ-reduction (the substitution of a definiens with its definiendum), and it may become larger when sigma types or primitive inductive types are introduced in the calculus.

Most of the time the amount of reduction performed in the type-checking of a term is quite limited, consisting only of a few unfolding of definitions (δ-reduction steps) and β-reduction steps to instantiate general properties (e.g. symmetry) to specific arguments (the relation that is symmetric).

Sometimes, when two level reasoning is exploited, the amount of reduction becomes significant. The two level reasoning approach [3], also called reflection, really sticks to the Poincaré principle: an algorithm to verify a property for an internalised form of a formula is implemented in the calculus and a proof that the algorithm is correct is also provided; the proof of the property for a given formula is just the application of the proof of correctness to the internalised formula (that is computed in the meta-language). Type-checking the proof requires a conversion check that involves running the algorithm on the formula. The latter computation can be arbitrarily complex. For instance, recently the two level approach has been exploited to check in the Coq system the four colour theorem, that requires several days to be type-checked.

Strategies may play a significant role for reduction, and different reduction strategies may lead to different convertibility checks on the reduced subterms.

Moreover, convertibility is never checked by reducing both terms to normal form and then testing α-conversion. On the contrary, strategies are used to guide a controlled reduction of one or both terms, in order to quickly detect non convertible terms (before reaching their normal forms) and in order to speed up conversion by finding a common reduct that is not yet in normal form.

This work tries to assess the effects, on type-checking time, of reduction and convertibility strategies for the Calculus of (co)Inductive Constructions (CIC). In this first part we fix a conversion algorithm, called the basic conversion algorithm, that is parameterized over a conversion strategy. In the next part we will consider a second conversion algorithm, still parameterized over a conversion strategy, that outperforms the one described here and that is also used in the Coq proof assistant. Studying both algorightms helps in understanding the importance of the conversion and reduction strategies independently of the conversion algorithm.

To perform the benchmarks we have implemented the strategies in the kernel of the Matita interactive theorem prover (http://matita.cs.unibo.it) that is compatible with the proof terms exported from the Coq proof assistant [8]. So we can run the benchmarks on the library of Coq, that comprises about 40,000 theorems and definitions, several of them proved with two level reasoning.

In Sect. 2 we describe the syntax, reduction and convertibility rules of CIC. In Sect. 3 we describe the basic conversion algorithm that is a generalisation of the simplest syntax directed algorithm for testing convertibility. In Sect. 4 we compare several reduction and convertibility strategies for the basic conversion algorithm and the reduction procedure it calls. Conclusions and future work are in Sect. 5, that is followed by two appendices that detail the benchmarks we performed.

2 The Calculus of (co)Inductive Construction

We present now the Calculus of (co)Inductive Constructions with de Bruijn indexes, and its reduction and extended conversion rules. We omit the typing rules, the pragmatic and also the meta-theory of the calculus since they are not relevant to this work. The presentation we give is adapted from the author's PhD. thesis [7] and is as close as possible to the calculus implemented in the proof assistant Coq, version 8.0. It is also a strict subset of the calculus implemented in the proof assistant Matita under development at the University of Bologna.

2.1 The Syntax

With a small notational abuse, all the sequences indexed from 1 to n will also comprise the empty sequence, unless stated otherwise. Moreover, for the sake of readability, we adopt the compressed syntax $\overrightarrow{\phi[\alpha]}$ for the list $\phi[1]\ldots\phi[n]$ where each list item $\phi[i]$ is obtained by syntactically replacing α with i. The dual notation $\overleftarrow{\phi[\alpha]}$ stands for the same list in reverse order. When two compressed expressions indexed by the same Greek variable occur in the same term, the two expanded lists are meant to have the same length. If two compressed expressions indexed by the same Greek variable are nested, each variable is meant to be bound in the innermost compressed list.

In the rest of the paper we reserve $c, c_1, \ldots, c_n$ for constant names, $i, i_1, \ldots, i_n$ for inductive type names, $k, k_1, \ldots, k_n, k_1^1, \ldots, k_{n_1}^1, \ldots, k_1^m, \ldots, k_{n_m}^m$ for inductive constructor names, $s, s_1, \ldots, s_n$ for sorts of the form $Set, Prop, Type(j)$ for some j, t, f, u, T, U, N, M for well formed terms and j, n, m, l, r for positive integers.

Table 1. CIC terms syntax

$t ::= n$	de Bruijn index
$\mid\ c$	constant
$\mid\ i$	(co)inductive type
$\mid\ k$	(co)inductive constructor
$\mid\ Set \mid Prop \mid Type(j)$	sort
$\mid\ t\ t$	application
$\mid\ \lambda : t.t$	λ-abstraction
$\mid\ \lambda := t.t$	local definition
$\mid\ \Pi : t.t$	dependent product
$\mid\ \langle t \rangle_h t\{\overrightarrow{t}\}$	case analysis
$\mid\ \mu_l\{\overrightarrow{t : t / n_\alpha}\}$	well-founded mutually recursive definition
$\mid\ \nu_l\{\overrightarrow{t : t}\}$	mutually corecursive definitions

Well formed terms are inductively defined in Table 1. We will use parentheses to disambiguate expressions and we assume the usual convention that application is left associative and λ-abstraction and local definition are right associative.

Note that, in CIC, there is no syntactic distinction between terms and types. For the sake of clarity, we will write unknown terms as T if meant to be types and as f if meant to be function bodies.

Only the last three constructors deserve an explanation, all the others coming from the theory of Pure Type Systems [1] and from the Extended Calculus of Constructions [5]. We just remind that local definition, λ-abstractions and dependent products bind a single variable in their second argument. The first argument is the type of the bound variable for λ-abstractions and dependent products; it is the definiens for local definitions.

The case analysis constructor $\langle T \rangle_h t\{\overrightarrow{t_\alpha}\}$ performs pattern matching on the term t, that must inhabit an inductive type family with h parameters. One branch t_α is associated to each constructor of the inductive type, with the intended meaning that, if t reduces to the application $(k_i\ t_1\ \ldots\ t_h\ t'_1\ \ldots\ t'_n)$, the whole expression will reduce to the branch t_i applied to $t'_1 \ldots t'_n$. The arguments $t_1\ \ldots\ t_h$ are dropped since they are used only for typing purposes to identify the family instance. Finally, the term T can be thought of as the dependent type of each branch and is also used for typing purposes only.

Example 1. Let List be a family of inductive types indexed by one parameter A that is the type of the elements of the list, and let Empty of type $\Pi A : Set.ListA$ and Cons of type $\Pi A : Set.\Pi h : A.\Pi t : ListA.ListA$ be its constructors. The function that returns true if its argument is an empty list can be defined as

$$\begin{array}{l}\lambda A : Set.\lambda l : \text{List } A.\\ \quad \textbf{match } l \textbf{ with } Empty \;\Rightarrow true \mid Cons\ he\ tl \;\Rightarrow false\\ \quad \textbf{return type } \lambda\_ : \text{List } A.bool\end{array}$$

in an ML-like syntax with bound variables, that in CIC syntax with de Bruijn indexes becomes

$$\lambda : Set.\lambda : \text{List } 1.\langle \lambda : \text{List } 2.\text{bool}\rangle_1 1\{\text{true} \;;\; \lambda : 2.\lambda : \text{List } 2.\text{false}\}$$

If we apply the term to bool and (Cons bool true (Empty bool)) and we perform two β-reduction steps we obtain

$$\langle \lambda : \text{List bool}.bool\rangle_1(\text{Cons bool true (Empty bool)})\{\text{true} \,;\, \lambda : \text{bool}.\lambda : \text{List bool.false}\}$$

that in one step (see Sect. 4) reduces to

$$(\lambda : \text{bool}.\lambda : \text{List bool.false}) \text{ true Empty bool}$$

Notice that the actual parameter bool of the constructor Cons is thrown away during reduction, since it instantiates a family parameter (as reminded by the subscript 1 that says that there is only one family parameter expected).

The $\mu_l\{\overrightarrow{t_\alpha : T_\alpha/n_\alpha}\}$ constructor simultaneously defines a block of mutual recursive functions and returns the l-th function in the block. The α-th function in the block has body t_α and type T_α and is defined by structural recursion over his n_α-th argument. Each defined function is bound in the bodies, but not in the types.

Example 2. To the following fragment of ML-like code

```
let rec reverse (acc : List bool) (l : List bool) =
 match l with
    Empty => acc
  | Cons he tl => reverse (Cons he acc) tl
in reverse
```

corresponds the following CIC term

$$\begin{array}{l}\mu_1\{\lambda : \text{List bool}.\lambda : \text{List bool}.\\ \qquad \langle \lambda : \text{List bool.List bool}\rangle_1 1\{2 \;;\; \lambda : \text{bool}.\lambda : \text{List bool}.5 \text{ (Cons bool 2 4) } 1)\\ \quad\; : \Pi : \text{List bool}.\Pi : \text{List bool.List bool}\\ \quad /2\}\end{array}$$

The $\nu_l\{\overrightarrow{t_\alpha : T_\alpha}\}$ defines a block of mutually co-recursive functions and picks the l-th defined function; it is syntactically very similar to $\mu_l\{\overrightarrow{t_\alpha : T_\alpha/n_\alpha}\}$, but for the third argument of each recursive function that is missing in the co-recursive case.

Lifting and Substitution Since we have adopted a syntax based on de Bruijn indexes, we do not have to worry about α-conversion, but: 1) we need to introduce a lifting operation (see Table 2) to move terms under one or several binders; 2) we need to define a substitution operation (see Table 3) to replace the first de Bruijn index with a term that avoids capturing on the substituted term and also decrements all the free de Bruijn indexes that are not substituted. The definition of the two functions is completely standard.

Table 2. Lifting

$$
\begin{array}{l}
\uparrow^m t = \uparrow^m_0 t \\
\\
\uparrow^m_b n = n \quad \text{if } n <= b \\
\uparrow^m_b n = n + m \quad \text{if } n > b \\
\uparrow^m_b t = t \quad \text{if } t \in \{c, i, k, Set, Prop, Type(j)\} \\
\uparrow^m_b (t_1\ t_2) = \uparrow^m_b t_1 \uparrow^m_b t_2 \\
\uparrow^m_b \lambda : T.t = \lambda : \uparrow^m_b T. \uparrow^m_{b+1} t \\
\uparrow^m_b \lambda := t_1.t_2 = \lambda := \uparrow^m_b t_1. \uparrow^m_{b+1} t_2 \\
\uparrow^m_b \Pi : T.t = \Pi : \uparrow^m_b T. \uparrow^m_{b+1} t \\
\uparrow^m_b \langle t \rangle_h t' \{\overrightarrow{t_\alpha}\} = \langle \uparrow^m_b t \rangle_h \uparrow^m_b t' \{\overrightarrow{\uparrow^m_b t_\alpha}\} \\
\uparrow^m_b \mu_l \{\overrightarrow{t_\alpha : T_\alpha / n_\alpha}\} = \mu_l \{\overrightarrow{\uparrow^m_{b+r} t_\alpha : \uparrow^m_b T_\alpha / n_\alpha}\} \\
\uparrow^m_b \nu_l \{\overrightarrow{t_\alpha : T_\alpha / n_\alpha}\} = \nu_l \{\overrightarrow{\uparrow^m_{b+r} t_\alpha : \uparrow^m_b T_\alpha / n_\alpha}\}
\end{array}
$$

Table 3. Substitution

$$
\begin{array}{l}
m\{N/m\} = \uparrow^{m-1} N \\
n\{N/m\} = n \quad \text{if } n < m \\
n\{N/m\} = n - 1 \quad \text{if } n > m \\
t\{N/m\} = t \quad \text{if } t \in \{c, i, k, Set, Prop, Type(j)\} \\
(t_1\ t_2)\{N/m\} = t_1\{N/m\}\ t_2\{N/m\} \\
(\lambda : T.t)\{N/m\} = \lambda : T\{N/m\}.t\{N/m+1\} \\
(\lambda := t_1.t_2)\{N/m\} = \lambda := t_1\{N/m\}.t_2\{N/m+1\} \\
(\Pi : T.t)\{N/m\} = \Pi : T\{N/m\}.t\{N/m+1\} \\
(\langle t \rangle_h t' \{\overrightarrow{t_\alpha}\})\{N/m\} = \langle t\{N/m\} \rangle_h t'\{N/m\} \{\overrightarrow{t_\alpha\{N/m\}}\} \\
(\mu_l \{\overrightarrow{t_\alpha : T_\alpha / n_\alpha}\})\{N/m\} = \mu_l \{\overrightarrow{t_\alpha\{N/m+r\} : T_\alpha\{N/m\} / n_\alpha}\} \\
(\nu_l \{\overrightarrow{t_\alpha : T_\alpha / n_\alpha}\})\{N/m\} = \nu_l \{\overrightarrow{t_\alpha\{N/m+r\} : T_\alpha\{N/m\} / n_\alpha}\}
\end{array}
$$

2.2 Reduction, Convertibility, Cumulativity

Reduction is defined only for CIC terms that are closed in a given *environment* and *context*.

An *environment* E associates constant definitions to constant names, declarations of mutually (co)inductive types to inductive names and declarations of constructors of mutually (co)inductive types to constructor names. For the sake of reduction only, an environment can be seen as an abstract data type with only one lookup operation $E(c)$ that returns the definiens of c in E.

A *context* Γ is an ordered list of anonymous declarations $(: T)$ or definitions $(:= t)$. If the de Bruijn index i occurs free in a term, it is supposed to be an occurrence of the i-th "constant" declared or defined in Γ. We write $\Gamma(i) = t$ to say that the i-th entry in Γ is a definition whose definiens is t.

Table 4. Reduction

$E[\Gamma] \vdash (\lambda : T.M)N \rhd_\beta M\{N/1\}$	β-reduction
$E[\Gamma] \vdash \lambda :=\ t.M \rhd_\zeta M\{t/1\}$	ζ-reduction
$E[\Gamma] \vdash c\ \rhd_\delta t \quad \text{if } E(c) = t$	δ-reduction
$E[\Gamma] \vdash n \rhd_\delta \uparrow^n t \quad \text{if } \Gamma(i) = t$	δ-reduction
$E[\Gamma] \vdash \langle T\rangle_h(k_j\ t_1\ \ldots\ t_h\ t'_1\ \ldots\ t'_{n_j})\{\overrightarrow{f_\alpha}\} \rhd_\iota (f_j\ t'_1 \ldots t'_{n_j})$	ι-reduction
$E[\Gamma] \vdash \mu_j\{\overrightarrow{f_\alpha : T_\alpha/n_\alpha}\}\ t_1\ \ldots\ t_{n_j-1}\ (k\ t'_1\ \ldots\ t'_m)$	
$\rhd_\mu f_j\{\overrightarrow{\mu_\alpha\{\overrightarrow{f_\beta : T_\beta/n_\beta}\}/1}\}\ t_1\ \ldots\ t_{n_j-1}\ (k\ t'_1\ \ldots\ t'_m)$	μ-unfolding
$E[\Gamma] \vdash \langle T\rangle_h(\nu_j\{\overrightarrow{f_\alpha : T_\alpha}\})\{\overrightarrow{t_\beta}\} \rhd_\nu \langle T\rangle_h(f_j\{\overrightarrow{\nu_\alpha\{\overrightarrow{f_\gamma : T_\gamma}\}/1}\})\{\overrightarrow{t_\beta}\}$	ν-unfolding

$\rhd$ is the reflexive, transitive and contextual closure of $\rhd_\beta \cup \rhd_\zeta \cup \rhd_\delta \cup \rhd_\iota \cup \rhd_\mu \cup \rhd_\nu$

In Table 4 we have collected all the one step reduction rules of CIC. The formulation of β-, ζ-and δ-reduction is the standard one when de Bruijn indexes are used. Definiens coming from the environment E are not lifted during δ-reduction since the typing rules grant that every definiens in E is a closed term. On the contrary, a term in Γ can depend on the terms occurring after it in Γ, seen as an ordered list. Thus the need for the lifting.

The ι-reduction rule has already been discussed in Sect. 2.1. The μ- and ν-unfolding rules, usually also called ι-reduction rules, are restricted forms of the usual unfolding rule given by (co)fixpoints. In particular, a recursive function definition can be unfolded only when applied to a constructor (possibly applied to some arguments) and, dually, a co-recursive function definition can be unfolded only when it is the argument of a destructor (here called case analysis). Together with additional typing restrictions, this is sufficient to grant strong normalisation [9, 2] for the well-typed terms of the calculus (supposing E and Γ also well-typed). Notice that the constraint on the unfolding of co-recursive functions forces a call-by-name strategy for the co-recursive fragment: unfolding is allowed only when the function is in head position.

Table 5. Convertibility and cumulativity

The convertibility equivalence relation $\simeq$ is the symmetric closure of $\triangleright$.
$E[\Gamma] \vdash T_1 \preceq T_2$ (i.e. T_1 is a “subtype” of T_2 up to universe cumulativity) iff

– $T_1 =_{\beta\delta\iota\zeta} T_2$ or
– $T_1 = \mathbf{Type}(i)$ and $T_2 = \mathbf{Type}(j)$ and $i \leq j$ or
– $T_1 = \mathbf{Prop}$ and $T_2 = \mathbf{Type}(i)$ or
– $T_1 = \mathbf{Set}$ and $T_2 = \mathbf{Type}(i)$ or
– $T_1 = \Pi x : S_1.T_1'$ and $T_2 = \Pi x : S_2.T_2'$ and $S_1 \simeq S_2$ and $T_1' \preceq T_2'$

Convertibility and Cumulativity Convertibility is defined in Table 5. Since the reduction relation is strongly normalising, convertibility is trivially decidable by reducing both terms to their normal form and syntactically comparing them. Conversion and reduction strategies to decide convertibility avoiding unnecessary computation are the topic of this paper and will be explored in further sections.

In the tradition of the Extended Calculus of Constructions [5], the convertibility relation is weakened to an order relation called *cumulativity* that takes into account the inclusion of lower universes into higher ones and that is also defined in Table 5.

Cumulativity plays the role of a subtype relation in the typing rules of CIC. Since any algorithm that decides convertibility can be easily adapted to decide cumulativity, we will speak of conversion strategies including also strategies to decide cumulativity. Moreover, in the rest of the paper we will consider only algorithms that decide convertibility.

3 The Basic Conversion Algorithm

In this section we present the idea behind a simple algorithm to test convertibility that is easily adapted to cumulativity. We call it the *basic conversion algorithm.* The algorithm will be presented as an almost syntax directed judgement that, seen as a rewriting system, presents critical pairs that must be solved using strategies. Moreover, the judgement is parameterised over a class of reduction algorithms that leave great freedom in the choice of the reduction strategy. In the following section we will discuss a few strategies.

The basic conversion algorithm can be regarded as folklore for calculi simpler than CIC. In [6] it is presented for the pure Calculus of Constructions. It consists in intertwining weak head reduction steps with α-conversion steps, according to the observation that two terms in weak head normal form can be equivalent only if their fixed heads are so.

Instead of presenting the algorithm in its usual form, in Fig. 1 we provide a new judgement $E[\Gamma] \vdash t_1 \downarrow t_2$ that can be proved to be equivalent to $E[\Gamma] \vdash t_1 \simeq t_2$, but that is more direct to implement since it is almost syntax directed. The last two rules are parameterised over another judgement $E[\Gamma] \vdash t \triangleright_h t'$ that must satisfy the following property.

$$\frac{}{E[\mathit{\Gamma}] \vdash t \downarrow t}\ t \in \{n, c, i, k, s\} \qquad \frac{E[\mathit{\Gamma}] \vdash T \downarrow T' \qquad E[\mathit{\Gamma}; (: T)] \vdash t \downarrow t'}{E[\mathit{\Gamma}] \vdash \lambda : T.t \downarrow \lambda : T'.t'}$$

$$\frac{E[\mathit{\Gamma}] \vdash t_1 \downarrow t_1' \qquad E[\mathit{\Gamma}] \vdash t_2 \downarrow t_2'}{E[\mathit{\Gamma}] \vdash (t_1\ t_2) \downarrow (t_1'\ t_2')} \qquad \frac{E[\mathit{\Gamma}] \vdash t \downarrow t' \qquad E[\mathit{\Gamma}] \vdash T \downarrow T' \qquad \overrightarrow{E[\mathit{\Gamma}] \vdash t_\alpha \downarrow t_\alpha'}}{E[\mathit{\Gamma}] \vdash \langle T \rangle_h t\{\overrightarrow{t_\alpha}\} \downarrow \langle T' \rangle_h t'\{\overrightarrow{t_\alpha'}\}}$$

$$\frac{\overrightarrow{E[\mathit{\Gamma}] \vdash T_\alpha \downarrow T_\alpha'} \qquad \overrightarrow{E[\mathit{\Gamma}\ \overrightarrow{(: T_\alpha)}] \vdash f_\alpha \downarrow f_\alpha'}}{E[\mathit{\Gamma}] \vdash \mu_j\{\overrightarrow{f_\alpha : T_\alpha/n_\alpha}\} \downarrow \mu_j\{\overrightarrow{f_\alpha' : T_\alpha'/n_\alpha}\}} \qquad \frac{E[\mathit{\Gamma}] \vdash t_1 \downarrow t_1' \qquad E[\mathit{\Gamma}; (:= t_1)] \vdash t_2 \downarrow t_2'}{E[\mathit{\Gamma}] \vdash \lambda := t_1.t_2 \downarrow \lambda := t_1'.t_2'}$$

$$\frac{\overrightarrow{E[\mathit{\Gamma}] \vdash T_\alpha \downarrow T_\alpha'} \qquad \overrightarrow{E[\mathit{\Gamma}\ \overrightarrow{(: T_\alpha)}] \vdash f_\alpha \downarrow f_\alpha'}}{E[\mathit{\Gamma}] \vdash \nu_j\{\overrightarrow{f_\alpha : T_\alpha}\} \downarrow \nu_j\{\overrightarrow{f_\alpha' : T_\alpha'}\}} \qquad \frac{E[\mathit{\Gamma}] \vdash T \downarrow T' \qquad E[\mathit{\Gamma}; (: T)] \vdash t \downarrow t'}{E[\mathit{\Gamma}] \vdash \mathit{\Pi} : T.t \downarrow \mathit{\Pi} : T'.t'}$$

$$\textsc{Conv-Whd-l}\ \frac{E[\mathit{\Gamma}] \vdash t_1 \rhd_h t_1' \qquad E[\mathit{\Gamma}] \vdash t_1' \downarrow t_2}{E[\mathit{\Gamma}] \vdash t_1 \downarrow t_2} \qquad \textsc{Conv-Whd-r}\ \frac{E[\mathit{\Gamma}] \vdash t_2 \rhd_h t_2' \qquad E[\mathit{\Gamma}] \vdash t_1 \downarrow t_2'}{E[\mathit{\Gamma}] \vdash t_1 \downarrow t_2}$$

Fig. 1. Almost syntax directed convertibility judgement

Definition 1 (Weak head progress). *A judgement $E[\mathit{\Gamma}] \vdash t \rhd_h t'$ satisfies weak head progress (or, abusing the standard terminology, is a* weak head reduction*) if whenever the judgement holds we have that either t has no redex in head position and t and t' are the same term, or t has a redex in head position, $E[\mathit{\Gamma}] \vdash t \rhd t'$ and the redex in head position in t has been reduced in the latter reduction.*

The redex in head position in a CIC term is defined as follows:

Definition 2 (Redex in head position (r.h.p.)). *The terms $i, k, Set, Prop, Type(j), \lambda : T.t, \mathit{\Pi} : T.t, \mu_l\{\overrightarrow{t_\alpha : T\_alpha}\}, \nu_l\{\overrightarrow{t_\alpha : T\_alpha}\}$ have no r.h.p. The terms n, c are a r.h.p. if they can be δ-reduced. The term $(t_1\ t_2)$ is a r.h.p. if it is a β-redex (i.e. t_1 is a λ-abstraction) or if it is a μ-redex (i.e. t_1 is an application $(\mu_l\{\overrightarrow{t\_alpha : T\_alpha/n_\alpha}\}\ t_1'\ \ldots\ t_{n_l-1}')$ and t_2 is a constructor or the application of a constructor to some arguments). If $(t_1\ t_2)$ is not a r.h.p. then its r.h.p. (if there is one) is: the r.h.p. of t_1 (if there is one) if t_1 is not an application $(\mu_l\{\overrightarrow{t\_alpha : T\_alpha/n_\alpha}\}\ t_1'\ \ldots\ t_{n_l-1}')$; the r.h.p. of t_2 (if there is one) otherwise. The term $\langle T \rangle_h t\{\overrightarrow{t_\alpha}\}$ is a r.h.p. if it is a ι-redex (i.e. if t is a constructor) or if it is a ν-redex (i.e. t is a co-recursive function definition). If the term $\langle T \rangle_h t\{\overrightarrow{t_\alpha}\}$ is not a r.h.p., then its r.h.p. (if there is one) is the r.h.p. of t (if there is one).*

The reader that considers quite baroque the previous definition (and consequently also the reduction rules of the calculus) can try to develop by himself the calculus that is to CIC what the $\bar{\lambda}$-calculus [4] is to the λ-calculus. In that representation the terms of the calculus are actually states of a reduction machine and the subterm in head position is clearly separated from its context, leading to a cleaner and more elegant definition of the reduction rules and the weak head normal forms (w.h.n.f.). This unpublished representation of the term is also actually used in the kernel of Coq to implement lazy reduction and, of top of it, the convertibility and cumulativity checks. However, this representation is way less understandable to the user, requiring a transparent translation back and forth to the syntax adopted in this paper. We will better describe it in the second part of the present work, where it can be better appreciated.

A first important observation on weak head reduction judgements is that they do not need to be implemented in the usual way, that consists of performing the usual call-by-name computation of the w.h.n.f. of the input. Indeed, only one head reduction step is required and moreover reduction in non-head position is allowed. The latter observation tells us that we can employ any reduction strategy we want, such as call-by-name and call-by-need, and any reduction technology such as reduction machines or term rewriting systems.

The second important observation is that the new convertibility judgement is not completely syntax directed, since most of the rules form a critical pair with the Conv-Whd-l or the Conv-Whd-r rule any time one of the two terms is not in w.h.n.f. However, there exists one and only one complete strategy that does not employ backtracking. It is the strategy that always applies one of the Conv-Whd-* rules, unless the two terms are already in w.h.n.f. Since reduction on well typed terms is strongly normalising, the two rules cannot be applied for ever. We call this the *simplest convertibility strategy.* Any other strategy can be completed by means of backtracking: every time one of the two terms is not yet in weak head normal form and the strategy prefers a different rule over the appropriate Conv-Whd-* rule, in case of failure the rule Conv-Whd-* rule must be immediately applied before continuing as before.

We analyse now the impact of several reduction and conversion strategies on the efficiency of the basic convertibility algorithm.

4 Reduction and Conversion Strategies

The simplest convertibility strategy allows to quickly detect non convertible terms without performing full reduction. Instead, if the two terms are convertible, no computation is avoided. Interactive theorem provers that record proof terms that are certified by a trusted kernel use convertibility and type-checking in two different places: inside the kernel (to check the correctness of the terms produced outside) and outside the kernel, for instance in the implementation of tactics. Since the kernel is supposed to check well typed terms, for the first usage we expect the two terms to be always convertible. Thus, at least in this case, the simplest convertibility strategy is not very satisfactory (see benchmarks in App. B). Thus in the next sections we will explore different convertibility strategies to optimise this particular case.

4.1 An Improved Convertibility Strategy

We assume now that the two terms whose convertibility must be checked are almost always convertible. One special case of convertibility is α-convertibility, that reduces to a check for identity when de Bruijn indexes are employed. Identity is recognised by our judgement when we use the strategy that never applies the rules Conv-Whd-l and Conv-Whd-r. We call this (incomplete) strategy the α-convertibility strategy (even if we employ de Bruijn indexes).

Actually, our statistics show that, when type-checking real world terms, most of the terms checked for convertibility are actually identical. This suggest a simple but very effective strategy: the reduction rules Conv-Whd-* are used only to make the α-convertibility strategy complete, as explained in Sect. 3. More concretely, the two terms are recursively compared using every rule but Conv-Whd-*; if the comparison fails, they are both reduced to w.h.n.f. and compared again. A second failure grants that the two terms are not convertible.

The benchmarks in App. B show a remarkable improvement over the simplest convertibility strategy. We observe that the improvement does not derive only from the reduced reduction time: since weak head normal forms are usually larger than the input terms, checking convertibility of their normal forms requires a significantly larger amount of time that is saved in the new strategy.

4.2 Further Improvement of the Convertibility Strategy

Let us consider now the improved convertibility strategy of the previous section. The strategy is optimal for identical terms. Let t_1 and t_2 be the two smallest non identical subterms in corresponding position (i.e. in two identical contexts). To check their convertibility, both terms are reduced to their w.h.n.f., possibly performing other reduction steps as well according to the reduction strategy.

In practice, our statistics say that quite often one term can be reduced to the other one without computing the w.h.n.f. This is for instance the case when, during a proof, the user unfolds a definition (performs a δ-reduction step). In this case the two terms to be compared will be the δ-redex and its δ-reduct. As before, we would like to avoid unnecessary computation but also the additional time spent in checking convertibility of large w.h.n.f.

The way to improve the situation is: 1) to be able to detect in advance which one of the two terms is more likely to reduce to the second one; 2) reduce it only until the second one or a w.h.n.f. is reached.

Empirical observations suggest that, in CIC, long chains of β-reduction steps are rare and that either the bound variable occurs linearly or the substituted terms are small (i.e. they are formulae, but not long proof terms). Moreover real computations exploited by the user correspond to long chains of ι- and μ- or ν-reduction steps and are unlikely to be avoidable during convertibility. The conclusion is that δ-reduction steps are the important ones to address, since they often produce large reducts that can even start long reduction chains. Here we will address only δ-reduction steps of constants for implementation reasons.

To control δ-reduction steps we propose a strategy that behaves as the one in the previous paragraph until the first comparison fails. In this case it performs weak reduction of both t_1 and t_2 avoiding δ-steps for constants, until a normal form is reached. There are now three possibilities: 1) both terms are in w.h.n.f. and the algorithm proceeds with the second pass; 2) one term has a head δ-redex and the other one is a w.h.n.f.: the first term is reduced to w.h.n.f. before proceeding with the second pass; 3) both terms have head δ-redexes: we use a heuristic to decide what term (or terms) must be head reduced and when reduction should stop.

The heuristic we propose is based on the following metric.

Definition 3 (Height of a constant). *Let E be an environment. The height $h(c)$ of a constant c such that $E(c) = t$ is defined by $h(c) = 1 + \sum_{c' \in t} h(c')$ where $c' \in t$ if c' occurs in t. If no constant occurs in c the height of c is 1.*

We claim that in practice it is often the case that given a δ-redex $(c\ t_1 \ldots t_n)$ whose δ-reduct reduces to $(c'\ t'_1 \ldots t'_m)$ we have $h(c) > h(c')$.

For instance, consider the two convertible terms $(*\ 1000\ 100)$ and $(+\ 100\ (*\ 999\ 100))$ that have both a δ-redex in head position. The former term reduces to the latter and since product is defined in terms of addition, we have $h(*) > h(+)$. Thus it is a good idea to perform head reduction on the first term unless a δ-redex in head position of height less than $h(+)$ is found. If the reduced term has height $h(+)$, we can hope that its head is an addition and that the two terms are now convertible. This is indeed the case in our not so artificial example.

As a counterexample to the property above, if c is defined as the identity function then we have $(c\ c') \rhd_h c'$ and $1 = h(c) \not> h(c')$. Notice that this is quite a rare case: it can occur only if the argument of a constant can occur in head position during reduction.

According to the previous metric, the heuristic of our new strategy is defined as follows: the height h_1 and h_2 of the two head δ-redices are compared; if one (say h_1) is greater than the other, its term is head reduced until it becomes a w.h.n.f. or a δ-redex of height less or equal to h_2; otherwise both terms are head reduced until they become w.h.n.f. or a δ-redex of height less than h_1. Then the algorithm proceeds with the second pass.

The benchmarks in App. B show that the proposed improvement is really effective in decreasing type-checking time, independently from the reduction strategy it is associated with. The improvement could also be applied to the code of the Coq proof assistant, that right now delays δ-conversion as we suggest, but does not exploit any heuristic similar to ours both to choose which term must be reduced when two δ-redexes are met and to guide the reduction avoiding intermediate usually useless convertibility checks.

4.3 Reduction Strategies

Having considered three different convertibility strategies, we want now to compare their behaviour when combined with different reduction strategies. To make the comparison, in our PhD. thesis [7] we have described GKAM$_{CIC}$, a generic reduction machine, based on the abstract machine of Krivine, that is parameterised over the reduction strategy.

The status of the Krivine's abstract machine (KAM) is made of an environment, a code and a stack. The code is the term to be reduced. Its free variables are assigned values by the environment, that plays the role of an explicit simultaneous substitution. When an application is processed, its argument is moved to the stack together with a pointer to the environment, forming a closure. When a λ-abstraction (part of a β-redex) is processed, the top of the stack is simply moved to the top of the environment. Finally, when a de Brujin index is processed, the n-th component of the stack is fetched and becomes the new term to be processed (together with the new environment). As an example, consider the following reduction of the identity function applied to a closed term t:

$$\langle \emptyset, (\lambda.1\ t), \emptyset\rangle \rhd \langle \emptyset, \lambda.1, \langle \emptyset, t\rangle.\emptyset\rangle \rhd \langle \emptyset.\langle \emptyset, t\rangle, 1, \emptyset\rangle \rhd \langle \emptyset, t, \emptyset\rangle$$

Since the argument t is never reduced before reaching the weak head position, the machine implements a call-by-name strategy. The $GKAM_{CIC}$ (Generalized KAM for CIC) generalizes the KAM in three ways. 1) Arbitrary reductions of the argument of an application are now allowed when the argument is moved to the stack, to the environment or in code position. This way any kind of reduction strategy can be implemented. 2) The data structures of the elements of the stack and of the environment become parameters. This helps in implementing strategies such as call-by-need. As a consequence, we also introduce as parameters read-back functions from stack and environment items to terms. The read-back functions are used when the machine becomes stuck to map the machine status to the corresponding computed term. 3) The machine is extended to CIC.

We describe now the generic reduction machine, that is parameterised over a few functions and datatypes that are instantiated by each reduction strategy. The machine is actually implemented as an ML functor whose input is a module that describes the reduction strategy.

Definition 4 (Closures, environments and stacks of the GKAM$_{CIC}$). *Let EItem and SItem be datatypes to be instantiated by the strategy.*
$Environment \overset{def}{=} EItem\ List;\ Stack \overset{def}{=} SItem\ List;$
$Closure \overset{def}{=} Environment * Term;\ Configuration \overset{def}{=} Environment * Term * Stack$
The following functions must be instantiated by the strategy:
$to\_stack : Closure \rightarrow SItem;\ to\_env : SItem \rightarrow EItem;$
$from\_env : SItem \rightarrow Configuration;\ \ \mathcal{T}_s : SItem \rightarrow Term;\ \ \mathcal{T}_e : EItem \rightarrow Term$
For each strategy there must be a strict and well-founded order $<_E\ \subseteq\ Environment^2$ *such that* $\forall \xi : Environment .\forall \alpha : EItem .\xi <_E \xi.\alpha$

The transition rules for the GKAM$_{CIC}$ are shown in Table 6. The initial configuration of the machine to reduce a term t is $(\emptyset, t, \emptyset)$. The final configurations of the machine are the special configurations $(-, (\xi, t, S), -)$. We write $\mathcal{R}(\xi, t, S) = (\xi', t', S')$ if the machine reduces in many steps the configuration (ξ, t, S) to the configuration $(-, (\xi', t', S'), -)$.

Table 6. GKAM$_{CIC}$ reduction rules

Before			After		
Env.	Code	Stack	Env.	Code	Stack
ξ	i (where $i \leq \lvert\xi\rvert$)	S	π_1(from_env ξ_i)	π_2(from_env ξ_i)	π_3(from_env ξ_i)$.S$
ξ	i (where $i > \lvert\xi\rvert$ and $\Gamma(i - \lvert\xi\rvert) = b$)	S	$\emptyset$	$\uparrow^{i-\lvert\xi\rvert} b$	S
ξ	i (otherwise)	S	-	(ξ, i, S)	-
ξ	c (where $E(c) = b$)	S	$\emptyset$	b	S
ξ	c (otherwise)	S	-	(ξ, c, S)	-
ξ	i	S	-	(ξ, i, S)	-
ξ	k	S	-	(ξ, k, S)	-
ξ	s	$\emptyset$	-	$(\emptyset, \emptyset, s, \emptyset)$	-
ξ	$\Pi : T.t$	$\emptyset$	-	$(\xi, \Pi : T.t, \emptyset)$	-
ξ	$\lambda : T.M$	$\alpha.S$	ξ.(to_env α)	M	S
ξ	$\lambda : T.M$	$\emptyset$	-	$(\xi, \lambda : T.M, \emptyset)$	-
ξ	MN	S	ξ	M	(to_stack (ξ,N))$.S$
ξ	$\lambda := M.N$	S	ξ.to_env (to_stack(ξ, M))	N	S
ξ	$\langle T\rangle_h t\{\overrightarrow{t_\alpha}\}$ (when †1 holds)	S	ξ	t_i	$I'_1 \dots I'_l.S$
ξ	$t_0 = \langle T\rangle_h t\{\overrightarrow{t_\alpha}\}$ (otherwise)	S	-	(ξ, t_0, S)	-
ξ	$\mu_l\{\overrightarrow{f_\alpha : T_\alpha/n_\alpha}\}$ (when †2 holds)	S	$\overleftarrow{\xi\text{.to\_env(to\_stack}(\xi, \mu_\alpha\{\overrightarrow{f_\alpha : T_\alpha/n\_alpha}\}))}$	f_i	S''
ξ	$t_0 = \mu_l\{\overrightarrow{f_\alpha : T_\alpha/n_\alpha}\}$ (otherwise,)	S	-	(ξ, t_0, S'')	-
ξ	$t_0 = \nu_l\ \{\overrightarrow{f_\alpha : T_\alpha}\}$	S	-	(ξ, t_0, S)	-

†1: Let $\mathcal{R}(\xi, t, S) = (\xi_1, t_1, S_1)$. If $t_1 = \nu_l\{\overrightarrow{f_\alpha : T_\alpha}\}$ then let $\mathcal{R}(\overleftarrow{\xi_1\text{.to\_env(to\_stack}(\xi_1, \nu_\alpha\{\overrightarrow{f\_alpha : T\_alpha}\}))}, f_l, S_1) = (\xi_2, t_2, S_2)$. Otherwise let $(\xi_2, t_2, S_2) = (\xi_1, t_1, S_1)$. The rule is fired only if $(t_2, S_2) = (k_i, I_1 \dots I_h.I'_1 \dots I'_l)$.

†2: Let $\mathcal{R}(\text{from\_env(to\_env}(S_{n_l}))) = (\xi', t', S')$ and let S'' be equal to S but for the n_l-th entry that is replaced with to_stack$(\emptyset, \mathcal{T}(\xi', t', S'))$. The rule is fired if t' is a constructor. Note that S'' is used anyway also in the rule that is applied when this rule fails. The same rules modified posing $S'' = S$ are also admissible as call-by-name variants; in this variant the read back functions are never used by the reduction loop.

We define the following read-back functions from machine configurations and closures to terms:

Definition 5. *Read-back functions*

$$\begin{array}{l}\mathcal{T}(-,(\xi,t,S),-) \overset{def}{=} \mathcal{T}(\xi,t,S)\\ \mathcal{T}(\xi,t) \overset{def}{=} \mathcal{T}(\xi,t,\emptyset)\\ \mathcal{T}([\alpha_1,\ldots,\alpha_n],t,[\beta_1,\ldots\,\beta_m]) \overset{def}{=}\\ \quad (t\{\,\mathcal{T}_e(\alpha_1)/1\ ;\ \ldots\ ;\ \mathcal{T}_e(\alpha_n)/n\}\ \mathcal{T}_s(\beta_1)\ \ldots\ \mathcal{T}_s(\beta_m)))\end{array}$$

where the simultaneous substitution $\{\sigma\}_m$ *maps every de Bruijn index in its domain to its image.*

We can now implement head reduction as reduction of the initial machine configurations followed by read-back: $E[\Gamma] \vdash t \rhd_h t''$ iff $\mathcal{R}(\emptyset,t,\emptyset) = (\xi,t',S)$ and $\mathcal{T}(\xi,t',S) = t''$.

The following correctness and liveness conditions on reduction strategies grant the corresponding properties for the reduction machine (proof given in [7]).

