RWTH Aachen

Department of Computer Science
Technical Report

WRS'04
4th International Workshop on Reduction Strategies in Rewriting and Programming

Sergio Antoy and Yoshihito Toyama (eds.)

ISSN 0935–3232 · Aachener Informatik Berichte · AIB-2004- 06

RWTH Aachen · Department of Computer Science · June 2004

Preface

This volume contains the preliminary proceedings of the **Fourth International Workshop on Reduction Strategies in Rewriting and Programming (WRS'04)**. The final prooceedings of the workshop will appear as a volume of the Electronic Notes in Theoretical Computer Science series. The workshop was held on June 2nd, 2004, in Aachen, Germany, as part of the Federated Conference on Rewriting, Deduction, and Programming (RDP'04).

Reduction strategies in rewriting and programming continue to attract attention. As new strategies are discovered and investigated, new results on rewriting and on computation under particular strategies become available. A number of programming languages, e.g., Elan, Maude, *OBJ* and Stratego, and systems, e.g., Clean, Curry, and Haskell, permit the explicit definition or modification of the computational reduction strategy. Research in this field ranges from primarily theoretical questions about reduction strategies to very practical application and implementation issues. There is a need for a deeper understanding of reduction strategies in rewriting and programming, both in theory and practice, since they bridge the gap between unrestricted general rewriting rewriting with particular strategies. Moreover, reduction strategies bridge investigations of operational principles, e.g., graph and term rewriting, narrowing and lambda-calculus, and semantics, e.g., normalization, computation of values, infinitary normalization and head-normalization, with implementations of programming languages.

The workshop is the fourth 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 this area. The previous WRS editions were: WRS 2001 (Utrecht, The Netherlands), WRS 2002 (Copenhagen, Denmark), and WRS 2003 (Valencia, Spain). The next edition is planned for April 2005 in Nara, Japan.

We would like to thank the program committee for providing three timely reviews for each submitted paper, the invited speakers for accepting our invitation, the members of the round table and the moderator for enriching the program, Portland State University and Tohoku University for some financial support, the local organization, and our predecessors for leading the way.

Sergio Antoy and Yoshihito Toyama
Program Co-Chairs

Program Committee

Sergio Antoy	Portland (USA)
Roberto Di Cosmo	Paris (France)
Jürgen Giesl	Aachen (Germany)
Bernhard Gramlich	Wien (Austria)
Salvador Lucas	Valencia (Spain)
Aart Middeldorp	Innsbruck (Austria)
Jaco van de Pol	Amsterdam (The Netherlands)
Pierre Réty	Orléans (France)
Amr Sabry	Bloomington (USA)
Yoshihito Toyama	Sendai (Japan)

The technical program consists of four regular papers plus one position paper, two invited talks by Jan Willem Klop and Olivier Danvy, and a round table moderated by Salvador Lucas with Francisco Duràn, Claude Kirchner, and Ralf Laemmel.

Additional Reviewers

Florent Jacquemard, Joost-Pieter Katoen, Sebastien Limet.

Contents

Reduction Cycles

Jan Willem Klop

Vrije Universiteit Amsterdam
The Netherlands
`jwk@cs.vu.nl`

Autowrite: A Tool for Term Rewrite Systems and Tree Automata

Irène Durand

LaBRI, Université Bordeaux 1, France
idurand@labri.fr

Abstract Autowrite is an experimental software tool written in Common Lisp for handling term rewrite systems and bottom-up tree automata. A graphical interface has been written using McCLIM, (the free implementation of the CLIM specification) in order to free the user of any Lisp knowledge. Software and documentation can be found at

`http://dept-info.labri.u-bordeaux.fr/~idurand/autowrite`

Autowrite was initially designed to check call-by-need properties of term rewrite systems. For this purpose, it implements the tree automata constructions used in [10,3,5,13] and many useful operations on terms, term rewrite systems and tree automata. In the first version of Autowrite [4], only the call-by-need properties and a few other simple properties were available from the graphical interface. This new version of Autowrite includes many new functionalities. There are new functionalities related to TRSs, but the most interesting new feature is the possibility to directly handle (load, save, combine with boolean operations) bottom-up tree automata. In addition, we have added on-line timing information. Since the first version the run-times have been considerably improved due to better choices of data structures. The first version of Autowrite was used to check call-by-need for most of the examples presented in [6]. Most of the time no alternative proofs exists. The new features allowed testing many properties of examples presented in [7] for which no easy proof can be written.

1 Introduction

Huet and Lévy [9] showed that for the class of orthogonal term rewrite systems (TRSs) every term not in normal form contains a needed redex (i.e., a redex contracted in every normalizing rewrite sequence) and that repeated contraction of needed redexes results in a normal form if it exists. However, neededness is in general undecidable. In order to obtain a decidable approximation to neededness, Huet and Lévy introduced the subclass of *strongly sequential* TRSs.

In a strongly sequential TRS, at least one of the needed redexes in every reducible term can be effectively computed. Moreover, Huet and Lévy showed that strong sequentiality is a decidable property of orthogonal TRSs. Strong sequentiality is based on the idea of approximating the TRS by replacing each right-hand side of every rule bu a fresh variable (this known as the *strong approximation*). That always makes a really rough approximation of the system.

Several authors [1,3,10,12,13,14,15] proposed decidable extensions of the class of strongly sequential TRSs. Since Comon's [1] and Jacquemard's [10] work, all the corresponding decidability proofs have been expressed using tree automata techniques: typically a property is satisfied if and only if some associated automaton recognizes the empty language. A uniform framework for the study of call-by-need (sequentiality) is described in [3] where classes of term rewrite systems are parameterized by approximation mappings. Nowadays the best known approximation (easily definable) for which call-by-need is decidable is the growing approximation studied in [13]. No real system is strong (equal to its strong approximation). In return, many real TRSs are already growing (equal to their growing approximation). For these systems call-by-need is decidable which makes the growing approximation very valuable.

2 Preliminaries

We assume that the reader is familiar with the basics of term rewriting [11] and tree automata [2]. Familiarity with the theory of call-by-need [1,3,5,6,9,10,13,14] is helpful.

A *term rewrite system* (TRS for short) $\mathcal{R}$ over a finite *signature* $\mathcal{F}$ (set of symbols with fixed arity) consists of *rewrite rules* $l \to r$ between terms in $\mathcal{T}(\mathcal{F}, \mathcal{V})$ that satisfy $l \notin \mathcal{V}$ and $\mathcal{V}(r) \subseteq \mathcal{V}(l)$. Here $\mathcal{V}$ is a denumerable set of *variables*. If the second condition is not imposed we find it useful to speak of *extended* TRSs (eTRSs). Such systems arise naturally when we approximate TRSs, as explained in Section 2.2.

2.1 Call-by-need

Let $\mathcal{R}_\bullet$ be the eTRS $\mathcal{R} \cup \{\bullet \to \bullet\}$ over the extended signature $\mathcal{F}_\bullet = \mathcal{F} \cup \{\bullet\}$. We say that redex Δ in $C[\Delta] \in \mathcal{T}(\mathcal{F})$ is *$\mathcal{R}$-needed* if there is no term $t \in \mathsf{NF}(\mathcal{R})$ such that $C[\bullet] \to_{\mathcal{R}}^* t$. Finally, we say that $\mathcal{R}$ is *in $\mathcal{CBN}$ (Call-By-Need)*if every reducible term in $\mathcal{T}(\mathcal{F})$ contains a $\mathcal{R}$-needed redex.

Theorem 1. *Let $\mathcal{R}$ be an orthogonal TRS.*

1. *Every reducible term contains a needed redex.*
2. *Repeated contraction of needed redexes results in a normal form, whenever the term under consideration has a normal form.*

□

So, for orthogonal TRSs, the strategy that always selects a needed redex for contraction is normalizing and optimal. Unfortunately, needed redexes are not computable in general. Hence, in order to obtain a *computable* optimal strategy, we need to find (1) decidable approximations of neededness and (2) (decidable) classes of rewrite systems which ensure that every reducible term has a needed redex identified by (1).

2.2 Approximation mappings

Let $\mathcal{R}$ and $\mathcal{S}$ be eTRSs over the same signature. We say that $\mathcal{S}$ approximates $\mathcal{R}$ if $\to_{\mathcal{R}}^* \subseteq \to_{\mathcal{S}}^*$ and $\mathsf{NF}(\mathcal{R}) = \mathsf{NF}(\mathcal{S})$. An approximation mapping is a mapping α from TRSs to eTRSs with the property that $\alpha(\mathcal{R})$ approximates $\mathcal{R}$, for every TRS $\mathcal{R}$. Given a TRS $\mathcal{R}$, we say that $\mathcal{R}$ is *in $\mathcal{CBN}_\alpha$ (α-sequential)* if $\alpha(\mathcal{R})$ is in $\mathcal{CBN}$.

Next we define the approximation mappings s, nv, and gr. Let $\mathcal{R}$ be a TRS. The *strong* approximation $\mathrm{s}(\mathcal{R})$ is obtained from $\mathcal{R}$ by replacing the right-hand side of every rewrite rule by a variable that does not occur in the corresponding left-hand side. The *non-variable* approximation $\mathrm{nv}(\mathcal{R})$ is obtained from $\mathcal{R}$ by replacing the variables in the right-hand sides of the rewrite rules by pairwise distinct variables that do not occur in the corresponding left-hand sides. The *growing* approximation $\mathrm{gr}(\mathcal{R})$ is obtained from $\mathcal{R}$ by renaming the variables in the right-hand sides that occur at a depth greater than 1 in the corresponding left-hand side. Given a TRS $\mathcal{R}$ and $\alpha \in \{\mathrm{s}, \mathrm{nv}, \mathrm{gr}\}$, it is decidable whether $\alpha(\mathcal{R})$ is in $\mathcal{CBN}$. However the decision procedures are very complex (exponential for s, doubly exponential for nv and gr [5]) so it is impossible to show by hand that a particular system is in $\mathcal{CBN}$, even for very small systems. However showing that a system is not in $\mathcal{CBN}$ can be done more easily by exhibiting a term with no needed redex. This latter property of a term can be proved in polynomial time with regards to the size of the term plus the size of the TRS.

The following example is taken from [12].

Example 1.

$$\mathcal{R}_1 = \begin{cases} \mathsf{f}(\mathsf{g}(\mathsf{a},x),\mathsf{a}) & \to x \\ \mathsf{f}(\mathsf{g}(x,\mathsf{a}),\mathsf{c}) & \to x \\ \mathsf{f}(\mathsf{d},x) & \to x \\ \mathsf{g}(\mathsf{e},\mathsf{e}) & \to \mathsf{e} \end{cases}$$

For $\mathcal{R}_1$, we obtain the following approximated TRSs:

$$\mathrm{s}(\mathcal{R}_1) = \begin{cases} \mathsf{f}(\mathsf{g}(\mathsf{a},x),\mathsf{a}) & \to y \\ \mathsf{f}(\mathsf{g}(x,\mathsf{a}),\mathsf{c}) & \to y \\ \mathsf{f}(\mathsf{d},x) & \to y \\ \mathsf{g}(\mathsf{e},\mathsf{e}) & \to y \end{cases} \quad \mathrm{nv}(\mathcal{R}_1) = \begin{cases} \mathsf{f}(\mathsf{g}(x,\mathsf{a}),\mathsf{a}) & \to y \\ \mathsf{f}(\mathsf{g}(\mathsf{a},x),\mathsf{c}) & \to y \\ \mathsf{f}(\mathsf{d},x) & \to y \\ \mathsf{g}(\mathsf{e},\mathsf{e}) & \to \mathsf{e} \end{cases} \quad \mathrm{gr}(\mathcal{R}_1) = \begin{cases} \mathsf{f}(\mathsf{g}(x,\mathsf{a}),\mathsf{a}) & \to y \\ \mathsf{f}(\mathsf{g}(\mathsf{a},x),\mathsf{c}) & \to y \\ \mathsf{f}(\mathsf{d},x) & \to x \\ \mathsf{g}(\mathsf{e},\mathsf{e}) & \to \mathsf{e} \end{cases}$$

The next example ($\mathcal{R}_2$) comes from [14].

Example 2.
$$\mathcal{R}_2 = \begin{cases} \mathsf{f}(\mathsf{g}(\mathsf{a},x),\mathsf{a}) & \to \mathsf{c} \\ \mathsf{f}(\mathsf{g}(x,\mathsf{a}),\mathsf{b}) & \to \mathsf{c} \\ \mathsf{f}(\mathsf{k}(\mathsf{a}),x) & \to \mathsf{c} \\ \mathsf{g}(\mathsf{b},\mathsf{b}) & \to \mathsf{h}(\mathsf{b}) \\ \mathsf{h}(x) & \to \mathsf{k}(x) \end{cases} \qquad \mathrm{nv}(\mathcal{R}_2) = \begin{cases} \mathsf{f}(\mathsf{g}(\mathsf{a},x),\mathsf{a}) & \to \mathsf{c} \\ \mathsf{f}(\mathsf{g}(x,\mathsf{a}),\mathsf{b}) & \to \mathsf{c} \\ \mathsf{f}(\mathsf{k}(\mathsf{a}),x) & \to \mathsf{c} \\ \mathsf{g}(\mathsf{b},\mathsf{b}) & \to \mathsf{h}(\mathsf{b}) \\ \mathsf{h}(x) & \to \mathsf{k}(y) \end{cases}$$

$\mathrm{gr}(\mathcal{R}_2) = \mathcal{R}_2$

The last example ($\mathcal{R}_3$) is an extension of Berry's example.

Example 3.
$$\mathcal{R}_3 = \begin{cases} \mathsf{f}(x,\mathsf{a},\mathsf{b}) & \to \mathsf{h}(x) \\ \mathsf{f}(\mathsf{b},x,\mathsf{a}) & \to \mathsf{h}(x) \\ \mathsf{f}(\mathsf{a},\mathsf{b},x) & \to \mathsf{h}(x) \\ \mathsf{h}(\mathsf{k}(x)) & \to \mathsf{g}(x,x) \\ \mathsf{g}(\mathsf{a},\mathsf{a}) & \to \mathsf{g}(\mathsf{a},\mathsf{a}) \\ \mathsf{g}(\mathsf{a},\mathsf{b}) & \to \mathsf{a} \\ \mathsf{g}(\mathsf{b},\mathsf{a}) & \to \mathsf{b} \end{cases} \qquad \mathrm{gr}(\mathcal{R}_3) = \begin{cases} \mathsf{f}(x,\mathsf{a},\mathsf{b}) & \to \mathsf{h}(x) \\ \mathsf{f}(\mathsf{b},x,\mathsf{a}) & \to \mathsf{h}(x) \\ \mathsf{f}(\mathsf{a},\mathsf{b},x) & \to \mathsf{h}(x) \\ \mathsf{h}(\mathsf{k}(x)) & \to \mathsf{g}(y,y) \\ \mathsf{g}(\mathsf{a},\mathsf{a}) & \to \mathsf{g}(\mathsf{a},\mathsf{a}) \\ \mathsf{g}(\mathsf{a},\mathsf{b}) & \to \mathsf{a} \\ \mathsf{g}(\mathsf{b},\mathsf{a}) & \to \mathsf{b} \end{cases}$$

Autowrite computes $\alpha(\mathcal{R})$ for any approximation α in $\{\mathrm{s}, \mathrm{nv}, \mathrm{gr}\}$.

Theorem 2. *The approximation mappings* s*,* nv*, and* gr *are regularity preserving.* □

Nagaya and Toyama [13] proved the above result for the growing approximation; the tree automaton that recognizes $(\to_{\mathrm{gr}}^*)[L]$ is defined as the limit of a finite saturation process. This saturation process is similar to the ones defined in Comon [1] and Jacquemard [10], but by working exclusively with deterministic tree automata, non-right-linear rewrite rules can be handled.

3 Real problems solved by Autowrite

3.1 Convince someone that $\mathcal{R} \in \mathcal{CBN}$ for a given $\mathcal{R}$

It is quite easy to convince someone that a TRS is not in $\mathcal{CBN}$ by exhibiting a term with no $\mathcal{R}$-needed redex. However convincing someone that a TRS is in $\mathcal{CBN}$ is not an easy matter; any attempt generally ends up in a long and tedious proof which in addition will work only for the particular TRS considered. Often, in papers about Call-By-Need (or Sequentiality) the authors always prove that some $\mathcal{R} \notin \mathcal{CBN}$ but never that some $\mathcal{R} \in \mathcal{CBN}$. Usually, they just conjecture or say they think that the TRS is in $\mathcal{CBN}$.

For TRSs of reasonable size, we can use Autowrite to convince the reader that a TRSs is in $\mathcal{CBN}$. Also, when looking for a TRS in $\mathcal{CBN}$ and having particular properties, we were often surprised to learn from Autowrite that the TRSs was not in $\mathcal{CBN}$ contrary to our intuition. With the term with no $\mathcal{R}$-needed redex exposed by Autowrite we would be right away convinced of our mistake.

Take for instance the example of Oyamaguchi who in [14] conjectured that the TRS $\mathcal{R}_2$ is nv-sequential. With Autowrite one can easily check that $\mathcal{R}_2 \in \mathcal{CBN}_{\text{nv}}$ which we think implies that it is nv-sequential.

3.2 Properties related to signature extension

Let $\mathcal{R}$ be a left-linear growing eTRS. In [6,7] we have studied the question whether the property that $\mathcal{R} \in \mathcal{CBN}$ is preserved after adding new function symbols. For that problem, we need to specify the underlying signature in our notation. We write $(\mathcal{R}, \mathcal{G})$ instead of just $\mathcal{R}$ to indicate which signature is used. We write $\mathsf{NF}(\mathcal{R}, \mathcal{F})$ for the set of ground normal forms of an eTRS $\mathcal{R}$ over a signature $\mathcal{F}$. We denote by $\mathsf{WN}(\mathcal{R}, \mathcal{F})$ the set of all ground terms in $\mathcal{T}(\mathcal{F})$ that rewrite in $\mathcal{R}$ to a normal form in $\mathsf{NF}(\mathcal{R}, \mathcal{F})$. Let $\mathcal{F} \subseteq \mathcal{G}$. We denote by $\mathsf{WN}(\mathcal{R}, \mathcal{G}, \mathcal{F})$ the set of terms in $\mathcal{T}(\mathcal{F})$ that have a normal form with respect to $(\mathcal{R}, \mathcal{G})$. We write $\mathsf{WN}_\bullet(\mathcal{R}, \mathcal{G}, \mathcal{F})$ for $\mathsf{WN}(\mathcal{R}_\bullet, \mathcal{G}_\bullet, \mathcal{F}_\bullet)$. An easy (but unpublished) result states that for $(\mathcal{R}, \mathcal{F}) \in \mathcal{CBN}$, if $(\mathcal{R}, \{\mathcal{F} \cup @\}) \in \mathcal{CBN}$ (for some fresh constant @) then $(\mathcal{R}, \mathcal{G}) \in \mathcal{CBN}$ for any $\mathcal{G}$ such that $\mathcal{F} \subseteq \mathcal{G}$.

This why Autowrite provides the possibility of testing whether $(\mathcal{R}, \mathcal{G}) \in \mathcal{CBN}$ with $\mathcal{G} = \mathcal{F} \cup \{@\}$.

A normal form is *external* if it is not an instance of a proper non variable subterm of a left-handside of a rewrite rule in $\mathcal{R}$. The set of all ground external normal forms of a TRS $\mathcal{R}$ is denoted by $\mathsf{ENF}(\mathcal{R})$.

$\mathsf{ENF}(\mathcal{R}) \neq \emptyset$ is a sufficient condition for $\mathcal{R} \in \mathcal{CBN}$ being preserved $\mathcal{CBN}$ under signature extension.

When $\mathsf{ENF}(\mathcal{R}) = \emptyset$, orthogonality is needed in all the sufficient conditions that we have obtained.

A rewrite rule $l \to r$ is *collapsing* if $r \in \mathcal{V}$ and so is an eTRS containing a collapsing rule.

For orthogonal nv eTRSs, the condition that $\mathcal{R}$ is collapsing and $\mathsf{WN}(\mathcal{R}, \mathcal{G}, \mathcal{F}) = \mathsf{WN}(\mathcal{R}, \mathcal{F})$ is sufficient. Autowrite helped us find examples showing that both restrictions where essential.

The following example shows the necessity of the collapsing condition:

Example 4. Let $\mathcal{R}_4$ be the following orthogonal TRS:

$$\begin{array}{rcl} \mathsf{f}(x,\mathsf{a},\mathsf{b}(x_1,x_2)) \to \mathsf{c}(\mathsf{i}) & \quad & \mathsf{f}(\mathsf{c}(x),\mathsf{c}(x_1),x_2) \to \mathsf{i} \\ \mathsf{f}(x,\mathsf{a},\mathsf{c}(x_1)) \to \mathsf{i} & & \mathsf{g}(x) \to \mathsf{b}(x,\mathsf{i}) \\ \mathsf{f}(\mathsf{a},\mathsf{a},\mathsf{a}) \to \mathsf{i} & & \mathsf{h}(\mathsf{a}) \to \mathsf{i} \\ \mathsf{f}(\mathsf{a},\mathsf{b}(x,x_1),x_2) \to \mathsf{a} & & \mathsf{h}(\mathsf{b}(\mathsf{a},x)) \to \mathsf{a} \\ \mathsf{f}(\mathsf{a},\mathsf{c}(x),x_1) \to \mathsf{i} & & \mathsf{h}(\mathsf{b}(\mathsf{b}(x,x_1),x_2)) \to \mathsf{b}(\mathsf{i},\mathsf{i}) \\ \mathsf{f}(\mathsf{b}(x,x_1),x_2,\mathsf{a}) \to \mathsf{a} & & \mathsf{h}(\mathsf{b}(\mathsf{c}(x),x_1)) \to \mathsf{i} \\ \mathsf{f}(\mathsf{b}(x,x_1),\mathsf{b}(x_2,x_3),\mathsf{b}(x_4,x_5)) \to \mathsf{i} & & \mathsf{h}(\mathsf{c}(x)) \to \mathsf{i} \\ \mathsf{f}(\mathsf{b}(x,x_1),\mathsf{b}(x_2,x_3),\mathsf{c}(x_4)) \to \mathsf{i} & & \mathsf{j}(\mathsf{a},\mathsf{a}) \to \mathsf{i} \\ \mathsf{f}(\mathsf{b}(x,x_1),\mathsf{c}(x_2),\mathsf{b}(x_3,x_4)) \to \mathsf{i} & & \mathsf{j}(\mathsf{a},\mathsf{b}(x,x_1)) \to \mathsf{i} \\ \mathsf{f}(\mathsf{b}(x,x_1),\mathsf{c}(x_2),\mathsf{c}(x_3)) \to \mathsf{i} & & \mathsf{j}(\mathsf{a},\mathsf{c}(x)) \to \mathsf{i} \\ \mathsf{f}(\mathsf{c}(x),\mathsf{a},\mathsf{a}) \to \mathsf{i} & & \mathsf{j}(\mathsf{b}(x,x_1),x_2) \to \mathsf{i} \\ \mathsf{f}(\mathsf{c}(x),\mathsf{b}(x_1,x_2),\mathsf{a}) \to \mathsf{i} & & \mathsf{j}(\mathsf{c}(x),x_1) \to \mathsf{a} \\ \mathsf{f}(\mathsf{c}(x),\mathsf{b}(x_1,x_2),\mathsf{c}(x_3)) \to \mathsf{i} & & \mathsf{i} \to \mathsf{i} \\ \mathsf{f}(\mathsf{c}(x),\mathsf{b}(x_1,x_2),\mathsf{b}(x_3,x_4)) \to \mathsf{i} & & \end{array}$$

over the signature $\mathcal{F}$ consisting of all symbols appearing in the rewrite rules and let $\mathcal{G} = \mathcal{F} \cup \{@\}$.

Autowrite is able to check that

- $\mathsf{ENF}(\mathcal{R}_4) = \emptyset$,
- $(\mathcal{R}_4, \mathcal{F}) \in \mathcal{CBN}_{\mathrm{nv}}$,
- $(\mathcal{R}_4, \mathcal{G}) \notin \mathcal{CBN}_{\mathrm{nv}}$ as shown by the term with no $\mathrm{nv}(\mathcal{R}_4), \mathcal{G})$-needed redex $\mathsf{j}(\mathsf{f}(\Delta,\Delta,\Delta),@)$ with $\Delta = \mathsf{h}(\mathsf{g}(@))$,
- $\mathsf{WN}(\mathrm{nv}(\mathcal{R}_4), \mathcal{G}, \mathcal{F}) = \mathsf{WN}(\mathrm{nv}(\mathcal{R}_4), \mathcal{F})$.

One can verify easily that $\mathsf{j}(\mathsf{f}(\Delta,\Delta,\Delta),@)$ has no $\mathrm{nv}(\mathcal{R}_4), \mathcal{G})$-needed redex:

$\Delta = \mathsf{h}(\mathsf{g}(@)) \to_{\mathrm{nv}} \mathsf{h}(\mathsf{b}(\mathsf{a},\mathsf{i})) \to_{\mathrm{nv}} \mathsf{a}$ $\qquad$ $\Delta = \mathsf{h}(\mathsf{g}(@)) \to_{\mathrm{nv}} \mathsf{h}(\mathsf{b}(\mathsf{b}(\mathsf{a},\mathsf{a}),\mathsf{i})) \to_{\mathrm{nv}} \mathsf{b}(\mathsf{i},\mathsf{i})$

$$\begin{array}{ll} \mathsf{j}(\mathsf{f}(\bullet,\Delta,\Delta),@) \to_{\mathrm{nv}} \mathsf{j}(\mathsf{f}(\bullet,\mathsf{a},\mathsf{b}(\mathsf{i},\mathsf{i})),@) \to_{\mathrm{nv}} \mathsf{j}(\mathsf{c}(\mathsf{i}),@) \to_{\mathrm{nv}} \mathsf{a} & \in \mathsf{NF}(\mathcal{R},\mathcal{G}) \\ \mathsf{j}(\mathsf{f}(\Delta,\bullet,\Delta),@) \to_{\mathrm{nv}} \mathsf{j}(\mathsf{f}(\mathsf{b}(\mathsf{i},\mathsf{i}),\bullet,\mathsf{a}),@) \to_{\mathrm{nv}} \mathsf{j}(\mathsf{a},@) & \in \mathsf{NF}(\mathcal{R},\mathcal{G}) \\ \mathsf{j}(\mathsf{f}(\Delta,\Delta,\bullet),@) \to_{\mathrm{nv}} \mathsf{j}(\mathsf{f}(\mathsf{a},\mathsf{b}(\mathsf{i},\mathsf{i}),\bullet),@) \to_{\mathrm{nv}} \mathsf{j}(\mathsf{a},@) & \in \mathsf{NF}(\mathcal{R},\mathcal{G}) \end{array}$$

The next example in this section shows the necessity of the restriction to $\alpha \in \{\mathrm{s}, \mathrm{nv}\}$.

