Rule-Based Programming

9th International Workshop, RULE 2008, Hagenberg Castle, Austria, June 18, 2008

Pre-proceedings
Volume Editors

Günter Kniesel
Institut für Informatik III, Rheinische Friedrich-Wilhelm-Universität Bonn
Römerstr. 164, D-53117 Bonn, Deutschland
gk@cs.uni-bonn.de

Jorge Sousa Pinto
Departamento de Informática, Universidade do Minho
Campus de Gualtar, 4710 Braga, Portugal
jsp@di.uminho.pt

Proceedings of the 9th International Workshop on Rule-Based Programming, RULE 2008, Hagenberg Castle, Austria, June 18, 2008

Technical Report IAI-TR-08-02
ISSN: 0944-8535

Institut für Informatik III
Rheinische Friedrich-Wilhelm-Universität Bonn, 2008

http://iai.uni-bonn.de
Foreword

This volume contains the pre-proceedings of the Ninth International Workshop on Rule-Based Programming (RULE 2008). The final version of this volume will appear as an issue of the Electronic Notes in Theoretical Computer Science (ENTCS) series, published electronically through the facilities of Elsevier Science B.V. and its auspices. The ENTCS volume can be accessed at http://www.elsevier.com/locate/entcs.

This year, RULE is associated with RTA 2008, the 19th International Conference on Rewriting Techniques and Applications. Other affiliated events include the 8th International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2008), the 5th Workshop on Constraint Handling Rules (CHR 2008), and the 22nd International Workshop on Unification (UNIF 2008). The conference and workshops will run from July 14 to 18, 2008 at Hagenberg Castle, Hagenberg, Austria. Details about the conference and affiliated meetings can be found at the URL www.risc.uni-linz.ac.at/about/conferences/rta2008.

The fundamental concepts of rule-based programming are present in many areas of computer science, from theory to practical implementations. In programming languages, term rewriting is used in semantics as well as in implementations that use bottom-up rewriting for code generation. Rules are also used to perform computations in various systems; to describe logical inference in theorem provers; to specify and implement constraint-based algorithms and applications; and to describe and implement program transformations. Rule-based programming provides a common framework for viewing computation as a sequence of transformations on some shared structure such as a term, graph, proof, or constraint store. Rule selection and application is typically governed by a rich set of sophisticated mechanisms for recognizing and manipulating structures.

After the development of the principles of rewriting logic and of the rewriting calculus in the nineties, languages and systems such as ASF+SDF, BURG, CHRs, Claire, ELAN, Maude, and Stratego contributed to demonstrate the importance of rule-based programming. The area has since been experiencing a period of growth with the emergence of new concepts, systems, and application domains, such as Domain Specific Languages, Generative and Aspect-Oriented Programming, and Software Engineering activities like maintenance, reverse engineering, and testing.

The goal of this workshop is to bring together researchers from the various communities working on rule-based programming to foster advances in the foundations and research on rule-based programming methods and systems; and to promote cross-fertilization between theory and practice, and the application of rule-based programming in various important domains.

Rule'08 is the ninth in a series of workshops. The first Rule workshop was held in Montréal in 2000, and subsequent editions took place in Firenze, Pittsburgh, Valencia, Aachen, Nara, Seattle, and Paris.
Program Committee

Program co-chairs:

- Günter Kniesel, University of Bonn, Germany
- Jorge Sousa Pinto, Universidade do Minho, Portugal

Program Committee Members:

- Mark van den Brand, TU Eindhoven, Netherlands
- Horatiu Cirstea, IUT Nancy Charlemagne, France
- Steven Eker, SRI International, USA
- Maribel Fernandez, King’s College London, UK
- Jeffrey G. Gray, University of Alabama at Birmingham, USA
- Ralf Lämmel, University of Koblenz-Landau, Germany
- Salvador Lucas, Universidad Politécnica de Valencia, Spain
- Ugo Montanari, Università di Pisa, Italy
- Joost Visser, Software Improvement Group, The Netherlands
- Jan Wielemaker, University of Amsterdam, The Netherlands
- Victor Winter, University of Nebraska at Omaha, USA

Acknowledgements

We would like to thank the program committee members for their help in evaluating the papers, and also Alberto Lluch Lafuente, Alessandro Sperduti, Guillem Godoy, José Bacelar Almeida, José Barros, and Paliath Narendran, who have done additional reviewing.

We would also like to thank the RTA organization committee for taking care of the local organization of the workshop. We are grateful to Elsevier Science Publishers for publishing these proceedings in the Electronic Notes in Theoretical Computer Science (ENTCS) and Professor Michael Mislove for providing and adapting the style files for ENTCS.

Finally, a word of acknowledgment to our institutions for their support, respectively the Universities of Bonn (Germany) and Minho (Portugal).

Günter Kniesel
Jorge Sousa Pinto
Program and Table of Contents

9:00  Welcome, mutual introduction

9:15  Invited talk: Rule-Based Programming in Java.
Pierre-Etienne Moreau

10:15 Coffee Break

10:40 New Undecidability Results for Properties of Term Rewrite Systems.
Rakesh Verma

11:20 Order-sorted Unification Revisited.
Joe Hendrix and Jose Meseguer

12:00 Lunch

13:30 Constraint-aware Schema Transformations.
Tiago Alves, Paulo F. Silva and Joost Visser

14:10 A Rewriting-Based Model Checker for the Temporal Logic of Rewriting.
Kyungmin Bae and Jose Meseguer

14:50 Coffee Break

15:20 Neural Networks in Maude.
Gustavo Santos, Miguel Palomino and Alberto Verdejo

16:00 A Rewriting Logic Approach to Static Checking of Units of Measurement in C.
Mark Hills, Feng Chen and Grigore Rosu

16:40 Farewell Coffee Break
New Undecidability Results for Properties of Term Rewrite Systems

Rakesh Verma\textsuperscript{1,2}

\textit{Computer Science Department}
\textit{University of Houston}
\textit{Houston, TX}
\textit{USA}

Abstract

This paper is on several basic properties of term rewrite systems: reachability, joinability, uniqueness of normal forms, unique normalization, confluence, and existence of normal forms, for subclasses of rewrite systems defined by syntactic restrictions on variables. All these properties are known to be undecidable for the general class and decidable for ground (variable-free) systems. Recently, there has been impressive progress on efficient algorithms or decidability results for many of these properties. The aim of this paper is to present new results and organize existing ones to clarify further the boundary between decidability and undecidability for these properties. Another goal is to spur research towards a complete classification of these properties for subclasses defined by syntactic restrictions on variables. The proofs of the presented results may be intrinsically interesting as well due to their economy, which is partly based on improved reductions between some of the properties.

Keywords: Term Rewrite Systems, Decision Problems, Reachability, Confluence, Normalization Properties

1 Introduction

Programming language interpreters, proving equations, abstract data types, program transformation and optimization, and even computation itself can all be specified by a set of rules, called a rewrite system. The rules are used to replace ("reduce") subexpressions of given expressions by other expressions (usually equivalent ones in some sense). Rewriting is at the core of theorem provers and symbolic algebra algorithms for simplification. Rewrite systems can be specification languages and even programming languages.

Recently, there has been exciting progress on efficient algorithms and decision procedures for several fundamental properties of rewrite systems including reachability, joinability, confluence, unique normalization ($UN^-$), uniqueness of normal

\textsuperscript{1} Research supported in part by the National Science Foundation grant CCF-0306475.

\textsuperscript{2} Email: rmverma@cs.uh.edu

This paper is electronically published in
\textit{Electronic Notes in Theoretical Computer Science}
URL: \texttt{www.elsevier.nl/locate/entcs}
forms ($UN^=$) and existence of normal form for wider subclasses of rewrite systems. All these properties are well-known to be undecidable for the unrestricted case (see for instance [19]) and decidable for the ground (variable-free) case, [16,2,3,1,18,17].

Reachability, joinability and confluence were shown to be decidable for the class of linear, shallow rewrite systems [7], for right-ground systems [8], and for shallow right-linear systems [6]. $UN^=$ is shown decidable for linear, shallow rewrite systems [20]. $UN^-$ was shown to be decidable for left-linear, right-ground systems [17] and normalization is shown to be decidable [5] (from which decidability of existence of normal forms also follows we show here) for shallow right-linear and linear right-shallow systems. Reachability and joinability are also decidable for left-linear, growing systems [14]. Progress has also been made on the termination problem but it is not studied here.

On the other hand, as far as undecidability results for these problems are concerned, latest progress can be summarized as follows. Reachability, joinability and confluence were shown undecidable for flat systems in [11] and $UN^-$ is shown undecidable for linear, right shallow case in [5] and for right ground systems in [17]. $UN^=$ is also undecidable for right ground systems [17].

The goal of this paper is to explore the extent to which the recent decidability results can be extended to more general subclasses of rewrite systems. It turns out that for some properties - e.g., joinability, reachability, existence of normal forms and normalization - and corresponding subclasses of rewrite systems there is less room for extension and for a few (e.g., $UN^-$ and $UN^=$) there appears to be some scope for generalization. Specifically, it is proved that the following problems are all undecidable:

(i) Reachability, joinability, existence of normal form and normalization for confluent, linear, nonoverlapping, noncollapsing, var-preserving, constructor-based, and depth two systems. These results improve the corresponding results of [19], where results were given for linear systems.

(ii) Confluence for linear, noncollapsing, constructor-based, and depth two systems.

(iii) $UN^=$ for linear, noncollapsing, var-preserving, and depth two systems, and for right-ground, right-flat systems.

(iv) Joinability for linear, left-flat, var-preserving and noncollapsing systems in which right-hand sides are of depth at most two.

(v) Not $UN^-$ for linear, right-flat, var-preserving and noncollapsing systems in which lhs's are of depth at most two. This improves a result of [5].

(vi) Not $UN^-$ for left flat, right-linear, noncollapsing systems in which right-hand sides are of depth at most two; and for right-ground, right-flat systems.

(vii) Reachability, confluence, existence of normal form, and normalization for left-flat, right-linear, noncollapsing systems in which right-hand sides are of depth at most two.

(viii) Existence of normal form and normalization for flat systems.

These undecidability results are useful since they limit the search for potential decidability results and the reachability results also yield restricted varieties of rules.
that are universal for computation. Tighter reductions are also presented between some problems, the motivation being to derive new results more economically than to start from scratch every time for a decidability issue.

The organization of this paper is as follows. Section 2 defines the terms used and includes some useful new and existing results. Section 3 is devoted to two-counter machines, Section 4 to reachability, Section 5 to joinability, Section 6 to confluence, Section 7 to $UN^\sim$ and Section 8 to existence of normal form and normalization, and Section 9 to $UN^\sim$. Section 10 concludes the paper.

## 2 Preliminaries

Familiarity with basic notions of rewriting is assumed [4]. Let $\Sigma$ be a set, called a signature, with an associated arity function $\alpha : \Sigma \rightarrow \mathbb{N}$. Let $\mathcal{V}$ be a countable set disjoint from $\Sigma$. The set $T(\Sigma, \mathcal{V})$ of terms (over $\Sigma$) is defined as the smallest set containing $\mathcal{V}$ and such that $f(t_1, \ldots, t_n) \in T(\Sigma, \mathcal{V})$ whenever $f \in \Sigma$, $\alpha(f) = n$ and $t_1, \ldots, t_n \in T(\Sigma, \mathcal{V})$. The elements of the sets $\Sigma$ and $\mathcal{V}$ are respectively called function symbols and variables. Note that elements $a \in \Sigma$ for which $\alpha(a) = 0$, called constants, are included in the set $T(\Sigma, \mathcal{V})$. For a term $s$, $\text{Var}(s)$ denotes the set of variables in $s$. The symbols $s, t, u, \ldots$, with possible subscripts, are used to denote terms; $f, g, \ldots$, function symbols; and $a, b, \ldots$, constants. A function symbol $f \in \Sigma$ such that $\alpha(f) = m$ is also denoted by $f^m$. A term $f(t_1, \ldots, t_n)$ is written without parenthesis, $ft_1 \ldots t_n$, when it is unambiguous to do so. The top, $\text{top}(t)$, of a term is $t$ if $t \in \mathcal{V}$ and is $f$ if $t = ft_1 \ldots t_m$. The size, $|t|$, of a term is defined as $1$ if $t \in \mathcal{V}$ and if $t = ft_1 \ldots t_n$ it is $1 + \sum_{i=1}^{n} |t_i|$. The depth of a term $s$ is $0$ if $s$ is a variable or a constant, and $1 + \max \{\text{depth}(s_i)\}$ if $s = fs_1 \ldots s_m$.

The following definitions are from [5]. A term $t$ is called ground if $t$ does not contain variables. It is called shallow if all variable positions in $t$ are at depth $0$ or $1$. It is flat if its depth is at most $1$. It is called linear if every variable occurs at most once in $t$.

A position is a possibly empty sequence of positive numbers. If $p$ is a position and $t$ is a term, then $t[p]$ denotes the subterm of $t$ at position $p$, defined as, if $t[\lambda] = t$ (where $\lambda$ denotes the empty sequence) and $(ft_1 \ldots t_n)[i,p] = t_i[p]$ if $1 \leq i \leq n$ (and is undefined if $i > n$). Also write $t[s][p]$ (or just $t[s]$ when $p$ is clear from the context) to denote the term obtained by replacing in $t$ the subterm at position $p$ by the term $s$. For example, if $t = f(a, g(b, h(c)), d)$, then $t[2.2.1] = c$, and $t[d][2.2] = f(a, g(b, d), d)$. A substitution, denoted by $\sigma$, is a mapping from $\mathcal{V}$ to $T(\Sigma, \mathcal{V})$, homomorphically extended to a mapping from $T(\Sigma, \mathcal{V})$ to $T(\Sigma, \mathcal{V})$. Application of $\sigma$ is denoted using a postfix notation.

An (undirected) equation is an unordered pair of terms, written $s = t$ satisfying the condition that both $s$ and $t$ cannot be distinct variables (this condition ensured nontriviality of the word problem defined below). The equation is ground if $s$ and $t$ are ground terms. It is var-preserving if $\text{Var}(s) = \text{Var}(t)$. A directed equation or rule is an ordered pair of terms, written $s \rightarrow t$ satisfying the condition $\text{Var}(t) \subseteq \text{Var}(s)$ and $s$ cannot be a variable.\(^3\) This rule is ground (shallow, linear, flat) if $s$ and

\(^3\) Note that these conditions are the same as in [19]. The first condition ensures that some of the problems dealt with here do not become trivial.
t are ground (shallow, linear, flat) terms. A finite set $R$ of (ground) rules is called a (ground) rewrite system. It is called right-ground (right-shallow, right-linear, right-flat) if $t$ is ground (shallow, linear, flat). It is called left-shallow (left-linear, left-flat) if $s$ is shallow (linear, flat). For a rewrite system to be called shallow, left-shallow or any other such property every rule has to satisfy that property.

Terms $s$ and $t$ are unifiable if and only if there exists a ground term $C$ which is an instance of both $s$ and $t$. Say that $s$ overlaps $t$ if and only if a non-variable subterm $u$ of one of the two terms unifies with the other term. (To check for overlaps, relabel the variables in $s$ and $t$ so that they do not share any variables.) A set $S \subseteq T$ is nonoverlapping if and only if for all $s, t \in S$, $\not\text{overlap}(s, t)$. (Since $s$ and $t$ could be equal, the definition of nonoverlapping does not allow self-overlapping rules like associativity.) A system $R$ is nonoverlapping if and only if the set of lhs’s is nonoverlapping.

Definition 2.1 Rewrite systems in which both left- and right-hand sides are of depth at most two will be called depth two systems.

The set $D(R)$ of defined symbols of a rewrite system $R$ is $D(R) = \{\text{root}(l) \mid l \to r \in R\}$. A rewrite system is called constructor-based if no defined symbol appears at a nonroot position in any left-hand side. A rewrite system is var-preserving if $\text{Var}(s) = \text{Var}(t)$ for every rule $s \to t$ in the system. The size of an equation $s \to t$ or a rule $s \to t$ is defined to be $||s|| + ||t||$. If $R$ is a set of rules, then we define $R^- = \{s \to t \mid t \to s \in R\}$. We say that $s$ rewrites to $t$ in one step at position $p$ (by $R$), denoted by $s \to_{R,p} t$, if $s[p] = l\sigma$ and $t = s[r\sigma]_p$, for some $l \to r \in R$ and substitution $\sigma$. If $p = \lambda$, then the rewrite step is said to be applied at the topmost position (at the root) and is denoted by $s \to_R t$; it is denoted by $s \to_{nR} t$ otherwise. The rewrite relation $\to_R$ induced by $R$ on $T(\Sigma, \{\})$ is defined by $s \to_R t$ if $s \to_{R,p} t$ for some position $p$. The size, $||R||$, of a set $R$ of equations or rules is the sum of the sizes of individual equations or rules in $R$. The cardinality of a set $R$ is denoted by $|R|$. A rule (equation) is collapsing if its right-hand side (either side) is a variable. A non-collapsing rewrite system (equational theory) has no collapsing rules (equations).

If $\to$ is a binary relation, then $\leftarrow$ denotes its inverse, $\leftrightarrow$ its symmetric closure, $\to^*$ its transitive closure and $\to^+_E$ its reflexive-transitive closure. Thus, $\leftrightarrow_E$ and $\to^+_E$ denote identical relations. A reduction sequence of $s \to^+_E t$ (using $R$) is a finite sequence $s = s_0 \to_R s_1, s_1 \to_R s_2, \ldots, s_{k-1} \to_R s_k = t(k \geq 0)$, which is usually written in abbreviated form as $s = s_0 \to_R s_1 \to_R \ldots \to_R s_k = t(k \geq 0)$.