Definition 6. *GKAM$_{CIC}$ Correctness Conditions:*

1. $\forall(\xi,t)$: *Closure* $.\mathcal{T}(\xi,t) \rhd \mathcal{T}_s(\mathit{to\_stack}\ (\xi,t))$
2. $\forall\alpha$: *SItem* $.\mathcal{T}_s(\alpha) \rhd \mathcal{T}_e(\mathit{to\_env}\ \alpha)$
3. $\forall\alpha$: *EItem* $.\mathcal{T}_e(\alpha) \rhd \mathcal{T}(\mathit{from\_env}\ \alpha)$

Definition 7. *GKAM$_{CIC}$ Liveliness Condition:* $\pi_1\circ\mathit{from\_env}\circ\mathit{hd}\circ\pi_1$ *is strictly decreasing according to the ordering* $<_E$*;* π_n *stands for the n-th projection of a tuple.*

A few benchmarks comparing different reduction strategies are given in App. B. App. A shows how the GKAM$_{CIC}$ can be instantiated to obtain a few typical reduction strategies used in the benchmarks. Notice that we allow ourselves to call recursively $\mathcal{R}(\_,\_,\_)$ and $\mathcal{T}(\_,\_,\_)$ even if the calls are not tail recursive. Non tail recursive calls are usually not allowed in reduction machines for performance reasons. What we gain is the ability to switch the reduction strategy easily for direct comparison.

The benchmarks stress our feeling that, at least in the average case, avoiding reduction and hence comparison of usually larger reducts is better than optimising reduction. We can observe this in two different ways: 1) convertibility strategies seem more effective than reduction strategies; 2) call-by-name (also in the call-by-need variant) gives better results than call-by-value, and a strategy that is somehow intermediate between the two of them from the point of view of convertibility or reducts gives intermediate results. However, we can also observe the existence of a very heavy theorem that can be type-checked in reasonable time only in a call-by-value setting. Similar theorems, all based on two level reasonings, are also found outside the standard library, in the user contributions. Notice, however, that several other theorems proved with the same approach in the standard library are type-checked in a reasonable time with our best combination of strategies.

5 Conclusions and Future Work

We have presented the basic conversion algorithm, a simple almost syntax directed judgement to test convertibility (and with minimal modifications also cumulativity) of terms of the Calculus of (co)Inductive Constructions (CIC). Both convertibility and reduction strategies can be imposed to obtain executable algorithms from the judgement. We have presented a few improvements on the simplest convertibility strategy and we have assessed their effect with benchmarks on a real world library. One of the improvements could also be applied to the code of the Coq proof assistant, with expected performance increasing. We have also presented a generic reduction machine parameterised over the reduction strategies. The machine has been used to perform the benchmarks, varying over the reduction strategy.

The aim of the paper was mainly investigative, since, as far as we know, precise comparisons of several reduction strategies for CIC and their role in type-checking are not available in the literature.

Even with the best convertibility and reduction strategy the benchmarks show that, for a few theorems, the basic conversion algorithm is not competitive with the one now adopted in the Coq proof assistant. Profiling the code it becomes evident that the bottleneck is the read-back function $\mathcal{T}$ that is invoked after each reduction, to convert configurations back to terms. Thus the evident improvement consists in avoiding the read-back function, considering a new almost syntax directed and strategy driven judgement to check convertibility over machine configurations. This solution, that is the one adopted for a particular strategy in the Coq proof assistant, has also been implemented in Matita giving results compatible with the ones of Coq. The description of this alternative algorithm and the effects of strategies for it will be the subject of the second part of this work.

References

1. H. P. Barendregt. “Lambda calculi with types”, Handbook of logic in computer science (vol. II), pages 117–309, 1992.
2. E. Gimenez. “Codifying guarded definitions with recursive schemes”. Proceedings of the 1994 Workshop on Types for Proofs and Programs, LNCS 996, pages 39-59.
3. J. Harrison. “Metatheory and reflection in theorem proving: A survey and critique”. Technical Report CRC-053, SRI Cambridge, 1995.
4. H. Herbelin. *Séquents qu'on calcule: de l'interprétation du calcul des séquents comme calcul de lambda-termes et comme calcul de stratégies gagnantes*, PhD thesis, Université Paris VII, 1995.
5. Z. Luo. *An Extended Calculus of Constructions.* PhD thesis, University of Edinburgh, 1990.
6. R. Harper and R. Pollack. “Type checking with universes”. Theoretical Computer Science, 89:107-136, 1991.
7. C. Sacerdoti Coen. *Mathematical Knowledge Management and Interactive Theorem Proving.* PhD. thesis, University of Bologna, 2004.

8. C. Sacerdoti Coen. “From Proof-Assistants to Distributed Libraries of Mathematics: Tips and Pitfalls”. In Proc. Mathematical Knowledge Management 2003, Lecture Notes in Computer Science, Vol. 2594, pp. 30–44, Springer-Verlag.
9. B. Werner. *Une Theorie des Constructions Inductives.* PhD. thesis, Université Paris VII, 1994.

A Implementation of Strategies for the GKAM$_{CIC}$

Here we show (in pseudo ML syntax) how to instantiate the GKAM$_{CIC}$ to obtain both standard and non-conventional reduction strategies. The third strategy performs in parallel both call-by-value and call-by-name. When environment items are fetched during computation, the call-by-value component is returned; when they are fetched in the read-back procedure, the call-by-name components is returned to simulate “undoing” reduction of redexes not in head position. The proof that the three more standard strategies really implement what is expected can be found in the author’s PhD. thesis [7] for an extended version of the calculus.

call-by-name:
```
    SItem = term
    EItem = term
    to_stack$(\xi, t) = \mathcal{T}(\xi, t)$
    to_env $t = t$
    from_env $t = (\emptyset, t, \emptyset)$
    $\mathcal{T}_s(t) = t$
    $\mathcal{T}_e(t) = t$
```

call-by-value:
```
    SItem = closure
    EItem = term
    to_stack $(\xi, t) = (\xi, t)$
    to_env$(\xi, t) = \mathcal{R}(\xi, t, \emptyset)$
    from_env $t = (\emptyset, t, \emptyset)$
    $\mathcal{T}_s(\xi, t) = \mathcal{T}(\xi, t)$
    $\mathcal{T}_e(t) = t$
```

call-by-value, read-back by name:
```
    SItem = closure
    EItem = term * term
    to_stack $(\xi, t) = (\xi, t)$
    to_env$(\xi, t) = (\mathcal{R}(\xi, t, \emptyset), \mathcal{T}(\xi, t))$
    from_env $(t_v, t_n) = (\emptyset, t_v, \emptyset)$
    $\mathcal{T}_s(\xi, t) = \mathcal{T}(\xi, t)$
    $\mathcal{T}_e(t_v, t_n) = t_n$
```

call-by-need:
```
    SItem = closure
    EItem = (bool $*$ configuration) ref
    to_stack $(\xi, t) = (\xi, t)$
    to_env$(\xi, t) = ref(false, (\xi, t, \emptyset))$
    from_env $c$ =
        match $!t$ with
          $true, c' \to c'$
        $|false, c' \to c := \mathcal{R}(c');\ snd\ !c$
    $\mathcal{T}_s(\xi, t) = \mathcal{T}(\xi, t)$
    $\mathcal{T}_e(c) = \mathcal{R}(snd\ !c)$
```

B Benchmarks

The benchmarks in Table 7 show the effect of convertibility and reduction strategies on type-checking time. The benchmarks have been performed running the kernel of the Matita interactive theorem prover on subsets of the library of the Coq proof assistant. All the tests have been run on a Pentium IV 2.5 GHz with 1GB of RAM. Matita is written in OCaml.

Table 7. Benchmarks

Conversion strategy	Reduction Strategy	Standard library	Skipped theorems	Heavy theorems	Heaviest theorem
SS	call-by-name	1285.71s	375	170	29.6s
IS	call-by-name	246.76s	1	15	6.9s
IS	call-by-value, read-back by name	279.58s	1	23	13.0s
IS	call-by-value	422.51s	1	36	9.7s
IS+	call-by-name	199.26s	1	2	2.2s
IS+	call-by-need	201.71s	1	3	1.5s
IS+	call-by-value, read-back by name	220.54s	1	9	10.1s
IS+	call-by-value	391.36s	0	19	11.8s
Coq		40.87s	0	2	2.5s

SS = Simplest strategy
IS = Improved Strategy (Sect. 4.1)
IS+ = Furtherly Improved Strategy (Sect. 4.2)

The "standard library" of Coq is the library developed by the authors of Coq and distributed with the system. It is made of 5904 theorems and definitions.

Skipped theorems are theorems that require more than 30s to be type-checked. The overall and mean type-checking times shown in the table do not take in account the first 30s spent on skipped theorems.

Heavy theorems are non skipped theorems whose type-checking type requires more than 1s. Since we consider 1s to be an acceptable type-checking time for a single theorem, a satisfactory choice of strategies should produce no skipped theorems and no heavy theorems.

The last line of the table shows the time required by Coq when run on the same machine. Remember that Coq does not employ the basic conversion algorithm, but the one that will be described in the second part of this work. Moreover, constants can be marked in Coq automatically or by the user as "opaque", preventing their δ-expansion and seriously speeding up conversion in some frequent situations. Our implementation does not exploit opaque constants since opacity is an information that is not available in the library exported from Coq.

Since the type-checker implementation of Coq is not the same of Matita, we cannot say if the better performances are all due to the conversion algorithm and to the opacity trick or if they are partly due to a most performant implementation of type-checking. The benchmarks planned for the second part of this work will be based on the type checker of Matita, allowing a direct comparison without biases of the different conversion and reduction strategies and algorithms.

The Power of Closed Reduction Strategies*

S. Alves[1], M. Fernández[2], M. Florido[1], and I. Mackie[2,3]**

[1] University of Porto, Department of Computer Science & LIACC, R. do Campo Alegre 823, 4150-180, Porto, Portugal
[2] King's College London, Department of Computer Science, Strand, London WC2R 2LS, U.K.
[3] LIX, École Polytechnique, 91128 Palaiseau Cedex, France

Abstract. Closed reduction strategies in the λ-calculus restrict the reduction rules: the idea is that reductions can only take place when certain terms are closed (i.e. do not contain free variables). This has lead to various applications, such as an α-conversion free calculus of explicit substitutions, and an efficient abstract machine. The main contribution of this paper is a new application of this strategy to a linear version of Gödel's System T. We show that a linear System T with closed reduction offers a huge increase in expressive power over the usual linear systems, which are 'closed by construction' rather than 'closed at reduction'.

1 Introduction

Hidden in a correctness proof of Girard's Geometry of Interaction [7] is a strategy for cut-elimination in linear logic [6]. This strategy restricts cut-elimination steps so that they can only take place when the exponential boxes are closed. Not only is this strategy for cut-elimination simpler than the general one, it is also exceptionally efficient in terms of the number of cut-elimination steps.

There are several translations of the λ-calculus into linear logic, which inspired the work on closed reduction in the λ-calculus [4, 5]. Closed reductions avoid α-conversion by restricting β-reduction, so that only closed substitutions are generated. In contrast with standard weak strategies, which also avoid α-conversion, closed reduction strategies allow β-reductions to take place under lambdas, which means that more sharing can be achieved. In addition to offering efficient reduction strategies, applications such as an α-conversion free calculus of explicit substitution were obtained (see [5] for more details).

These are just some of the pieces of evidence that show that closed reductions have interesting properties. The purpose of this paper is to examine this family of strategies further. We will show that closed reduction strategies can also be

*Research partially supported by the Treaty of Windsor Grant: "Linearity: Programming Languages and Implementations".

**Projet Logical, Pôle Commun de Recherche en Informatique du plateau de Saclay, CNRS, École Polytechnique, INRIA, Université Paris-Sud.

applied with great benefit in other areas, more precisely, in this paper we will analyse the computational power of linear λ-calculi using closed reductions.

In [2] we defined a linear version of Gödel's System T with closed reductions, which we called System $\mathcal{L}$. We showed that this linear system has exactly the same computational power as System T. This result is surprising because usual definitions of linear systems are strictly less powerful than System T [11, 9].

In this paper, we claim that the use of closed reduction is the key to the power of System $\mathcal{L}$. To support this claim, we analyse the interplay between linearity and closed reduction, and compare the computational power of linear systems with and without closed reduction.

We will define two linear systems: $\mathcal{L}^{\mathbb{N}}$ and $\mathcal{L}_0^{\mathbb{N}}$. Both systems are extensions of the linear λ-calculus [1] with numbers, booleans, pairs of natural numbers, and an iterator. System $\mathcal{L}^{\mathbb{N}}$ has the same syntax as System $\mathcal{L}$ [2], and as System $\mathcal{L}$ it uses the closed reduction strategy. However, its type system is more restrictive than System $\mathcal{L}$. We will show that System $\mathcal{L}^{\mathbb{N}}$ can encode all the primitive recursive functions and more general functions such as Ackermann. On the other hand, System $\mathcal{L}_0^{\mathbb{N}}$ does not restrict to closed reduction strategies, but to be linear, it has to restrict the set of terms. Actually, System $\mathcal{L}_0^{\mathbb{N}}$ can be seen as a subsystem of Dal Lago's linear language $H(\emptyset)$ [11], albeit with a different syntax. We will show that in System $\mathcal{L}_0^{\mathbb{N}}$ we can encode only primitive recursive functions (Ackermann's function is not definable), therefore this system is strictly weaker than System $\mathcal{L}^{\mathbb{N}}$, and therefore weaker than System $\mathcal{L}$.

In the next section we recall some background material. In Section 3 we define the linear systems $\mathcal{L}^{\mathbb{N}}$ and $\mathcal{L}_0^{\mathbb{N}}$, and in Section 4 we demonstrate that we can encode all the primitive recursive functions in these calculi. In Section 5 we show that System $\mathcal{L}^{\mathbb{N}}$ goes considerably beyond this class of functions. Finally we conclude the paper in Section 6.

2 Background: Closed Reduction, Linear Systems

Çağman and Hindley [3] observed that α-conversion can be avoided if β-redexes are closed (i.e. $(\lambda x.t)u$ does not contain free variables). However, this is a strong restriction, and the resulting calculus is very weak. In [4, 5] closed reduction strategies were investigated which are less restrictive than this. The motivation for this study was to understand efficiency issues in addition to finding calculi that were free from α-conversion.

Two different versions of closed reduction were studied, based around the following two options:

$$\begin{array}{ll} (\lambda x.t)u \to t[u/x] & \text{if } \mathsf{fv}(\lambda x.t) = \varnothing \\ (\lambda x.t)u \to t[u/x] & \text{if } \mathsf{fv}(u) = \varnothing \end{array}$$

which correspond to closed function (**cf**) and closed argument (**ca**) respectively. In both cases, there are a number of variants that lead to calculi with different properties. Substitution is taken to be explicit in these systems.

The closed argument strategy was used in [2] to define an extension of the linear λ-calculus [1] with natural numbers, booleans, linear pairs, linear conditionals and a linear iterator (and with implicit rather than explicit substitution). This linear version of Gödel's System T was called System $\mathcal{L}$.

In [12] it was shown that the linear λ-calculus is the internal language for symmetric monoidal closed categories (the analogous result to the λ-calculus being the internal language to Cartesian Closed Categories). The addition of natural numbers and an iterator corresponds to adding a natural number object in the category. Note that, in this linear setting, the iterator is only allowed to iterate closed linear functions. More precisely, the typing rule for iterators requires the function to be typed in an empty environment, that is, iterators are "closed by *construction*":

$$\frac{\Gamma \vdash n : \mathbb{N} \quad \Delta \vdash b : A \quad \vdash f : A \multimap A}{\Gamma, \Delta \vdash \mathsf{iter}\ n\ b\ f}$$

In the same line, the linear System T of [11, 9], called $H(\emptyset)$, only allows the construction of iterators on closed functions, and can only encode primitive recursive functions. On the other hand, System $\mathcal{L}$ has the power of the full System T. To understand what gives these two linear versions of System T so different properties, in the following sections we will define two linear systems, one which allows us to build iterators on functions with free variables, but requires that reduction takes place only after the functions become closed, and another that does not use closed reduction, but requires iterators to be closed by construction.

3 Linear Systems with and without Closed Reduction

In this section we will define two linear systems: System $\mathcal{L}^{\mathbb{N}}$ and System $\mathcal{L}_0^{\mathbb{N}}$. Both systems extend the linear λ-calculus with booleans, numbers, pairs of natural numbers, and an iterator. While System $\mathcal{L}_0^{\mathbb{N}}$ has the usual open β-reduction rule but, when building an iterator, requires the iterated function to be closed (therefore avoiding copying of free variables), System $\mathcal{L}^{\mathbb{N}}$ uses a closed reduction strategy [5, 8] and allows the use of open functions in iterators.

We start by defining the syntax and the set of types, which will be common to the two systems.

3.1 Linear Terms and Types

The linear λ-terms are terms from the λ-calculus restricted in the following way ($\mathsf{fv}(t)$ denotes the set of free variables of t).

$$\begin{array}{ll} x & \\ \lambda x.t & \text{if } x \in \mathsf{fv}(t) \\ tu & \text{if } \mathsf{fv}(t) \cap \mathsf{fv}(u) = \varnothing \end{array}$$

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).

Next we add to this linear λ-calculus:

Pairs:

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

Note that when projecting from a pair, we use both projections. A simple example of such a term is the function that swaps the components of a pair:

$$\lambda x.\mathtt{let}\ \langle y, z\rangle = x\ \mathtt{in}\ \langle z, y\rangle.$$

Booleans: true and false, and a conditional:

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

Note that this linear conditional uses the same resources in each branch.

Numbers: 0 and S, and an iterator:

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

Table 1 summarises the syntax of System $\mathcal{L}^{\mathbb{N}}$ and System $\mathcal{L}_0^{\mathbb{N}}$, showing in parallel the term construction, variable constraints and free variables of terms.

Construction	**Variable Constraint**	**Free Variables** (fv)
$0, \mathsf{true}, \mathsf{false}$	$-$	$\varnothing$
$\mathsf{S}\ t$	$-$	$\mathsf{fv}(t)$
$\mathsf{iter}\ t\ u\ v$	$\mathsf{fv}(t) \cap \mathsf{fv}(u) = \mathsf{fv}(u) \cap \mathsf{fv}(v) = \mathsf{fv}(t) \cap \mathsf{fv}(v) = \varnothing$	$\mathsf{fv}(t) \cup \mathsf{fv}(u) \cup \mathsf{fv}(v)$
x	$-$	$\{x\}$
tu	$\mathsf{fv}(t) \cap \mathsf{fv}(u) = \varnothing$	$\mathsf{fv}(t) \cup \mathsf{fv}(u)$
$\lambda x.t$	$x \in \mathsf{fv}(t)$	$\mathsf{fv}(t) \smallsetminus \{x\}$
$\langle t, u\rangle$	$\mathsf{fv}(t) \cap \mathsf{fv}(u) = \varnothing$	$\mathsf{fv}(t) \cup \mathsf{fv}(u)$
$\mathtt{let}\ \langle x, y\rangle = t\ \mathtt{in}\ u$	$\mathsf{fv}(t) \cap \mathsf{fv}(u) = \varnothing,\ x, y \in \mathsf{fv}(u)$	$\mathsf{fv}(t) \cup (\mathsf{fv}(u) \smallsetminus \{x, y\})$
$\mathtt{cond}\ t\ u\ v$	$\mathsf{fv}(u) = \mathsf{fv}(v), \mathsf{fv}(t) \cap \mathsf{fv}(u) = \varnothing$	$\mathsf{fv}(t) \cup \mathsf{fv}(u)$

Table 1.

Types The set of *linear types* is generated by the grammar:

$$A, B ::= \mathbb{N} \mid \mathbb{B} \mid A \multimap B \mid \mathbb{N} \otimes \mathbb{N}$$

That is, we consider two base types (natural numbers and booleans), linear arrows, and linear products on natural numbers.

3.2 System $\mathcal{L}^{\mathbb{N}}$

We now define the reduction rules and the typing rules for System $\mathcal{L}^{\mathbb{N}}$.

Definition 1 (Closed Reduction). *Table 2 gives the reduction rules for System $\mathcal{L}^{\mathbb{N}}$, substitution is a meta-operation defined as usual. Reductions can take place in any context. We use $\longrightarrow$ to denote the one-step reduction relation, and $\longrightarrow^*$ for its reflexive and transitive closure.*

Name	Reduction		Condition
Beta	$(\lambda x.t)v$	$\longrightarrow t[v/x]$	$\mathsf{fv}(v) = \varnothing$
Let	$\mathtt{let}\ \langle x, y\rangle = \langle t, u\rangle\ \mathtt{in}\ v$	$\longrightarrow (v[t/x])[u/y]$	$\mathsf{fv}(t) = \mathsf{fv}(u) = \varnothing$
Cond	$\mathsf{cond\ true}\ u\ v$	$\longrightarrow u$	
Cond	$\mathsf{cond\ false}\ u\ v$	$\longrightarrow v$	
Iter	$\mathsf{iter}\ (\mathsf{S}\ t)\ u\ v$	$\longrightarrow v(\mathsf{iter}\ t\ u\ v)$	$\mathsf{fv}(tv) = \varnothing$
Iter	$\mathsf{iter}\ 0\ u\ v$	$\longrightarrow u$	$\mathsf{fv}(v) = \varnothing$

Table 2. Closed reduction

Reduction is weak: for example, $\lambda x.(\lambda y.y)x$ is a normal form. Note that all the substitutions created during reduction (rules *Beta, Let*) are closed; this corresponds to a closed argument reduction strategy (ca, see [5]). Also note that *Iter* rules cannot be applied if the function v is open.

System $\mathcal{L}^{\mathbb{N}}$'s syntax and reduction rules are the same as System $\mathcal{L}$'s [2], as a consequence we inherit from System $\mathcal{L}$ the following properties, for the untyped calculus:

Lemma 1 (Correctness of Substitution). *Let t and u be valid terms, $x \in \mathsf{fv}(t)$, and $\mathsf{fv}(u) = \emptyset$, then $t[u/x]$ is valid.*

Lemma 2 (Correctness of $\longrightarrow$). *Let t be a valid term, and $t \longrightarrow u$, then:*

1. $\mathsf{fv}(t) = \mathsf{fv}(u)$;
2. *u is a valid term.*

Lemma 3 (Confluence). *If $t \longrightarrow^* t_1$ and $t \longrightarrow^* t_2$, then there is a term t_3 such that $t_1 \longrightarrow^* t_3$ and $t_2 \longrightarrow^* t_3$.*

We associate types to terms in System $\mathcal{L}^{\mathbb{N}}$ using the typing rules given in Figure 1.

Since we are in a linear system, we do not have Weakening and Contraction rules. The only structural rule in Figure 1 is Exchange. For the same reason, the logical rules split the context between the premises. The rules for numbers are standard. In the case of a term of the form $\mathsf{iter}\ t\ u\ v$, we check that t is a term of type $\mathbb{N}$ and that v and u are compatible.

Also note that we allow the typing of $\mathsf{iter}\ t\ u\ v$ even if v is open (in contrast with [11, 9]), but we do not allow reduction until v is closed. We will show later

Axiom and **Structural Rule**:

$$\frac{}{x : A \vdash_{\mathcal{L}^{\mathbb{N}}} x : A}\,(\mathsf{Axiom}) \qquad \frac{\Gamma, x : A, y : B, \Delta \vdash_{\mathcal{L}^{\mathbb{N}}} t : C}{\Gamma, y : B, x : A, \Delta \vdash_{\mathcal{L}^{\mathbb{N}}} t : C}\,(\mathsf{Exchange})$$

Logical Rules:

$$\frac{\Gamma, x : A \vdash_{\mathcal{L}^{\mathbb{N}}} t : B}{\Gamma \vdash_{\mathcal{L}^{\mathbb{N}}} \lambda x.t : A \multimap B}\,(\multimap\mathsf{Intro}) \qquad \frac{\Gamma \vdash_{\mathcal{L}^{\mathbb{N}}} t : A \multimap B \qquad \Delta \vdash_{\mathcal{L}^{\mathbb{N}}} u : A}{\Gamma, \Delta \vdash_{\mathcal{L}^{\mathbb{N}}} tu : B}\,(\multimap\mathsf{Elim})$$

$$\frac{\Gamma \vdash_{\mathcal{L}^{\mathbb{N}}} t : \mathbb{N} \qquad \Delta \vdash_{\mathcal{L}^{\mathbb{N}}} u : \mathbb{N}}{\Gamma, \Delta \vdash_{\mathcal{L}^{\mathbb{N}}} \langle t, u\rangle : \mathbb{N} \otimes \mathbb{N}}\,(\otimes\mathsf{Intro})$$

$$\frac{\Gamma \vdash_{\mathcal{L}^{\mathbb{N}}} t : \mathbb{N} \otimes \mathbb{N} \qquad x : \mathbb{N}, y : \mathbb{N}, \Delta \vdash_{\mathcal{L}^{\mathbb{N}}} u : C}{\Gamma, \Delta \vdash_{\mathcal{L}^{\mathbb{N}}} \mathtt{let}\ \langle x, y\rangle = t\ \mathtt{in}\ u : C}\,(\otimes\mathsf{Elim})$$

Numbers

$$\frac{}{\vdash_{\mathcal{L}^{\mathbb{N}}} 0 : \mathbb{N}}\,(\mathsf{Zero}) \qquad \frac{\Gamma \vdash_{\mathcal{L}^{\mathbb{N}}} n : \mathbb{N}}{\Gamma \vdash_{\mathcal{L}^{\mathbb{N}}} \mathsf{S}\ n : \mathbb{N}}\,(\mathsf{Succ})$$

$$\frac{\Gamma \vdash_{\mathcal{L}^{\mathbb{N}}} t : \mathbb{N} \quad \Theta \vdash_{\mathcal{L}^{\mathbb{N}}} u : A \quad \Delta \vdash_{\mathcal{L}^{\mathbb{N}}} v : A \to A}{\Gamma, \Theta, \Delta \vdash_{\mathcal{L}^{\mathbb{N}}} \mathsf{iter}\ t\ u\ v : A}\,(\mathsf{Iter})$$

Booleans

$$\frac{}{\vdash_{\mathcal{L}^{\mathbb{N}}} \mathsf{true} : \mathbb{B}}\,(\mathsf{True}) \qquad \frac{}{\vdash_{\mathcal{L}^{\mathbb{N}}} \mathsf{false} : \mathbb{B}}\,(\mathsf{False})$$

$$\frac{\Delta \vdash_{\mathcal{L}^{\mathbb{N}}} t : \mathbb{B} \quad \Gamma \vdash_{\mathcal{L}^{\mathbb{N}}} u : A \quad \Gamma \vdash_{\mathcal{L}^{\mathbb{N}}} v : A}{\Gamma, \Delta \vdash_{\mathcal{L}^{\mathbb{N}}} \mathtt{cond}\ t\ u\ v : A}\,(\mathsf{Cond})$$

Fig. 1. Type System for System $\mathcal{L}^{\mathbb{N}}$

that this feature gives our system more power (whereas systems that do not allow building iter with v open are strictly weaker [11]).

Subject Reduction for System $\mathcal{L}^{\mathbb{N}}$ can be proved as for System $\mathcal{L}$ [2].

Theorem 1 (Subject Reduction). *If $\Gamma \vdash_{\mathcal{L}^{\mathbb{N}}} M : A$ and $M \longrightarrow N$, then $\Gamma \vdash_{\mathcal{L}^{\mathbb{N}}} N : A$.*

Note that confluence of the untyped calculus, together with subject reduction, implies confluence of the typed calculus.

Since terms typable in System $\mathcal{L}^{\mathbb{N}}$ are also typable in System $\mathcal{L}$, we inherit the strong normalisation property:

Theorem 2 (Strong Normalisation). *If $\Gamma \vdash_{\mathcal{L}^{\mathbb{N}}} t : T$, t is strongly normalisable.*

3.3 System $\mathcal{L}_0^{\mathbb{N}}$

The set of terms for System $\mathcal{L}_0^{\mathbb{N}}$ is built in the same way as for System $\mathcal{L}^{\mathbb{N}}$, except that when building an iterator, we don't allow the iterated function to be an open term. Thus iterators in this system have the following definition (note the additional constraint $\mathsf{fv}(v) = \varnothing$):

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

We now define the reduction rules and the typing rules for System $\mathcal{L}_0^{\mathbb{N}}$.

Definition 2 (Reduction). *Table 3 gives the reduction rules for System $\mathcal{L}_0^{\mathbb{N}}$, substitution is a meta-operation defined as usual. Reductions can take place in any context.*

Name	Reduction	
Beta	$(\lambda x.t)v$	$\longrightarrow t[v/x]$
Let	$\mathtt{let}\ \langle x,y\rangle = \langle t,u\rangle\ \mathtt{in}\ v$	$\longrightarrow (v[t/x])[u/y]$
Cond	$\mathsf{cond\ true}\ u\ v$	$\longrightarrow u$
Cond	$\mathsf{cond\ false}\ u\ v$	$\longrightarrow v$
Iter	$\mathsf{iter}\ (\mathsf{S}\ t)\ u\ v$	$\longrightarrow v(\mathsf{iter}\ t\ u\ v)$
Iter	$\mathsf{iter}\ 0\ u\ v$	$\longrightarrow u$

Table 3. Reduction for System $\mathcal{L}_0^{\mathbb{N}}$

Correctness of Substitution is proved as for System $\mathcal{L}$ [2], but α-conversion must be used in substitution whenever necessary. Note that α-conversion was not needed in System $\mathcal{L}$ and therefore in System $\mathcal{L}^{\mathbb{N}}$, because all the substitutions take a closed term.

Lemma 4 (Correctness of Substitution). *Let t and u be valid terms, such that $\mathsf{fv}(t) \cap \mathsf{fv}(u) = \emptyset$ and $x \in \mathsf{fv}(t)$, then $t[u/x]$ is valid.*

Lemma 5 (Correctness of $\longrightarrow$). *Let t be a valid term, and $t \longrightarrow u$, then:*

1. $\mathsf{fv}(t) = \mathsf{fv}(u)$*;*
2. *u is a valid term.*

Proof. (Sketch) The only reduction rules that copy or erase terms, are the rules for iterators, which either copy or erase the iterated function. However, because of the condition that the iterated function must be closed when constructing the term $\mathsf{iter}\ t\ u\ v$, then reducing an iterator will either copy or erase a closed term. Therefore the set of free variables is preserved and the term obtained is valid.

Note that in System $\mathcal{L}^{\mathbb{N}}$ we do not have the constraint on the iterator term, but we have a condition in the reduction rules for iterators, which also guarantees that reducing an iterator will either copy or erase a closed term.

In Figure 2 we show how types are assigned to terms in System $\mathcal{L}_0^{\mathbb{N}}$. The only difference between the rules for this system and for System $\mathcal{L}^{\mathbb{N}}$ is in the (Iter) rule, where the context for the iterated function is always empty.

As before, Subject Reduction for System $\mathcal{L}_0^{\mathbb{N}}$ can be proved as for System $\mathcal{L}$ [2].

Axiom and **Structural Rule**:

$$\frac{}{x : A \vdash_{\mathcal{L}_0^{\mathbb{N}}} x : A}\ (\mathsf{Axiom}) \qquad \frac{\Gamma, x : A, y : B, \Delta \vdash_{\mathcal{L}_0^{\mathbb{N}}} t : C}{\Gamma, y : B, x : A, \Delta \vdash_{\mathcal{L}_0^{\mathbb{N}}} t : C}\ (\mathsf{Exchange})$$

Logical Rules:

$$\frac{\Gamma, x : A \vdash_{\mathcal{L}_0^{\mathbb{N}}} t : B}{\Gamma \vdash_{\mathcal{L}_0^{\mathbb{N}}} \lambda x.t : A \multimap B}\ (\multimap\mathsf{Intro}) \qquad \frac{\Gamma \vdash_{\mathcal{L}_0^{\mathbb{N}}} t : A \multimap B \qquad \Delta \vdash_{\mathcal{L}_0^{\mathbb{N}}} u : A}{\Gamma, \Delta \vdash_{\mathcal{L}_0^{\mathbb{N}}} tu : B}\ (\multimap\mathsf{Elim})$$

$$\frac{\Gamma \vdash_{\mathcal{L}_0^{\mathbb{N}}} t : \mathbb{N} \qquad \Delta \vdash_{\mathcal{L}_0^{\mathbb{N}}} u : \mathbb{N}}{\Gamma, \Delta \vdash_{\mathcal{L}_0^{\mathbb{N}}} \langle t, u \rangle : \mathbb{N} \otimes \mathbb{N}}\ (\otimes\mathsf{Intro})$$

$$\frac{\Gamma \vdash_{\mathcal{L}_0^{\mathbb{N}}} t : \mathbb{N} \otimes \mathbb{N} \qquad x : \mathbb{N}, y : \mathbb{N}, \Delta \vdash_{\mathcal{L}_0^{\mathbb{N}}} u : C}{\Gamma, \Delta \vdash_{\mathcal{L}_0^{\mathbb{N}}} \mathtt{let}\ \langle x, y \rangle = t\ \mathtt{in}\ u : C}\ (\otimes\mathsf{Elim})$$

Numbers

$$\frac{}{\vdash_{\mathcal{L}_0^{\mathbb{N}}} 0 : \mathbb{N}}\ (\mathsf{Zero}) \qquad \frac{\Gamma \vdash_{\mathcal{L}_0^{\mathbb{N}}} n : \mathbb{N}}{\Gamma \vdash_{\mathcal{L}_0^{\mathbb{N}}} \mathsf{S}\ n : \mathbb{N}}\ (\mathsf{Succ})$$

$$\frac{\Gamma \vdash_{\mathcal{L}_0^{\mathbb{N}}} t : \mathbb{N} \quad \Theta \vdash_{\mathcal{L}_0^{\mathbb{N}}} u : A \quad \vdash_{\mathcal{L}_0^{\mathbb{N}}} v : A \to A}{\Gamma, \Theta \vdash_{\mathcal{L}_0^{\mathbb{N}}} \mathsf{iter}\ t\ u\ v : A}\ (\mathsf{Iter})$$

Booleans

$$\frac{}{\vdash_{\mathcal{L}_0^{\mathbb{N}}} \mathsf{true} : \mathbb{B}}\ (\mathsf{True}) \qquad \frac{}{\vdash_{\mathcal{L}_0^{\mathbb{N}}} \mathsf{false} : \mathbb{B}}\ (\mathsf{False})$$

$$\frac{\Delta \vdash_{\mathcal{L}_0^{\mathbb{N}}} t : \mathbb{B} \quad \Gamma \vdash_{\mathcal{L}_0^{\mathbb{N}}} u : A \quad \Gamma \vdash_{\mathcal{L}_0^{\mathbb{N}}} v : A}{\Gamma, \Delta \vdash_{\mathcal{L}_0^{\mathbb{N}}} \mathtt{cond}\ t\ u\ v : A}\ (\mathsf{Cond})$$

Fig. 2. Type System for System $\mathcal{L}_0^{\mathbb{N}}$

Theorem 3 (Subject Reduction). *If $\Gamma \vdash_{\mathcal{L}_0^{\mathbb{N}}} M : A$ and $M \longrightarrow N$, then $\Gamma \vdash_{\mathcal{L}_0^{\mathbb{N}}} N : A$.*

Note that any term typable in System $\mathcal{L}_0^{\mathbb{N}}$ is also typable in System $\mathcal{L}^{\mathbb{N}}$, therefore in System $\mathcal{L}$. Thus, System $\mathcal{L}_0^{\mathbb{N}}$ is strongly normalising.

Theorem 4 (Strong Normalisation). *If $\Gamma \vdash_{\mathcal{L}_0^{\mathbb{N}}} t : T$, t is strongly normalisable.*

Confluence for typable terms in System $\mathcal{L}_0^{\mathbb{N}}$ is a direct consequence of strong normalisation and the fact that the rules are non-overlapping (using Newmann's Lemma [13]). Moreover, we can apply directly Klop's result [10] to the untyped calculus because the system is orthogonal (that is, left-linear and non-overlapping):

Lemma 6 (Confluence). *If $t \longrightarrow^* t_1$ and $t \longrightarrow^* t_2$, then there is a term t_3 such that $t_1 \longrightarrow^* t_3$ and $t_2 \longrightarrow^* t_3$.*

4 Primitive Recursive Functions

Based on the results obtained for System $\mathcal{L}$ [2], in this section we show how we can define the primitive recursive functions in both System $\mathcal{L}^{\mathbb{N}}$ and System $\mathcal{L}_0^{\mathbb{N}}$. We choose to present an encoding that satisfies the term conditions of System $\mathcal{L}_0^{\mathbb{N}}$, since these are more restrictive than those of System $\mathcal{L}^{\mathbb{N}}$. In the next section we show that in System $\mathcal{L}^{\mathbb{N}}$ we can encode substantially more than primitive recursive functions.

We recall that a function $f : \mathbb{N}^n \to \mathbb{N}$ is primitive recursive if it can be defined using: the natural numbers; the projections: $\pi_i(x_1, \ldots, x_n) = x_i$ $(1 \leq i \leq n)$; composition; and the primitive recursive scheme, which allows us to define a recursive function h using two auxiliary (primitive recursive) functions f, g:

$$\begin{aligned} h(x, 0) &= f(x) \\ h(x, n+1) &= g(x, h(x, n), n) \end{aligned}$$

4.1 Erasing linearly

Although System $\mathcal{L}^{\mathbb{N}}$ and System $\mathcal{L}_0^{\mathbb{N}}$ are linear calculi, we can erase numbers. In particular, we can define the projection functions on $\mathbb{N}^2$ $\mathsf{fst}, \mathsf{snd} : \mathbb{N} \otimes \mathbb{N} \multimap \mathbb{N}$ as follows:

$$\begin{aligned} \mathsf{fst} &= \lambda x.\mathtt{let}\ \langle u, v\rangle = x\ \mathtt{in}\ \mathsf{iter}\ v\ u\ (\lambda z.z) \\ \mathsf{snd} &= \lambda x.\mathtt{let}\ \langle u, v\rangle = x\ \mathtt{in}\ \mathsf{iter}\ u\ v\ (\lambda z.z) \end{aligned}$$

Lemma 7. *For any numbers $\bar{a}$ and $\bar{b}$, $\mathsf{fst}\langle \bar{a}, \bar{b}\rangle \longrightarrow^* \bar{a}$ and $\mathsf{snd}\langle \bar{a}, \bar{b}\rangle \longrightarrow^* \bar{b}$.*

Proof. We show the case for fst. Let $\bar{a} = \mathsf{S}^n\ 0$, $\bar{b} = \mathsf{S}^m\ 0$.

$$\begin{aligned} \mathsf{fst}\langle \bar{a}, \bar{b}\rangle &\longrightarrow (\mathtt{let}\ \langle u, v\rangle = \langle \mathsf{S}^n\ 0, \mathsf{S}^m\ 0\rangle\ \mathtt{in}\ \mathsf{iter}\ v\ u\ \lambda z.z) \\ &\longrightarrow \mathsf{iter}\ (\mathsf{S}^m\ 0)\ (\mathsf{S}^n\ 0)\ \lambda z.z \longrightarrow^* \mathsf{S}^n\ 0 = \bar{a}. \end{aligned}$$

Note that fst and snd are valid typable terms in System $\mathcal{L}_0^{\mathbb{N}}$ (see Figure 3), and they are closed terms.

$$\dfrac{\dfrac{x : \mathbb{N} \otimes \mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} x : \mathbb{N} \otimes \mathbb{N} \qquad \dfrac{v : \mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} v : \mathbb{N} \qquad u : \mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} u : \mathbb{N} \qquad \dfrac{z : \mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} z : \mathbb{N}}{\vdash_{\mathcal{L}_0^{\mathbb{N}}} \lambda z.z : \mathbb{N} \multimap \mathbb{N}}}{u : \mathbb{N}, v : \mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} \mathsf{iter}\ v\ u\ \lambda z.z : \mathbb{N}}}{x : \mathbb{N} \otimes \mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} \mathtt{let}\ \langle u, v\rangle = x\ \mathtt{in}\ \mathsf{iter}\ v\ u\ \lambda z.z : \mathbb{N}}}{\vdash_{\mathcal{L}_0^{\mathbb{N}}} \lambda x.\mathtt{let}\ \langle u, v\rangle = x\ \mathtt{in}\ \mathsf{iter}\ v\ u\ \lambda z.z : (\mathbb{N} \otimes \mathbb{N}) \multimap \mathbb{N}}$$

Fig. 3. Typing of fst

4.2 Copying linearly

We can also copy natural numbers in these linear calculi. For this, we define a function $C : \mathbb{N} \multimap \mathbb{N} \otimes \mathbb{N}$ that given a number $\bar{n}$ returns a pair $\langle \bar{n}, \bar{n} \rangle$:

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

Lemma 8. *For any number $\bar{n}$, $C\ \bar{n} \longrightarrow^* \langle \bar{n}, \bar{n}\rangle$.*

Proof. By induction on $\bar{n}$.

$$\begin{array}{lcl}
C\ 0 & \longrightarrow & \mathsf{iter}\ 0\ \langle 0,0\rangle\ (\lambda x.\mathtt{let}\ \langle a,b\rangle = x\ \mathtt{in}\ \langle \mathsf{S}a, \mathsf{S}b\rangle) \longrightarrow \langle 0,0\rangle \\
C\ (\mathsf{S}^{t+1}\ 0) & = & \mathsf{iter}\ (\mathsf{S}^{t+1}\ 0)\ \langle 0,0\rangle\ (\lambda x.\mathtt{let}\ \langle a,b\rangle = x\ \mathtt{in}\ \langle \mathsf{S}a, \mathsf{S}b\rangle) \\
 & \longrightarrow^* & (\lambda x.\mathtt{let}\ \langle a,b\rangle = x\ \mathtt{in}\ \langle \mathsf{S}a, \mathsf{S}b\rangle)\langle t,t\rangle \\
 & \longrightarrow & \mathtt{let}\ \langle a,b\rangle = \langle t,t\rangle\ \mathtt{in}\ \langle \mathsf{S}a, \mathsf{S}b\rangle \longrightarrow \langle \mathsf{S}t, \mathsf{S}t\rangle
\end{array}$$

Again, C is valid in System $\mathcal{L}_0^{\mathbb{N}}$ (see Figure 4), and a closed term.