Example 5. Let $\mathcal{R}_5$ be the following orthogonal eTRS:

$$\begin{array}{rcl} \mathsf{f}(x,\mathsf{a},\mathsf{b}(x_1),x_2) \to \mathsf{g}(x_2) & \quad & \mathsf{g}(\mathsf{a}) \to \mathsf{i} \\ \mathsf{f}(\mathsf{b}(x),x_1,\mathsf{a},x_2) \to \mathsf{g}(x_2) & & \mathsf{g}(\mathsf{b}(x)) \to \mathsf{i} \\ \mathsf{f}(\mathsf{a},\mathsf{b}(x),x_1,x_2) \to \mathsf{g}(x_2) & & \mathsf{h}(\mathsf{a}) \to \mathsf{i} \\ \mathsf{f}(\mathsf{a},\mathsf{a},\mathsf{a},x) \to \mathsf{i} & & \mathsf{h}(\mathsf{b}(x)) \to \mathsf{j}(\mathsf{i},x) \\ \mathsf{f}(\mathsf{b}(x),\mathsf{b}(x_1),\mathsf{b}(x_2),x_3) \to \mathsf{i} & & \mathsf{j}(x,\mathsf{a}) \to \mathsf{a} \\ \mathsf{i} \to \mathsf{i} & & \mathsf{j}(x,\mathsf{b}(x_1)) \to \mathsf{b}(\mathsf{a}) \end{array}$$

over the signature $\mathcal{F}$ consisting of all symbols appearing in the rewrite rules. Note that the growing approximation only modifies the rule $\mathsf{j}(x,\mathsf{b}(x_1)) \to \mathsf{j}(\mathsf{i},x)$ into $\mathsf{j}(x,\mathsf{b}(x_1)) \to \mathsf{j}(\mathsf{i},y)$. Let $\mathcal{G} = \mathcal{F} \cup \{@\}$.

Autowrite is able to check that

- $\mathsf{ENF}(\mathcal{R}_5) = \emptyset$,
- $(\mathcal{R}_5, \mathcal{F}) \in \mathcal{CBN}_{\mathrm{g}}$,
- $(\mathcal{R}_5, \mathcal{G}) \notin \mathcal{CBN}_{\mathrm{g}}$ as shown by the term with no $\mathrm{gr}(\mathcal{R}_5)$-needed redex $\mathsf{f}(\varDelta, \varDelta, \varDelta, @)$, with $\varDelta = \mathsf{h}(\mathsf{j}(@))$,
- $\mathsf{WN}(\mathrm{gr}(\mathcal{R}_5), \mathcal{F}) = \mathsf{WN}(\mathrm{gr}(\mathcal{R}_5), \mathcal{G}, \mathcal{F})$.

Note that $\mathcal{R}$ is not collapsing. This is not essential, since adding the single collapsing rule $\mathsf{k}(x) \to x$ to $\mathcal{R}$ does not affect any of the above properties.

For an nv eTRS $\mathcal{R}$, we have the nice property that $\mathsf{WN}(\mathcal{R}, \mathcal{G}, \mathcal{F}) = \mathsf{WN}(\mathcal{R}, \mathcal{F}) \Rightarrow \mathsf{WN}_\bullet(\mathcal{R}, \mathcal{G}, \mathcal{F}) = \mathsf{WN}_\bullet(\mathcal{R}, \mathcal{F})$. Autowrite helped us show that the restriction to nv is essential for this implication.

Example 6. Let $\mathcal{R}_6$ be the following orthogonal TRS:

$$\begin{array}{rlrl}
\mathsf{f}(x, \mathsf{a}) & \to \mathsf{a} & \mathsf{h}(x, \mathsf{a}, \mathsf{a}) & \to \mathsf{i} \\
\mathsf{f}(\mathsf{a}, \mathsf{b}(x)) & \to \mathsf{i} & \mathsf{h}(x, \mathsf{a}, \mathsf{b}(x_1)) & \to \mathsf{i} \\
\mathsf{f}(\mathsf{b}(x), \mathsf{b}(x_1)) & \to \mathsf{i} & \mathsf{h}(x, \mathsf{b}(x_1), \mathsf{a}) & \to \mathsf{i} \\
\mathsf{g}(\mathsf{a}, \mathsf{a}) & \to \mathsf{i} & \mathsf{h}(x, \mathsf{b}(x_1), \mathsf{b}(x_2)) & \to \mathsf{b}(\mathsf{g}(x_1, \mathsf{f}(x, x_2))) \\
\mathsf{g}(\mathsf{b}(x), \mathsf{a}) & \to \mathsf{i} & \mathsf{i} & \to \mathsf{b}(\mathsf{i}) \\
\mathsf{g}(x, \mathsf{b}(x_1)) & \to \mathsf{a} & &
\end{array}$$

over the signature $\mathcal{F}$ consisting of all symbols appearing in the rewrite rules and let $\mathcal{G} = \mathcal{F} \cup \{@\}$.

Autowrite is able to check

- $\mathsf{ENF}(\mathcal{R}_6) = \emptyset$,
- $\mathsf{WN}(\mathrm{gr}(\mathcal{R}_6), \mathcal{G}, \mathcal{F}) = \mathsf{WN}(\mathrm{gr}(\mathcal{R}_6), \mathcal{F})$,
- $\mathsf{WN}_\bullet(\mathrm{gr}(\mathcal{R}_6), \mathcal{G}, \mathcal{F}) \neq \mathsf{WN}_\bullet(\mathrm{gr}(\mathcal{R}_6), \mathcal{F})$ as shown by the term $t = \mathsf{h}(\bullet, \mathsf{i}, \mathsf{i})$.

4 The inside of Autowrite

The most important object in Autowrite is the tree automaton. Since the first version of Autowrite [4], much care has been devoted to improve the representation of automata. Consequently, the performances have improved significantly.

Each state of an automaton is represented by a unique Common Lisp object. Comparing two states is then very cheap: we just need to compare the references of the states. An automaton is represented by its signature (a list of symbols), a list of references to its states and its rules. A TRS is represented by its signature and its rules. The set of rules (of an automaton or a TRSs) is represented by a hash-table which given a key associated with a left-hand side of a rule gives the corresponding right-handside (or a list of corresponding right-handsides if the automaton is not deterministic). Given a left-hand side $f(q_1, \ldots, q_n)$, the corresponding key consists of a list containing the root symbol f followed by the references of the states $q_1, \ldots, q_n$.

During the construction of an automaton (for instance during the construction of the $\mathcal{C}_{\mathcal{R},\mathcal{A}}$) the rules of an automaton may be represented as a simple list of rules. But as soon as the construction is completed, the list of rules is converted into a hash-table as described above.

In general we use as much as possible '"sharing" versus "copying" data structures and use hashtables instead of lists. When possible we use memoizing techniques to avoid recomputing several times identical calls. The later may explain differences of timing when the same operations are performed in different order.

5 The outside of Autowrite

5.1 Automata operations performed by Autowrite

Checking properties of an automaton Here are the different decision problems about an automaton that can be solved with Autowrite.

- Given an automaton $\mathcal{A}$: decide whether $L(\mathcal{A})$ is empty.
- Given two automata $\mathcal{A}$ and $\mathcal{B}$: decide whether $L(\mathcal{A}) \subseteq L(\mathcal{B})$.
- Given two automata $\mathcal{A}$ and $\mathcal{B}$: decide whether $L(\mathcal{A}) = L(\mathcal{B})$.
- Given two automata $\mathcal{A}$ and $\mathcal{B}$: decide whether $L(\mathcal{A}) \cap L(\mathcal{B})$ is empty.
  For this latter operation the intersection automaton is not computed, rather we incrementally compute its accessible states and stop as soon as a final state is found.

When a property is not satisfied Autowrite exhibits a ground term exposing the failure.

Building new automata Here are the different automata transformations or constructions handled by Autowrite.

- Given an automaton $\mathcal{A}$: compute $Det(\mathcal{A})$, the determinized version of $\mathcal{A}$.
- Given an automaton $\mathcal{A}$: compute $\mathcal{A}^c$ recognizing $L(\mathcal{A})^c$ the complement of $L(\mathcal{A})$ in the whole set of ground terms.
- Given an automaton $\mathcal{A}$: compute $Red(\mathcal{A})$, the *reduced* version of $\mathcal{A}$ *i.e.* such that every state is accessible.
- Given two automata $\mathcal{A}$ and $\mathcal{B}$: compute $\mathcal{A} \cap \mathcal{B}$.
- Given two automata $\mathcal{A}$ and $\mathcal{B}$: compute $\mathcal{A} \cup \mathcal{B}$.
- Given a set of linear terms $\mathcal{L}$: compute an automaton $\mathcal{A}_{\mathcal{L}}$ such that
  $L(\mathcal{A}_{\mathcal{L}}) = \{\sigma(t) \mid t \in \mathcal{L} \text{ and } \sigma \text{ is a ground substitution }\}$.

5.2 Building automata related to a left-linear eTRSs

Let $\mathcal{R}$ be a left-linear eTRS. Autowrite can build the following automata:

- Build an automaton $\mathcal{A}_{\mathsf{NF}(\mathcal{R})}$ such that $L(\mathcal{A}_{\mathsf{NF}(\mathcal{R})}) = \mathsf{NF}(\mathcal{R})$.
- Build an automaton $\mathcal{A}_{\mathsf{ENF}(\mathcal{R})}$ such that $L(\mathcal{A}_{\mathsf{ENF}(\mathcal{R})}) = \mathsf{ENF}(\mathcal{R})$.

  The two following automata can be constructed only if $\mathcal{R}$ also growing:

- Given a tree automaton $\mathcal{A}$: build a deterministic (Toyama and Nagaya's algorithm) or non-deterministic (Jaquemard's algorithm) automaton $\mathcal{C}_{\mathcal{R},\mathcal{A}}$ (as described in [13]) such that $L(\mathcal{C}_{\mathcal{R},\mathcal{A}}) = (\xrightarrow{*})[L(\mathcal{A})]$.
- Build an automaton $\mathcal{D}_{\mathcal{R}}$ such that $L(\mathcal{D}_{\mathcal{R}}) = \emptyset$ is equivalent to $\mathcal{R} \in \mathcal{CBN}$ [5].

For $\mathcal{C}_{\mathcal{R},\mathcal{A}}$, both Jacquemard's algorithm for linear-growing systems and Toyama and Nagaya's for left-linear-growing systems have been implemented. For $\mathcal{D}_{\mathcal{R}}$, we have implemented the algorithm presented in [5]. In fact these three algorithms have been adapted in order to directly compute automata with only accessible states. This complicates the code but reduces considerably the size of the construction.

The main idea is to compute the automaton incrementally. We start start building the rules having a constant left-hand side. This gives the first set of accessible states. Then we compute the rules whose left-handsides contain the current accessible states which may give new accessible states. We stop when no new accessible state is created.

5.3 General properties of eTRS

These are easy properties but may be useful for checking big TRSs.

- Check whether a TRS is left-linear
- Check whether a TRS is overlapping
- Check whether a TRS is orthogonal
- Check whether a TRS is collapsing

5.4 Properties of left-linear eTRSs

Let $\mathcal{R}$ be a left-linear eTRS. The first set of properties concern only the left-hand sides of $\mathcal{R}$.

- Decide whether the set of normal forms is empty.

- Decide whether the set of external normal forms is empty.
  We will see in section 3.2 that non-emptiness of $\mathsf{ENF}(\mathcal{R})$ is a sufficient condition for preserving the property that $\mathcal{R} \in \mathcal{CBN}$ when the signature is extended.

Much more interesting problems can be solved when we consider left-linear **growing** eTRSs:

- Given a tree automaton $\mathcal{A}$ and a term t, decide whether $t \in (\rightarrow^*_{\mathcal{R}})[L(\mathcal{A})]$.
  This is done by computing $\mathcal{C}_{\mathcal{R},\mathcal{A}}$ (see section 5.2) and check whether t is recognized by $\mathcal{C}_{\mathcal{R},\mathcal{A}}$.
  Note that this solves the accessibility problem (given two terms t, s, does $t \rightarrow^*_{\mathcal{R}} s$) as a single term s forms a regular language recognizable by a tree automaton.
- Decide whether $\mathcal{R} \in \mathcal{CBN}$.
  The method consists of building the automaton $\mathcal{D}_{\mathcal{R}}$ (see section 5.2) and then check whether $L(\mathcal{D}_{\mathcal{R}}) = \emptyset$. If so Autowrite concludes that $\mathcal{R} \in \mathcal{CBN}$, otherwise it exhibits a ground term of $L(\mathcal{D}_{\mathcal{R}})$ which is a term with no $\mathcal{R}$-needed redex. Note that to build $\mathcal{D}_{\mathcal{R}}$, Autowrite must previously compute $\mathcal{C}_{\mathcal{R},\mathcal{A}_{\mathsf{NF}(\mathcal{R})}}$.
- Decide whether $\mathcal{R}$ is *arbitrary i.e.* whether there exists a ground term $t \in \mathcal{T}(\mathcal{F})$ such that $t \rightarrow^*_{\mathcal{R},\mathcal{F}} \bullet$ (this means that there exists a term that may reduce to any other term). That latter property is relevant for the problem of signature extension (see section 3.2).

Concerning checking that $\mathcal{R} \in \mathcal{CBN}$, one cannot hope to use Autowrite for big TRSs because the size of the constructed automaton $\mathcal{D}_{\mathcal{R}}$ is in $\mathcal{O}(2^{2^{\|\mathcal{R}\|}})$ as shown in [5].

6 Experimental results

In the following table we present a few results with various approximations of the above TRSs. We present the number of states (st) and rules (rl) of the automata $\mathcal{C}_{\alpha(\mathcal{R})}$ and $\mathcal{D}_{\alpha(\mathcal{R})}$ built to decide whether $\mathcal{R} \in \mathcal{CBN}_\alpha$. If the TRS is not in $\mathcal{CBN}_\alpha$, we give the witness term with no $\alpha(\mathcal{R})$-needed redex found by Autowrite. The last three columns concern the computation of the non-deterministic automaton $\mathcal{C}_{\alpha(\mathcal{R})}$ with Jacquemard's algorithm which is applicable only in the linear case. Due practical reasons, the following times were obtained on different machines.

$\mathcal{R}$	α	$\mathcal{C}_{\alpha(\mathcal{R})}$ Toyama			$\mathcal{D}_{\alpha(\mathcal{R})}$			$(\mathcal{R},\mathcal{F}) \in \mathcal{CBN}_\alpha$	$\mathcal{C}_{\alpha(\mathcal{R})}$ Jacquemard		
		st	rl	Time	st	rl	Time		st	rl	Time
$\mathcal{R}_1,\mathcal{F}$	s	17	584	4.24s	32	727	0.47s	$\mathsf{f(g(g(e,e),g(e,e)),g(e,e))}$	12	342	14.79s
	nv	21	888	1m9s	45	4055	4.17s	Yes	12	332	14.39s
	gr	29	1688	5m12	174	60557	8m56s	Yes	12	200	8.63s
$\mathcal{R}_2,\mathcal{F}$	s	16	548	18s	32	687	0.4s	$\mathsf{f(g(h(b),h(b)),h(b))}$	11	380	9.12s
	nv	11	268	5s	17	615	0.26s	Yes	11	179	1.84s
$\mathcal{R}_3,\mathcal{F}$	s	7	409	24.44s	11	162	0.12s	$\mathsf{f(g(b,a),g(b,a),g(b,a))}$	6	213	5.76s
	nv	9	831	1m31s	20	594	0.46	$\mathsf{f(f(b,b,a),f(b,b,a),f(b,b,a))}$	6	162	4.06s
	gr	6	267	8.15s	17	5238	5.21s	Yes	NA	NA	NA
$\mathcal{R}_4,\mathcal{F}$	s	14	3181	58m11s	12	24	0.12s	$\mathsf{f(i,i,i)}$	11	1329	2m3s
	nv	18	6537	58m2s	85	628832	40m6s	Yes	11	249	5.7s
$\mathcal{R}_4,\mathcal{G}$	nv	26	19010	8h53m	115	353013	26m	$\mathsf{j(f(h(g(@),h(g(@),h(g(@)),@)}$	13	434	15.07s
$\mathcal{R}_5,\mathcal{F}$	s	5	668	23.65s	8	28	0.08s	$\mathsf{f(i,i,i,a)}$	4	286	13.53s
	nv	5	688	48.53s	19	130741	3m3s	Yes	4	94	1.25s
$\mathcal{R}_5,\mathcal{G}$	nv	6	1354	4m5s	25	160463	3m44s	$\mathsf{f(h(b(a)),h(b(a)),h(b(a)),@)}$	5	260	7.99s
$\mathcal{R}_6,\mathcal{F}$	s	5	183	2s	8	650	0.4s	Yes	4	125	1.99s

Here we report the time taken by tests $\mathsf{WN}(\mathcal{R},\mathcal{G},\mathcal{F}) = \mathsf{WN}(\mathcal{R},\mathcal{F})$ and $\mathsf{WN}_\bullet(\mathcal{R},\mathcal{G},\mathcal{F}) = \mathsf{WN}_\bullet(\mathcal{R},\mathcal{F})$ assuming that the correponding $\mathcal{C}_{\alpha(\mathcal{R})}$ automata have already been constructed. The time corresponds then merely to checking both inclusions.

$\mathcal{R}$	α	$\mathsf{WN}(\mathcal{R},\mathcal{G},\mathcal{F}) = \mathsf{WN}(\mathcal{R},\mathcal{F})$	Time	$\mathsf{WN}_\bullet(\mathcal{R},\mathcal{G},\mathcal{F}) = \mathsf{WN}_\bullet(\mathcal{R},\mathcal{F})$	Time
$\mathcal{R}_5$	gr	Yes	0.88s	Yes	0.67s
$\mathcal{R}_6$	nv	Yes	36.33s	Yes	36.84s
	gr	Yes	8.28s	$\mathsf{h(\bullet,i,i)}$	8.48s

7 Comparison with other systems

We are aware of two other distributed tools implementing tree automata: Timbuk [8] and RX [16]. Timbuk requires the installation of `ocaml` and `RX` requires the installation of `ghc` while Autowrite comes self-contained. We were able able to use Timbuk (easier to install than RX). Timbuk was initially designed for computing over-approximations of the set of descendants $(\rightarrow^*_\mathcal{R})[L]$ for a regular language L and a TRS $\mathcal{R}$ and then, use it to prove unreachability. Autowrite can be used to check reachability (whether some term $t \in (\rightarrow^*_\mathcal{R})[L]$) but only for left-linear growing systems. Timbuk can handle some non-growing or non-linear cases. However, concerning efficiency of tree automata operations, Autowrite seems much faster: we have tried the determinization of the automaton $\mathcal{C}_{\mathrm{nv}(\mathcal{R}_5)}$ computed by Jacquemard's algorithm which runs in 2 seconds with Autowrite and took about 3 hours with Timbuk. The lattest version of Autowrite is able to load Timbuk specifications defining TRSs, sets of terms and automata.

8 Practical information and perspectives

The Autowrite project has a web page:

`http://dept-info.labri.u-bordeaux.fr/~idurand/autowrite`

¿From that page one can download the graphical version of Autowrite. The file is rather big because it contains a big part of Common Lisp and McCLIM. But the advantage is that Autowrite is self-contained and requires no other software. The Autowrite sources contain about 6000 lines of Common Lisp (including the graphical interface). On the Web page one can find installation directives, an on-line User's Guide and useful links. The example of an Autowrite session should be useful for a new user. The code can still be improved for better performances. We plan to add the possibility of minimizing a tree automaton and to work on any interesting suggestion that users might make.

References

1. H. Comon. Sequentiality, monadic second-order logic and tree automata. *Information and Computation*, 157:25–51, 2000.
2. H. Comon, M. Dauchet, R. Gilleron, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications, 1998. Draft, available from `http://www.grappa.univ-lille3.fr/tata/`.
3. I. Durand and A. Middeldorp. Decidable call by need computations in term rewriting (extended abstract). In *Proc. 14th CADE*, volume 1249 of *LNAI*, pages 4–18, 1997.
4. Irène Durand. Autowrite: A tool for checking properties of term rewriting systems. In *Proceedings of the 13th International Conference on Rewriting Techniques and Applications*, volume 2378 of *Lecture Notes in Computer Science*, pages 371–375, Copenhagen, 2002. Springer-Verlag.
5. Irène Durand and Aart Middeldorp. On the complexity of deciding call-by-need. Technical Report 1196–98, LaBRI, 1998.
6. Irène Durand and Aart Middeldorp. On the modularity of deciding call-by-need. In *Foundations of Software Science and Computation Structures*, volume 2030 of *Lecture Notes in Computer Science*, pages 199–213, Genova, 2001. Springer-Verlag.
7. Irène Durand and Aart Middeldorp. Decidable call-by-need computations in term rewriting. *Submitted to Information and Computation*, 2003.
8. Thomas Genêt and Valérie Viet Triem Tong. Reachability analysis of term rewriting systems with timbuk. In *Proc. 8th LPAI*, volume 2250 of *LNAI*, pages 691–702. Springer-Verlag, 2001.
9. G. Huet and J.-J. Lévy. Computations in orthogonal rewriting systems, i and ii. In *Computational Logic, Essays in Honor of Alan Robinson*, pages 396–443. The MIT Press, 1991. Original version: Report 359, Inria, 1979.
10. F. Jacquemard. Decidable approximations of term rewriting systems. In *Proc. 7th RTA*, volume 1103 of *LNCS*, pages 362–376, 1996.
11. J.W. Klop. Term rewriting systems. In *Handbook of Logic in Computer Science, Vol. 2*, pages 1–116. Oxford University Press, 1992.
12. J.W. Klop and A. Middeldorp. Sequentiality in orthogonal term rewriting systems. *Journal of Symbolic Computation*, 12:161–195, 1991.
13. T. Nagaya and Y. Toyama. Decidability for left-linear growing term rewriting systems. *Information and Computation*, 178(2):499–514, 2002.
14. M. Oyamaguchi. NV-sequentiality: A decidable condition for call-by-need computations in term rewriting systems. *SIAM Journal on Computation*, 22:114–135, 1993.
15. Y. Toyama. Strong sequentiality of left-linear overlapping term rewriting systems. In *Proc. 7th LICS*, pages 274–284, 1992.
16. J. Waldmann. Rx: an interpreter for rational tree languages. `http://www.informatik.uni-leipzig.de/~joe/rx/`, 1998.

Integrating Decision Procedures in Reflective Rewriting-Based Theorem Provers*

Manuel Clavel, Miguel Palomino, and Juan Santa-Cruz

Departamento de Sistemas Informáticos y Programación
Universidad Complutense de Madrid, Spain
{clavel,miguelpt,juansc}@sip.ucm.es

Abstract We propose a design for the integration of decision procedures in reflective rewriting-based equational theorem provers. Rewriting-based equational theorem provers use term rewriting as their basic proof engine; they are particularly well suited for proving properties of equational specifications. A reflective rewriting-based theorem prover is itself an executable equational specification, which has reflective access to the rewriting engine responsible of its execution to efficiently accomplish rewriting-based equational proofs. This reflective access means that the built-in rewriting engine can be called with different rewriting commands. This opens up the possibility of interpolating calls to appropriate decision procedures in the midst of a rewriting-based proof step —typically to solve a condition in the application of a conditional equation. To illustrate our proposal and show its feasibility, we explain how our reflective design ideas can be used to integrate a decision procedure for Presburger arithmetic in the ITP tool, a reflective rewriting-based equational theorem prover, written entirely in Maude, for proving properties of Maude equational specification.

1 Introduction

Many authors have stressed the importance of integrating decision procedures, that is, algorithms that for particular theories can automatically decide whether a given formula is valid or not, in the proof engines of (semi-)automated theorem provers. As Boyer and Moore wrote [3]: "It is generally agreed that when practical theorem provers are finally available they will contain both heuristic components and many decision procedures." Decision procedures are indeed at the core of many industrial-strength verification systems such as ACL2 [16], PVS [22], STeP [18], or Z/Eves [23]. The crucial role of decision procedures motivates the development of ICS [10], an efficient decision procedure for a fragment of first-order logic, that can be used as a standalone application and may also be included as a library in any application that requires embedded deduction.

Rewriting-based theorem provers [12,19,11,15] use term rewriting [1] as their basic proof engine, and they are particularly useful for proving properties of equational specifications. As is well known, when a finite equational specification is Church-Rosser and terminating, term rewriting can be used as an efficient procedure for deciding equalities between terms: two terms are provably equal if and only if their canonical forms, which are computed by using the equations as simplifications rules, are syntactically identical. There are, however, many practical equational specifications that contain function symbols that are not equationally defined. Consider, for example, those specifications that use first-order arithmetic: typically, they do not contain the equational definitions of the arithmetic function symbols and relations. This omission poses no problem when *evaluating* functional expressions not containing indeterminate values, since practical executable specification languages [7,12,9] provide internal links to built-in implementations of the arithmetic functions and relations. The problem arises, however, when *proving* properties.[1] In general, these proofs require solv-

* Research supported by Spanish MCYT Projects MELODIAS TIC2002-01167 and MIDAS TIC2003-01000.

[1] The inequality function symbol `_=/=_`, that is provided as a built-in function in [7,12], gives rise to similar complications, as discussed in detail in [13, Sec.2.1.1].

ing arithmetic formulas containing indeterminate values. In those situations, the built-in implementations of the arithmetic functions and relations are useless, and the lack of equations explicitly defining them prevents us from applying term rewriting.[2] In some cases, however, we can overcome the difficulty in a general, non-ad hoc form by calling appropriate decision procedures. This is the case, for example, when the formula to be solved falls within the class of Presburger linear arithmetic formulas.

In summary, decision procedures are also important for practical rewriting-based theorem provers, and they must be integrated with their basic rewriting engines in such a way that calls to the appropriate decision procedure can be interpolated in the midst of rewriting-based proof steps. The design and implementation of the RRL [15] reflects indeed the relevance of decision procedures in rewriting-based theorem provers, and the experiments reported in [14] show that "the use of the procedure for Presburger arithmetic has made the proofs compact and relatively easier to automate and understand in contrast to proofs generated without using Presburger arithmetic." In this paper we propose a novel *reflective design* for the integration of decision procedures in rewriting-based equational theorem provers. Although our proposal can be formulated in a general form using the axiomatic definitions of reflective logics and reflective programming languages [5], we rather illustrate it here by explaining the integration of a decision procedure for Presburger arithmetic in the ITP tool.

The ITP tool [4] is an experimental rewriting-based theorem prover for proving properties of equational specifications; it accepts equational specifications presented as functional modules of the Maude system [7]. The equational logic on which Maude functional modules are based is an extension of order-sorted equational logic called membership equational logic [21]. Thus functional modules support multiple sorts, subsort relations, operator overloading, and assertions of membership in a sort. In addition, operators can be declared with any combination of the following equational attributes: associativity, commutativity, idempotency, and identity.[3] In the presence of equational attributes, equational simplification using the other equations in the module does not take place at the purely syntactic level, but is understood *modulo* those equational attributes. The current Maude implementation can execute syntactic rewriting with typical speeds from half a million to several million rewrites per second. Similarly, associative and associative-commutative equational rewriting with term patterns used in practice can be performed at the typical rate of several hundred thousand rewrites per second.