Two terms $s$ and $t$ are joinable by $R$, or $R$-joinable (notation: $s \downarrow_R t$), if there exists a term $u$ such that $s \to^+_R u$ and $t \to^+_R u$. The terms $s$ and $t$ are equivalent by $R$, or $R$-equivalent, if $s \to^+_R t$ (also written $s =_R t$). A rewrite system $R$ is confluent if every pair of $R$-equivalent terms is $R$-joinable. A left-linear, nonoverlapping system is also called orthogonal and orthogonal systems are confluent [10,15].

A term $s$ is irreducible or an $R$-normal form ($R$ will be dropped when clear from the context) if there is no term $t$ such that $s \to_R t$. An $R$-normal form of a term $s$ is an $R$-normal form $t$ such that $s \to^*_R t$. A rewrite system is $UN^=$ (alternatively has the $UN^=$ property) if whenever $n =_R N$ for normal forms $n$ and $N$ and $N$ are syntactically identical. It is $UN^-$ (alternatively has the $UN^-$ property)
if every term has at most one $R$ normal form.

**Reachability/Joinability.** *Instance:* Rewrite system, $R$, terms $s$ and $t$. *Question:* Does $s \rightarrow_R t$ or $s \downarrow_R t$?

**Normal form reachability.** *Instance:* Rewrite system, $R$, terms $s$ and $t$, where $t$ is a normal form w.r.t. $R$. *Question:* Does $s \rightarrow_R t$?

For the three problems above, also define versions in which the input rewrite system satisfies some property $P$. To indicate these versions, the phrase “for $P$ systems” is attached to the basic problem.

**$UN^= / UN^-$ / Confluence.** *Instance:* Rewrite system $R$. *Question:* Does $R$ have the $UN^= / UN^-$ / Confluence property?

**Existence of normal form.** *Instance:* Rewrite system $R$ and term $s$. *Question:* Does $s$ have an $R$ normal form?

**Word problem (WP).** *Instance:* Finite set of equations $E$ (or rewrite system $R$), terms $s, t$. *Question:* Does $s =_E t$ (or $s =_R t$)?

**Normalization.** *Instance:* Rewrite system $R$. *Question:* Does every non-variable term have an $R$ normal form?

### 2.1 Some New and Existing Useful Results

In this section, we collect some existing results and observations and some new results (in particular, Corollary 2.2, Theorem 4, Theorem 5 and Corollary 2.7) that we need for the rest of the paper. As usual, $A \leq_P B$ signifies a polynomial-time reduction from $A$ to $B$.

**Theorem 1 ([11])** Reachability, joinability and confluence are undecidable for flat systems.

**Theorem 2 ([5])** Not $UN^-$ is undecidable for linear right-shallow systems.

We improve the above result of [5] below (the Appendix includes their construction to aid the reader).

**Corollary 2.2** Not $UN^-$ is undecidable for linear, right-flat, var-preserving, and noncollapsing systems in which left-hand sides are of depth at most two.

**Proof.** First, observe that no right-hand side of any rule in the reduction from Post correspondence problem (PCP) to $UN^-$ in [5] is a variable and all right-hand sides are of depth at most one. Now there are two obstacles to proving the result: the reduction of [5] is not var-preserving, and some left-hand sides are of arbitrary depth. The first is fixed as follows. Delete constant $loop$ from the signature and the rule $loop \rightarrow loop$. Next, each rule of the form $l \rightarrow loop$, which is not var-preserving and the purpose of which is to destroy undesirable normal forms, is replaced by the rule $l \rightarrow l$.

For the second, proceed as follows. Let $PCP = \{(u_i, v_i) \mid 1 \leq i \leq n\}$ be a Post Correspondence Problem over $\{a, b\}$. Replace the rule $pcp(x, y) \rightarrow pair(x, y)$ in [5] by $n$ new rules $pcp(x, y) \rightarrow pair_{v_1}(x, y)$. Now delete the rule $pair(u_i(x), v_i(y)) \rightarrow pair(x, y)$, whose left-hand sides can have arbitrary depth, in [5]. Let $m(i) = \max(|u_i|, |v_i|)$. Without loss of generality assume that $|u_i| \geq |v_i|$. Then, $u_i = \ldots$
Reachability is undecidable for confluent, monadic (flat right-hand to get Joinability is undecidable for linear, left-flat, var-preserving and non-collapsing systems in which the right-hand sides are of depth at most two. For noncollapsing, var-preserving systems: common ancestor problem.

Corollary 2.3 For noncollapsing, var-preserving systems: common ancestor problem \( \leq_P \) joinability and joinability \( \leq_P \) common ancestor problem.
Theorem 5 Normal form reachability and reachability are undecidable for: left-flat, right-linear, noncollapsing systems in which right-hand sides are of depth at most two.

Proof. Let $R, s, t$, be an instance of joinability with $R$ a linear, left-flat, var-preserving and noncollapsing system in which the right-hand sides are of depth at most two. This result follows from observations in [19] and Theorem 4, nevertheless it is included here to aid the reader. Construct $R', s', t'$, an instance of reachability as follows. Let $true$ be a new constant and $equal$ a new binary function symbol. Let $\Sigma' = \Sigma \cup \{ equal, true \}$ and $R' = R \cup \{ equal(x, x) \rightarrow true \}$.

Let $s' = equal(s, t)$ and $t' = true$, which is an $R'$ normal form. Since $s$ and $t$ may be assumed to be ground terms over $\Sigma$, it is easy to see that $s' \xrightarrow{R'} t'$ iff $s \downarrow_{R} t$. The reduction preserves right-linearity, shallowness, noncollapsing property and the maximum depth of the right-hand sides does not increase. Note that only one shallow, right-linear, noncollapsing rule, with right-hand side of depth 0 was added to get $R'$. So the rest follows from Theorem 4.

The following three reductions (self-reducibilities) help in eliminating the non-collapsing restriction from many reductions of [19] and also simplify some of the reductions and their proofs below.

Lemma 2.4 ([17]) $WP \leq_{P} WP$ for non-collapsing equations.

Proof (sketch). Let $E, s, t$ be the instance of $WP$. Construct $E', s', t'$, where $E'$ is non-collapsing as follows. Set $s' = s$ and $t' = t$. To construct $E'$, first we discard equations of the form $x = x$ from $E$ (if they exist) without any problem. Next, each equation of the form $l = x$ in $E$, where $x$ is a variable and $l$ not, is replaced in $E'$ by the set of equations $\{ l \sigma \rightarrow fx_1, \ldots, x_n | f^{(n)} \in \Sigma \}$ and $\sigma = \{ x \mapsto fx_1, \ldots, x_n \}$. Here $x_1, \ldots, x_n$ are new variables not appearing in $l$. Equations of the form $x = r$, where $x$ is a variable and $r$ not, are handled in the same way. It is easy to verify that the reduction can be carried out in polynomial time. Correctness of the reduction is proved in the Appendix.

Similarly, it has been proved that:

Lemma 2.5 ([17]) $Reachability \leq_{P} reachability$ for non-collapsing systems.

Remark 0. Note that normal form reachability reduces in polynomial time to normal form reachability for non-collapsing systems.

Lemma 2.6 ([17]) $Joinability \leq_{P} joinability$ for non-collapsing systems.

Note that in all three reductions above if we start with a flat system, then the depth of the left-hand sides increases by one and the right-hand sides remain flat after reduction. Therefore, by Theorem 1 and Lemmas 2.5 and 2.6 we get:

Corollary 2.7 $Reachability$ for non-collapsing systems, normal form reachability for non-collapsing systems and joinability for non-collapsing systems are undecidable for systems in which left-hand sides are of depth at most two and right-hand sides are flat.

The corollary is interesting for two reasons. First, the systems constructed in [11] are collapsing systems. Hence the status of all three problems in corollary is not settled by their results. Second, it can be used to prove undecidability results
for other properties as well, which are not known to be undecidable for right-flat systems.

3 Two-Counter Machines

A two-counter machine is a Turing Machine with two semi-infinite tapes [9,13]. The tape alphabet of the machine consists of just two symbols Z and B (blank). Moreover, the symbol Z, which serves as a bottom of counter marker, appears initially in both counters. In one move the machine can change state and independently either increment, or ignore, or conditionally decrement each counter. The condition for decrementing a counter is that the counter must be positive. A transition of the two-counter machine is therefore a pair of triples of the form ((p, C₁, C₂), (q, A, B)), where p, q are states, C₁, C₂ ∈ {Z, B} and A, B ∈ {−1, 0, +1}, with the obvious restriction on the decrement −1 action. A two-counter machine is deterministic if no two transitions have the same first triple. As for Turing machines, the transition function of the two-counter machine is allowed to be a partial function. The configuration of a two-counter machine, and the yields in one step relation between configurations and its reflexive and transitive closure can all be defined formally and analogously to the corresponding definitions for Turing machines. Minsky proved the following theorem for two counter machines. See also [9], who show that the two counter machine can simulate a Turing machine [9].

**Theorem 6** Given a deterministic two-counter machine, the problem of determining whether the machine accepts the empty string is undecidable.

4 Reachability

The above theorem is used to prove the following result for reachability.

**Theorem 7** Reachability is undecidable even for confluent, noncollapsing, linear, and depth-two rewrite systems. Further it remains undecidable for rewrite systems that are in addition constructor-based and var-preserving.

**Proof.** The idea is to associate a rewrite system R(M) with a given deterministic, two-counter machine M. Without loss of generality, we may assume that: the two-counter machine has only one final state f, it empties the two counters before accepting, and there are no transitions from the final state.

For a proof that there is such a universal two-counter machine we use the deterministic, universal, two-counter machine model of [13] in which there is only one final state qf and there are no transitions from qf. We add three transitions from qf to empty the counters.

\[((q_f, B, Z), (q_f, -1, 0)), ((q_f, Z, B), (q_f, 0, -1)), ((q_f, B, B), (q_f, -1, -1))\]

and one transition \(((q_f, Z, Z), (f, Z, Z))\), where f is the only new final state of the machine so constructed (qf becomes a nonfinal state of the new machine).

Directly encoding the transitions of M as rules in a rewrite system is possible and seems natural but it leads to two problems: overlapping rules and the depth of variables can exceed two. To avoid these problems two mechanisms are used: a
checking module and a “delayed” two-step incrementing procedure for incrementing nonempty counters.

There is a constant \( p \) in the signature of the rewrite system for every state \( p \) of the two-counter machine and a constant \( Z \) to represent the empty counter. The blank symbol is simulated by a unary symbol \( B \). The signature also contains a ternary symbol \( h \), a binary symbol \( \text{equal} \), several auxiliary function symbols, and the constant \( \text{true} \).

First, we group all the transitions of \( M \) that take place on the same state, i.e., for which the first component of the first triple in the transition is the same, together. If the common state for a transition group is \( p \), we call them \( p \)-transitions. For each \( p \)-transition group of \( M \) we have a rule in \( R \) of the form \( h(p, x, y) \rightarrow h_p(\text{equal}(x, Z), \text{equal}(y, Z)) \). The rules for equal are two: \( \text{equal}(Z, Z) \rightarrow \text{true} \) and \( \text{equal}(B(x), Z) \rightarrow B(x) \). Next, we have up to four rules of the following forms, where the right-hand sides are generic terms to be specified below:

(i) \( h_p(\text{true}, \text{true}) \rightarrow h(q, A_1, B_1) \)

(ii) If \( M \) increments the second counter, then the rule is \( h_p(\text{true}, B(x)) \rightarrow h_{p2}(q, A_2, B_2) \). If it does not, then the rule is \( h_p(\text{true}, B(x)) \rightarrow h(q, A_2, B_2) \).

(iii) If \( M \) increments the first counter, then the rule is \( h_p(B(x), \text{true}) \rightarrow h_{p1}(q, A_3, B_3) \) otherwise it is \( h_p(B(x), \text{true}) \rightarrow h(q, A_3, B_3) \).

(iv) If \( M \) increments both counters, the rule is

\[
h_p(B(x), B(y)) \rightarrow h_{p12}(q, A_4, B_4).
\]

If it increments the first only, the rule is \( h_p(B(x), B(y)) \rightarrow h_{p1}(q, A_4, B_4) \). If it increments the second only, the rule is \( h_p(B(x), B(y)) \rightarrow h_{p2}(q, A_4, B_4) \). If it does not increment either counter, the rule is \( h_p(B(x), B(y)) \rightarrow h(q, A_4, B_4) \).

When \( M \) increments a non-empty counter, the simulation needs to break this into two reduction steps to avoid increasing the depth of variables beyond two. The right-hand side of these rules are obtained from the right-hand sides of the \( p \)-transitions by finding the transitions that apply to the four cases: both counters are initially zero; the first counter is initially zero, the second is not; the first is not zero, the second is; and both counters are non-zero initially. The constant \( q \) is determined by the state component of the second triple of the corresponding transition. \( A_1 \) (also \( B_1, A_2, B_3 \)) is \( Z \) if \( M \) ignores the counter and \( B(Z) \) if it increments it (it cannot decrement any counter in this case since the counter is zero). \( B_2 \) (\( A_3, A_4 \)) is \( x \) if \( M \) decrements the corresponding counter, and \( B(x) \) if it ignores or increments the corresponding counter. \( B_3 \) is \( y \) if \( M \) decrements the corresponding counter and \( B(y) \) if it ignores or increments it.

The rules for incrementing the correct counters are introduced as needed and are as follows:

\[
\begin{align*}
&h_{p1}(q, x, y) \rightarrow h(q, B(x), y) \\
&h_{p2}(q, x, y) \rightarrow h(q, x, B(y)) \\
&h_{p12}(q, x, y) \rightarrow h(q, B(x), B(y))
\end{align*}
\]

The specification of \( R(M) \) is complete. By inspection, \( R(M) \) is linear, var-preserving, constructor-based and depth-two. It is also nonoverlapping since \( M \)
is deterministic and by the construction. Since it is linear and nonoverlapping, it is also confluent.

Now let $s$ be the term $h(s_0, Z, Z)$, where $s_0$ is the initial state of $M$, and let $t$ be the term $h(f, Z, Z)$ where $f$ is the only final state of the two-counter machine. We prove that $s \xrightarrow{R(M)} t$ iff the two-counter machine $M$ accepts the empty string. The proof of the if direction is straightforward by induction on the number of steps taken by $M$ in the accepting computation.

For the only if direction, we prove the following more general result: if $h(p, A_1, A_2) \xrightarrow{R(M)} h(q, B_1, B_2)$, where $p, q$ are any state constants, and $A_i, B_i \in B^*(Z)$ \footnote{0 or more applications of B} for $i \in \{1, 2\}$, then the configuration $(p, A_1, A_2)$ yields in 0 or more steps of $M$ the configuration $(q, B_1, B_2)$. Clearly, this result implies what is needed above. The proof of the more general result proceeds by induction on the number of intermediate terms of the form $h(\ldots)$ in the reduction sequence. Details can be easily filled in by the reader. This completes the proof.

5 Joinability

The proof of Theorem 7 also yields the following results:

**Theorem 8** (i) Joinability is undecidable for confluent, linear, nonoverlapping, noncollapsing, var-preserving, constructor-based, and depth-two rewrite systems.

(ii) Normal form reachability is undecidable for confluent, linear, nonoverlapping, noncollapsing, var-preserving, constructor-based, and depth-two rewrite systems.

*Proof.* Because we started with a two-counter machine that has no transitions from the final state $f$, the term $h(f, Z, Z)$ is a normal form. This immediately proves the second statement. Now $h(s_0, Z, Z) \downarrow_{R(M)} h(f, Z, Z)$ iff $h(s_0, Z, Z) \xrightarrow{R(M)} h(f, Z, Z)$ since $h(f, Z, Z)$ is a normal form. Thus, the first statement also follows. □

6 Confluence

The following reduction is from [17]. A proof sketch is included to show how the depth bound changes in the reduction. It is partly recovered in the Remark afterward.

**Theorem 9** ([17]) Normal form reachability for non-collapsing rewrite systems \leq_p confluence.

*Proof (sketch).* Let $R, s, t$ be an instance of normal form reachability, where $t$ is an $R$-normal form and $R$ is non-collapsing. Let $a$ be a new constant and $h$ a new binary function symbol not in $\Sigma$. Let $\Sigma' = \Sigma \cup \{h, a\}$.

Let $R_1 = \{ f_1, \ldots, x_n \rightarrow h(s, f x_1 \ldots x_n) \mid f^{(n)} \in \Sigma' - \{a\}\}$.

Let $R' = R \cup R_1 \cup \{h(a, x) \rightarrow a\} \cup \{t \rightarrow a\}$. 
Note that every term $u$ in $T(\Sigma', V)$, except variables and $a$, reduces via the $R_1$ rules to the term $h(s, u)$. Since $R$ is non-collapsing and the new rules are non-collapsing, $R'$ is non-collapsing. Moreover, since no left-hand side of a rule in $R$ can be a variable and the new rules also satisfy this condition, so $R'$ also satisfies this condition. It is shown in [17] that $R'$ is confluent iff $s \xrightarrow{R} t$. □

**Remark 1.** The above reduction does not preserve groundness, but it does preserve left-linearity, right-linearity, linearity, and noncollapsing property. It also preserves left-flatness and the right-hand sides of the new rules introduced are of depth at most two (for this $s$ and $t$ must be flattened first - using the procedure of [7] for example - and their new names must be used in the new rules).

**Corollary 6.1** Normal form reachability for non-collapsing rewrite systems $\leq_P$ local confluence.

**Proof.** A very similar argument to that given in the proof of Theorem 9. Note that all critical pairs in $R'$ are joinable iff $s \rightarrow_R t$. □

**Corollary 6.2** Local confluence and confluence are undecidable for:

(i) left-flat, right-linear and noncollapsing systems in which right-hand sides are of depth at most two.