Consider $F = (\lambda x.\mathtt{let}\ \langle a,b\rangle = x\ \mathtt{in}\ \langle \mathsf{S}a, \mathsf{S}b\rangle)$

$$\dfrac{\dfrac{x : \mathbb{N}\otimes\mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} x : \mathbb{N}\otimes\mathbb{N} \qquad \dfrac{\dfrac{a : \mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} a : \mathbb{N}}{a : \mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} \mathsf{S}a : \mathbb{N}} \quad \dfrac{b : \mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} b : \mathbb{N}}{b : \mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} \mathsf{S}b : \mathbb{N}}}{a : \mathbb{N}, b : \mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} \langle \mathsf{S}a, \mathsf{S}b\rangle : \mathbb{N}\otimes\mathbb{N}}}{x : \mathbb{N}\otimes\mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} \mathtt{let}\ \langle a,b\rangle = x\ \mathtt{in}\ \langle \mathsf{S}a, \mathsf{S}b\rangle : \mathbb{N}\otimes\mathbb{N}}}{\vdash_{\mathcal{L}_0^{\mathbb{N}}} (\lambda x.\mathtt{let}\ \langle a,b\rangle = x\ \mathtt{in}\ \langle \mathsf{S}a, \mathsf{S}b\rangle) : (\mathbb{N}\otimes\mathbb{N}) \multimap (\mathbb{N}\otimes\mathbb{N})}$$

$$\dfrac{\dfrac{x : \mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} x : \mathbb{N} \qquad \dfrac{\vdash_{\mathcal{L}_0^{\mathbb{N}}} 0 : \mathbb{N} \quad \vdash_{\mathcal{L}_0^{\mathbb{N}}} 0 : \mathbb{N}}{\vdash_{\mathcal{L}_0^{\mathbb{N}}} \langle 0,0\rangle : \mathbb{N}\otimes\mathbb{N}} \qquad \vdash_{\mathcal{L}_0^{\mathbb{N}}} F : (\mathbb{N}\otimes\mathbb{N}) \multimap (\mathbb{N}\otimes\mathbb{N})}{x : \mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} \mathsf{iter}\ x\ \langle 0,0\rangle\ (\lambda x.\mathtt{let}\ \langle a,b\rangle = x\ \mathtt{in}\ \langle \mathsf{S}a, \mathsf{S}b\rangle) : \mathbb{N}\otimes\mathbb{N}}}{\vdash_{\mathcal{L}_0^{\mathbb{N}}} \lambda x.\mathsf{iter}\ x\ \langle 0,0\rangle\ (\lambda x.\mathtt{let}\ \langle a,b\rangle = x\ \mathtt{in}\ \langle \mathsf{S}a, \mathsf{S}b\rangle) : \mathbb{N} \multimap (\mathbb{N}\otimes\mathbb{N})}$$

Fig. 4. Typing of C

4.3 Primitive Recursive Scheme

We have already shown we can project, and of course we have composition. We now show how to encode, using iterators, a function h defined by primitive recursion from f and g.

First, assume h is defined by the following, simpler scheme (it uses n only once in the second equation):

$$\begin{array}{ll}
h(x,0) & = f(x) \\
h(x,n+1) & = g(x,h(x,n))
\end{array}$$

Given the closed functions $G : \mathbb{N} \multimap \mathbb{N} \multimap \mathbb{N}$ and $F : \mathbb{N} \to \mathbb{N}$, representing g and f, let g' be the term:

$$\lambda y.\lambda z.\texttt{let}\ \langle z_1, z_2\rangle = C\ z\ \texttt{in}\ Gz_1(yz_2) : (\mathbb{N} \multimap \mathbb{N}) \multimap (\mathbb{N} \multimap \mathbb{N})$$

then $h(x, n)$ is defined by the term $(\mathsf{iter}\ n\ F\ g')\ x : \mathbb{N}$, (see Figure 5). Note that the encoding of h is a closed term. This term is valid because F is closed (by assumption), and g' is closed since G is a closed term (by assumption).

Indeed, we can show by induction that $(\mathsf{iter}\ n\ F\ g')\ x$, where x and n are numbers, reduces to the number $h(x, n)$; we use Lemma 8 to copy numbers:

$$\begin{array}{lll}
(\mathsf{iter}\ 0\ F\ g')\ x & \longrightarrow & (F\ x) = h(x,0) \\
(\mathsf{iter}\ (\mathsf{S}^{n+1}\ 0)\ F\ g')\ x & \longrightarrow^* & \texttt{let}\ \langle z_1, z_2\rangle = \langle x, x\rangle\ \texttt{in}\ Gz_1((\mathsf{iter}\ (\mathsf{S}^n\ 0)\ F\ g')z_2) \\
 & \longrightarrow & G\ x((\mathsf{iter}\ (\mathsf{S}^n\ 0)\ F\ g')x) = h(x, n+1) \text{ by induction.}
\end{array}$$

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ \vdash_{\mathcal{L}_0^{\mathbb{N}}} G : \mathbb{N} \multimap \mathbb{N} \multimap \mathbb{N}\end{array} \quad z_1 : \mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} z_1 : \mathbb{N}}{z_1 : \mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} Gz_1 : \mathbb{N} \multimap \mathbb{N}} \qquad \dfrac{y : \mathbb{N} \multimap \mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} y : \mathbb{N} \multimap \mathbb{N} \quad z_2 : \mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} z_2 : \mathbb{N}}{y : \mathbb{N} \multimap \mathbb{N}, z_2 : \mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} yz_2 : \mathbb{N}}}{y : \mathbb{N} \multimap \mathbb{N}, z_1 : \mathbb{N}, z_2 : \mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} Gz_1(yz_2) : \mathbb{N}}$$

$$\dfrac{\dfrac{\dfrac{z : \mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} C\ z : \mathbb{N} \otimes \mathbb{N} \quad y : \mathbb{N} \multimap \mathbb{N}, z_1 : \mathbb{N}, z_2 : \mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} Gz_1(yz_2) : \mathbb{N}}{y : \mathbb{N} \multimap \mathbb{N}, z : \mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} \texttt{let}\ \langle z_1, z_2\rangle = C\ z\ \texttt{in}\ Gz_1(yz_2) : \mathbb{N}}}{y : \mathbb{N} \multimap \mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} \lambda z.\texttt{let}\ \langle z_1, z_2\rangle = C\ z\ \texttt{in}\ Gz_1(yz_2) : \mathbb{N} \multimap \mathbb{N}}}{\vdash_{\mathcal{L}_0^{\mathbb{N}}} \lambda yz.\texttt{let}\ \langle z_1, z_2\rangle = C\ z\ \texttt{in}\ Gz_1(yz_2) : (\mathbb{N} \multimap \mathbb{N}) \multimap (\mathbb{N} \multimap \mathbb{N})}$$

$$\dfrac{\dfrac{x : \mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} x : \mathbb{N} \quad \begin{array}{c}\vdots\\ \vdash_{\mathcal{L}_0^{\mathbb{N}}} F : \mathbb{N} \multimap \mathbb{N}\end{array} \quad \vdash_{\mathcal{L}_0^{\mathbb{N}}} g' : (\mathbb{N} \multimap \mathbb{N}) \multimap (\mathbb{N} \multimap \mathbb{N})}{n : \mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} \mathsf{iter}\ n\ F\ g' : \mathbb{N} \multimap \mathbb{N} \quad x : \mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} x : \mathbb{N}}}{n : \mathbb{N}, x : \mathbb{N} \vdash_{\mathcal{L}_0^{\mathbb{N}}} (\mathsf{iter}\ n\ F\ g')x : \mathbb{N}}$$

Fig. 5. Typing of functions defined by primitive recursion

Now to encode the standard primitive recursive scheme, which has an extra n in the last equation, all we need to do is copy n:

$$\begin{array}{l}
h(x, n) = \texttt{let}\ \langle n_1, n_2\rangle = C\ n\ \texttt{in}\ sx \text{ where} \\
\qquad s = \mathsf{iter}\ n_2\ F\ (\lambda y.\lambda z.\texttt{let}\ \langle z_1, z_2\rangle = C\ z\ \texttt{in}\ Gz_1(yz_2)n_1)
\end{array}$$

5 Beyond Primitive Recursion

In this section we show that it is possible to encode more than primitive recursive functions in System $\mathcal{L}^{\mathbb{N}}$, by giving the encoding of a non primitive recursive function: Ackermann's function.

$$\begin{array}{ll} ack(0,n) & = \mathsf{S}\ n \\ ack(\mathsf{S}\ n, 0) & = ack(n, \mathsf{S}\ 0) \\ ack(\mathsf{S}\ n, \mathsf{S}\ m) & = ack(n, ack(\mathsf{S}\ n, m)) \end{array}$$

In a higher-order functional language, it can be defined as follows:
Let $\mathsf{succ} = \lambda x.\mathsf{S}\ x : \mathbb{N} \multimap \mathbb{N}$, then $ack(m,n) = a\ m\ n$ where:

$$\begin{array}{llll} a\ 0 & = \mathsf{succ} & A\ g\ 0 & = g(\mathsf{S}\ 0) \\ a\ (\mathsf{S}\ n) & = A\ (a\ n) & A\ g\ (\mathsf{S}\ n) & = g(A\ g\ n) \end{array}$$

Lemma 9. *Both definitions are equivalent: $a\ x\ y = ack(x,y)$, for all numbers x, y.*

Proof. By induction on x, proving first by induction on n that if $g = \lambda y.ack(x,y)$ then $A\ g\ n = ack(\mathsf{S}\ x, n)$. The case $x = 0$ is trivial. By definition, $a\ (\mathsf{S}\ n) = A\ (a\ n)$, and by induction this is $A(\lambda y.ack(n,y))$. Therefore by Lemma 10 below, $a(\mathsf{S}\ n)z = ack(\mathsf{S}\ n, z)$.

Lemma 10. *If $g = \lambda y.ack(x,y)$ then $A\ g\ n = ack(\mathsf{S}\ x, n)$.*

Proof. By induction on n.

We can define a and A in System $\mathcal{L}^{\mathbb{N}}$ as follows:

$$\begin{array}{ll} a & = \lambda n.\mathsf{iter}\ n\ \mathsf{succ}\ A : \mathbb{N} \multimap \mathbb{N} \multimap \mathbb{N} \\ A\ g\ n & = \lambda g\ n.\mathsf{iter}\ (\mathsf{S}\ n)\ (\mathsf{S}\ 0)\ g : (\mathbb{N} \multimap \mathbb{N}) \multimap \mathbb{N} \multimap \mathbb{N} \end{array}$$

We show by induction that this encoding is correct:

– $a\ 0 = \mathsf{iter}\ 0\ \mathsf{succ}\ A \longrightarrow \mathsf{succ}$
  $A\ g\ 0 = \mathsf{iter}\ (\mathsf{S}\ 0)\ (\mathsf{S}\ 0)\ g \longrightarrow g(\mathsf{S}\ 0)$
– $a\ (\mathsf{S}\ n) = \mathsf{iter}\ (\mathsf{S}^n\ 0)\ \mathsf{succ}\ A \longrightarrow A(\mathsf{iter}\ n\ \mathsf{succ}\ A) = A(a\ n)$
  $A\ g\ (\mathsf{S}\ n) = \mathsf{iter}\ (\mathsf{S}(\mathsf{S}\ n))\ (\mathsf{S}\ 0)\ g \longrightarrow g(\mathsf{iter}\ (\mathsf{S}\ n)\ (\mathsf{S}\ 0)\ g) = g(A\ g\ n)$.

Then Ackermann's function can be defined in System $\mathcal{L}^{\mathbb{N}}$ (see Figure 6) as:

$$ack = \lambda m\ n.(\mathsf{iter}\ m\ \mathsf{succ}\ (\lambda gu.\mathsf{iter}\ (\mathsf{S}\ u)\ (\mathsf{S}\ 0)\ g))\ n : \mathbb{N} \multimap \mathbb{N} \multimap \mathbb{N}$$

The correctness of this encoding follows directly from the lemma above.

Note that $\mathsf{iter}\ (\mathsf{S}\ u)\ (\mathsf{S}\ 0)\ g$ cannot be typed in System $\mathcal{L}_0^{\mathbb{N}}$, because g is a free variable. System $\mathcal{L}^{\mathbb{N}}$ allows building the term with the free variable g, but does not allow reduction until it is closed.

Every function in System $\mathcal{L}_0^{\mathbb{N}}$ can be defined in the system $H(\emptyset)$ studied in [11]: the syntax is different but the typing rules are equivalent. It has been proved (see [11] for details) that Ackermann's function cannot be represented in $H(\emptyset)$, therefore it cannot be represented in System $\mathcal{L}_0^{\mathbb{N}}$ either. Thus, System $\mathcal{L}_0^{\mathbb{N}}$ is strictly less powerful than System $\mathcal{L}^{\mathbb{N}}$.

$$\dfrac{\dfrac{x:\mathbb{N}\vdash_{\mathcal{L}_0^{\mathbb{N}}} x:\mathbb{N}}{x:\mathbb{N}\vdash_{\mathcal{L}_0^{\mathbb{N}}} \mathsf{S}\,x:\mathbb{N}}}{\vdash_{\mathcal{L}_0^{\mathbb{N}}} \mathsf{succ} = \lambda x.\mathsf{S}\,x : \mathbb{N}\multimap\mathbb{N}}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{u:\mathbb{N}\vdash_{\mathcal{L}_0^{\mathbb{N}}} u:\mathbb{N}}{u:\mathbb{N}\vdash_{\mathcal{L}_0^{\mathbb{N}}} \mathsf{S}\,u:\mathbb{N}} \quad \dfrac{\vdash_{\mathcal{L}_0^{\mathbb{N}}} 0:\mathbb{N}}{\vdash_{\mathcal{L}_0^{\mathbb{N}}} \mathsf{S}\,0:\mathbb{N}} \quad g:\mathbb{N}\multimap\mathbb{N}\vdash_{\mathcal{L}_0^{\mathbb{N}}} g:\mathbb{N}\multimap\mathbb{N}}{g:\mathbb{N}\multimap\mathbb{N}, u:\mathbb{N}\vdash_{\mathcal{L}_0^{\mathbb{N}}} \mathsf{iter}\;(\mathsf{S}\,u)\;(\mathsf{S}\,0)\;g:\mathbb{N}}}{g:\mathbb{N}\multimap\mathbb{N}\vdash_{\mathcal{L}_0^{\mathbb{N}}} \lambda u.\mathsf{iter}\;(\mathsf{S}\,u)\;(\mathsf{S}\,0)\;g:(\mathbb{N}\multimap\mathbb{N})}}{\vdash_{\mathcal{L}_0^{\mathbb{N}}} A = \lambda gu.\mathsf{iter}\;(\mathsf{S}\,u)\;(\mathsf{S}\,0)\;g:(\mathbb{N}\multimap\mathbb{N})\multimap(\mathbb{N}\multimap\mathbb{N})}$$

$$\dfrac{\dfrac{\dfrac{m:\mathbb{N}\vdash_{\mathcal{L}_0^{\mathbb{N}}} m:\mathbb{N} \quad \vdash_{\mathcal{L}_0^{\mathbb{N}}} \mathsf{succ}:\mathbb{N}\multimap\mathbb{N} \quad \vdash_{\mathcal{L}_0^{\mathbb{N}}} A:(\mathbb{N}\multimap\mathbb{N})\multimap(\mathbb{N}\multimap\mathbb{N})}{m:\mathbb{N}\vdash_{\mathcal{L}_0^{\mathbb{N}}} \mathsf{iter}\;m\;\mathsf{succ}\;A:\mathbb{N}\multimap\mathbb{N}} \quad n:\mathbb{N}\vdash_{\mathcal{L}_0^{\mathbb{N}}} n:\mathbb{N}}{m:\mathbb{N}, n:\mathbb{N}\vdash_{\mathcal{L}_0^{\mathbb{N}}} (\mathsf{iter}\;m\;\mathsf{succ}\;A)\;n:\mathbb{N}}}{\vdash_{\mathcal{L}_0^{\mathbb{N}}} \lambda m\,n.(\mathsf{iter}\;m\;\mathsf{succ}\;A)\;n:\mathbb{N}\multimap\mathbb{N}\multimap\mathbb{N}}$$

Fig. 6. Typing of Ackermann's function

6 Conclusions and Future Work

Closed reduction strategies impose strong constraints on the application of reduction rules, but despite this fact, they can simulate both call-by-name and call-by-value evaluations in the λ-calculus, and also more efficient evaluations (since reductions can take place under abstractions and thus achieve more sharing of computations); similar results hold for PCF (see [5]).

In this paper we have shown that in the case of a linear λ-calculus with iterators, the use of closed reduction strategies has another benefit: not only we can gain in efficiency, but also we gain in computational power, thanks to the fact that we can relax the constraints on the construction of iterator terms.

The linear system with iterators is not computationally complete (it is strongly normalising). A question that remains to study is whether it is possible to define a linear and computationally complete version of PCF using closed reductions.

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 linear functions, 2006. Available from www.dcs.kcl.ac.uk/staff/ian/papers/powlf.pdf.
3. N. Çağman and J. R. Hindley. Combinatory weak reduction in lambda calculus. *Theoretical Computer Science*, 198(1–2):239–249, 1998.

4. M. Fernández and I. Mackie. Closed reduction in the λ-calculus. In J. Flum and M. Rodríguez-Artalejo, editors, *Proceedings of Computer Science Logic (CSL'99)*, volume 1683 of *Lecture Notes in Computer Science*, pages 220–234. Springer-Verlag, September 1999.
5. M. Fernández, I. Mackie, and F.-R. Sinot. Closed reduction: explicit substitutions without alpha conversion. *Mathematical Structures in Computer Science*, 15(2):343–381, 2005.
6. J.-Y. Girard. Linear Logic. *Theoretical Computer Science*, 50(1):1–102, 1987.
7. 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.
8. J.-Y. Girard. Towards a geometry of interaction. In J. W. Gray and A. Scedrov, editors, *Categories in Computer Science and Logic: Proc. of the Joint Summer Research Conference*, pages 69–108. American Mathematical Society, Providence, RI, 1989.
9. M. Hofmann. Linear types and non-size-increasing polynomial time computation. In *Proc. Logic in Computer Science (LICS'99)*. IEEE Computer Society, 1999.
10. J.-W. Klop, V. van Oostrom, and F. van Raamsdonk. Combinatory reduction systems, introduction and survey. *Theoretical Computer Science*, 121:279–308, 1993.
11. U. D. Lago. The geometry of linear higher-order recursion. In P. Panangaden, editor, *Proceedings of the Twentieth Annual IEEE Symp. on Logic in Computer Science, LICS 2005*, pages 366–375. IEEE Computer Society Press, June 2005.
12. I. Mackie, L. Román, and S. Abramsky. An internal language for autonomous categories. *Journal of Applied Categorical Structures*, 1(3):311–343, 1993.
13. M. Newman. On theories with a combinatorial definition of "equivalence". *Annals of Mathematics*, 43(2):223–243, 1942.

Transformation for Refining Unraveled Conditional Term Rewriting Systems

Naoki Nishida, Tomohiro Mizutani, and Masahiko Sakai

Graduate School of Information Science, Nagoya University
Furo-cho, Chikusa-ku, Nagoya 464-8603, Japan
{`nishida@`, `mizutani@sakabe.i.`, `sakai@`}`is.nagoya-u.ac.jp`

Abstract. Unravelings, which transform conditional term rewriting systems (CTRSs) into unconditional term rewriting systems, are useful for analyzing properties of CTRSs. To compute reduction sequences of CTRSs, the restriction by a particular context-sensitive and membership condition is imposed on reductions of the unraveled CTRSs. The condition is determined by extra function symbols introduced due to the unravelings. In this paper, we propose a method to weaken the restriction, that is, to reduce the number of extra symbols. We first improve the unraveling for deterministic CTRSs, and then propose a transformation that folds two successively used rewrite rules in the unraveled CTRSs, which satisfy a condition, to a rewrite rule that simulates reductions by the two rules.

1 Introduction

Unravelings are transformations from conditional term rewriting systems (for short, CTRSs) into unconditional term rewriting systems (TRSs). They are useful for analyzing properties of CTRSs. For example, 'effective termination', in which CTRSs are terminating and the recursive reduction of the instantiated conditional parts also terminates, is an important property of CTRSs and it can be guaranteed by termination of the unraveled CTRSs [6, 11]. An unraveling for normal CTRSs was investigated by Bergstra and Klop [3]. This concept was revisited by Marchiori who discussed its properties such as syntactic ones, termination, modularity, and so on [6]. He also proposed the unraveling for join CTRSs. Ohlebusch proposed an unraveling for deterministic 3-CTRSs to prove termination of logic programs [10]. A variant of Ohlebusch's unraveling is used in several papers [4, 7–9].

It is well-known that reductions of CTRSs are much more complicated than those of TRSs. One of the reasons is that the recursive reduction is necessary to evaluate instantiated conditional parts. To compute reduction sequences of CTRSs, unravelings appear attractive. An unraveling is said to be *simulation-complete* for a CTRS over a signature if both *reachability* and *unreachability* of terms over the signature are preserved by the unraveling [7–9]. In general,

$$\rho : l \to r \Leftarrow s_1 \to t_1 \land \cdots \land s_k \to t_k$$
$$\Downarrow \mathbb{U}$$
$$\{\ l \to \mathtt{u}^{\rho}_1(s_1, \overrightarrow{x_1}),\ \mathtt{u}^{\rho}_1(t_1, \overrightarrow{x_1}) \to \mathtt{u}^{\rho}_2(s_2, \overrightarrow{x_2}),\ \cdots,\ \mathtt{u}^{\rho}_k(t_k, \overrightarrow{x_k}) \to r\ \}$$

Fig. 1. Outline of the unraveling for deterministic CTRSs.

unravelings are not simulation-complete for arbitrary target CTRSs because the unraveled CTRSs are simple approximations of the original CTRSs [6, 11]. However, it was shown that the restriction by a particular *context-sensitive* and *membership* condition to reductions of the unraveled CTRSs preserves unreachability of the original CTRSs, that is, simulation-completeness of the unravelings [8].

Unravelings are generally done by decomposing each conditional rule to some unconditional rules that are supposed to be used in a fixed order (see Fig. 1). A reduction from $l\sigma$ to $r\sigma$ by the conditional rule ρ is simulated by a reduction sequence by the corresponding unconditional rules; the sequence starts from $l\sigma$; in the sequence, each extra function symbol $\mathtt{u}^{\rho}_i$ (called a *U symbol*) not in the original signature checks sequentially reachability from $s_i\sigma$ to $t_i\sigma$ (evaluates the condition $s_i \to t_i$ with σ); the sequence ends at $r\sigma$ after all conditions are evaluated successfully. We are sure that the unravelings preserve reachability on terms over the original signatures. On the other hand, the unravelings do not preserve unreachability for all CTRSs because unexpected reduction sequences are sometimes caused by disobeying the application order of rules whose left-hand sides are rooted with the U symbols [6, 11]. To avoid this, a restriction to reductions of the unraveled CTRSs is required, which prohibits reductions associated with the following redexes:

– (context-sensitive condition) redexes that occur strictly below U symbols, except for the first arguments of the U symbols , or
– (membership condition) redexes that contains a U symbol in their proper subterms.

In this way, the restriction by the above context-sensitive and membership condition is imposed on reductions of the unraveled CTRSs to maintain simulation-completeness [8]. As another approach to simulation-completeness, it was shown that the unraveled CTRSs are simulation-complete for the original CTRSs if the unraveled ones are either left-linear or both right-linear and non-erasing [7].

In this paper, we try to construct unconditional TRSs that are simulation-complete for the original CTRSs without the context-sensitive and membership condition. We first improve the unraveling for deterministic CTRSs so that the number of unraveled rules is less than those with the ordinary unraveling. We then propose a transformation, which is applied to the unraveled CTRSs, to remove the U symbols as many as possible from the unraveled CTRSs. The transformation folds two rules used successively in reduction sequences into one rule (see Fig. 2). We show a delicate condition that U symbols to be removed should satisfy, and we tighten it to maintain an advantage of CTRSs associated with the ‘let’ structure of functional programs. Removing U symbols leads to the

$$\left(\begin{array}{l} S \cup \{\, l_1 \to \mathtt{u}_i^\rho(t_{i,1}\delta, \ldots, t_{i,m_i}\delta, \overrightarrow{x_i}), \\ \qquad\qquad\quad \mathtt{u}_i^\rho(t_{i,1}, \ldots, t_{i,m_i}, \overrightarrow{x_i}) \quad \to r_2 \,\} \end{array}, \mu \right) \quad \Longrightarrow^{\mathbb{T}} \quad (S \cup \{\, l_1 \to r_2\delta \,\}, \mu')$$
$$\text{where } \mu \text{ is updated to } \mu' \text{ w.r.t. } \mathtt{root}(r_2)$$

Fig. 2. Outline for removing U symbols by the transformation $\mathbb{T}$.

relaxation of the restriction by the context-sensitive and membership condition because the condition depends on the existence of U symbols. We also show correctness of the transformation, and show that the composition of the unraveling and the transformation is also an unraveling. In the case that all U symbols are removed, we require no longer any context-sensitive and membership condition for simulation-completeness. We also show that the transformation preserves confluence of CTRSs.

Unfortunately, the transformation often fails to remove all U symbols. However, we have some advantages even if not all U symbols are removed.

- The context-sensitive condition is sometimes removed.
- Non-termination of CTRSs is preserved by the transformation. Thus, by showing termination of the unraveled CTRSs, we can prove 'effective termination' of the original CTRSs.

There are some cases in which the improvement in this paper increases the effect of the transformation (see Section 4). If we succeed in removing all U symbols, there are furthermore advantages as follows.

- The context-sensitive and membership condition is not necessary.
- Confluence of CTRSs is preserved. Accordingly, to prove confluence of the CTRSs, we can use many techniques for proving confluence of TRSs.

Therefore, the transformation is always harmless and we can sometimes obtain some advantages.

The unraveling for deterministic CTRSs is used in the *inversion compilers* proposed in [8, 9]. The compilers transform a given constructor TRS into a CTRS that computes (partial) inverse images of functions defined in the TRS. The compilers then unravel the CTRS to a TRS whose rules may have extra variables. Since inverse images are not mappings in general, CTRSs obtained by the compilers are not always confluent. From this reason, this paper does not assume confluence for CTRSs. The transformation in this paper is sometimes useful for simplifying TRSs obtained by the compilers. We will show an example at the end of this paper.

This paper is organized as follows. In Section 2, we give notations of term rewriting. In Section 3, we improve the unraveling for deterministic CTRSs. In Section 4, we propose a transformation that removes extra function symbols introduced due to the improved unraveling. In Section 5, we discuss confluence of CTRSs and the unraveled CTRSs. In Section 6, we enhance the condition for removing the extra function symbols in the transformation. In Section 7, we offer some concluding remarks.

2 Preliminaries

This paper follows the basic notions of term rewriting [2, 11]. In this section we outline the basic notations.

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

An *(oriented) conditional rewrite rule* over a signature $\mathcal{F}$ is a triple $(l, r, \mathcal{C}nd)$, denoted by $l \to r \Leftarrow \mathcal{C}nd$, such that the *left-hand side* (*lhs*) l is a non-variable term in $\mathcal{T}(\mathcal{F},\mathcal{V})$, the *right-hand side* (*rhs*) r is a term in $\mathcal{T}(\mathcal{F},\mathcal{V})$, and the *conditional part* $\mathcal{C}nd$ is in form of $s_1 \to t_1 \wedge \cdots \wedge s_n \to t_n$ $(n \geq 0)$ of terms s_i and t_i in $\mathcal{T}(\mathcal{F},\mathcal{V})$. In particular, the conditional rewrite rule $l \to r \Leftarrow \mathcal{C}nd$ is said to be an *(unconditional) rewrite rule* if $n = 0$, and we may abbreviate it to $l \to r$. We say that a binary relation $\approx$ and a substitution σ *satisfy* the conditional part $\mathcal{C}nd$, written by $\mathcal{C}nd(\sigma, \approx)$, if $s_i\sigma \approx t_i\sigma$ for $1 \leq i \leq n$. We denote $l \to r \Leftarrow \mathcal{C}nd$ with a unique label ρ by $\rho : l \to r \Leftarrow \mathcal{C}nd$. To simplify notations, we may write labels instead of the corresponding rules. For a conditional rewrite rule $\rho : l \to r \Leftarrow \mathcal{C}nd$, variables occurring not in l but in either r or $\mathcal{C}nd$ are called *extra variables* of ρ. The set of all extra variables of ρ is denoted by $\mathcal{EV}ar(\rho)$.

Let R be a finite set of conditional rewrite rules over a signature $\mathcal{F}$. The *n-level rewrite relation* $\xrightarrow[n]{}_R$ of R is defined inductively as follows: $\xrightarrow[0]{}_R = \emptyset$ and $\xrightarrow[n+1]{}_R = \{(C[l\sigma]_p, C[r\sigma]_p) \mid \rho : l \to r \Leftarrow \mathcal{C}nd \in R, \mathcal{C}nd(\sigma, \xrightarrow[n]{*}_R)\}$. The *rewrite relation* $\to_R$ of R is defined as $\to_R = \bigcup_{n\geq 0} \xrightarrow[n]{}_R$. To specify the position p and the rule ρ, we write $s \to^p_R t$ or $s \to^{[p,\rho]}_R t$. An *(oriented) conditional rewriting system* (*CTRS*) over a signature $\mathcal{F}$ is an abstract reduction system $(\mathcal{T}(\mathcal{F},\mathcal{V}), \to_R)$ of $\mathcal{T}(\mathcal{F},\mathcal{V})$ and the rewrite relation of a finite set R of conditional rewrite rules over $\mathcal{F}$. We use the set R of rules to denote the CTRS $(\mathcal{T}(\mathcal{F},\mathcal{V}), \to_R)$. A CTRS is called a *term rewriting system with extra variables* (*EV-TRS*) if it contains only unconditional rewrite rules. Specifically, it is a *term rewriting system* (*TRS*) if $\mathcal{V}ar(l) \supseteq \mathcal{V}ar(r)$ for every its rule $l \to r$.

A CTRS R is called a *1-CTRS* if every rule in R has no extra variable, a *2-CTRS* if every rule in R has no extra variable in its right-hand side, a *3-CTRS* if for every rule in R all extra variables of the rule appear in the conditional part, and a *4-CTRS* if no restriction is imposed. A conditional rewrite rule $\rho : l \to r \Leftarrow s_1 \to t_1 \cdots s_k \to t_k$ is called *deterministic* if $\mathcal{V}ar(s_i) \subseteq \mathcal{V}ar(l, t_1, \ldots, t_{i-1})$ for $1 \leq i \leq k$. A CTRS is called *normal* if every its rule $l \to r \Leftarrow s_1 \to t_1 \wedge \cdots \wedge s_k \to t_k$ satisfies that $t_1,\ldots,t_k$ are ground normal forms of $R_u = \{\, l \to r \mid l \to r \Leftarrow \mathcal{C}nd \in R \,\}$.

We use the notion of *context-sensitive reduction* in [5]. Let $\mathcal{F}$ be a signature. A *context-sensitive condition* (*replacement mapping*) μ is a mapping from $\mathcal{F}$ to a set of integer lists such that $\mu(f) \subseteq \{1,\ldots,n\}$ for n-ary symbols f *in* $\mathcal{F}$. When $\mu(f)$ is not defined explicitly, we assume that $\mu(f) = \{1,\ldots,n\}$. The set $\mathcal{O}_\mu(t)$ of *replacing* (*active*) *positions* of a term t is defined inductively as follows: $\mathcal{O}_\mu(x) = \emptyset$ if $x \in \mathcal{V}$, and $\mathcal{O}_\mu(f(t_1,\ldots,t_n)) = \{ip \mid f \in \mathcal{F}, i \in \mu(f), p \in \mathcal{O}_\mu(t_i)\}$. The *context-sensitive reduction* of an EV-TRS R with μ is defined as $\rightarrow_{(R,\mu)} = \{(s,t) \mid s \rightarrow^p_R t, p \in \mathcal{O}_\mu(s)\}$. An abstract reduction system $(\mathcal{T}(\mathcal{F},\mathcal{V}), \rightarrow_{(R,\mu)})$, denoted by (R,μ), is called a *context-sensitive reduction system* (*CS-TRS*).

In this paper we use a simple variant of *membership-conditional systems* [13]. For an EV-TRS R, the *membership-conditional reduction* of $\rightarrow_R$ by a *membership condition* $\in T$ (where $T \subseteq \mathcal{T}(\mathcal{F},\mathcal{V})$) is defined as $\xrightarrow[\in T]{}_R = \{(C[l\sigma]_p, C[r\sigma]_p) \mid l \rightarrow r \in R, (\forall x \in \mathcal{V}ar(l,r), x\sigma \in T)\}$. The membership-conditional reduction for $\rightarrow_{(R,\mu)}$ is defined similarly as $\xrightarrow[\in T]{}_{(R,\mu)}$.

3 Improvement of Unraveling for Deterministic CTRSs

In this section, we improve the unraveling (denoted by $\mathbb{U}_O$ in this paper) for deterministic CTRSs, which is proposed in [4, 7–9]. The unraveling $\mathbb{U}_O$ is a variant of Ohlebusch's unraveling [10]. The idea for this improvement is based on the unraveling for normal CTRSs [6], which is denoted by $\mathbb{U}_N$.

We first explain the intuitive idea of our improvement method. The unraveling $\mathbb{U}_O$ decomposes each conditional rewrite rule ρ having k conditions into $k+1$ unconditional rewrite rules that are used to evaluate the conditions in left-to-right order, introducing 'fresh' extra function symbols, called *U symbols* (see Fig. 1). For example, the conditional rewrite rule

$$\rho_1 : \mathsf{f}(x,y) \rightarrow z \Leftarrow \mathsf{g}(x) \rightarrow w \wedge \mathsf{g}(y) \rightarrow z \wedge \mathsf{h}(w,x) \rightarrow z$$

is unraveled into the following four unconditional rewrite rules, by introducing U symbols u_1, u_2 and u_3:

$$\mathbb{U}_O(\rho_1) = \left\{ \begin{array}{ll} \mathsf{f}(x,y) \rightarrow \mathsf{u}_1(\mathsf{g}(x),x,y), & \mathsf{u}_1(w,x,y) \rightarrow \mathsf{u}_2(\mathsf{g}(y),w,x), \\ \mathsf{u}_2(z,w,x) \rightarrow \mathsf{u}_3(\mathsf{h}(w,x),z), & \mathsf{u}_3(z,z) \rightarrow z \end{array} \right\}.$$

The application order of these rules in a reduction sequence corresponds exactly to the order of evaluating the conditions. However, the order between u_1 and u_2 is not necessary because the first and second conditions $\mathsf{g}(x) \rightarrow w$ and $\mathsf{g}(y) \rightarrow z$ can be evaluated in parallel. The reason is that all variables x, y used in the evaluation already appear in the lhs $\mathsf{f}(x,y)$ of the conditional rule. From this fact, we can combine u_1 and u_2 into one symbol u'_1 as follows:

$$\mathsf{f}(x,y) \rightarrow \mathsf{u}'_1(\mathsf{g}(x),\mathsf{g}(y),x) \text{ and } \mathsf{u}'_1(w,z,x) \rightarrow \mathsf{u}_3(\mathsf{h}(w,x),z).$$

Thus, to allow simultaneous evaluation of conditions that can be evaluated in parallel, we improve the ordinary unraveling $\mathbb{U}_O$ so that some conditional rules are decomposed to less unconditional rules. This idea comes from the unraveling $\mathbb{U}_N$ for normal CTRSs [6].

This improvement is formalized as follows. Here, we denote by $\overrightarrow{T}$ the sequence of the elements (in some fixed order) in the finite set T of terms, and denote $\bigcup_{t \in T} \mathcal{V}ar(t)$ by $\mathcal{V}ar(T)$.

Definition 1. *Let R be a deterministic CTRS over a signature $\mathcal{F}$. We consider a conditional rewrite rule $\rho : l \to r \Leftarrow \bigwedge_{j=1}^{m_1} s_{1,j} \to t_{1,j} \wedge \cdots \wedge \bigwedge_{j=1}^{m_k} s_{k,j} \to t_{k,j} \in R$ [1] such that $\mathcal{V}ar(s_{i,j}) \subseteq \mathcal{V}ar(l) \cup \mathcal{V}ar(T_1) \cup \cdots \cup T_{i-1})$ for all i and j, where $T_i = \{t_{i,1}, \ldots, t_{i,m_i}\}$. For every conditional rewrite rule ρ in the above form, let $|\rho|$ denote the number of groups of conditions in ρ (that is, $|\rho| = k$), and we need k 'fresh' function symbols $\mathtt{u}_1^\rho, \ldots, \mathtt{u}_k^\rho$, called* U symbols*, in the transformation. We transform ρ into a set $\mathbb{U}(\rho)$ of $k+1$ unconditional rewrite rules as follows:*

$$\mathbb{U}(\rho) = \begin{cases} \hfill l \to \mathtt{u}_1^\rho(s_{1,1}, \ldots, s_{1,m_1}, \overrightarrow{X_1}), \\ \mathtt{u}_1^\rho(t_{1,1}\ldots, t_{1,m_1}, \overrightarrow{X_1}) \to \mathtt{u}_2^\rho(s_{2,1}, \ldots, s_{2,m_2}, \overrightarrow{X_2}), \\ \hfill \vdots \hfill \\ \mathtt{u}_k^\rho(t_{k,1}, \ldots, t_{k,m_k}, \overrightarrow{X_k}) \to r \end{cases}$$

where $S_i = \{s_{i,1}, \ldots, s_{i,m_i}\}$ and $X_i = (\mathcal{V}ar(l) \cup \mathcal{V}ar(T_1 \cup \cdots \cup T_{i-1})) \cap (\mathcal{V}ar(T_i) \cup \mathcal{V}ar(S_{i+1} \cup T_{i+1} \cup \cdots \cup S_k \cup T_k) \cup \mathcal{V}ar(r))$ for $1 \leq i \leq k$. The set $\mathbb{U}(R) = \bigcup_{\rho \in R} \mathbb{U}(\rho)$ is an EV-TRS over the extended signature $\mathcal{F}_{\mathbb{U}(R)} = \mathcal{F} \cup \{\mathtt{u}_i^\rho \mid \rho \in R, 1 \leq i \leq |\rho|\}$.

The set X_i in the above definition plays the role of delivering values to the later conditions; these values are obtained via variables in either l, T_1, $\cdots$ or T_{i-1}, and they are used in either r, $S_{i+1}, \ldots, S_k$ or $T_i, \ldots, T_k$. The above unraveling $\mathbb{U}$ is based on the unraveling $\mathbb{U}_O$ [4, 7–9], in which the definition of X_i is different from the original definition [10]. For this reason, all results in this paper or [7–9] do not hold for the original unraveling. In the above definition, one can freely divide a conditional part into groups of conditions that satisfy the variable-occurrence condition. The set $\mathbb{U}(\rho)$ is equal to $\mathbb{U}_O(\rho)$ if $m_i = 1$ for every i, and it is equal to $\mathbb{U}_N(\rho)$ if $k = 1$. Thus, $\mathbb{U}_O$ and $\mathbb{U}_N$ are special cases of $\mathbb{U}$. For the purpose of reducing the number of unconditional rules, this paper assumes that ρ in the above definition satisfies $\mathcal{V}ar(s_{i,j}) \not\subseteq \mathcal{V}ar(l) \cup \mathcal{V}ar(T_1 \cup \cdots \cup T_{i-2})$ for $1 < i \leq k$ and $1 \leq j \leq m_i$. Under this assumption, $\mathbb{U}(\rho)$ is determined uniquely.

Example 2. The conditional rule ρ_1 is unraveled by $\mathbb{U}$ into $\mathbb{U}(\rho_1) = \{\, \mathsf{f}(x, y) \to \mathtt{u}_1'(\mathsf{g}(x), \mathsf{g}(y), x), \mathtt{u}_1'(w, z, x) \to \mathtt{u}_3(\mathsf{h}(w, x), z), \mathtt{u}_3(z, z) \to z\}$. The number of rules obtained by $\mathbb{U}$ is five while that obtained by $\mathbb{U}_O$ is six.

Next, we give the notion of *simulation-completeness* based on completeness of *ultra-properties* [6].