A key feature of the ITP is its reflective design. The tool is written entirely in Maude and is in fact an executable specification of the formal inference system that it implements. Maude supports reflective computations through a predefined module called `META-LEVEL`, which includes different built-in functions providing direct access to the Maude rewriting engine itself. The ITP tool extends the module `META-LEVEL` with equationally defined functions defining the effect of the different proof commands, and uses the built-in functions

[2] Notice also that, as pointed out in [13], "in fact, there is no set of equations that can allow the automatic verification of *all* properties of integer expressions which contain indeterminate values [...]; in other words, first order arithmetic is 'undecidable' [20]."

[3] The one restriction is that the idempotency attribute cannot be used together with the associativity attribute, since the combination of these two attributes is not currently supported by the Maude implementation.

provided by `META-LEVEL` to efficiently accomplish rewriting-based equational proofs.[4] This direct access to the Maude rewriting engine can justify in itself the reflective design of the ITP. But the reflective capabilities of Maude provide also the possibility of defining *rewriting commands* different from the Maude's default rewriting command, and of applying them to specific terms in the midst of a rewriting computation. There is in fact great freedom for defining different rewriting commands inside Maude, with ease and competitive performance, since they will typically use the built-in functions in the module `META-LEVEL` as the basic components of their definitions [5,6]. In particular, it is possible to define a rewriting command that calls the appropriate decision procedures, in the midst of a rewriting-based proof step, to solve, for example, a condition in a conditional equation. This capability is the base for the integration of decision procedures in the ITP basic rewriting proof engine.

Organisation The paper is organised as follows. In Section 2 we introduce the ITP tool, its default rewriting command (which is directly based on Maude's default rewriting command) and its limitations. In Section 3 we describe the implementation of a different, more granular rewriting command. In Section 4 we explain how a decision procedure for Presburger arithmetic, written in Maude, can be integrated with the rewriting command introduced in Section 3. This extended rewriting command makes appropriate calls to the decision procedure when the condition of a conditional equation falls within the class of Presburger linear arithmetic formulas. Finally, in Section 5, we conclude with a description of the current state of the ITP tool, and of our plans for further extending the tool with other decision procedures.

2 The ITP tool and its default rewriting command

The ITP tool [4] is an experimental interactive theorem prover, written in Maude, for proving properties of Maude [6,7] functional modules, which are equational theories in membership equational logic [21]. The fact that membership equational logic is a reflective logic [8], and that Maude efficiently supports reflective membership equational logic computations is systematically exploited in the tool. Maude supports reflective computations through its predefined `META-LEVEL` module. In this module, Maude functional modules can be metarepresented as terms of a certain sort, which can then be efficiently manipulated and transformed by appropriate built-in functions. In particular, the module `META-LEVEL` includes a built-in function `metaReduce` that can be used to reduce a term in a functional module to canonical form, and a built-in function `metaXmatch` that can be used to try to match two terms in a functional module. The function `metaReduce` takes the metarepresentations of a module M and a term t, and it returns the metarepresentation of the fully reduced form of t, using the equations in M, together with the metarepresentation of its corresponding sort. The reduction strategy used by `metaReduce` coincides with that of the `reduce` command in Maude. The function `metaXmatch` takes the metarepresentations of a module M and two terms, t_1 and t_2 (and four more arguments that we can safely ignore here), and tries to match t_1 with any subterm of t_2 in the module M. If successful, it returns the representations of a substitution σ and a context C such that $C[\sigma(t_1)] \equiv t_2$, where we use $\equiv$ to denote equality *modulo* the equational attributes declared in the module for

[4] A typical metalevel computation only pays the cost (linear in the size of the term) of changing the representation from the metalevel to the object level and back only at the beginning and at the end of the computation [7].

the operators involved; otherwise, the result is `noMatch` A full description of the module `META-LEVEL` can be found in [7, Chapter 10].

The ITP tool extends the module `META-LEVEL` with sorts for representing both the ITP goals and the ITP database of modules about which those goals have to be proved. In addition, it contains equationally defined functions specifying the effects of the different proof commands, like commands for carrying out inductive proofs, proofs based on case analysis, or proofs by rewriting. The basic ITP proof command is the `rwr` command that rewrites both sides of an equality to canonical form, using the equations contained in the module associated to the goal as simplification rules. As expected, the function that implements the `rwr` command directly calls the built-in function `metaReduce` to efficiently accomplish its task. And this, of course, may be sufficient when the equations in the module associated to the goal are Church-Rosser and terminating, and all the functions declared in that module are equally defined. There are, however, many practical equational specifications that do not satisfy such conditions. Consider, for example, the following specification in Maude of a sorting algorithm for lists of integers; the specification of the `length` function is included here for the sake of the example below.

```
fmod INS-SORT is
  protecting INT .
  sorts List .
  op nil : -> List [ctor] .
  op _:_ : Int List -> List [ctor] .
  op ins : Int List -> List .
  op sort : List -> List .
  op sorted : List -> Bool .
  op length : List -> Int .
  vars N M : Int . var L : List .
  --- ins
  eq ins(N, nil) = N : nil .
  ceq ins(N, M : L) = N : M : L
      if N <= M = true .
  ceq ins(N, M : L) = M : ins(N, L)
      if N > M = true .
  --- sort
  eq sort(nil) = nil .
  eq sort(N : L) = ins(N, sort(L)) .
  --- length
  eq length(nil) = 0 .
  eq length(N : L) = 1 + length(L) .
endfm
```

The module `INS-SORT` imports, through its `protecting` declaration, the predefined module `INT` that defines the integers with the expected arithmetic functions and relations; as usual, the latter are defined as Boolean functions. The module `INT` does not contain, however, the equational specification of the arithmetic functions but rather it provides internal links to built-in implementations of those functions. The module `INS-SORT` also imports, by default, the predefined module `BOOL` that defines the Boolean values. In this situation, the following property can easily be proved using Skolemization and term rewriting:

$$\forall\{N\}(\mathtt{sort}(N\,\mathtt{:nil}) = N\,\mathtt{:nil}) \tag{1}$$

since, for `N*` a *new* (Skolem) constant of sort `Int`, the canonical form of `sort(N*:nil)` is syntactically identical to `N*:nil`. Consider, however, the following property about `INS-SORT`:

$$\forall\{N, M\}(\mathtt{length}(\mathtt{ins}(N, M\,\mathtt{:nil})) = 2 \tag{2}$$

Again, by Skolemization, it will be sufficient to prove that

$$\mathtt{length}(\mathtt{ins}(\mathtt{N*}, \mathtt{M*:nil})) = 2 \tag{3}$$

for `N*` and `M*` new (Skolem) constants of sort `Int`. Given the conditional specification of `ins`, term rewriting will be useless at this point since no equations can be applied, and it is necessary to split the goal, using a Boolean-case analysis principle, into two subgoals that become associated to two different extensions of `INS-SORT`: one in which we add to `INS-SORT` the equation `N* <= M*` = `true`, and another in which the equation that we add is `N* <= M*` = `false`. In the first case, the proof is easily completed by applying term rewriting. However, in the second case, term rewriting will still remain useless since, in order to apply the second equation that specifies `ins`, we must prove first that `N* > M*` = `true`. But this cannot be proved simply by term rewriting, even when the module associated to this subgoal contains the equation `N* <= M*` = `false`, since `INS-SORT` does not contain any equation specifying `>`. Of course, for this particular case, the problem has two easy solutions: either we add to `INS-SORT` the equation `N > M` = `true if N <= M` = `false`, or we replace the second equation specifying `ins` by the conditional equation `ins(N, M : L)` = `M : ins(N, L) if N <= M` = `false`. But both solutions are manifestly ad-hoc.[5] A more general solution consists in integrating the appropriate decision procedure in the function implementing the `rwr` command. And this can be easily accomplished in a reflective rewriting-based theorem prover, as we show in the next two sections.

3 A different, non-default rewriting command for the ITP

In Section 2 we explained the implementation of the ITP default rewriting command `rwr`: how it uses the built-in function `metaReduce`, provided in the module `META-LEVEL`, to efficiently accomplish its task; and how its applicability is limited by the fact that `metaReduce` reduces terms in a module to canonical form using exclusively Maude's `reduce` command. In this section we show how the built-in function `metaXmatch`, also provided in the module `META-LEVEL`, allows us to implement, with ease and efficiency, a different, more granular rewriting command, `nrwr`, which does not call Maude's `reduce` command and includes, in particular, the implementation of the process of solving conditions when a conditional

[5] Solutions of this sort can be found, however, in the literature. In the otherwise excellent textbook [13], the authors propose an extension of the OBJ's built-in representation of the integers with an equality predicate and with some equations that are useful for manipulating inequalities. In particular, these equations are useful as lemmas in the correctness proofs given in the book. But the authors warn the reader that they are not strong enough to allow all properties of integers to be proved by reduction. In general, they say, if a property of the integers is needed for a correctness proof, then an appropriate equation will need to be added as a lemma for the proof.

equation is applied to a term. In Section 4 we then explain how the implementation of `nrwr` can be easily extended to a command `xrwr` in order to integrate decision procedures in the rewriting process, to solve, in particular, linear arithmetic conditions in the application of a conditional equation. A detailed explanation of how to define in Maude, using its reflective capabilities, different rewriting commands can be found in [7, Chapter 10].

Like all ITP commands, the `nrwr` rewriting command is implemented in the ITP tool by equationally defining a function in an extension of the module `META-LEVEL`. We call *red* the function that implements the `nrwr` command. Like `metaReduce`, *red* takes the metarepresentations of a module and a term as arguments, and returns the metarepresentation of the reduced term; also as `metaReduce`, equations are applied in *red* only from left to right. We use the symbol $\triangleq$ for definitional equality.

$$\mathit{red}(M, t) \triangleq \mathit{redAux}(M, t, \mathit{getEqs}(M))$$

The function *getEqs* extracts from the metarepresentation of module M its set of equations; we omit here its definition. The behaviour of the auxiliary function *redAux* is simple. For each equation in the module it tries to match its left-hand side with any subterm of the term being reduced: if there is no match, it discards the equation; otherwise it reduces the term accordingly (after solving the condition if it is a conditional equation) and restarts the process again from the beginning. Its implementation is immediate using the built-in function `metaXmatch`.

$$\begin{aligned}
&\mathit{redAux}(M, t, \emptyset) \triangleq t\\
&\mathit{redAux}(M, t, \{l = r\} \cup \mathit{Eq}) \triangleq \begin{cases} \mathit{red}(M, C[\sigma(r)]) & \text{if } t \equiv C[\sigma(l)] \\ \mathit{redAux}(M, t, \mathit{Eq}) & \text{otherwise} \end{cases}\\
&\mathit{redAux}(M, t, \{l = r \textbf{ if } \mathit{Cond}\} \cup \mathit{Eq}) \triangleq \begin{cases} \mathit{red}(M, C[\sigma(r)]) & \text{if } t \equiv C[\sigma(l)] \text{ and} \\ & \qquad \mathit{solveCond}(M, \sigma(\mathit{Cond})) \\ \mathit{redAux}(M, t, \mathit{Eq}) & \text{otherwise} \end{cases}
\end{aligned}$$

The function *solveCond* tries to solve the equations in the condition by just reducing both sides as much as possible using *red*.

$$\begin{aligned}
&\mathit{solveCond}(M, \emptyset) \triangleq \mathit{true}\\
&\mathit{solveCond}(M, \{l = r\} \cup \mathit{Eq}) \triangleq \begin{cases} \mathit{solveCond}(M, \mathit{Eq}) & \text{if } \mathit{red}(M, l) \equiv \mathit{red}(M, r) \\ \mathit{false} & \text{otherwise} \end{cases}
\end{aligned}$$

Note that, as for the case of `metaReduce`, the function *red* assumes that the equations in the module are Church-Rosser and terminating, and therefore that they can be applied in arbitrary order and to arbitrary redexes. Of course, lack of confluence would result in incompleteness (but not unsoundness) of the `nrwr` command; and, if the equations were not terminating then `nrwr` would not terminate in some cases either.

4 A rewriting command with integrated decision procedures for the ITP

We explain in this section the implementation in the ITP tool of the `xrwr` command that extends `nrwr` by integrating a decision procedure for Presburger arithmetic in the process of reducing a term. The technique used in the implementation of `xrwr` also applies to the implementation of similar commands that integrate other decision procedures.[6] The general technique is based on the fact that any decision procedure is a computable function, and, therefore, by the metatheorem of Bergstra and Tucker [2], it can be equationally specified by a finite set of Church-Rosser and terminating equations. To implement in the ITP tool a rewriting command that integrates a certain decision procedure in the rewriting process, all we have to do is to modify the function `nrwr` in such a way that the function implementing the given decision procedure is called at the appropriate times on the appropriate expressions. The function *redPlus* below is an example of this; but before introducing this function we briefly present the decision procedure that is thus integrated in the rewriting process.

In [24], Shostak describes a decision procedure for quantifier-free Presburger arithmetic. Presburger expressions are those that can be built up from integers, integer variables, and addition. (Arbitrary multiplication is not allowed, but it is convenient to use multiplication by constants as an abbreviation for repeated addition.) Linear inequalities are constructed by combining Presburger expressions with the usual arithmetic relations ($\leq, <, \geq, >, =$) and the propositional logic connectives. The procedure to check validity of a formula φ consists in expanding its negation into disjunctive normal form and expressing each disjunction as a conjunction of linear inequalities of the form $A \leq B$. Then, φ is valid if and only if its negation is not satisfiable, which is checked by looking for a solution in integers for each of the disjunctions with the help of an integer programming algorithm. This is an NP-complete problem and there is a certain tradeoff between using complete algorithms (in the sense of always terminating with a correct solution) or efficient but incomplete ones. Our design decision has been to use an efficient and terminating algorithm introduced in [24] that, however, yields no result in some uncommon situations.

The `xrwr` command is implemented in the ITP tool by an equationally defined function *redPlus* that, following the general technique described above, simply modifies the function *red* by introducing a new layer that corresponds to the decision procedure. It first checks whether the term is amenable to being dealt with (if it is a linear inequality in our case) and depending on the answer it calls a different auxiliary function.

$$\mathit{redPlus}(M,t) \triangleq \begin{cases} \mathit{redPlusAux1}(M,t) & \text{if } \mathit{isLinIneq?}(t) \\ \mathit{redPlusAux2}(M,t,\mathit{getEqs}(M)) & \text{otherwise} \end{cases}$$

The function *isLinIneq?* decides if a term corresponds to the representation of a linear inequality. To understand the definition of *redPlusAux1* below think of the term t as being, for example, the Boolean expression `N* <= M*`. We start by checking if the expression represented by t holds in M. For that, we use a function *getLinIneqs* that refines *getEqs* with the help of *isLinIneq?* and extracts from the metarepresentation of a module only those equations that involve linear inequalities. Then, if we let φ stand for the formula $\mathit{getLinIneqs}(M) \to t = \mathit{true}$, we make use of the function *isSatisfiable?*, that implements the decision procedure for quantifier-free Presburger arithmetic, to check the satisfiability of the negation of φ. If this negation is not satisfiable then φ is valid and t is provably equal to *true* in M, and thus can be reduced to *true*. If not, we try to see if it is provably equal to *false* by checking the satisfiability of the negation of $\mathit{getLinIneqs}(M) \to t = \mathit{false}$. Of

[6] In fact, the current implementation of the `xrwr` command in the ITP tool integrates, using the technique described here, a decision procedure introduced in [25] for an extension of quantifier-free Presburger arithmetic that permits arbitrary uninterpreted function symbols; this theory includes many of the formulas that one tends to encounter in program verification.

course, because of the incompleteness of the decision procedure used to check satisfiability, it could actually be the case that t is provably equal to *true* or *false*, but we cannot decide it. In this uncommon situation, we call the *redPlusAux2* function.

$$
\mathit{redPlusAux1}(M,t) \triangleq
$$
$$
\begin{cases}
\mathit{true} & \text{if not}(\mathit{isSatisfiable?}(\neg(\mathit{getLinIneqs}(M) \to t = \mathit{true}))) \\
\mathit{false} & \text{if not}(\mathit{isSatisfiable?}(\neg(\mathit{getLinIneqs}(M) \to t = \mathit{false}))) \\
\mathit{redPlusAux2}(M,t,\mathit{getEqs}(M)) & \text{otherwise}
\end{cases}
$$

The function *redPlusAux2* is called on terms that are not linear inequalities, or that are linear inequalities but the (incomplete) decision procedure has not succeeded in deciding them. Its definition is completely analogous to *redAux*, except for the fact that we use now *redPlus* and *solveCondPlus* instead of *red* and *solveCond*.

$$
\mathit{redPlusAux2}(M,t,\emptyset) \triangleq t
$$
$$
\mathit{redPlusAux2}(M,t,\{l=r\}\cup Eq) \triangleq \begin{cases}
\mathit{redPlus}(M,C[\sigma(r)]) & \text{if } t \equiv C[\sigma(l)] \\
\mathit{redPlusAux2}(M,t,Eq) & \text{otherwise}
\end{cases}
$$
$$
\mathit{redPlusAux2}(M,t,\{l=r \textbf{ if } \mathit{Cond}\}\cup Eq) \triangleq
$$
$$
\begin{cases}
\mathit{redPlus}(M,C[\sigma(r)]) & \text{if } t \equiv C[\sigma(l)] \text{ and } \mathit{solveCondPlus}(M,\sigma(\mathit{Cond})) \\
\mathit{redPlusAux2}(M,t,Eq) & \text{otherwise}
\end{cases}
$$

Finally, *solveCondPlus* deviates from *solveCond* in that, for equations whose terms are Presburger expressions, it also calls the decision procedure to check their validity. The function *isPresExp?* recognises those terms that represent Presburger expressions; we also omit here its definition.

$$
\mathit{solveCondPlus}(M,\emptyset) \triangleq \mathit{true}
$$
$$
\mathit{solveCondPlus}(M,\{l=r\}\cup Eq) \triangleq
$$
$$
\begin{cases}
\mathit{solveCondPlus}(M,Eq) & \text{if } \big(\mathit{isPresExp?}(l) \text{ and } \mathit{isPresExp?}(r) \text{ and} \\
& \quad \mathit{isSatisfiable?}(l=r)\big) \text{ or} \\
& \quad \mathit{redPlus}(M,l) \equiv \mathit{redPlus}(M,r) \\
\mathit{false} & \text{otherwise}
\end{cases}
$$

We end this section with a high-level description of how the rewriting command implemented by the function *redPlus* solves the difficulty we faced in proving (3) in Section 2. Let us denote by `INS-SORT+` the module `INS-SORT` extended with the equation `eq N* <= M* = false`. Recall that the problem was that the additional information provided by this equation could not be used to reduce the term `ins(N*,M*:nil)`. Now, when we apply *redPlus* to the metarepresentations of `INS-SORT+` and `ins(N*,M*:nil)`, since `ins(N*,M*:nil)` is not a linear inequality, the function *redPlusAux2* is called. Eventually, the equation `ins(N,M:L) = M:ins(N,L) if N > M = true` will be tried, and the function *solveCondPlus* will then be invoked to discharge the condition `N* > M* = true`. At this point, since `true` is not a Presburger expression, *redPlus* is invoked to reduce both `N* > M*` and `true`. Here is when the decision procedure comes into action: *redPlus* detects that `N* > M*` is a linear inequality and calls *redPlusAux1*. This function extracts the linear inequalities in `INS-SORT+` and uses the decision procedure to check if they imply `N* > M* = true`. Since this is actually the case, *redPlusAux1* reduces `N* > M*` to `true`. At this point, the condition `N* > M* = true` is discharged and `ins(N*,M*:nil)` is reduced to `M*:ins(N*:nil)`. The function *redPlus* will then eventually reduce `length(M*:ins(N*:nil))` to 2, as required, using the equations in `INS-SORT+` as simplification rules.

5 Conclusions and future work

In this work we have shown how the reflective design of a rewriting-based theorem prover like the ITP can be easily extended with decision procedures. The smoothness with which we have been able to integrate in the ITP tool an extension of the decision procedure described in Section 4 (which can be applied to the more general case of quantifier-free Presburger arithmetic with uninterpreted function symbols [25]) is encouraging. In this latest version of the ITP we have proved, as basic examples, the correctness of three different sorting functions, specifying respectively the insertion, merge, and quicksort algorithms. The correctness proofs for the merge and quicksort algorithms proceed by induction on the length of the list and make heavy use of the integrated decision procedure.

As a follow-up of this work we plan to add other decision procedures to our tool. As we have already pointed out, their integration will follow exactly the same steps as those described here and thus should pose no special difficulty. An important task ahead is to combine these decision procedures to be able to tackle expressions that involve diverse semantic constructs that belong, not just to one, but to several of them. In this regard, the recent generalisation of the Nelson-Oppen combination method to order-sorted theories [26] and the recent studies clarifying the relation between Nelson-Oppen's and Shostak's procedures [17] seem particularly relevant. Finally, we also plan to compare in more detail our reflective design for the integration of decision procedures in the ITP with the non-reflective one used in RRL. This comparison should be possible and interesting since both tools pursue similar goals and share a similar logical foundation.

Acknowledgements

We would like to thank José Meseguer for many interesting discussions on the subject of reflective rewriting-based theorem provers. We also thank José Meseguer, Peter Ölveczky, and the anonymous referees for their detailed and helpful comments on earlier versions of this paper.

References

1. F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, August 1999.
2. J. Bergstra and J. Tucker. Characterization of computable data types by means of a finite equational specification. In J. W. de Bakker and J. van Leeuwen, editors, *Automata Languages and Programming, Seventh Colloquium*, volume 81 of *Lecture Notes in Computer Science*, pages 76–90. Springer-Verlag, 1980.
3. R. S. Boyer and J. S. Moore. Integrating decision procedures into heuristic theorem provers: A case study for arithmetics. *Machine Intelligence*, 11:83–124, 1988.
4. M. Clavel. The ITP tool's home page. March 2004. The site contains the latest version of the ITP tool, the documentation currently available, and several examples of proof scripts of different complexity. `http://geminis.sip.ucm.es/~clavel/itp`.
5. M. Clavel. *Reflection in Rewriting Logic: Metalogical Foundations and Metaprogramming Applications*. CSLI Publications, 2000.
6. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 2002.
7. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. Maude Manual (Version 2.1). Manual distributed as documentation of the Maude system. `http://maude.cs.uiuc.edu`, 2004.

8. M. Clavel, J. Meseguer, and M. Palomino. Reflection in membership equational logic, many-sorted equational logic, Horn logic with equality, and rewriting logic. In F. Gadducci and U. Montanari, editors, *Proc. Fourth International Workshop on Rewriting Logic and its Applications*, volume 71 of *Electronic Notes in Theoretical Computer Science*, pages 63–78. Elsevier, 2002.
9. R. Diaconescu and K. Futatsugi. *CafeOBJ Report. The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification*, volume 6 of *AMAST Series in Computing*. World Scientific, 1998.
10. J.-C. Filliâtre, S. Owre, H. Rueß, and N. Shankar. ICS: integrated canonizer and solver. In G. Berry, H. Comon, and A. Finkel, editors, *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.
11. J. Goguen, C. Kirchner, H. Kirchner, A. Mégrelis, J. Meseguer, and T. Winkler. An introduction to OBJ3. In S. Kaplan and J.-P. Jouannaud, editors, *Conditional Term Rewriting Systems, 1st International Workshop Orsay, France, July 8–10, 1987, Proceedings*, volume 308 of *Lecture Notes in Computer Science*, pages 258–263. Springer-Verlag, 1988.
12. J. Goguen, A. Stevens, H. Hilberdink, and K. Hobley. 2OBJ: A metalogical framework theorem prover based on equational logic. *Philosophical Transactions of the Royal Society of London*, 339:69–86, 1992.
13. J. A. Goguen and G. Malcolm. *Algebraic Semantics of Imperative Programs*. The MIT Press, 1996.
14. D. Kapur and X. Nie. Reasoning about numbers in Tecton. In Z. W. Ras and M. Zemankova, editors, *Proc. of 8th International Symposium on Methodologies for Intelligent Systems (ISMIS'94)*, volume 869 of *Lecture Notes in Computer Science*, pages 57–70. Springer-Verlag, 1994.
15. D. Kapur and H. Zhang. An overview of rewrite rule laboratory (RRL). *Journal of Computer and Mathematics with Applications*, 29(2):91–114, 1995.
16. M. Kauffmann and J. S. Moore. An industrial strength theorem prover for a logic based on common lisp. *IEEE Transactions on Software Engineering*, 23(4):202–213, April 1997.
17. S. Krstić and S. Conchon. Canonization for disjoint union of theories. In F. Baader, editor, *Automated Deduction — CADE-19: 19th International Conference on Automated Deduction, Miami Beach, FL, USA, July 28 – August 2, 2003, Proceedings*, volume 2741 of *Lecture Notes in Computer Science*, pages 197–211. Springer-Verlag, 2003.
18. Z. Manna and the STeP group. STeP: Deductive-algorithmic verification of reactive and real-time systems. In R. Alur and T. A. Henzinger, editors, *Computer Aided Verification (CAV 96)*, volume 1102 of *lncs*, pages 415–418. Springer-Verlag, July/August 1996.
19. U. Martin and J. M. Wing, editors. *First International Workshop on Larch*. Springer-Verlag, 1992.
20. E. Mendelson. *Introduction to Mathematical Logic*. Van Nostrand, 2nd edition edition, 1979.
21. J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce, editor, *Recent Trends in Algebraic Development Techniques, 12th International Workshop, WADT'97, Tarquinia, Italy, June 3–7, 1997, Selected Papers*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer-Verlag, 1998.
22. S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752. Springer-Verlag, 1992.
23. M. Saaltink. The Z/EVES system. In *ZUM'97: The Formal Specification Notation, 10th International Conference on Z Users*, volume 1212 of *lncs*, pages 72–85. Springer-Verlag, April 1997.
24. R. E. Shostak. On the SUP-INF method for proving Presburger formulas. *Journal of the Association for Computing Machinery*, 24:529–543, 1977.
25. R. E. Shostak. A practical decision procedure for arithmetic with function symbols. *Journal of the Association for Computing Machinery*, 26:351–360, 1979.
26. C. Tinelli and C. Zarba. Combining decision procedures for theories in sorted logics. Technical Report 04-01, Department of Computer Science, The University of Iowa, Feb. 2004.

Some Undecidable Approximations of TRSs

Jeroen Ketema

Department of Computer Science
Faculty of Sciences, Vrije Universiteit Amsterdam
De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands

Abstract In this paper we study the decidability of reachability, normalisation, and neededness in n-shallow and n-growing TRSs. In an n-growing TRS, a variable that occurs on both the left-hand and right-hand side of a rewrite must be at depth n on the left-hand side an at a depth greater than n on the right-hand side. In an n-shallow TRS, a variable that occurs on both the left-hand and right-hand side of a rewrite rule must be at depth n on both sides.