(ii) linear, noncollapsing, and depth two systems.

**Proof.** (i) Follows from Theorem 5 and Theorem 9. (ii) Follows from Theorem 7 and Theorem 9. □

## 7 Unique Normalization

We now prove several limits on $UN^-$: for left-flat, right-linear, noncollapsing systems in which right-hand sides are of depth at most two; for linear, noncollapsing and depth two systems; and for right-ground, right-flat systems.

Observe that the reduction from joinability in [17] to not $UN^-$ cannot be used here since that reduction does not preserve flatness and left-linearity. We wish to preserve flatness and linearity so we need a more complex reduction and proof and we use reachability as the starting point.

Note that in a non-collapsing system it is not possible to reduce a non-variable term to a variable. Hence we cannot have violation of $UN^-$ involving a normal form that is a variable.

**Theorem 10** Reachability for non-collapsing rewrite systems $\leq_P$ not $UN^-$. 

**Proof.** Let $R, s, t$ be an instance of reachability with $R$ non-collapsing. We first flatten $s$ and $t$ using flattening rules (see for example [7]). These rules preserve $R$ reachability of terms and are flat. Let $c_s$ and $c_t$ be the new “names” for $s$ and $t$ respectively. We add all the new names so introduced to $\Sigma$ and the flattening rules to $R$. Let $true, false$ be two new constants and $H = \{h_f \mid f^{(m)} \in \Sigma, m > 0\}$ be a set of new function symbols not in $\Sigma$. For each $f^{(m)} \in \Sigma$ the corresponding function symbol $h_f$ has arity $m + 2$. Let $\Sigma' = \Sigma \cup H \cup \{true, false\}$.

Let $T = \{c \rightarrow c \mid c \in \Sigma' - \{true, false\}\} \cup \{fx_1...x_n \rightarrow fx_1...x_n \mid f^{(n)} \in \Sigma', n > 0\}$. So that all non-variable terms in $T(\Sigma', V)$ are reducible except $true$ and $false$. Let
Let $S = \{ c \rightarrow h(c, c) \mid c \in \Sigma' - \{true, false\} \} \cup \{ fx_1 \ldots x_n \rightarrow h_f(c, x_1 \ldots x_n) \mid f^{(n)} \in \Sigma \}$.

Let $V = \{ h_f(c_1, y_1, \ldots, y_n) \rightarrow true \mid f^{(n)} \in \Sigma \}$.

Let $R' = R \cup S \cup T \cup V \cup \{ c_i \rightarrow false \}$.

To show that $R'$ is not $UN^-$ iff $s \not\Rightarrow_R t$. Suppose that $R'$ is not $UN^-$. Consider, a term $A \in \mathbb{T}(\Sigma', V)$ and suppose that $A$ has two distinct normal forms $B$ and $C$. Because of the non-collapsing requirement on $R$ and construction of $R'$, $R'$ is also non-collapsing and so neither $B$ nor $C$ can be a variable. Hence, we must have reduction sequences $p: A \not\Rightarrow_R true$ and $q: A \not\Rightarrow_R false$. We claim that $p$ and $q$ imply $s \not\Rightarrow_R t$. Note that since $R'$ is a rewrite system $A$ cannot be a variable. Then, $top(A) \in \Sigma$, since otherwise $A \not\Rightarrow_R false$ is not possible. Now since $top(A) \in \Sigma$ and there are no collapsing rules, the rule $top(A)(x_1, \ldots, x_n) \rightarrow h_f(c, x_1, \ldots, x_n)$ must have been applied at the root in sequence $p$ and subsequently the rule $h_f(c_1, y_1, \ldots, y_n) \rightarrow true$ must have been applied at the root in $p$. This implies that $c_1 \not\Rightarrow_R c$, which implies $s \not\Rightarrow_R t$. Now, through a similar argument as in the full proof of Theorem 9 ([17]) we get that $s \not\Rightarrow_R t$. Now suppose that $s \not\Rightarrow_R t$, which implies $s \not\Rightarrow_R t$. Clearly, $R'$ is not $UN^-$ since $t \not\Rightarrow_R c_1 \not\Rightarrow_R h(c, c_1) \not\Rightarrow_R h(c_1, c_1) \not\Rightarrow_R true$ and $t \not\Rightarrow_R c_1 \not\Rightarrow_R false$ and $true$ and $false$ are distinct normal forms.

Note that the above reduction preserves flatness and linearity.

**Corollary 7.1** Not $UN^-$ is undecidable for:

(i) left-flat, right-linear, noncollapsing systems in which right-hand sides are of depth at most two.

(ii) linear, noncollapsing, and depth-two systems.

**Proof.** (i) Theorem 10 and Theorem 5 (ii) Theorem 10 and Theorem 7.

In [17], it is shown that $UN^-$ is undecidable for right-ground systems. We can apply the flattening procedure of [20], which preserves the $UN^-$ property to get:

**Theorem 11** $UN^-$ is undecidable for right-ground, right-flat rewrite systems.

## 8 Existence of Normal Form and Normalization

**Theorem 12** Existence of normal form is undecidable for confluent, linear, non-collapsing, var-preserving, constructor-based and depth two systems.

**Proof.** Let $\Sigma$ denote the signature of $R(M)$ constructed in Theorem 7 above. We construct $R'$ and $s'$, an instance of existence of normal form, as follows. Let $T = \{ c \rightarrow c \mid c \in \Sigma \} \cup \{ hx_1, \ldots, x_n \rightarrow hx_1, \ldots, x_n \mid h \in \Sigma \}$, so that all non-variable terms in $\mathbb{T}(\Sigma, V)$ are reducible.

Let $R' = R(M) \cup T \cup \{ h(f, Z, Z) \rightarrow d \}$, where $d$ is a new constant, and let $s' = h(s_0, Z, Z)$. Then, clearly $s'$ has an $R'$ normal form (viz. $d$) iff $s \not\Rightarrow_R h(f, Z, Z)$. Note that $R'$ is var-preserving, non-collapsing, linear, and depth two. It is also confluent since all critical pairs are parallel closed [10] trivially.

**Theorem 13** Reachability $\leq P$ existence of normal form.

**Proof.** Let $R, s, t$ be an instance of reachability. Construct $R'$ and $s'$, an instance of
existence of normal form, as follows. First flatten $t$ using flattening rules [7]. These rules preserve $R$ reachability of terms. Let $c_t$ be the new “name” for $t$. We add all the new names so introduced to $\Sigma$ and the flattening rules to $R$. Let $d$ be a new constant not in $\Sigma$. Let $T = \{c \rightarrow c \mid c \in \Sigma\} \cup \{hx_1, \ldots, x_n \rightarrow hx_1, \ldots, x_n \mid h \in \Sigma\}$.

Let $R' = R \cup T \cup \{c_t \rightarrow d\}$ and $s' = s$.

Then, clearly $s'$ has an $R'$ normal form (viz. $d$) if $s \xrightarrow{R} t$. Recall that $s$ can be assumed ground. We do not assume a fixed signature so this does not affect the normal form property of $s$ since the new constants are $R$-irreducible (because they are new). If we have to introduce new constants to force groundness of $s$, $t$, then we make them part of $\Sigma$ and make them reducible w.r.t. $R'$.

The above reduction preserves left-linearity, right-linearity, flatness, noncollapsing property and is var-preserving. It can also be modified to preserve groundness (see [19]).

**Corollary 8.1** (a) Existence of normal form is undecidable for: (i) flat systems and (ii) left-flat, right-linear, noncollapsing, var-preserving systems in which right-hand sides are of depth at most two.

(b) Normalization is undecidable for: (i) linear, noncollapsing, var-preserving and depth two systems, (ii) flat systems, and (iii) left-flat, right-linear, noncollapsing, var-preserving systems in which right-hand sides are of depth at most two.

(c) Existence of normal form is decidable for: (i) linear, right-shallow and (ii) shallow, right-linear systems.

**Proof.**

(a) (i) Theorem 13 and Theorem 1. (ii) Theorems 13 and 5.

(b) Reachability can be reduced to normalization. The proof is quite similar to that of the reduction to existence of normal forms problem so we describe only the changes. Using the notations of Theorem 13 construct $R''$ an instance of normalization as follows: $R'' = R' \cup S$, where $R'$ is as in Theorem 13 and $S = \{c \rightarrow c_s \mid c \in \Sigma\} \cup \{hx_1, \ldots, x_n \rightarrow c_s \mid h^{(n)} \in \Sigma, n > 0\}$. Recall that $c_s$ is obtained by flattening $s$. If $s$ is flat, we still introduce the constant $c_s$ and the auxiliary rules. However, we will show something stronger, viz., the existence of a reduction between existence of normal form and normalization.

Existence of normal form is a special case of the normalization problem. In case that is not convincing to the reader, we show that there exists a reduction from existence of normal form to the normalization problem as follows. Let $R$, $s$ be an instance of the existence of normal form problem. Flatten $s$ as in [20] and call the resulting rewrite system also $R$. Construct a Turing Machine $N$ that accepts exactly the ground $R$ normal forms in a single final state, say $f$. Now simulate $N$ using a two-counter machine say $M$. Next simulate $M$ using a rewrite system $R(M)$ and let $\Sigma(M)$ denote the alphabet of $R(M)$ (we ensure that $f$ is a constant in $\Sigma(M)$).

Now using the notations of Theorem 13 construct $R'$ an instance of normalization as follows: $R' = R \cup S \cup R(M)$, where $S = \{c \rightarrow c_s \mid c \in \Sigma \cup \Sigma(M) - \{f\}\} \cup \{hx_1, \ldots, x_n \rightarrow c_s \mid h^{(n)} \in \Sigma, n > 0\}$. The $S$ rules ensure that $f$ is the only ground normal form of $R'$ and that every non-variable term, except $f$, reduces to $c_s$, the new name for $s$. Then, every term has an $R'$ normal form iff $s$ has an $R$ normal form. Notice the reduction preserves linearity and depth two. It can be modified to preserve flatness as follows. Instead of simulating $N$ by a two-counter machine
construct a PCP instance and then simulate the PCP instance via a rewrite system as in [11]. Details of this construction are deferred to the full version of this paper.

(c) Both parts follow from the reduction of part (b) to normalization and decidability results for normalization in [5].

9 Uniqueness of Normal Forms

As a corollary of Theorem 8 and the reduction from joinability for confluent systems to uniqueness of normal forms [19], we have the following result:

**Theorem 14** $UN^{-}$ is undecidable for linear, noncollapsing, var-preserving, and depth-two rewrite systems.

**Proof.** The reduction of [19] preserves linearity, noncollapsing, and var-preserving properties and can be modified using the flattening procedure of [20] on the (ground) terms $s$ and $t$ to preserve the depth. In [17], it is shown that $UN^{-}$ is undecidable for right-ground systems. We can apply the flattening procedure of [20], which preserves the $UN^{-}$ property to get:

**Theorem 15** $UN^{-}$ is undecidable for right-ground, right-flat rewrite systems.

10 Conclusions and Future Work

This paper shows that for several fundamental properties of rewrite systems the class of linear, depth two systems is a close boundary as far as restrictions on occurrences of variables and their depth is concerned. Further, joinability is undecidable for linear, left-flat systems in which the right-hand sides are of depth at most two. This is a sharper lower bound than linear, depth two systems. Reachability, confluence and unique normalization, existence of normal form and normalization are all undecidable for left-flat, right-linear systems in which right-hand sides are of depth at most two as well. This represents an exchange of left-linearity with left-flatness. Existence of normal form and normalization are also undecidable for flat systems. Thus, these results together with [19], [14], [20], and [5] leave the following four out of 15 subclasses $^6$ of systems as far as the last two properties and reachability are concerned: left-linear, right-shallow; left-linear, left-shallow; right-linear, right-shallow; and linear, left-shallow for which the status of these two properties is still open as far as I know. The status of confluence is open for two more subclasses, viz., linear, right-shallow systems and left-linear, shallow systems to my knowledge. The status of joinability is now open for only three out of 15 subclasses. The situation for $UN^{-}$ and $UN^{-}$ is the least satisfactory, which are open for seven and 10 subclasses respectively to my knowledge. Of course, this classification is with respect to occurrences of variables and depth of variables, which is the scope of this paper. Other properties may be brought to bear (e.g., see [12]) opening new vistas.

---

References


11 Appendix

Lemma 11.1 ([17]) WP ≤p WP for non-collapsing equations.

Proof. Let E, s, t be the instance of WP. Construct E’, s’, t’, where E’ is non-collapsing as follows. Set s’ = s and t’ = t. To construct E’, first we discard equations of the form x = x from E (if they exist) without any problem. Next, each
equation of the form \( l = x \) in \( E \), where \( x \) is a variable and \( l \) not, is replaced in \( E' \) by the set of equations \( \{ l\sigma \rightarrow fx_1, \ldots, x_n \mid f^{(n)} \in \Sigma \} \) and \( \sigma = \{ x \mapsto fx_1, \ldots, x_n \} \). Here \( x_1, \ldots, x_n \) are new variables not appearing in \( l \). Equations of the form \( x = r \), where \( x \) is a variable and \( r \) not, are handled in the same way. It is easy to verify that the reduction can be carried out in polynomial time.

The correctness of the reduction now follows from the following proposition and the fact that we may assume \( s \) and \( t \) are ground.

**Proposition 11.2** Given \( E, s \) and \( t \), with \( s =_E t \), then there is an equational proof \( s =_E t \) such that every equation instance used in this proof contains only function symbols and constants from \( \Sigma \). Moreover, if \( s \) and \( t \) are ground, then there exists an equational proof with the additional property that every equation instance used in this proof is ground.

**Proof.** By induction on the length of the proof \( s =_E t \) using the fact that any new symbols must be in the substitution part of the equation used and so may be replaced with symbols that appear in terms of \( E, s \) or \( t \). For the groundness part, observe that we can define a substitution \( \sigma \) that maps variable \( x \) to any constant in \( \Sigma \) and by stability of equational proofs under substitutions we get that \( s\sigma =_E t\sigma \), but since \( s \) and \( t \) are ground so \( s = ss\sigma =_E t\sigma = t \). \( \square \)


To make the paper self-contained we include the reduction of PCP to \( UN^- \) given in [5]. Let PCP = \( \{(u_i, v_i) \mid 1 \leq i \leq n\} \) be a Post Correspondence Problem over \( \{a, b\} \). The TRS defined by [5] is: \( pcp(x, y) \rightarrow eq(x, y), pcp(x, y) \rightarrow pair(x, y), eq(a(x), a(y)) \rightarrow eq(x, y), eq(b(x), b(y)) \rightarrow eq(x, y), eq(c, c) \rightarrow nf1, pair(u_i(x), v_i(y)) \rightarrow pair(x, y), 1 \leq i \leq n, pair(u_i(c), v_i(c)) \rightarrow nf2, 1 \leq i \leq n, a(x) \rightarrow loop, b(x) \rightarrow loop, c \rightarrow loop, pair(x, y) \rightarrow loop, eq(x, y) \rightarrow loop, pcp(x, y) \rightarrow loop, \) and \( loop \rightarrow loop. \)
Order-sorted Equational Unification Revisited

Joe Hendrix and José Meseguer

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
Email: {jhendrix,meseguer}@cs.uiuc.edu

Abstract
This paper presents a rule-based algorithm for performing order-sorted $E$-unification using an unsorted $E$-unification decision procedure under assumptions about $E$ that are commonly satisfied in practice. We have implemented this algorithm in Maude for use with the Maude-NRL protocol analyzer and have used CiME for unsorted $E$-unification for $E$ any set of AC and ACU axioms. In many examples of interest, using order-sorted unification over unsorted unification is able to reduce the total number of unifiers considered, and dramatically improve the performance of the Maude-NRL tool.

Keywords: Order-sorted unification, rule-based programming.

1 Introduction

Unification is a fundamental operation in many applications. For example, in solving reachability problems using narrowing, the unification procedure is called many times to unify terms representing reachable states against the left-hand sides of rewrite rules. This process is computationally expensive and often generates a large number of different terms — many of which may represent states that do not correspond to legal states. In order to avoid this problem tools such as the Maude-NRL protocol analyzer [4,5] use order-sorted algebras and rely on the sorts to only consider well-formed terms.

We present an algorithm which can use a procedure for unsorted $E$-unification to perform order-sorted $E$-unification under conditions general enough to cover many practical applications. This algorithm solves a key challenge faced by the Maude-NRL protocol analyzer — most existing unification tools only support unsorted unification and ignore the sort information. Since equational unification procedures are often quite complex, it requires significantly less work to use an existing unification tool rather than writing an order-sorted equational unification procedure from scratch.

1 The research has been supported in part by ONR grant N00014-02-1-0715 and NSF grant CNS-07-16638.

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs
The order-sorted unification algorithm we present in this work can be naturally described by a terminating and confluent set of rewrite rules which compute order-sorted unifiers $\theta_1, \ldots, \theta_n$ for each unsorted unifier $\bar{\theta}$ returned by the unsorted unification procedure. We have implemented the algorithm in Maude, and have used CiME as the unsorted equational unification procedure. Our experimental results so far have shown that, although technically there may be many order-sorted unifiers for each unsorted unifier, this is rarely the case in practice. In fact, in practice there are usually fewer order-sorted unifiers than unsorted unifiers, and the use of order-sorted unification is essential for both correctness and performance, that is, so that the terms explored are always well-formed terms, and to ensure that the Maude-NRL analyzer is capable of handling real problems.