Definition 3. *Let U be an unraveling and R be a CTRS over a signature $\mathcal{F}$.*

– *U is said to be $\xrightarrow{*}_R$-*preserving *for R if U preserves reachability of R, that is, for all terms s and $t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$, $s \xrightarrow{*}_R t$ implies $s \xrightarrow{*}_{U(R)} t$.*

[1] It is clear that every deterministic conditional rewrite rule can be expressed like this.

- *U is* simulation-sound *for R if U is sound for unreachability of R, that is, for all s and $t \in \mathcal{T}(\mathcal{F},\mathcal{V})$, $s \xrightarrow{*}_R t$ if $s \xrightarrow{*}_{U(R)} t$.*
- *U is* simulation-complete *for R if U is complete ($\xrightarrow{*}_R$-preserving and sound for $\xrightarrow{*}_R$), that is, for all s and $t \in \mathcal{T}(\mathcal{F},\mathcal{V})$, $s \xrightarrow{*}_R t$ if and only if $s \xrightarrow{*}_{U(R)} t$.*

We similarly define these properties for the unraveled system $U(R)$.

The definition of simulation-completeness in [7–9] is different from that used in this paper. More precisely, simulation-completeness in [7–9] corresponds to simulation-soundness in this paper. However, discussions on the simulation-completeness in those papers are essentially equivalent because $\xrightarrow{*}_R$-preserving holds for all CTRSs.

All proposed unravelings are $\xrightarrow{*}_R$-preserving for every target CTRS because $\xrightarrow{*}_R$-preserving is a necessary condition for a transformation that is an 'unraveling'. On the other hand, in general, they are not simulation-sound for all target CTRSs, and hence are simulation-incomplete. The cause is that the unraveled CTRSs are approximations of the original CTRSs. In [6], we can find a counterexample against simulation-completeness of $\mathbb{U}_N$, $\mathbb{U}_O$ and Ohlebusch's unraveling.

A restriction to reductions of the unraveled CTRSs for avoiding this difficulty on simulation-incompleteness of $\mathbb{U}_O$ is shown in [8], which is done by a particular *context-sensitive* and *membership* condition that prohibits reductions associated with the following redexes:

- redexes that occur strictly below U symbols, except for the first arguments of the U symbols, or
- redexes that contain a U symbol in their proper subterms.

The context-sensitive condition μ_ρ for ρ in Definition 1 and the membership condition become as follows:

- $\mu_\rho(\mathsf{u}_i^\rho) = \{1,\ldots,m_i\}$ for every u_i^ρ, and
- the membership condition is "$\in \mathcal{T}(\mathcal{F},\mathcal{V})$".

The context-sensitive condition μ_R for R is defined as $\mu_R(\mathsf{u}_i^\rho) = \mu_\rho(\mathsf{u}_i^\rho)$ (and $\mu(f) = \{1,\ldots,n\}$ for all n-ary symbols $f \in \mathcal{F}$). For $\mathbb{U}(\rho_1)$ in Example 2, the context-sensitive condition μ_{ρ_1} is specified as $\mu_{\rho_1}(\mathsf{u}_1') = \{1,2\}$ and $\mu_{\rho_1}(\mathsf{u}_3) = \{1\}$. We denote the CS-TRSs $(\mathbb{U}(\rho),\mu_\rho)$, $(\mathbb{U}(R),\mu_R)$ and $(\mathbb{U}_O(R),\mu_R)$ by $\mathbb{U}_\mu(\rho)$, $\mathbb{U}_\mu(R)$ and $\mathbb{U}_{O\mu}(R)$, respectively. We consider $\mathbb{U}_\mu$ and $\mathbb{U}_{O\mu}$ as unravelings from CTRSs to CS-TRSs.

Theorem 4 ([8]). *For every deterministic CTRS R over a signature $\mathcal{F}$, $\mathbb{U}_{O\mu}$ is simulation-complete (with respect to the membership-condition "$\in \mathcal{T}(\mathcal{F},\mathcal{V})$"), that is, for all s and $t \in \mathcal{T}(\mathcal{F},\mathcal{V})$, $s \xrightarrow{*}_R t$ if and only if $s \xrightarrow[\in \mathcal{T}(\mathcal{F},\mathcal{V})]{*}_{\mathbb{U}_{O\mu}(R)} t$.*

In the rest of this paper, we assume that the membership condition "$\in \mathcal{T}(\mathcal{F},\mathcal{V})$" is imposed on reductions.

Similarly to other unravelings, $\mathbb{U}$ is not simulation-complete for all CTRSs while $\mathbb{U}$ is $\xrightarrow{*}_R$-preserving. However, $\mathbb{U}_\mu$ is always simulation-complete for R with respect to $\xrightarrow[\in \mathcal{T}(\mathcal{F},\mathcal{V})]{}_{\mathbb{U}_\mu(R)}$.

Theorem 5. *Theorem 4 also holds for* $\mathbb{U}_\mu$.

Proof (Sketch). We only show that the CS-TRS $\mathbb{U}_\mu(R)$ is simulation-sound for R, that is, for all s and t in $\mathcal{T}(\mathcal{F},\mathcal{V})$, $s \xrightarrow[\in\mathcal{T}(\mathcal{F},\mathcal{V})]{*}_{\mathbb{U}_\mu(R)} t$ implies $s \xrightarrow{*}_R t$. This claim can be straightforwardly proved by induction on the lexicographic products of term structure and steps k of $s \xrightarrow[\in\mathcal{T}(\mathcal{F},\mathcal{V})]{k}_{\mathbb{U}_\mu(R)} t$.

Another approach to this proof is to construct the following rule from ρ in Definition 1; $\rho' : l \to r \Leftarrow \mathsf{tp}_{m_1}(s_{1,1},\ldots,s_{1,m_1}) \to \mathsf{tp}_{m_1}(t_{1,1},\ldots,t_{1,m_1}) \wedge \cdots \wedge \mathsf{tp}_{m_k}(s_{k,1},\ldots,s_{k,m_k}) \to \mathsf{tp}_{m_k}(t_{k,1},\ldots,t_{k,m_k})$ where tp_j is a fresh constructor not in $\mathcal{F}$ that represents the tuple of j terms $t_1,\ldots,t_j$. This ρ' is deterministic and satisfies that $\mathbb{U}_O(\rho') = \mathbb{U}(\rho')$ and $\mu_{\rho'}(\mathsf{u}_i^{\rho'}) = \{1\}$. Let R' be a CTRS obtained by the above transformation of the rules in R; then it is clear that $\to_R = \to_{R'}$ and $\xrightarrow[\in\mathcal{T}(\mathcal{F},\mathcal{V})]{*}_{(\mathbb{U}_\mu(R))} = \xrightarrow[\in\mathcal{T}(\mathcal{F},\mathcal{V})]{*}_{(\mathbb{U}_O(R'),\bigcup_{\rho'\in R'}\mu_{\rho'})}$ on terms in $\mathcal{T}(\mathcal{F},\mathcal{V})$. It follows from Theorem 4 that $\xrightarrow{*}_{R'} = \xrightarrow[\in\mathcal{T}(\mathcal{F},\mathcal{V})]{*}_{(\mathbb{U}_O(R'),\bigcup_{\rho'\in R'}\mu_{\rho'})}$ on $\mathcal{T}(\mathcal{F},\mathcal{V})$. Therefore, we have $\xrightarrow{*}_R = \xrightarrow[\in\mathcal{T}(\mathcal{F},\mathcal{V})]{*}_{(\mathbb{U}_\mu(R))}$ on $\mathcal{T}(\mathcal{F},\mathcal{V})$. □

The transformation in the above proof is also adequate for our purpose. However, we proposed $\mathbb{U}$ because $\mathbb{U}$ helps us to describe the transformation proposed later.

4 Reducing Context-Sensitive and Membership Conditions

In this section, we propose a transformation to relax the context-sensitive and membership condition of $(\mathbb{U}(R),\mu_R)$. In fact, the transformation reduces the number of U symbols in $\mathbb{U}(R)$. This leads to the relaxation of the condition because the condition depends on the existence of U symbols. Simply speaking, the transformation folds two rules having the same U symbol into one rule, that is, the replacement of $l_1 \to l_2\delta$ and $l_2 \to r_2$ with $l_1 \to r_2\delta$ where $\mathtt{root}(l_2)$ is a U symbol (see Fig. 2). When all U symbols are removed from $\mathbb{U}(R)$, we can obtain an unconditional system that works equally for R without the context-sensitive and membership condition. There are some cases where the context-sensitive condition is not necessary even if U symbols are still remaining.

We first give examples showing our intuitive idea of the transformation process. For an EV-TRS R, we say that a context-sensitive condition μ is *ineffective* for R if $\mu(f) = \{1,\ldots,n\}$ for all n-ary symbols f that may be a U symbol. Let us consider a conditional rewrite rule $\rho_2 : \mathsf{f}(x,y) \to z \Leftarrow \mathsf{g}(x) \to w \wedge \mathsf{f}(w,y) \to z$. This is unraveled by $\mathbb{U}_\mu$ to $(\mathbb{U}(\rho_2),\mu_{\rho_2})$ where

$$\mathbb{U}(\rho_2) = \{\mathsf{f}(x,y) \to \mathsf{u}_4(\mathsf{g}(x),y),\quad \mathsf{u}_4(w,y) \to \mathsf{u}_5(\mathsf{f}(w,y)),\quad \mathsf{u}_5(z) \to z\}$$

and $\mu_{\rho_2}(\mathsf{u}_4) = \mu_{\rho_2}(\mathsf{u}_5) = \{1\}$. The first and second rules are used in order like "$\cdots \xrightarrow[\in\mathcal{T}(\mathcal{F},\mathcal{V})]{*} \mathsf{f}(x,y)\sigma_1 \to \mathsf{u}_4(\mathsf{g}(x),y)\sigma_1 \xrightarrow[\in\mathcal{T}(\mathcal{F},\mathcal{V})]{*} \mathsf{u}_4(w,y)\sigma_2 \to \mathsf{u}_5(\mathsf{f}(w,y)\sigma_2) \xrightarrow[\in\mathcal{T}(\mathcal{F},\mathcal{V})]{*} \cdots$" where we ignore contexts over this sequence. This reduction sequence can be simulated by the rule $\mathsf{f}(x,y) \to \mathsf{u}_5(\mathsf{f}(\mathsf{g}(x),y))$ as like $\cdots \xrightarrow[\in\mathcal{T}(\mathcal{F},\mathcal{V})]{*}$

$\mathsf{f}(x,y)\sigma_1 \to \mathtt{u}_5(\mathsf{f}(\mathsf{g}(x),y)\sigma_1) \xrightarrow[\in\mathcal{T}(\mathcal{F},\mathcal{V})]{*} \mathtt{u}_5(\mathsf{f}(w,y)\sigma_2) \xrightarrow[\in\mathcal{T}(\mathcal{F},\mathcal{V})]{*} \cdots$. In a similar fashion, we also remove $\mathtt{u}_5$ as follows:

$$\{\, \mathsf{f}(x,y) \to \mathsf{f}(\mathsf{g}(x),y) \,\}.$$

The above rule has no U symbol which means the context-sensitive and membership condition is not necessary.

Let us consider the more complicated case of the rule ρ_1. This rule is unraveled to $\mathbb{U}(\rho_1)$ in Example 2 with μ_{ρ_1}. Similarly to the previous example ρ_2, the first and second rules are replaced with $\mathsf{f}(x,y) \to \mathtt{u}_3(\mathsf{h}(\mathsf{g}(x),x),\mathsf{g}(y))$. At this time, possible reductions at position 2 of $\mathtt{u}'_1(\mathsf{g}(x),\mathsf{g}(y),x)$ must be done at position 2 of $\mathtt{u}_3(\mathsf{h}(\mathsf{g}(x),x),\mathsf{g}(y))$. To allow these reductions, the context-sensitive condition μ_{ρ_1} must be updated as $\mu'_{\rho_1}(\mathtt{u}_3) = \{1,2\}$. Since we have only one U symbol $\mathtt{u}_3$, the context-sensitive condition μ'_{ρ_1} is ineffective. In this way, we reduce the number of U symbols from $\mathbb{U}(R)$, reducing and updating the context-sensitive conditions.

The transformation removing U symbols is formalized as follows:

Definition 6. *Let ρ be a deterministic conditional rewrite rule over a signature $\mathcal{F}$. We define pairs (S_i,μ_i) recursively as follows:*

1. $(S_0,\mu_0) := (\mathbb{U}(\rho),\mu_\rho)$ [2].
2. *Select a* removable *U symbol* $\mathtt{u}^\rho_j$ *from* S_i *such that* $S_i = \{l \to \mathtt{u}^\rho_j(t_1\delta,\ldots,t_m\delta),\ \mathtt{u}^\rho_j(t_1,\ldots,t_m) \to r\,\} \uplus R'$ [3] *for some substitution δ, that is,*
   - (guarding replacing positions) $t_k\delta \equiv t_k$ *for all* $k \notin \mu_i(\mathtt{u}^\rho_j)$ [4], *and*
   - (RMC) *if* $\mathtt{root}(r)$ *is a U symbol (let* $\mathtt{root}(r) = \mathtt{u}$*), then no variable in* $\mathcal{D}om(\delta)$ *is shared between terms at positions in* $\mu_i(\mathtt{u})$ *and at positions not in* $\mu_i(\mathtt{u})$ [5].

   We let $S_{i+1} := \{l \to r\delta\} \cup R'$, $\mu_{i+1}(f) := \mu_i(f)$ *for* $f \in \mathcal{F}_{\mathbb{U}(\rho)} \setminus \{\mathtt{u}^\rho_j\}$ *and*
   - (updating μ) *if* $\mathtt{root}(r)$ *is a U symbol, let* $\mathtt{root}(r) = \mathtt{u}$, *then* $\mu_{i+1}(\mathtt{u}) := \mu_i(\mathtt{u}) \cup \{\, k \mid 1 \le k \le m, r|_k \in \mathcal{D}om(\delta)\,\}$.

We denote (S_i,μ_i) by $\mathbb{T}_i(\mathbb{U}_\mu(\rho))$, and define $\mathbb{T}(\mathbb{U}_\mu(\rho)) = (S_{i'},\mu_{i'})$ where $(S_{i'},\mu_{i'}) = (S_{i'+1},\mu_{i'+1})$. For a deterministic CTRS R, we define $\mathbb{T}(\mathbb{U}_\mu(R)) = (\bigcup_{\rho\in R} R_\rho, \bigcup_{\rho\in R}\mu_\rho)$ where $\mathbb{T}(\mathbb{U}_\mu(\rho)) = (R_\rho,\mu_\rho)$. Note that $\bigcup_{\rho\in R}\mu_\rho$ is well-defined as a mapping because the domains of μ_ρs are disjoint.

The above transformation always terminates because the number of U symbols are finite and a U symbol is removed at every step, that is, i' is at most $|\rho|$.

Example 7. $\mathbb{U}_\mu(\rho_1)$ is transformed by $\mathbb{T}$ into $\mathbb{T}(\mathbb{U}_\mu(\rho_1)) = (R_1,\mu_{R_1})$ where $R_1 = \{\, \mathsf{f}(x,y) \to \mathtt{u}_3(\mathsf{h}(\mathsf{g}(x),x),\mathsf{g}(y)),\ \mathtt{u}_3(z,z) \to z \,\}$ and $\mu_{R_1}(\mathtt{u}_3) = \{1,2\}$. The membership condition is necessary for the above system because of the existence of U

[2] We write $\mu = \mu'$ if $\mu(f) = \mu'(f)$ for all f.

[3] These two rules are the only rules in S_i which contain $\mathtt{u}^\rho_j$.

[4] More precisely, $\mathcal{D}om(\delta) \subseteq (\bigcup_{k\in\mu_i(\mathtt{u}^\rho_j)} \mathcal{V}ar(t_k)) \setminus (\bigcup_{k\notin\mu_i(\mathtt{u}^\rho_j)} \mathcal{V}ar(t_k))$.

[5] That is, $\mathcal{D}om(\delta) \cap (\bigcup_{k\in\mu_i(\mathtt{u})} \mathcal{V}ar(t_k) \cap \bigcup_{k\notin\mu_i(\mathtt{u})} \mathcal{V}ar(t_k)) = \emptyset$.

symbols u_3. On the other hand, the above μ_{R_1} is ineffective for R_1. Therefore, we succeed in removing the context-sensitive condition, although the membership condition still remains.

There are non-deterministic choices for selecting U symbols at the second step in Definition 6 because there are possibly some removable U symbols. This means that the final products of $\mathbb{T}$ for $\mathbb{U}_\mu(R)$ are not unique in general. For example, consider the conditional rule $\rho_3 : \mathsf{f}(x, x') \to z \Leftarrow \mathsf{g}(x) \to y \wedge \mathsf{g}(x') \to z \wedge \mathsf{g}(y) \to w \wedge \mathsf{h}(w, z) \to z$. Here, there are two results of $\mathbb{T}(\mathbb{U}_\mu(\rho_3))$ while they become unique if the fourth condition $\mathsf{f}(w, z) \to z$ is replaced with $\mathsf{f}(w, z) \to v$. The same is said of $\mathbb{U}_O(R)$. As another example, consider the rule $\rho_4 : \mathsf{f}(x, x') \to \mathsf{h}(y, w) \Leftarrow \mathsf{g}(x) \to y \wedge \mathsf{g}(x') \to z \wedge \mathsf{h}(y, z) \to w \wedge \mathsf{g}(y) \to \mathsf{b}$. There are two results of $\mathbb{T}(\mathbb{U}_{O\mu}(\rho_4))$ and they become unique if the fourth condition is removed from ρ_4. On the other hand, $\mathbb{T}(\mathbb{U}_\mu(\rho_4))$ is unique. This means that the improvement of $\mathbb{U}_O$ in Section 3 is effective for some cases. In this way, the result of $\mathbb{T}$ is not always unique. However, it is clear that the number of all possible results is finite. Therefore, one can select the most 'favorite' in all results, for instance, one of the results whose number of rules is the least. Note that the transformation $\mathbb{T}$ does not always succeed in removing all U symbols even if we search all possible results exhaustively. To determine $\mathbb{T}(\mathbb{U}_\mu(R))$ uniquely, in this paper, we select the u^ρ_j at every step of S_i, whose index j is the greatest in all removable U symbols of ρ.

The condition Rmc in Definition 6 is necessary for preserving simulation-completeness. In other words, ignoring this condition leads to systems without simulation-completeness. For example, consider the CTRS $R_2 = \{\rho_3\} \cup R_3$ where $R_3 = \{\, \mathsf{g}(\mathsf{a}) \to \mathsf{b},\ \mathsf{g}(\mathsf{b}) \to \mathsf{c},\ \mathsf{h}(\mathsf{g}(x), \mathsf{g}(\mathsf{a})) \to \mathsf{b} \,\}$. The CTRS R_2 is unraveled by $\mathbb{U}$ and transformed by $\mathbb{T}$ into (R'_2, μ_2) where $R'_2 = R_3 \cup \{\, \mathsf{f}(x, x') \to \mathsf{u}_6(\mathsf{g}(\mathsf{g}(x)), \mathsf{g}(x')),\ \mathsf{u}_6(w, z) \to \mathsf{u}_7(\mathsf{h}(w, z), z),\ \mathsf{u}_7(z, z) \to z \,\}$ and $\mu_2(\mathsf{u}_7) = \{1\}$. Furthermore, consider the CS-TRS (R_4, μ_4) where $R_4 = R_3 \cup \{\, \mathsf{f}(x, x') \to \mathsf{u}_7(\mathsf{h}(\mathsf{g}(\mathsf{g}(x)), \mathsf{g}(x')), \mathsf{g}(x'))),\ \mathsf{u}_7(z, z) \to z \,\}$ and $\mu_4(\mathsf{u}_7) = \{1, 2\}$. The system (R_4, μ_4) is obtained by applying $\mathbb{T}$ to (R'_2, μ_2), ignoring Rmc. This system is not simulation-complete for $\mathbb{U}_\mu(R_2)$ because we have $\mathsf{f}(\mathsf{a}, \mathsf{a}) \xrightarrow{*}_{(R_4,\mu_4)} \mathsf{b}$ but not $\mathsf{f}(\mathsf{a}, \mathsf{a}) \xrightarrow{*}_{\mathbb{U}_\mu(R_2)} \mathsf{b}$. The variable z at position 2 of the term $\mathsf{u}_6(\mathsf{h}(y, z), z)$ should be used only for delivering value. For this reason, this z should not be instantiated by $\mathbb{T}$ with any term that does not finish being evaluated. This observation brings the condition Rmc to the transformation $\mathbb{T}$.

One may think that 'simplification' in completion procedures appear adequate. However, it is too powerful for folding rules and hence it does not always preserve simulation-completeness and it sometimes collapses the feature of the conditional rules that we will describe later. The reason is that applying 'simplification' ignores Rmc. Thus, 'simplification' is not adequate for our purpose.

Finally, we show correctness of $\mathbb{T}$, that is, simulation-completeness of $\mathbb{T}$.

Lemma 8. *Let ρ be a conditional rewrite rule in a deterministic CTRS R over a signature $\mathcal{F}$, and s and t be terms in $\mathcal{T}(\mathcal{F}, \mathcal{V})$. Suppose that $\mathbb{T}_i(\mathbb{U}_\mu(\rho))$*

$= (R_i, \mu_i)$, $\mathbb{T}_{i+1}((R_i, \mu_i)) = (R_{i+1}, \mu_{i+1})$ and $\mathbb{U}_\mu(R \setminus \{\rho\}) = (R', \mu')$. Then $s \xrightarrow[\in \mathcal{T}(\mathcal{F},\mathcal{V})]{*}_{(R_i \cup R', \mu_i \cup \mu')} t$ if and only if $s \xrightarrow[\in \mathcal{T}(\mathcal{F},\mathcal{V})]{*}_{(R_{i+1} \cup R', \mu_{i+1} \cup \mu')} t$.

Proof (Sketch). Since we can easily prove the case that r in Definition 6 is not rooted with a U symbol, we only consider the remaining case. Moreover, proving the *only-if* part is not difficult. Hence, we only prove the *if* part by induction on the lexicographic products of term structure and the length of the reduction sequences. To simplify this proof, we use underlines for active positions, and $\xrightarrow[\in \mathcal{T}(\mathcal{F},\mathcal{V})]{}_{(R_i \cup R', \mu_i \cup \mu')}$ and $\xrightarrow[\in \mathcal{T}(\mathcal{F},\mathcal{V})]{}_{(R_{i+1} \cup R', \mu_{i+1} \cup \mu')}$ are denoted by $\to_i$ and $\to_{i+1}$, respectively.

We can assume without loss of generality the following:

- $R_i \setminus R_{i+1} = \{\, l \to \mathfrak{u}_j^\rho(\underline{f(u, u, u', y)}, z),\ \mathfrak{u}_j^\rho(\underline{f(x, x, x', y)}, z) \to \mathfrak{u}(\underline{s'}, x, y, z) \,\}$,
- $R_{i+1} \setminus R_i = \{\, l \to \mathfrak{u}(\underline{s'\delta, x\delta}, y, z) \,\}$,
- $\mathfrak{u}(\underline{t'}, x, y, z) \to r' \in R_i$ and $\mathfrak{u}(\underline{t', x}, y, z) \to r' \in R_{i+1}$.

where $\delta = \{x \mapsto u, x' \mapsto u'\}$, $\mu_i(\mathfrak{u}_j^\rho) = \mu_i(\mathfrak{u}) = \{1\}$ and $\mu_{i+1}(\mathfrak{u}) = \{1, 2\}$. It follows from Rmc that $x \notin \mathcal{V}ar(s')$. We only show the most difficult case. Suppose that $s \xrightarrow{*}_{i+1} l\sigma_1 \to_{i+1} \mathfrak{u}(\underline{s'\delta, x\delta}, y, z)\sigma_1 \xrightarrow{*}_{i+1} \mathfrak{u}(\underline{t', x}, y, z)\sigma_2 \to_{i+1} r'\sigma_2 \xrightarrow{*}_{i+1} t$ where $\mathcal{R}an(\sigma_1) \cup \mathcal{R}an(\sigma_2) \subseteq \mathcal{T}(\mathcal{F}, \mathcal{V})$. Then, it follows from the context-sensitive condition that $y\sigma_1 \equiv y\sigma_2$ and $z\sigma_1 \equiv z\sigma_2$. By the induction hypothesis, we have $s \xrightarrow{*}_i l\sigma_1$, $s'\delta\sigma_1 \xrightarrow{*}_i t'\sigma_2$, $x\delta\sigma_1 \xrightarrow{*}_i x\sigma_2$, and $r'\sigma_2 \xrightarrow{*}_i t$. It follows from $x \notin \mathcal{V}ar(s')$ that $s'\delta\sigma_1 \equiv s'\sigma_1$. Let $\theta = \{x \mapsto x\sigma_2, x' \mapsto u'\sigma_1, y \mapsto y\sigma_2, z \mapsto z\sigma_2\}$. Therefore, we have $s \xrightarrow{*}_i l\sigma_1 \to_i \mathfrak{u}_j^\rho(\underline{f(u, u, u', y)}, z)\sigma_1 \xrightarrow{*}_i \mathfrak{u}_j^\rho(\underline{f(x\sigma_2, x\sigma_2, u'\sigma_1, y\sigma_1)}, z\sigma_1)$ $\equiv \mathfrak{u}_j^\rho(\underline{f(x, x, x', y)}, z)\theta \to_i \mathfrak{u}(\underline{s'}, x, y, z)\theta \equiv \mathfrak{u}(\underline{s'\sigma_1}, x\sigma_2, y\sigma_2, z\sigma_2) \xrightarrow{*}_i \mathfrak{u}(\underline{t'\sigma_2}, x\sigma_2, y\sigma_2, z\sigma_2) \to_i r'\sigma_2 \xrightarrow{*}_i t$. □

Theorem 9. *Let R be a deterministic CTRS over a signature $\mathcal{F}$. For all $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$, $s \xrightarrow[\in \mathcal{T}(\mathcal{F},\mathcal{V})]{*}_{\mathbb{U}_\mu(R)} t$ if and only if $s \xrightarrow[\in \mathcal{T}(\mathcal{F},\mathcal{V})]{*}_{\mathbb{T}(\mathbb{U}_\mu(R))} t$.*

From Lemma 8 and Theorems 9 and 5, the composition $\mathbb{T}(\mathbb{U}_\mu(\cdot))$ of the transformations can be considered as an unraveling with simulation-completeness.

Corollary 10. *Theorem 4 also holds for $\mathbb{T}(\mathbb{U}_\mu(\cdot))$.*

5 On Confluence of CTRSs

To prove confluence of CTRSs, simulation-completeness of the unravelings enable us to use confluence of the unraveled CTRSs.

Theorem 11. *Let R be a deterministic CTRS over a signature $\mathcal{F}$. If $\mathbb{U}(R)$ is confluent, then R is confluent.*

On the other hand, confluence of CTRSs is not preserved by unravelings, that is, the converse of Proposition 11 does not always hold in general. Consider a normal form of a confluent CTRS over a signature, which are matched with the lhs of a conditional rule with at least a condition. The normal form sometimes becomes reducible on the unraveled CTRS to determine whether the original conditional part is satisfied, although the conditional part is not satisfied. The normal form is not reachable to any terms over the original signature, and hence it is reduced to a normal form containing a U symbol. Thus, we can see that terms containing U symbols prevent the unravelings from preserving confluence of CTRSs. For this observation, as far as terms without U symbols are concerned, confluence of CTRSs are preserved by the unravelings if simulation-completeness is preserved. The unraveling $\mathbb{U}_\mu$ and the transformation $\mathbb{T}$ preserve simulation-completeness. Moreover, $\mathbb{T}$ sometimes remove all U symbols. In such cases, confluence of the systems obtained by $\mathbb{T}(\mathbb{U}_\mu(\cdot))$ coincides with that of the original CTRSs.

Corollary 12. *A deterministic CTRS R over a signature $\mathcal{F}$ is confluent if and only if $\mathbb{U}_\mu(R)$ (respectively $\mathbb{T}(\mathbb{U}_\mu(R))$) is confluent on $\mathcal{T}(\mathcal{F},\mathcal{V})$* [6]*. Especially, let $(R',\mu') = \mathbb{T}(\mathbb{U}_\mu(R)))$ and suppose that R' has no U symbol, then $\xrightarrow{*}_R = \xrightarrow{*}_{R'}$ (more precisely, $\rightarrow_{R'} \subseteq \rightarrow_R \subseteq \xrightarrow{+}_{R'}$), that is, R is confluent if and only if R' is.*

As long as we know, there are no methods to show confluence of $\mathbb{U}_\mu(R)$ and $\mathbb{T}(\mathbb{U}_\mu(R))$ on $\mathcal{T}(\mathcal{F},\mathcal{V})$ if U symbols still remain. However, to decide confluence of R, we can use ordinary techniques for deciding confluence of $\mathbb{T}(\mathbb{U}(\mu(R)))$ if $\mathbb{T}$ removes all U symbols.

The method in this paper appears to counter the other approaches to confluence, such as Bergstra and Klop's method [3]. In fact, the unraveled CTRSs often lose confluence of the original CTRSs as described above. However, the transformation $\mathbb{T}$ recovers the confluence that is lost in the process of the unravelings if all U symbols are removed successfully. Therefore, the transformation $\mathbb{T}$ is sometimes effective for preserving confluence of CTRSs.

6 Refinement of the Condition for Removing U Symbols

It is probably impossible to relax the condition Rmc in Definition 6. To the contrary, we should tighten Rmc for maintaining a feature of conditional rules associated with efficiency of reductions. Consider the following 'ML' program.

```
fun twofib 0 = (0,1)
  | twofib n = let val m = twofib (n-1)
                 in (#2 m, (#2 m) + (#1 m) ) end;
```

It is known that the function `twofib` efficiently computes pairs of two continuous Fibonacci numbers. Such efficiency comes from the 'let' structure, and the first part of the 'let' structure can be considered as a conditional part. From this observation, the above program is regarded as the following CTRS:

$$R_5 = \begin{cases} \mathsf{twofib}(0) \rightarrow \mathsf{tp}_2(0,\mathsf{s}(0)), \\ \mathsf{twofib}(\mathsf{s}(n)) \rightarrow \mathsf{tp}_2(\#2(m), \mathsf{add}(\#2(m), \#1(m))) \Leftarrow \mathsf{twofib}(n) \rightarrow m, \\ \quad\vdots \end{cases}$$

[6] For all $s, t_1, t_2 \in \mathcal{T}(\mathcal{F},\mathcal{V})$, if $s \xrightarrow{*}_{(\mathbb{U}(R),\mu_R)} t_1$ and $s \xrightarrow{*}_{(\mathbb{U}(R),\mu_R)} t_2$ then there exists a term $u \in \mathcal{T}(\mathcal{F},\mathcal{V})$ such that $t_1 \xrightarrow{*}_{(\mathbb{U}(R),\mu_R)} u$ and $t_2 \xrightarrow{*}_{(\mathbb{U}(R),\mu_R)} u$.

where $\mathsf{tp}_2(t_1, t_2)$ denotes the pair of two terms t_1 and t_2. The second rule is unraveled into the system (R_6, μ_{R_6}) where

$$R_6 = \{\mathsf{twofib}(\mathsf{s}(n)) \to \mathsf{u}_8(\mathsf{twofib}(n)),\ \mathsf{u}_8(m) \to \mathsf{tp}_2(\mathsf{\#2}(m), \mathsf{add}(\mathsf{\#2}(m), \mathsf{\#1}(m)))\}$$

and $\mu_{R_6}(\mathsf{u}_8) = \{1\}$. Under innermost reduction strategy, efficiency is still alive in (R_6, μ_{R_6}). The system (R_6, μ_{R_6}) can be transformed by $\mathbb{T}$ as follows:

$$\mathsf{twofib}(\mathsf{s}(n)) \to \mathsf{tp}_2(\mathsf{\#2}(\mathsf{twofib}(n)), \mathsf{add}(\mathsf{\#2}(\mathsf{twofib}(n)), \mathsf{\#1}(\mathsf{twofib}(n)))).$$

$\mathbb{T}$ succeeded in removing all U symbols from (R_6, μ_{R_5}). This corresponds to the following 'ML' program.

```
fun twofib2 0 = (0,1)
  | twofib2 n = ( (#1 (twofib2 (n-1))),
                  (#2 (twofib2 (n-1)))+(#1 (twofib2 (n-1))) );
```

However, the above 'ML' program loses efficiency.

The 'let' structure provides a facility that separates the parallel evaluations of terms that are identical into one. For example, `twofib2 (n-1)` is evaluated once in the first 'ML' program and three times in the second 'ML' program. The advantage of coming from the 'let' structure is lost in the transformation $\mathbb{T}$, by instantiating variable m in $\mathsf{tp}_2(\mathsf{\#2}(m), \mathsf{add}(\mathsf{\#2}(m), \mathsf{\#1}(m)))$, whose occurrence is non-linear, with $\mathsf{twofib}(n)$. In order to prevent such instantiation in these cases, we enhance the condition Rmc as follows:

(Rmc$'$) *r is linear with respect to $\mathcal{D}om(\delta)$.*

It is clear that Rmc$'$ implies Rmc. The enhanced condition Rmc$'$ does not cause the target systems to lose the essential advantage of the original CTRSs, such as efficiency that comes from 'let' structure. For confluent CTRSs, simulation-completeness holds without Rmc. However, Rmc$'$ should not be ignored because of the points outlined in the above discussion.

7 Concluding Remarks and Related Works

We firstly show an application of our method. Consider the following rule obtained by the inversion compiler [8], from the TRS that computes multiplication:

$$\rho_{\mathsf{div}} : \mathsf{div}(\mathsf{s}(z), \mathsf{s}(y)) \to \mathsf{tp}_1(\mathsf{s}(x)) \Leftarrow \mathsf{sub}(z, y) \to \mathsf{tp}_1(w) \wedge \mathsf{div}(w, \mathsf{s}(y)) \to \mathsf{tp}_1(x),$$

where div and sub compute division and subtraction of natural numbers, respectively, and $\mathsf{tp}_i(t_1, \ldots, t_i)$ denotes the tuple of i terms $t_1, \ldots, t_i$. Since we can consider $\mathsf{tp}_1(t)$ as t similarly to several functional languages, we can easily see that the following rule seems to be similar to the above rule in the sense of computing division [7]:

$$\rho'_{\mathsf{div}} : \mathsf{div}(\mathsf{s}(z), \mathsf{s}(y)) \to \mathsf{s}(x) \Leftarrow \mathsf{sub}(z, y) \to w \wedge \mathsf{div}(w, \mathsf{s}(y)) \to x.$$

[7] Note that $\mathsf{tp}_1(t)$ cannot be abbreviated to t in all cases.

This rule is transformed by $\mathbb{T}(\mathbb{U}_\mu(\cdot))$ into the following rule:

$$\mathsf{div}(\mathsf{s}(z), \mathsf{s}(y)) \to \mathsf{s}(\mathsf{div}(\mathsf{sub}(z, y), y)).$$

Using $\mathbb{T}$, we succeeded in removing all U symbols from $\mathbb{U}(\rho'_{\mathsf{div}})$, and the above rule coincides with the typical rewrite rule of division $\mathsf{s}(x) \div \mathsf{s}(y) \to \mathsf{s}((x - y) \div \mathsf{s}(y))$. This tells us that the program generated by the compiler seems to be correct, in comparison with the handmade program.

Finally, we briefly offer some extra remarks.

– Two syntactic conditions to preserve simulation-completeness without the context-sensitive and membership condition [7] also hold for $\mathbb{U}$ and $\mathbb{T}(\mathbb{U}(\cdot))$. Neither of the two syntactic conditions are sufficient and necessary condition for removing all U symbols successfully.
– 'Effective termination' of CTRSs is preserved by $\mathbb{U}_\mu$ and $\mathbb{T}$. Thus, termination of $\mathbb{T}(\mathbb{U}_\mu(R))$ guarantees 'effective termination' of R. When $\mathbb{T}(\mathbb{U}_\mu(R))$ has no U symbols, termination of $\mathbb{T}(\mathbb{U}_\mu(R))$ coincides with 'effective termination' of R. Therefore, several methods of proving termination of TRSs are applicable for proving 'effective termination' of R.
– Given a conditional rule, the recursive reduction of the conditional part that is not terminating sometimes become terminating. Consider the CTRS $R_7 = \{\ \mathsf{f}(x, y) \to z \Leftarrow \mathsf{g}(x) \to z,\ \mathsf{a} \to \mathsf{g}(\mathsf{a})\ \}$. This CTRS R_7 is transformed by $\mathbb{T}(\mathbb{U}_\mu(\cdot))$ into $R'_7 = \{\ \mathsf{f}(x, y) \to \mathsf{g}(x),\ \mathsf{a} \to \mathsf{g}(\mathsf{a})\ \}$. When $\mathsf{f}(x, y) \to z \Leftarrow \mathsf{g}(x) \to z$ is applied to $\mathsf{f}(\mathsf{a}, \mathsf{a})$, the recursive reduction of the instantiated condition $\mathsf{g}(\mathsf{a})$ does not terminate. On the other hand, in the case of applying $\mathsf{f}(x, y) \to \mathsf{g}(x)$, the conditional part is no longer concerned, that is, the reduction of the condition does terminate.
– It is clear that all CS-TRSs in the process of $\mathbb{T}$ can be considered as the unraveled systems for some CTRSs. For example, R_1 corresponds to the conditional rule $\mathsf{f}(x, y) \to z \Leftarrow \mathsf{g}(y) \to z \wedge \mathsf{h}(\mathsf{g}(x), x) \to z$.

As another approach to CTRSs, Viry proposed the transformation of normal or join CTRSs into TRSs [14]. Unlike unravelings, his transformation does not introduce U symbols but extends the arity of defined symbols. Similarly to unravelings, his transformation is not simulation-complete for all CTRSs. The example in [6] is also a counterexample against simulation-completeness of his transformation. Antoy, Brassel, and Hanus applied Viry's transformation to conditional narrowing of constructor-based CTRSs that are restricted normal CTRSs. [1]. Rosu proposed the transformation of join CTRSs for implementing an efficient conditional rewriting engine [12]. His transformation seems to produce unconditional systems that are simulation-complete. However, the main part to evaluate conditional parts is not defined by rewrite rules but implemented. Thus, his transformation is not suitable for analyzing ultra-properties of CTRSs. Moreover, neither of Viry's and Rosu's transformations are applicable to deterministic 3-CTRSs.

Acknowledgement. This work is partly supported by MEXT. KAKENHI #17700009 and #18500011. We thank Professor Yoshihito Toyama for pointing out some interesting issues about confluence.

References

1. Antoy, S., Brassel, B., Hanus, M.: Conditional narrowing without conditions. In: Proceedings of the 5th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP'03), ACM (2003) 20–31
2. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press (1998)
3. Bergstra, J.A., Klop, J.W.: Conditional rewrite rules: Confluence and termination. Journal of Computer and System Sciences **32** (1986) 323–362
4. Durán, F., Lucas, S., Meseguer, J., Marché, C., Urbain, X.: Proving termination of membership equational programs. In Heintze, N., Sestoft, P., eds.: Proceedings of the 2004 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation (PEPM), ACM (2004) 147–158
5. Lucas, S.: Context-sensitive computations in functional and functional logic programs. Journal of Functional and Logic Programming **1998** (1998)
6. Marchiori, M.: Unravelings and ultra-properties. In: Proceedings of the 5th International Conference on Algebraic and Logic Programming (ALP'96). Volume 1139 of Lecture Notes in Computer Science, Springer (1996) 107–121
7. Nishida, N., Sakai, M., Sakabe, T.: On simulation-completeness of unraveling for conditional term rewriting systems. IEICE Technical Report SS2004-18, the Institute of Electronics, Information and Communication Engineers (IEICE) (2004) Vol. 104, No. 243, pp. 25–30.
8. Nishida, N., Sakai, M., Sakabe, T.: Partial inversion of constructor term rewriting systems. In: Proceedings of the 16th International Conference on Rewriting Techniques and Applications. Volume 3467 of Lecture Notes in Computer Science, Springer (2005) 264–278
9. Nishida, N., Sakai, M., Sakabe, T.: Generation of inverse computation programs of constructor term rewriting systems. The IEICE Transactions on Information and Systems **J88-D-I** (2005) 1171–1183 (in Japanese).
10. Ohlebusch, E.: Termination of logic programs: Transformational methods revisited. Applicable Algebra in Engineering, Communication and Computing **12** (2001) 73–116
11. Ohlebusch, E.: Advanced Topics in Term Rewriting. Springer (2002)
12. Rosu, G.: From conditional to unconditional rewriting. In: Revised selected papers of the 17th International Workshop on Recent Trends in Algebraic Development Techniques (WADT 2004). Volume 3423 of Lecture Notes in Computer Science, Springer (2005) 218–233
13. Toyama, Y.: Confluent term rewriting systems with membership conditions. In: Proceedings of the 1st International Workshop on Conditional Term Rewriting Systems (CTRS). Volume 308 of Lecture Notes in Computer Science, Springer (1987) 228–241
14. Viry, P.: Elimination of conditions. Journal of Symbolic Computation **28**(3) (1999) 381–401

An Account of Implementing Applicative Term Rewriting

Muck van Weerdenburg[⋆]

Eindhoven University of Technology[⋆⋆]
Department of Mathematics and Computer Science

Abstract. Generation of labelled transition systems from system specifications is highly dependent on efficient rewriting (or related techniques). We give an account of the implementation of two rewriters of the mCRL2 toolset and evaluate them by comparing them with other commonly used efficient rewriters.