The concepts n-growing and n-shallow are generalisations of the concepts growing and shallow, as introduced respectively by Jacquemard and Comon. For both shallow and growing TRSs reachability, normalisation, and neededness are decidable. However, as we show, these results do not generalise to n-growing and n-shallow TRSs. Consequently, a needed reduction strategy cannot be effectively employed in n-growing or n-shallow TRS.

1 Introduction

As is well-known, given an arbitrary term rewriting system (TRS), the following questions are undecidable.

- Reachability: is a term reachable from another term [4]?
- Normalisation: does a term have a normal form [1]?
- Neededness: does a term have a needed redex [4]?

However, for some classes of TRSs these properties are decidable. These classes are often used as approximations. That is, let $\mathcal{R}$ and $\mathcal{S}$ be TRSs over the same signature, then $\mathcal{S}$ is an *approximation* of $\mathcal{R}$ if $\to_{\mathcal{R}}^{*}\subseteq\to_{\mathcal{S}}^{*}$ and $\mathsf{NF}_{\mathcal{R}} = \mathsf{NF}_{\mathcal{S}}$. Here, $\to_{\mathcal{R}}^{*}$ and $\to_{\mathcal{S}}^{*}$ denote the rewrite relations of $\mathcal{R}$ and $\mathcal{S}$, and $\mathsf{NF}_{\mathcal{R}}$ and $\mathsf{NF}_{\mathcal{S}}$ denote the sets of normal forms of $\mathcal{R}$ and $\mathcal{S}$.

Two remarks are in order with respect to classes for which reachability, normalisation, and neededness are decidable. First, in most of these classes the shapes of the rewrite rules are restricted. See, for example, [4,2,5,9]. Second, given the decidability of neededness a needed reduction strategy can effectively be employed. See, for example, [4,3,2,5].

In this paper we explore the boundaries of the decidability of reachability, normalisation, and neededness. We do this by introducing n-growing and n-shallow TRSs. These TRSs are generalisations of the growing and shallow TRSs as introduced respectively by Jacquemard [5] and Comon [2]. Although reachability, normalisation, and neededness are decidable for growing and shallow TRSs we show that these properties are undecidable for our generalisations.

The n-growing and n-shallow TRSs are also closely related to four other classes of TRSs for which reachability, normalisation, and neededness are undecidable [5,6,7]. We show that n-growing and n-shallow TRSs are different from those classes of TRSs except in one instance.

We proceed as follows. In Sect. 2 we give some preliminary definitions. Then, in Sect. 3 we define two variants of Post's Correspondence Problem (PCP). We use these variants in Sect. 4 and Sect. 5 to show that reachability, normalisation, and neededness are undecidable for n-growing and n-shallow TRSs. In Sect. 6 we compare the n-growing and n-shallow TRSs to the other four classes of TRSs for which reachability, normalisation, and neededness are undecidable. In the final section, Sect. 7, we give some directions for further research.

2 Preliminaries

Throughout this paper we assume Γ is an arbitrary alphabet. By Γ^* and Γ^+ we denote the set of finite strings and the set finite non-empty strings over Γ. Moreover, by ϵ we denote the empty string, and if $s \in \Gamma^*$, then $|s|$ denotes the length of s.

If $s, t \in \Gamma^*$, then $s \cdot t$ denotes the concatenation of s and t. The empty string ϵ is the neutral element for concatenation. If $a \in \Gamma$ and $n \in \mathbb{N}$, then $a^0 = \epsilon$ and $a^{n+1} = a \cdot a^n$.

By $\mathcal{T}er(\Sigma, X)$ we denote the set of *terms* over the signature Σ and the set of variables X. The set Σ_n denotes the subset of Σ whose elements have arity n. If $t \in \mathcal{T}er(\Sigma, X)$, then $\mathcal{V}ar(t)$ denotes the set that contains the variables that occur in t We call t *linear* if each variable occurs at most once in t.

We confuse signatures consisting only of unary function symbols and alphabets. Hence, given a unary function symbol f and an $n \in \mathbb{N}$ we have $f^0(x) = x$ and $f^{n+1}(x) = f(f^n(x))$.

We denote the set of positions of a term $t \in \mathcal{T}er(\Sigma, X)$ by $\mathcal{P}os(t) \subseteq \mathbb{N}^*$. The *depth* of a subterm at $p \in \mathcal{P}os(t)$ is $|p|$. When t is linear, each $x \in \mathcal{V}ar(t)$ has a unique depth. In that case, $d_t(x)$ denotes the depth of x in t.

By $\mathcal{R} = (\Sigma, R)$ we denote a term rewriting system (TRS) with the signature Σ and the set of rewrite rules R. The elements of R are denoted $l \to r$. As usual in the study of approximations we only require $l \notin X$. We do not require not $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$. The transitive reflexive closure of $\to$ is denoted by $\to^*$. The rule $l \to r$ is called linear if l and r are linear.

Let $\mathcal{R} = (\Sigma, R)$ be a TRS and $s, t \in \mathcal{T}er(\Sigma, X)$. The term t is *reachable* from s if $s \to^* t$. Moreover, s *normalises* if $s \to^* t$ and if t is a normal form. A redex in s is *needed* if a descendant of the redex is contracted in every reduction from s to a normal form. A *needed reduction strategy* contracts in every term an arbitrary needed redex.

3 Variants of Post's Correspondence Problem

In this section we introduce two variants of Post's Correspondence Problem (PCP). In the definition of the variants we use the following notation.

Definition 1. Let $s \in \Gamma^+$. Given an $n \in \mathbb{N}$, define $s^{[n]}$ by

- $s^{[n]} = a^n$ if $s = a$ with $a \in \Gamma$, and
- $s^{[n]} = a^n \cdot t^{[n]}$ if $s = a \cdot t$ with $a \in \Gamma$ and $t \in \Gamma^+$.

Note that this is not exponentiation. That is defined as $s^0 = \epsilon$ and $s^{n+1} = s \cdot s^n$. Assuming $\Gamma = \{a, b\}$, we have $(ab)^{[2]} = aabb$ and $(ab)^2 = abab$.

We now define three kinds of pairs. We use them in the definition of PCP and the two PCP variants.

Definition 2. Let $u, v \in \Gamma^+$ and n a natural number. Then, (u, v) is called a *PCP pair*, $(u^{[n]}, v^{[n]})$ is called an *n-PCP pair*, and $(u^{[n]} \cdot e^{[kn]}, v^{[n]} \cdot e^{[ln]})$ is called a *padded n-PCP pair* if $k = \max\{|u|, |v|\} - |u|$, $l = \max\{|u|, |v|\} - |v|$, and $e \notin \Gamma$.

The intuition behind a padded n-PCP pair is that it is an n-PCP pair in which the shortest string is padded with e symbols. This gives both strings in the pair the same length. We have for any padded n-PCP pair $(u^n \cdot e^{kn}, v^n \cdot e^{ln})$ that

$$n(|u| + k) = |u^{[n]} \cdot e^{[kn]}| = |v^{[n]} \cdot e^{[ln]}| = n(|v| + l)$$

and that

$$((u^{[n]} \cdot e^{[kn]})[e := \epsilon], (v^{[n]} \cdot e^{[ln]})[e := \epsilon]) = (u^{[n]}, v^{[n]}) .$$

We now define PCP and the two PCP variants.

Question 1 (PCP). Let P be a finite set of PCP pairs. Does there exist an $m \geq 1$ such that $u_1 \cdot \ldots \cdot u_m = v_1 \cdot \ldots \cdot v_m$ with $(u_i, v_i) \in P$ for all $1 \leq i \leq m$?

Question 2 (n-PCP). Let P be a finite set of n-PCP pairs. Does there exist an $m \geq 1$ such that $u_1 \cdot \ldots \cdot u_m = v_1 \cdot \ldots \cdot v_m$ with $(u_i, v_i) \in P$ for all $1 \leq i \leq m$?

Question 3 (Padded n-PCP). Let $e \notin \Gamma$ and let P be a finite set of padded n-PCP pairs. Does there exist an $m \geq 1$ such that $(u_1 \cdot \ldots \cdot u_m)[e := \epsilon] = (v_1 \cdot \ldots \cdot v_m)[e := \epsilon]$ with $(u_i, v_i) \in P$ for all $1 \leq i \leq m$?

The three questions can be transformed into each other, as there exists for each kind of pair a related pair of each of the other kinds. For example, assuming again $\Gamma = \{a, b\}$, we have for the PCP pair (a, ab) that $(a^{[n]}, (ab)^{[n]})$ is a related n-PCP pair and that $((ae)^{[n]}, (ab)^{[n]})$ is related padded n-PCP pair. This leads to the following theorem.

Theorem 1. *PCP, n-PCP, and padded n-PCP are reducible to each other for $n \geq 1$.*

Proof. Using the previously described relations between the pairs, this follows directly from the definitions of PCP, n-PCP and padded n-PCP. □

By the previous theorem and the undecidability of PCP [8], we have the following.

Corollary 1. *The n-PCP and padded n-PCP questions are undecidable for $n \geq 1$.*

For padded n-PCP note that if $(u_1 \cdot \ldots \cdot u_m)[e := \epsilon] = (v_1 \cdot \ldots \cdot v_m)[e := \epsilon]$, then the number of e occurrences must be equal in $u_1 \cdot \ldots \cdot u_m$ and $v_1 \cdot \ldots \cdot v_m$. If not, the lengths of $(u_1 \cdot \ldots \cdot u_m)[e := \epsilon]$ and $(v_1 \cdot \ldots \cdot v_m)[e := \epsilon]$ are different which contradicts equality.

Using the previous fact and assuming a string rewrite system (SRS) with for all $a \in \Gamma$ and $e \notin \Gamma$ the rewrite rules $a \cdot e \to e \cdot a$ and $e \cdot a \to a \cdot e$, we can rephrase padded n-PCP.

Question 4 (Padded n-PCP). Let $\Delta = \Gamma \cup \{e\}$ and let P be a finite set of padded n-PCP pairs. Does there exist an $m \geq 1$ and an $s \in \Delta^+$ such that $u_1 \cdot \ldots \cdot u_m \to^* s$ and $v_1 \cdot \ldots \cdot v_m \to^* s$ with $(u_i, v_i) \in P$ for all $1 \leq i \leq m$?

4 Undecidable n-Growing Term Rewriting Systems

In this section we describe our first class of TRSs for which reachability, normalisation, and neededness are undecidable. The class is defined as follows.

Definition 3. Let $l \to r$ be a rewrite rule. The rule is *n-growing* if it is linear and if for all $x \in \mathcal{V}ar(l) \cap \mathcal{V}ar(r)$ it holds that $d_l(x) = n$ and $d_r(x) > n$. A TRS is *n-growing* if all its rewrite rules are n-growing.

Observe that in n-growing TRSs we restrict the shapes of the rewrite rules. Moreover, observe that n-growing rewrite rules and TRSs are closely related to the following rewrite rules and TRSs, as defined by Jacquemard [5].

Definition 4. Let $l \to r$ be a rewrite rule. The rule is *growing* if it is linear and if for all $x \in \mathcal{V}ar(l) \cap \mathcal{V}ar(r)$ it holds that $d_l(x) = 1$. A TRS is *growing* if all its rewrite rules are growing.

Obviously, for $n = 1$ the n-growing TRSs form a sub-class of the growing TRSs. For $n > 1$ the n-growing TRSs do not form a sub-class. We have, for example, the n-growing rewrite rule

$$f^n(x) \to f^{n+1}(x)\,.$$

For $n > 1$ this rewrite rule is not growing, as $d_{f^n(x)}(x) = n > 1$.

The growing TRSs do not form a sub-class of the n-growing TRSs for any n. We have, for example, the growing rewrite rule

$$f(x) \to x\,.$$

This rewrite rule is not n-growing, as $d_x(x) = 0$.

Using tree automata techniques, Jacquemard [2] proves that reachability and normalisation are decidable for growing TRSs. Durand and Middeldorp [3] prove that neededness is decidable for growing TRSs. As each 1-growing TRS is growing, we also have decidability of reachability, normalisation, and neededness for 1-growing TRSs. However, as we show next, these results do not generalise to n-growing TRSs with $n > 1$.

Theorem 2. *Let $n \geq 1$. Reachability is undecidable for $n+1$-growing TRSs.*

Proof. We reduce n-PCP to reachability in an $n+1$-growing TRS. Suppose we have a finite set P of n-PCP pairs. Define the signature $\Sigma = \Gamma \uplus \{c, d, f, g, h\}$, where $\uplus$ denotes the disjoint union. The arities are as follows

- c and d are constants,
- g, h and the elements of Γ have arity 1, and
- f has arity 2.

Also define for all $(u, v) \in P$ and $a \in \Gamma$ the following rewrite rules

$$\begin{aligned} c &\to f(g^n(u(d)), g^n(v(d))) && (1)\\ f(g^n(x), g^n(y)) &\to f(g^n(u(x)), g^n(v(y))) && (2)\\ f(g^n(x), g^n(y)) &\to h^{n+1}(f(x, y)) && (3)\\ f(a^n(x), a^n(y)) &\to h^{n+1}(f(x, y)) && (4)\\ f(d, d) &\to d && (5)\\ h^{n+1}(d) &\to d && (6) \end{aligned}$$

As is easy to see, we have a finite number of n-growing rewrite rules. Hence, the rewrite rules form an n-growing TRS. By $n \geq 1$, we have for the TRS that d is reachable from c if and only if n-PCP has a solution for P.

To see that d is reachable from c if n-PCP has a solution for P, suppose that $u_1 \cdot \ldots \cdot u_m = v_1 \cdot \ldots \cdot v_m$ is a solution. We can now construct the reduction sequence

$$\begin{aligned} c &\to_{(1)} f(g^n(u_m(d)), g^n(v_m(d))) \\ &\to^*_{(2)} f(g^n(u_1 \cdot \ldots \cdot u_m(d)), g^n(v_1 \cdot \ldots \cdot v_m(d))) \\ &\to_{(3)} h^{n+1}(f(u_1 \cdot \ldots \cdot u_m(d), v_1 \cdot \ldots \cdot v_m(d))) \end{aligned}$$

As $u_1 \cdot \ldots \cdot u_m = v_1 \cdot \ldots \cdot v_m$, we can, for some k, extend the reduction sequence to

$$\begin{aligned} c &\to^* \ h^{n+1}(f(u_1 \cdot \ldots \cdot u_m(d), v_1 \cdot \ldots \cdot v_m(d))) \\ &\to^*_{(4)} h^{k(n+1)}(f(d,d)) \\ &\to^*_{(5)} f(d,d) \\ &\to_{(6)} d \end{aligned}$$

Hence, d is reachable from c.

To see that n-PCP has a solution for P if d is reachable from c, note that the only way to reduce c to d is to first perform an (1)-step and a number of (2)-steps, then to perform a (3)-step and a number of (4)-steps, and to finally perform a (5)-step and a number of (6)-steps. Also note that the reduction sequence only ends in d if the (1)-step and the (2)-steps give us a term $f(g^n(u_1 \cdot \ldots \cdot u_m(d)), g^n(v_1 \cdot \ldots \cdot v_m(d)))$ with $u_1 \cdot \ldots \cdot u_m = v_1 \cdot \ldots \cdot v_m$. That is, as n-PCP has a solution for P.

Thus, n-PCP is reducible to a reachability problem in an $n+1$-growing TRS. As n-PCP is undecidable for $n \geq 1$, so is reachability for $n+1$-growing TRSs with $n \geq 1$. □

Observe that if we assume $n = 0$ in the previous proof, the rewrite rule

$$f(a^n(x), a^n(y)) \to h^{n+1}(f(x,y))$$

collapses to

$$f(x,y) \to h^{n+1}(f(x,y))\,.$$

As a consequence, d is no longer reachable form c. Something like this was to be expected, as reachability is decidable for 1-growing TRSs.

We now extend the above result to normalisation and neededness.

Theorem 3. *Let $n \geq 1$. Normalisation is undecidable for $n+1$-growing TRSs.*

Proof. We reduce n-PCP to normalisation in an $n+1$-growing TRS employing the proof showing that n-PCP is reducible to reachability.

Note that adding the $n+1$-growing rewrite rule

$$h^{n+1}(x) \to h^{n+1}(h^{n+1}(x)) \tag{7}$$

to the TRS from the proof of Theorem 2 does not change the fact that d is reachable from c if and only if n-PCP has a solution for P. However, by adding the rule, if d is reachable from c, the term d becomes the only normal form of c, else c does not have a normal form. By this fact and the fact that d is reachable from c if and only if n-PCP has a solution for P, we have that c has a normal form if and only if n-PCP has a solution for P.

Thus, n-PCP is reducible to normalisation in an $n+1$-growing TRS. As n-PCP is undecidable for $n \geq 1$, so is normalisation for $n+1$-growing TRSs with $n \geq 1$. □

Theorem 4. *Let $n \geq 1$. Neededness is undecidable for $n+1$-growing TRSs.*

Proof. We reduce n-PCP to neededness in an $n+1$-growing TRS employing the proof showing that n-PCP is reducible to normalisation.

By the proof of Theorem 3 only d can be a normal form of c. Hence, the only redex in c is needed if and only if d actually is a normal form of c. By this fact and the fact that c normalises if and only if n-PCP has a solution for P, we have that c has a needed redex if and only if n-PCP has a solution for P.

Thus, n-PCP is reducible to neededness in an $n+1$-growing TRS. As n-PCP is undecidable for $n \geq 1$, so is neededness for $n+1$-growing TRSs with $n \geq 1$. □

As a consequence of the previous theorem we have that a needed reduction strategy cannot be effectively employed in an $n+1$-growing TRS with $n \geq 1$.

5 Undecidable n-Shallow Term Rewriting Systems

In this section we describe our second class of TRSs for which reachability, normalisation, and neededness are undecidable. The class is defined as follows.

Definition 5. Let $l \to r$ be a rewrite rule. The rule is *n-shallow* if it is linear and if for all $x \in \mathcal{V}ar(l) \cap \mathcal{V}ar(r)$ it holds that $d_l(x) = d_r(x) = n$. A TRS is *n-shallow* if all its rewrite rules are n-shallow.

Observe that, like n-growing TRSs, n-shallow TRSs have restrictions on the shapes of their rewrite rules. The n-shallow TRSs form neither a sub-class nor a super-class of the n-growing TRSs. Consider, for example, the n-shallow rewrite rule

$$f^n(x) \to f^n(x)\,.$$

This rewrite rule is not n-growing, as $d_{f^n(x)}(x) = d_{f^n(x)}(x)$. We also have the n-growing rewrite rule

$$f^n(x) \to f^{n+1}(x)\,.$$

This rewrite rule is not n-shallow, as $d_{f^n(x)}(x) < d_{f^{n+1}(x)}(x)$.

The n-shallow rewrite rules and TRSs are closely related to the following rewrite rules and TRSs, as defined by Comon [2].

Definition 6. Let $l \to r$ be a rewrite rule. The rule is *shallow* if it is linear and if for all $x \in \mathcal{V}ar(l) \cap \mathcal{V}ar(r)$ it holds that $d_l(x) = 1$ and $d_r(x) \leq 1$. A TRS is *shallow* if all its rewrite rules are shallow.

Obviously, for $n = 1$ the n-shallow TRSs form a sub-class of the shallow TRSs. For $n > 1$ the n-shallow TRSs do not form a sub-class. For example, consider again the n-shallow rewrite rule

$$f^n(x) \to f^n(x)\,.$$

For $n > 1$ this rewrite rule is not shallow, as $d_{f^n(x)}(x) = d_{f^n(x)}(x) = n > 1$.

The shallow TRSs do not form a sub-class of the n-shallow TRSs for any n. This follows by the same example that shows that the growing TRSs do not form a sub-class of the n-growing TRSs for any n.

Using tree automata techniques, Comon [2] proves that reachability and normalisation are decidable for shallow TRSs. Durand and Middeldorp [3] prove that neededness is decidable for shallow TRSs. As each 1-shallow TRS is growing, we also have decidability of reachability, normalisation, and neededness for 1-shallow TRSs. However, as we show next, these results do not generalise to n-shallow TRSs with $n > 1$.

In the proofs below, we denote $[a, b]$ with $a, b \in \Delta = \Gamma \cup \{e\}$ and $e \notin \Gamma$ a unary function symbol. We also use the following definition.

Definition 7. Let $u, v \in \Delta^+$ with $|u| = |v|$. For $a, b \in \Delta$ and $u', v' \in \Delta^+$, define $[u, v](x)$ by

- $[u, v](x) = [a, b](x)$ if $u = a$ and $v = b$, and
- $[u, v](x) = [a, b]([u', v'](x))$ if $u = a \cdot u'$ and $v = b \cdot v'$.

Theorem 5. *Let $n \geq 1$. Reachability is undecidable for $n+1$-shallow TRSs.*

Proof. We reduce padded $n+1$-PCP to reachability in an $n+1$-shallow TRS. Suppose we have a finite set P of padded $n+1$-PCP pairs and an $e \notin \Gamma$. Let $\Delta = \Gamma \cup \{e\}$. Define the signature $\Sigma = \{[a, b] \mid a, b \in \Delta\} \uplus \{c, d\}$, with each $[a, b]$ a unary function symbol and c, d constants. Also define for all $(u, v) \in P$ and $a \in \Delta$ the following rewrite rules

$$c \to [u, v](c) \tag{1}$$

$$[a, a](c) \to d \tag{2}$$

$$[a, a](d) \to d \tag{3}$$

Moreover, define for all $u, u', v, v' \in \Delta^+$ with $u[e := \epsilon] = u'[e := \epsilon]$, $v[e := \epsilon] = v'[e := \epsilon]$, and $|u| = |u'| = |v| = |v'| = n + 1$ the following rewrite rule

$$[u, v](x) \to [u', v'](x) \tag{4}$$

This last rewrite rule can be considered as a transitive application to the strings u and v of the rewrite rules given just before Question 4.

As is easy to see, we have a finite number of n-shallow rewrite rules. Hence, the rewrite rules form an $n+1$-shallow TRS. By $n \geq 1$, we have for the TRS that d is reachable from c if and only if padded $n+1$-PCP has a solution for P.

To see that d is reachable from c if padded n-PCP has a solution for P, suppose that for some $s \in \Delta^+$ we have $u_1 \cdot \ldots \cdot u_m \to^* s$ and $v_1 \cdot \ldots \cdot v_m \to^* s$. That is, padded n-PCP has a solution for P. We can construct the following reduction sequence

$$\begin{aligned} c &\to_{(1)} [u_1, v_1](c) \\ &\to^*_{(1)} [u_1 \cdot \ldots \cdot u_m, v_1 \cdot \ldots \cdot v_m](c) \end{aligned}$$

As $u_1 \cdot \ldots \cdot u_m \to^* s$ and $v_1 \cdot \ldots \cdot v_m \to^* s$, we can, for some k, extend the reduction sequence to

$$\begin{aligned} c &\to^* \quad [u_1 \cdot \ldots \cdot u_m, v_1 \cdot \ldots \cdot v_m][c] \\ &\to^*_{(4)} [s, s](c) \\ &\to_{(2)} \cdots \\ &\to^*_{(3)} d \end{aligned}$$

Hence, d is reachable from c.

To see that n-PCP has a solution for P if d is reachable from c, note that the only way to reduce c to d is to first perform a number of (1)-steps, then to perform a (2)-step, and to finally perform a number of (3)-steps and to interleave all these steps with (4)-steps. Also note that the reduction sequence only ends in d if the (1)-steps together with a number of (4)-steps give us a term $[s,s](c)$ for some $u_1 \cdot \ldots \cdot u_m \to^* s$ and $v_1 \cdot \ldots \cdot v_m \to^* s$. That is, as padded n-PCP has a solution for P.

Thus, padded $n+1$-PCP is reducible to a reachability problem in an $n+1$-shallow TRS. As padded $n+1$-PCP is undecidable for $n \geq 1$, so is reachability for $n+1$-shallow TRSs with $n \geq 1$. $\square$

Observe that if we assume $n = 0$ in the previous proof, the last rewrite rule collapses to

$$[a,b](x) \to [a,b](x)$$

with $a, b \in \Delta$. As a consequence, d is no longer reachable from c. Something like this was to be expected, as reachability is decidable for 1-shallow TRSs.

Note that by the TRS specified in the previous proof an even stronger property holds.

Theorem 6. *Let $n \geq 1$. Reachability is undecidable for $n+1$-shallow TRSs in which every rewrite rule has at most one variable which occurs both at the left-hand and right-hand side of the rewrite rule.*

We now extend the above result to normalisation and neededness.

Theorem 7. *Let $n \geq 1$. Normalisation is undecidable for $n+1$-shallow TRSs.*

Proof. We reduce padded $n+1$-PCP to normalisation in an $n+1$-shallow TRS employing the proof showing that n-PCP is reducible to reachability.

Note that adding for all $u, v \in \Delta^+$ with $|u| = |v| \leq n+1$ the $n+1$-shallow rewrite rule

$$[u,v](d) \to [u,v](d) \tag{5}$$

to the TRS from the proof of Theorem 5 does not change the fact that d is reachable from c if and only if padded n-PCP has a solution for P. However, by adding the rule, if d is reachable for c, the term d becomes the only normal form of c, else c does not have a normal form. By this fact and the fact that d is reachable from c if and only if padded $n+1$-PCP has a solution for P, we have that c has a normal form if and only if padded $n+1$-PCP has a solution for P.

Thus, padded $n+1$-PCP is reducible to normalisation in an $n+1$-shallow TRS. As padded $n+1$-PCP is undecidable for $n \geq 1$, so is normalisation for $n+1$-shallow TRSs with $n \geq 1$. $\square$

Theorem 8. *Let $n \geq 1$. Neededness is undecidable for $n+1$-shallow TRSs.*

Proof. We reduce padded $n+1$-PCP to neededness in an $n+1$-shallow TRS. We employ the proof showing that padded $n+1$-PCP is reducible to normalisation.

By the proof of Theorem 7 only d can be a normal form of c. Hence, the only redex in c is needed if and only if d actually is a normal form of c. By this fact and the fact that c normalises if and only if padded $n+1$-PCP has a solution for P, we have that c has a needed redex if and only if padded $n+1$-PCP has a solution for P.

Thus, padded $n+1$-PCP is reducible to neededness in an $n+1$-shallow TRS. As padded $n+1$-PCP is undecidable for $n \geq 1$, so is neededness for $n+1$-shallow TRSs with $n \geq 1$. $\square$

As a consequence of the previous theorem we have that the needed reduction strategy cannot be effectively employed in an $n+1$-shallow TRS with $n \geq 1$.