Our idea is not new, and was presented in [11] and more recently without a proof of correctness in [4]. However, after implementing these ideas in the Maude-NRL protocol analyzer, we felt that a new paper presenting the basic ideas was in order for several reasons:

- Our experience with the Maude-NRL protocol analyzer so far has suggested that for theories with AC operators, for practical protocol verification tools based on narrowing it is essential to use the sort information during unification. However, most existing unification procedures only perform unsorted $E$-unification and do not support sorts and subsorts. By using the techniques described in this work, one can obtain an order-sorted $E$-unification procedure from an unsorted one with very little effort for many equational theories.

- The algorithm in [11] was buried in a function’s definition appearing in the proof of Theorem 34 in [11]. In this paper, we present a simple rule-based algorithm which is almost directly implementable in Maude. The algorithm only consists of three confluent and terminating rewrite rules, and it should be easily possible to compose these rules with inference rewrite steps in a modular way in other reasoning tools using unification.

- Perhaps most important from a technical perspective, the correctness results in [11] imposed unnecessarily strong technical conditions which excluded the majority of $E$-unification problems when $E$ contains collapsing equations like idempotence $x + x = x$ and identity $x + 0 = x$. As identity was important for the Maude-NRL protocol analyzer and idempotence is a common axiom in many $E$-unification algorithms, in this paper we prove the correctness results under weaker assumptions about the equational theory and some technical assumptions about the unification engine. The assumptions about the unification engine should be satisfied in practice. Additionally, we show specifically how the algorithm can be used in Maude for equational theories with any combination of free, commutative, AC, and ACU symbols.

This paper is organized as follows. In Section 2, we review basic definitions of order-sorted algebra and unification. In Section 3, we present our algorithm to compute order-sorted unifiers from unsorted unifiers. In Section 4, we illustrate how it can be used for AC and ACU order-sorted unification in Maude and, in Section 5 we prove its correctness. Finally, in Section 6, we discuss related work and suggest directions for future research.
2 Preliminaries

2.1 Order Sorted Algebra

An order-sorted signature \( \Sigma = (S, F, \leq) \) consists of a set of sorts \( S \), a family of operators \( F = \{ F_{w,s} \}_{(w,s) \in S^* \times S} \), and a partial order \( \leq \subseteq S \times S \) called the subsort ordering. We let \( X = \{ X_s \}_{s \in S} \) denote a fixed \( S \)-sorted family of infinite sets of variables that are both pairwise disjoint for different sorts, i.e., \( X_s \cap X_{s'} = \emptyset \) for distinct \( s, s' \in S \), and disjoint from the operators \( F \). As a notational convenience, we write \( x_s \) to denote that \( x \in X_s \) when the variables \( X \) are clear from the context. The \( \Sigma \)-terms with variables \( X \) are members of the family \( T_\Sigma(X) = \{ T_\Sigma(X)_s \}_{s \in S} \), where \( T_\Sigma,s(X) \) denotes the \( \Sigma \)-terms with any sort \( s' \leq s \). An order-sorted theory \( \mathcal{E} = (\Sigma, E) \) consists of an order-sorted signature \( \Sigma \) and a finite set \( E \) of equations \( l = r \) where \( l, r \in T_\Sigma,s(X) \) for some sort \( s \in S \). An order-sorted substitution is a function \( \theta : Y \rightarrow T_\Sigma(X) \) with \( Y \) a finite subset of \( X \), and for each variable \( x_s \in Y, x_s \theta \in T_\Sigma(X)_s \). We let \( \text{rvars}(\theta) \) denote the variables occurring in a terms in the codomain of \( \theta \), i.e., \( \text{rvars}(\theta) = \bigcup_{x \in Y} \text{vars}(x \theta) \). Given substitutions \( \theta_1, \theta_2 : Y \rightarrow T_\Sigma(X) \), we write \( \theta_1 =_\Sigma \theta_2 \) if \( x \theta_1 =_\Sigma x \theta_2 \) for all \( x \in Y \), and we write \( \theta_1 \geq_\Sigma \theta_2 \) if there is a substitution \( \psi : \text{rvars}(\theta_1) \rightarrow T_\Sigma(X) \) such that \( \theta_1 \psi =_\Sigma \theta_2 \). For an equational theory \( \mathcal{E} = (\Sigma, E) \) with \( \Sigma = (S, F, \leq) \), there is an underlying unsorted theory \( \overline{\mathcal{E}} = (\overline{\Sigma}, E) \) over variables \( \overline{X} = \bigcup_{s \in S} X_s \) such that \( \overline{\Sigma} \) is a ranked alphabet containing an operator \( f \) with arity \( n \) if there is an operator \( f \in F_{s_1 \ldots s_n,s} \) for some sorts \( s_1, \ldots, s_n, s \in S \). Observe that a \( \Sigma \)-equation \( l = r \in E \) is always a \( \overline{\Sigma} \)-equation.

2.2 Order-sorted Equational Unification

For a fixed order-sorted theory \( \mathcal{E} = (\Sigma, E) \) with \( \Sigma = (S, F, \leq) \), we define an order-sorted unification problem to be a finite conjunctive set \( \Gamma \) of \( \Sigma \)-equations \( t = u \) where \( t \) and \( u \) are terms in \( T_\Sigma(X) \) whose sorts belong to the same connected component in \( (S, \leq) \). A \( \mathcal{E} \)-unifier for \( \Gamma \) is an order-sorted substitution \( \theta : \text{vars}(\Gamma) \rightarrow T_\Sigma(X) \) such that \( t \theta =_\Sigma u \theta \) for each equation \( t = u \in \Gamma \). We denote the set of \( \mathcal{E} \)-unifiers for \( \Gamma \) by \( \text{Un}_\mathcal{E}(\Gamma) \), and we let \( \text{Un}_\mathcal{E}(\Gamma) = \bigcup_{\theta \in \text{Un}_\mathcal{E}(\Gamma)} \theta \) denote the syntactic unifiers for \( \Gamma \), i.e., \( \text{Un}_\mathcal{E}(\Gamma) = \bigcup_{\theta \in \text{Un}_\mathcal{E}(\Gamma)} \theta \). A set \( S \subseteq \text{Un}_\mathcal{E}(\Gamma) \) of \( \mathcal{E} \)-unifiers of \( \Gamma \) is complete if for all unifiers \( \psi \in \text{Un}_\mathcal{E}(\Gamma) \), there is a unifier \( \theta \in S \) such that \( \theta \geq_\Sigma \psi \). A set of \( \mathcal{E} \)-unifiers \( S \) is most-general if for distinct substitutions \( \theta_1, \theta_2 \in S, \theta_1 \not\geq_\Sigma \theta_2 \). A given theory \( \mathcal{E} \) has a finitary unification problem if there is a complete finite set of \( \mathcal{E} \)-unifiers \( S \) for each unification problem \( \Gamma \).

3 Order-sorted Unification

Our main goal in this work is to develop a clear rule-based algorithm for solving order-sorted \( \mathcal{E} \)-unification problems using an unsorted \( \overline{\mathcal{E}} \)-unification procedure. In order to show that the rule-based algorithm returns a complete set of most-general unifiers, there are some technical requirements placed on the order-sorted theory.
\( \mathcal{E} \) as well as on the most-general unifiers \( \mathcal{U} \) returned by the unsorted \( \mathcal{E} \)-unification procedure. The basic techniques behind our algorithm were described in [11]. However the correctness shown in [11] imposed conditions that are too strong when the theory \( \mathcal{E} \) contains collapsing equations like identity or idempotence axioms.

Our approach to find suitable requirements is then to relax the requirements on \( \mathcal{E} \) while making requirements on the unsorted unification procedure in relation to the theory \( \mathcal{E} \). At first this appears to be less general than the approach in [11], since that work did not make any assumption about the unsorted unification procedure. However, as we will discuss later, the theories we are interested in are such that every practical unification procedure will satisfy the requirements. Most importantly for our work, this includes theories with identity axioms.

In this section, we assume the following conditions on the order-sorted theory \( \mathcal{E} = (\Sigma, \mathcal{E}) \) and the unsorted unification procedure for \( \mathcal{E} \).

(i) \( \Sigma \) is preregular [7], that is every term \( t \in T_\Sigma(X) \) has a least sort \( \text{ls}(t) \in S \).

(ii) \( \mathcal{E} \) is sort-independent which means that for all order-sorted terms \( t, u \in T_\Sigma(X) \),

\[
  t =_\mathcal{E} u \Rightarrow t =_\mathcal{E} u.
\]

(iii) For each unification problem \( \Gamma \), the unsorted unification procedure generates a complete finite set of most-general sort-preserving unifiers \( \mathcal{U} \) for \( \Gamma \). We can split the process of solving an order-sorted unification problem \( \Gamma = t_1 =_\mathcal{E} u_1 \land \cdots \land t_n =_\mathcal{E} u_n \) into two phases: an unsorted unification phase and a sort propagation phase.

Unsorted Unification. First, we call the unsorted \( \mathcal{E} \)-unification procedure on the unsorted \( \mathcal{E} \)-unification problem \( \Gamma = t_1 =_\mathcal{E} u_1 \land \cdots \land t_n =_\mathcal{E} u_n \) to obtain a finite complete set of most-general sort-preserving unifiers \( \mathcal{U} \) for \( \Gamma \).

Sort Propagation. In the second phase, for each unsorted unifier \( \overline{\theta} \in \mathcal{U} \), we use the membership propagation algorithm described below to generate a set of variable renamings. In this context, a variable renaming is an injective function \( \rho : \text{rvars}(\overline{\theta}) \rightarrow X \). For each variable renaming \( \rho \) generated for an unsorted unifier \( \overline{\theta} \in \mathcal{U} \), our procedure returns \( \overline{\theta}\rho \) as one element in the complete set of most-general unifiers.

The membership propagation algorithm is described by a set of rules that maintain a disjunctive set \( D \) of membership constraints. Each membership constraint \( M \in D \) is a conjunctive formula of the form \( M = t_1 : s_1 \land \cdots \land t_n : s_n \), and \( D \) is a finite set \( D = \{ M_1, \ldots, M_p \} \) of membership constraints. A membership constraint \( M \) captures constraints for an unsorted unifier to be an formed order-sorted unifier. For each unsorted unifier \( \overline{\theta} \in \mathcal{U} \), we initially generate a singleton set \( D(\overline{\theta}) \) reflecting the sort constraints on the variables appearing in the original unification
Given an initial set of membership constraints $D$, we freely apply the rules below to obtain a final set of constraints $D^*$.

**Intersection**  
$$ \{ t : s_1 \land t : s_2 \land M \} \rightarrow \bigcup_{s \in \text{glb}_\Sigma(s_1, s_2)} \{ t : s \land M \} $$

**Propagation**  
$$ \{ f(t_1, \ldots, t_n) : s \land M \} \rightarrow \bigcup_{s_1, \ldots, s_n \in \text{arg}_\Sigma(f, s, n)} \{ t_1 : s_1 \land \cdots \land t_n : s_n \land M \} $$

**Subsumption**  
$$ \{ M_1, M_2 \} \rightarrow \{ M_1 \} \text{ if } M_1 \geq M_2 $$

where  
$$ \text{glb}_\Sigma(s_1, s_2) = \sup_{\leq}\{ s \in S \mid s \leq s_1 \land s \leq s_2 \} , $$

$$ \text{arg}_\Sigma(f, s, n) = \sup_{\leq}\{ w \in S^n \mid (\exists s' \in S) f \in F_w, s' \land s' \leq s \} , $$

and  
$$ M_1 \geq M_2 \iff (\forall t : s \in M_1)(\exists s' \in S) s' \leq s \land t : s' \in M_2 . $$

Fig. 1. Sort Propagation Algorithm

problem $\Gamma$.  
$$ D(\overline{\theta}) = \{ \bigwedge_{x \in \text{vars}(\Gamma)} x_\overline{\theta} : s \} . $$

We then apply the three rewrite rules in Fig. 1 to $D(\overline{\theta})$ until termination. The **Intersection** rule exploits the preregularity assumption to simplify multiple membership constraints $t : s_1$ and $t : s_2$ on the same term $t$. The **Propagation** rule simplifies constraints on terms $f(t_1, \ldots, t_n) : s$ to the smaller terms $t_1, \ldots, t_n$. Finally, the **Subsumption** rule is used to eliminate membership constraints that are subsumed by other more-general membership constraints. We let $D^*$ denote the unique normal form obtained by rewriting $D$ until completion.

Upon termination of the rules each membership constraint $M \in D^*$ will have the form $x_1 : s_1 \land \cdots \land x_n : s_n$ where $x_i \neq x_j$ if $i \neq j$. We call membership constraints with this form reduced. A reduced membership constraint can be viewed as a function $\text{sort}_M : \text{rvars}(\overline{\theta}) \rightarrow S$ that maps each variable $x_i \in \text{rvars}(\overline{\theta})$ to the sort $s_i \in S$. Furthermore, for each reduced membership constraint $M$, we let $\rho_M : \text{rvars}(\overline{\theta}) \rightarrow X$ be a variable renaming which maps each variable $x \in \text{rvars}(\overline{\theta})$ to a fresh variable $x\rho_M$ with sort $\text{sort}_M(x)$.

For the set of unsorted sort-preserving unifiers $\overline{U} \subseteq \text{Un}_{\Gamma}(\Gamma)$, we define the set  
$$ \text{OS}(\overline{U}) = \{ \overline{\theta}\rho_M \mid \overline{\theta} \in \overline{U} \land M \in D(\overline{\theta})^* \} . $$

As an example, consider the unification problem $x_{\mathbb{N}\mathbb{N}} = y_{\mathbb{N}} + z_{\mathbb{N}}$ over an order-sorted theory $\mathcal{E} = (F, E)$ where $+$ contains the following operator declarations:

$$ + : \mathbb{N} \rightarrow \mathbb{N} \quad + : \mathbb{N}\mathbb{N} \rightarrow \mathbb{N}\mathbb{N} \quad + : \mathbb{N}\mathbb{N} \mathbb{N} \rightarrow \mathbb{N}\mathbb{N} $$

where the declaration $f : s_1 s_2 \rightarrow s$ means that $f \in F_{s_1 s_2, s}$. In this case, the unsorted
unification engine can return a single unifier

$$\theta = (x \mapsto y + z, y \mapsto y, z \mapsto z).$$

However, $\theta$ is not an order-sorted unifier, because $y + z$ does not have sort $\mathbb{NzNat}$. We pass $\theta$ to the sort-propagation algorithm, which generates the initial set of membership constraints

$$D(\theta) = \{ y + z : \mathbb{NzNat} \land y : \mathbb{Nat} \land z : \mathbb{Nat} \}.$$ 

For this simple example, a single application of Propagation yields the membership constraints:

$$D(\theta)' = \{ (y : \mathbb{Nat} \land z : \mathbb{NzNat} \land y : \mathbb{Nat} \land z : \mathbb{Nat}), (y : \mathbb{NzNat} \land z : \mathbb{Nat} \land y : \mathbb{Nat} \land z : \mathbb{Nat}) \}.$$ 

From $D(\theta)'$, we only need to apply Intersection several times to yield the final set of membership constraints:

$$D(\theta)^* = \{ (y : \mathbb{Nat} \land z : \mathbb{NzNat}), (y : \mathbb{NzNat} \land z : \mathbb{Nat}) \}.$$ 

From $D(\theta)^*$, we can extract two variables renamings. When applied to the initial unsorted unifier $\theta$, this yields the final complete set of order-sorted unifiers:

$$\text{OS}(\theta) = \{ (x \mapsto u_\mathbb{Nat} + v_\mathbb{NzNat}, y \mapsto u_\mathbb{Nat}, z \mapsto v_\mathbb{NzNat}), (x \mapsto u_\mathbb{NzNat} + v_\mathbb{Nat}, y \mapsto u_\mathbb{NzNat}, z \mapsto v_\mathbb{Nat}) \}.$$ 

We prove the following result in Section 5 to show that our algorithm is correct,

**Theorem** Let $E = (\Sigma, E)$ denote an order-sorted theory satisfying requirements (i) and (ii) above, then given a unification problem $\Gamma$ with a complete set of most-general sort-preserving unsorted unifiers $\overline{U}$, $\text{OS}(\overline{U})$ is a complete set of most-general order-sorted unifiers for $\Gamma$.

### 4 Order-sorted AC + ACU unification

As the requirements on $E$ and $\overline{U}$ seem rather technical, to give the reader a more intuitive feel for them, we show how the requirements are satisfied by many order-sorted equational theories specified as Maude modules having free, commutative, AC, and ACU symbols. Essentially, each such Maude module can be viewed as an order-sorted theory $E = (\Sigma, E)$ with $\Sigma = (S, F, \leq)$ such that:

(a) Each equivalence class $[s] \in S$/ $\equiv_{\leq}$ contains a maximal element $k_s$ called the kind of $s$ where $\equiv_{\leq}$ denotes the equivalence relation generated by $\leq$. Moreover, for each operator declaration $f \in F_{s_1 \ldots s_n, s}$, there is also a declaration $f \in F_{k_{s_1} \ldots k_{s_n}, k_s}$.  

21
(b) $E$ contains axioms of the following forms:

\[
\begin{align*}
f(f(x, y), z) &= f(x, f(y, z)) & \text{associativity} \\
f(x, y) &= f(y, x) & \text{commutativity} \\
f(c, x) &= x & \text{unit}
\end{align*}
\]

where the sorts of $x, y, z$ are maximal sorts, that is, sorts of the form $k_s$ for some $s \in S$, and for each binary symbol $f \in F$, either $f$ does not appear in $E$, or $E$ contains commutativity $(C)$, associativity and commutativity $(AC)$, or associativity, commutativity, and unit $(ACU)$ axioms for $f$.

(c) $\Sigma$ is preregular.

(d) Each axiom $l = r \in E$ is sort-preserving, that is for each variable mapping $\rho : X \rightarrow X$, $\text{ls}(l\rho) = \text{ls}(r\rho)$.