1 Introduction

The mCRL2 language and toolset [7] support modelling and verification of systems. Verification mainly consists of model checking, which means that a Labelled Transition System (LTS) is generated from a system specification and requirements are checked on this LTS. However, the task of generating LTSs is very time and space demanding. Generation of typical state-spaces of, say, 10^7 transitions requires at least a few times more than 10^7 calls to the rewriter. In fact, inspection of this process in the mCRL2 toolset shows that more than 90% of the time of generating an LTS is spent rewriting.

Apart from on-the-fly LTS reductions, there are two clear paths towards optimisation of LTS generation. One is to reduce the number of times the LTS generator uses the rewriter. The other, which we consider here, is to optimise the rewriting procedure.

We discuss two implementations we have made for the mCRL2 toolset. One uses innermost rewriting, the other JITty [15]. The latter is a strategy close to lazy rewriting (i.e. rewriting (sub)terms only when needed). Essential is that these rewriters are *compiling* rewriters, meaning that they generate a specialised rewriter for a given specification. Also, they support rewriting of *open* terms (i.e. terms in which (free) variables may occur), which is required for LTS generation.

As mCRL2 has a higher-order data language, rewriting is on higher-order (applicative) terms. Due to the fact that higher-order matching is at least as hard as NP-complete problems [2], we restrict the rewriting to using only simple syntactic pattern matching. This basically boils down to rewriting applicative terms without being able to do η-reductions. It seems that this restriction does

[⋆] E-Mail: M.J.van.Weerdenburg@tue.nl
[⋆⋆] Address: P.O. Box 513, 5600 MB Eindhoven, The Netherlands

not really restrict the practical use of the rewriter (at least not with case studies, such as [10], so far), but the precise implications should be subject to future research. But even if our choice is too restricted for general purpose, a limited, but *fast* rewriter is still very useful for a large set of problems.

In order to implement efficient matching we use an adaptation of existing algorithms that, instead of matching each rule separately, combine sets of rules into a tree structure that allows to match these rules simultaneously. Although implementations of such algorithms often require left-hand sides to contain each variable at most once (i.e. the left-hand sides must be *linear*), our implementation does not have this restriction.

Another important optimisation is to avoid rewriting normal forms multiple times. Although this is fairly easy with innermost rewriting, it is much more involved in the JITty rewriter.

In short, we have implemented a compiling JITty rewriter for conditional rewrite rules on open applicative term, making use of efficient matching of non-linear applicative terms, which is the first of its kind (as far as we know).

We first introduce the part of the mCRL2 data language that is relevant for rewriting and the general architecture of our implementations in Sect. 2. In Sect. 3 we discuss the matching algorithm used and Sect. 4 and 5 contain the descriptions of the innermost and JITty rewriters, respectively. We conclude with an analysis of some benchmarks in Sect. 6.

2 Preliminaries

The data language we consider here is the core data language of mCRL2. It has only one operator, viz. application. The complete data language contains many additional constructs for ease of modelling (including λs), but they are all expressible in this core. From this point on, we will refer to this core simply as the mCRL2 language (or even just mCRL2).

The **signature** (Σ) of mCRL2 consist of a set of **basic sorts** $\mathbb{S}_B$, a set of **variables** $\mathbb{V}$ and a set of **function symbols** $\mathbb{F}$. Each variable or function symbol has a **sort**. Sorts s are defined as follows, where $b \in \mathbb{S}_B$ and $\to$ is right-associative:

$$s \ := \ b \ | \ s \to s$$

With $x_s \in \mathbb{V}$ a variable of sort s and $f_s \in \mathbb{F}$ a function symbol of sort s, the definition of mCRL2 **terms** t_s of sort s is as follows:

$$t_s \ := \ x_s \ | \ f_s \ | \ t_{s' \to s}(t_{s'})$$

Typical basic sorts are the booleans $\mathbb{B}$ or the integers $\mathbb{Z}$. Function symbols are, for example, *true* or *even*. The sorts as subscripts of terms are usually omitted. Given a term $f(t_1)\ldots(t_n)$ we call f the **head symbol** and t_i the ith **argument**. The **arity** of a function symbol is the maximal number of arguments it can have.

For readability we usually write terms with sequences of **applications** (i.e. terms $t(u)$) such as $((f(w))((g(x))(y)))(z)$ simply as $f(w, g(x, y), z)$.

Rewrite rules are of the form $t \to u$ **if** c, where terms t and u have the same sort. Term c of sort $\mathbb{B}$ is the condition of a rewrite rule indicating whether or not the rule may be applied (i.e. only when c rewrites to *true*). Often we omit this condition in the case c is (syntactically) equal to *true*.

We write $(\Sigma, \to)$ for a signature Σ and set of rewrite rules $\to$ to denote a Term Rewrite System (**TRS**).

The architecture of the rewriters is as follows. The rewriters first preprocess the TRS by sorting the rules by head symbol. For each head symbol f and number of arguments n that f can have, we create a specialised function $\mathit{rewr}_f(t_1, \ldots, t_n)$ that returns a normal form of the term $f(t_1)\ldots(t_n)$. The code of a function rewr_f is the implementation of the match tree(s) generated for the set of rules of f. Also a main rewrite function is added that takes a single term t and calls the specialised function for the head symbol of t. Depending on the strategy it also rewrites the arguments of a function symbol, before calling its specialised function.

For reasons of efficiency we use **implicit substitutions**. This means that, instead of first substituting specific values for variables and then rewriting the term, we apply substitution on-the-fly during rewriting (i.e. we rewrite in a *context* of substitutions). This basically boils down to replacing a variable with its value as soon as it is encountered. We can, however, also encounter terms of the form $x(t_1, \ldots, t_n)$. In the case that x is not bound to a value we can just ignore it and rewrite its arguments. Otherwise, we need to get the value of x, append the arguments $t_1, \ldots, t_n$ and then rewrite that term.

For the implementation of the data terms we use the ATerm [19] library. This automatically gives us term sharing[1] and constant time equality tests. Construction of terms, however, is more expensive.

3 Match trees

Straightforward implementations of rewrite systems will try to match the term to be rewritten with every left-hand side of a rewrite rule separately. For example, with the system $\{t_1 \to u_1, t_2 \to u_2\}$ one could first try to match a term with t_1 and afterwards, if it did not match, with t_2 (or vice versa).

That this is not a very efficient manner of matching can be seen clearly by looking at rules for equality functions. Assuming a sort S with n simple constructors (i.e. without arguments), the equality on S needs n^2 rules (for every pair in $S \times S$).[2] However, by combining these rules into a specific tree structure, we can test for a match in the order of n.

[1] That is, equal (sub)terms are only stored once in memory. Note that changing a term in one place will not automatically change (equal) terms in other places.

[2] Note that many languages allow for more compact notations by assuming an order on rules. Such features are in general not safe when rewriting with open terms (e.g.

This method is similar to the ones used in the ASF+SDF [17] rewriters [18] and ELAN [20]. For rules with linear left-hand sides (i.e. left-hand sides in which variables occur at most once), algorithms to create such trees can be found in [12, 1, 14]. As we have applicative terms and allow nonlinear rewrite rules, our approach deviates a bit. Note that in ASF+SDF nonlinear rules are also allowed, but converted to linear rules, which requires additional side conditions.

Match trees determine the way a term is matched; each node of a tree represents a basic instruction and guides the path through the tree. We start at the root and walk up the tree, choosing branches based on the result of matching so far. For example, one node could be to check whether a (sub)term has a specific head symbol. Matching continues with one branch if the symbol was found and the other branch otherwise.

The way a term is traversed during matching is as follows. Matching a term $f(t_1, \ldots, t_k)$ according to a match tree m starts with argument t_1 of f and executing the specific functionality of m. We do not have to match f itself as we make a specialised rewrite function that handles only terms starting with f (for each symbol f). At any point during execution of the matching algorithm there is a context of values bound to variables (i.e. a context of substitutions) and a stack of terms to be matched. Initially the context is empty and the stack consist of the arguments t_1 to t_k of f (with t_1 on top). The matching algorithm always considers the top of the stack, which we refer to as $g(u_1, \ldots, u_l)$. During matching the context will be built up, resulting in a substitution that makes the left-hand side of the matching rule equal to $f(t_1, \ldots, t_k)$.

Our **match trees** m have the following structure, with $x \in \mathbb{V}$, $f \in \mathbb{F}$ and term t.

$$m ::= S(x, m) \mid M(x, m, m) \mid F(f, m, m) \mid N(m) \mid C(t, m, m) \mid R(t) \mid X$$

We give an intuition of functionality of the trees before giving the actual matching function. A $S(x, m)$ binds the top of the stack to variable x and continues with tree m. Such a value bound to x is tested for equality with the top of the stack with $M(x, m, n)$, which continues with tree m on equality and n otherwise. With $F(f, m, n)$ matching continues with $u_1, \ldots, u_l$ on top of the stack and tree m if f is equal to g. If not, tree n is used without changing the stack. Node $N(m)$ removes the top of the stack and continues with m. A condition b can be checked with $C(b, m, n)$. A successful match is indicated by $R(t)$, where t is the result of applying a matching rule. Unsuccessful matches occur with X and when the stack is empty (i.e. there are too few arguments).

Let σ be a context, $\sigma[x \mapsto t]$ the context σ in which term t is bound to variable x and $\sigma(t)$ a term t in which every variable is replaced by the value bound to it in σ. Also let $[]$ denote the empty stack and $t \rhd s$ term t on top of stack s. The definition of the **matching function** μ, which returns either X (no match) or $R(t)$ (match with result t), is as follows:

rewriting $f(x)$ does not terminate in the system $f(0) \to e$; $f(n) \to g(f(n-1))$). In mCRL2 we use standard conditional rewriting.

$$\begin{array}{llll}
\mu(m, & \sigma, []) & = X & \\
\mu(S(x,m), & \sigma, t \rhd s) & = \mu(m, \sigma[x \mapsto t], t \rhd s) & \\
\mu(M(x,m,n), & \sigma, t \rhd s) & = \mu(m, \sigma, t \rhd s) & \text{if } \sigma(x) = t \\
\mu(M(x,m,n), & \sigma, t \rhd s) & = \mu(n, \sigma, t \rhd s) & \text{if } \sigma(x) \neq t \\
\mu(F(f,m,n), & \sigma, g(u_1, \ldots, u_n) \rhd s) & = \mu(m, \sigma, u_1 \rhd \ldots \rhd u_n \rhd s) & \text{if } f = g \\
\mu(F(f,m,n), & \sigma, g(u_1, \ldots, u_n) \rhd s) & = \mu(n, \sigma, g(u_1, \ldots, u_n) \rhd s) & \text{if } f \neq g \\
\mu(N(m), & \sigma, t \rhd s) & = \mu(m, \sigma, s) & \\
\mu(C(b,m,n), & \sigma, t \rhd s) & = \mu(m, \sigma, t \rhd s) & \text{if } \sigma(b) \\
\mu(C(b,m,n), & \sigma, t \rhd s) & = \mu(n, \sigma, t \rhd s) & \text{if } \neg\sigma(b) \\
\mu(R(r), & \sigma, t \rhd s) & = R(\sigma(r)) &
\end{array}$$

To illustrate the use of the match trees and give some intuition on how we build such trees, we consider the rewrite rules $f(g(x),x) \to x$ and $f(x,x) \to c$ if $h(x)$. In Fig. 1 the match tree for the first rule is shown. We can see that the root node (on the far left) checks whether the head symbol of the first argument is a g or not. If this is the case, it binds the argument of g to x and proceeds to the next argument. As g has only one argument, this means we look at the next argument of the enclosing function f. The M node checks to see if this argument is the same as the value of x and returns the result (also x) if this is the case. Note that the head symbol f is not in the tree at all as we make trees for rules with the same head symbol.

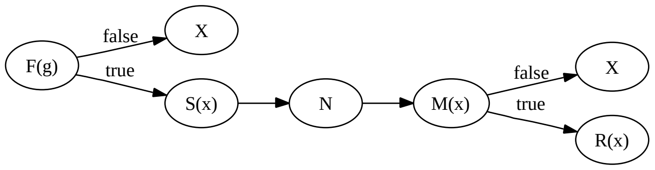


Fig. 1. Match tree for $f(g(x),x) \to x$

The tree for the conditional rule is shown in Fig. 2. Here we see that the first argument is stored and the second argument is matched with the first argument. If they are the same, the condition $h(x)$ is checked, using the value bound to x, before returning the result c.

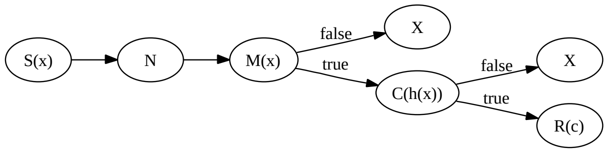


Fig. 2. Match tree for $f(x,x) \to c$ if $h(x)$

Finally, we combine both trees to the complete match tree for function symbol f, as shown in Fig. 3.

Such a combination is made by interleaving the trees and synchronising on N nodes. The following rules give a simplified version of our algorithm to compute

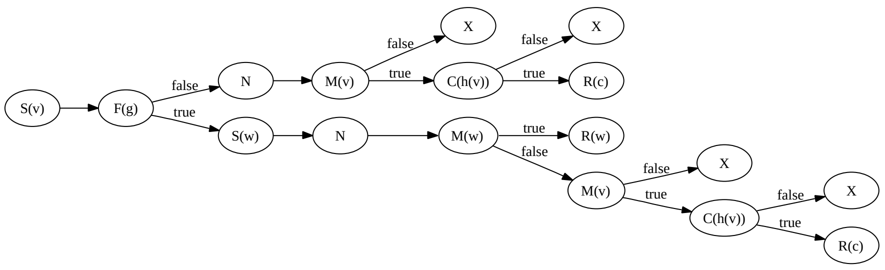


Fig. 3. Combined match tree for f

$comb(T)$, the combination of the trees in T. If more than one rule can be applied, the one that occurs first in the list below is applied. We write T for a set of trees, which we can partition in T_f and $T \backslash T_f$, of which the former contains all F nodes that check for symbol f (and only those nodes). Projection functions π_1 and π_2 are used to filter a set of $F(f, m, n)$ nodes to the m, respectively n values. We write $N_f(T)$ for T with N nodes added to every tree in it; the amount of added nodes corresponds to the number of arguments f has (in the pattern). The substitution of a variable x by y in tree m is denoted by $m[y/x]$. With x' we indicate a fresh variable (i.e. one not occurring in any of the trees).

$$\begin{array}{ll}
comb(\{R(t)\} \cup T) & \to R(t) \\
comb(\{C(t,m,n)\} \cup T) & \to C(t, comb(\{m\} \cup T), comb(\{n\} \cup T)) \\
comb(\{M(x,m,n)\} \cup T) & \to M(x, comb(\{m\} \cup T), comb(\{n\} \cup T)) \\
comb(\{S(x,m)\} \cup T) & \to S(x', comb(\{m[x'/x]\} \cup T)) \\
comb(\{F(f,m,n)\} \cup T) & \to F(f,\ comb(\pi_1(T_f) \cup N_f(T \setminus T_f)), \\
 & \qquad comb(\pi_2(T_f) \cup (T \setminus T_f))\ ) \\
comb(\{N(m_0), \ldots, N(m_k)\}) & \to N(comb(\{m_0, \ldots, m_k\})) \\
comb(\{X\} \cup T) & \to comb(T) \\
comb(\emptyset) & \to X
\end{array}$$

The first rule indicates that as soon as there is a tree indicating a positive match, we can just return that match and ignore the other trees. In the rule for S we introduce a fresh variable to avoid conflicts with variables in other trees. When applying the F rule for a symbol f, we consider all trees that have such a root node. This is done as the first subtree of an F processes arguments of the matched function symbol and this can only be done once (due to the matching function). Also, during matching of the arguments (of the subterm), the other trees that do not participate need to be ignored until f and its arguments are completely matched. For this reason we add the necessary N nodes to these trees.

There are several optimisation to the above. For example, between two N nodes, we can ensure that matching a variable occurs only once and we can combine all S nodes into one, as they all store the same term. In case both subtrees of an M or C node are the same, we can replace it with the subtree itself. Also, S nodes that bind a value to a variable that is never used in the subtree can be replaced by the subtree.

4 Compiling Innermost Rewriter

The implementation of the innermost rewriter is very similar to that of the μCRL toolset [3] and ASF+SDF. We discuss the main points. To achieve optimal performance, compilation of a specific rewrite system is essential. This is done as described in Sect. 2. The main rewrite function would be of the following form (not considering implicit substitutions and variables as head symbols):

```
function $innermost(f(t_1, \ldots, t_n))$
    for $i \in \{1, \ldots, n\}$ do
        $t_i := innermost(t_i)$
    return $rewr_f(t_1, \ldots, t_n)$
```

A specialised function for a function symbol f uses the match tree for f to see if any rule can be applied. If this is the case, the right-hand side of that rule is built and the generic rewrite function is called on this term. If no rule matches, then the original term is built and returned. An example of the code that would be generated of a function with rewrite rule $f(c, x) = g(h(x), x)$ is as follows.

```
function $rewr_f(arg_1, arg_2)$
    if $arg_1 = c$ then
        return $rewrite(g(h(arg_2), arg_2))$
    else
        return $f(arg_1, arg_2)$
```

One important optimisation is that of avoiding needless traversal of *normal forms*. The main observation here is that one can assume that the arguments of a specific rewrite function are already in normal form. This is the case when called from the main rewrite function, as it first explicitly rewrites these arguments, and also needs to be the case when called from a specific rewrite function.

We achieve this optimisation by taking the instantiations of the variables of the matching rewrite rule, which are in normal form by definition, and building up the term around it with the appropriate specific rewrite functions. For example, if we need to build a term $g(h(x), x)$, we call the specific rewrite function of h on the instantiation of x, returning the normal form of $h(x)$, and then call the specific rewrite function of g with the previous result and the instantiation of x. The rewrite function for f then becomes as follows:

```
function $rewr_f(arg_1, arg_2)$
    if $arg_1 = c$ then
        $tmp := rewr_h(arg_2)$
        return $rewr_g(tmp, arg_2)$
    else
        return $f(arg_1, arg_2)$
```

In our case we also have to consider applicative terms. This means that a function of arity n has at most n arguments (instead of exactly n). This is solved by generating specific rewrite functions for each function symbol and number of arguments allowed. So, for f we would have two additional rewrite functions (i.e. one for one argument and another for no arguments at all).

5 Compiling JITty Rewriter

When rewriting a term $f(t_1, \ldots, t_n)$ the JITty strategy delays rewriting of arguments t_i as long as they are not needed for matching. By doing so, it avoids rewriting terms that can be removed without ever being used. A typical example is the *if*, which often has the following rules:

$$\begin{aligned} \alpha &: \mathit{if}(\mathit{true}, x, y) \rightarrow x \\ \beta &: \mathit{if}(\mathit{false}, x, y) \rightarrow y \\ \gamma &: \mathit{if}(b, x, x) \quad\;\; \rightarrow x \end{aligned}$$

Instead of rewriting all arguments first and then matching these rules, like innermost rewriting does, JITty uses a strategy to, for example, only rewrite the first argument and then check rules α and β. Only if these rules do not match, the other arguments are rewritten and γ is matched. Such a strategy, written as $[\{1\}, \{\alpha, \beta\}, \{2, 3\}, \{\gamma\}]$, can be computed automatically. Note that strategies need to be *full* and *in-time* [15], which means that all rules and argument indices must occur in the strategy and every argument index must occur before the rules that need that argument for matching.

Concerning code generation, this strategy differs from innermost in the fact that the generic rewrite function no longer rewrites the arguments of a function before calling its specific rewrite function. Instead the specific rewrite function itself does this, as specified by the strategy for this function symbol. Also, where there is only one match tree for all rules (with the same head symbol) in innermost, with JITty we have a match tree per set of rewrite rules in the strategy. In the above example this would mean there is a tree matching both rule α and β and a tree matching γ.

The code for a strategy is generated such that the elements in the strategy are executed in order. For the *if* this would mean that the corresponding specific function will consist of first rewriting the first argument, then the code for the match tree of $\{\alpha, \beta\}$, etc., as can be seen in the following code.

```
function $\mathit{rewr}_{\mathit{if}}(\mathit{arg}_1, \mathit{arg}_2, \mathit{arg}_3)$
    $\mathit{arg}_1 := \mathit{rewrite}(\mathit{arg}_1)$
    if $\mathit{arg}_1 = \mathit{true}$ then
        return $\mathit{arg}_2$
    else if $\mathit{arg}_1 = \mathit{false}$ then
        return $\mathit{arg}_3$
    else
        $\mathit{arg}_2 := \mathit{rewrite}(\mathit{arg}_2)$
        $\mathit{arg}_3 := \mathit{rewrite}(\mathit{arg}_3)$
        if $\mathit{arg}_2 = \mathit{arg}_3$ then
            return $\mathit{arg}_2$
        else
            return $\mathit{if}(\mathit{arg}_1, \mathit{arg}_2, \mathit{arg}_3)$
```

5.1 Strategy generation

Because we do not want to burden our users with supplying strategies themselves, we need to generate reasonable strategies from a given set of rewrite rules (i.e. one strategy per function symbol). This is done by observing which arguments need to be rewritten to be able to match a given rule. An argument that is needed for matching by *most* rules is added to the strategy, indicating it needs to be rewritten first. When there are rules for which all arguments essential for matching are rewritten, this rule is added to the strategy. This process continues until all rules and arguments are in the strategy.

More formally, let $dep(r)$ be a function that returns the indices of the arguments that need to be rewritten before matching rule r, i.e. (with $vars(t)$ the variables occurring in t)

$$dep(f(t_1,\ldots,t_k) \to u) = \{i \ : \ t_i \notin \mathbb{V} \ \vee \ t_i \in \bigcup_{j \neq i} vars(t_j)\}$$

Also, let $occ(i, R_f)$ be a function that returns the number of rules of R_f that require argument i:

$$occ(i, R_f) = \#\{r \in R_f \ : \ i \in dep(r)\}$$

We denote the empty strategy with [] and a set S of argument indices or rewrite rules prepended to a strategy l by $S \rhd_c l$. Here, $\rhd_c$ only adds S to l if S is not empty (i.e. $\emptyset \rhd_c l = l$). A strategy for a set of rules R_f is generated with $strat(R_f, \emptyset)$, where $strat(R, I)$ is defined as follows, for any set of rules $R \subseteq R_f$ and set of indices $I \subseteq \{1,\ldots,ar(f)\}$ (with I the set of argument indices added to the strategy so far and $\uparrow$ the maximum quantifier):

$$\begin{array}{ll} strat(\emptyset, I) = (\{1,\ldots,ar(f)\} \setminus I) \rhd_c [] & \\ strat(R, I) = T \rhd_c J \rhd_c strat(R \setminus T, I \cup J) & \text{if } R \neq \emptyset \\ \quad\quad \textbf{where} \ \ T = \{r \in R \ : \ dep(r) \subseteq I\}, & \\ \quad\quad\quad\quad\quad\ \ U = \{i \ : \ i \notin I \ \wedge \ occ(i, R \setminus T) = \uparrow_{j \notin I} occ(j, R \setminus T)\} & \end{array}$$

If we look at the *if* above, we see that the first argument is needed for two rules (α and β) and the other arguments are needed only for γ. So, the first argument is added to the (empty) strategy, after which all essential arguments for α and β are in the strategy and they can be added to the strategy. Then only γ remains to be added, which means that the remaining arguments be added first.

Our approach deviates from the *just-in-time* strategy as defined in [15] in two ways. First of all, we do not require arguments to be rewritten in order. This way we basically get the same strategy as before when we permute the arguments of the *if*. We also do not preserve in any way the order in which rules were specified by the user while *just-in-time* would (as far as a strategy allows this).

5.2 Normal forms

Unlike innermost rewriting, JITty rewriting does not allow for a simple build up mechanism (as described in Sect. 4). To avoid rewriting normal forms we want to tag terms to indicate that they are in normal form (or not). A simple way is to add an extra function symbol ν, such that $\nu(t)$ means that t is in normal form (which is done in [16]). However, such an addition results in a time penalty due to additional construction of terms.

Our approach is to introduce extra function symbols f^s for each original function symbol f. Each extra symbol f^s has an annotation s indicating which of its arguments is in normal form. For example, f^{011} indicates that the second and third arguments are in normal form. We will write ϵ for the absence of an annotation (i.e. f^ϵ is equal to f). Note that having these additional symbols does not add extra costs in construction of terms as the construction only differs in which function symbol is used. And because of the way it is used, normal forms will always be built up of the original function symbols, thus matching does not change at all. The only change is the increase of the number of rewrite methods, which only effects initialisation time and needed (static) memory.

To use these annotations we need to convert the rewrite rules in such a way that they use the annotations. Given a set of variables N and a term t we define $\psi(t, N)$ to be the annotated version of t under the assumption that (the values bound to) the variables of N are in normal form. More precisely (where $[\mathit{true}] = 1$ and $[\mathit{false}] = 0$):

$$\begin{array}{lcl}
\psi(x, N) & = & x \\
\psi(f(t_1, \ldots, t_n), N) & = & f^{[t_1 \in N] \ldots [t_n \in N]}(\psi(t_1, N), \ldots, \psi(t_n, N))
\end{array}$$

Let $ar(f)$ denote the arity of function symbol f, $\mathit{vars}(t)$ the set of variables occurring in t and $\mathit{dep}_f(r)$ the indices of arguments of f that the JITty strategy will have rewritten before trying to apply rewrite rule r. We define a **transformation function** ϕ on TRSs such that $\phi((\Sigma, \rightarrow)) = (\Sigma', \rightarrow')$, where Σ' and $\rightarrow'$ are defined as follows:

$$\begin{array}{lcl}
\Sigma' & = & \{ f^s \ : \ f \in \Sigma \ \wedge \ s \in \bigcup_{0 \le i \le ar(f)} \{0,1\}^i \ \} \\
\rightarrow' & = & \{ f^s(t_1, \ldots, t_n) \rightarrow u' \ \mathbf{if} \ c' \ : f(t_1, \ldots, t_n) \rightarrow u \ \mathbf{if} \ c = r \ \in \rightarrow \ \wedge \\
& & \qquad\qquad\qquad\qquad\qquad s \ \in \ \{\epsilon\} \cup \{0,1\}^n \ \wedge \\
& & \qquad\qquad\qquad\qquad\qquad N = \bigcup_{i \in \mathit{dep}_f(r) \ \vee \ s.i = 1} \mathit{vars}(t_i) \ \wedge \\
& & \qquad\qquad\qquad\qquad\qquad c' = \psi(c, N) \ \wedge \ u' = \psi(u, N) \\
& & \quad \} \ \cup \ \{ f^s \rightarrow f \ \mathbf{if} \ \mathit{true} \ : \ s \neq \epsilon \ \wedge \ f^s \in \Sigma' \}
\end{array}$$

This translation adds the annotated function symbols and annotated copies of the rewrite rules. It makes sure that the right-hand side of rules correctly uses the annotations based on the annotation of the head symbol of the left-hand side and which arguments will be rewritten before application. It also adds rules to remove the annotations.

For these latter rules the code generation has to be adapted such that these are only applied in case no other rule matches. This way we make sure that normal forms are always without annotations, which ensures that matching does not have to consider annotations at all. The function symbols with an annotation indicating that none of the arguments are in normal form can be safely replaced by the unannotated version.

To illustrate the translation, we look at the following example. Assume the following rules (where [] is the empty list and $a \rhd l$ is the list l prepended with a):

$$\begin{array}{lll}\alpha: & \mathit{len}([]) & \to 0\\ \beta: & \mathit{len}(a \rhd l) & \to 1 + \mathit{len}(l)\end{array}$$

Given the above transformation, we obtain the following set of rules. Note that we have annotated the name of the rules as well with the effect that they have on the annotation of *len*.

$$\begin{array}{lllll}\alpha & : & \mathit{len}([]) & \to 0\\ \alpha^1 & : & \mathit{len}^1([]) & \to 0\\ \beta^{\to 1} & : & \mathit{len}(a \rhd l) & \to 1 + \mathit{len}^1(l)\\ \beta^{1\to 1} & : & \mathit{len}^1(a \rhd l) & \to 1 + \mathit{len}^1(l)\\ {}^{1\to} & : & \mathit{len}^1(l) & \to \mathit{len}(l)\end{array}$$

Note that in practice it might not be feasible to use $\phi(R)$ instead of TRS R because of the exponential increase in size. However, it is often sufficient to limit the annotations to, say, 3 arguments.

6 Evaluation

We evaluate the implementations of our mCRL2 rewriters by looking at some benchmarks. These are divided into two parts, viz. benchmarks for rewriting a single closed term and benchmarks for generating labelled transition systems. The reason for this division is that LTS generation, at least as it is implemented in μCRL and mCRL2, uses rewriters in a very specific way.

6.1 LTS generation

The μCRL and mCRL2 toolsets first convert the specification to a *symbolic* LTS, which consists of a list of guarded transitions and the effect on the state these have. Such a guard is an open term that indicates under which valuation of the variables a transition can happen. To generate all such valuations we use a form of narrowing [5]; we repeatedly do case distinction on a variable and rewrite the guard to see if it evaluates to *true* or *false*.

As only a small change is made in each step, most of the time the rewriter will be busy reestablishing that large parts of the guard are still in normal form. Optimisations that avoid normal form rewriting are actually less effective in this setting, as they always need to traverse a term at least once to establish that it is a normal form.

For the LTS benchmarks we have taken four specifications (chatboxt, 1394-fin, ccp33 and commprot) from the μCRL toolset, converted them to symbolic LTSs that are easily translatable to LOTOS [8] (for the CADP toolset [6]) and mCRL2. The used specifications differ slightly from the versions in the μCRL toolset to be able to translate to CADP. Note that, unlike the μCRL and mCRL2 toolsets, CADP is not specialised in handling these symbolic LTSs, which can negatively influence their results. All tools were used on the same machine with 2 gigabytes of memory (of which the tools were only allowed to use 1.5 gigabytes to avoid swapping). Note that we write OoM (out of memory) in case a tool was terminated because it needed more than the allowed amount of memory. For this reason we included additional variants of benchmarks limited to an amount of states that all tools could handle.

Table 1. LTS generation benchmarks

	# states	CADP	μCRL	mCRL2	
				Innermost	*JITty*
chatboxt	65536	1.3s	5.0s	4.0s	3.5s
1394-fin	400	65.3s	0.1s	0.5s	0.4s
1394-fin	371804	OoM	103.8s	212.1s	92.3s
ccp33	7000	25.5s	27.6s	61.8s	8.7s
ccp33	20000	OoM	79.0s	171.9s	26.2s
commprot	700	53.9s	11.0s	12.4s	13.0s
commprot	5000	OoM	77.8s	92.1s	93.0s

Looking at Table 1, we see that our JITty implementation performs better on average than any of the others. The exact difference depends highly on the chosen example, as some depend more heavily on functions that allow for JITty techniques. In the CADP column we see several OoMs indicating the tool needed more than the allowed amount of memory.

Our innermost implementation is about two times as slow as μCRL in the, calculational-wise, heavier cases. This could be either because μCRL also applies JITty-like techniques in a limited fashion or because their implementation does not need to deal with applicative terms. The implementation is otherwise very similar. Given the times in Table 1 it is clear that only in case there is a significant difference in execution time between the mCRL2 implementations there is also a significant difference with μCRL. This seems to support the idea that our innermost rewriter is slower than μCRL because the latter also applies some JITty techniques.

6.2 Closed term rewriting

To investigate the performance of our rewriters in a more general setting we look at the benchmarks in Table 2. These benchmarks consist of only a single closed data term that needs to be rewritten to normal form. In order to test the rewriters of the LTS generation tools we again use μCRL specifications as before, only with a single process that can do precisely one transition which has the term to be rewritten as an argument (such that these tools are effectively only rewriting). In addition to the LTS generation tools we also consider the functional language tools Maude [4], Glasgow Haskell Compiler (GHC) [9], Clean [13] and ASF+SDF. For these tools the process part of the specification is discarded in the conversion.

The benchmarks we use are a naive Fibonacci implementation (fib(32)), benchmarks as used in [11] (evalexp, evalsym, evaltree) and a binary search (b.search). Fibonacci and evalsym are mainly calculational benchmarks, evalexpr differentiates eager and lazy implementations and evaltree is a memory extensive benchmark. The binary search is a benchmark that takes an increasing function, a value and a bound and searches that function (in the domain determined by the bound) for the given value. This benchmark is mainly a test for applicative terms (as the search function takes a function as argument), but also requires a lazy implementation for reasonable execution. The function we use as argument is the Fibonacci function. We write NA (not applicable) in Table 2 for tools that do not support applicative terms.

Table 2. Closed term rewriting benchmarks

	Maude	GHC	Clean	ASF+SDF	CADP	μCRL	mCRL2	
							Innermost	*JITty*
fib(32)	23.4s	4.0s	2.6s	2.7s	2.4s	2.3s	4.0s	11.2s
evalexpr	3.3s	0.4s	0.3s	OoM	0.5s	OoM	OoM	5.4s
evalsym	231.3s	18.7s	15.8s	36.3s	OoM	19.0s	49.3s	254.2s
evaltree	16.7s	OoM	2.1s	1.6s	0.6s	1.0s	1.9s	25.6s
b.search	NA	4.5s	2.5s	NA	NA	NA	OoM	10.8s

From the benchmarks in Table 2 we can see that in general the rewriters of the LTS generators can compete with the fastest rewriters for functional languages available, which seems to indicate that supporting open term rewriting and implicit substitution is not a bottleneck. We can also see that our JITty implementation is often significantly slower than the others and is more comparable to Maude, which uses an interpreting rewriter. This is likely due to the fact that JITty always has to build the result of rule application before rewriting that term, which is very expensive in our implementation. The memory extensive evaltree benchmarks, where JITty is about twelve times slower than our innermost rewriter, seems to support this. Also note that the evalsym benchmark, meant to test pure calculation speed, favors those that use a lazy implementation (ASF+SDF and the mCRL2 innermost rewriter are the only strict innermost rewriters).

7 Conclusion

We have described the implementation of the rewriters of the mCRL2 toolset. The implementation of the innermost rewriter is very similar to the implementation of the μCRL rewriter and the rewriter used in ASF+SDF. The second implementation is that of a compiling JITty rewriter, which is, as far as we know, the first of its kind.

Benchmarks are given to illustrate the improvement this JITty rewriter is over the innermost rewriters used for LTS generation. For closed term rewriting we have shown that our innermost rewriter can compete with the best rewriters currently available (ignoring the effects of lazy rewriting) and that JITty is a bit slower. The latter is likely due to the fact that in this implementation more intermediate terms have to be constructed, which is quite expensive.

The fact that the rewriters used for LTS generation can clearly compete with the fastest rewriters for functional languages seems to suggest that adapting the latter to support open term rewriting (which is essential for LTS generation) should not be a problem. That is, unless these functional languages support additional features with respect to the more basic languages used in process specifications that are fundamentally in conflict with efficient open term rewriting. In any case, such an adaptation would allow developers and users of tools centered around process behaviour and theorem proving (and most likely other fields as well) to have direct access to the functionality offered by the expertise of the functional programming community.

Most significant future work will be the improvement of the JITty rewriter for closed term rewriting and especially the study of the implications of the restrictions we have put on higher-order rewriting.

References

1. Augustsson, L.: Compiling pattern matching. In Jouannaud, J.P., ed.: Proceedings of a Conference on Functional Programming Languages and Computer Architecture (FPCA). Volume 523 of Lecture Notes in Computer Science., Springer-Verlag (1985) 368–381
2. Baxter, L.D.: The complexity of unification. PhD thesis, University of Waterloo (1976)
3. Blom, S.C.C., Fokkink, W.J., Groote, J.F., van Langevelde, I., Lisser, B., van de Pol, J.C.: μCRL: A toolset for analysing algebraic specifications. In Berry, G., Comon, H., Finkel, A., eds.: Computer Aided Verification: 13th International Conference, CAV 2001, Paris, France, July 18-22, 2001, Proceedings. Volume 2102 of Lecture Notes in Computer Science., Springer-Verlag (2001) 250–254
4. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Quesada, J.F.: Maude: specification and programming in rewriting logic. Theoretical Computer Science **285**(2) (2002) 187–243
5. Dershowitz, N., Jouannaud, J.P.: Rewrite systems. In: Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B). MIT Press (1990) 243–320

6. Garavel, H., Lang, F., Mateescu, R.: An overview of CADP 2001. European Association for Software Science and Technology (EASST) Newsletter **4** (2002)
7. Groote, J.F., Mathijssen, A.H.J., van Weerdenburg, M.J., Usenko, Y.S.: From μCRL to mCRL2: Motivation and outline. In Aceto, L., Gordon, A.D., eds.: Proc. Workshop Algebraic Process Calculi: The First Twenty Five Years and Beyond. Number NS-05-3 in BRICS Notes Series (2005) 126–131
8. ISO: ISO 8807: Information processing systems – open systems interconnection – LOTOS – a formal description technique based on the temporal ordering of observational behaviour. Standard, International Standards Organization, Geneva, Switzerland (1987) First edition.
9. Launchbury, J., Sansom, P.M., eds.: The Glasgow Haskell Compiler: A Retrospective. In Launchbury, J., Sansom, P.M., eds.: Functional Programming, Glasgow 1992, Proceedings of the 1992 Glasgow Workshop on Functional Programming, Ayr, Scotland, 6-8 July 1992. Workshops in Computing, Springer (1993)
10. Mathijssen, A.H.J., Pretorius, A.J.: Specification, analysis and verification of an automated parking garage. Technical Report 05/25, Eindhoven University of Technology, ISSN 0926-4515 (2005)
11. Olivier, P.: A Framework for Debugging Heterogeneous Applications. PhD thesis, University of Amsterdam (2000)
12. Peyton Jones, S.L.: The implementation of functional programming languages. Prentice-Hall (1987)
13. Plasmeijer, M.J.: Clean: a programming environment based on term graph rewriting. Electronic Notes in Theoretical Computer Science **2** (1995) 215–221
14. Schnoebelen, P.: Refined compilation of pattern-matching for functional languages. Science of Computer Programming **11**(2) (1988) 133–159
15. van de Pol, J.: Just-in-time: On strategy annotations. In Gramlich, B., Lucas, S., eds.: WRS 2001, 1st International Workshop on Reduction Strategies in Rewriting and Programming. Volume 57 of Electronic Notes in Theoretical Computer Science. (2001) 41–63
16. van de Pol, J.C.: JITty: a rewriter with strategy annotations. In Tison, S., ed.: Rewriting Techniques and Applications : 13th International Conference, RTA 2002, Copenhagen, Denmark, July 22-24, 2002. Proceedings. Volume 2378 of Lecture Notes in Computer Science., Springer-Verlag (2002) 367–370
17. van den Brand, M., van Deursen, A., Heering, J., de Jong, H.A., de Jonge, M., Kuipers, T., Klint, P., Moonen, L., Olivier, P.A., Scheerder, J., Vinju, J.J., Visser, E., Visser, J.: The asf+sdf meta-environment: A component-based language development environment. In Wilhelm, R., ed.: Compiler Construction: 10th International Conference, CC 2001. Volume 2027 of Lecture Notes of Computer Science., Springer (2001) 365–370
18. van den Brand, M.G.J., Heering, J., Klint, P., Olivier, P.A.: Compiling language definitions: the ASF+SDF compiler. ACM Transactions on Programming Languages and Systems (TOPLAS) **24**(4) (2002) 334–368
19. van den Brand, M.G.T., de Jong, H.A., Klint, P., Olivier, P.A.: Efficient annotated terms. Software: Practice & Experience **30**(3) (2000) 259–291
20. Vittek, M.: A compiler for nondeterministic term rewriting systems. In: Rewriting Techniques and Applications, 7th International Conference, RTA-96, New Brunswick, NJ, USA, July 27-30, 1996, Proceedings. Volume 1103 of Lecture Notes in Computer Science., Springer (1996) 154–167

Using Maude and its strategies for defining a framework for analyzing Eden semantics*

Mercedes Hidalgo-Herrero[1], Alberto Verdejo[2], and Yolanda Ortega-Mallén[2]

[1] Departamento de Didáctica de las Matemáticas
[2] Departamento de Sistemas Informáticos, Universidad Complutense de Madrid

Abstract. Eden is a parallel extension of the functional language Haskell. On behalf of parallelism Eden overrides Haskell's pure lazy approach, combining a non-strict functional application with eager process creation and eager communication. We desire to investigate alternative semantics for Eden in order to analyze the consequences of some of the decisions adopted during the language design. In this paper we show how to implement in Maude the operational semantics of Eden in such a way that semantic rules can be modified easily. Moreover, other semantic features can be implemented by means of parameterized modules that allow to instantiate in different ways several parameters of the semantics but without modifying the semantic rules.

Keywords: Operational semantics, parallel functional languages, Eden, rewriting logic, Maude, rewrite strategies.