6 Related Work

In this section, we compare n-growing and n-shallow TRSs to four other classes of TRSs for which reachability, normalisation, and neededness are undecidable [5,6,7]. We use the following five rewrite rules in the comparison.

$$f(x,x) \to g(x) \tag{1}$$
$$f(f(x)) \to x \tag{2}$$
$$f(g(x)) \to c \tag{3}$$
$$f^n(x) \to f^{n+1}(x) \tag{4}$$
$$f^n(x) \to f^n(x) \tag{5}$$

Note that (1) and (2) are neither n-growing nor n-shallow, that (3) is n-growing and n-shallow, and that (4) and (5) are respectively n-growing and n-shallow.

The first class of TRSs, defined by Jacquemard [5], requires for all $l \to r$ and $x \in \mathcal{V}ar(l) \cap \mathcal{V}ar(r)$ that $d_l(x) \leq 1$. This definition is equal to the definition of growing, except that rewrite rules no longer need to be linear. Hence, 1-growing and 1-shallow TRSs form sub-classes. For $n > 1$ the n-growing and n-shallow TRSs do not form sub-classes by (4) and (5). Moreover, for all n the n-growing and n-shallow TRSs do not form super-classes by (1), which is not n-growing or n-shallow as the left-hand side is not linear.

The second class of TRSs, again defined by Jacquemard [5], requires for all $l \to r$ and $x \in \mathcal{V}ar(l) \cap \mathcal{V}ar(r)$ that either $d_l(x) \leq 1$ or $d_r(x) \leq 1$ and that $l \to r$ is linear. Again, this definition is equal to the definition of growing, except that in this case $d_r(x) \leq 1$ is allowed. Consequently, 1-growing and 1-shallow TRSs again form sub-classes. For $n > 1$ the n-growing and n-shallow TRSs do not form sub-classes by (4) and (5). Moreover, for all n the n-growing and n-shallow TRSs do not form super-classes by (2), which is not n-growing or n-shallow as $d_x(x) = 0$.

The third class of TRSs, also defined by Jacquemard [6], requires for all $l \to r$ that l and r are of the shape $f(t_1, \ldots, t_k)$ with $f \in \Sigma \cup X$ and with t_i either a variable or a ground term for all $1 \leq i \leq k$. For all n the n-growing and n-shallow TRSs do not form sub-classes by (4). Moreover, for all n the n-growing and n-shallow TRSs do not form super-classes by (1), which is not n-growing or n-shallow as the left-hand side is not linear.

The fourth class, defined by Jacquemard, Meyer, and Weidenbach [7], requires the rewrite rules to be of the following shapes.

$$f(g(x)) \to h(k(x))$$
$$f(x) \to t$$
$$t \to f(x)$$

with $f, g, h, k \in \Sigma_1$ and t a ground term. For all n the classes of n-growing and n-shallow TRSs do not form sub-classes by (3). Moreover, for all $n \neq 2$ the n-growing and n-shallow TRSs do not form super-classes by $f(g(x)) \to h(k(x))$. For $n = 2$ the n-growing do not form super-classes again by $f(g(x)) \to h(k(x))$, but the n-shallow TRSs do obviously form a super-class. Hence, Jacquemard, Meyer, and Weidenbach already proved that reachability, normalisation, and neededness are undecidable for 2-shallow TRSs.

7 Further Directions

At least two questions remain. First of all, are reachability, normalisation, and neededness undecidable for TRSs in which for each rewrite rule $l \to r$ we have for all $x \in \mathcal{V}ar(l) \cap \mathcal{V}ar(r)$ that $d_l(x) = n$ and $d_r(x) < n$? Second, are reachability, normalisation, and neededness also undecidable for n-shallow and n-growing in case we require the TRSs to be orthogonal? It is highly likely that this is the case. However, our proofs are no longer applicable, as they make heavy use TRSs which are not orthogonal.

Acknowledgements. I would like to thank Jan Willem Klop, Aart Middeldorp, Femke van Raamsdonk, and Roel de Vrijer and the anonymous referees for their helpful comments and remarks.

References

1. F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.
2. H. Comon. Sequentiality, monadic second-order logic and tree automata. *Information and Computation*, 157(1–2):25–51, 2000.
3. I. Durand and A. Middeldorp. Decidable call by need computations in term rewriting. In W. McCune, editor, *Proc. of the 14th Int. Conf. on Automated Deduction (CADE-14)*, volume 1249 of *LNCS*, pages 4–18. Springer-Verlag, 1997.
4. G. Huet and J.-J. Lévy. Computations in orthogonal rewriting systems. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic*, pages 395–443. MIT Press, 1991.
5. F. Jacquemard. Decidable approximations of term rewriting systems. In H. Ganzinger, editor, *Proc. of the 7th Int. Conf. on Rewriting Techniques and Applications (RTA '96)*, volume 1103 of *LNCS*, pages 362–376. Springer-Verlag, 1996.
6. F. Jacquemard. Reachability and confluence are undecidable for flat term rewriting systems. *Information Processing Letters*, 87(5):265–270, 2003.
7. F. Jacquemard, C. Meyer, and C. Weidenbach. Unification in extensions of shallow equational theories. Technical Report MPI-I-98-2-002, Max-Planck-Institut für Informatik, 1998.
8. E. L. Post. A variant of a recursively unsolvable problem. *Bulletin of the American Mathematical Society*, 52:264–268, 1946.
9. H. Seki, T. Takai, Y. Fujinaka, and Y. Kaji. Layered transducing term rewriting system and its recognizability preserving property. In S. Tison, editor, *Proc. of the 13th Int. Conf. on Rewriting Techniques and Applications (RTA 2002)*, volume 2378 of *LNCS*, pages 98–113. Springer-Verlag, 2002.

Normalization by Evaluation

Olivier Danvy

University of Aarhus
Denmark
danvy@brics.dk

Invariant-Driven Strategies for Maude*

Francisco Durán, Manuel Roldán and Antonio Vallecillo

DLCC, Universidad de Málaga, Campus de Teatinos, Málaga, Spain
{duran,mrc,av}@lcc.uma.es

Abstract We propose generic invariant-driven strategies that control the execution of systems by guaranteeing that the given invariants are satisfied. Our strategies are generic in the sense that they are parameterized by the system whose execution they control, by the logic in which the invariants are expressed, and by the invariants themselves. We illustrate the use of the strategies in the case of invariants expressed in propositional logic. However, the good properties of Maude as a logical and semantic framework, in which many different logics and formalisms can be expressed and executed allow us to use other logics as parameter of our strategies.

1 Introduction

To deal with nonterminating and nonconfluent systems, we need good ways of controlling the rewriting inference process. In this line, different languages offer different mechanisms, including approaches based on metaprogramming, like Maude [2], or on strategy languages, like ELAN [1]. However, although the separation of logic and control greatly simplifies such a task, these mechanisms are sometimes hard to use, specially for beginners, and usually compromise fundamental properties like extensibility, reusability, and maintainability.

Formalisms like Z and UML suggest an interesting alternative, since they allow to define invariants or constraints as part of the system specifications. Although executing or simulating Z specifications may be hard, we can still find tools like Possum [8] or Jaza [11], which can do a reasonable simulation of such specifications. We find something somehow similar in UML, where, by specifying OCL constraints on our specifications, they can be made executable [12].

The execution or simulation of specifications with constraining invariants is typically based on integrating somehow the invariants into the system code. However, such an integration is clearly unsatisfactory: the invariants get lost amidst the code, and become difficult to locate, trace, and maintain. Moreover, the programs and the invariants to be satisfied on them are usually expressed in different formalisms, and *live at different levels of abstraction*: invariants are defined *on* programs. Therefore, it is interesting to have some way of expressing them separately, thus avoiding the mixing of invariants and code.

Maude does not provide direct support for expressing execution invariants. However, it does provide reflective capabilities and support to control the execution process, being also an excellent tool in which to create executable environments for various logics and models of computation [3]. Thus, it turns out to be a very good candidate for giving support to different types of invariants, which may be expressed in different formalisms.

In this paper we propose generic invariant-driven strategies to control the execution of systems by guaranteeing that the given invariants are always satisfied. Our strategies are generic in the sense that they are parameterized by the system whose execution they control, by the logic in which the invariants are expressed, and by the invariants themselves. The good properties of Maude as a logical and semantic framework [7], in which many different

* Partially supported by projects TIC 2001-2705-C03-02 and 2002-04309-C02-02.

logics and formalisms can be expressed and executed allows us to say that other logics and formalisms may be used as parameters of our strategies. We will use in this paper the case of propositional logic, although we have also experimented with future time linear temporal logic.

The paper is structured as follows. Section 2 serves as a brief introduction to rewriting logic and Maude. Section 3 introduces the definition of strategies in Maude, and serves as a basis for the introduction of invariant-guided strategies in Section 4. Section 5 describes as an example the case of invariants expressed using propositional calculus. Finally, Section 6 draws some conclusions.

2 Rewriting Logic and Maude

Maude [2] is a high-level language and a high-performance interpreter and compiler in the OBJ [4] algebraic specification family that supports membership equational logic [10] and rewriting logic [9] specification and programming of systems.

Membership equational logic is a Horn logic whose atomic sentences are equalities $t = t'$ and *membership assertions* of the form $t : S$, stating that a term t has sort S. Such a logic extends order-sorted equational logic, and supports sorts, subsort relations, subsort polymorphic overloading of operators, and the definition of partial functions with equationally defined domains.

Rewriting logic is a logic of change that can naturally deal with state and with highly nondeterministic concurrent computations. In rewriting logic, the state space of a distributed system is specified as an algebraic data type in terms of an equational specification (Σ, E), where Σ is a signature of sorts (types) and operations, and E is a set of (conditional) equational axioms. The dynamics of a system in rewriting logic is then specified by rewrite *rules* of the form $t \to t'$, where t and t' are Σ-terms. These rules describe the local, concurrent transitions possible in the system, i.e. when a part of the system state fits the pattern t then it can change to a new local state fitting pattern t'. Rules may be conditional, in which case the guards act as blocking pre-conditions, in the sense that a conditional rule can only be fired if the condition is satisfied.

In Maude, object-oriented systems are specified by object-oriented modules in which classes and subclasses are declared. A class is declared with the syntax

$$\texttt{class}\ C\ \texttt{|}\ a_1 : S_1\texttt{,}\ \ldots\texttt{,}\ a_n : S_n,$$

where C is the name of the class, a_i are attribute identifiers, and S_i are the sorts of the corresponding attributes. Objects of a class C are then record-like structures of the form

$$\texttt{<}\ O\ \texttt{:}\ C\ \texttt{|}\ a_1 : v_1\texttt{,}\ \ldots\texttt{,}\ a_n : v_n\ \texttt{>},$$

where O is the name of the object, and v_i are the current values of its attributes. Objects can interact in a number of different ways, including message passing. Messages are declared in Maude in `msg` clauses, in which the syntax and arguments of the messages are defined.

In an object-oriented system, a state, which is called a *configuration*, has the structure of a multiset made up of objects and messages that evolves by rewriting using rules that describe the effects of the communication events of objects and messages. The general form of such rewrite rules is

$$
\begin{array}{l}
\texttt{crl [}r\texttt{] :} \\
\quad \texttt{<}\ O_1\ \texttt{:}\ C_1\ \texttt{|}\ \mathit{atts}_1\ \texttt{>}\ \ldots\ \texttt{<}\ O_n\ \texttt{:}\ C_n\ \texttt{|}\ \mathit{atts}_n\ \texttt{>} \\
\quad M_1\ \ldots\ M_m \\
\quad \texttt{=>}\ \texttt{<}\ O_{i_1}\ \texttt{:}\ C'_{i_1}\ \texttt{|}\ \mathit{atts}'_{i_1}\ \texttt{>}\ \ldots\ \texttt{<}\ O_{i_k}\ \texttt{:}\ C'_{i_k}\ \texttt{|}\ \mathit{atts}'_{i_k}\ \texttt{>} \\
\qquad \texttt{<}\ Q_1\ \texttt{:}\ C''_1\ \texttt{|}\ \mathit{atts}''_1\ \texttt{>}\ \ldots\ \texttt{<}\ Q_p\ \texttt{:}\ C''_p\ \texttt{|}\ \mathit{atts}''_p\ \texttt{>} \\
\qquad M'_1\ \ldots\ M'_q \\
\quad \texttt{if}\ \mathit{Cond}\ \texttt{.}
\end{array}
$$

where r is the rule label, $M_1...M_m$ and $M'_1...M'_q$ are messages, $O_1...O_n$ and $Q_1...Q_p$ are object identifiers, $C_1...C_n$, $C'_{i_1}...C'_{i_k}$ and $C''_1...C''_p$ are classes, $i_1...i_k$ is a subset of $1...n$, and $Cond$ is a Boolean condition (the rule's *guard*). The result of applying such a rule is that: (a) messages $M_1...M_m$ disappear, i.e., they are consumed; (b) the state, and possibly the classes of objects $O_{i_1}...O_{i_k}$ may change; (c) all the other objects O_j vanish; (d) new objects $Q_1...Q_p$ are created; and (e) new messages $M'_1...M'_q$ are created, i.e., they are sent. Rule labels and guards are optional.

For instance, the Maude module `DINING-PHILOSOPHERS` below specifies the well known problem of the hungry philosophers. The problem assumes five philosophers sitting around a table, on which five plates and five chopsticks are laid out. A philosopher can do two things, either think, or eat. When he thinks, a philosopher does not need the chopsticks; on the other hand, when thinking, he ends up being hungry. To eat, he needs the two chopsticks which are disposed on each side of his plate. Once he has finished eating, the philosopher releases the chopsticks and starts thinking, and then will be hungry again, etc.

Philosophers are modeled using a class with two attributes. The attribute `state` represents the state of the philosopher—which can be `thinking`, `hungry`, or `eating`—and the attribute `sticks` represents the number of chopsticks he holds. Moreover, a message `chopstick(`N`)` has been defined, indicating that the chopstick N is free. Philosophers and chopsticks are named with numbers from one to five, in such a way that the chopsticks besides the philosopher i are i and $i+1$, or i and 1 if i is 5. Note the subsort declaration `Nat < Oid` making a natural number a valid object identifier.

The system behavior is defined by four rules, each one representing a local transition of the system. For example, the rule labeled as `grab` may be fired when a philosopher object I is hungry and it receives a message indicating that the chopstick J is free, being the chopstick J one of the chopsticks I can grab. As a result, the message is consumed, and the number of chopsticks grabbed by the philosopher is increased. The syntax for rules and conditional rules is, respectively, `rl [`l`] :` t `=>` t' and `rl [`l`] :` t `=>` t' `if` c, with l a rule label, t and t' terms, and c a rule condition.

```
(omod DINING-PHILOSOPHERS is
   protecting NAT .
   subsort Nat < Oid .    *** Object identifiers are numbers!

   sort Status .
   ops thinking hungry eating : -> Status [ctor] .

   class Philosopher | state : Status, sticks : Nat .
   msg chopstick : Nat -> Msg .

   vars I J K : Nat .

   op _can‘use_ : Nat Nat -> Bool .
   eq I can use J = (I == J) or (s(I) == J) or (I == 5 and J == 1) .
```

```
  rl [hungry] :
    < I : Philosopher | state : thinking >
    => < I : Philosopher | state : hungry > .
  crl [grab] :
    chopstick(J)
    < I : Philosopher | state : hungry, sticks : K >
    => < I : Philosopher | sticks : K + 1 >
    if I can use J .
  rl [eat] :
    < I : Philosopher | state : hungry, sticks : 2 >
    => < I : Philosopher | state : eating > .
  rl [full] :
    < I : Philosopher | state : eating >
    => < I : Philosopher | state : thinking, sticks : 0 >
       chopstick(I)
       chopstick(s(I)) .
endom)
```

In Maude, those attributes of an object that are not relevant for an axiom do not need to be mentioned. Attributes not appearing in the right-hand side of a rule will maintain their previous values unmodified.

3 Execution strategies in Maude

System modules and object-oriented modules in Maude do not need to be Church-Rosser and terminating, therefore the system state may evolve in different directions depending on the order in which we apply the rules describing such a system. Maude provides two built-in strategies: The *rewrite* command follows a top-down lazy rule-fair strategy, and the *frewrite* command follows a position-fair bottom-up strategy. Although enough in many cases, the rewriting inference process could not terminate or go in many undesired directions. Thanks to the reflective capabilities that Maude provides, we can define our own strategies, which in fact are defined using statements in a normal module.

Maude provides key metalevel functionality for metaprogramming and for writing execution strategies. In general, strategies are defined in extensions of the predefined module `META-LEVEL` by using predefined functions in it, like `metaReduce`, `metaApply`, `metaXapply`, etc. as building blocks. `META-LEVEL` also provides sorts `Term` and `Module`, so that the representations of a term T and of a module M are, respectively, a term $\overline{T}$ of sort `Term` and a term $\overline{M}$ of sort `Module`. Constants (resp. variables) are metarepresented as quoted identifiers that contain the name of the constant (resp. variable) and its type separated by a dot (resp. colon), e.g., `'true.Bool` (resp. `'B:Bool`). Then, a term is constructed in the usual way, by applying an operator symbol to a comma-separated list of terms. For example, the term `S |= True` of sort `Bool` in the module `PL-SATISFACTION` below is metarepresented as the term `'_|=_['S:State, 'True.Formula]` of sort `Term`.

Of particular interest for our current purposes are the partial functions `metaReduce` and `metaXapply`.[1]

```
op metaReduce : Module Term ~> Term .
op metaXapply : Module Term Qid ~> Term .
```

[1] We have simplified the form of these functions for presentation purposes, since we do not need here their complete functionality. See [2] for the actual descriptions.

metaReduce takes a module $\overline{M}$ and a term $\overline{T}$, and returns the metarepresentation of the normal form of T in M, that is, the result of reducing T as much as possible using the equations in M. metaXapply takes as arguments a module $\overline{M}$, a term $\overline{T}$, and a rule label L, and returns the metarepresentation of the term resulting from applying the rule with label L in M on the term T.

To illustrate the general approach, and as a first step towards our final goal, let us suppose that we are interested in a strategy that rewrites a given term by applying on it all the rules in a given module, in any order. The strategy should just try to apply the rules one by one on the current term until it gets rewritten. Once a rule can be applied on it, the term resulting from such an application becomes the current term, and we start again. If none of the rules can be applied on a term, then it is returned as the result of the rewriting process. Such a strategy can be specified as follows:

```
op rew : Module Term -> Term .
op rewAux : Module Term ContStruct -> Term .

eq rew(M, T) = rewAux(M, T, cont(M)) .
eq rewAux(M, T, C) = T if final(C) .
ceq rewAux(M, T, C)
  = if T' :: Term
    then rewAux(M, T', reset(C))
    else rewAux(M, T, C')
    fi
  if C' := next(C) /\ T' := metaXapply(M, T, getLabel(C')) .
```

The operation rew takes two arguments: the (metarepresentation of) the module describing the system whose execution we wish to control, and the term representing the initial state of the system. rewAux takes three arguments: the module describing the system, the term being rewritten, and a continuation structure with the labels of the rules in the module, which allows us to iterate on the labels in some order. The strategy gives as result a term which cannot be further rewritten.

In the equations defining rew we assume a function cont that takes a module and returns a continuation structure for it, a structure which contains the module's rule labels and keeps control on the last label requested. We also assume the following functions on the sort ContStruct of continuation structures: final, which returns a Boolean value indicating whether there are more labels in the structure; reset, which initializes the structure, that is, it returns the structure with the next label set to be the first one; next, which returns the structure with the next label set to be the next one; and getLabel, which returns the next label in the sequence. Note that we do not assume a concrete structure; depending on the particular structure used, and on the definition of these operations, the order in which the labels are considered may be different, which provides extra adaptability for our strategy.

Note the use of the metaXapply function. A rewriting step $T \xrightarrow{L} T'$ is accomplished only if the rule labeled L is applicable on the term T, being T' the term returned by metaXapply(M, T, L). The membership assertion "T' :: Term" is used to check whether the result of the application of the rule is of sort Term or not. Note that in case the rule cannot be applied, metaXapply returns an error term in a supersort of Term.

4 Using invariants to guide the system execution

Basically, an invariant is a property that a specification or program always requires to be true. Instead of using an external monitor to verify a given system specification against an invariant, we propose using invariants as part of our specifications, making it *internal*. We suggest exploiting the possibility of defining execution strategies to *drive* the system execution in such a way that we can guarantee that every obtained state complies with the invariant, thus avoiding the execution of actions conducting the system to states not satisfying the invariant. If we want to define a strategy which guarantees the invariant, we may use a variant of the strategy in Section 3: We just need to check that the invariant is satisfied by the initial state and by every candidate to new state in a rewriting step.

To implement this new strategy we assume a satisfaction Boolean predicate `_|=_` such that, given a state of the system S and an invariant I, then S `|=` I evaluates to `true` or `false` depending on whether the state S satisfies the invariant I or not. The new strategy requires two additional parameters: (the metarepresentation of) the invariant predicate, and (the metarepresentation of) the module defining the satisfaction relation in the logic used for expressing such an invariant. The new strategy can be written as follows:

```
op rewInv : Module Module Term Term ~> Term .
op rewInvAux : Module Module Term Term ContStruct -> Term .

ceq rewInv(M, M', T, I)
  = rewInvAux(M, M', T, I, cont(M))
  if metaReduce(M', '_|=_[T, I]) = 'true.Bool .
ceq rewInvAux(M, M', T, I, C) = T if final(C) .
ceq rewInvAux(M, M', T, I, C)
  = if T' :: Term
       and-then metaReduce(M', '_|=_[T', I]) == 'true.Bool
    then rewInvAux(M, M', T', I, reset(C))
    else rewInvAux(M, M', T, I, next(C))
    fi
  if L := getLabel(next(C)) /\ T' := metaXapply(M, T, L) .
```

Now the auxiliary function is invoked if the initial state satisfies the invariant. Notice that the operator `rewInv` is declared using `~>`, meaning that if not reduced, it will return an error term of sort `[Term]`, which represents the kind of the sort `Term` and all sorts related to it. A kind is semantically interpreted as the set containing all the well-formed expressions in the sorts determining it, and also error expressions. Moreover, the strategy takes a rewriting step only if the term can be rewritten using a particular rule and it yields to a next state which satisfies the invariant. An invariant I is checked by evaluating the expression T' `|=` I, for a given candidate transition $T \xrightarrow{L} T'$. Note however that the rewriting process takes place at the metalevel, and we use `metaReduce` for evaluating the satisfaction of the property.

Notice also that the rules describing the system can be written independently from the invariants applied to them, and the module specifying the system is independent of the logic in which the invariants are expressed, thus providing the right kind of independence and modularity between the system actions and the system invariants. In fact, the strategy is parameterized by the system to be executed (M), the invariant to be preserved (I) and the module defining the satisfaction relation (M'). This allows using different logics to express the invariant without affecting the strategy or the system to execute.

5 Defining logics for driving the system execution: Propositional calculus

The logic in which the invariants are expressed is independent of the system to be executed. This would allow us to use one logic or another to express our invariants depending on our needs. We illustrate our approach with propositional logic.

If we want to use a specific logic to express the invariant predicates we need to define the syntax of such a logic and a satisfaction relation for it. Given a set of atomic propositions, which corresponds to the sort `Proposition`, we define the formulae of the propositional calculus in the following module `PROPOSITIONAL-CALCULUS`.

```
(fmod PROPOSITIONAL-CALCULUS is
   sort Proposition Formula .
   subsort Proposition < Formula .

   ops True False : -> Formula .
   op _and_ : Formula Formula -> Formula [assoc comm prec 55] .
   op _or_ : Formula Formula -> Formula [assoc comm prec 59] .
   op _xor_ : Formula Formula -> Formula [assoc comm prec 57] .
   op not_ : Formula -> Formula [prec 53] .
   op _implies_ : Formula Formula -> Formula [prec 61] .
   op _iff_ : Formula Formula -> Formula [assoc prec 63] .

   vars A B C : Formula .

   eq True and A = A .
   eq False and A = False .
   eq A and A = A .
   eq False xor A = A .
   eq A xor A = False .
   eq A and (B xor C) = A and B xor A and C .
   eq not A = A xor True .
   eq A or B = A and B xor A xor B .
   eq A implies B = not(A xor A and B) .
   eq A iff B = A xor B xor True .
 endfm)
```

The module `PROPOSITIONAL-CALCULUS` introduces the sort `Formula` of well-formed propositional formulae, with two designated formulae, namely `True` and `False`, with the obvious meaning. The sort `Proposition`, corresponding to the set of atomic propositions, is declared as subsort of `Formula`. `Proposition` is by the moment left unspecified; we shall see below how such atomic propositions are defined for a given system module. Then, the usual operators are declared. These declarations follow quite closely the definition of Boolean values in Maude and OBJ3 [4], which are based on the decision procedure proposed by Hsiang [6]. This procedure reduces valid propositional formulae to the constant `True`, and all the others to some canonical form which consists of an exclusive or of conjunctions.

The following module `PL-SATISFACTION` defines a satisfaction relation for propositional formulae.

```
(fmod PL-SATISFACTION is
   protecting PROPOSITIONAL-CALCULUS
   sorts State Formula .
   op _|=_ : State Formula -> Bool .
```

```
  var  S : State .
  vars F F' : Formula .

  eq S |= (F and F') = (S |= F) and (S |= F') .
  eq S |= (F xor F') = (S |= F) xor (S |= F') .
  eq S |= not F = not S |= F .
  eq S |= True = true .
  eq S |= False = false .
endfm)
```

As said above, the satisfaction relation `_|=_` is a Boolean predicate such that, given a state (the sort `State` will be defined for each particular problem) and a formula, evaluates to `true` or `false` depending on whether the given state satisfies such a formula or not. Notice that `_|=_` takes a propositional formula as second argument and returns a Boolean value, being `Bool` a predefined sort in Maude.

If we want to use propositional calculus to define invariant predicates for a given problem, we need to define the atomic propositions of interest for such a problem. For example, we could define an invariant predicate for guiding the execution of our philosophers example in such a way that we avoid deadlock situations.

The system would go into deadlock if we reach a state where each philosopher has one chopstick. We define what a `State` is—in this example, a `Configuration`—and the proposition `fork(`P`,`N`)`, which holds if the philosopher P has N chopsticks, in the following module:

```
(omod DINING-PHILOSOPHERS-PL-PREDS is
  protecting DINING-PHILOSOPHERS .
  including PL-SATISFACTION .
  subsort Configuration < State .
  op forks : Oid Nat -> Proposition .

  vars I N M : Nat .
  var  C : Configuration .

  eq < I : Philosopher | sticks : N > C |= forks(I, M)
    = N == M .
endom)
```

Once we have defined the atomic proposition `forks`, it may be used to define the intended invariant for guiding the execution. Thus, the invariant to avoid deadlock states may be expressed as follows:

```
not(forks(1, 1) and forks(2, 1) and forks(3, 1) and forks(4, 1) and forks(5, 1))
```

Let us denote $\bar{t}$ and $\overline{M}$ the metarepresentations of a term t and a module M . We can rewrite an initial state for the `DINING-PHILOSOPHERS` system, given by a constant `initial-state`, with the strategy `rewInv` with the previous invariant as follows.

```
red rewInv(DINING-PHILOSOPHERS,
           DINING-PHILOSOPHERS-PL-PREDS,
           initial-state,
           not(forks(1, 1) and forks(2, 1) and
               forks(3, 1) and forks(4, 1) and forks(5, 1))) .
```

With this command, we execute the system by allowing the nondeterministic application of the rules in the module, but with the guarantee that the invariant is satisfied by all the states in the trace.