The assumptions (a)–(d) are quite reasonable for order-sorted Maude specifications with free, commutative, AC, and ACU operators. Maude will automatically introduce additional top-most sorts $k_s$, and requires that associativity, commutativity, and unit axioms satisfy the requirements in (b). Maude does allow associative symbols that are not commutative, however unification for such theories may be infinitary [12] and is not considered here. The preregularity requirement is checked automatically by Maude when the module is entered. The sort-preservation requirement (d) is essential as the sort-propagation algorithm described in the previous section operates syntactically on terms, and disregards the possibility that applying an equation may change the sort of a term. It is guaranteed by a three-pronged approach:

- For each associativity axiom $f(f(x, y), z) = f(x, f(y, z))$, Maude checks that it is sort preserving by considering possible variable mappings.
- For each commutativity axiom $f(x, y) = f(y, x)$ and each declaration $f : s_1 s_2 \rightarrow s$, Maude completes the theory by adding a declaration $f : s_2 s_1 \rightarrow s$.
- For each pair of identity axioms $f(x, c) = x$ and $f(c, x) = x$, our unification procedure completes the theory by introducing a fresh sort $s_c$ together with: (1) an operator declaration $c : s \rightarrow s_c$, (2) a subset declaration $s_c < \text{ls}(t)$; and (3) for each sort $s \in [\text{ls}(t)]$, operator declarations $f : s s_c \rightarrow s$ and $f : s_c s \rightarrow s$.

We now focus on the relationship between the assumptions (a)–(d) and the earlier requirements (i)–(iii). The first preregularity requirement follows from the preregularity assumption. The sort-independence requirements follows form the assumptions (a) and (b).

**Theorem 4.1** If $\mathcal{E} = (\Sigma, E)$ is an order-sorted theory satisfying assumptions (a) and (b) above, then $\mathcal{E}$ is sort-independent.

**Proof.** Showing that $\mathcal{E}$ is sort-independent requires showing that for all $t, u \in T_{\Sigma}(X)$, $t \models_\mathcal{E} u$ implies $t \models u$.

We first partition $E$ into disjoint sets $E = R \cup A_x$ where $A_x$ contains the associativity and commutativity equations in $E$ and the identity equations $f(c, x) = x$ in $E$ are interpreted as rules $f(c, x) \rightarrow x$ in $R$. It is not difficult to see that the rules $R$ modulo $A_x$ are terminating and confluent, and therefore $t \models_\mathcal{E} u$ iff
$t \downarrow_{R/A_x} = A_x \downarrow_{R/A_x}$.

As $A_x$ only contains associativity and commutativity axioms, if $t \in T\Sigma(X)_{k_s}$ for some maximal sort $k_s$ and $t = A_x v$, then it easily follows that $v \in T\Sigma(X)_{k_s}$ and $t = A_x v$ by the requirement (a). It also easily follows that if $t \rightarrow_{R/A_x}^* v$, then $v \in T\Sigma(X)_{k_s}$ and $t \rightarrow_{R/A_x}^* v$. From this, we can conclude that

$$t \downarrow_{R/A_x} = A_x \downarrow_{R/A_x} \implies t \downarrow_{R/A_x} = A_x \downarrow_{R/A_x}.$$ 

It easily follows that $t = \mathcal{E} u$, and thus $\mathcal{E}$ is sort-independent. \hfill \Box

In general, the requirement that the unification procedure is sort-preserving does not follow from the assumptions given above. For an example, consider the theory $\mathcal{E}$ with two unrelated top-most sorts $Nat$ and $Cns$ where $Nat$ contains the ACU symbol $+$ with the identity element $0$, and $Cns$ contains the constant $a$. Given the unification problem $x_{Cns} = a$, it would be permissible for the unsorted unification procedure to return the unifiers

$$\mathcal{U} = \{ (x \mapsto a + 0) \}.$$ 

This is a complete set of unsorted unifiers due to the identity axiom, but unsuitable for our sort propagation algorithm as $a + 0$ is not a legal term. This counterexample illustrates why the earlier work [11] imposed significant restrictions on theories with collapsing equations like identity.

These stronger restrictions appear unnecessary in practice — in our experience, the procedure will not introduce extra symbols, and in this case return the simpler unifier $x \mapsto a$. The reason that unsorted AC and ACU unification procedures satisfy this assumption is that the unifiers are computed from the terms appearing in equations $l = r \in \Gamma$. When those subterms are well-typed with the same top-most sort $k$, substitutions generated by the unsorted unification procedure should be well-typed as well. Provided that the sorts of fresh variables in the right hand side of a variable are given the appropriate top sort $k$, due to our assumption (a), we have found it is safe to assume the following:

(e) For each unifier $\theta$ in the set of unifiers $\mathcal{U}$ returned by the unsorted unification procedure for the order-sorted unification problem $\Gamma$, and for each variable $x_s \in \text{vars}(\Gamma)$, $x_s \theta \in T\Sigma(X)_{k_s}$.

To validate these ideas and test this assumption, we have extended an alpha version of Maude so that it may communicate with CiME [2,3] by passing unsorted unification problems as strings, and parsing the unsorted unifiers returned from CiME back into Maude terms. As an additional safeguard, the parsing process checks the substitutions returned by CiME to verify that assumption (e) is satisfied. These checks have always been satisfied in our experience using the procedure so far. We then apply the sort propagation algorithm described in the previous section to generate order-sorted $\mathcal{E}$-unifiers. The order-sorted unification procedure is used to analyze cryptographic protocols with algebraic properties of associativity and commutativity using the Maude-NRL protocol analyzer [4].
5 Correctness Proof

The goal of this section is to show the correctness of our approach to order-sorted equational unification. Before we can show this, we need several intermediate lemmas. The first lemma shows how preregularity is used.

Lemma 5.1 If \( \Sigma = (S, F, \leq) \) is preregular, then for all sorts \( s_1, s_2 \in S \) and terms \( t \in T_{\Sigma}(X) \),

\[ t \in T_{\Sigma}(X)_{s_1} \cap T_{\Sigma}(X)_{s_2} \iff (\exists s \in \text{glb}(s_1, s_2)) t \in T_{\Sigma}(X)_s \]

where \( \text{glb}(s_1, s_2) = \sup_{\leq}(\{ s \in S \mid s \leq s_1 \land s \leq s_2 \}) \).

Proof. If there is a sort \( s \in \text{glb}(s_1, s_2) \) such that \( t \in T_{\Sigma}(X)_s \), then \( t \in T_{\Sigma}(X)_{s_1} \cap T_{\Sigma}(X)_{s_2} \) as \( s \leq s_1 \) and \( s \leq s_2 \). We still must show that \( t \in T_{\Sigma}(X)_{s_1} \cap T_{\Sigma}(X)_{s_2} \) implies that there is a sort \( s \in \text{glb}(s_1, s_2) \) such that \( t \in T_{\Sigma}(X)_s \). However, this follows immediately as \( t \) must have a least sort \( s' \in S \). It must be the case that \( s' \leq s_1 \) and \( s' \leq s_2 \). Therefore, there is an \( s \in \text{glb}(s_1, s_2) \) such that \( s' \leq s \). As \( T_{\Sigma}(X)_{s'} \subseteq T_{\Sigma}(X)_s \), it follows that \( t \in T_{\Sigma}(X)_s \). \( \square \)

Lemma 5.2 For all terms \( f(t_1, \ldots, t_n) \in T_{\Sigma}(X) \) and sorts \( s \in S \),

\[ f(t_1, \ldots, t_n) \in T_{\Sigma}(X)_s \iff (\exists s_1 \ldots s_n \in \text{arg}(f, s, n)) t_1 \in T_{\Sigma}(X)_{s_1} \land \cdots \land t_n \in T_{\Sigma}(X)_{s_n} \]

where \( \text{arg}(f, s, n) = \sup_{\leq}(\{ w \in S^n \mid (\exists s' \in S) f \in F_{w, s', s'} \land s' \leq s \}) \).

Proof. If there are sorts \( s_1 \ldots s_n \in \text{arg}(f, s, n) \) such that \( t_i \in T_{\Sigma}(X)_{s_i} \) for \( i \in [1, n] \), then there must be a sort \( s' \leq s \) such that \( f \in F_{s_1 \ldots s_n, s'} \). It follows that \( f(t_1, \ldots, t_n) \in T_{\Sigma}(X)_{s'} \), and thus \( f(t_1, \ldots, t_n) \in T_{\Sigma}(X)_s \).

On the other hand, if \( f(t_1, \ldots, t_n) \in T_{\Sigma}(X)_s \), then there is some \( s' \leq s \) such that \( f \in F_{s_1 \ldots s_n, s'} \) and \( t_i \in T_{\Sigma}(X)_{s_i'} \) for \( i \in [1, n] \). It follows that there are sorts \( s_1 \ldots s_n \in \text{arg}(f, s, n) \) such that \( s_i' \leq s_i \) for \( i \in [1, n] \). Consequently, \( t_i \in T_{\Sigma}(X)_{s_i} \) for \( i \in [1, n] \). \( \square \)

For a membership constraint \( M \), we define the unifiers for \( M \), denoted \( \text{Un}_{\Sigma}(M) \) to be the set of unsorted substitutions \( \theta : \mathcal{X} \to T_{\Sigma}(\mathcal{X}) \) such that for each membership \( t : s \in M \), \( t \theta \in T_{\Sigma}(X)_s \).

Lemma 5.3 For each order-sorted signature \( \Sigma = (S, F, \leq) \) and pair of membership constraints \( M_1 \) and \( M_2 \),

\[ M_1 \geq M_2 \implies \text{Un}_{\Sigma}(M_1) \supseteq \text{Un}_{\Sigma}(M_2). \]

Proof. To show that \( \text{Un}_{\Sigma}(M_1) \supseteq \text{Un}_{\Sigma}(M_2) \), we must show for each substitution \( \theta \in \text{Un}_{\Sigma}(M_2) \) and membership \( t : s \in M_1 \), we have \( t \theta \in T_{\Sigma}(X)_s \). However, since \( M_1 \geq M_2 \), we know that for each \( t : s \in M_1 \), there is a membership \( t : s' \in M_2 \) such that \( s' \leq s \). By definition \( t \theta \in T_{\Sigma}(X)_{s'} \), and therefore \( t \theta \in T_{\Sigma}(X)_s \). \( \square \)

When the membership constraints \( M_1 \) and \( M_2 \) are reduced, the previous implication holds in the other direction.
Lemma 5.4 For each order-sorted signature $\Sigma = (S,F,\leq)$ and pair of reduced membership constraints $M_1$ and $M_2$ such that $\text{vars}(M_1) = \text{vars}(M_2)$,

$$\text{Un}_\Sigma(M_1) \supseteq \text{Un}_\Sigma(M_2) \implies M_1 \geq M_2$$

Proof. Since both $M_1$ and $M_2$ are reduced and $\text{vars}(M_1) = \text{vars}(M_2)$, to show that $M_1 \geq M_2$, it is sufficient to show that for each $x \in \text{vars}(M_1)$, $\text{sort}_{M_1}(x) \geq \text{sort}_{M_2}(x)$. Since $M_2$ is reduced, there is a substitution $\rho_{M_2} : \text{vars}(M_2) \to T_{\Sigma}(X)$ which maps each variable $x \in \text{vars}(M_2)$ to the a fresh variable $x'$ with sort $\text{sort}_{M_2}(x) \in S$. Clearly $\rho_{M_2} \in \text{Un}_\Sigma(M_2)$, and so $\rho_{M_2} \in \text{Un}_\Sigma(M_1)$ by assumption. It follows that for each $x \in \text{vars}(M_2)$ that $\text{sort}_{M_1}(x) \geq \text{sort}_{M_2}(x)$ since $x\rho_{M_2}$ is a variable with sort $\text{sort}_{M_2}(x)$ and $x\rho_{M_2} \in \text{Un}_\Sigma(M_2)$. \hfill $\Box$

For a disjunctive set of membership constraints $D$, we let $\text{Un}_\Sigma(D)$ denote the set of unsorted substitutions that are unifiers for a set of membership constraints $M \in D$, i.e.,

$$\text{Un}_\Sigma(D) = \bigcup_{M \in D} \text{Un}_\Sigma(M).$$

The key correctness property of the inference rules in Fig. 1 is captured by the following lemma.

Lemma 5.5 For an preregular order-sorted signature $\Sigma$, if $D_1 \rightarrow^* D_2$ using the inference rules in Fig. 1, then $\text{Un}_\Sigma(D_1) = \text{Un}_\Sigma(D_2)$.

Proof. To show this it is enough to show the single step case that $D_1 \rightarrow D_2$ implies $\text{Un}_\Sigma(D_1) = \text{Un}_\Sigma(D_2)$. The full lemma follows easily by induction on the number of rules used to show $D_1 \rightarrow^* D_2$. To show the single step case, we must consider three separate cases, one for each of the inference rules in Fig. 1.

- If Intersection is used, we know that $D_1$ and $D_2$ have the forms $D_1 = \{ t : s_1 \land t : s_2 \land M \} \cup D$ and $D_2 = \bigcup_{s \in \text{glb}_\Sigma(s_1, s_2)} \{ t : s \land M \} \cup D$. However, for each order-sorted substitution $\theta$ we know by Lemma 5.1 that $t\theta \in T_{\Sigma}(X)_{s_1} \cap T_{\Sigma}(X)_{s_2}$ iff there is a sort $s \in \text{glb}_\Sigma(s_1, s_2)$ such that $t\theta \in T_{\Sigma}(X)_s$. It follows that $\text{Un}(D_1) = \text{Un}(D_2)$.

- If Propagation is used, we know that $D_1$ and $D_2$ have the forms $D_1 = \{ f(t_1, \ldots, t_n) : s \land M \} \cup D$ and $D_2 = \bigcup_{s_1, \ldots, s_n \in \text{arg}_\Sigma(f,s)} \{ t_1 : s_1 \land \cdots \land t_n : s_n \land M \} \cup D$. However, for each order-sorted substitution $\theta$ we know by Lemma 5.2 that $f(t_1\theta, \ldots, t_n\theta) \in T_{\Sigma}(X)_s$ iff there are sorts $(s_1, \ldots, s_n) \in \text{arg}_\Sigma(f,s)$ such that $t_i\theta \in T_{\Sigma}(X)_{s_i}$ for $i \in [1,n]$.

- If Subsumption is used, we know that $D_1$ and $D_2$ have the forms $D_1 = \{ M_1, M_2 \} \cup D$ and $D_2 = \{ M_1 \} \cup D$ where $M_1 \geq M_2$. However, it follows easily that $\text{Un}(D_1) = \text{Un}(D_2)$ as for each substitution $\text{Un}(M_2) \subseteq \text{Un}(M_1)$ by Lemma 5.3. \hfill $\Box$

In order to preserve the set of substitutions, we also need to show that the inference rules do not discard variables or introduce new ones:

Lemma 5.6 If $D_1 \rightarrow^* D_2$ using the inference rules in Fig. 1 and each set of membership constraints $M_1 \in D_1$ has the same variables $\text{vars}(M_1) = X$, then for all $M_2 \in D_2$, $\text{vars}(M_2) = X$. 

25
Proof. This is a straightforward induction over the number of rewrites used to show $D_1 \rightarrow^* D_2$ and considering each rule separately.

The following lemma is useful to show that the inference rules terminate with a unique set of membership constraints.

Lemma 5.7 If $D_1$ and $D_2$ are both disjunctive sets of membership constraints that are irreducible by the inference rules in Fig. 1, then $\text{Un}_\Sigma(D_1) = \text{Un}_\Sigma(D_2)$ implies $D_1 = D_2$.

Proof. We show this by showing that $D_1 \neq D_2$ implies $\text{Un}(D_1) \neq \text{Un}(D_2)$. If $D_1 \neq D_2$, then there must be a conjunction of membership constraints $M_1$ which is in $D_1 \setminus D_2$ or $D_2 \setminus D_1$. We assume that the $M_1$ is in $D_1$ as the other case is symmetric. Since the rules in Fig. 1 cannot be applied to $D_1$, we know that $M_1$ must be reduced, and hence has the form $M_1 = x_1 : s_1 \land \cdots \land x_n : s_n$ with $x_i \neq x_j$ for each $i \neq j$. Let $\rho_{M_1}$ denote the substitution mapping each variable $x \in \text{vars}(M_1)$ to a fresh variable $x_{\rho_{M_1}}$ with sort $\text{sort}_{M_1}(x) \in S$. By definition $\rho_{M_1} \in \text{Un}_\Sigma(M_1)$ and therefore $\rho_{M_1} \in \text{Un}(D_1)$. If $\rho_{M_1} \notin \text{Un}(D_2)$, then $\text{Un}(D_1) \neq \text{Un}(D_2)$, and consequently we are done. Otherwise, $\rho_{M_1}$ is in $\text{Un}(D_2)$, and so there must be a membership $M_2 \in \text{Un}(D_2)$ such that for each membership $x : s \in M_2$ there is a membership $x : s' \in M_1$ with $s' \leq s$. It follows that $M_2 \geq M_1$. Since $D_1$ is fully reduced by the rules in Fig. 1, it follows that the substitution $\rho_{M_2}$ is not in $\text{Un}_\Sigma(D_1)$, since this would imply that there is a mapping $M \geq M_2 \geq M_1$ in $D_1$. This is impossible since $D_1$ has been fully reduced by the rules in Fig. 1.

Using the previous lemmas, it is not difficult to show the following Termination Theorem which shows that the inference rules terminate with a unique set of membership constraints.

Theorem 5.8 (Termination Theorem) For each disjunctive set of membership constraints $D$, there is a unique set of membership constraints $D^*$ such that $D \rightarrow^1 D^*$ using the inference rules in Fig. 1.