1 Introduction

It is well-known that functional languages offer great possibilities for parallel programming, ranging from a completely implicit parallelism —for instance an automatic parallelization— to an explicit parallelism where the programmer distributes the computation among a set of communicating processes that even may be located by the programmer himself at designated processors. The parallel language Eden lies more closely to this latter approach, extending Haskell [11] with coordination features for creating processes with stream-based communication.

Haskell is a lazy language, i.e. it adopts normal order evaluation, avoiding repeated computations by sharing reductions. The lazy approach restricts the exploitation of parallelism because expressions are evaluated only under demand. Therefore, Eden overrides the pure lazy approach, combining a non-strict functional application with eager process creation and eager evaluation of communication values. This may produce *speculative* computation, i.e. the calculation of results that may never be used. The amount of speculative computation produced during the evaluation of an Eden program is variable, depending on the number of processors, the speed of basic operations, etc. This interplay between

* Research supported by MCyT Spanish project *MIDAS* (TIC200301000).

laziness and eagerness is precisely established by Eden's operational semantics [5,7]. Moreover, this semantics defines two extreme degrees of speculative computation: minimal and maximal.

We desire to investigate alternative semantics for Eden in order to analyze the consequences of some of the decisions adopted during the language design. For this purpose, it is extremely useful to have a framework where Eden's operational semantics can be easily programmed and that provides mechanisms to reflect with small effort changes in the semantics. Rewriting logic [10] and Maude [3] are excellent candidates for this aim. First, Eden's syntax can be represented literally. Second, Eden's operational semantics rules can be represented in Maude quite literally in most cases, so keeping the representation distance as short as possible. Third, since Maude specifications are executable, we directly get an implementation of Eden where program examples can be executed and analyzed. Finally, a recently proposed *strategy* language [9] for Maude can be used to control in every desired way the application of semantic rules.

In this paper we show how to implement in Maude the operational semantics of Eden in such a way that two main objectives are possible: (1) the semantic rules can be modified in an easy manner so that in a near future we can investigate with different possibilities, and (2) several measures —parallelism, speculative computation, communications, etc.— can be taken by changing some parameters of the semantics (which is defined in a parameterized module) without modifying the semantic rules.

From the point of view of Eden, this is the first step towards a framework where Eden expressions can be evaluated according to different semantics in order to be compared and analyzed. From the point of view of the implementation of operational semantics in Maude, this work constitutes another step in a continued effort to represent semantics for more complex languages. The simplest concurrent language we have considered is Milner's CCS in [13], that does not require any strategies. This is not the case with Cardelli and Gordon's Ambient Calculus that we have tackled in [12]. However, the use of strategies in the latter solves problems different from the ones we consider in this paper, where we take into account that Eden's semantics inherently depends upon an order of application of the rules, thus exploiting the strategy language expressiveness.

The rest of the paper is organized as follows. First we present a brief introduction to Maude; for a complete treatment we refer the reader to the Maude manual [3]. Section 3 gives an overview of Eden and implements its kernel syntax, while Section 4 is devoted to the operational semantics and its implementation in Maude. In Section 5 we extend our framework in order to be able to obtain measures from the computations. The last section presents our conclusions and outlines future work.

2 Visiting Maude

In Maude the state of a system is formally specified as an algebraic data type by means of an equational specification. Maude uses a very expressive version of equational logic, namely *membership equational logic* [2]. In this kind of specifications we can define new types (by means of the keyword `sort(s)`); subtype relations between types (`subsort`); operators (`op`) for building values of these types, giving the types of their arguments and result, and which may have attributes as being associative (`assoc`) or commutative (`comm`), for example; equations (`eq`) that identify terms built with these operators; and memberships (`mb`) $t : s$ stating that the term t has sort s. Both equations and memberships can be conditional. Conditions are formed by a conjunction (written `/\`) of equations and memberships. Equations are assumed to be confluent and terminating, that is, we can use the equations from left to right to reduce a term t to a unique (modulo the operator attributes as associativity, commutativity, and identity) canonical form t' that is equivalent to t, i.e. they represent the same value.

The *dynamic* behavior of a system is specified by rewrite rules of the form

$$t \longrightarrow t' \ \mathit{if} \ \ (\bigwedge_i u_i = v_i) \wedge (\bigwedge_j w_j : s_j) \wedge (\bigwedge_k p_k \longrightarrow q_k)$$

that describe the local, concurrent transitions of the system. That is, when part of a system matches the pattern t and the conditions are fulfilled, it can be transformed into the corresponding instance of the pattern t'.

Maude modules can be *parameterized* with one or more parameters, each of which is expressed by means of one theory that defines the interface of the module, that is, the structure and properties required of an actual parameter.

Rewrite rules need be neither confluent nor terminating. This theoretical generality requires some control when the specifications become executable, because it must be ensured that the rewriting process does not go in undesired directions. We have defined a strategy language for Maude that can be used to control how rules are applied to rewrite a term [9]. The simplest strategies are the constants `idle`, which always succeeds by doing nothing, and `fail`, which always fails. The basic strategies consist of the application of a rule (identified by the corresponding rule label) to a given term, and with the possibility of providing a substitution for the variables in the rule. In this case a rule is applied *anywhere* in the term where it matches satisfying its condition. When the rule being applied is a conditional rule with rewrites in the conditions, the strategy language allows to control by means of search expressions how the rewrite conditions are solved. An operation `top` to restrict the application of a rule just to the *top* of the term is also provided. Basic strategies are then combined so that strategies are applied to execution paths. Some strategy combinators are the typical regular expression constructions: concatenation (`;`), union (`|`), and iteration (`*` for 0 or more iterations, `+` for 1 or more, and `!` for a "repeat until the end" iteration). Another strategy combinator is a typical if-then-else, but generalized so that the first argument is also a strategy. The language provides a `(x)matchrew` combinator that allows a term to be split in subterms, and specifies how these subterms have to be rewritten.

3 A quick excursion to Eden

Eden [7] extends the non-strict functional language Haskell with a set of *coordination* features to control parallel evaluation of processes. Coordination in Eden is based on two principal concepts: *explicit definition of processes* and *implicit stream-based communication*, i.e. there are not communication primitives such as *send* and *receive*. As well as there is a distinction between function definition and function application, Eden includes *process abstractions*, i.e. abstract schemes for process behavior, and *process instantiations* for the actual creation of processes. Moreover, *nondeterminism* is introduced in Eden by means of a predefined process abstraction which is used to instantiate nondeterministic processes that fairly merge several input streams into a single output stream.

For the purpose of this paper we just concentrate on Eden's essentials, which are captured by the untyped λ-calculus whose abstract syntax is given next, where $x \in \mathit{Var}$ represents identifiers and $E \in \mathit{Exp}$ represents expressions:

$$\begin{array}{rll} E ::= & x & \text{identifier} \\ \mid & \lambda x.E & \lambda\text{-abstraction} \\ \mid & E_1 E_2 & \text{application} \\ \mid & E_1 \texttt{\#} E_2 & \text{process creation} \\ \mid & \texttt{let}\ \{x_i = E_i\}_{i=1}^{n}\ \texttt{in}\ E & \text{local declaration} \end{array}$$

When evaluating the expression $E_1\texttt{\#}E_2$ inside a process p, a new child process q is created together with two communication channels. The child is fed with the value of E_2 via the input channel by its parent process p. Process q evaluates $E_1\ E_2$ and returns the result (to its parent) via the output channel.

The language is normalized to a restricted syntax where all subexpressions, except for the body of λ-abstractions, are replaced by variables defined in `let`-expressions. This guarantees that subexpressions are shared, and are evaluated at most once. We also assume a general renaming of variables for avoiding name clashes during expression evaluation.

For instance, the evaluation of the following expression

$$\texttt{let}\ x_0 = x_1\texttt{\#}x_1, x_1 = \backslash x.x, x_2 = 1, x_3 = x_4\, x_0, x_4 = x_5\, x_2, x_5 = \backslash y.(\backslash z.z)\ \texttt{in}\ x_3$$

gives place to a process creation: the main process evaluates x_3 while the child computes the application $x_1\, x_1$. In order to do that, the child process needs the value of x_1 twice:

1. for obtaining the λ-abstraction: the definition is copied to the child's heap, and
2. for getting the argument: the parent communicates the value to the child.

By the end of the evaluation of the application, the resulting value is sent back to the parent process.

3.1 Representation in Maude

We define in Maude the syntax of the kernel of Eden given above. We use sorts and subsorts to represent the different syntactic categories and their relations. Having different sorts allows us to concrete the patterns used in rewrite rules by using (Maude) variables of the most appropriate sort. We have sorts for ordinary variables (`Std`), for channels (`Cha`) and for the union of both sets (`Var`). We also use two sorts for distinguishing between expressions that are in weak head normal form (`Whnf`) and those that are not (`NonWhnf`). Both are Eden expressions (`Exp`).

```
sorts Std Cha Var Whnf NonWhnf Exp .
subsorts Std Cha < Var < NonWhnf .
subsorts Whnf NonWhnf < Exp .
```

We define constructors for building expressions. For each constructor, the most concrete sort is used as the result sort; for example, a λ-expression `\_._` is a weak head normal form, whereas an application `__` (empty syntax) is not. Strings are used as variable identifiers.

```
op s : String -> Std .                    op c : String -> Cha .
op \_._ : Std Exp -> Whnf .               op __ : Exp Exp -> NonWhnf .
op let_in_ : LetBinds Exp -> NonWhnf .    op _#_ : Exp Exp -> NonWhnf .
```

4 Operational semantics

In this section we describe an operational semantics in the style of [1]. It is our purpose just to describe the structure of the semantics and to present some of the transition rules focusing on how they have been implemented in Maude, but neither to explain nor to justify their definition. For a more detailed overview of Eden's semantics, the reader is referred to [7]; a version extended with streams for communication, dynamic channels and nondeterminism can be found in [5]. Correctness proofs, examples and applications are gathered in [4].

4.1 A two-level transition system

A process is represented by a pair $\langle p, H\rangle$, where p is a process identifier and H is the heap collecting the variable-to-expression bindings that model the closures corresponding to the process evaluation state. Each binding is considered a potential thread to be executed by the available processors, so that a label indicates the thread state: $x \overset{\alpha}{\mapsto} E$, where $\alpha ::= I|A|B$ corresponds to *Inactive* (either not yet demanded or already completely evaluated), *Active* (or demanded), and *Blocked* (demanded but waiting for the value of another binding), respectively. Channel identifiers can appear on either side of a binding: on the left-hand side they represent outports; while on the right-hand side they denote inports.

In the following, we will use $x, y, z \in \mathit{Std}$ for ordinary variables, $c \in \mathit{Chan}$ for channels, $\theta \in \mathit{Var} = \mathit{Std} \cup \mathit{Chan}$, and p and q for process identifiers.

The model of evaluation is represented by a sequence of systems —a system is a set of parallel processes— regulated by the transition rules. Some of the bindings in a heap are executed in parallel, sharing the data of the corresponding process; but bindings in different processes can only share information through process communication. The semantics needs small-step transitions to model parallelism in a synchronous way, in the sense that single reductions are local and independently carried out at each process and then combined before proceeding to the next step. The semantics reflects the distinction between the two sub-languages (computation and coordination) that configure Eden, so that it consists of a two-level transition system: the lower level handles local effects within processes, while the upper level describes the effects global to the whole system, like process creation and data communication.

4.2 Representing the transition system in Maude

A thread is built with a variable, a thread state (of sort `TState`), and an expression. A heap is a set of threads: `none` represents the empty heap, a thread represents a singleton heap, `subsort Thread < Heap`, and the union `_+_` of heaps builds them. The union constructor is declared to be associative, commutative, and with the empty heap as the identity element; pattern matching will take place modulo these properties. Finally, the process constructor has four arguments: a string corresponding to the process identifier; a heap; and two counters: one represents the number of children of this process and the other indicates the maximum number used to build new variables (incremented when renamings are needed because of the generation of new variables). There is also a union operator `__` (with empty syntax) for building systems.

```
sorts TState Thread Heap Process System .
subsort Thread < Heap .
subsort Process < System .
ops A I B : -> TState .
op _|-_->_ : Var TState Exp -> Thread .
op none : -> Heap .
op _+_ : Heap Heap -> Heap [assoc comm id: none] .
op <_,_,_,_> : String Heap Nat Nat -> Process .
op empty : -> System .
op __ : System System -> System [assoc comm id: empty] .
```

We have defined several auxiliary operations needed by the semantics: substitution, renaming of variables in a heap, normalization, etc. They are defined structurally by means of equations and using the `owise` (otherwise) Maude attribute. For the complete Maude code we refer the reader to [6].

4.3 Local process evolution

Local transitions express the reduction of an active thread in the context of a single process. This internal activity affects only the corresponding heap. The evaluation of an expression terminates when it reaches a whnf value ($W \in \mathit{Whnf}$).

$$H + \{x \overset{I}{\mapsto} W\} : \theta \overset{A}{\mapsto} x \longrightarrow H + \{x \overset{I}{\mapsto} W, \theta \overset{A}{\mapsto} W\} \qquad \textbf{(value)}$$

$$\text{if } E \notin \mathit{Whnf},\ H + \{x \overset{IAB}{\longmapsto} E\} : \theta \overset{A}{\mapsto} x \longrightarrow H + \{x \overset{AAB}{\longmapsto} E, \theta \overset{B}{\mapsto} x\} \qquad \textbf{(demand)}$$

$$H : x \overset{A}{\mapsto} x \longrightarrow H + \{x \overset{B}{\mapsto} x\} \qquad \textbf{(blackhole)}$$

$$\text{if } E \notin \mathit{Whnf},\ H + \{x \overset{IAB}{\longmapsto} E\} : \theta \overset{A}{\mapsto} x\,y \longrightarrow H + \{x \overset{AAB}{\longmapsto} E, \theta \overset{B}{\mapsto} x\,y\} \qquad \textbf{(app-demand)}$$

$$H + \{x \overset{I}{\mapsto} \lambda z.E\} : \theta \overset{A}{\mapsto} x\,y \longrightarrow H + \{x \overset{I}{\mapsto} \lambda z.E, \theta \overset{A}{\mapsto} E[y/z]\} \qquad (\beta\textbf{-reduction})$$

$$H : \theta \overset{A}{\mapsto} \texttt{let}\ \{x_i = E_i\}\ \texttt{in}\ x \longrightarrow H + \{y_i \overset{I}{\mapsto} E_i\sigma\}_{i=1}^{n} + \{\theta \overset{A}{\mapsto} \sigma(x)\} \qquad \textbf{(let)}$$

$$\text{where } \mathit{fresh}(y_i)\ (1 \le i \le n) \text{ and } \sigma := [y_1/x_1, \ldots, y_n/x_n]$$

Fig. 1. Local transition rules

Local transitions take the form $H : \theta \overset{A}{\mapsto} E \longrightarrow H'$, which is read as "the evaluation of the *active thread* $\theta \overset{A}{\mapsto} E$ transforms the heap $H + \{\theta \overset{A}{\mapsto} E\}$ into H'". In Figure 1 we show the local rules expressing how lazy evaluation progresses under demand. We avoid writing multiple similar transition rules by allowing a binding to appear with several labels, corresponding to the different possibilities admitted by the rule. Thus, $x \overset{IAB}{\longmapsto} E$ on the left-hand side of rule **(demand)**, and $x \overset{AAB}{\longmapsto} E$ on the right-hand side means that the thread corresponding to the closure $x \mapsto E$ becomes active in the case it was inactive, and remains active or blocked otherwise.

In Maude we represent the semantic rules as rewrite rules. There are several ways of mapping inference systems into rewriting logic [8]. In the structural operational semantics case, judgements typically have the form of some kind of transition $P \to Q$ between states, so that it makes sense to map directly this transition relation between states to a rewriting relation between terms representing the states. Thus, an inference rule of the form

$$\frac{P_1 \to Q_1 \quad \ldots \quad P_n \to Q_n}{P_0 \to Q_0}$$

becomes a *conditional* rewrite rule of the form

$$P_0 \longrightarrow Q_0 \quad \mathit{if} \quad P_1 \longrightarrow Q_1 \wedge \ldots \wedge P_n \longrightarrow Q_n.$$

In this way the semantic rules become (conditional) rewrite rules: the transition in the conclusion becomes the main rewrite of the rule, and the transitions in the premises become rewrite conditions [13].

The local transition rules (as those given in Figure 1) are translated quite literally. We introduce two new constructors for representing heaps. The first one, `_:_`, is already used in the semantic rules in order to separate the leading thread —that which is going to evolve— from the rest of the heap. The second one, `_&_`,

is used in the right-hand side of rewrite rules in order to separate the modified threads from the unmodified ones because this separation will be useful later. Actually, the rule **(demand)** puts together three transition rules, one for each possible state of the thread consulted in the heap. The rewrite rule `demand` given below represents the three semantic rules at the same time, by using a variable `T` of sort `TState` and auxiliary operations for detecting if the thread is modified or not. Notice how the variable `NW` of sort `NonWhnf` is used to ensure the condition $E \notin \mathit{Whnf}$ in the semantic rule. The rule `let` also uses auxiliary operations to build the new heap on the right. This rule rewrites a process instead of only a heap because, due to the renaming, the fourth argument has to be incremented.

```
rl [value] : H + X |- I -> W : Theta |- A -> X
          => H + X |- I -> W & Theta |- A -> W .

rl [demand] : H + X |- T -> NW : Theta |- A -> X
           => H + nmd(X |- T -> NW) & md(X |- T -> NW) + Theta |- B -> X .

rl [blackhole] : H : X |- A -> X
              => H & X |- B -> X .

rl [app-demand] : H + X |- T -> NW : Theta |- A -> X Y
         => H + nmd(X |- T -> NW) & md(X |- T -> NW) + Theta |- B -> X Y .

rl [beta-reduction] : H + X |- I -> \ Z . E : Theta |- A -> X Y
                   => H + X |- I -> \ Z . E & Theta |- A -> E [Y / Z] .

rl [let] :  < p, H : Theta |- A -> let LBS in X, N, M >
   => < p, H & letBindsToHeap(Theta, LBS, X, newvars(p, M, numvars(LBS))),
        N, M + numvars(LBS) > .
```

4.4 Local parallelism

Local evolutions —corresponding to the local transition rules— are considered to occur simultaneously, entwined in a parallel step. The rule given in Figure 2 expresses the evolution of parallel threads inside a process, where $\mathcal{ET}(S)$ is the set of active threads in the system S that are allowed to evolve. $H^{(i,1)}$ is the part of H that remains unchanged during the application of the corresponding local rule, while $K^{(i,2)}$ contains the bindings from $H^{(i,2)}$ that have been modified. It is guaranteed that there is no interference among local transitions. There are several possibilities for defining $\mathcal{ET}(S)$, depending on the number of available processors, the allowed degree of speculative computation, the priority given to some threads, etc. Maude modularity, by means of parameterized modules, is very useful to implement and then compare different scheduling strategies, as we will see in Section 4.6.

The rule **(local parallel)** is quite “abstract.” First, it has a variable amount of premises, depending on the number of threads returned by $\mathcal{ET}(S)$; and second, it makes separations of the heaps distinguishing between modified and unmodified threads. For its implementation, we have solved the second problem by

$$\frac{\left\{H^{(i,1)} + H^{(i,2)} : \theta^i \overset{A}{\mapsto} E^i \longrightarrow H^{(i,1)} + K^{(i,2)} \quad \text{s.t. } H = H^{(i,1)} + H^{(i,2)} + \{\theta^i \overset{A}{\mapsto} E^i\} \text{ and } \theta^i \overset{A}{\mapsto} E^i \in \mathcal{ET}(S) \cap H\right\}_{i=1}^{n}}{H \xrightarrow{par}_S (\cap_{i=1}^n H^{(i,1)}) \cup (\cup_{i=1}^n K^{(i,2)})}$$

where $n = |\mathcal{ET}(S) \cap H|$

Fig. 2. (local parallel) rule

modifying the right-hand side of local rules with the _&_ operator. To deal with the first problem we have devised several approaches; we show here the one which represents the resolution of premises, and the calculation of intersections and unions in the right-hand side of the conclusion *step by step*, by means of rewrite rules. We have chosen this form because it is similar to its mathematical presentation, it simplifies the strategies needed, and it is more efficient.

We consider the following three rewrite rules as the basic steps of an algorithm that implements the **(local parallel)** rule. The rule extend adds to the process three arguments: the first one is the set of variables associated with threads that have to evolve (new variable VS, explained below), the second represents the (partial) evaluation of the intersection of unmodified threads (initially the whole heap), and the third represents the (partial) evaluation of the union of modified threads (initially the empty heap). The rule parallel-step performs the main step of the algorithm, by solving one premise each time. It is a conditional rewrite rule: the first two conditions (which are matching equations) extract the thread corresponding to the variable Theta from the heap H, and the third (rewrite) condition represents the premise in **(local parallel)** corresponding to the variable Theta. This last condition has to be solved by using one of the local transition rules. Notice that the heap H is kept unmodified because it is used in the resolution of each premise, and the variable Theta is removed from the set VS. Finally, the rule contract removes the extra arguments from a process, and performs a final union of heaps.

```
rl [extend] :  < p, H, N, M > => < p, H, VS, H, none, N, M > .
crl [parallel-step] :  < p, H, Theta . VS, H’, K, N, M > =>
                       < p, H, VS, int(H’,H1), K + K1, N’, M’ >
   if Theta |- T -> E := lookUp(Theta, H) /\ H1-2 := filter(Theta, H) /\
      < p, H1-2 : Theta |- T -> E, N, M > => < p, H1 & K1, N’, M’ > .
rl [contract] :  < p, H, mt, H’, K, N, M > => < p, H’ + K, N, M > .
```

The application of these three rules has to be controlled. First of all, the rule extend is applied by providing the concrete value for variable VS, namely the variables in $\mathcal{ET}(S) \cap P$, where P is the process being rewritten and S is the whole system[3]; then, the rule parallel-step is applied as many times as possible, i.e. once for each thread in $\mathcal{ET}(S) \cap P$; and finally, the rule contract

[3] This intersection is computed when the strategy -par-> is called from strategy =par=> bellow.

has to be applied. The following strategy `-par->`, that receives as argument a set of variables, corresponds to this concrete application of the rules. It represents the relation $\xrightarrow{par}_S$ in the semantics (defined in Figure 2).

```
sop -par-> : VarSet .
seq -par->(ActVS:VarSet) = extend[VS:VarSet <- ActVS:VarSet] ;
                           ( parallel-step ! ) ; contract .
```

An alternative way of implementing the relation $\xrightarrow{par}_S$ would put more control in the strategy, making it to traverse the set of evolvable variables and applying a local rule to each of these variables (by means of other strategies). Although in this case the rule `parallel-step` would be simplified, the approach presented before has proved to be more efficient by doing the rewrite rules more powerful, and by simplifying the strategies.

4.5 Global system evolution

At an upper level we define global transitions between process systems represented by sets of processes. A global transition takes the general form:

$$S \overset{\diamond}{\Longrightarrow} \{\langle p, H'_p\rangle\}_{\langle p,H_p\rangle\in S} \cup S'$$

where each heap H_p (associated to a process p in the system S) is transformed to H'_p, while new processes (in S') may be created. The diamond $\diamond$ is a place-holder for the name of the rule.

Parallel Now we consider the parallel evolution of processes within a system S:

$$\textbf{(parallel)}\ \frac{\{H_p \xrightarrow{par}_S H'_p\}_{\langle p,H_p\rangle\in S}}{S \overset{par}{\Longrightarrow} \{\langle p, H'_p\rangle\}_{\langle p,H_p\rangle\in S}}$$

This rule has a variable number of premises, one for each process in the system S. Each premise makes the corresponding process to evolve exactly once through the transition $\xrightarrow{par}_S$. We implement this rule by means of the strategy `=par=>` that applies the strategy `-par->` to each process in a system. This strategy is recursive and it terminates when the rest of the system (represented by the variable `S:System` below) is empty. The strategy `=par=>` receives as argument the variables corresponding to the threads returned by the function $\mathcal{ET}$ applied to the whole system. Strategy `-par->` is called with the set of evolvable variables of process `P`, calculated by function `inters`.

```
sop =par=> : VarSet .
seq =par=>(VS:VarSet) = if (match empty) then idle
        else (matchrew P:Process S:System by
                P:Process using -par->(inters(P:Process, VS:VarSet)),
                S:System using =par=>(VS:VarSet) ) fi .
```

$$\begin{array}{c}
(S, \langle p, H + \{\theta \overset{\alpha}{\mapsto} x\#y\}\rangle) \xrightarrow{pc} (S, \langle p, H + \{\theta \overset{B}{\mapsto} c_1, c_2 \overset{A}{\mapsto} y\}\rangle, \\
\qquad\qquad \langle q, \eta(\mathsf{nh}(x, H)) + \{c_1 \overset{A}{\mapsto} \eta(x)\, z, z \overset{B}{\mapsto} c_2\}\rangle) \\
\text{if } \mathsf{nf}(x, H + \{\theta \overset{\alpha}{\mapsto} x\#y\}) = \emptyset \\
q, z, c_1, c_2 \text{ are fresh identifiers and substitution } \eta \text{ replaces all variables by fresh ones}
\end{array}$$

Fig. 3. (process creation) rule

Multi-step rules After each process has internally evolved, the following tasks have to be done at the system level: process creation, interprocess communication and state management (thread unblocking and deactivation). In general, these tasks imply multiple single steps, each involving at most two processes. Let S be a process system, and $\diamond$ the name of a rule ($\diamond \neq \mathit{par}$), for each single-step rule $S \xrightarrow{\diamond} S'$ we define a multi-step rule $S \overset{\diamond}{\Longrightarrow} S'$ satisfying: $S \xrightarrow{\diamond}{}^{*} S'$ and, there is no S'' such that $S' \xrightarrow{\diamond} S''$. The application of a single-step rule $\diamond$ to some binding in some process may enable the application of the same rule $\diamond$ to other bindings —in the same or in other processes— but it can never disable applications of rule $\diamond$ which were enabled before the former application.

Single-step rules are implemented in Maude as rewrite rules, while the relations $\overset{\diamond}{\Longrightarrow}$ are built by means of strategies. Afterwards, these relations are combined through more strategies.

Process creation The initial heap of a child process contains all the bindings that are needed for the evaluation of the dependent variables in the process body; these are copied from the parent to the child heap by the function nh (*needed heap*): $\mathsf{nh}(x, H)$ collects all the bindings in H that are reachable from x. A renaming η with fresh variables is applied to avoid name clashes. A process creation (see $\xrightarrow{pc}$ rule in Figure 3) is blocked if there is some dependency on values that have to be communicated. The function nf (*needed free*) collects the dependencies derived from the free variables.

Let us consider again the expression given as example in Section 3. After the application of the **(let)** local rule, the resulting heap is the one shown in the left-hand side of the following picture; and the **(process creation)** rule generates the structure in the right-hand side:

$$\begin{array}{|l|}
\hline
\textbf{main}\text{ (N. Children: 0)} \\
\quad \mathit{main} \overset{A}{\mapsto} x_3 \\
\quad x_0 \overset{I}{\mapsto} x_1\#x_1 \\
\quad x_1 \overset{I}{\mapsto} \backslash x.x \\
\quad x_2 \overset{I}{\mapsto} 1 \\
\quad x_3 \overset{I}{\mapsto} x_4\, x_0 \\
\quad x_4 \overset{I}{\mapsto} x_5\, x_2 \\
\quad x_5 \overset{I}{\mapsto} \backslash y.(\backslash z.z) \\
\hline
\end{array}
\quad \Longrightarrow \quad
\begin{array}{|l|l|}
\hline
\textbf{main}\text{ (N. Children: 1)} & \mathbf{p_1}\text{ (N. Children: 0)} \\
\quad \mathit{main} \overset{B}{\mapsto} x_3 & \quad x_6 \overset{B}{\mapsto} c_0 \\
\quad x_0 \overset{B}{\mapsto} c_1 & \quad x_8 \overset{I}{\mapsto} \backslash x_7.x_7 \\
\quad x_1 \overset{I}{\mapsto} \backslash x.x & \quad c_1 \overset{A}{\mapsto} x_8\, x_6 \\
\quad x_2 \overset{I}{\mapsto} 1 & \\
\quad x_3 \overset{A}{\mapsto} x_4\, x_0 & \\
\quad x_4 \overset{I}{\mapsto} x_5\, x_2 & \\
\quad x_5 \overset{I}{\mapsto} \backslash y.(\backslash z.z) & \\
\quad c_0 \overset{A}{\mapsto} x_1 & \\
\hline
\end{array}$$

where c_7 is the inport of the child whereas c_8 is the outport; both of them are internal variables that have not been defined by the programmer since communications in Eden are implicit.

The following rewriting rule implements in Maude the $\xrightarrow{pc}$ rule. It uses auxiliary functions to rename the heap copied into the child, and to build new variables and channels.

```
crl [pc] : < p, H + Theta |- T -> X # Y, N, M >
        => < p, H + Theta |- B -> c1 + c2 |- A -> Y,N + 1, M + 1 >
           < q, H' + c1 |- A -> (searchVar(X,VVL) Z) + Z |- B -> c2, 0, M' >
if nf(X, H + Theta |- T -> X # Y) = none /\  q :=  childName(p, N) /\
   c2 := c(newvar(p, M)) /\ c1 := c(newvar(q, 0)) /\
   Z := s(newvar(q, 1))  /\ < H',VVL,M' > := renH(nh(X,H),q,2) .
```

When designing Eden there was great discussion about how to distribute computation between a process and its children. In the one extreme the parent would advance as much work as possible, so that every dependent variable of the instantiation body should be bound to a whnf before creating the child process. But this may lead to a poor parallelization, where a process has to do too much computation before delegating work to a helping process. In the other —we could say the "laziest"— extreme the parent would pass on all the work to its offspring, so that for a normalized expression $x\#y$, the argument y would be evaluated by the parent, while the body x as well as the application, $x\,y$, would be evaluated by the newborn child. This may lead to repeated calculations, because certain subexpressions may get evaluated independently by several children of the same parent. But this can be easily avoided by the programmer, by forcing the evaluation in the parent of these common subexpressions. The latter option has been adopted for Eden and its actual implementation, and this has been reflected in the operational semantics presented in [7]. We can represent the different approaches in our semantics just by modifying the equational definition of the function `nf`. Each definition is specified in a different module that then is used to instantiate the parameterized module defining the semantics, which has as a parameter a theory requiring a function `nf`.

The relation $\overset{pc}{\Longrightarrow}$ is implemented in Maude as the following strategy, that iterates the application of the rule `pc` as many times as possible:

```
  sop =pc=> .
  seq =pc=> = pc ! .
```

Communication The rule for value communication can be easily understood by looking at its implementation in Maude:

```
crl [com] :
    < p, Hp + ch |- T -> W, N, M > < c, Hq + Theta |- B -> ch, N', M' >
 => < p, Hp, N, M >  < c, Hq + H' + Theta |- A -> (msubs(W',VVL)),N',N2 >
if nf(W, Hp) = none /\ < H',VVL,N1 > := renH(nh(W, Hp),c,M') /\
   < W',N2 > := renL(W,c,N') .
```

When communicating a value it is mandatory to copy —from the producer's heap to the consumer's heap— all the bindings needed for the evaluation of the dependent variables in the value. This copy can only take place if the value does not depend on pending communications.

Scheduling Once all the enabled process creations and communications have been done, the following tasks have to be achieved:

– Unblocking bindings depending on a variable bound to a whnf value meanwhile (*wUnbl*).
– Deactivating bindings to values in whnf (*deact*).
– Blocking process creations that could not be executed (*bpc*).
– Demanding bindings needed for pending process creations and/or communications (*pcd* and *vComd*).

The corresponding rules are given in [5,7] and they are easily readable in the Maude implementation [6]. Their iteration and sequential composition produce a new global rule $\overset{unbl}{\Longrightarrow} = \overset{wUnbl}{\Longrightarrow} ; \overset{deact}{\Longrightarrow} ; \overset{bpc}{\Longrightarrow} ; \overset{pcd}{\Longrightarrow} ; \overset{vComd}{\Longrightarrow}$ that is combined with the other two rules explained before to obtain the global transition $\overset{sys}{\Longrightarrow} = \overset{comm}{\Longrightarrow} ; \overset{pc}{\Longrightarrow} ; \overset{unbl}{\Longrightarrow}$. The following Maude strategies define both relations:

```
sop =unbl=> .
seq =unbl=> = =wUnbl=> ; =deact=> ; =bpc=> ; =pcd=> ; =vComd=> .
sop =sys=> .
seq =sys=> = =com=> ; =pc=> ; =unbl=> .
```

Transition system step Finally, each transition step of the system is defined as $\Longrightarrow = \overset{par}{\Longrightarrow} ; \overset{sys}{\Longrightarrow}$. In Maude, the following strategy allows to compute a transition step. It will be applied to the whole system S that is being evolved, and first it applies strategy `=par=>` by passing as argument the set of evolvable threads returned by $\mathcal{ET}(S)$.

```
sop ==> .
seq ==> = (matchrew S:System by S:System using =par=>(ET(S:System))
          ) ; =sys=> .
```

4.6 Speculative parallelism

In any concrete implementation the evaluation of an Eden program may give rise to different computations. The exact amount of speculative parallelism depends on the number of available processors, the scheduler decisions and the speed of basic instructions. Hence, the execution of a program may range from reducing the speculation to the *minimum* —only what is effectively demanded is computed— to expanding it to the *maximum* —every speculative computation is carried out. While the former would be equivalent to executing the program on a single processor with the scheduler giving priority to the demand originated by the main thread, the latter would correspond to having an unlimited set of processors for evaluating the output of every generated process. It is also possible to reflect in the semantics the distribution of a *limited number of processors* among the active threads following different rules, for instance: randomly among the threads, or fairly distributing the processors among the threads, or even giving priority to the demands of the main thread and distributing the rest of the processors among the other threads.

Once again, the facilities and modularity of Maude allow us to produce an implementation where to experiment different alternatives by selecting the appropriate definition of the functions `nf` and `ET`. These functions can be defined in different ways, thus obtaining different semantics for Eden. In the present implementation we have put each definition in a different Maude module. By instantiating the module defining the semantics rules with a module with a concrete definition of `nf` and a module with a concrete definition of `ET`, we obtain a complete specification of Eden.

5 Computation measures

In this section we show how our framework can be extended in order to perform measurements over the computations, such as work done, degree of parallelism, amount of communications, and so on. Modularity, particularly the separation between rules and strategies, is again an useful instrument because the necessary changes do not imply to modify the already implemented semantic rules.

First of all, the term being rewritten is extended with the actual values of the measures. One possible way to do that is by means of a set of attributes together with their values. One of these attributes contains the Eden system (`Sys`), that will be rewritten by the semantics rules shown in the previous sections. Here we show some examples of attributes.

```
sorts Attr AttrSet .
subsorts Attr < AttrSet .
op nilAS : -> AttrSet .
op __ : AttrSet AttrSet -> AttrSet [assoc comm id: nilAS] .
op Sys : System -> Attr .
op Work : Nat -> Attr .       --- Number of evolved active threads
op NumProc : Nat -> Attr .    --- Number of processes
op MaxPar : Nat -> Attr .     --- Maximum thread parallelism
op AvPar : Nat -> Attr .      --- Average thread parallelism
op AvProcPar : Nat -> Attr .  --- Average process parallelism
```

Then, rewrite rules have to be defined to describe the modification of these measures. For example, the following rule `addPC` increments by one the number of processes, and the rule `addET` increments by a varying amount `CardET` (a new variable in the right-hand side that will be instantiated by a strategy) the done work and updates the maximal thread parallelism.

```
rl [addPC] : NumProc(N) => NumProc(N + 1) .
rl [addET] : MaxPar(Max) Work(W) =>
             MaxPar(max(Max, CardET)) Work(W + CardET) .
```

And finally, we need to modify the strategies in order to apply these rules together with the semantics rules. We show below two of these new modified strategies. Strategy `=pc=>` now applies rule `addPC` after applying rule `pc` (process creation). And strategy `==>` updates the values of measures `MaxPar` and `Work` using the number of evolvable threads computed by expression `size(ET(S:System))`.

```
seq =pc=> = (pc ; addPC) ! .
seq ==> = (xmatchrew Sys(S:System) MaxPar(Max:Nat) Work(W:Nat)
              by S:System using =par=>(ET(S:System)) ,
                 MaxPar(Max:Nat) Work(W:Nat) using
                     addET[CardET <- size(ET(S:System))]
           ) ; =sys=> .
```

We consider once again the example of Section 3. We have instantiated the semantics with two different definitions of function `ET` corresponding to the *minimal* semantics (the final configuration is the first one where the main variable becomes inactive), and the *maximal* semantics (the execution continues until there is no more active thread in the system). The results are shown in the following table.

	Minimal	Maximal
Execution time/global steps	12	7
Total work done	12	10
Average thread parallelism	1	1.43
Maximal thread parallelism	1	3
Average process parallelism	1.92	1.87
Speed	1.71	

Now we have to look for representative examples that exploit the differences between the semantics, and study other alternative definitions of function `ET`; thus, obtaining conclusions of the measurements.

6 Conclusions and future work

The conjugation of Eden operational semantics and Maude has proved to be fruitful because the characteristics of the latter meet Eden semantics implementation needs very faithfully. For instance, Maude rewrite rules mechanism has been an excellent tool for implementing the reduction steps in Eden semantics.

Furthermore, Maude modularity has helped to implement different language design decisions, or the scheduler options, which depend on the threads that are allowed to evolve at each step. We have been able to implement a prototype tool where the user can play with different parameters of the semantics.

Moreover, Maude high level of abstraction has allowed us to obtain an implementation of the rules very similar to the operational rules. Consequently, the code is exceptionally readable, in fact, its reading is almost equal to reading the original semantics. Maude not only has been useful in the semantic aspects, but also at the syntactical level: thanks to Maude operators we have defined the syntax in a very direct way. Besides, the existence of subsorts has facilitated the expression classification into variables, weak head normal forms, and so on.

Once this implementation is stable, this versatile interpreter is to be used for analyzing computations obtained by using different language design options. These analysis will be based on the measures mentioned above and on the computations themselves. The comparisons will focus on the efficiency, the duplication of work, the amount of speculation, the termination of computations, etc. Afterwards, the language will be extended with other features of Eden such as communication via streams, and nondeterminism.

References

1. C. Baker-Finch, D. King, and P. Trinder. An operational semantics for parallel lazy evaluation. In *ACM-SIGPLAN International Conference on Functional Programming (ICFP'00)*, pp. 162–173, 2000.
2. A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236:35–132, 2000.
3. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *Maude Manual (Version 2.2)*, 2005. `http://maude.cs.uiuc.edu/manual`.
4. M. Hidalgo-Herrero. *Semánticas formales para un lenguaje funcional paralelo.* PhD thesis, Universidad Complutense de Madrid, 2004.
5. M. Hidalgo-Herrero and Y. Ortega-Mallén. An operational semantics for the parallel language Eden. *Parallel Processing Letters*, 12(2):211–228, 2002.
6. M. Hidalgo-Herrero, A. Verdejo, and Y. Ortega-Mallén. Looking for Eden through Maude and its strategies. Web page `http://maude.sip.ucm.es/eden`, 2006.
7. R. Loogen, Y. Ortega-Mallén, and R. Peña-Marí. Parallel functional programming in Eden. *Journal of Functional Programming*, 15(3):431–445, 2005.
8. N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. In D. M. Gabbay and F. Guenthner, eds., *Handbook of Philosophical Logic, Second Edition, Volume 9*, pp. 1–87. Kluwer, 2002.
9. N. Martí-Oliet, J. Meseguer, and A. Verdejo. Towards a strategy language for Maude. In N. Martí-Oliet, ed., *Proc. Fifth Int. Workshop on Rewriting Logic and its Applications, WRLA 2004*, ENTCS 117, pp. 417–441. Elsevier, 2004.
10. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
11. S. Peyton Jones. *Haskell 98 language and libraries: the Revised Report.* Cambridge University Press, 2003.
12. F. Rosa-Velardo, C. Segura, and A. Verdejo. Typed mobile ambients in Maude. In H. Cirstea and N. Martí-Oliet, eds., *Proc. 6th Int. Workshop on Rule-Based Programming, RULE 2005*, ENTCS 147, pp. 135–161. Elsevier, 2006.
13. A. Verdejo and N. Martí-Oliet. Implementing CCS in Maude 2. In F. Gadducci and U. Montanari, eds., *Proc. Fourth Int. Workshop on Rewriting Logic and its Applications, WRLA 2002*, ENTCS 71, pp. 239–257. Elsevier, 2002.

New Evaluation Strategies for Functional Languages [⋆]

Horatiu Cirstea[1], Germain Faure[1], Maribel Fernández[2],
Ian Mackie[3,⋆⋆], and François-Régis Sinot[3,⋆⋆]

[1] LORIA, BP 239 54506 Vandoeuvre-lès-Nancy Cedex, France
[2] Dept. of Computer Science, King's College London Strand, London WC2R 2LS, UK
[3] LIX, École Polytechnique, 91128 Palaiseau, France

Abstract. We use the ρ-calculus as an intermediate language to compile functional languages with pattern-matching features, and give an interaction net encoding of the ρ-terms arising from the compilation. This encoding gives rise to new strategies of evaluation, where pattern-matching and 'traditional' β-reduction can proceed in parallel without overheads.