6 Concluding Remarks

We have proposed generic invariant-driven strategies, which control the execution of systems by guaranteeing that the given invariants are satisfied. Our strategies are generic in the sense that they are parameterized by the system whose execution they control, by the logic in which the invariants are expressed, and by the invariants themselves. This parameterization, together with the level of modularization of the approach, allows improving quality factors such as extensibility, understandability, usability, or maintainability. We have illustrated its use with invariants expressed in propositional calculus. However, the good properties of Maude as a logical and semantic framework [7], in which many different logics and formalisms can be expressed and executed, allow us to use other logics as parameters of our strategies.

The strategy `rewInv` given in Section 4, although valid for logics like propositional logic, has to be slightly modified in the case of logics like temporal logics. We have already experimented with future time linear temporal logic (LTL for short). In this case, the satisfaction of LTL formulae cannot be decided considering particular states, but we need to look at complete traces. For example, consider the invariant restriction `[]`P (P always holds). This invariant requires any future state to maintain the property P, and obviously this cannot be guaranteed just considering the actual state. Our approach to deal with temporal logic is based on the one proposed by Havelund and Roşu in [5] for monitoring Java programs, based on the progressive transformation of the invariant restrictions when the system state evolves, possibly obtaining a new invariant when the system state changes.

References

1. P. Borovanský, H. Cirstea, H. Dubois, C. Kirchner, H. Kirchner, P.-E. Moreau, C. Ringeissen, and M. Vittek. ELAN v 3.3 user manual. Technical report, INRIA Lorraine & LORIA, 1998.
2. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. Maude manual. Available in `http://maude.cs.uiuc.edu`., 2004.
3. M. Clavel, F. Durán, S. Eker, J. Meseguer, and M.-O. Stehr. Maude as a formal meta-tool. In J. Wing, J. Woodcock, and J. Davies, eds., *FM'99 (Vol. II)*, LNCS 1709:1684–1704. Springer, 1999.
4. J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. In J. Goguen and G. Malcolm, editors, *Software Engineering with OBJ: Algebraic Specification in Action*. Kluwer, 2000.
5. K. Havelund and G. Roşu. Rewriting-based techniques for runtime verification. To appear in *Journal of Automated Software Engineering*.
6. J. Hsiang. Refutational theorem proving using term rewriting systems. *Artificial Intelligence*, (25):255-300, 1985.
7. N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. vol. 9, pp. 1–87. Kluwer, 2002.
8. T. McComb and G. Smith. Animation of Object-Z specifications using a Z animator. In *Procs. SEFM'03*, 2003.
9. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.
10. J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce (ed.), *Recent Trends in Algebraic Development Techniques*, LNCS 1376:18–61. Springer, 1998.
11. M. Utting. The Jaza animator. `http://www.cs.waikato.ac.nz/~marku/jaza/`.
12. J. Warmer and A. Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley, 2003.

Data Structures for Fast Normalization and Strategies*

Rakesh Verma and James Thigpen

Computer Science Department, University of Houston, TX 77204-3010 USA. rmverma@cs.uh.edu

Abstract Much of the time in normalization is spent in traversing the term with the goal of finding a match with a rule. In this paper, we present the DS-list and DS-forest - new data structures for speeding up normalization. The DS-forest can also be used to allow the efficient and smooth integration of many popular strategies for rewriting. We present results showing that the DS-list and a preliminary implementation of DS-forest outperform the best strategy of Laboratory for Rapid Rewriting (LRR) presented in 1999. We also present results for two other strategies that we recently implemented in LRR.

1 Introduction

Fast rewriting is needed for equational programming, rewrite based formal verification methods, and symbolic computing systems. In any implementation of rewriting techniques efficiency is a critical issue [5]. Much of the time in normalizing a term is spent in traversing the term to find the next match. In this paper, we present the DS-list and DS-forest, new data structures for speeding up normalization and efficiently, naturally implementing many different strategies for rewriting. We first present the DS-list structure and then a more sophisticated DS-forest structure. Results on eight benchmarks show that these structures outperform the best strategy of LRR [9] presented in 1999. We also present results for two other strategies that we recently implemented in LRR. DS-forest is the overall winner.

2 The Data Structures

Previously [8], we presented an optimization designed to reduce the needless traversal of the expression called "dont-reduce signatures", which has been implemented in LRR 1.2. The new data structures proposed are expected to yield even greater benefits. We present the DS-list and DS-forest as separate structures here for convenience, although the structures can be built on top of the expression tree or graph itself analogous to a threaded list/tree structure. The advantage of building them on the expression is that there is no need to maintain correspondence between the nodes in the DS-list/DS-forest and the expression.

The DS-list and DS-forest cut down on the needless traversal of the expression graph by keeping track of the subexpressions that have a defined symbol at the root. The DS-forest also allows for *efficient* implementation of the leftmost/rightmost innermost/outermost strategies. A *defined symbol* is a symbol that appears as the top symbol of some left-hand side in the rewrite system. The rest of the symbols are called *constructors*. Note that "built-in" symbols such as arithmetic and relational operators, etc., are not included in the list of defined symbols. This happens naturally since no rules can be given for built-in's.

DS-list. The DS-list is a circular doubly-linked list that contains pointers to subterms of the current term being reduced that have defined symbols at the top. As the given term is parsed, the DS-list is initialized to contain pointers to subterms with defined symbols at the root occurring in the given term.

* Research supported in part by NSF grant CCF 0306475

A pointer to the *current* node in the list is always maintained. During reduction, this pointer is advanced from the current node to the next, attempting to match the subexpression pointed to by the node. Upon finding a match, the list is updated to contain nodes pointing to any new resulting subexpressions with defined symbols at the top. After this the reduction procedure continues traveling around the DS-list, checking for matches. Reduction is complete when the DS-list size is 0 or when a complete traversal around the list is made without any matches.

DS-forest. The DS-forest is a collection of DS-trees with the roots of the trees linked in a doubly linked list. Each DS-tree is a rooted tree in which each node has 2 data fields, viz., expr and dist, and 5 pointer fields for pointers to its: parent, leftmost child, rightmost child, left sibling and right sibling. The expr field stores a pointer to a subexpression with a defined symbol at its root. Initially, the parent of node n in the DS-tree is the closest ancestor of n labeled by a defined symbol in the given expression. After some successful matches the parent of node n in the DS-tree is the closest ancestor of n labeled by a defined symbol in the resulting expression. If the parent pointer is not null, then the node also contains, in the dist data field, the distance to the parent in the current expression to be normalized. The children of each node are linked into a doubly linked list for easy insertions and deletions. The DS-forest is initialized when the expression to be normalized is parsed.

Updating the DS-forest. For updates to the data structure on successful matches a template of a DS-forest is constructed for each rhs, which tracks the defined symbols and their distance in the rhs. The root nodes of the template are linked into a doubly linked list. This template has special nodes called “holes” corresponding to the variables in the rhs. In the template the dist field is initialized for each defined symbol relative to the closest ancestor (if any) in the rhs. If there is no such ancestor it is set to 0. On a successful match the forest must be updated to remove the nodes corresponding to defined symbols in the template (non-variable) part of the left-hand side and a copy of the rhs template must be inserted into the DS-forest with appropriate field values. This means that the dist field of root nodes and the holes in the template must be updated. The parent pointer of the roots node in the template being inserted is set to the parent pointer of the node in the DS-forest corresponding to the subexpression that matched the lhs. Special care must be taken to handle the correspondence between the DS-forest and the expression to be normalised.

Variables in the rules give rise to two issues. First, if a variable has more occurrences in the rhs than in the lhs, then duplication of the substitution part of DS-forest must be done in order to maintain the tree structure. Hence, a DS collection based on DAG’s as the basic unit may be more efficient in this case. If DAG’s are chosen, then the parent and dist fields are not unique - details of how to resolve these issues are left for a full version of this paper. For rewrite engines that share common subexpressions completely (such as Smaran), no duplication is necessary. The second issue is that the top-most defined symbols in the substitution part of each variable in the lhs are needed to update the DS-tree correctly (when the variable matches a subexpression whose top part is labeled with constructors). This may require a potentially time-consuming traversal of the substitution part of a variable and could reduce the advantage of the DS-tree structure somewhat. However, the dont-reduce optimization we had introduced earlier in [8] can cut down this traversal time partially because it allows us to ignore the maximal subexpressions that contain only constructors in the substitution.

Matching with the DS-forest. The search for a match begins at a node, n, in the DS-forest and continues in the expression to be normalised using the expr field of the node to the expression. If the match fails, one of the children (or a parent) of node n at which the match failed is selected for initiating another match attempt. The child (or parent) is selected based on the reduction strategy being implemented. If the match succeeds, then the update procedure is called and following that the next node for a match attempt is selected based on the reduction strategy.

3 Implementing Strategies with the DS-forest

Implementing the leftmost/rightmost outermost/innermost strategies is straightforward with the DS-forest, if pointers are kept to the leaf nodes in the forest as well (for innermost strategies). The leftsibling field is used for leftmost and the rightsibling field is used for rightmost strategies.

For outermost strategies, matching attempts are initiated starting from roots and proceed to leaves in the DS-forest until the first match is found, say at node n. After a match, if the rules are all left-linear, then match attempts proceed to the closest ancestor of n such that the sum of the dist fields of the node from n to that ancestor does not exceed the maximum height of any lhs, say m. If there is no such ancestor, then match attempts proceed downwards, otherwise a match attempt is initiated at the ancestor of n. The idea is to take advantage of the fact that a left-linear rule that did not match earlier at a node higher than m levels up in the expression tree will still not match after a successful match at a node that is deeper than its height. In programming applications, the rules are usually left-linear so this backtracking method is expected to outperform routinely backtracking to root nodes. For innermost strategies, normalization proceeds from leaves towards roots. On unsuccessful match attempts, the procedure continues upwards. On a successful match the procedure initiates subsequent matches at the lowest defined symbols in the rhs instance (this idea can be improved somewhat but we omit the optimizations here for lack of space).

4 Experimental Results

We have implemented the DS-list and a preliminary version of DS-forest (dist fields are ignored) structure in LRR 1.2. A linux version of LRR 1.2 and some examples can be downloaded from `http://www.cs.uh.edu/~rmverma/linux_lrr` . LRR consists of a term graph interpreter TGR, and a term graph rewriter that tables or stores the history of its reductions, called Smaran, based on the congruence closure normalization algorithm. This algorithm treats rules as equations, hence it keeps equivalence classes of terms. Terms are represented implicitly via *signatures* and there is at most one special signature in each class called the *unreduced signature* of the class (for details, please see [7,1]).

Version 1.2 of LRR provides: (i) the original reduction strategies for Smaran and TGR described in [9], which we call the Smaran strategy and TGR strategy respectively, (ii) an efficient version of leftmost-outermost for left-linear rules for TGR and leftmost-outer for Smaran, (iii) a combination of aspects of Smaran strategy and the leftmost-outer strategy, for Smaran, (iv) DS-list and DS-forest strategies (currently available for Smaran).

Briefly, as explained in [9], the Smaran strategy on a successful match immediately tries to reduce the right-hand side instance *or its subexpressions*. If it failed to reduce any subexpression of the rhs instance, the program would backtrack all the way to the *root* of the current expression to be reduced.

In the combination strategy implemented for Smaran, the action on a successful match is to repeatedly try and reduce the rhs instance *but not its subexpressions*. On failure to reduce the rhs instance, the program backtracks to the subexpression which is exactly m levels above the rhs instance. Here $m = min(max\{height(l) \mid l \to r \in R\}$, level of rhs instance) and R denotes the rewrite system. This means that we backtrack either to the root of the current expression or to a subexpression whose distance from the rhs instance is no more than the maximum height of any lhs, whichever is closer. Because Smaran has full sharing of common subexpressions, failure to match inside the backtracked subexpression means that we must backtrack to the root eventually in all "smarter" outer strategies.

The leftmost-outermost in LRR 1.2 is a pure strategy, i.e., it does not try to reduce the rhs instance after a successful match and immediately backtracks to the node m levels up, where m is as given above (backtracking must go eventually to the root). This strategy correctly implements leftmost-outermost for left-linear rules in TGR (but it is not necessarily outermost for Smaran). For nonleftlinear rules the correct strategy must backtrack all the way to the root and this will be provided as an option in the future.

Performance Results. For lack of space, we present only summary of results to illustrate the level of efficiency achieved by the LRR 1.2. LRR is implemented in C and runs on Solaris/Linux. Timings are on a 600 MHz Pentium 3 Redhat 7.3 linux kernel 2.4.18 system with 256MB of RAM using the gcc compiler (v. 2.96) with optimization level 3. Tables 1–2 summarize results for eight benchmarks (one difference is that in Fibonacci benchmark the term fib(20) is used for TGR instead of fib(60) for Smaran) that are also at the URL given above. The reductions columns show that tabling is indeed useful for some benchmarks including CTL model checking, quicksort, etc. The timings in seconds exclude initialization time of .55–.60 seconds and parsing times ranging from 0.57 to 1.38 seconds approx. for Smaran and initialization time of 0 and parsing time ranging from 0.06 to 0.74 seconds for TGR. Tables 1–2 show the different strategies available in LRR 1.2. Note that since two strategies are currently only available in the Smaran component, Table 1 does not show timings for TGR. The reductions columns show successful tabled reductions for Smaran and successful untabled reductions for TGR. The number of matching attempts is not shown.

Strategy	Tot. Time Smaran (s)	Total Reductions (Smaran)
DS-list	34.89	2,359,411
DS-tree	26.85	2,362,638
Combination	30.76	2,362,872

Table 1. Summary Results for 8 Benchmarks

Strategy	Tot. time TGR (s)	Tot. Red's (TGR)	Tot. time Smaran (s)	Tot. Red's (Smaran)
Original	114.07	2,467,832	162.7	2,362,174
leftmostouter	49.87	2,467,832	29.9	2,362,872

Table 2. Summary Results for 8 Benchmarks

Comparison with other systems. We are aware of the difficulties of comparing software systems using experimental results, which can be sensitive to the choice of benchmarks, architectures, different focus, etc. A major problem is that most systems do not even support tabling and our focus has been more on optimizing the tabled version. Nevertheless, with the ELAN interpreter (version 3.6e) and the Maude 2.0 system [4] we ran three out of eight benchmarks (Dfa, Qsort and Rev). The total normalization time was ELAN – 839.58

seconds (Rev – created a list of 15000 elements and reversed it instead of 30000 in Tables 1–2), and for Maude – 180.4 (Qsort – only 2500 elements instead of 5000 in Tables 1–2).

5 Discussion and Future Work

We presented DS-list and DS-forest - new data structures for speeding up normalization and efficiently, naturally implementing many reduction strategies. The DS-list beats the best earlier strategy [9] and is competitive with the new strategies. A preliminary implementation of DS-forest in the tabling component of LRR outperforms all other strategies. Hence, we plan to do a full implementation of the DS-forest data structure.

Acknowledgements. We thank K.B. Ramesh and S. Kolli for initial work on Smaran, S. Senanayake for work on LRR, and Z. Liang for the CTL model checking program.

References

1. L. Bachmair, C. R. Ramakrishnan, I. V. Ramakrishnan, and A. Tiwari. Normalization via rewrite closures. *Lecture Notes in Computer Science*, 1631:190–204, 1999.
2. P. Borovansky et al. ELAN user manual, September 2002. ELAN Version 3.6.
3. E. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, 1999.
4. M. Clavel, F. Durán, S. Eker, et al. The Maude 2.0 system. *Proc. Rewriting Techniques and Applications (RTA 2003)*, LNCS 2706, pages 76–87. Springer-Verlag, June 2003.
5. M. Hermann, C. Kirchner, and H. Kirchner. Implementations of term rewriting systems. *Computer Journal*, 34(1):20–33, 1991.
6. M. van den Brand, J. Heering, P. Klint, et al. Compiling rewrite systems: The asf+sdf compiler. *ACM Transactions on Programming Languages and Systems*, 24:334–368, 2002.
7. Rakesh M. Verma. A theory of using history for equational systems with applications. *Journal of the ACM*, 42(5):984–1020, 1995. Also in the 32nd IEEE FOCS Symposium, 1991.
8. Rakesh M. Verma. Static analysis techniques for equational logic programming. In *Proc. (elec.) of 1st ACM SIGPLAN Workshop on Rule-based Programming*, 2000.
9. Rakesh M. Verma and Shalitha A. Senanayake. $\boldsymbol{LR^2}$: A laboratory for rapid term graph rewriting. In *Proc. Conf. on Rewriting Techniques & Applications*, pages 252–255, 1999.

Strategies in Programming Languages Today

The role of strategies in the realistic modeling, analysis, optimization and application of existing programming languages.

Round table: Salvador Lucas moderator

Participants: Francisco Durán and Claude Kirchner, Ralf Lämmel.

Our abstract models, definitions and analysis techniques are basically developed for full rewriting with TRSs and they do not easily apply to real programs using

- strategy languages (or a fixed strategy),
- conditional TRSs,
- type/sort information,
- higher-order functions,
- polymorphism,
- data structures,
- built-in symbols,
- ACI symbols,
- shared information (graphs instead of terms),
- modules,
- ...

In this setting, we should ask whether we have the appropriate

- notion of strategy?
- definition and methods for analyzing termination of programs?
- definition and methods for analyzing determinism, uniqueness of NFs, ... ?
- approach for analyzing complexity and measuring efficiency?
- strategy languages?

Maude's Internal Strategies

Francisco Durán

DLCC, Universidad de Málaga, Campus de Teatinos, Málaga, Spain
duran@lcc.uma.es

Abstract Maude is a reflective language supporting both rewriting logic and membership equational logic. Reflection is systematically exploited in Maude, endowing the language with powerful metaprogramming capabilities, including declarative strategies to guide the deduction process.

1 Introduction

Maude [3,4] is a high-level language and high-performance system supporting both equational and rewriting logic computation for a wide range of applications. Rewriting logic [13] is a logic of change that can naturally deal with state and with highly nondeterministic concurrent computations. In rewriting logic, the state space of a distributed system is specified as an algebraic data type in terms of an equational specification (Σ, E), where Σ is a signature of sorts (types) and operations, and E is a set of (conditional) equational axioms. In Maude, the underlying equational logic is *membership equational logic* [14], a Horn logic whose atomic sentences are equalities $t = t'$ and *membership assertions* of the form $t : s$, stating that a term t has sort s.

Maude's functional and system modules are, respectively, membership equational theories and rewrite theories. The equations in functional modules, considered as rules in the left to right direction, are assumed to be Church-Rosser and terminating. Therefore, canonical forms are reached by canonical simplification regardless of the order of application. Although the (equational) reductions in Maude are basically *innermost* (or eager), Maude is able to exhibit an outermost (or lazy) behavior on particular operator arguments by using *strategy annotations* [10].

A rewrite theory is a pair (T, R), with T a membership equational theory, and R a collection of (labelled and possibly conditional) *rewrite rules* involving terms in the signature of T. Rewriting in (T, R) happens *modulo* the equational axioms in T.[1] The rules in R need not be Church-Rosser and need not be terminating, opening up in this way a whole world of new applications. This generality needs some control when the specifications become executable, because the user needs to make sure that the rewriting process does not go in undesired directions.

In those cases in which we just want to test for executability, or consider the evolution of the system with no specific interest in a concrete execution path, Maude provides two built-in strategies: The *rewrite* command follows a top-down lazy rule-fair strategy, and the *frewrite* command follows a position-fair bottom-up strategy. Maude also provides a *search* command, for those cases in which we are interested in exploring all possible execution paths from the starting term for states satisfying some property. The *search* command does a breadth-first exploration of the tree of possible rewrites.

In general however we may be interested in other forms of execution, and the choice of appropriate *strategies* is crucial for executing rewrite theories. In the Maude system, this need for providing strategies for controlling the rewriting process has been satisfied by developing strategies at the metalevel. Strategies are defined in extensions of the predefined module `META-LEVEL` by using predefined functions in it, like `metaReduce`, `metaApply`, `metaXapply`, etc. as building blocks. It is in this way possible to define at the metalevel a whole variety of internal strategy languages [2,5], that is, the strategy language is defined inside the same rewriting logic framework, instead of being defined as an add-on extralogical feature.

2 Reflection and the `META-LEVEL` module

Informally, a reflective logic is a logic in which important aspects of its metatheory can be represented at the object level in a consistent way, so that the object-level representation correctly simulates the relevant metatheoretic aspects. In particular, rewriting logic is reflective [2], and Maude's language design and implementation make systematic use of the fact that rewriting logic is reflective. The predefined functional module `META-LEVEL` efficiently implements key functionality of the universal theory $\mathcal{U}$.

[1] Maude supports rewriting modulo all combinations of associativity, commutativity, and identity.

`META-LEVEL` has sorts `Term` and `Module`, so that the representations of a term t and of a module $\mathcal{R}$ are, respectively, a term $\overline{t}$ of sort `Term` and a term $\overline{\mathcal{R}}$ of sort `Module`. The module `META-LEVEL` also provides key metalevel functions for rewriting and evaluating terms at the metalevel, namely, `metaApply`, `metaXapply`, `metaRewrite`, `metaReduce`, etc. For example, the function `metaReduce` takes as arguments the representation of a module $\mathcal{R}$ and the representation of a term t in that module, and returns the representation of the fully reduced form of the term t using the equations in $\mathcal{R}$, together with its corresponding sort or kind:

```
op metaReduce : Module Term -> ResultPair [special ...] .
op {_,_} : Term Type -> ResultPair [ctor] .
```

The operation `metaXapply` applies a rule on a term in any possible position. The first four arguments are the metarepresentation of a module $\mathcal{R}$, the metarepresentation of a term t in $\mathcal{R}$, a label l of some rules in $\mathcal{R}$, and a set of assignments (possibly empty) defining a partial substitution σ for the variables in those rules. The last natural number enumerates the solutions, since there can be different such rewrites with different substitutions and at different positions. The other two numeric arguments indicate the minimum and maximum depth in the term where the application of the rule can take place.

```
op metaXapply : Module Term Qid Substitution Nat Bound Nat
     ~> Result4Tuple? [special ...] .
op {_,_,_,_} : Term Type Substitution Context
     -> Result4Tuple [ctor] .
```

`metaXapply` returns a tuple of sort `Result4Tuple` consisting of a term, with the corresponding sort or kind, a substitution, and the context inside the given term where the rewriting has taken place.

3 Internal Strategies

There is great freedom for defining many different types of strategies, or even many different strategy languages inside Maude. This can be done in a completely user-definable way, so that users are not limited by a fixed and closed particular strategy language.

Rewriting logic has very good properties as a logical and semantic framework, in which many other logics and many semantic formalisms can be naturally represented [11,15]. In Maude, the meta-theory of rewriting logic is accessible to the user in a clear and principled way, giving to Maude very good properties as a logical and semantic framework, in which many different logics and formalisms can be expressed and executed. In fact, some of the most interesting applications of Maude are *metalanguage* applications, in which Maude is used to create executable environments for different logics, theorem provers, languages, and models of computation.

Reflection allows a complete control of the rewriting of a given term using the rewrite rules in a theory. This expressive power has been used in different applications. For example, in Real Time Maude [17] modules there is a distinction between eager and lazy rules, and only rewriting paths that satisfy the requirement that lazy rules are only applied when no eager rule can be applied make sense for this kind of modules; a object-fair strategy was used in Mobile Maude [6] a long time before such an strategy was available in Maude (such an internal strategy was in fact a prototype specification of the `frewrite` object-fair strategy currently available in Maude); Durán, Escobar and Lucas have proposed in [7] an extension of Full Maude which includes commands that compute (constructor) normal forms of initial expressions even when the use of strategy annotations together with the built-in computation strategy of Maude is not able to obtain them; the same authors have proposed in [8] another extension furnishing Maude with the ability of dealing with on-demand strategy annotations; Braga [1] extended Full Maude to support rewrites in the conditions of rules some time before it was available in Maude to be able to represent Action Semantics [16]; Pita and Martí-Oliet proposed in [18] the use of a meta-object to control the execution of a set of rules, which had to be applied following a specific order; etc.

Although very powerful, there are many applications in which simpler strategies are enough, for which it would be desirable to provide ways of avoiding the conceptual complexity of going to the metalevel. In this line, Martí-Oliet, Meseguer, and Verdejo have proposed in [12] an object-level basic strategy language for Maude very close to the ELAN strategies, and Durán, Roldán and Vallecillo have proposed generic invariant-driven strategies that control the execution of systems by guaranteeing that the given invariants are satisfied [9]. Both proposals has been implemented in Maude (the first one as an extension of Full Maude), which shows the expressiveness of the reflective capabilities of Maude for defining strategies.

The Maude system, its documentation, a collection of examples and case studies, and a list of related papers are available (free of charge) at `http://maude.cs.uiuc.edu`.

References

1. C. O. Braga. *Rewriting Logic as a Semantic Framework forModular Structural Operational Semantics*. PhD thesis, Departamento de Informática, Pontificia Universidade Católica de Rio de Janeiro, Brasil, 2001.
2. M. Clavel. *Reflection in General Logics and in Rewriting Logic with Applications to the Maude Language*. PhD thesis, Universidad de Navarra, 1998.
3. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, 2002.
4. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. Maude 2.0 manual. Available in `http://maude.cs.uiuc.edu.`, June 2003.
5. M. Clavel and J. Meseguer. Reflection in conditional rewriting logic. *Theoretical Computer Science*, 285, 2002.
6. F. Durán, S. Eker, P. Lincoln, and J. Meseguer. Principles of Mobile Maude. In *Proceedings of Joint Symposium ASA/MA 2000*, volume 1882 of *Lecture Notes in Computer Science*. Springer, 2000.
7. F. Durán, S. Escobar, and S. Lucas. New evaluation commands for maude within full maude. In *Proceedings of 5th International Workshop on Rewriting Logic and its Applications (WRLA'04)*, 2004.
8. F. Durán, S. Escobar, and S. Lucas. On-demand evaluation for maude. In *Proceedings of RULE'04*, 2004.
9. F. Durán, M. Roldán, and A. Vallecillo. Invariant-driven strategies for maude. In *Proceedings of WRS'04*, 2004.
10. S. Eker. Term rewriting with operator evaluation strategy. In C. Kirchner and H. Kirchner, editors, *Proceedings of 2nd International Workshop on Rewriting Logic and its Applications (WRLA'98)*, volume 15 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1998. Available at `http://www.elsevier.nl/locate/entcs/volume15.html`.
11. N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. volume 9, pages 1–87. Kluwer Academic Publishers, second edition, 2002.
12. N. Martí-Oliet, J. Meseguer, and A. Verdejo. Towards a strategy language for Maude. In N. Martí-Oliet, editor, *Proceedings of 5th International Workshop on Rewriting Logic and its Applications (WRLA'04)*.
13. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.
14. J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce, editor, *Recent Trends in Algebraic Development Techniques*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer, 1998.
15. J. Meseguer. Research directions in rewriting logic. In U. Berger and H. Schwichtenberg, editors, *Computational Logic*. Springer, 1999.
16. P. Mosses. *Action Semantics*. Cambridge University Press, 1992.
17. P. C. Ölveczky and J. Meseguer. Specification of real-time and hybrid systems in rewriting logic. *Theoretical Computer Science*, 285, 2002.
18. I. Pita and N. Martí-Oliet. A Maude specification of an object-oriented model for telecommunication networks. *Theoretical Computer Science*, 285, 2002.