Proof. Showing this requires proving that: (1) the rules in Fig. 1 are terminating and (2) if $D \rightarrow^1 D_1$ and $D \rightarrow^1 D_2$, then $D_1 = D_2$. The rules in Fig. 1 are terminating, because each rewrite either reduces the size of a term in a membership, or preserves the terms while reducing the total number of memberships. To show (2), observe that if $D \rightarrow^1 D_1$ and $D \rightarrow^1 D_2$, then $\text{Un}_\Sigma(D_1) = \text{Un}_\Sigma(D) = \text{Un}_\Sigma(D_2)$ by Lemma 5.5. Therefore, $D_1 = D_2$ by Lemma 5.7.

We are now ready to conclude with a proof of the main theorem of the paper:

Theorem 5.9 Let $\mathcal{E} = (\Sigma, E)$ denote a preregular and sort-independent order-sorted theory, then given a unification problem $\Gamma$ with a complete set of most-general sort-preserving unsorted unifiers $\mathcal{U}$, $\text{OS}(\mathcal{U})$ is a complete set of most-general order-sorted unifiers for $\Gamma$.

Proof. Since $\mathcal{E}$ is sort-independent, we can assume that $E$ can be partitioned into rewrite rules $R$ and equations $A$ such that $\bar{R}$ is confluent and terminating modulo $\bar{A}$. Moreover, we can assume that each substitution $\bar{\theta} \in \mathcal{U}$ is $\bar{R}/\bar{A}$-irreducible.
Proving the above theorem requires showing three things: (1) each element of \( \text{OS}(\overline{U}) \) is an order-sorted unifier for \( \Gamma \); (2) the set of unifiers \( \text{OS}(\overline{U}) \) is complete; (3) the set of unifiers \( \text{OS}(\overline{U}) \) is most-general. We show each of these facts separately.

1. For each element \( \theta \in \text{OS}(\overline{U}) \), there is an unordered unifier \( \bar{\theta} \in \overline{U} \) and reduced membership constraints \( M \in D(\bar{\theta})^* \) such that \( \theta = \bar{\theta} \rhd M \). We first show that \( x_\theta \in T_\Sigma(X)_s \) for each variable \( x_\theta \in \text{vars}(\Gamma) \). To see this, observe that by definition \( \rho_M \in \text{Un}_\Sigma(D(\bar{\theta})^*) \), and so by Lemma 5.6, \( \rho_M \in \text{Un}_\Sigma(D(\overline{U})) \). Furthermore, by Lemma 5.6, we know that \( \text{vars}(M) = \text{vars}(D(\overline{U})) = \text{vars}(\Gamma) \). It follows by definition that for each variable \( x_\theta \in \text{vars}(\Gamma) \), \( x_\theta = x_\theta \rho_M \) is in \( T_\Sigma(X)_s \). For each equation \( t = u \in \Gamma \), we know that both \( t \) and \( u \) are well-sorted terms belonging to the same component. It follows that \( \theta \) and \( u \theta \) are well-sorted terms with the same connected component. By definition \( \theta \theta = \varepsilon u \theta \), and as \( \varepsilon \) is sort-independent it follows that \( \theta \theta = \varepsilon u \theta \).

2. To show that \( \text{OS}(\overline{U}) \) is complete, we must show for each order-sorted unifier \( \psi \in \text{Un}_\Sigma(\Gamma) \), there is an unifier \( \theta \in \text{OS}(\overline{U}) \) and order-sorted substitution \( \phi : \text{vars}(\theta) \rightarrow T_\Sigma(X) \) such that \( \psi = \varepsilon x_\theta \phi \) for each \( x \in \text{vars}(\Gamma) \). Let \( \psi \) be a unifier in \( \text{Un}_\Sigma(\Gamma) \). As \( U \) is a complete set of sort-preserving unifiers, there is an unifier \( \bar{\theta} \in U \) such that \( \psi = \varepsilon \bar{\theta} \phi \) for some unordered substitution \( \bar{\phi} : Y \rightarrow T_\Sigma(X) \) with \( Y = \text{vars}(\bar{\theta}) \). Moreover, since \( x_\psi \in T_\Sigma(X)_s \) for each variable \( x_\psi \in \text{vars}(\Gamma) \), we can assume that \( x_\psi \bar{\phi} \in T_\Sigma(X)_s \) since \( U \) is sort-preserving. It follows that \( \bar{\phi} \in \text{Un}_\Sigma(D(\overline{U})) \). By Lemma 5.5 and Theorem 5.8, there must be a reduced set of membership constraints \( M \in D(\bar{\theta})^* \) such that

\[
(\forall x \in Y) x_\psi \bar{\phi} \in T_\Sigma(X)_{\text{sort}(\bar{\psi})}.
\]

Since \( M \) is reduced, there is a variable renaming \( \rho_M \) with maps each variable \( x \in Y \) to a fresh variable \( x' \) with sort \( \text{sort}(\bar{\psi}) \). Let \( \rho_M^{-1} \) denote the inverse of that renaming. By using (1), it should be clear that \( \rho_M^{-1} ; \bar{\psi} \) is an order-sorted substitution. Moreover, as \( \rho_M \in \text{Un}_\Sigma(M) \) and therefore in \( \text{Un}_\Sigma(D(\overline{U})) \) by Lemma 5.5, \( \bar{\theta} ; \rho_M \) must be an order-sorted substitution. Since \( \bar{\theta} ; \rho_M \in \text{OS}(U) \) and \( \psi = (\bar{\theta} ; \rho_M) ; (\rho_M^{-1} ; \bar{\psi}) \), it follows that \( \text{OS}(U) \) is a complete set of unifiers.

3. To show that \( \text{OS}(\overline{U}) \) is a most-general set of unifiers, we must show for all distinct substitutions \( \theta_1, \theta_2 \in \text{OS}(\overline{U}) \), we have \( \theta_1 \not= \theta_2 \). We prove this by contradiction. Assume there are substitutions \( \theta_1, \theta_2 \in \text{OS}(\overline{U}) \) and a substitution \( \psi : Y \rightarrow T_\Sigma(X) \) such that \( \theta_1 = \theta_2 \psi \), where \( Y \) denotes the variables in the right-hand side of \( \theta_2 \). Since both \( \theta_1 \) and \( \theta_2 \) are in \( \text{OS}(\overline{U}) \), we know they must have the form \( \theta_1 = \overline{\theta}_1 ; \rho_{M_1} \) and \( \theta_2 = \overline{\theta}_2 ; \rho_{M_2} \) with \( \overline{\theta}_1, \overline{\theta}_2 \in U \), \( M_1 \in D(\overline{\theta}_1)^* \), and \( M_2 \in D(\overline{\theta}_2)^* \). Our assumption \( \theta_1 = \theta_2 \psi \) implies that \( \overline{\theta}_1 = \overline{\theta}_2 \psi ; (\rho_{M_2} ; \psi ; \rho_{M_1}^{-1}) \). Since \( U \) is most-general, this can only be the case if \( \overline{\theta}_1 = \overline{\theta}_2 \). As we assumed that \( \theta_1 = \theta_2 \psi \), it is not difficult to show that \( \psi = \rho_{M_2} \rho_{M_1} \). For each variable \( x \in Y \), we know that the sort of the variable \( x \rho_{M_2}^{-1} \rho_{M_1} \) is sort \( M_1 \{ x \rho_{M_2}^{-1} \} \), and sort \( M_1 \{ x \rho_{M_2}^{-1} \} \leq \text{sort}(M_2 \{ x \rho_{M_2}^{-1} \}) \) since \( \psi \) is an order-sorted substitution. It follows that \( M_2 \geq M_1 \) which is impossible since both \( M_1, M_2 \in D(\overline{\theta}_1)^* \) and \( D(\overline{\theta}_1)^* \) have been fully normalized using the inference rules in Fig. 1.

\[ \square \]
6 Conclusions and Related Work

There is a considerable amount of research already in unification in theories with sorts and subsorts (e.g., [1,6,8,11,13,14]) due to the improved expressiveness of order-sorted algebras and ability to simplify automated reasoning. The use of rule-based algorithms in describing unification has a long history as well with the most well-known example being Martelli and Montanari’s algorithm for syntactic unification [9]. Our use of a rule-based approach to order-sorted unification is not particularly novel; however we wanted to revisit order-sorted equational unification after discovering that the soundness for the AC and ACI unification problems we were trying to solve did not follow from previous results.

Due to our experience with the order-sorted unification in the Maude-NRL analyzer, an order-sorted unification engine is planned to be included in the next Maude release. This unification engine will make similar assumptions to our own ones about the supported theories, however it should have better performance as it will no longer need to parse unsorted unifiers back as strings, and can more tightly integrate the order-sorted constraints into the core unification routines. In fact, a prototype BDD-based approach to solving the sort constraints is currently being developed by Steven Eker. This BDD-based approach has the advantage that the subsumption checks can be handled automatically by the BDD generation-algorithms.

Our aim in this paper is more general than the Maude-based applications of our algorithm. Our aim is one of modularity, so that different formal tool building efforts needing equational order-sorted unification procedures may be able to modularly decompose such a procedure into its unsorted part where several existing tools may be used and the rule-based sort propagation algorithm that we have presented and proved correct.

Acknowledgments: The authors would like to thank the referees for comments which helped to improve the paper.

References


HENDRIX AND MESEGUER


Constraint-aware Schema Transformation

Tiago L. Alves
Universidade do Minho, Portugal
Software Improvement Group, The Netherlands

Paulo F. Silva
Universidade do Minho, Portugal

Joost Visser
Software Improvement Group, The Netherlands

Abstract

Data schema transformations occur in the context of software evolution, refactoring, and cross-paradigm data mappings. When constraints exist on the initial schema, these need to be transformed into constraints on the target schema. Moreover, when high-level data types are refined to lower level structures, additional target schema constraints must be introduced to balance the loss of structure and preserve semantics.

We introduce an algebraic approach to schema transformation that is constraint-aware in the sense that constraints are preserved from source to target schemas and that new constraints are introduced where needed. Our approach is based on refinement theory and point-free program transformation. Data refinements are modeled as rewrite rules on types that carry point-free predicates as constraints. At each rewrite step, the predicate on the reduct is computed from the predicate on the redex. An additional rewrite system on point-free functions is used to normalize the predicates that are built up along rewrite chains.

We implemented our rewrite systems in a type-safe way in the functional programming language Haskell. We demonstrate their application to constraint-aware hierarchical-relational mappings.

Keywords: Schema transformation, Constraints, Invariants, Data refinement, Strategic rewriting, Point-free program transformation, Haskell.

1 Introduction

Data schemas lie at the heart of software systems. Examples are relational database schemas, XML document schemas, grammars, and algebraic datatypes in formal specification. Data schemas prescribe not only the formats to which data instances must conform, but they also dictate the well-formedness of data queries and update functions. Generally, schema definitions consist of a structural description
augmented with constraints that capture additional semantic restrictions, e.g. SQL and XSD schemas may declare referential integrity constraints, grammars include operator precedences, VDM specifications contain datatype invariants.

Data schema transformations occur in a variety of contexts. For example, software maintenance commonly involves enhancement of the data formats employed for storing or exporting application data. Likewise, evolution of programming languages brings along modification of their grammars between versions. More complex schema transformations are involved in data mappings between programming paradigms [14], such as between XML and SQL.

When a data schema is transformed, the corresponding data instances, queries, and constraints must also be adapted. For example, when mapping an XML schema to an SQL schema, data conversion functions between schemas are required. When an XML schema is augmented with new document elements, the queries developed for that schema may need to be adapted to take these elements into account. When a datatype in a formal specification is adapted, so must its invariant and update functions.

In previous work, we and others have addressed the problem of transforming schemas together with the data instances and queries that are coupled with them. We have shown that data refinement theory can be employed to formalize schema transformation [1] as well as the transformation of the corresponding data instances [9]. In combination with point-free program transformation, this formalization extends to migration of data processors [11] including structure-shy queries and update functions [12]. We have harnessed this theoretical treatment in various type-safe rewrite systems and applied these to VDM-SL specifications [1], XML schemas and queries [11,5], and SQL databases [1,5].

We have also addressed the problem of propagation and introduction of constraints [5]. However, this approach was not theoretically supported, did not achieve type-safeness, and was limited to referential integrity constraints only.

In this paper, we propose an improved approach to constraint-aware schema transformation. Figure 1 concisely represents the new approach. Rather than labeling the types being transformed with cross-reference information as in [5], we augment them with general constraints represented by strongly-typed function representations. Constraint-propagation is achieved by composing a constraint \( \phi \) on a source data type with a backward conversion function \( \text{from} \) between target and
source type. Constraint-introduction is achieved by logical conjunction of a new constraint $\psi$ to the propagated constraint. Finally, point-free program transformation is applied to fuse the various ingredients of the synthesized constraint into a simplified form.

The paper is structured as follows. We introduce the problem of constraint-aware schema transformation with a motivating example in Section 2. We provide background about refinement theory and point-free program calculation in Section 3. Theoretical support about constraints representation and rewriting is provided in Section 4. In Section 5, we explain how this theory can be made operational in the form of strongly-typed rewriting systems implemented in the functional programming language Haskell. We return to the motivating example in Section 6 to demonstrate the application of our rewriting system to schema-aware hierarchical-relational mapping. We discuss related work in Section 7 and conclude in Section 8.

## 2 Motivating Example

To illustrate the objectives of our approach, we will pick up the motivating example from [5]. The diagram in Figure 2 represents an XML schema for a database of movies and TV series. The schema indicates that the database contains two main collections: one for shows (movies or TV series), and one for actors that play in those shows. Apart from the structure of the database, the following uniqueness constraints are present:

(i) A show is identified by its year and title.
(ii) An actor is identified by his/her name.
(iii) A season is identified by its yr.
(iv) A played element is identified by its year, title, and role.

Also the following referential integrity constraint is present:

(v) The year and title of a played element refer to the year and title of a show.

In XML Schema, such uniqueness and referential integrity constraints are defined by so-called identity constraints, using the key, keyref, and unique elements. More constraints could exist, such as that value is always non-zero, or that the name of

![Fig. 2. Schema for an XML database of movies and TV series, inspired by IMDb (http://www.imdb.com/). The shaded elements indicate unique keys in the respective collection elements. In addition, the year and title of the played element are a foreign key into the show collection.](image)
an episode is different from the title of the corresponding series. Such constraints could be expressed by general queries, e.g. using XPath.

When an XML-to-SQL data mapping is applied to our XML schema, an SQL database schema should result where the various constraints are propagated appropriately. In addition, new constraints would need to exist on the SQL schema that balance the loss of structure due to the flattening to relational form. The schema we need is the following:

```sql
shows(year,title)
reviews(id,year,title,review)
    foreign key (year,title) references shows(year,title)
movies(year,title,director)
    foreign key (year,title) references shows(year,title)
boxoffices(id,year,title,country,value)
    foreign key (year,title) references movies(year,title)
series(year,title)
    foreign key (year,title) references shows(year,title)
seasons(year,title,yr)
    foreign key (year,title) references series(year,title)
episodes(id,year,title,role,year,role)
    foreign key (year,title) references seasons(year,title,role)
actors(name)
    foreign key (name) references actors(name)
playeds(name,year,title,role)
    foreign key (name) references playeds(name,year,title,role)
awards(name,year,title,role,id,award)
    foreign key (name,year,title,role) references playeds(name,year,title,role)
```

In this pseudo-SQL notation, primary keys are indicated by underlining. The first foreign key constraint is an example of a newly introduced constraint. It arises from the fact that reviews were nested inside shows in the XML schema, but appear in a separate top-level table in the SQL schema. The first foreign key on the `playeds` table is an example of a constraint that was present in the original XML schema and was propagated through the data mapping.

In the remainder of this paper, we will demonstrate how schema transformations such as this XML-to-SQL data mapping can be constructed from strongly-typed algebraic combinators. The propagation of initial constraints and the introduction of new constraints will come for free.

### 3 Background

In this section, we will explain how schema transformation can be formalized by data refinement theory and point-free program transformation. We start in Section 3.1 by providing background on data refinement theory and its application to two-level transformation. In Section 3.2, we recapitulate point-free program transformation and show how it can be combined with data refinement to model query migration driven by schema transformation.

#### 3.1 Two-level transformation as data refinement

Data refinement theory provides an algebraic framework for calculating with datatypes [20,16,18]. The following inequation captures the essence of refining a datatype $A$ to a datatype $B$:

$$A \rightarrow B$$
Sequential and structural composition

\[
\begin{align*}
\text{if } A & \leq B \text{ and } B \leq C \text{ then } A \leq C \\
\text{if } A \leq B \text{ then } F A & \leq F B
\end{align*}
\]

Hierarchical-relational data mapping

\[
\begin{align*}
A^* & \leq N \rightarrow A \quad \text{List elimination} \\
2^A & \cong A \rightarrow 1 \quad \text{Set elimination} \\
A? & \cong 1 \rightarrow A \quad \text{Optional elimination} \\
A + B & \leq A? \times B? \quad \text{Sum elimination} \\
A \times (B + C) & \cong (A \times B) + (A \times C) \quad \text{Distribute product over sum} \\
A \rightarrow (B + C) & \leq (A \rightarrow B) \times (A \rightarrow C) \quad \text{Distribute map over sum (range)} \\
(B + C) \rightarrow A & \cong (B \rightarrow A) \times (C \rightarrow A) \quad \text{Distribute map over sum (domain)} \\
A \rightarrow (B \times (C \rightarrow D)) & \leq (A \rightarrow B) \times (A \times C \rightarrow D) \quad \text{Flatten nested map}
\end{align*}
\]

Fig. 3. Summary of data refinement theory. For a complete account, the reader is referred to Oliveira [20].
Note that \(\cdot \rightarrow \cdot\) denotes a simple relation, of which finite maps are a special case.

\[
A \leq B \quad \text{where} \quad \begin{cases}
to : A \rightarrow B \text{ injective and total} \\
\text{from} : B \rightarrow A \text{ surjective} \\
\text{from} \cdot \text{to} = \text{id}_A
\end{cases}
\]

Here, \(\text{id}_A\) is the identity function on datatype \(A\). Thus, the inequation \(A \leq B\) expresses that \(B\) is a refinement of \(A\), which is witnessed by the conversion functions \(\text{to}\) and \(\text{from}\). (In fact, \(\text{to}\) can be any injective and total relation, not necessarily a function.) In the special case where the refinement works in both ways we have an isomorphism \(A \cong B\). On the basis of this formalization of data refinement, an algebraic theory for calculation with datatypes has been constructed. This theory is summarized in Figure 3.