1 Introduction

The λ-calculus is usually put forward as the abstract computational model underlying functional programming, and graph rewriting or environment machines are used to describe evaluation strategies and to derive concrete implementations (see for instance [26]). However, modern functional programming languages have pattern-matching features which cannot be directly expressed in the λ-calculus. To palliate this problem, pattern-calculi [23, 22, 4, 6, 8, 13] have been introduced. The ρ-calculus [6, 8] is a pattern calculus combining the expressiveness of pure functional calculi and algebraic term rewriting. It is an extension of the λ-calculus where we can abstract on patterns, not just on variables: *abstractions* are written $(p \twoheadrightarrow t)$ where p is a pattern and t is the body. The rule describing the dynamics of application introduces a *matching constraint*:

$$(\rho) \quad (p \twoheadrightarrow t)\ u \to [p \ll u]t$$

and the (σ) rule solves this constraint and applies the matching solution $\sigma_{p \ll u}$ to t.

$$(\sigma) \quad [p \ll u]t \to \sigma_{p \ll u}(t)$$

⋆ Partially funded by the Alliance France-UK project "Implementation Techniques for the Rho Calculus".

⋆⋆ Projet Logical, Pôle Commun de Recherche en Informatique du plateau de Saclay, CNRS, École Polytechnique, INRIA, Université Paris-Sud.

The ρ-calculus is parametric in the matching theory: we can use syntactic matching or any arbitrary matching theory, even without unique principal solutions.In the latter case, we can use a *structure* to deal with the multiple solutions.

As an intermediate language for the compilation of functional languages, the ρ-calculus has several advantages: patterns are an integral part of the framework, which allows us to reason about pattern-matching and to study the interaction between pattern-matching and β-reduction at an abstract level; and the ρ-calculus can be used to model not only functional behaviour but also imperative features [17], object-oriented features [7], etc.

In this paper we exploit the first point above: we use the ρ-calculus as an intermediate language to compile functional languages with pattern-matching features, and adapt the evaluation strategies developed for the ρ-calculus to the specific constraints arising from typed functional programs. We then use interaction nets to define and implement the evaluation strategies. This methodology gives rise to new, efficient strategies of evaluation for functional languages, which we describe below.

In [11] we defined two alternative encodings of the ρ-calculus in interaction nets [15]. Interaction nets are graph rewrite systems which have been used for the implementation of efficient reduction strategies for the λ-calculus [12, 1, 20]. Since interactions are local and strongly confluent, they can take place in any order, even in parallel (see [24]), which makes interaction nets well-suited for the implementation of programming languages [10]. The first encoding of the ρ-calculus in interaction nets given in [11] is simple and exploits the implicit parallelism of rules (ρ) and (σ): a term t with a matching constraint (generated by an application of ρ) can be applied to another term (again using ρ) while the matching constraint is being solved. However, this *simple encoding* can only model a *strict* semantics (see [8]) where a ρ-calculus term with a blocked matching evaluates to $\perp$ (fail). The second encoding of [11], which introduces a matching agent and will be called the *explicit encoding*, can implement either a strict or a non-strict semantics, but it looses parallelism.

In the case of *typed* functional languages with pattern-matching, the ρ-terms arising from the compilation of programs do not remain blocked. More precisely, a matching failure may occur only if the definitions by pattern-matching are non-exhaustive. We will show that, in this case, a combination of the simple interaction net encoding and the explicit encoding provides an implementation where pattern-matching and 'traditional' β-reduction can proceed in parallel, without additional overheads. For example, if we have a function with two branches (patterns), say `cons x nil` and `nil`, and the argument is a `cons`, this will compile into a net which, after selecting the `cons` branch, will check that the nested `nil` pattern matches while the substitution for `x` is being performed. We give more examples in Section 4. This is the main contribution of this paper: indeed, the compilation of functional programs in the ρ-calculus, and the subsequent interaction net encoding, uncover a new strategy of evaluation which naturally exploits the implicit parallelism of the ρ and σ rules.

This paper is organised as follows: after giving some background (Section 2), in Section 3 we define a minimalistic functional language, and give a compilation into the ρ-calculus. Section 4 shows an interaction net encoding for this intermediate language. We conclude in Section 5.

2 Background

We assume familiarity with the λ-calculus [2], and start with a short presentation of the ρ-calculus; for more details see [6, 8, 3]. We write $x, y, \ldots$ for variables and $f, g, \ldots$ for constants. The set of *ρ-terms* (or just terms, ranged over by t, u, v) $\mathcal{T}$ is defined by:

$$t, u ::= x \mid f \mid p \twoheadrightarrow t \mid [p \ll u]t \mid (t\ u) \mid \langle t, u \rangle$$

where $\mathcal{P}$ is an arbitrary subset of $\mathcal{T}$ ($p \in \mathcal{P}$ are called *patterns*); $p \twoheadrightarrow t$ is a *generalised abstraction* (it can be seen either as a λ-abstraction on a pattern p instead of a single variable, or as a standard term rewriting rule); $[p \ll u]t$ is a *delayed matching constraint* denoting a matching problem $p \ll u$ whose solutions (if any) will be applied to t; $(t\ u)$ denotes an *application* (we omit brackets whenever possible, and associate to the left); and finally, $\langle t, u \rangle$ is called a *structure*. Terms are always considered modulo α-conversion (later this will be realised for free in interaction nets).

As usual substitutions are mappings from variables to terms, with finite domain, written $\{x_1 := t_1, \ldots, x_n := t_n\}$. We write substitutions postfix: $t\sigma$ denotes the term obtained by applying the substitution σ to t.

The ρ-calculus is parameterised by the set $\mathcal{P}$ of patterns. Here we use *linear* (i.e., each variable occurs at most once) *algebraic patterns*: $p ::= x \mid f\ p_1 \ldots p_n$

Example 1. The boolean function `null` that tests if its argument is the empty list can be defined in the ρ-calculus as follows:

$$\texttt{null} = l \twoheadrightarrow (\langle Nil \twoheadrightarrow True,\ Cons\, x\, y \twoheadrightarrow False \rangle\ l)$$

The following reduction rules give the dynamics of the calculus. We write the reduction $\to_i$ (for implicit) or simply $\to$ when there is no risk of confusion:

$$\begin{array}{llcl} (\rho) & (p \twoheadrightarrow t)\ u & \to & [p \ll u]t \\ (\sigma) & [p \ll u]t & \to & t\sigma_{p \ll u} \\ (\delta) & \langle t, u \rangle\ v & \to & \langle t\ v, u\ v \rangle \end{array}$$

The rule (σ) asks for an *external* matching algorithm to find a solution of the matching of p with u, and applies the corresponding substitution to t. In this paper we assume linear syntactic matching; under this assumption the calculus is confluent [8].

In order to implement the ρ-calculus we need to make explicit the specification of the matching algorithm. We recall the explicit ρ-calculus of [11] (see also [5]), and extend it with rules to customise structures. Substitution will remain implicit, but we introduce an explicit application symbol $\bullet$ in patterns.

We write reduction in the explicit ρ-calculus $\to_x$ (for explicit) or simply $\to$ when there is no risk of confusion.

The rule (ρ) remains unchanged. We can decompose the rule (σ) into a finite set of local rules:

$$\begin{array}{lrcl}
(a_c) & f\ t & \to & f \bullet t \\
(a_a) & (t \bullet u)\ v & \to & (t \bullet u) \bullet v \\
(\sigma_a) & [(p \bullet r) \ll (u \bullet v)]t & \to & [p \ll u][r \ll v]t \\
(\sigma_c) & [f \ll f]t & \to & t \\
(\sigma_v) & [x \ll u]t & \to & t\{x := u\}
\end{array}$$

A matching problem $(p \ll u)$ may have no solution; this is called a *blocked matching*. We add rules to detect failure (i.e., a clash):

$$\begin{array}{lrcll}
(\bot_1) & [f \ll g]t & \to & \bot & \text{if } f \neq g \\
(\bot_2) & [f \ll (u \bullet v)]t & \to & \bot & \\
(\bot_3) & [f \ll (p \twoheadrightarrow u)]t & \to & \bot & \\
(\bot_4) & [(u \bullet v) \ll f]t & \to & \bot & \\
(\bot_5) & [(u \bullet v) \ll (p \twoheadrightarrow s)]t & \to & \bot &
\end{array}$$

and rules to propagate $\bot$. There are mainly two options:

1. Strict Semantics: $\quad (\mathit{strict}) \quad C[\bot] \quad \to \quad \bot \quad$ for any context $C[\cdot]$
   This rule corresponds to an exception-like semantics of matching failure, as in ML (e.g., even if the argument of an application is not used by the function, the result is $\bot$). In this semantics, a higher priority is given to this rule than to any other applicable rule (i.e., this rule is tried before the others).
2. Non-Strict Semantics: The rule *(strict)* defined above can be weakened to a particular class $\mathcal{C}$ of strict contexts (for instance, $\mathcal{C} = \{([\,]\ t), t \in \mathcal{T}\}$):

$$(\mathit{non\text{-}strict}) \quad C[\bot] \quad \to \quad \bot \qquad \text{for any } C[\cdot] \in \mathcal{C}$$

We now turn our attention to structures. Since we will focus on ρ-terms arising from functional programs, structures will only be created by the compilation of a function defined by cases. Hence, structures will have the form $\langle p_1 \twoheadrightarrow t_1, \ldots, p_n \twoheadrightarrow t_n \rangle$. Using (δ) and (ρ), an application of such structure to an argument u produces $\langle [p_1 \ll u]t_1, \ldots, [p_n \ll u]t_n \rangle$ where only one branch will succeed. In our equational theory for structures $\bot$ should be a neutral element. This is achieved by the rules:

$$\begin{array}{lrcll}
(\mathit{stk}) & \langle t_1, \ldots, t_{i-1}, \bot, t_{i+1}, \ldots, t_n \rangle & \to & \langle t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_n \rangle & 1 \leq i \leq n \\
(\mathit{singleton}) & \langle t \rangle & \to & t &
\end{array}$$

The rule (stk) was used previously (see [27, 9]) to encode term rewriting systems in the ρ-calculus. We could be more specific and force evaluation from left to right for instance, but we prefer not to fix the strategy of evaluation yet.

Notice that a naive implementation of (δ) would copy the argument u, which is inefficient. Since our use of structures will be limited to the compilation of case constructs in typed programs, we will actually be able to use the information provided by the type system to avoid copying the argument, thus optimising the reduction of structures (see Section 4.4).

Example 2 (Fixpoints). A fixpoint operator is a term Y such that for all terms t, $Y\,t \to^* t\,(Y\,t)$. It is easy to check that the following terms are fixpoint operators (the second has the advantage of being well-typed [27]):

- $Y_T = (y \twoheadrightarrow x \twoheadrightarrow x\,(y\,y\,x))\,(y \twoheadrightarrow x \twoheadrightarrow x\,(y\,y\,x))$
- $Y_{rec} = x \twoheadrightarrow ((z \twoheadrightarrow z\,(rec\,z))\,(rec\,f \twoheadrightarrow (x\,(f\,(rec\,f)))))$ where rec is a constant.

3 From a functional language to the $\boldsymbol{\rho}$-calculus

We consider a simple functional language with terms built from variables $x, y, \ldots$, functional abstraction, application, data constructors C (each with a fixed arity), and a case construct to define functions by pattern-matching on constructors. We abbreviate $t_1, \ldots, t_n$ as $\boldsymbol{t}$. Patterns are defined by the following grammar:

$$p ::= x \mid C(\boldsymbol{p})$$

with the usual linearity constraint (each variable may occur at most once in a pattern). The syntax of terms is given by the grammar:

$$\begin{aligned} t, u ::= {} & x \mid \texttt{fn}\; x.t \mid t\; u \mid C(\boldsymbol{t}) \\ & \mid \texttt{case}\; t\; \texttt{of}\; (p_i \rightsquigarrow u_i)_{i \in I} \\ & \mid \texttt{fix}(\texttt{fn}\; f.t) \end{aligned}$$

A `case` branch of the form $(p_i \rightsquigarrow \cdot)$ acts as a binder i.e., $\mathsf{fv}(p_i \rightsquigarrow u_i) = \mathsf{fv}(u_i) \setminus \mathsf{fv}(p_i)$ where $\mathsf{fv}(u_i)$ denotes the set of free variables of u_i.

We assume the language is typed. For simplicity, we consider a simply-typed system where each constructor is associated to a datatype. We will base this discussion on the following form of a datatype declaration, which introduces a datatype DT with constructors $C_1, \ldots, C_n$, using some predefined types $\boldsymbol{\alpha_i}$.

$$DT = C_1(\boldsymbol{\alpha_1}) \mid \cdots \mid C_n(\boldsymbol{\alpha_n})$$

Example 3. In the sequel we will use the following datatypes for numbers and lists with elements of type α:

$$\begin{aligned} Int &= Z \mid S(Int) \\ List\ \alpha &= Nil \mid Cons(\alpha, List\ \alpha) \end{aligned}$$

As usual, the type system ensures that in a case construct $\texttt{case}\ t\ \texttt{of}\ (p_i \rightsquigarrow u_i)_{i\in I}$ all the branches have the same type and t has the same type as the patterns p_i (for all $i \in I$), that is, some datatype DT. We do not assume that the cases are exhaustive, but we do assume they are non-overlapping for simplicity. We use a strict matching semantics, as in ML (i.e., an application of a function to an argument that is not covered by the case definition will produce a runtime error). We omit the typing rules, which are standard.

The dynamics of the language is given by the following reduction rules (reduction is denoted by $\rightarrow_f$ or simply $\rightarrow$) where $\{x := u\}$ denotes the substitution of x by u.

$$\begin{aligned}(\texttt{fn}\ x.t)\ u &\rightarrow t\{x := u\}\\ \texttt{case}\ t\ \texttt{of}\ (p_i \rightsquigarrow u_i)_{i\in I} &\rightarrow u_k\ \sigma \qquad \text{(if } t \text{ matches } p_k \text{ with substitution } \sigma)\\ \texttt{fix}(\texttt{fn}\ f.t) &\rightarrow (\texttt{fn}\ f.t)\ \texttt{fix}(\texttt{fn}\ f.t)\end{aligned}$$

Since the rewrite rules are left-linear and non-overlapping (that is, they define an orthogonal system [14]), the language is confluent. It is easy to see that it is not terminating, due to the presence of the fixpoint operator `fix`.

Programs in this language are well-typed, closed terms (i.e., terms with no free variables). We give now some simple examples.

Example 4. 1. Assuming that Nil with arity 0, and $Cons$ with arity 2, are used to define the datatype $List$ as in Example 3, and that $True$ and $False$ are the boolean constants, we can define the boolean function `null` by pattern-matching as follows:
$\texttt{null} \triangleq \texttt{fn}\ l.\texttt{case}\ l\ \texttt{of}\ (Nil \rightsquigarrow True, Cons(x,y) \rightsquigarrow False)$

2. Assuming that Z with arity 0, and S with arity 1 are used to define the datatype Int as in Example 3, the recursive function `length` can be defined by pattern-matching as follows:
$\texttt{length} \triangleq \texttt{fix}(\texttt{fn}\ len.\texttt{fn}\ l.\texttt{case}\ l\ \texttt{of}\ (Nil \rightsquigarrow Z, Cons(x,y) \rightsquigarrow S(len\ y)))$

Notice that we have not included a conditional in the syntax of the language, but it can be easily encoded with a `case` over the booleans $True$, $False$. Also, we do not have named functions and `letrec` but these can be easily encoded using `fix`.

3.1 Compilation

The following compilation function, defined by induction on terms, translates terms in the typed functional language into the ρ-calculus:

$$\begin{aligned}[\![x]\!] &= x\\ [\![\texttt{fn}\ x.t]\!] &= (x \twoheadrightarrow [\![t]\!])\\ [\![t\ u]\!] &= [\![t]\!]\ [\![u]\!]\\ [\![C(t_1,\dots,t_n)]\!] &= C\ [\![t_1]\!]\dots[\![t_n]\!]\\ [\![\texttt{case}\ t\ \texttt{of}\ (p_1 \rightsquigarrow u_1,\dots,p_n \rightsquigarrow u_n)]\!] &= \langle [\![p_1]\!] \twoheadrightarrow [\![u_1]\!],\dots,[\![p_n]\!] \twoheadrightarrow [\![u_n]\!]\rangle\ [\![t]\!]\\ [\![\texttt{fix}(\texttt{fn}\ f.t)]\!] &= Y[\![\texttt{fn}\ f.t]\!]\end{aligned}$$

where Y is a fixpoint operator of the explicit ρ-calculus (see Example 2). We leave Y abstract because it is an implementation choice. In particular, this will enable us to use a more efficient translation into interaction nets.

Example 5. We can check that the compilation of the function `null` defined in Example 4 gives the function `null` in the ρ-calculus as given in Example 1.

Note that case constructs are compiled into structures applied to an argument, and can be reduced using the δ rule. The interaction net encoding will ensure that $[\![t]\!]$ is not copied, and moreover it will allow matching to be carried in parallel with other reductions, if possible.

We define the compilation of $\sigma = \{x_1 := u_1, \ldots, x_n := u_n\}$ to be the substitution $[\![\sigma]\!] = \{x_1 := [\![u_1]\!], \ldots, x_n := [\![u_n]\!]\}$.

We now state some soundness invariants.

Proposition 6. *1. For all terms t and all substitutions σ, $[\![t\sigma]\!] = [\![t]\!][\![\sigma]\!]$.*

2. For all patterns p and all terms u:
   - *The matching problem $p \ll t$ has a solution iff the matching problem $[\![p]\!] \ll [\![t]\!]$ has a solution.*
   - *The substitution σ is a solution of the matching problem $p \ll t$ iff the substitution $[\![\sigma]\!]$ is a solution of the matching problem $[\![p]\!] \ll [\![t]\!]$.*

3. For all terms t and u, if $t \to_f u$, then $[\![t]\!] \to_x^* [\![u]\!]$.

One can notice that in the ρ-calculus the granularity of the reduction is finer than in the chosen functional language and thus, the intermediate ρ-terms obtained during the reduction of the translation of a program t do not necessarily correspond to a program. More precisely, for a reduction $[\![t]\!] \to_x u$ we cannot always exhibit a term u' such that $t \to_f u'$ and $[\![u']\!] = u$. Nevertheless, if the reduction of the term u continues then the term $[\![u']\!]$ is eventually reached.

Lemma 7. *For all programs t and for all terms u such that $[\![t]\!] \to_x u$ there exists a program v such that $u \to_x^* [\![v]\!]$ and $t \to_f^* v$.*

The following proposition is a corollary of the previous results.

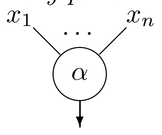
Proposition 8 (Correctness). *Let t be a program and v be a normal form, then $t \to_f^* v$ iff $[\![t]\!] \to_x^* [\![v]\!]$.*

In the following section we give an interaction net implementation for the functional language defined above, which defines a strategy of evaluation based on the encodings of the ρ-calculus presented in [11] and the coding of datatypes discussed in [21]. Although we focus on implementation in this paper, the intermediate ρ-calculus compilation has also interesting applications for programming language design (for instance one could study the properties of a more general language including non-linear patterns, or non-syntactic matching theories) and could also be used to study program transformations (in the same way as, for instance, explicit substitution calculi) and to prove correctness of program optimisations.

4 Interaction Net Encoding

4.1 Preliminaries

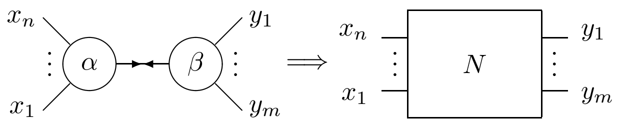
A system of interaction nets is specified by a set Σ of symbols with fixed arities, and a set $\mathcal{R}$ of interaction rules. An occurrence of a symbol $\alpha \in \Sigma$ is called an *agent*. If the arity of α is n, then the agent has $n+1$ *ports*: a *principal port* depicted by an arrow, and n *auxiliary ports*.



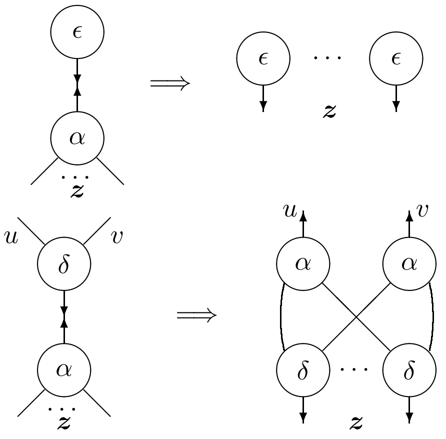
Intuitively, a net N is a graph (not necessarily connected) with agents at the vertices and each edge connecting at most 2 ports. The ports that are not connected to another agent are *free*. There are two special instances of a net: a wiring (no agents) and the empty net; the extremes of wirings are also called free ports. The interface of a net is its set of free ports.

An interaction rule $((\alpha, \beta) \Longrightarrow N) \in \mathcal{R}$ replaces a pair of agents $(\alpha, \beta) \in \Sigma \times \Sigma$ connected together on their principal ports (an *active pair* or *redex*) by a net N with the same interface. Reduction is local, and there may be at most one rule for each pair of agents.

The following diagram shows the format of interaction rules (N can be any net built from Σ).



We show as an example the interaction rules of two ubiquitous agents, namely the *erase* (ϵ), of arity 0, which deletes everything it interacts with, and the *duplicator* (δ), of arity 2, which copies everything. These are represented by the following diagrams, where α is any node. We refer to [15] for more details and examples.

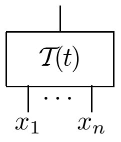


We use the notation $\Longrightarrow$ for the one-step reduction relation and $\Longrightarrow^*$ for its transitive and reflexive closure. If a net does not contain any active pairs then it is in normal form. The key property of interaction nets, besides locality of reduction, is strong confluence. There are several implementations of interaction nets, see for instance [16] and [25]; the latter has been designed to take advantage of additional processors, thus giving a parallel implementation of interaction nets.

4.2 Implementing the Language

We will assume that the problems of binding and substitution can be solved as in any off-the-shelf interaction net encoding of the λ-calculus (see for instance [18, 19]), and concentrate on the encoding of the explicit matching and structure rules given in Section 2. This methodology is justified by the fact that the terms $p \twoheadrightarrow t$ and $x \twoheadrightarrow [p \ll x]t$ are extensionally equivalent, so that we can safely precompile terms in order to abstract only on variables, as in the λ-calculus, and have explicit matching constraints from the beginning. Also, there is a standard, efficient way to encode recursion in interaction nets for the λ-calculus, which consists of building a cyclic structure which explicitly "ties the knot". The idea corresponds exactly to an encoding of recursion in graph reduction (see Peyton Jones [23] for instance), and was adapted to interaction nets in [18]. We use this for the encoding of Y (see [20] for details).

We now define by induction a function $\mathcal{T}(\cdot)$ to translate the ρ-terms arising from the compilation of functional programs into interaction nets, and we give the interaction rules that will be used to evaluate them. As in the *simple* interaction net encoding of the ρ-calculus described in [11], a ρ-term t with free variables $\mathsf{fv}(t) = \{x_1, \ldots, x_n\}$ will be translated to a net $\mathcal{T}(t)$ with the root edge at the top, and n free edges corresponding to the free variables.



Variable: If t is a variable then $\mathcal{T}(t)$ is just a wire.

Constant: For each constant f we introduce an agent as shown in Figure 1 (left).

Matching Constraint: A term of the form $[p \ll u]t$ is encoded as shown in Figure 1 (right)[4] which can be interpreted as the substitution in t of the (possible) solution of the matching (the left subnet corresponds to the matching problem $p \ll u$).

Structure: We will discuss the encoding of structures at the end of the section.

Abstraction: We assume that terms have been precompiled to abstract only on variables, as described above; hence we can reuse the abstraction of the λ-calculus.

[4] A dashed edge represents a bunch of edges (a *bus*).

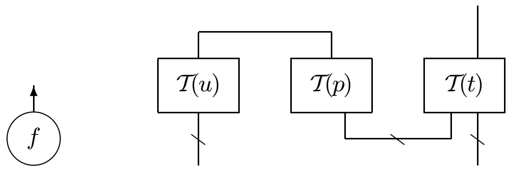


Fig. 1. Translation of constants (left) and matching constraints (right).

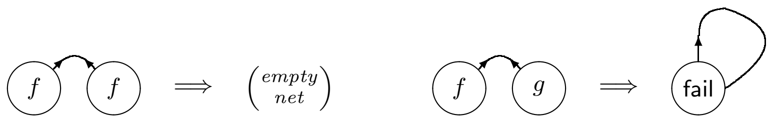


Fig. 2. Matching of constants (success and failure)

Application: Similarly for application, we introduce an agent @ with its principal port oriented towards the left subterm, so that interaction with an abstraction is possible. To implement the rule (ρ), we define an interaction rule between abstraction and application as in the λ-calculus (see for instance [19]).

4.3 Matching Rules

The matching rules are inspired by the "simple" encoding of [11]. Assume we have just one matching constraint to solve (the general case of a structure with multiple branches will be treated below). The matching algorithm is initiated by connecting the root of a pattern with the term to match. Thus, the rule (σ_v) (matching against a variable) is realised for free, as in the λ-calculus. To simulate (σ_a) and ($\perp_1$), constants will interact: Two identical constants cancel each other to give the empty net, as indicated in Figure 2 (left). If the agents are not the same, then we introduce an agent fail, which represents a failure in the matching algorithm, as indicated in Figure 2 (right). We interpret a net containing an agent fail as an overall failure, thus implementing the strict matching semantics.

We also need interaction rules to convert a usual application (@) into a pattern application (•) when it is part of an algebraic pattern (or term), these are shown in Figure 3; and a rule to match applications, which is given in Figure 4, as well as interaction rules corresponding to the rules ($\perp_2$) and ($\perp_4$) which we omit. We do not need interaction rules corresponding to ($\perp_3$) and ($\perp_5$) since the language is typed.

We refer to [11] for a detailed description and correctness proofs for matching constraints. In particular, in [11] it is shown that with this encoding of matching we can only implement a strict ρ-calculus semantics, but, on the positive side, it allows us to obtain a strategy of evaluation with a good potential for parallelism.

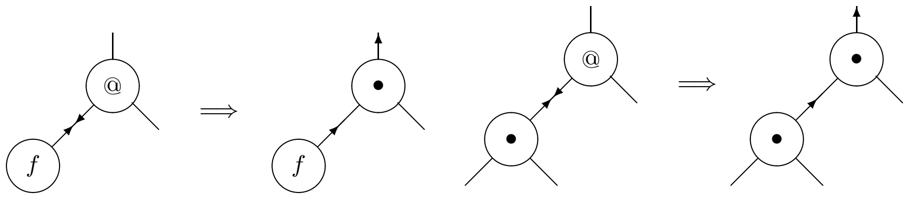


Fig. 3. Rules to transform patterns

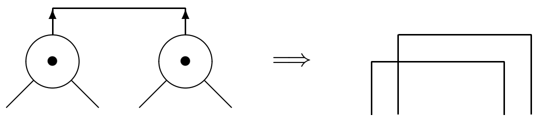


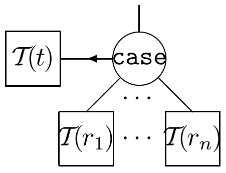
Fig. 4. Matching applications

This is because matching interactions involving the constraint associated to an abstraction can take place in parallel with a traditional β/ρ reduction involving the same abstraction, without introducing any 'administrative' agents (i.e., no overheads). We will use this feature in the encoding of functional programs below, to derive an evaluation strategy with the same potential for parallelism. We give examples at the end of the section.

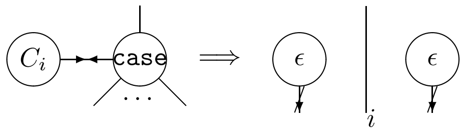
4.4 Structures

We will now describe the encoding of structures and the rule (δ). First remark that structures only arise from the compilation of `case` constructs, more precisely, structures can only occur in subterms of the form: $\langle l_1 \twoheadrightarrow r_1, \ldots, l_n \twoheadrightarrow r_n\rangle\ t$. The goal is to avoid making multiple copies of t in the implementation of (δ), and to permit matching to proceed in parallel with functional computation, whenever possible. For these reasons, we will not treat these terms as standard applications. Instead, for each structure $\langle l_1 \twoheadrightarrow r_1, \ldots, l_n \twoheadrightarrow r_n\rangle\ t$ (with $n > 1$; if $n = 1$ we can treat it as an abstraction) occurring in the compilation of a program we will introduce an agent `case` as explained below, where we build a net that minimises the number of selections necessary. To keep the diagrams simple, we show the compilation in stages.

First, we consider the case when each l_i is a different constant. We can then encode the structure using a simple case agent as follows:

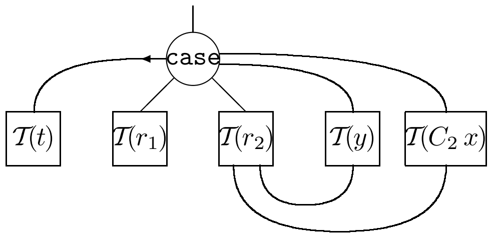


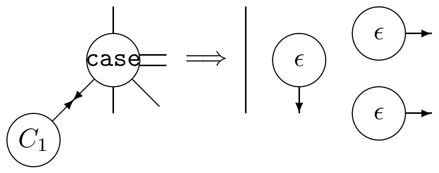
with the following collection of rules which select the appropriate branch of the case, and erase all other options using ϵ agents.

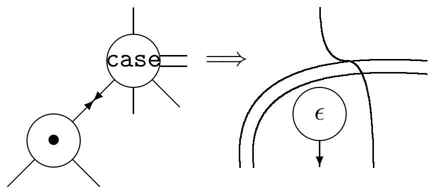


The top auxiliary port of a `case` agent represents the output; the interaction rule above selects the branch i corresponding to the constructor C_i and connects it to the output port (all other branches are erased). It is a straightforward exercise to verify that this indeed mimics the corresponding reduction rule. Note that we are assuming that all patterns are disjoint (non-overlapping) but they may be non-exhaustive. Note also that garbage collection (using ϵ) is explicit in interaction nets, since interaction rules must preserve the interface of the net.

Next we deal with deeper patterns, including variables. To give the idea we consider the case where there is just one pattern of depth greater than 1 (i.e., the root is an application), for instance: $\langle C_1 \twoheadrightarrow r_1, C_2\ x\ y \twoheadrightarrow r_2\rangle\ t$. The compilation and interaction rules are as follows:



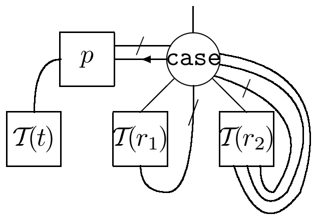




Again the top auxiliary port of `case` is the output; the first rule above corresponds to a pattern of depth 1 (as before). Note that in the second interaction rule, the right hand side has a wire to connect the net $\mathcal{T}(r_2)$ to the output port of the `case` agent. In the compilation, the nets $\mathcal{T}(y)$ and $\mathcal{T}(C_2\ x)$ are there precisely to complete the pattern matching, even though the branch would have already been selected. Any resulting substitutions generated are connected to the free variables of $\mathcal{T}(r_2)$. The extension to the case where there are more constant patterns is straightforward.

Next we examine the case when there are more than one application cases to consider. Again, to keep the diagrams simple, we will concentrate on this aspect, and ignore the patterns of depth one that were given previously (extra ports in the `case` agent would be needed). Consider the example: $\langle C_1 x \twoheadrightarrow r_1, C_2\ y\ z \twoheadrightarrow r_2\rangle\ t$. Both patterns have an application at the root, so we cannot use a case to distinguish them. However, we can identify where the patterns disagree, and consume that part before using a case agent, as above. Once the common prefix has been consumed, then we are left with a situation which is exactly as explained in the previous case (i.e., constant and an application).

The following is the compilation, where the net p is the common prefix, with principal ports pointing towards $\mathcal{T}(t)$ (so the rules in Subsection 4.3 will apply). In the diagram below we assume that there is nothing else to the pattern, as this situation has already been dealt with previously.



The interaction rules are now identical to the ones previously given for the case agent, except that in addition we must connect the additional bindings to the correct branch. This completes the encoding of structures, which requires the combination of the above features. In addition, when the terms r_i have common free variables we must use extra agents to allow these variables to be shared. The compilation for such a feature is standard and will be omitted here; we refer the reader to [21] for details.

Pattern matching is slightly more efficient if we use an alternative encoding for patterns, where a constructor of arity n in the functional language is represented by an agent of arity n (instead of a 0-ary constant). This has the advantage of avoiding interactions between case agents and the algebraic application agent.

At the end of the section we give an example showing how the encoding of the ρ-calculus used here allows us to exploit the implicit parallelism between matching and functional computations.

4.5 The Parallel Strategy at Work

To illustrate the potential for parallelism, we give an example using a variant of the Ackermann function on coloured trees, which is based on the datatype:

$$Tree = Nil \mid Red(Int, Tree, Tree) \mid Black(Int, Tree, Tree)$$

Let `ack` be the Ackermann function. The function `ackt` takes two trees and computes a new tree where the nodes contain integers obtained by applying `ack` to the corresponding nodes of the arguments, but only when the trees have the same alternating colours. It is defined in our functional language as follows.

```
ackt ≜ fix(fn ackt.fn t_1.fn t_2.case t_1 of
        (
        Red(x_1, Black(y_1, t_L, t_R), Black(z_1, s_L, s_R)) ⇝
          case t_2 of
            (
            Red(x_2, Black(y_2, t'_L, t'_R), Black(z_2, s'_L, s'_R)) ⇝
                Red(ack(2 * x_1, x_2),
                    ackt(Black(y_1, t_L, t_R), Black(y_2, t'_L, t'_R)),
                    ackt(Black(z_1, s_L, s_R), Black(z_2, s'_L, s'_R)))
            )

        Black(x_1, Red(y_1, t_L, t_R), Red(z_1, s_L, s_R)) ⇝
          case t_2 of
            (
            Black(x_2, Red(y_2, t'_L, t'_R), Red(z_2, s'_L, s'_R)) ⇝
                Black(ack(2 + x_1, x_2),
                    ackt(Red(y_1, t_L, t_R), Red(y_2, t'_L, t'_R)),
                    ackt(Red(z_1, s_L, s_R), Red(z_2, s'_L, s'_R)))
            )
        )
```

The compilation of this program in the ρ-calculus and subsequent encoding in interaction nets produces a net with an active pair between the agent case representing the first case in the program and the agent *Red*.

After the interaction between the first case agent and *Red*, the actual value of x_1 gets connected to the multiplication agent in the first branch of the case, so that we can start computing $2 * x_1$ in parallel with the rest of the matching. Then, after the interaction between the second case agent and *Red*, we also get the value of x_2 connected to the net representing the Ackermann function and we can then compute in parallel the value of $\mathtt{ack}(2 * x_1, x_2)$ while the rest of the pattern (i.e., Black, Black) is checked.

5 Conclusion

We have proposed to use the ρ-calculus as an alternative foundation for functional programming languages, and provided a compilation of a simple functional programming language into the ρ-calculus. This calculus is better adapted than the λ-calculus for representing features, specifically pattern matching, of functional languages. One of the main features of our compilation is that we can experiment with different pattern-matching algorithms and matching strategies, in a modular way. We have thus a powerful formalism for programming language design, and for reasoning about functional program implementation. Using this as an intermediate language, we have demonstrated that we can compile, also in a modular way, into interaction nets, and obtain new strategies of evaluation of programs with pattern-matching. Since the translation into interaction nets is modular, the strategy specified here can be combined with any β-reduction strategy, including an optimal one.

The interaction net encoding, although derived from a strategy of evaluation in the ρ-calculus, could of course be defined directly on the functional programs, without the intermediate compilation. We hope that the compilation into the ρ-calculus will allow us to transfer other results into the functional language (e.g. extensions to accommodate imperative features).

Acknowledgements: We thank Gilles Dowek for useful discussions on the subject of this paper.

References

1. A. Asperti, C. Giovannetti, and A. Naletto. The Bologna optimal higher-order machine. *Journal of Functional Programming*, 6(6):763–810, Nov. 1996.
2. H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* North-Holland, revised edition, 1984.
3. G. Barthe, H. Cirstea, C. Kirchner, and L. Liquori. Pure patterns type systems. In *Principles of Programming Languages - POPL2003, New Orleans, USA*. ACM, Jan. 2003.
4. V. Breazu-Tannen, D. Kesner, and L. Puel. A typed pattern calculus. In *Proceedings of the 8th Annual IEEE Symposium on Logic in Computer Science (LICS'93), Montréal, Canada*, 1993.
5. H. Cirstea, G. Faure, and C. Kirchner. A rho-calculus of explicit constraint application. In *Proceedings of the 5th workshop on rewriting logic and applications.* Electronic Notes in Theoretical Computer Science, 2004.
6. 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.
7. H. Cirstea, C. Kirchner, and L. Liquori. Matching Power. In A. Middeldorp, editor, *Proceedings of RTA'2001*, Lecture Notes in Computer Science, Utrecht (The Netherlands), May 2001. Springer-Verlag.
8. H. Cirstea, C. Kirchner, and L. Liquori. Rewriting calculus with(out) types. In F. Gadducci and U. Montanari, editors, *Proceedings of the fourth workshop on rewriting logic and applications*, Pisa (Italy), Sept. 2002. Electronic Notes in Theoretical Computer Science.
9. H. Cirstea, L. Liquori, and B. Wack. Rewriting calculus with fixpoints: Untyped and first-order systems. In *Post-proceedings of TYPES*, Lecture Notes in Computer Science. Springer-Verlag, 2003.
10. M. Fernández and I. Mackie. Interaction nets and term rewriting systems. *Theoretical Computer Science*, 190(1):3–39, January 1998.
11. M. Fernández, I. Mackie, and F.-R. Sinot. Interaction nets vs. the rho-calculus: Introducing bigraphical nets. In *Proceedings of EXPRESS'05, satellite workshop of Concur, San Francisco, USA, 2005*, Electronic Notes in Computer Science. Elsevier, 2005.
12. G. Gonthier, M. Abadi, and J.-J. Lévy. The geometry of optimal lambda reduction. In *Proceedings of the 19th ACM Symposium on Principles of Programming Languages (POPL'92)*, pages 15–26. ACM Press, Jan. 1992.

13. C. B. Jay and D. Kesner. Pure pattern calculus. In *Proceedings of the European Symposium on Programming (ESOP) LNCS 3924*, 2006.
14. J.-W. Klop, V. van Oostrom, and F. van Raamsdonk. Combinatory reduction systems, introduction and survey. *Theoretical Computer Science*, 121:279–308, 1993.
15. Y. Lafont. Interaction nets. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages (POPL'90)*, pages 95–108. ACM Press, Jan. 1990.
16. S. Lippi. in^2 : A graphical interpreter for interaction nets. In S. Tison, editor, *Rewriting Techniques and Applications (RTA'02)*, volume 2378 of *Lecture Notes in Computer Science*, pages 380–386. Springer, 2002.
17. L. Liquori. iRho: The Software (system demonstration). In *Proceedings of Developments in Computational Models (DCM'05, Lisbon, Portugal, 2005*, Electronic Notes in Theoretical Computer Science. Elsevier, 2005.
18. I. Mackie. *The Geometry of Implementation*. PhD thesis, Department of Computing, Imperial College of Science, Technology and Medicine, September 1994.
19. I. Mackie. YALE: Yet another lambda evaluator based on interaction nets. In *Proceedings of the 3rd International Conference on Functional Programming (ICFP'98)*, pages 117–128. ACM Press, 1998.
20. I. Mackie. Efficient λ-evaluation with interaction nets. In V. van Oostrom, editor, *Proceedings of the 15th International Conference on Rewriting Techniques and Applications (RTA'04)*, volume 3091 of *Lecture Notes in Computer Science*, pages 155–169. Springer-Verlag, June 2004.
21. I. Mackie. An interaction net implementation of additive and multiplicative structures. *Journal of Logic and Computation*, 15(2):219–237, April 2005.
22. V. Oostrom. Lambda calculus with patterns. Technical Report IR 228, Vrije Universiteit, Amsterdam, November 1990.
23. S. L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall International, 1987.
24. J. S. Pinto. Sequential and concurrent abstract machines for interaction nets. In J. Tiuryn, editor, *Proceedings of Foundations of Software Science and Computation Structures (FOSSACS)*, volume 1784 of *Lecture Notes in Computer Science*, pages 267–282. Springer-Verlag, 2000.
25. J. S. Pinto. Parallel evaluation of interaction nets with mpine. In A. Middeldorp, editor, *RTA*, volume 2051 of *Lecture Notes in Computer Science*, pages 353–356. Springer, 2001.
26. R. Plasmeijer and M. Eekelen. *Functional Programming and Parallel Graph Rewriting*. Addison Wesley, 1993.
27. B. Wack. *Typage et déduction dans le calcul de réécriture*. PhD thesis, Université Henri Poincaré - Nancy I, 2005.