Strategic Rewriting

Claude Kirchner

LORIA & INRIA
Nancy, France
`Claude.Kirchner@loria.fr`

Abstract This is a position paper preparing the round table organized during the 4th International Workshop on Reduction Strategies in Rewriting and Programming. I sketch what I believe to be important challenges of strategic rewriting.

1 What and How

Programming in declarative styles is highly desirable in many domains and allows us to describe *what* we wants to do rather than *how*. Not only it allows for a better understanding of what is programmed, but it permits to make safer programs and systems by allowing to describe their properties and to search or check their proofs.

This is typical of rule based programming (either in its algebraic form or "business" rules one), constraint programming, logic programming, functional programming, spreadsheet programming, . . .

Both because the declarative capabilities extend and because we want to express clever controls on them, the description of these controls them self become declarative and therefore them self subject to meta control.

2 What do we program?

Indeed, the first "devices" to be programmed are human: since the very beginning of humanity, we (human) learned for example elementary "recipes" or how to perform elementary arithmetical computations like addition.

Of course computers are today's the big programming construction site. But this is not it, as we would like to program also molecules, cells and complex organism.

In all the cases, even if some implementation details differ, we are faced with the same programming challenges for which the What and How should be specified and possibly executed.

3 Strategies in rule based languages

What makes rule based programming so particularly attractive is first, its ability to discriminate, using pattern matching, the configurations on which we want to act. Second, its capability to perform action specific to this discrimination.

Therefore the understanding of elementary actions is local and usually easy. Of course the global picture, i.e. the semantics of the action of a set of rules is much more elaborated and depends of the control put on the firing of rules.

Because this control could be itself described in a declarative way, it has been called strategies in particular in the term rewriting community. But one speak also of plans or of tactics and tacticals to program the proof search in many theorem provers.

Because programmers tend to model in more abstract ways, the programmed objects become more symbolic and therefore subject to pattern matching. Typical objects are therefore terms, graphs, propositions or clauses, constraints, . . .

4 The ELAN experience

When we started the design and implementation of the ELAN language at the beginning of the nineties[15,23], it was first to model inference processes using an almost identity function from the latex file to the ELAN's one, i.e. being able to express the inference process in a syntax very close to the mathematical one. But making such a model executable faced immediately the challenge to describe a strategy for the inference rule application.

For a given rewrite system, we defined a strategy to be just a subset of all the rewrite derivations. This set could be very elaborated and even non recursive. What is interesting here is the design and study of formalisms to describe strategies of interest. Having in mind the design of strategies for ELAN, we formalized this using either proof terms of an appropriate rewriting logic, or as an appropriate rewriting calculus term. The first approach is fully declarative and do not make precise how to describe this set. The former is less declarative but provides us with a constructive way to build the intended derivations.

We called the couple made of a rule system and a strategy a "computational system" but the term "strategic rewriting" inspired by the "strategic programming" terminology, coined by the `Stratego` team, is probably more appealing. The first strategies introduced in the language apart from the standard ones like ";", `identity, repeat, while` ... were `first, dk, dc` (don't know and don't care) followed soon by more elaborated ones like `try` [2].

Semantics The semantics of the ELAN's strategies have been first designed using rewriting logic [15,3]. A more functional semantics [4] leads finally to the emergence, study and application of the rewriting calculus [7,8][1].

Agility As experimented when coding the many applications developed in ELAN, these strategies combinators are quite useful and allow to develop concise models[2]. But they also have limitations in their agility when modeling for example intelligent backtracking when solving CSPs [5], or simply breath first search, or probabilistic choices.

Some flexibility can be achieved using an extended strategy language as developed for ELAN in [1], or by using a reflexive approach [18]. But in general, one would like to have a full strategy language having a clear and simple semantics and good practical efficiency.

5 Strategic challenges

The ELAN experience has been fruitful and inspiring (see `Stratego`[22][3] or the new `Maude` strategy language [20]) and its design and use has provided many feedbacks on what is needed to model strategies.

Here are what I currently see as challenges, of various difficulties, in the research on strategic programming as well as implementation and dissemination.

Specifying strategies The first challenge is of course to have better ways to specify strategies, taking in mind that we would like to perform of them proofs, evaluation and sharing.

From the existing strategy languages, we should find the most appropriate ways to describe strategies and the best combinators to build them. A typical instance of this problem is to setup a strategy language to conveniently model probabilistic algorithm as well as properties and proof tools for them. A different but related problem consist in describing probabilistic proof methods.

From what has been achieved so far, the rewriting calculus is the framework that I consider currently as the most promising to uniformly express strategies at the appropriate level of detail, in particular because it provides us with an extension of the lambda calculus having matching (modulo) capabilities, explicit non-determinism, and also elaborated exceptions and control mechanisms that are needed to finely express strategies. This still needs a lot of work.

Properties of strategic rewriting As soon as we change the rewriting relation by adding some restrictions on the allowed derivations, we have to re-think the associated proof tools for the generated relation.

The usual properties of confluence, termination, completeness of definition, to mention a few, can no more be checked as before since the relation has been profoundly changed.

So we need to provide new proof tools for strategy guided rewriting. First results have been achieved in this direction since a few year [19,17,12,13], but mainly every previously known result on the rewrite relation generated by a rewrite system should now be generalized to deals with a strategically controlled rewrite system.

[1] See also the rewriting calculus web page; `www.loria.fr/~faure/TheRhoCalculusHomePage`.

[2] Examples, applications and contributions are available on the ELAN web page: `elan.loria.fr`

[3] www.stratego-language.org/Stratego/StrategoHistory

Of course the computational capabilities of strategic rewrite systems change with respect to the one of rewrite systems. Therefore, this opens the door to new problems about the decidability and complexity of standard problems like reachability, termination, confluence, ..., for specific strategic rewrite systems. Static analysis, in particular via type systems, model checking or abstract interpretation are promising for guaranteeing suitable soundness program properties.

Strategies evaluations A common belief is that by making the strategies explicit, we gain in clarity and expressiveness, at the risk of a less efficient evaluation.

This is certainly true in a first approximation. But making the control explicit allows us to perform cleaver analysis to simplify or optimize the control and to generate efficient code. For example simplification of strategies are described in [1,13]. The code generation could also benefit of these partial evaluations, and we are still at the beginning of understanding how this could be adapted to strategic rewriting.

Sharing strategies A huge effort is devoted to writing strategies, and this is highly non-trivial. Of course sharing this work and knowledge is of fundamental interest and challenge. A first work in this direction concerns the sharing of strategies between the PVS and Coq proof search programming environments [16].

In general, we wish to disconnect as much as possible the strategies from the data and basic actions to which they apply, building an approach similar to the TOM one, which pattern matching and rewriting abilities are independent of the language on which they are anchored [21][4].

Strategic applications and transfers An important effort should be devoted to the in depth transfer of strategy results and technology to practical applications.

Of course a main stream is theorem proving (i.e. proof search specification) and verification as each prover or verifier is centered, either implicitly or explicitly, around clever strategies. With this respect, our experiences in using ELAN or TOM to perform model checking show that strategic programming could be at least as efficient as a dedicated model checker [9,6] but using a much more abstract description of the model.

The application to strategic programming is still in its infancy. Indeed, as shown by the experiences conducted in ELAN, `Maude`, `Stratego` or `Rogue`, the way algorithms are written is quite different when taking a strategic programming point of view. This means that both the researches for to best way to express algorithms using this programming paradigm should be developed, but also that this programming approach should be largely teach, which is not the case currently.

This may change rapidly because of the fundamental interest of the approach due to its strong distinction between the what and how, but also since applications based on XML are becoming extremely popular. This means that symbolic and in particular term structures are omnipresent and that rule base programming becomes a must. Programming languages like XSLT do not take advantage of the explicit use of strategies: strategic programming has here a big chance of exposition and development.

Another field of application is rule based system in the sense of production or business rules [14]. Here one of the fundamental ingredient are strategies, in particular in the rule activation [10]. Since this becomes also a very popular way to understand and model the organization and analysis of companies, markets, ecological, industrial or economical processes, strategic programming will also play a main role here.

As I mention at the beginning, simulating and programming biologic entities is one of the main challenge of this century. Because strategic programming provides both declarative and strategy use, and in particular permits a natural specification of concurrency, it could be one of the best programming paradigm to model complex systems like the biological ones. With this respect the first experiences using `Maude` [11] looks extremely promising and should be developed in the near future from an even more strategy oriented view point.

References

1. P. Borovanský. *Le contrôle de la réécriture: étude et implantation d'un formalisme de stratégies.* Thèse de Doctorat d'Université, Université Henri Poincaré – Nancy 1, France, October 1998. also TR LORIA 98-T-326.
2. P. Borovanský, H. Cirstea, H. Dubois, C. Kirchner, H. Kirchner, P.-E. Moreau, Q.-H. Nguyen, C. Ringeissen, and M. Vittek. ELAN *V 3.6 User Manual.* LORIA, Nancy (France), fifth edition, February 2004.

[4] See also `tom.loria.fr`.

3. P. Borovansky, C. Kirchner, H. Kirchner, and P.-E. Moreau. ELAN from a rewriting logic point of view. *Theoretical Computer Science*, 2(285):155–185, July 2002.
4. P. Borovanský, C. Kirchner, H. Kirchner, and C. Ringeissen. Rewriting with strategies in ELAN: a functional semantics. *International Journal of Foundations of Computer Science*, 12(1):69–98, February 2001.
5. C. Castro. Building Constraint Satisfaction Problem Solvers Using Rewrite Rules and Strategies. *Fundamenta Informaticae*, 34:263–293, September 1998.
6. H. Cirstea. Specifying Authentication Protocols Using Rewriting and Strategies. In *Third International Symposium on Practical Aspects of Declarative Languages*, volume 1990 of *Lecture Notes in Computer Science*, pages 138–153, Las Vegas, USA, March 2001.
7. 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.
8. H. Cirstea, C. Kirchner, L. Liquori, and B. Wack. Rewrite strategies in the rewriting calculus. In B. Gramlich and S. Lucas, editors, *Electronic Notes in Theoretical Computer Science*, volume 86. Elsevier, 2003.
9. H. Cirstea, P.-E. Moreau, and A. Reilles. Rule based programming in java for protocol verification. In N. Martí-Oliet, editor, *Proceedings Fifth International Workshop on Rewriting Logic and its Applications, WRLA 2004, Barcelona, Spain, March 27 – April 4, 2004*, Electronic Notes in Theoretical Computer Science, pages 199–217. Elsevier, 2004.
10. H. Dubois and H. Kirchner. Rule Based Programming with Constraints and Strategies. In K. Apt, A. A. C. Kakas, E. Monfroy, and F. Rossi, editors, *New Trends in Constraints, Papers from the Joint ERCIM/Compulog-Net Workshop, Cyprus, October 25-27, 1999*, volume 1865 of *Lecture Notes in Artificial Intelligence*, pages 274–297. Springer-Verlag, 2000.
11. S. Eker, M. Knapp, K. Laderoute, P. Lincoln, J. Meseguer, and K. Sonmez. Pathway logic: Symbolic analysis of biological signaling. In *Proceedings of the Pacific Symposium on Biocomputing*, pages 400–412, January 2002.
12. O. Fissore, I. Gnaedig, and H. Kirchner. CARIBOO : An induction based proof tool for termination with strategies. In *Proceedings of the Fourth International Conference on Principles and Practice of Declarative Programming*, pages 62–73, Pittsburgh (USA), October 2002. ACM Press.
13. O. Fissore, I. Gnaedig, and H. Kirchner. Simplification and termination of strategies in rule-based languages. In *Proceedings of the Fifth International Conference on Principles and Practice of Declarative Programming (PPDP)*, Uppsala, Sweden, August 2003. ACM Press.
14. C. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19(1):17–37, September 1982.
15. C. Kirchner, H. Kirchner, and M. Vittek. Designing CLP using Computational Systems. In P. Van Hentenryck and S. Saraswat, editors, *Principles and Practice of Constraint Programming*. The MIT press, 1995.
16. F. Kirchner. Towards a common tactical language : The case of coq and pvs. Master's thesis, Université Denis Diderot, Paris, France, September 2003.
17. H. Kirchner and I. Gnaedig. Termination and normalisation under strategy - Proofs in ELAN. In *Proceedigs of the Third International Workshop on Rewriting Logic and its Applications*, volume 36 of *Elecronic Notes In Theoretical Computer Science*, Kanazawa, JAPAN, September 2000. Elsevier Science Publishers B. V. (North-Holland).
18. H. Kirchner and P.-E. Moreau. A reflective extension of Elan. In J. Meseguer, editor, *Proceedings of the first international workshop on rewriting logic*, volume 4, Asilomar (California), September 1996. Electronic Notes in Theoretical Computer Science.
19. S. Lucas. Termination of context-sensitive rewriting by rewriting. In *Proceedings of the 23rd International Colloquium on Automata, Languages and Programming*, volume 1099 of *Lecture Notes in Computer Science*, pages 122–133. Springer-Verlag, 1996.
20. N. Martí-Oliet, J. Meseguer, and A. Verdejo. Towards a strategy language for Maude. In N. Martí-Oliet, editor, *Proceedings Fifth International Workshop on Rewriting Logic and its Applications, WRLA 2004, Barcelona, Spain, March 27 – April 4, 2004*, Electronic Notes in Theoretical Computer Science, pages 391–414. Elsevier, 2004.
21. P.-E. Moreau, C. Ringeissen, and M. Vittek. A pattern matching compiler for multiple target languages. In G. Hedin, editor, *International Conference on Compiler Construction - CC'2003, Varsovie, Pologne*, volume 2622 of *Lecture notes in Computer Science*, pages 61–76, Apr 2003.
22. E. Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag, May 2001.

23. M. Vittek. ELAN*: Un cadre logique pour le prototypage de langages de programmation avec contraintes.* Thèse de Doctorat d'Université, Université Henri Poincaré – Nancy 1, October 1994.

Programmable rewriting strategies in Haskell
— White Paper —*

Ralf Lämmel [1,2]

[1] Vrije Universiteit Amsterdam
[2] Centrum voor Wiskunde en Informatica
ralf@cwi.nl

Abstract Programmable rewriting strategies provide a valuable tool for implementing traversal functionality in grammar-driven (or schema-driven) tools. The working Haskell programmer has access to programmable rewriting strategies via two similar options: (i) the *Strafunski* bundle for generic functional programming and language processing, and (ii) the "*Scrap Your Boilerplate*" approach to generic functional programming. Basic rewrite steps are encoded as simple functions on datatypes. Rewriting strategies are polymorphic functions composed from appropriate basic strategy combinators.

We will briefly review programmable rewriting strategies in Haskell. We will address the following questions:

- What are the merits of Haskellish strategies?
- What is the relation between strategic programming and generic programming?
- What are the challenges for future work on functional strategies?

Acknowledgements.

The *Strafunski* project [1,27,19,25,26,28] is joint work with Joost Visser.
The "*Scrap Your Boilerplate*" approach [2,22,23] is joint work with Simon Peyton Jones.

1 Strategic programming

Our use of the term 'strategy' originates from the work on programmable rewriting strategies for term rewriting à la Stratego [30,40,38]. Strategic programmers can separate basic rewrite steps from the overall scheme of traversal and evaluation. These schemes are programmable by themselves! There are one-layer traversal primitives that facilitate the definition of whatever recursion pattern for traversal. There are further, perhaps less surprising, basic combinators for controlling the evaluation in terms of the order of steps, the choices to be made, the fixpoints to be computed, and others. An extended exposition of what we call 'strategic programming' can be found in [24].

Related forms of programmable strategies permeate computer science. For instance, evaluation strategies without any traversal control are useful on their own in rewriting [7,4]. In theorem proving, one uses a sort of strategies as proof tactics and tacticals [33]. In parallel functional programming, one uses a sort of strategies to synthesise parallel programs [36].

2 Functional strategies in *Strafunski*

The *Strafunski* project [1,27,19,25,26,28] incarnated programmable rewriting strategies for functional programming, namely for Haskell. Strategies are essentially polymorphic functions on datatypes (or 'term types'). The basic rewrite steps are readily written down as monomorphic functions on datatypes. For instance, the following rewrite step encodes some sort of constant elimination for arithmetic expressions:

```
const_elim  :: Expr -> Maybe Expr
const_elim  ((Const 0) `Plus` x)  =  Just x
const_elim  _                     =  Nothing
```

* This white paper served as an invited position paper for the 4th International Workshop on *Reduction Strategies in Rewriting and Programming* (WRS 2004), June 2, 2004, Aachen, Germany. The paper was presented at the WRS 2004 round table "*Strategies in programming languages today*".

In concrete syntax, and without Haskellish noise, this reads as "`0 + x -> x`". In the example, we wrap the result of the rewrite step in the `Maybe` monad, which allows us to observe success vs. failure of a rewrite step. We can use arbitrarily stacked monads (rather than just `Maybe`) in rewrite steps and strategies. This allows us to deal with state, environment, nondeterminism, backtracking, and so on.

In *Strafunski*, there are two (monadic) types of strategies:

- `TP` — type-preserving strategies: domain and co-domain coincide.
- `TU` — type-unifying strategies: all datatypes are mapped to one result type.

Strafunski's strategy library supports a small set of predefined strategy combinators:

- `idTP` — the identity function.
- `failTP` — the always failing strategy.
- `adhocTP` — update strategy in one type.
- `seqTP` — sequential composition.
- `choiceTP` — left-biased choice.
- `allTP` — apply a strategy to all immediate subterms.
- `oneTP` — apply a strategy to one immediate subterm.
- Similar operators are offered for `TU`.

Rewrite steps can be turned into functional strategies using the `adhocTP` combinator.
The strategy `(idTP ‘adhocTP‘ const_elim)` will succeed for all types other than `Expr`.
The strategy `(failTP ‘adhocTP‘ const_elim)` will fail for all types other than `Expr`.

We can now define all kinds of reusable evaluation and traversal schemes, e.g.:

```
-- Exhaustive application of a strategy
repeatTP        :: MonadPlus m => TP m -> TP m
repeatTP s      =  (s ‘seqTP‘ (repeatTP s)) ‘choiceTP‘ idTP

-- Full type-preserving traversal in top-down order.
full_tdTP       :: Monad m => TP m -> TP m
full_tdTP s     =  s ‘seqTP‘ (allTP (full_tdTP s))

-- Type-preserving traversal stopping at successful branches.
stop_tdTP       :: MonadPlus m => TP m -> TP m
stop_tdTP s     =  s ‘choiceTP‘ (allTP (stop_tdTP s))

-- One-hit type-preserving traversal in bottom-up order.
once_buTP       :: MonadPlus m => TP m -> TP m
once_buTP s     =  (oneTP (once_buTP s)) ‘choiceTP‘ s
```

The essence of "The essence of strategic programming" [24]

As an illustrative use case, the strategy

```
once_buTP (idTP ‘adhocTP‘ const_elim)
```

attempts a single constant elimination when given a term. Applying this strategy exhaustively (cf. the evaluation strategy `repeatTP`), amounts to (naive) innermost normalisation. This use case demonstrates the overall tenor of strategic programming:

> *Separate problem-specific rewrite steps (i.e.,* `const_elim`*) from the overall, possibly reusable scheme for traversal and evaluation (i.e.,* `once_buTP`*). Both parts are put together by mere parameter passing, or by function composition. The schemes for traversal and evaluation are fully programmable by the virtue of one-layer traversal primitives (i.e.,* `allTP` *and* `oneTP`*).*

3 What are the merits of Haskellish strategies?

Applied setup

Haskellish strategies were born in an applied programming context. That is, we have designed them in an attempt to make functional programming fit for the implementation of program analyses and transformations — as relevant in the context of language implementation, software reverse engineering, re-engineering, and others. For instance, *Strafunski*'s functional strategies readily deal with huge systems of algebraic datatypes as opposed to making assumptions such as use of single datatypes [15] or functorial encodings [35]. Also, functional strategies are versatile in terms of the recursion schemes that can be accommodated — when compared to programming with merely generalised folds [31]. Furthermore, functional strategies are conveniently customisable, whereas customisation is considered as a subordinated issue in other setups, which offer fully generic functions such as generic maps [14,13]. Customisation is crucial for strategic programming because traversal strategies involve type-specific cases on a regular basis.

Functional strategies have been used in various ways, e.g.:

- State-of-the-art Haskell refactoring tools [29].
- Language extension for Fortran [9].
- Java refactoring [27] (a subset of Java to be precise).
- Simple software metrics for Java [28].
- Reverse engineering for Cobol [28] (call-graph extraction).
- A framework for language-parametric refactoring [20].

Language economy

Functional strategies are easily supported in Haskell. There are different implementational models [27,19,26]. No proper language extension is needed. Class instances of a simple structure are sufficient to support the basic strategy combinators. The derivation of these instances is automated. Most strategic idioms are readily provided by Haskell. That is, rewrite steps are just functions defined by pattern matching. Functional strategies are nothing but Haskell functions. Monads [41] fit nicely with the effects that one encounters during strategic programming. The `Maybe` monad models the potential of failure. The list monad (and friends) is used to deal with nondeterminism and backtracking, alike for the state and the environment monad. Haskell has a strong record in implementing combinator libraries for programming domains, e.g., for parsing, pretty printing, XML processing, graphical user interfaces, and data structures. *Strafunski*'s strategies come just as another combinator library. Strategic programming in Haskell means that debugging, compilation, type checking, type inference, etc. come for free.

Strongly typed, first-class strategies

Strategy combinators are higher-order functions, which carry interesting types. So Haskell, again, is the right choice. Firstly, the type of a strategy combinator clarifies if it is type-preserving ("`TP`") or type-unifying ("`TU`"). Secondly, the chosen `Monad` instance in the type points out effects including potential of failure. Thirdly, the type indicates possible arguments that need to be passed in addition to the term, on which traversal is performed. While the influential system Stratego is largely untyped (but it could be typed [21]), Haskellish strategies are typed in all beauty of polymorphism and higher-order functions. This is taken to a limit in "*The Sketch of a Polymorphic Symphony*" [19], where we define 'the mother of traversal', which is a highly parametric traversal scheme.

We adopt an example from [20] to illustrates the virtue of typed, higher-order strategies. The following function signature types a strategy `extract` for a language-parametric program transformation. That is, the strategy models the *extraction refactoring* for whatever abstraction form — be it a method declaration, a function declaration, or others:

```
extract :: Abstraction abstr name tpe apply
    => TU [(name,tpe)] Identity         -- Recognise declarations
    -> TU [name] Identity               -- Recognise using references
    -> (apply -> Maybe apply)           -- Recognise focused fragment
    -> ([abstr] -> [abstr])             -- Mark host for new abstraction
```

```
-> ([abstr] -> Maybe [abstr])       -- Remove marking for host
-> ([(name,tpe)] -> apply -> Bool)  -- Side conditions on fragment
-> name                             -- Name for new abstraction
-> prog                             -- Input program
-> Maybe prog                       -- Output program
```

The above Haskell type clearly identifies 4 type parameters for syntactical categories `prog` (programs), `abstr` (abstraction form), `name` (name of parameters and abstractions), and `tpe` (type of parameters) with a relationship `Abstraction` on them for the sake of making the function `extract` parametric with regard to the relevant abstraction form.

Using an untyped `extract` is beyond a Haskell programmer's imagination. How would one possibly understand and correctly use an *untyped* function with 8 value arguments; 6 of the 8 of a higher-order type; 2 out of the 6 of a strategically polymorphic type?

4 Isn't strategic programming just generic programming?

In *Strafunski*, strategy types are opaque. *Strafunski*'s strategy library really provides an abstract datatype for strategies. This allows for different models of strategies. Some models have been described in the literature [27,19,26]. Some strategic improvements could be accommodated by new models without changing *Strafunski*'s API. The opaque status also encourages a point-free style (or combinator style) of strategic programming. We can clearly see that *Strafunski*'s strategy types are opaque because there are even basic combinators for strategy application, which resemble function application:

```
applyTP      :: (Monad m, Term t) => TP m -> t -> m t
applyTP s t  =  ... -- opaque implementation omitted
```

However, strategy types are not inherently opaque, and in the "*Scrap Your Boilerplate*" approach to generic programming [2,22,23] they indeed aren't. In this approach, generic traversal schemes and all that are just straight polymorphic functions, possibly of a rank-2 type (as supported by the GHC implementation of Haskell, but also elsewhere). The "*Scrap Your Boilerplate*" approach is based on two Haskell classes `Typeable` and `Data` (the former being a superclass of the latter) for a handful of generic function combinators. (The GHC implementation of Haskell derives these classes automatically.) *Strafunski*'s strategy library can be reconstructed in this framework [26] by basically using just two of its combinators: `cast` for type-safe cast and `gfoldl` for one-layer traversal.

Strategies types become very non-opaque, concise and versatile now:

```
type GenericM m = forall a. Data a => a -> m a  -- corresponds to TP m
type GenericT   = forall a. Data a => a -> a    -- non-monadic variation on TP m
type GenericQ r = forall a. Data a => a -> r    -- the type-unifying scheme for 'queries'
```

In *Strafunski*, we did not favour variations like `GenericT` because this would have implied a proliferation of combinators for the various opaque types. To illustrate the use of these `forall` types, we reconstruct the traversal scheme `stop_tdTP`:

```
stop_tdTP :: GenericM Maybe -> GenericT
stop_tdTP s x = case s x of
                  Nothing -> gmapT (stop_tdTP s) x
                  Just x' -> x'
```

We have used here `gmapT :: GenericT -> GenericT`, which is the non-monadic variation on `allTP` [22]. The type of `stop_tdTP` says that this combinator takes a polymorphic function and returns one. We use the type aliases for readability; we could as well inline the `forall` types. As an exercise in versatility, we have reconstructed a more specifically typed scheme `stop_tdTP`. The original scheme involved the opaque type `TP m`, where `m` could be instantiated later to any instance of `MonadPlus`. The reconstructed scheme fixes the monad for the argument type to `Maybe`, which allows us to guarantee success of the composed strategy (cf. the non-monadic result type `GenericT`).