Data refinement theory can be used to formalize coupled transformation of schemas and their instances [9]. Such two-level transformations can be captured by sequential and structural compositions of data refinement rules. In particular, hierarchical-relational data mappings can be modeled by repeated application of elimination, distribution, and flattening rules, until a fixpoint is reached [1].

3.2 Point-free program transformation

In his 1977 Turing Award lecture, Backus advocated a variable-free style of functional programming, on the basis of the ease of formulating and reasoning with algebraic laws over such programs [3]. After Backus, others have adopted, complemented, and extended his work; an overview of this point-free style of programming is found in [10]. Some function combinators and associated laws that are used in the current paper are shown in Figure 4.

Point-free program transformation can be used after schema transformation to simplify the calculated conversion functions and to migrate queries from source
to target type or vice-versa [11]. In Section 5.4 we will use point-free program transformation to migrate and simplify constraints during schema transformation.

4 Refinement of datatypes with constraints

In this section we provide theoretical support about constraints representation and rewriting. The formalization of constraints is presented in Section 4.1. Section 4.2 discusses how constraints can be added to data refinement laws to formalize the propagation and introduction of constraints during schema transformation.

4.1 Data types with constraints

A constraint on a datatype can be modeled as a unary predicate, i.e. a boolean function which distinguishes between legal values and values that violate the constraint. To associate a constraint to a type, we will write it as a subscript: \( A_\phi \) where \( \phi : A \to B \) total and functional. This notation, as well as some of the results below, originates in [19]. We will write constraints as much as possible as point-free expressions, to enable subsequent calculation with them. For example, the following datatype represents two tables with a foreign key constraint:

\[
((A \to B) \times (C \to D))_{\mathit{set} \pi_1 \circ \rho_2 \subseteq \Delta \pi_1}
\]

Here we use projection functions \( \pi_1 \) and \( \pi_2 \) to select the left or right table, we use \( \delta \) and \( \rho \) to select the domain and range of a map, and \( \mathit{set} f \) to map a function \( f \) over the elements of a set. Additionally, we use a variant of the set inclusion operated lifted to point-free functions: \( \subseteq : (A \to \mathit{Set} B) \to (A \to \mathit{Set} B) \to (A \to B) \).

Hence, the defined constraint states that all values of \( A \) defined in the left table must be contained in the set of keys of the right table.
When a second constraint is added to a constrained datatype, both constraints can be composed with logical conjunction: \((A_{\phi})_{\psi} \equiv A_{\phi \land \psi}\).

When a constraint is present on a datatype under a functor, the constraint can be pulled up through the functor (for a categorical proof, see [19]):
\[
F(A_{\phi}) \equiv (FA)(F_{\phi})
\]
functor pull

For example, a constraint on the elements of a list can be pulled up to a constraint on the list: \((A_{\phi})^* \equiv (A^*)_{\text{list} \phi}\).

### 4.2 Introducing, propagating, and eliminating constraints

The laws of the data refinement calculus must be enhanced to deal with constrained datatypes. Firstly, if a constrained datatype is refined with a ‘classic’ law, i.e. a law that does not involve constraints, the constraint must be properly propagated through the refinement:
\[
\text{if } A \leq B \text{ then } A_{\phi} \leq B_{\phi \text{ from}}
\]

Thus, the constraint of the source datatype is propagated to the target datatype, where it is post-composed with the backward conversion function \(\text{from}\). Such compositions can give rise to opportunities for point-free program transformation, as we will see further on.

Several refinement laws can be changed from inequations to isomorphisms by adding a constraint to the target type. For example, the laws from Figure 3 for sum elimination, distribution of map over sum in its range, and flattening of nested maps can be enhanced as follows:
\[
A + B \cong A? \times B^2(\epsilon \circ \pi_1) \oplus (\epsilon \circ \pi_2)
\]
\[
A \to (B + C) \cong (A \to 1) \times (A \to B) \times (A \to C)(\delta \circ \pi_2 \subseteq \delta \circ \pi_1) \land (\delta \circ \pi_3 \subseteq \delta \circ \pi_1)
\]
\[
A \to (B \times (C \to D)) \cong (A \to B) \times (A \times C \to D)_{\text{set } \pi_1}(\delta \circ \pi_2 \subseteq \delta \circ \pi_1)
\]

Here, we have used point-free variants of exclusive disjunction (\(\oplus\)) and a test for emptiness of an optional (\(\epsilon\)).

When applying a law that introduces a constraint to a datatype that already has a constraint, the new and existing constraints must be combined:
\[
\text{if } A \leq B_{\psi} \text{ then } A_{\phi} \leq (B_{\psi})_{\phi \text{ from}} \equiv B_{\psi \land (\phi \text{ from})}
\]

This is the invariant pulling theorem of [19]. A more general case arises when not only the target, but also the source is constrained in the law that is applied:
\[
\text{if } A_{\chi} \leq B_{\psi} \text{ and } \phi \Rightarrow \chi \text{ then } A_{\phi} \leq B_{\psi \land (\phi \text{ from})}
\]

Here we use a point-free variant on logical implication (\(\Rightarrow\)) to state that the actual constraint \(\phi\) on \(A\) must imply the required constraint \(\chi\).

In addition to introduction and propagation, constraints can also be weakened or even eliminated, by virtue of the following: if \(\phi \Rightarrow \psi\) then \(A_{\phi} \leq A_{\psi}\).
In the special case that $\psi$ is the constant true predicate, such weakening boils down to elimination of a constraint.

5 Constraint-aware rewriting

In this section, we show how the enhanced data refinement theory of the previous section can be captured in a rewriting system, implemented as a strategic functional program in the functional language Haskell. In Section 5.1 we recall how type-safe representations of types and functions can be constructed using generalized algebraic datatypes (GADTs). In Section 5.2 we extend the type representation to constrained types. In Section 5.3 and 5.4, we explain how rewrite systems can be constructed to transform such constrained types.

5.1 Representation of types and functions

To represent both types and functions in a type-safe manner, we rely on generalized algebraic data types (GADTs) [21]. To represent types, we use a GADT:

```
data Type t where 
  One :: Type () 
  List :: Type a → Type [a] 
  Set :: Type a → Type (Set a) 
  ∙→· :: Type a → Type b → Type (a → b)  -- We use lhs2TeX for
  ∙+· :: Type a → Type b → Type (a + b)  -- type-setting various
  ∙×· :: Type a → Type b → Type (a, b)  -- symbols in Haskell.
  String :: Type String ...
```

In the result types of the various constructors of this GADT, the parameter $t$ has been instantiated exactly to the type that is represented by the constructor. Such instantiation is what distinguishes a GADT from a traditional parameterized algebraic datatype. To represent functions, we also use a GADT:

```
data F f where 
  id :: F (a → a) 
  ∙·· :: F (b → c) → F (a → b) → F (a → c) 
  ∙·∧· :: F (a → IB) → F (a → IB) → F (a → IB) 
  ∙·⊆· :: F (a → (Set b)) → F (a → (Set b)) → F (a → IB) 
  π1 :: F ((a, b) → a) 
  π2 :: F ((a, b) → b) 
  ∙×· :: F (a → b) → F (c → d) → F ((a, c) → (b, d)) 
  ∙·△· :: F (a → b) → F (a → c) → F (a → (b, c)) 
```

Note that the parameters in the result types are instantiated exactly to the type of the function being represented. For brevity, only a few constructors are shown.

Function representations can be evaluated to the function that is represented: $eval :: Type (a→b) → F (a→b) → a → b$. 

Note that GADTs help us to enforce that the type of the function produced matches the type of the function representation.

5.2 Representation of constrained types

To represent constrained datatypes, the first GADT above needs to be enhanced:

```haskell
data Type t where
  ...
  (·) :: Type a → F (a → I) → Type a
```

Thus, the (·) constructor has as first argument the type that is being constrained, and as second argument the function that represents the constraint. The use of GADTs pays off here, since it enforces that the function is of the right type. This use of the function representation inside the type representation has as important consequence that the rewriting system for functions will be embedded into the rewrite system for types, as we will see later.

To verify if the constraints hold for a specific value we defined `check`:

```haskell
check :: Type t → t → Bool
check One _ = True
check (t1 × t2) (x, y) = check t1 x ∧ check t2 y
...
check (tφ) x = eval (Func t Bool) φ x ∧ check t x
```

This function descends through a type representation and the corresponding value. Each time a constraint is found the `eval` function is applied to check its value.

5.3 Rewriting types and functions

The laws of point-free program transformation can be captured in rewrite rules of the following type:

```haskell
type RuleF = ∀ a b . F (a → b) → Maybe (F (a → b))
```

We make use of the `Maybe` monad to deal with partiality. For example, the law stating that `id` is the identity of composition is defined as follows:

```haskell
idR :: RuleF
idR (f ∘ id) = return f
idR _ = mzero
```

Single step rules of this kind can be combined into full rewrite systems using combinators like the following:

```haskell
nop :: RuleF        -- identity
(⊕) :: RuleF → RuleF → RuleF    -- sequential composition
(⊙) :: RuleF → RuleF → RuleF   -- left-biased choice
```
many :: RuleF → RuleF  -- repetition
once :: RuleF → RuleF  -- arbitrary depth rule application

For the implementation of these and other combinators, we refer elsewhere [12,11], as well as for how they can be combined into rewrite systems such as:

simplify :: RuleF  -- exhaustively apply rules until reaching normal form

To implement type transformations, we need a two-level rewrite system. A two-level rewrite rule can be represented as follows [9]:

type Rule = ∀a . Type a → Maybe (View (Type a))
data View a where View :: (a→b) → (b→a) → Type b → View (Type a)

The View constructor expresses that a type a can be refined to a type b if a pair of conversion functions between them exist. Note that only the source type a escapes from the type constructor of View. The Rule type expresses that, when rewriting a type representation we do not replace it but augment it with representation functions to translate between the source and the target types.

To compose two-level rewrite systems out of single rules, strategic rewrite combinators are defined, similar to those for rewriting point-free functions. A strategy flatten for hierarchical-relational mappings is defined for example in [9].

5.4 Constructing constraint-aware rewrite rules

The construction of constraint-aware rewrite rules differs from normal rules in three important details. Firstly, the rules need to introduce constraints on the target types. Secondly, they need to take into account the possible existence of a constraint on the source type, which needs to be propagated and combined with the newly introduced constraint. Thirdly, some rules require the existence of a constraint on the source type, which must be checked before rule application. To illustrate the first two issues, consider the rule for flattening nested maps.

flatMap :: Rule
flatMap (a ↭ (b × (c ↭ d)))) = return (View \n⁻¹ \n (tφ))

where t = (a ↭ b) × ((a × c) ↭ d)
φ = ((set π₁) ∘ (δ ∘ π₂)) ⊆ (δ ∘ π₁)

flatMap t = propagate flatten t

The first equation takes care of invariant introduction, where constraint φ is attached to the result type t. The issue of constraint propagation is dealt with by the helper function propagate, defined as follows:

propagate rule (aφ) = do
(View to from b) ← rule a
let ψ = φ ∘ from
return $ View to from (bψ)
propagate _ _ = mzero
The *propagate* function applies its argument rewrite rule to a constrained datatype \( a \) to obtain a new datatype \( b \) and the conversion functions \( to \) and \( from \). It post-composes constraint \( \phi \) with the \( from \) to obtain the new constraint \( \psi \). Finally, that constraint is attached to the result type \( b \).

The third issue, of checking the existence of a required constraint, comes into play in the construction of the reciprocal rule, which nests one map into another:

\[
\text{nestMap :: Rule}\\
\text{nestMap } ((a \rightarrow b) \times ((c \rightarrow d)) ) = \text{return } \mathcal{M}_n \mathcal{M}_n^{-1} (a \rightarrow (b \times (c \rightarrow d)))\\
\text{nestMap } _- = \text{mzero}
\]

Here, pattern matching is performed on the type as well as the constraint. Only if this constraint is equal to the required constraint, the rule succeeds. This does not take into account the possibility that the actual constraint implies the required constraint, but is not equal to it. In that case, some satisfiability proof is needed, which falls outside the scope of this paper (but see [17]).

The function compositions and nested constraints that are created during the application of rewrite steps can be simplified with the following rules:

\[
\text{compose\_constraint :: Rule}\\
\text{compose\_constraint } ((a \phi \psi)) = \text{return } (\mathcal{V} \mathcal{id} \mathcal{id} (a (\phi \land \psi)))\\
\text{compose\_constraint } _- = \text{mzero}
\]

\[
\text{fuse\_constraint :: Rule}\\
\text{fuse\_constraint } (a \phi) = \text{do} \\
\quad \psi \leftarrow \text{simplify } \phi \\
\quad \text{return } (\mathcal{V} \mathcal{id} \mathcal{id} (a \psi))\\
\text{fuse\_constraint } _- = \text{mzero}
\]

In the latter rewrite rule, the rewrite system *simplify* for point-free functions is invoked, which means that our first function rewrite system will be embedded in our type rewrite system.

## 6 Application to hierarchical-relational mapping

We will now revisit the example of Section 2. The schema of Figure 2 can be captured by the following type representation:

\[
\text{imdb } = (\text{actors } \times \text{show})_{\text{imdb\_inv}}\\
\text{where } \text{imdb\_inv } = (\mathcal{S} \pi_1) \circ \text{fuse} \circ (\mathcal{S} \delta) \circ \rho \circ \pi_1 \subseteq \delta \circ \pi_2\\
\text{show } = ("Year" \times "Title") \rightarrow ((\mathcal{L} "Review") \times ("movie + series"))\\
\text{movie } = (\mathcal{L} ("Country" \times "Value")) \times "Director"\\
\text{series } = "Yr" \rightarrow (\mathcal{L} \text{episode})\\
\text{episode } = "Name" \times (\mathcal{M} \text{"Director"})\\
\text{actors } = "Name" \rightarrow \text{played}\\
\text{played } = (("Year" \times "Title") \times "Role") \rightarrow (\mathcal{L} "Award")
\]
The primary key constraints of the original schema are captured structurally, by the employment of finite maps. The foreign key constraint is captured by \textit{imdb\_inv}, which specifies that the values in the domain of \textit{played} are contained in the domain of \textit{show}, i.e. the year and title (defined in the domain) of \textit{played} are references to the year and title defined in \textit{(the domain of) show}. This constraint is expected to be propagated through the schema transformation process.

The result of the transformation from hierarchical to its relation equivalent, using the \textit{flatten} strategy of \cite{Alves2011}, followed by constraint simplification, is as follows:

\[
\text{played} \times \text{awards} \times \text{actors} \times \text{shows} \times \text{reviews} \times \text{seasons} \\
\times \text{episodes} \times \text{series} \times \text{movies} \times \text{boxoffices}\]

\textbf{where}

\textit{played} = \((\text{"Name"} \times \text{"Year"} \times \text{"Title"} \times \text{"Role"}) \rightarrow \text{One})

\textit{awards} = \((\text{"Name"} \times \text{"Year"} \times \text{"Title"} \times \text{"Role"}) \times \text{Int}) \rightarrow \text{"Award")}

\textit{actors} = \text{"Name"} \rightarrow \text{One}

\textit{shows} = \((\text{"Year"} \times \text{"Title"}) \rightarrow \text{One}

\textit{reviews} = \((\text{"Year"} \times \text{"Title"}) \times \text{Int}) \rightarrow \text{"Review"}

\textit{seasons} = \((\text{"Year"} \times \text{"Title"}) \times \text{"Yr"}) \rightarrow \text{One}

\textit{episodes} = \((\text{"Year"} \times \text{"Title"}) \times \text{"Yr"} \times \text{Int}) \rightarrow \text{("Name" \times \text{"Role")}

\textit{series} = \((\text{"Year"} \times \text{"Title"}) \rightarrow \text{One}

\textit{movies} = \((\text{"Year"} \times \text{"Title"}) \rightarrow \text{"Director")}

\textit{boxoffices} = \((\text{"Year"} \times \text{"Title"}) \times \text{Int}) \rightarrow \text{("Country" \times \text{"Value")}

\textit{inv} = \text{fk1} \land \text{fk2} \land \text{fk3} \land \text{fk4} \land \text{fk5} \land \text{fk6} \land \text{fk7} \land \text{fk8} \land \text{fk9}

\textit{fk1} = (\text{set \{pi\}}) \circ \delta \circ \pi_{\text{boxoffices}} \subseteq \delta \circ \pi_{\text{movies}}

\textit{fk2} = (\text{set \{pi\}}) \circ \delta \circ \pi_{\text{episodes}} \subseteq \delta \circ \pi_{\text{seasons}}

\textit{fk3} = (\text{set \{pi\}}) \circ \delta \circ \pi_{\text{episodes}} \subseteq \delta \circ \pi_{\text{seasons}}

\textit{fk4} = (\text{set \{pi\}}) \circ \delta \circ \pi_{\text{episodes}} \subseteq \delta \circ \pi_{\text{shows}}

\textit{fk5} = \delta \circ \pi_{\text{movies}} \subseteq \delta \circ \pi_{\text{shows}}

\textit{fk6} = \delta \circ \pi_{\text{movies}} \subseteq \delta \circ \pi_{\text{shows}}

\textit{fk7} = (\text{set \{pi\}}) \circ \delta \circ \pi_{\text{awards}} \subseteq \delta \circ \pi_{\text{played}}

\textit{fk8} = (\text{set \{pi\}} \circ \pi_{\text{actors}} \circ \pi_{\text{played}} \subseteq \delta \circ \pi_{\text{actors}}

\textit{fk9} = (\text{set \{pi\} \circ \pi_{\text{actors}}}) \circ \delta \circ \pi_{\text{played}} \subseteq \delta \circ \pi_{\text{shows}}

Here, we have introduced table names for readability and for comparison to the expected result shown in pseudo-SQL in Section 2. The result consists of 10 tables: 3 derived from the \textit{actor} subschema and 7 from \textit{show}. Additionally, 9 constraints are obtained from which 8 were introduced during transformation. Constraint \textit{fk9} results from the propagation of the original constraint \textit{imdb\_inv}. Note that without invocation of the \textit{flatten} rewrite system, the synthesized constraints would not be so concise. For example, without rewrite, an initial fragment of the \textit{fk2} constraint would be:

\[
fk2 = ((\text{set \{pi\}} \circ \delta \circ \pi_{\text{played}}) \subseteq \delta \circ \pi_{\text{actors}}) \circ (((\text{assocl} \rightarrow \text{id}) \times \text{id}) \circ (\text{id} \times (\text{assocl} \rightarrow \text{id})) \circ (\text{assoc\_value}) \times \text{id}) \circ (\text{id} \times (\text{assocr} \rightarrow \text{id})) \circ (\text{id} \times \pi_{\text{actors}}) \circ \pi_{\text{played}} \circ \pi_{\text{played}} \circ ...\]

Note that in general, simplification can not be postponed until after rewriting, since rules that match on constraints expect them to be in simplified form.