Strategic Graph Rewriting

Transforming and Traversing Terms with References

Karl Trygve Kalleberg[1] and Eelco Visser[2]

[1] University of Bergen, Norway, `karltk@ii.uib.no`
[2] Utrecht University, The Netherlands, `visser@cs.uu.nl`

Abstract. Some transformations and many analyses on programs are either difficult or unnatural to express using terms. In particular, analyses that involve type contexts, call- or control flow graphs are not easily captured in term rewriting systems. In this paper, we describe an extension to the System S term rewriting system that adds references. We show how references are used for graph rewriting, how we can express more transformations with graph-like structures using only local matching, and how references give a representation that is more natural for structures that are inherently graph-like. Furthermore, we discuss trade-offs of this extension, such as changed traversal termination and unexpected impact of reference rebinding.

1 Introduction

Strategic programming is a powerful technique for program analysis and transformation that offers the separation of local data transformations from data traversal logic. The technique is independent of language paradigm and underlying data structure, but is perhaps most frequently found in functional and term rewriting languages. Much of its power comes from the ability to define complex traversal strategies from a small set of traversal combinators. Traversal strategies are often used to traverse terms.

Terms, when implemented as maximally shared, directed acyclic graphs, have many desirable properties for representing abstract syntax trees (ASTs) as used in program transformation. In the ATerm model [5], maximal sharing of subterms means that all occurrences of a term are represented by the same node in memory. Consequently copying of terms is achieved by copying pointers, and term equality entails pointer comparison. The model ensures *persistence*; modifying a term means creating a *new* term, the old term is still present. This makes it easy to support backtracking. Destructive updates of term references are not permitted, which allows efficient memory usage.

A consequence of the DAG representation is that terms cannot have explicit backlinks, i.e. *references* to arbitrary subterms elsewhere in the same term. Such references to other parts of the AST, including an ancestor of a term, are useful for expressing the results of semantic analyses as local information for rewrite rules. Turning ASTs into terms with references can turn some transformation problems from global-to-local into local-to-local. By adding explicit references in the terms, we arrive at a variation of term graphs [12]. The added expressivity gained from term references allows us to

Sixth Int'l Workshop on Reduction Strategies in Rewriting and Programming (WRS'06)
S. Antoy (ed.), Seattle, Washington, August 11, 2006
Portland State University, Tech. Report 06-04, pages 96-111

encode graphs rather succinctly, and therefore also express structures that are inherently graph-like more naturally, such as high-level program models and many intermediate compiler representations, including call-, control flow- and type graphs, thus setting the stage for implementing transformations such as constant and copy propagation, type checking and various static analyses. We conserve the ability to express program transformations using local rewrite rules together with generic traversal strategies, obtaining a form of strategic graph rewriting. By adding explicit references, we also give up some of the desirable properties of terms mentioned above, as the references allow destructive updates, change the termination criteria for traversals, alter the matching behavior of rewrite rules and exhibit side effects due to reference rebinding.

In this paper, we explore the design of a strategic rewriting language on terms with references, and discuss the tradeoffs found in this design space. We will show that we can arrive at a practical and useful variation of term graphs that allows us to express graph algorithms and rewriting problems rather precisely.

The paper is organized as follows: In Sec. 2, we introduce basic term rewriting with Stratego, show new primitives for manipulating references and how existing constructs extend to handle terms with references. In Sec. 3, we show how the new language features are used to compute various common graph representations for programs from ASTs. In Sec. 4, we implement two basic graph algorithms: depth first search and strongly connected components, and their application to finding sets of mutually recursive functions. We also discuss an implementation of lazy graph loading. In Sec. 5, we discuss notable aspects of our implementation. In Sec. 6, we discuss related work. In Sec. 7, we discuss design tradeoffs and future work. We conclude in Sec. 8.

2 Extending Term Rewriting Strategies to Term Graphs

We now present the extension of the strategic term rewriting language Stratego with term references. First, we give an overview of the basic concepts of Stratego. Next, we extend terms to term graphs, i.e. terms with references, by introducing primitives for references. Finally, we discuss rewrite rules and generic traversals on term graphs.

2.1 Term Rewriting Strategies

The Stratego program transformation language [2] is an implementation of the System S [14] core language for term rewriting. We will not discuss all the features of System S and Stratego in this paper, but restrict ourselves mainly to conditional rewrite rules, strategies, traversals, and scoped, dynamic rules.

Terms A term t is an application $c(t_1,\ldots,t_n)$ of a constructor c to zero or more terms t_i. There are some special forms of terms such as lists ($[t_1,\ldots,t_n]$) and integer constants, but these are essentially sugar for constructor terms. A term pattern is a term p with variables x, written on the form $p(x)$.

Rewrite Rules A conditional rewrite rule, R `:` $p_l(x)$ `->` $p_r(x)$ `where` c, is a rule named *R* that transforms the left-hand side pattern p_l to an instantiation of the right-hand side pattern p_r if the condition *c* holds. The following rule can be used to simplify addition expressions.

```
Simplify: Add(Int(x), Int(y)) -> z where <add> (x, y) => z
```

When applied to the term `Add(Int(1),Int(2))`, it will execute the condition `<add> (x,y)`, which is a strategy expression for computing the sum of two integers. The resulting term, 3, will be bound to the variable `z` using the operator `=>` as assignment.

Strategies Strategies are used to implement rewriting algorithms. A *strategy* is built from primitive traversals (`one(s)`, `all(s)`, `some(s)`), combinators (`s1 <+ s2` (left choice), `s1 ; s2` (strategy composition)), identity (`id`), failure (`fail`) and invocations of rewrite rules or other strategies. The following strategy will attempt to apply the rule `Simplify` to all subterms of the current term. If `Simplify` fails, either because the left hand side pattern does not match, or because the condition does not hold, the strategy `id` will be applied instead. `id` always succeeds, and returns the identity (i.e. same term).

```
try-simplify = all(Simplify <+ id)
```

When applied to `Sub(Add(Int(1),Int(2)),Int(4))`, the first subterm of `Sub` will be rewritten to `3`, but `Simplify` will fail for the second subterm (`Int(4)`), and `id` will be applied instead. The end result is the term `Sub(Int(3),Int(4))`.

Strategy Expression	Meaning
`!`$p(x)$	*(build)* Instantiate the term pattern $p(x)$ and make it the current term
`?`$p(x)$	*(match)* Match the term pattern $p(x)$ against the current term
s_0 `<+` s_1	*(left choice)* Apply s_0. If s_0 fails, roll back, then apply s_1
s_0 `;` s_1	*(composition)* Apply s_0, then apply s_1. Fail if either s_0 or s_1 fails
`rec` $x(s(x))$	*(recursion)* Strategy x may be called in s for recursive application
`id, fail`	*(identity, failure)* Always succeeds/fails. Current term is not modified
`one(`s`)`	Apply s to one direct subterm of the current term
`some(`s`)`	Apply s to as many direct subterms of the current term as possible
`all(`s`)`	Apply s to all direct subterms of the current subterm
`\`$p_l(x)$ `->` $p_r(x)$`\`	Anonymous rewrite rule from term pattern $p_l(x)$ to $p_r(x)$
`?`x`@`$p(y)$	Equivalent to `?`x `;` `?`$p(y)$; bind current term to x then match $p(y)$
`<`s`>` $p(x)$	Equivalent to `!`$p(x)$ `;` s; build $p(x)$ then apply s
s `=>` $p(x)$	Equivalent to s `;` `?`$p(x)$; match $p(x)$ on result of s

Generic Traversal Strategies The primitive traversals `one(s)`, `some(s)` and `all(s)` can be composed using the strategy combinators to obtain *generic traversal strategies*, which are used to control the order of rewrite rule applications throughout a term.

```
topdown(s)  = s ; all(topdown(s))
bottomup(s) = all(bottomup(s)) ; s
```

The strategy `topdown(s)` will apply the strategy `s` to the current term before recursively applying itself to `all` the subterms of the new current term. `bottomup(s)` works similarly: `s` is applied to all subterms before the current term (with freshly rewritten subterms) is processed by `s`.

Build and Match Pattern matching is also available independently of rewrite rules by the match operator strategy, written `?`. The expression `?Add(Int(`x`),Int(`y`))` when applied to the term `Add(Int(1),Int(2))`, will bind `x` to 1 and `y` to 2. The inverse operator of match is build, written `!`, which will instantiate a pattern. Given the previous bindings of `x` and `y`, `!Add(Int(`x`),Int(`y`))` will instantiate to `Add(Int(1),Int(2))`. Figures 1(a), and 1(b) show simple ground terms and their corresponding ATerms.

2.2 References

Thus far, the language we have presented only operates on plain terms. We now extend terms with *references*. That is, in addition to a constructor application, a term can now also be a term reference r. A reference can be thought of as a pointer to another term. It is a a special kind of term that our language extension knows how to distinguish from other terms (refer to Sec. 5 for implementation details).

We conceptually extend the language with three new operators for operating on references: *create new reference*, *dereference*, and *bind reference*. The creation of a new reference produces a fresh, unbound reference with a unique identifier. The identifier is used to compare references with each other. Like terms, references may be passed around as parameters, copied, matched, and assigned to variables. Additionally, references may be bound. Binding a term to a reference is similar to binding a value to a variable. After binding, a reference may be dereferenced. A deference will produce the term which was previously bound. These conceptual operators are available inside pattern expressions in three different forms, giving us *term graph patterns*.

Build and bind: `!`r`~`$p(x)$ will instantiate the term pattern $p(x)$ and bind the resulting term to a new, unique reference which will be bound to the variable r. If the variable r is already bound to a reference, a new reference will not be created. Instead, the reference of r will be rebound to the new term.

Bind or match: `?`r`~`$p(x)$ matches a reference r bound to a term that matches the term pattern $p(x)$. More specifically, to succeed, this expression must be applied to a reference r', r' must have the same reference identifier as r, and r' must be bound to a term which matches the term pattern $p(x)$. If the variable r is unbound, r will be bound to r' before the matching starts.

Dereference: `^r` will dereference the reference *r*, i.e., if `r` is bound to the term `t`, `^r` will produce `t`. With `r` bound to `t`, `?^r` is equivalent to `?t` and `!^r` is equivalent to `!t`.

With these operations, we can instantiate the term graph in Fig. 1(c): `!r~Int(2); !Sub(r,r)`. Or more succinctly as one term graph pattern: `!Sub(r~Int(2),r)`. In the following we use some idioms: When we need a reference, but do not yet have its term, we use the expression `!r~()` to create a new reference bound to the "dummy" term `()`. If we want to match a reference, but do not care what it is bound to, we write `?r~_`.

General Graphs Term references may also be used to construct more general graphs, such as those shown in Figures 1(d) and 1(e). When constructing mutually dependent graphs, such as the `f()` and `g()` nodes in Fig. 1(d), term graph construction must always be split into two stages. First, one half must be built with an unbound reference, e.g. `!f~FunDef("f",[g~()])`, then the graph is connected when the other half is built: `!g~FunDef("g",[f])`. Note the use of the idiom `g~()` to break cycles.

2.3 Rewrite Rules and References

Conditional rewrite rules on term graphs, R: $g_l(x)$ `->` $g_r(x)$ `where` c, mirror rewrite rules on terms. $g_l(x)$ and $g_r(x)$ are term graph patterns, as described previously and c is the rule condition. `Simplify` can now be reformulated to work on term graphs:

```
Simplify: r0~Add(r1~Int(x~_), r2~Int(y~_)) -> r0~Int(r3)
  where !r3~Int(r4~<add> (^x,^y))
```

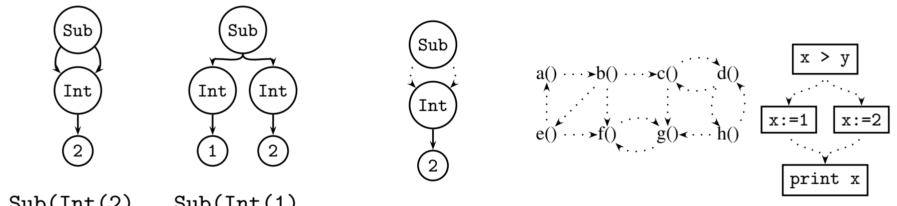
Rewrite rules on term graphs will not maintain maximal sharing unless the programmer takes explicit care. This leads to differences in the equality checking of term graphs compared to equality checking of term. For efficiency, the built-in comparison of term graphs only exists in a "shallow" form, i.e. identity checking: Two terms with references are equal iff all subterms are structurally equal, and all references have the same identity. This means that terms may now in fact be structurally equal, but differences in their reference identities will prevent the shallow equality test from uncovering this.

Rebinding of References For terms, maximal sharing and constant time equality checking is always guaranteed by the ATerm library. When matching a regular variable against a term, the pointer to that term gets copied when it is used in a build. If the original term is later modified, copy-on-write is performed behind the scenes to ensure referential transparency. For term graphs, this is no longer the case, as references may be rebound at any time. Consider the expression `!Sub(r~Int(2),r) => a ; !r~Int(3)`. Here, we assign the graph from Fig. 1(c) to the variable `a`, but subsequently change the binding of the references contained in the term graph of `a`. Effectively, this will change the value of `a` after `a` was bound. This may seem dangerous, as it opens up for problems related to lack of referential transparency. Certainly, these issues must be managed, but it is important to note that the binding of terms to references is always done explicitly. It is not possible to retroactively create a reference to a subterm of another term. E.g., if the term `Sub(Int(2),Int(2))` from Fig. 1(a) is bound to the variable `v`, it can never change, as it does not contain any references.

If side effects are unwanted, in the sense that references in the term of an already bound variable should never change, assignments of term graphs should be coupled with a call to `duprefs`, e.g. `!r~Int(2); !Sub(r,r); duprefs => a; !r~Int(3)`. Here, the references in the term graph `Sub(r,r)` will be replaced with new, unique references before the assignment, so that subsequent rebindings cannot affect the value of `a`.

2.4 Term Graph Traversal

We will now show how generic traversals are adapted to work on term graphs through an example of term graph normalization. Our goal is to use `Simplify`, shown earlier, to simplify the term graph `Sub(r~Add(r'~Int(1),r'), r)`. Specifically, we only want to simplify each referenced term once. This illustrates how "side-effects" can be used beneficially, and how term graph rewriting can be more efficient than term rewriting: we only need to consider identical terms once, because we can recognize them by their reference identifiers. This argument is only valid once a proper term graph has been constructed, however. Our current implementation makes no attempt at maintaining such term graph properties globally during arbitrary rewriting sequences.



`Sub(Int(2), Int(2))` (a) DAG

`Sub(Int(1), Int(2))` (b) DAG

`Sub(r~Int(2),r)` (c) Term graph

(d) Call Graph

(e) Flow graph

Fig. 1. Examples of graphs supported by our language. References are shown as stippled edges.

Phased Traversals To manage termination of graph traversals, we introduce a concept of *phases*. Phases are used to ensure that each reference is only visited once, so that loops in the graph do not result in non-termination. We do this by introducing a new primitive strategy, `phase(s)`, and new variants of the primitive traversal operators: `wone(s)`, `wsome(s)` and `wall(s)`.

When applied to a reference r, `wall(s)` will first dereference r, obtaining the term t, then apply `s` to all subterms of t. The resulting term is rebound to r. If any subterm of t is also a reference, it will be dereferenced before s is applied, and rebound afterwards. `wone(s)` and `wsome(s)` are similar.

`phase(s)` will internally instantiate a new, globally unique marker and then apply `s` to the current term. Any invocations of `wone`, `wsome` and `wall` in `s` will take the marker into consideration. As each reference is dereferenced during the traversal, using `wone`, `wsome` or `wall`, the marker is placed on the reference. Any subsequent attempt at dereferencing will not yield a term result, thus making the reference untraversable. When the phase is exited, all markers for that phase are removed. It is possible to nest phases. The inner phase will instantiate a new, unique marker and can revisit references already visited by its enclosing phase.

During a phased traversal, it is sometimes necessary to control whether the dereferences due to matching or the `^` operator should be marked with the current phase marker or not. This can be controlled by using `wrap-ref(s)`, which dereferences, applies `s`, then rebinds, irrespective of any phase markers. Analogously, `wrap-phase-ref(s)` can be used to only visit unmarked references, and mark the reference after a visit.

Reference Expression	Meaning
`!`r`~`$p(x)$	*(Build and bind)* Instantiate term pattern $p(x)$ and bind result to r.
`?`r`~`$p(x)$	*(Bind or match)* See text in Sec. 2.2.
`^`r	*(Dereference)* Look up term for r. Fail iff r is unbound.
`duprefs`	Replace all references in the current term with new, unique ones
`phase(`s`)`	For traversals done by s, visit each reference at most once
`wrap-ref(`s`)`	Apply s to reference and rebind, irrespective of phase
`wrap-phase-ref(`s`)`	Apply s to reference and rebind, while respecting phase

Generic Graph Traversals The following traversal strategies are adaptations of the generic traversals for terms. They use phase markers to avoid visiting the same reference more than once. These strategies use `wall` and will rebind references they encounter to rewritten terms.

```
wtopdown(s)    = phase(rec x(s ; wall(x)))
wbottomup(s)   = phase(rec x(wall(x) ; s))
wdownup(s1,s2) = phase(rec x(s1; wall(x); s2))
```

Term Graph Normalization With the phasing and generic graph traversals in place, we can now express term graph normalization simply and precisely as:

```
simplify = wbottomup(try(Simplify))
```

When applied to the term `Sub(r~Add(Int(1),Int(2)), r)`, the result of `simplify` is `Sub(0~Int(3), 0~)`. Here, `0~t` indicates that the reference 0 is bound to the term `t`. Bindings of references are only shown once, the first time they are encountered.

For more general rule sets, it may be necessary to exhaustively apply rules using a fixed-point strategy. The following strategy follows the same definition pattern as `innermost` for plain terms [6, 7]:

```
winnermost(s) = phase(rec x(wall(x); try(s; x)))
```

The difference is that the `phase` mechanism ensures that a node in a term graph (where all subterms are references) is visited only once. Thus the strategy performs a bottom-up traversal and tries to apply the normalization strategy `s` to each node. If that succeeds, the result is transformed by a recursive call to itself. This would entail a complete bottom-up traversal of the resulting term. However, the subterms that have already been visited, i.e., normalized, will not be visited again. This property ensures efficient implementation of the strategy, a result that was obtained in the term case only through a specialization of the strategy to its argument rules [6, 7].

3 From Terms to Term Graphs

In this section, we describe how ASTs can be turned into various types of graphs commonly found in compilers, such as use-def chains, call graphs and flow graphs. First, however, we turn our attention to the problem of computing term graphs from terms with maximal sharing. This is done using dynamic rules.

Dynamic Rules A dynamic rule S is a rewrite rule which is defined and possibly undefined at runtime, see [2]. The expression `rules(`S`: t -> r)` creates a new rule in the rule set for S. The scope operator `{| S :` s `|}` introduces a new scope for the rule set S around the strategy s. Changes (additions, removals) to the rule set S done by the strategy s are undone after s finishes (both in case of failure and success of s). Sometimes, multiple rules in a rule set S may match. To get the results of all matching rules in S, we can use `bagof-S`.

Computing Term Graphs The following strategy implements a top down traversal with a memoization scheme to efficiently construct term graphs from terms. For each term it encounters, the strategy checks if this term has been memoized in the dynamic rule S. If so, the term is replaced with its corresponding reference. If not, all its subterms are replaced with references recursively, then a new reference is created and recorded in S.

```
term-graph = {|S: rec x(S<+all(term-graph); ?t; !r~t; rules(S: t->r)|}
```

Applied to `A(A(B),A(A(B),A(B)))`, we get `3~A(1~A(0~B),2~A(1~,1~))`.

3.1 Use-Def Chains

The use-def chain is a data representation found in most compilers for recording links from the use of a variable to its closest definition or assignment. Such data flow information is the basis for many program optimizations, in particular constant and copy propagation. A variable is said to be *used* when its value is read; it is said to be *defined* when it is assigned to, either at its declaration or by a later assignment statement. Def-use chains are links from the definition of a variable to all its uses. The algorithm we present below will record both def-use and use-def chains.

```
use-def       = {| Use, Def: def-to-use ; use-to-def |}
def-to-use    = Var <+ VarRef <+ Assign <+ If <+ wall(def-to-use)
use-to-def    = topdown(try(add-ref-to-var <+ add-ref-to-assign))
new-def(|v,r) = rules(Def : v -> r)
add-use(|d,u) = rules(Use :+ d -> u)

add-ref-to-var    = ?r~Var(v,e,_); !r~Var(v,e, Uses(<bagof-Use> r))
add-ref-to-assign = ?r~Assign(v,e,_); !r~Assign(v,e,Uses(<bagof-Use> r))

If: If(c,t,e) -> If(c',t',e') where <def-to-use> c => c'
  ; <def-to-use> t => t' \Def/ <def-to-use> e => e'

Var: Var(v, e) -> r where <def-to-use> e => x
  ; !r~Var(v,x,Uses([])); new-def(|v, r)

VarRef: VarRef(v) -> r where <bagof-Def> v => defs
  ; !r~VarRef(v, Defs(defs)); <map(add-use(|<id>, r))> defs

Assign: Assign(v, e) -> r where <def-to-use> e => x
  ; !r~Assign(v, x, Uses([])); new-def(|v,r)
```

The `use-def` algorithm assumes the existence of the following term constructors: `If`, for `if` constructs, `Var`, for variable definitions, `VarRef` for variable (de)references and `Assign` for assignments. It is divided into two parts, `def-to-use` and `use-to-def`. For `def-to-use`: If a definition of a variable is seen, i.e. an `Assign` or `Var` term, this term is replaced with a reference to itself, and a mapping from the variable name to the reference is recorded in the dynamic rule `Def` using `new-def`. This is done in the `Var` and `Assign` rules. When a variable use is subsequently seen, it is also replaced by a term to itself by the `VarRef` rule. Its name is looked up in the `Def` rule, and references to the closest definitions are added using `bagof-Def`, see `VarRef`. The `Use` rule is updated to record the reference to this use, using `add-use`. Special care must be taken in the case of control constructs. We only show the case for `If`. Here, one rule set is computed for each branch, and the rule sets are joined afterwards, using the rule set union operator, `\Def/`. This ensures that new definitions from both branches are kept.

For `use-to-def`: In this pass, each `FunDef` is updated to contain references to the uses recorded by the previous pass, in the `Use` rule. When applied to a term for the program

```
var x := 0; if (x > 0) { x := 1 + x } else { x := 2 + x } ; print x
```

we get (read `Asn` as `Assign` and `VRef` as `VarRef`):

```
Block([~5,If(Int(0),Block([~1]),Block([~3])),Print(~0)])
~0 = VRef("x",Defs([~3,~1]))  ~1 = Asn("x",Add(Int(1),~2),Uses([~0]))
~2 = VRef("x",Defs([~5]))     ~3 = Asn("x",Add(Int(2),~4),Uses([~0]))
~4 = VRef("x",Defs([~5]))     ~5 = Var("x",Int("10"),Uses([~4,~2]))
```

3.2 Call Graphs

Another common program representation in modern compilers is the call graph. It records the interrelationships between the functions of a program, i.e. which functions call which, and is used for various static analyses such as reachability analysis, optimizations such as dead code removal and by documentation generation tools. The following code transforms an AST into a call graph by introducing references from all call sites (`Call` terms) to the corresponding function definition (`FunDef`) terms, and by adding a reference from the `FunDef` being called (callee) to the `FunDef` of the calling function (caller). Fig. 1(d) illustrates the forward direction.

```
compute-call-graph   = {| FunLookup:  add-refs ; add-call-markers |}
introduce-references = topdown(try(AddFunRef))
with-fundefs(s)      = Program(map(s), id)
register-fun         = ?r; ?r~FunDef(_,_,_,_); rules(CurFun: _ -> r)

AddFunRef: x@FunDef(n,_,_,_) -> r where !r~x; rules(FunLookup: n -> r)

add-call-markers = {| CalledBy, CurFun:
    with-fundefs(wdownup(try(register-fun), try(AddCallRef)))
  ; with-fundefs(wrap-ref(AddCalledByRef)) |}

AddCallRef: Call(n, xs) -> Call(n, xs, r)
where <FunLookup> n => r ; CurFun => z ; rules(CalledBy :+ n -> z)

AddCalledByRef: FunDef(n,a,t,b) -> FunDef(n,a,t,ns,b)
where <bagof-CalledBy> n => ns
```

Three dynamic rules are used in this algorithm. `FunLookup` is used to map names of functions to their corresponding `FunDef`. `CurFun` is used to keep a reference to the `FunDef` we are currently inside. `CalledBy` is used to accumulate a set of callees for a given function name.

The algorithm works as follows. First, we replace every `FunDef` n node with a reference to n, and record the function name in the `FunLookup` rule set. This is done by `add-refs`. Second, we do a downup traversal, where the current function is kept in the dynamic rule `CurFun` on the way down. On the way up, we add a reference to the destination `FunDef` f for any `Call` encountered, and register the current function in the `CalledBy` set for f. This is the first part of `add-call-markers`. Third, we place the callee sets collected in the `CalledBy` dynamic rule set on the corresponding `FunDef`s, finally obtaining a bidirectional call graph.

3.3 Flow Graphs

Flow graphs are used to represent the control flow of a program, analogously to the way use-def chains represent data flow. Flow graphs, along with use-def chains, are at the heart of many flow-sensitive optimizations, such as constant folding, loop optimization, and jump threading. We can compute a flow graph from the various statements in the AST as follows. In `If(`c`,`t`,`e`)`, flow goes from the condition c to both branches, t and e, which in turn go to the successor block of `if`. In `While(`c`,`b`)`, flow goes from the condition c to the body b, and from c to the successor block. The body b always flows back to the condition c. All other statements correspond to basic blocks: the flow from one statement goes directly to the successor block.

We show a three pass algorithm, `ast-to-flow-graph`. First, we do rewrites of control flow constructs locally, as described above, with the `MarkControlFlow` rule set. In the case of `If` and `While`, temporary `FlowT` blocks are inserted with dummy references, since the successor block is not known locally yet. Second, AST statement blocks are split into basic blocks, with `SplitBlocks`. Each non-control statement is rewritten to a Flow block. Third, the `FlowT` blocks are connected to the `Flow` blocks produced in (2), and rewritten to `Flow`, resulting in a flow graph, as seen in Fig. 1(e).

```
ast-to-flow-graph = bottomup(try(MarkControlFlow))
  ; bottomup(try(SplitBlocks))
  ; wbottomup(try(\ FlowT(x,y) -> Flow(x,y) \))

SplitBlocks: Block(xs) -> r where
  <map(?FlowT(_,_) <+ {r: \ t -> Flow(t, r) where !r~() \})> xs => xs'
; foldr(\ (f@Flow(t1, n), t2) -> f where !n~t2 \
     <+ \ (f@FlowT(t1, n), t2) -> f where !n~t2 \ |None)
; <Hd> xs' => hd ; !r~hd

MarkControlFlow: If(cond, thn, els) -> FlowT(If(cond', thn', els'), next)
where !next~(); ; !thn'~Flow(thn,[next])
  ; !els'~Flow(els,[next]); !cond'~Flow(cond,[thn', els'])

MarkControlFlow: While(cond, body) -> FlowT(r, next)
where !body'~(); !next~(); !cond'~Flow(cond, [next, body'])
  ; !body~Flow(body, [cond']); !r~While(cond', body')
```

4 Graph Algorithms and Applications

In this section, we show how some basic graph algorithms can be implemented using the reference mechanism we have described in Sec. 2.

Depth First Search Our depth first search implementation works on graphs where each node is a term. The algorithm takes two parameters, `l` and `es`. `es` will be used to compute the outgoing edges from each node. `dfs` keeps track of the current depth during visits. On a visit to a node, the strategy `l` will be called with the current depth value as parameter, so that it can be used to compute the label for the current node, or for other transformations.

```
dfs(l : a * a -> a, es)     = phase(wall(dfs(l, es | 0)))
dfs(l : a * a -> a, es | n) =
    wrap-phase-ref(where(es => edges)
  ; where(l(|n) => label)
  ; where(<wall(dfs(l, es | <inc> n))> edges) ; !label)
```

The traditional depth first search, as described in for example [4], is applied initially to the set V of a graph $G = (V, E)$. We get the same behavior by applying `dfs` to a list of references to all nodes in the graph. We will demonstrate the use of the `dfs` strategy next, when we discuss strongly connected components.

Strongly Connected Components The basic algorithm for strongly connected components (SCC) is also described in [4], and consists of four steps: First, call `DFS(G)` to compute finishing times `f[u]` for each vertex `u`. Second, compute `GT=transpose(G)`. Third, call `DFS(GT)`, but in the main loop of `DFS`, consider the vertices in order of decreasing `f[u]`. Fourth, produce as output the vertices of each tree in the `DFS` forest formed in point 3 as a separate strongly connected component.

In our implementation of SCC, shown below, we avoid actual graph transposition by requiring one strategy, `es` for computing forward edges from a node, and another, `res`, for computing reverse edges. We also combine the third and fourth step by using a modified `dfs`, called, `dfs-collect`, which collects each set of SCCs into a list during the third step.

```
dfs-collect(l : a * a -> a, es) =
  phase(all({|C: dfs-collect(l,es|0) ; bagof-C|}))

dfs-collect(l : a * a -> a, es | n) = ?r~_
  ; wrap-phase-ref(where(es => edges) ; where(l(|r) => label)
  ; where(<all(dfs-collect(l, es | <inc> n))> edges) ; !label)

sort-fundefs =
  sort-list(LSort(where((?r;!^r; FinishTime,?r';!^r'; FinishTime); gt)))
collect-components(|r) = rules(C :+ _ -> r)
inc-time = (Time <+ !0) => n ; where(inc => n'; rules(Time: _ -> n'))
time-count(|n) = ?x; where(inc-time => n'); rules(FinishTime: x -> n')
```

```
scc(l : a * a -> a, es, res) = {|FinishTime, Time:
    dfs(l, es)
  ; sort-fundefs
  ; dfs-collect(collect-components, res)
  ; filter(not(?[])) |}
```

The current time is maintained in the `Time` dynamic rule, and the finishing time in `FinishTime`. Our `scc` should normally be called with the `time-count` strategy as its first argument, but the user is free to adapt this.

4.1 Finding Mutually Recursive Functions

Suppose we want to use SCC to compute sets of mutually recursive functions. Then, each node in the graph is a function f. The outgoing edges of f are references to the functions *called by* f. The incoming edges of f are references to the functions *calling* f. This graph is what was computed by `call-graph`, discussed in Sec. 3.2. The following strategies may used for edge computations.

```
calls-as-outbound    = collect(\ Call(_,_,x) -> x \)
calledby-as-outbound = collect(\ FunDef(_,_,_,x,_) -> x \) ; concat
```

Applying `scc(time-count, calls-as-outbound, calledby-as-outbound)` to a list of references to all functions in a program, say Fig. 1(d), will produce the cliques (a, b, e), (f, g) and (c, d, h).

4.2 Lazy Graph Loading

Instead of binding terms to references, strategies may be bound instead. When a reference r with the strategy s bound to it is dereferenced, s is invoked, and the resulting term is taken as the term value for r. We call this an *active reference* since it has a strategy (i.e., function) attached to it that is activated and executed upon dereference. Active references are useful for term (graph) rewriting of larger terms, especially when doing sparse analyses on larger bodies of program code. With active references, terms for programs can be loaded as skeletons. For example, all bodies of functions or classes may be left out, and be parsed and loaded, or even generated, on demand.

5 Implementation

We have implemented a prototype of the language extension described in this paper. Our implementation is a conservative extension to the existing Stratego infrastructure: Every valid Stratego program retains its behavior and terms without references are still represented entirely as ATerms. References are introduced as a special kind of term, `Ref`, and we have modified the language implementation to recognize and treat terms of this type specially. `Ref`s are closely related to pointers, as found in C, and to references, as found in Java. Unlike pointers and Java references, a Stratego `Ref` always starts out as bound. It may subsequently be rebound to another term. The bindings from references to terms are maintained in a global table, or more precisely, in a global, dynamic rule set. When a new reference is bound, a new rule is added to the set. When an existing reference is rebound, its corresponding rule is changed. Using dynamic rule sets aids in implementing backtracking behavior. For left- and guarded choice, references rebound or introduced by a failed strategy should be backtracked before the program proceeds. This is implemented in our compiler by rewriting every left choice operator to

```
start-ref-cs ; s1 ; commit-ref-cs <+ discard-ref-cs ; s2
```

Here, `start-ref-cs` will make a change set for the global rule set. If `s1` succeeds, the change set is committed and changes are kept. If `s1` fails, all changes to the reference rule set by `s1` are undone.

Managing the revisitation of references in term graphs is crucial for ensuring termination. The `wrap-phase-ref` mentioned earlier is responsible for this.

```
wrap-phase-ref(s) = ?r@Ref(_) < seen-before < id
+ where(<phase-deref> r; s; bind-ref(|r)) + s
```

`wrap-phase-ref` is implemented using guarded choice s_1 `<` s_2 `+` s_3, which works as follows. If s_1 succeeds, s_2 will be applied to the resulting term. If it fails, s_3 will be applied to the initial term. If `wrap-phase-ref(s)` is applied to term, `s` is applied and we are done. When at a reference r, we first use `seen-before` to check if we have seen r before. If so, we ignore `s` (by applying `id`, then returning). If not, r is dereferenced and marked, using `phase-deref`, `s` is applied to its term, and r is rebound by `bind-ref`.

Using `wrap-phase-ref`, we can now implement new traversal primitives on references. Let us consider `wall(s)`:

```
wall(s) = is-ref
  < wrap-phase-ref(all(wrap-phase-ref(s))) + all(wrap-phase-ref(s))
```

If we are not at a reference, `all` will be applied to the current term and any references it has as direct subterms will be marked. If we are at a reference, we will mark, transform then rebind it. The markers used by `wrap-phase-ref` can be managed using `phase(s)`, given next:

```
phase(s) = where(local-phase-ctr => pc; inc-phase-ctrs)
  ; start-seen-cs; s; discard-seen-cs
  ; where(restore-local-phase-ctr(|pc))
```

`phase(s)` works as follows: Before s is applied, a new, unique, internal phase marker is produced using `local-phase-ctr`, then the counter is increased, preparing for the next invocation. `start-seen-cs` enters a new “scope” for this marker. The counter is maintained in a dynamic rule defined inside `increase-phase-ctrs`, and is later used by `phase-deref` and `seen-before`. Once s completes, all markers will be discarded.

Our implementation has not yet been tuned for performance. While we have used Stratego’s dynamic rules for implementation convenience, we only rely on the ability of dynamic rules to provide hash tables with change sets. In the current implementation, reference lookup time is linear in the depth of choices on the stack. A more efficient implementation of hash tables with change sets is likely to improve performance.

6 Related Work

Term graph rewriting theory is an active field. For an introduction and summary, see [12]. A calculus for rewriting on cyclic term graphs has been proposed in [1]. Many systems exist for general graph rewriting, such as PROGRES [13] and FUJABA [10]. Few term graph rewriting systems for practical applications exist. HOPS [8] and Clean [11] are a notable exceptions. Claessen and Sands [3] describe an extension to the Haskell language which adds references with equality tests. Their goal is to better describe circuits, which are graph structures with cycles, in a purely functional language. Their references are immutable once created, unlike ours, making rewriting more difficult express. Lämmel et al [9] discusses how strategic programming relates to adaptive programming, a technique found in aspect-oriented systems for traversing object structures. They show how traversal strategies may be implemented for cyclic structures, such as graphs, by keeping record of visited nodes. Our phased traversals expand upon this by allowing nested, overlapping traversals and fine-grained control of visitation marking. Our implementation shares some features with monadic programming. Monads are sometimes described as patterns for using functions to transmit state without mutation, and are described by Wadler [15]. In our implementation, dynamic rules are the functions used to transmit the state, namely the internal graph references. Some functional languages, such as Clean [11], are implemented as rewriting systems with implicit term graphs. Functions in Clean are graph rewrite rules on the underlying term graph. In our language, the choice between term and term graph rewriting and their corresponding tradeoffs is not fixed, but rather left to the programmer. An important goal for our language extension is to better capture graph-like program representations, and to offer convenient transformation capabilities for these. Many excellent and general graph libraries exist, and we are not aiming to replace these.

To the best of our knowledge, no other term graph rewriting system supports strategic term graph rewriting, using rewriting strategies and generic traversals.

7 Discussion and Further Work

The construction of use-def chains, call- and flow graphs shows how global-to-local problems are now local-to-local, as the remote context is available locally for rules to match on. The addition of references for this purpose also introduces the problem of traversal non-termination in the presence of cycles. We have shown how this can be managed by phases. Another issue of term references is the unexpected impact of reference rebinding, in the loss of referential transparency. The code `!Sub(r~Int(2)) => v ; !r~Int(3)` will alter the value of `v` after it is bound. Judicious use of `duprefs` can control this. Comparison of term graphs is currently done using weak equality; i.e., comparison references in terms is done based on identity, not structure, which allows constant time comparison. Deep comparison is available through the library, and is linear in the size of the term graphs. The pattern-based language constructs introduced in this paper for reference manipulation came about after trying to program with only the primitive operators create reference, bind reference and dereference. While these primitives are still at the heart of the implementation, the notation presented in this paper make them more convenient to use. Further exploration of the design space is warranted. One attractive extension is matching modulo references, which allows term patterns to be matched directly on terms with references, by implicitly visiting references during matching.

8 Conclusion

We have presented the design and implementation of an extension to the Stratego term rewriting language for rewriting on terms with references, and demonstrated its practical application through the construction of several common graph-based program representations found in compilers. The contributions of this paper include the introduction of language abstractions for dealing with references within a rule-based term rewriting language, a demonstration of how term matching, building and rewrite rules can be combined with term references, how benefits of generic term traversal can be kept by using phased traversals to deal with non-termination due to cyclic graphs, and how backtracking can be combined with destructive graph updates to retain the strategic programming flavor of Stratego. We showed how our language can be used to implement some basic graph algorithms and how these can be applied to graph-based program representations. We discussed design tradeoffs related to introducing references in terms, including traversal termination and impact of reference binding.

References

1. C. Bertolissi. *The graph rewriting calculus: properties and expressive capabilities*. Thèse de doctorat, Institut National Polytechnique Lorrain - INPL, Oct. 2005.
2. M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.16: Components for transformation systems. In F. Tip and J. Hatcliff, editors, *PEPM'06: Workshop on Partial Evaluation and Program Manipulation*, January 2006.
3. K. Claessen and D. Sands. Observable sharing for functional circuit description. In *ASIAN '99: Proceedings of the 5th Asian Computing Science Conference on Advances in Computing Science*, pages 62–73, London, UK, 1999. Springer-Verlag.
4. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1997.
5. M. G. T. V. den Brand, H. A. de Jong, P. Klint, and P. A. Olivier. Efficient annotated terms. *Softw. Pract. Exper.*, 30(3):259–291, 2000.
6. P. Johann and E. Visser. Fusing logic and control with local transformations: An example optimization. In B. Gramlich and S. Lucas, editors, *Workshop on Reduction Strategies in Rewriting and Programming (WRS'01)*, volume 57 of *Electronic Notes in Theoretical Computer Science*, Utrecht, The Netherlands, May 2001. Elsevier Science Publishers.
7. P. Johann and E. Visser. Strategies for fusing logic and control via local, application-specific transformations. Technical Report UU-CS-2003-050, Institute of Information and Computing Sciences, Utrecht University, February 2003.
8. W. Kahl. The term graph programming system HOPS. In R. Berghammer and Y. Lakhnech, editors, *Tool Support for System Specification, Development and Verification*, Advances in Computing Science, pages 136–149, Wien, Mar. 1999. Springer. ISBN: 3-211-83282-3.
9. R. Lämmel, E. Visser, and J. Visser. Strategic Programming Meets Adaptive Programming. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD 2003)*, pages 168–177, Boston, MA, 2003. ACM Press.

10. U. Nickel, J. Niere, and A. Zundorf. Tool demonstration: The FUJABA environment. In *The 22nd International Conference on Software Engineering (ICSE)*. ACM Press, 2004.
11. M. Plasmeijer and M. van Eekelen. Language report: Concurrent Clean. Technical Report CSI-R9816, Computing Science Inst., U. of Nijmegen, Nijmegen, The Netherlands, 1998.
12. D. Plump. Essentials of term graph rewriting. *Electr. Notes Theor. Comput. Sci.*, 51, 2001.
13. A. Schürr. *The PROGRES Language Manual Version 9.x.* Lehrstuhl für Informatik III, RWTH Aachen, Aachen, Germany, 2004.
14. E. Visser and Z.-e.-A. Benaissa. A core language for rewriting. In C. Kirchner and H. Kirchner, editors, *Second International Workshop on Rewriting Logic and its Applications (WRLA'98)*, volume 15 of *Electronic Notes in Theoretical Computer Science*, Pont-à-Mousson, France, September 1998. Elsevier Science Publishers.
15. P. Wadler. The essence of functional programming. In *POPL '92: Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–14, New York, NY, USA, 1992. ACM Press.

Author Index