Once we get used to forming generic function types, we will not limit ourselves to strategy types. That is, while strategies are unary polymorphic functions on datatypes, there are other polymorphic type schemes of interest. Generic functions do not need to be unary, neither do they need to be polymorphic in the argument position. For instance:

```
type GenericEq  = forall a b.
  (Data a, Data b) => a -> b -> Bool       -- generic equality
type GenericB   = forall a. Data a => a    -- build a term; no traversal!
type GenericR m = forall a. Data a => m a  -- read a term using a monad
```

So while the *Strafunski* approach emphasised unary term traversal, the "*Scrap Your Boilerplate*" approach to generic functional programming allows us to abstract over more than just unary term traversal. We can abstract over multi-parameter traversal, over term generation, serialisation, and de-serialisation, zipping, and others [23]. Especially the correspondence between term traversal and term building is a duality that was uncovered some time ago by squiggolists: given a regular datatype (such as lists), or perhaps even any datatype, one can fold a datum of the type ("traverse it"), and unfold it ("build it") [31,3]. Other generic programming approaches also serve this generality. For instance, generic programming extensions like PolyP [14] or Generic Haskell [12,6] provide special forms of polytypic or generic function declarations that use structural induction on the type structure as prime notion. In this context, the "*Scrap Your Boilerplate*" approach is characterised as follows:

- The approach blends well with normal Haskell programming.
- The approach is lightweight. It is based on two simple Haskell type classes.
- The approach does not require any compile-time code specialisation.
- Generic functions operate directly on Haskell datatypes without a representation layer.
- Generic functions are true first-class citizens, e.g., traversal schemes are higher-order.
- Generic functions are easily customised by (nominal) type case.

5 Where to go from here?

Strategic programming is a young research field. Several challenges are readily waiting. The following list is biased towards functional strategies, and relates to the current *Strafunski* and "*Scrap Your Boilerplate*" implementations, but most challenges are relevant for programmable rewriting strategies in general.

Analysis opportunities

The functional strategist might want to take advantage of analyses that improve static guarantees or run-time performance of his or her strategies. Some prime examples follow:

Termination Strategic traversal schemes are like recursion schemes: they are meant as disciplined replacement for free-wheeling recursive programming. Nevertheless, the versatility of strategies makes it still quite easy to encode diverging strategies. For instance, `(repeatTP idTP)` will diverge. The implied usage pattern for fixpoint iteration with `repeatTP` is that the argument strategy should eventually fail.

Stupidity Just as there are 'stupid casts' in object-oriented programming (i.e., type casts that cannot possibly succeed), so there are 'stupid strategies' in strategic programming. For instance, the strategy `(full_tdTP (failTP ‘adhocTP‘ f))` is stupid because a full traversal is meant to have a chance of succeeding for whatever type, but the given composition will undoubtedly fail for all types except for the domain of `f`.

Shortcutting On the basis of the type-specific cases of a strategy it would be often feasible to shortcut traversal leading to a more efficient traversal. For instance, the strategy `(full_tdTP (idTP ‘adhocTP‘ f))` does not need to be pushed into a term any further if it is clear that subterms of `f`'s domain are out of reach — on the basis of static type information. For such hopeless branches, the strategy can be shortcut to `idTP`.

Composability Chains of strategies need to cooperate in the sense that a given strategy in the chain should be enabled, or at least not disabled by earlier elements in the chain. (One could call this an advanced form of stupidity perhaps, so it is not stupid!) Enabling and disabling can be understood in terms of pre- and post-conditions for strategies, in which case work on program transformation might be of use [18,34].

Expressiveness opportunities

The functional strategist might even ask for extra expressiveness, which, in an extreme case, requires Haskell extensions. Alternatively, the extra-strategic expressiveness can also be accommodated by the virtue of a more open Haskell system, or by preprocessing, or perhaps by appropriate combinator libraries. Some prime examples follow:

Sexy types There is a potential need for designated types to declare, check, and infer success behaviour, determinism, and some forms of pre- and post-conditions. Also, the effects involved in strategies (such as failure, state, environment) were more conveniently used with an effect type system perhaps [10] — as opposed to explicit monad transformers. A Haskell 20XX with a very open type system would be of use here.

Object syntax The prime application domain of strategic programming is program analysis and transformation. Encoding rewrite steps in terms of abstract syntax is relatively inconvenient for real-world programming languages. Haskell could support concrete syntax, just as rewriting technology like ASF+SDF [17,5] does already for a long time. Stratego was also equipped with concrete object syntax [39].

Graphs Many program analyses and transformations favour *graph-based* intermediate representations. Haskell's laziness allows for cyclic data structures. Node identities have to be 'managed' carefully. Constructing and transforming many-sorted graphs is difficult in Haskell. We are in need of a typeful approach that retains the convenience of pattern matching and building, and that provides us with the illusion of destructive update.

Attribute grammars Strategies and attribute grammars are complementary in that the former are more operational, whereas the latter are more declarative. Also, the former emphasise traversal, whereas the latter emphasise attribute dependencies. Research on a possible marriage of strategies and attribute grammars promises interesting insights. Alike strategies, attribute grammars are conveniently embedded into Haskell [8].

Constraint programming Another unexplored combination of worlds is the integration of programmable strategies and constraint programming, or residuation and narrowing — as available in a hybrid language like Curry [11]. Constraints could provide a versatile means to make strategies less operational, more declarative. Constraints could also provide means to narrow down the search space for strategies.

XML & XPath Next to language processing on the basis of syntaxes, strategies are thought to be useful for XML document processing. Functional combinator libraries for XML processing do exist [42], but they lack the typing strength of functional strategies. It should be possible to use strategies as a means to provide the illusion of an XPath-like language for controlling fully typed XML transformations.

Strategy mining & refactoring to strategies

The modularity and conciseness of legacy Haskell programs could benefit from the strategic style of programming. This calls for 'strategy mining'. There exists related work on recovering recursion schemes like folds in legacy code [37]. When developing and enhancing existing Haskell programs, strategic style has to be installed or improved by means of refactoring. In fact, this is a form of 'refactoring to patterns' [16] because the strategic style of programming can be viewed as a collection of design patterns for traversal functionality [25]. In both cases, entangled traversal code is turned into strategically organised traversal code.

6 Concluding remark

We have briefly reviewed Haskell-based support for programmable rewriting strategies. We have also briefly discussed the link between rewriting strategies and generic programming. Finally, we have listed challenges for future work on Haskellish rewriting strategies.

Please, stay tuned at [1,2].

References

1. The *Strafunski* web site: examples, downloads, browsable library, papers, background, 2000–2004. `http://www.cs.vu.nl/Strafunski/`.
2. The "*Scrap your boilerplate*" web site: examples, browsable library, papers, background, 2003–2004. `http://www.cs.vu.nl/boilerplate/`.
3. L. Augusteijn. Sorting morphisms. In S.D. Swierstra, P.R. Henriques, and J.N. Oliveira, editors, *Advanced Functional Programming, 3rd International School, Braga, Portugal, September 12-19, 1998, Revised Lectures*, volume 1608 of *LNCS*, pages 1–27. Springer-Verlag, 1999.
4. P. Borovansky, C. Kirchner, and H. Kirchner. Controlling Rewriting by Rewriting. In Meseguer [32].
5. M.G.J. van den Brand, Arie van Deursen, J. Heering, H.A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P.A. Olivier, J. Scheerder, J.J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In R. Wilhelm, editor, *Proc. Compiler Construction (CC 2001)*, volume 2027 of *LNCS*, pages 365–370. Springer-Verlag, 2001.
6. D. Clarke, J. Jeuring, and A. Löh. The Generic Haskell User's Guide, 2002. Version 1.23 — Beryl release.
7. M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In Meseguer [32].
8. O. de Moor, K. Backhouse, and S.D. Swierstra. First Class Attribute Grammars. *Informatica: An International Journal of Computing and Informatics*, 24(2):329–341, June 2000. Special Issue: Attribute grammars and Their Applications.
9. M. Erwig and Z. Fu. Parametric Fortran – A Program Generator for Customized Generic Fortran Extensions. In B. Jayaraman, editor, *Proceedings Practical Aspects of Declarative Languages (PADL 2004)*, LNCS, Dallas, Texas, USA, 2004. Springer-Verlag. To appear.
10. A. Filinski. Representing layered monads. In *Proceedings 26th ACM SIGPLAN-SIGACT Symposium on Principles of programming languages*, pages 175–188, San Antonio, Texas, USA, 1999. ACM Press.
11. M. Hanus. The *Curry* web site; Curry — A Truly Integrated Functional Logic Language, 2004. `http://www.informatik.uni-kiel.de/~mh/curry/`.
12. R. Hinze. A generic programming extension for Haskell. In *Proceedings 3rd Haskell Workshop, Paris, France*, 1999. Technical report of Universiteit Utrecht, UU-CS-1999-28.
13. R. Hinze. A New Approach to Generic Functional Programming. In *Proceedings 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'00)*, pages 119–132. ACM Press, 2000.
14. P. Jansson and J. Jeuring. PolyP - a polytypic programming language extension. In *Proceedings 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)*, pages 470–482, Paris, France, 1997. ACM Press.
15. P. Jansson and J. Jeuring. A framework for polytypic programming on terms, with an application to rewriting. In J. Jeuring, editor, *Proceedings Workshop on Generic Programming (WGP2000), Ponte de Lima, Portugal, Technical report ICS Utrecht University, UU-CS-2000-19*, 2000.
16. J. Kierievsky. *Refactoring to Patterns.* Addison Wesley, 2004. To appear.
17. P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, 1993.
18. G. Kniesel and H. Koch. Static Composition of Refactorings. In R. Lämmel, editor, *Science in Computer Programming; Special issue on program transformation*. Elsevier Science, 2004. To appear.
19. R. Lämmel. The Sketch of a Polymorphic Symphony. In B. Gramlich and S. Lucas, editors, *Proceedings 2nd International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2002)*, volume 70 of *ENTCS*. Elsevier Science, 2002. 21 pages.
20. R. Lämmel. Towards Generic Refactoring. In *Proceedings 3rd ACM SIGPLAN Workshop on Rule-Based Programming RULE'02*, Pittsburgh, PA, USA, October 2002. ACM Press. 14 pages.
21. R. Lämmel. Typed Generic Traversal With Term Rewriting Strategies. *Journal of Logic and Algebraic Programming*, 54:1–64, 2003. Also available as arXiv technical report cs.PL/0205018.
22. R. Lämmel and S. Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. *ACM SIGPLAN Notices*, 38(3):26–37, March 2003. Proceedings ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003).
23. R. Lämmel and S. Peyton Jones. Scrap more boilerplate: reflection, zips, and generalised casts. Draft; Submitted for publication; Available from [2], March 2004.
24. R. Lämmel, E. Visser, and J. Visser. The Essence of Strategic Programming. Draft; Available at `http://www.cwi.nl/~ralf`, 2002–2004.
25. R. Lämmel and J. Visser. Design Patterns for Functional Strategic Programming. In *Proceedings 3rd ACM SIGPLAN Workshop on Rule-Based Programming RULE'02*, Pittsburgh, USA, October 2002. ACM Press. 14 pages.

26. R. Lämmel and J. Visser. Strategic polymorphism requires just two combinators! Technical Report cs.PL/0212048, arXiv, December 2002.
27. R. Lämmel and J. Visser. Typed Combinators for Generic Traversal. In S. Krishnamurthi and C.R. Ramakrishnan, editors, *Proceedings Practical Aspects of Declarative Languages (PADL 2002)*, volume 2257 of *LNCS*, pages 137–154, Portland, OR, USA, January 2002. Springer-Verlag.
28. R. Lämmel and J. Visser. A Strafunski Application Letter. In V. Dahl and P. Wadler, editors, *Proceedings Practical Aspects of Declarative Languages (PADL 2003)*, volume 2562 of *LNCS*, pages 357–375, New Orleans, LA, USA, January 2003. Springer-Verlag.
29. H. Li, C. Reinke, and S. Thompson. Tool support for refactoring functional programs. In *Proceedings ACM SIGPLAN Workshop on Haskell*, pages 27–38, Uppsala, Sweden, 2003. ACM Press.
30. B. Luttik and E. Visser. Specification of Rewriting Strategies. In M.P.A. Sellink, editor, *2nd International Workshop on the Theory and Practice of Algebraic Specifications (ASF+SDF'97)*, Electronic Workshops in Computing. Springer-Verlag, November 1997.
31. E. Meijer, M. Fokkinga, and R. Paterson. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In J. Hughes, editor, *Proceedings 5th ACM Conf. on Functional Programming Languages and Computer Architecture, FPCA'91*, volume 523 of *LNCS*, pages 124–144. Springer-Verlag, Cambridge, MA, USA, August 1991.
32. J. Meseguer, editor. *Proceedings 1st International Workshop on Rewriting Logic and its Applications, RWLW'96, (Asilomar, Pacific Grove, CA, USA)*, volume 4 of *ENTCS*, September 1996.
33. L.C. Paulson. A Higher-Order Implementation of Rewriting. *Science of Computer Programming*, 3(2):119–149, August 1983.
34. G. Sittampalam, O. de Moor, and K.F. Larsen. Incremental execution of transformation specifications. In *Proceedings 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2004)*, pages 26–38, Venice, Italy, 2004. ACM Press.
35. S.D. Swierstra, P.R.A. Alcocer, and J. Saraiva. Designing and implementing combinator languages. In S.D. Swierstra, P.R. Henriques, and J.N. Oliveira, editors, *Advanced Functional Programming, 3rd International School, Braga, Portugal, September 12-19, 1998, Revised Lectures*, volume 1608 of *LNCS*, pages 150–206. Springer-Verlag, 1999.
36. P.W. Trinder, K. Hammond, H.-W. Loidl, and S.L. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1):23–60, January 1998.
37. G. Villavicencio and J.N. Oliveira. Reverse Program Calculation Supported by Code Slicing. In P. Aiken, E. Burd, and R. Koschke, editors, *Proceedings Working Conference on Reverse Engineering (WCRE 2001)*, pages 35–48. IEEE Computer Society Press, October 2001.
38. E. Visser. Stratego: A Language for Program Transformation based on Rewriting Strategies. System Description of Stratego 0.5. In A. Middeldorp, editor, *Proceedings Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *LNCS*, pages 357–361. Springer-Verlag, May 2001.
39. E. Visser. Meta-Programming with Concrete Object Syntax. In D. Batory, C. Consel, and W. Taha, editors, *Generative Programming and Component Engineering (GPCE'02)*, volume 2487 of *LNCS*, pages 299–315, Pittsburgh, PA, USA, October 2002. Springer-Verlag.
40. E. Visser, Z. Benaissa, and A. Tolmach. Building Program Optimizers with Rewriting Strategies. In *Proceedings ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 13–26, Baltimore, September 1998. ACM Press.
41. P. Wadler. The essence of functional programming. In ACM, editor, *Conference record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Albuquerque, New Mexico, January 19–22, 1992*, pages 1–14. ACM Press, 1992.
42. M Wallace and C Runciman. Haskell and XML: Generic combinators or type-based translation. In *Proceedings ACM SIGPLAN International Conference on Functional Programming (ICFP'99)*, pages 148–159, Paris, France, September 1999. ACM Press.

Aachener Informatik-Berichte

This is a list of recent technical reports. To obtain copies of technical reports please consult `http://aib.informatik.rwth-aachen.de/` **or send your request to: Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen, Email:** `biblio@informatik.rwth-aachen.de`

95-11 *	M. Staudt / K. von Thadden: Subsumption Checking in Knowledge Bases
95-12 *	G.V. Zemanek / H.W. Nissen / H. Hubert / M. Jarke: Requirements Analysis from Multiple Perspectives: Experiences with Conceptual Modeling Technology
95-13 *	M. Staudt / M. Jarke: Incremental Maintenance of Externally Materialized Views
95-14 *	P. Peters / P. Szczurko / M. Jeusfeld: Business Process Oriented Information Management: Conceptual Models at Work
95-15 *	S. Rams / M. Jarke: Proceedings of the Fifth Annual Workshop on Information Technologies & Systems
95-16 *	W. Hans / St. Winkler / F. Sáenz: Distributed Execution in Functional Logic Programming
96-1 *	Jahresbericht 1995
96-2	M. Hanus / Chr. Prehofer: Higher-Order Narrowing with Definitional Trees
96-3 *	W. Scheufele / G. Moerkotte: Optimal Ordering of Selections and Joins in Acyclic Queries with Expensive Predicates
96-4	K. Pohl: PRO-ART: Enabling Requirements Pre-Traceability
96-5	K. Pohl: Requirements Engineering: An Overview
96-6 *	M. Jarke / W. Marquardt: Design and Evaluation of Computer–Aided Process Modelling Tools
96-7	O. Chitil: The ς-Semantics: A Comprehensive Semantics for Functional Programs
96-8 *	S. Sripada: On Entropy and the Limitations of the Second Law of Thermodynamics
96-9	M. Hanus (Ed.): Proceedings of the Poster Session of ALP'96 — Fifth International Conference on Algebraic and Logic Programming
96-10	R. Conradi / B. Westfechtel: Version Models for Software Configuration Management
96-11 *	C. Weise / D. Lenzkes: A Fast Decision Algorithm for Timed Refinement
96-12 *	R. Dömges / K. Pohl / M. Jarke / B. Lohmann / W. Marquardt: PRO-ART/CE* — An Environment for Managing the Evolution of Chemical Process Simulation Models
96-13 *	K. Pohl / R. Klamma / K. Weidenhaupt / R. Dömges / P. Haumer / M. Jarke: A Framework for Process-Integrated Tools
96-14 *	R. Gallersdörfer / K. Klabunde / A. Stolz / M. Eßmajor: INDIA — Intelligent Networks as a Data Intensive Application, Final Project Report, June 1996
96-15 *	H. Schimpe / M. Staudt: VAREX: An Environment for Validating and Refining Rule Bases
96-16 *	M. Jarke / M. Gebhardt, S. Jacobs, H. Nissen: Conflict Analysis Across Heterogeneous Viewpoints: Formalization and Visualization
96-17	M. Jeusfeld / T. X. Bui: Decision Support Components on the Internet
96-18	M. Jeusfeld / M. Papazoglou: Information Brokering: Design, Search and Transformation
96-19 *	P. Peters / M. Jarke: Simulating the impact of information flows in networked organizations
96-20	M. Jarke / P. Peters / M. Jeusfeld: Model-driven planning and design of cooperative information systems

96-21 * G. de Michelis / E. Dubois / M. Jarke / F. Matthes / J. Mylopoulos / K. Pohl / J. Schmidt / C. Woo / E. Yu: Cooperative information systems: a manifesto
96-22 * S. Jacobs / M. Gebhardt, S. Kethers, W. Rzasa: Filling HTML forms simultaneously: CoWeb architecture and functionality
96-23 * M. Gebhardt / S. Jacobs: Conflict Management in Design
97-01 Jahresbericht 1996
97-02 J. Faassen: Using full parallel Boltzmann Machines for Optimization
97-03 A. Winter / A. Schürr: Modules and Updatable Graph Views for PROgrammed Graph REwriting Systems
97-04 M. Mohnen / S. Tobies: Implementing Context Patterns in the Glasgow Haskell Compiler
97-05 * S. Gruner: Schemakorrespondenzaxiome unterstützen die paargrammatische Spezifikation inkrementeller Integrationswerkzeuge
97-06 M. Nicola / M. Jarke: Design and Evaluation of Wireless Health Care Information Systems in Developing Countries
97-07 P. Hofstedt: Taskparallele Skelette für irregulär strukturierte Probleme in deklarativen Sprachen
97-08 D. Blostein / A. Schürr: Computing with Graphs and Graph Rewriting
97-09 C.-A. Krapp / B. Westfechtel: Feedback Handling in Dynamic Task Nets
97-10 M. Nicola / M. Jarke: Integrating Replication and Communication in Performance Models of Distributed Databases
97-13 M. Mohnen: Optimising the Memory Management of Higher-Order Functional Programs
97-14 R. Baumann: Client/Server Distribution in a Structure-Oriented Database Management System
97-15 G. H. Botorog: High-Level Parallel Programming and the Efficient Implementation of Numerical Algorithms
98-01 * Jahresbericht 1997
98-02 S. Gruner/ M. Nagel / A. Schürr: Fine-grained and Structure-oriented Integration Tools are Needed for Product Development Processes
98-03 S. Gruner: Einige Anmerkungen zur graphgrammatischen Spezifikation von Integrationswerkzeugen nach Westfechtel, Janning, Lefering und Schürr
98-04 * O. Kubitz: Mobile Robots in Dynamic Environments
98-05 M. Leucker / St. Tobies: Truth — A Verification Platform for Distributed Systems
98-07 M. Arnold / M. Erdmann / M. Glinz / P. Haumer / R. Knoll / B. Paech / K. Pohl / J. Ryser / R. Studer / K. Weidenhaupt: Survey on the Scenario Use in Twelve Selected Industrial Projects
98-08 * H. Aust: Sprachverstehen und Dialogmodellierung in natürlichsprachlichen Informationssystemen
98-09 * Th. Lehmann: Geometrische Ausrichtung medizinischer Bilder am Beispiel intraoraler Radiographien
98-10 * M. Nicola / M. Jarke: Performance Modeling of Distributed and Replicated Databases
98-11 * A. Schleicher / B. Westfechtel / D. Jäger: Modeling Dynamic Software Processes in UML
98-12 * W. Appelt / M. Jarke: Interoperable Tools for Cooperation Support using the World Wide Web
98-13 K. Indermark: Semantik rekursiver Funktionsdefinitionen mit Striktheitsinformation
99-01 * Jahresbericht 1998
99-02 * F. Huch: Verifcation of Erlang Programs using Abstract Interpretation and Model Checking — Extended Version
99-03 * R. Gallersdörfer / M. Jarke / M. Nicola: The ADR Replication Manager
99-04 M. Alpuente / M. Hanus / S. Lucas / G. Vidal: Specialization of Functional Logic Programs Based on Needed Narrowing

99-07	Th. Wilke: CTL+ is exponentially more succinct than CTL
99-08	O. Matz: Dot-Depth and Monadic Quantifier Alternation over Pictures
2000-01 *	Jahresbericht 1999
2000-02	Jens Vöge / Marcin Jurdzinski: A Discrete Strategy Improvement Algorithm for Solving Parity Games
2000-04	Andreas Becks / Stefan Sklorz / Matthias Jarke: Exploring the Semantic Structure of Technical Document Collections: A Cooperative Systems Approach
2000-05	Mareike Schoop: Cooperative Document Management
2000-06	Mareike Schoop / Christoph Quix (eds.): Proceedings of the Fifth International Workshop on the Language-Action Perspective on Communication Modelling
2000-07 *	Markus Mohnen / Pieter Koopman (Eds.): Proceedings of the 12th International Workshop of Functional Languages
2000-08	Thomas Arts / Thomas Noll: Verifying Generic Erlang Client-Server Implementations
2001-01 *	Jahresbericht 2000
2001-02	Benedikt Bollig / Martin Leucker: Deciding LTL over Mazurkiewicz Traces
2001-03	Thierry Cachat: The power of one-letter rational languages
2001-04	Benedikt Bollig / Martin Leucker / Michael Weber: Local Parallel Model Checking for the Alternation Free mu-Calculus
2001-05	Benedikt Bollig / Martin Leucker / Thomas Noll: Regular MSC Languages
2001-06	Achim Blumensath: Prefix-Recognisable Graphs and Monadic Second-Order Logic
2001-07	Martin Grohe / Stefan Wöhrle: An Existential Locality Theorem
2001-08	Mareike Schoop / James Taylor (eds.): Proceedings of the Sixth International Workshop on the Language-Action Perspective on Communication Modelling
2001-09	Thomas Arts / Jürgen Giesl: A collection of examples for termination of term rewriting using dependency pairs
2001-10	Achim Blumensath: Axiomatising Tree-interpretable Structures
2001-11	Klaus Indermark / Thomas Noll (eds.): Kolloquium Programmiersprachen und Grundlagen der Programmierung
2002-01 *	Jahresbericht 2001
2002-02	Jürgen Giesl / Aart Middeldorp: Transformation Techniques for Context-Sensitive Rewrite Systems
2002-03	Benedikt Bollig / Martin Leucker / Thomas Noll: Generalised Regular MSC Languages
2002-04	Jürgen Giesl / Aart Middeldorp: Innermost Termination of Context-Sensitive Rewriting
2002-05	Horst Lichter / Thomas von der Maßen / Thomas Weiler: Modelling Requirements and Architectures for Software Product Lines
2002-06	Henry N. Adorna: 3-Party Message Complexity is Better than 2-Party Ones for Proving Lower Bounds on the Size of Minimal Nondeterministic Finite Automata
2002-07	Jörg Dahmen: Invariant Image Object Recognition using Gaussian Mixture Densities
2002-08	Markus Mohnen: An Open Framework for Data-Flow Analysis in Java
2002-09	Markus Mohnen: Interfaces with Default Implementations in Java
2002-10	Martin Leucker: Logics for Mazurkiewicz traces
2002-11	Jürgen Giesl / Hans Zantema: Liveness in Rewriting
2003-01 *	Jahresbericht 2002
2003-02	Jürgen Giesl / René Thiemann: Size-Change Termination for Term Rewriting
2003-03	Jürgen Giesl / Deepak Kapur: Deciding Inductive Validity of Equations

2003-04 Jürgen Giesl / René Thiemann / Peter Schneider-Kamp / Stephan Falke: Improving Dependency Pairs

2003-05 Christof Löding / Philipp Rohde: Solving the Sabotage Game is PSPACE-hard

2003-06 Franz Josef Och: Statistical Machine Translation: From Single-Word Models to Alignment Templates

2003-07 Horst Lichter / Thomas von der Maßen / Alexander Nyßen / Thomas Weiler: Vergleich von Ansätzen zur Feature Modellierung bei der Softwareproduktlinienentwicklung

2003-08 Jürgen Giesl / René Thiemann / Peter Schneider-Kamp / Stephan Falke: Mechanizing Dependency Pairs

2004-02 Benedikt Bollig / Martin Leucker: Message-Passing Automata are expressively equivalent to EMSO logic

2004-03 Delia Kesner / Femke van Raamsdonk / Joe Wells (eds.): Proceedings of the Second International Workshop on Higher-Order Rewriting (HOR 2004)

2004-04 Slim Abdennadher / Christophe Ringeissen (eds.): Proceedings of the Fifth International Workshop on Rule-Based Programming (RULE 2004)

2004-05 Herbert Kuchen (ed.): Proceedings of the 13th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2004)

2004-06 Sergio Antoy / Yoshihito Toyama (eds.): Proceedings of the 4th International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2004)

2004-07 Michael Codish / Aart Middeldorp (eds.): Proceedings of the 7th International Workshop on Termination (WST 2004)

* These reports are only available as a printed version.
Please contact `biblio@informatik.rwth-aachen.de` to obtain copies.