To validate the result we can insert information into the database and observe the constraint checking result. For example, we can add information about the role of an actor in a movie:

\[
\text{> db' \leftarrow addActorsPlayed db } ((\text{"Jet Li", (2001, \text{"The One")}}, \text{"Lawless")})
\text{> check imdbResult db'}
\text{False}
\]

42
The constraint correctly fails since neither the name of the actor nor the show exist. We should add that information first:

```sql
> db' ← addShow db (2001, "The One")
> db'' ← addActor db' "Jet Li"
> db''' ← addActorsPlayed db'' (("Jet Li", (2001, "The One")), "Lawless")
> check imdbResult db'''
```

Now the constraint check succeeds.

### 7 Related work

A large number of approaches has been proposed for mapping XML to relational databases \cite{7,6,2,4}, but usually without taking constraints into account. Lee et al \cite{15} first addressed the issue of constraint preservation. Their CPI algorithm deals with referential integrity and some cardinality constraints, which are stored in an annotated DTD graph. When the graph is serialized to an SQL schema, various SQL constraints are generated along with the tables. In contrast to our approach, this graph-based algorithm does not deal with arbitrary constraints, it is specific for hierarchical-relational mapping, and it lacks type-safety and formal justification.

A notion of XML Functional Dependency (XFD) was introduced by Chen et al \cite{8}, based on path expressions. Mapping algorithms are provided that propagate XFDs to the target relational schema and exploit XFDs to arrive at a schema with less redundancy. Davidson et al \cite{13} present an alternative constraint-preserving approach, also using path expressions. In contrast, our constraints are not restricted to relational integrity constraints. We have expressed constraints as point-free functions, which can be converted automatically to and from structure-shy programs including path expressions \cite{12}.

Barbosa et al \cite{4} discuss generation of constraints on relational schemas that make XML-relational mappings information preserving, i.e. isomorphic. Non-structural constraints on the initial XML schema are not taken into account. Constraints and conversion functions are expressed in (variations on) Datalog, which can be (manually) rewritten to normal form in a mechanical way.

Berdaguier et al. \cite{5} employ a type annotation mechanism to capture constraints. As a result, a smaller class of possible constraints is covered. Nevertheless, the annotation mechanism allows for a compositional treatment of constraint-aware schema transformation. Rather than path expressions or labels, our approach employs strongly-typed boolean functions to capture constraints. This has the advantage of being more expressive, and allowing a fully compositional treatment. Also note that our approach is not limited to hierarchical-relational mappings, as it can be used for schema transformation in general.

### 8 Concluding remarks

**Contributions** We have contributed a treatment of constraint-aware schema transformation to a line of research on the application of data refinement and point-
free program transformation to problems of coupled transformation of data schemas, data instances, and queries [20,1,9,11,12,5]. In particular:

• we have shown how data refinement theory [20] can be enhanced to include types constrained by boolean predicates, which amounts to extending the work of [19];

• we have enhanced refinement rules for hierarchical-relational mapping [1,5] such that appropriate constraints are introduced on target types, turning some refinements into isomorphisms;

• we have extended rewrite systems for two-level transformation [9] and coupled transformation [11] to include the propagation, introduction, and simplification of constraints. Moreover, we have shown that value-level rewriting needs to be done during type-level rewriting, because type-level rewrite rules may trigger on types with normalized constraints only;

• we have demonstrated the use of the extended rewriting systems for mapping XML schemas to SQL schemas where referential integrity constraints are generated automatically for the target schema. The approach taken in this paper is an alternative to the approach of [5], which was limited to constraints representable with a particular labeling trick while we deal with constraints in general.

Several directions of future work are envisioned.

Constraints as co-reflexive relations We have modeled constraints as boolean functions. Another approach is to model constraints as co-reflexive relations. One advantage of the alternative approach is that it would allow us to use a relational proof system [17] during rewriting to check whether the actual constraint of a redex implies the constraint required by a rule with constrained source datatype.

Integration We want to integrate the treatment of constraints presented here with the front-ends and name-preservation developed in the context of the label-based treatment [5]. Likewise, we want to integrate a rewrite system for structure-shy queries of [12] such that we can deal with structure-shy constraints.

Acknowledgement

Thanks to José Nuno Oliveira and Alcino Cunha for inspiring discussions. The first two authors are supported by the Fundação para a Ciência e a Tecnologia, grants SFRH/BD/30215/2006 and SFRH/BD/19195/2004, respectively.

References


P. Bohannon et al. From XML schema to relations: A cost-based approach to XML storage. In ICDE ’02: Proc. 18th Int. Conf. on Data Engineering, pages 64–. IEEE Computer Society, 2002.


A Rewriting-Based Model Checker for the Linear Temporal Logic of Rewriting

Kyungmin Bae
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, USA

José Meseguer
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, USA

Abstract
This paper presents a model checker for LTLR, a subset of the temporal logic of rewriting TLR∗ extending linear temporal logic with spatial action patterns. Both LTLR and TLR∗ are very expressive logics generalizing well-known state-based and action-based logics. Furthermore, the semantics of TLR∗ is given in terms of rewrite theories, so that the concurrent systems on which the LTLR properties are model checked can be specified at a very high level with rewrite rules. This paper answers a nontrivial challenge, namely, to be able to build a model checker to model check LTLR formulas on rewrite theories with relatively little effort by reusing Maude’s LTL model checker for rewrite theories. For this, the reflective features of both rewriting logic and its Maude implementation have proved extremely useful.

Keywords: temporal logic of rewriting, model checking, rewriting logic, reflective transformation

1 Introduction

In temporal logic and model checking one can distinguish two main camps: a state-based camp, in which all atoms in formulas are state predicates (e.g., LTL, CTL, and CTL∗ [?]); and an event-based camp, where the formulas’ atoms are actions or events (e.g., Hennesy-Milner’s logic [?], or De Nicola and Vaandrager’s A-CTL∗ logic [?]). At the semantic level, state based formulas are evaluated on Kripke structures. Instead, action-based formulas are evaluated on labeled transition systems.

Some properties can be naturally expressed in state-based logics and are difficult to express in action-based logics, whereas the opposite is the case for other properties. This means that, when the property does not fit well a given logic, one
has to “cook” in a possibly complex way both the system description (as a Kripke structure or a label transitions system depending on the logic’s semantics) and the property in order to model check it in the given logic. The situation is even more challenging for mixed properties such as fairness properties (see the discussion [?]), where both state-based predicates and actions are involved.

Reflecting on this situation we can speak, as in [?], of tandems, each given by a pair \( L_S/L_P \), where \( L_S \) is a formalism to specify systems (for example, the formalisms of Kripke structures or of label transition systems), and \( L_P \) is a formalism to describe properties (for example, some state-based or action-based temporal logic). The frequent need for “cooking” both the system and the property in both state-based and action-based tandems is then due to a lack of expressiveness in both cases.

1.1 An Example

To illustrate the lack of expressiveness, of either solely state-based or solely action-based systems, we use a variant of a simple parallel language, whose rewriting semantics is presented in [?], in which we can define Dekker’s algorithm for mutual exclusion and then model check some of its properties. The parallel language supports processes that execute concurrently on a shared memory machine and communicate with each other through shared variables.

Dekker’s algorithm has two processes with entirely symmetric code. Process 1 sets a Boolean variable \( c_1 \) to 1 to indicate that it wishes to enter its critical section. Process 2 does the same with variable \( c_2 \). If one process, after setting its variable to 1 finds that the variable of its competitor is 0, then it enters its critical section right away. In case of a tie (both variable set to 1) the tie is broken using a variable \( turn \) that takes values in \{1,2\}. For example, the code of process 1 is as follows:

\[
\text{repeat} \\
\quad 'c1 := 1 ; \\
\quad \text{while } 'c2 = 1 \text{ do} \\
\quad \quad \text{if } 'turn = 2 \text{ then} \\
\quad \quad \quad 'c1 := 0 ; \\
\quad \quad \quad \text{while } 'turn = 2 \text{ do skip od} ; \\
\quad \quad 'c1 := 1 \\
\quad \text{fi} \\
\quad \text{od} ; \\
\quad \text{crit ; 'turn := 2 ; 'c1 := 0 ; rem} \\
\quad \text{forever}
\]

where fragments of code for the critical section and for the remaining part of the program are respectively abstracted as constants \( \text{crit} \) and \( \text{rem}^{3} \).

Global states are modeled as pairs, with first component a set of processes, and second component a shared memory. In their Maude rewriting semantics such global states are instances of the pattern \( \{ \{I,R\} \mid S, M \} \), where \( \{I,R\} \) is one of the processes with \( I \) its process id and \( R \) its program code to be executed next by process, \( \mid \) is an associative-commutative parallel process composition operator, \( S \) is the remaining set of processes, and \( M \) is the shared memory. The language’s operational semantics is then defined by rewriting rules for each language feature.

A property to be checked (in fact it fails, see Section 5) for Dekker’s algorithm is the strong fairness property that executing infinitely often implies entering one’s

---

3 We assume that \( \text{crit} \) is \text{terminating}, but \( \text{rem} \) may not be. See [?] for a “cooked” version of the example, and [?] for the Maude specification of both the example and is \text{LTLR} properties.
critical section infinitely often, expressed as the LTL-like formula
\( \diamond \diamond \text{exec.p} \Rightarrow \diamond \diamond \text{in-crit.p}. \)

The predicate \( \text{in-crit.p} \) means that process 1 is in its critical section and can be defined easily by the equation
\( \{[p_1, \text{crit} ; R] \mid S, M\} \models \text{in-crit.p} = \text{true}. \)

However, the above strong fairness formula is only “LTL-like” and not really an LTL formula, since \( \text{exec.p} \), which asserts the execution of process 1, cannot be defined directly, because there is no way to know which process has executed some statement using the current state description. To express \( \text{exec.p} \) we need to “cook” both the state representation and the operational semantics by adding a third component to the state, indicating the last executed process, and modifying the semantic rules to update such third component. After this “cooking,” we can define \( \text{exec.p} \) as a state predicate by the equation
\( \{[I, R] \mid S, M, p_1\} \models \text{exec.p} = \text{true}. \)

This is just a convoluted way to represent what is really an action (\( \text{exec.p} \)) indirectly as a state predicate. Likewise, there are also natural properties in state-based system that would need a convoluted “cooking” to be represented as actions. For example, a predicate which is true when a process is in its rem part, say \( \text{in-rem.p} \), is nontrivial to define in an action-based system, while it is trivial to define in a state-based system by an equation similar to that for \( \text{in-crit.p} \).

In this paper we describe a new tandem \( \text{RewritingLogic}/\text{TLR}^* \), first proposed in [?], where the need for cooking both the system and the property disappears. For example, the above strong fairness formula for Dekker can be expressed directly in \( \text{TLR}^* \). We also present a Maude-based model checker for the LTLR sublogic of \( \text{TLR}^* \), which extends with actions the LTL logic. The point is that rewriting logic [?] is more expressive than both Kripke structures and labeled transitions systems, since in a rewrite theory state predicates can be equationally specified, and rewrite rules are labeled. The logic \( \text{TLR}^* \), called the temporal logic of rewriting, extends CTL* with spatial action patterns, which are quite expressive, since they can localize a rewrite rule’s action to a given context and a partial substitution.

The nontrivial challenge answered in this paper is to be able to build a model checker for the \( \text{RewritingLogic}/\text{LTLR} \) tandem with relatively little effort by reusing Maude’s LTL model checker for rewrite theories. For this, the reflective features of both rewriting logic and its Maude implementation [?] have proved extremely useful. In essence, the new model checker design uses a result in [?] by which the model checking of a LTLR formula on a rewrite theory \( \mathcal{R} \) can be reduced to the model checking of a translated LTL formula on a translated Kripke structure. Using reflection, this is here achieved by a reflective theory transformation associating each rewrite theory \( \mathcal{R} \) with a new theory such that the given LTLR formula holds for \( \mathcal{R} \) and a given initial state if and only if its LTL translation holds for the new theory.

The paper is organized as follows. Section 2 presents the necessary background and Section 3 provides the formal foundations, defining and proving correctness of the reflective construction. Section 4 explains the reflective design of a LTLR model checker based on the foundations. Section 5 illustrates the use of the model checker with an example; and Section 6 discusses related work and presents conclusions.
2 Rewriting Logic and the Temporal Logic of Rewriting

This section explains the concepts of RewritingLogic and TLR*, constituting the RewritingLogic/TLR* tandem, which is the semantic infrastructure of the model checker described in this paper.

2.1 Rewriting Logic

A rewrite theory is a formal specification of a concurrent system with static states and concurrent transitions between the states. More precisely, a rewrite theory is a triple \( R = (\Sigma, E, R) \)\(^4\) such that:

- \((\Sigma, E)\) is a many-sorted equational theory.\(^5\) The initial algebra \( T_{\Sigma,E} \) defined by the equational theory \((\Sigma, E)\) defines the states of the system specified by \( R \).
- \( R \) is a collection of rewrite rules of the form \( l : q \rightarrow r \), with \( l \) a label (which can be duplicated for several rules), \( q \) and \( r \) \( \Sigma \)-terms of the same sort, and such that the set of variables \( \text{vars}(r) \) is a subset of the variables \( \text{vars}(q) \). These rewrite rules define concurrent transitions between states.

More precisely, each state is modeled as an \( E \)-equivalence class \([t]_E\) of ground terms, and rewriting happens \emph{modulo} \( E \): that is, rewriting \( E \)-equivalence classes \([t]_E\) representing states, not just terms \( t \). A \emph{one-step rewrite} \([t]_E \rightarrow^R [t']_E\) exists in \( R \) iff there exists \( u \in [t]_E \) such that \( u \) can be rewritten to \( v \) using some rule \( l : q \rightarrow r \) in \( R \) in the standard way,\(^6\) denoted \( u \rightarrow^R v \), and we furthermore have \( v \in [t']_E \).

The most useful rewrite theories satisfy additional executability conditions, because for arbitrary \( E \) and \( R \), whether \([t]_E \rightarrow^*_R [t']_E\) is \emph{undecidable} in general. A rewrite theory \( R = (\Sigma, E \cup A, R) \) is \emph{computable} if the following conditions hold:

- (i) Equality modulo \( A \) is decidable, and there exists a matching algorithm modulo \( A \), producing a finite number of \( A \)-matching substitutions or failing otherwise, that can implement rewriting in \( A \)-equivalence classes.
- (ii) \((\Sigma, E \cup A)\) is \emph{ground terminating and confluent} modulo \( A \)\(^2\). That is: (i) there are no infinite sequences of rewritings with \( E \) module \( A \); and (ii) for each \([t]_A \in T_{\Sigma/A} \) there is a unique \( A \)-equivalence class \([\text{can}_{E/A}(t)]_A \in T_{\Sigma/A} \) called the \emph{\( E \)-canonical form} of \([t]_A \) modulo \( A \) such that the last term, which cannot be further rewritten with \( E \) module \( A \), of any terminating sequence beginning at \([t]_A \) is necessarily \([\text{can}_{E/A}(t)]_A \).
- (iii) The rules \( R \) are \emph{ground coherent} relative to the equations \( E \) modulo \( A \)\(^2\). That is, if \([t]_A \) is rewritten to \([t']_A \) by a rule \( l \) in \( R \), \([\text{can}_{E/A}(t)]_A \) is also rewritten to \([t'']_A \) by the same rule \( l \) such that \([\text{can}_{E/A}(t'')]_A = [\text{can}_{E/A}(t'')]_A \)

---

4 This definition can be extended to the more general rewrite theories in \([?], [?]\), which uses a more expressive equational logic, \emph{conditional} rewrite rules, and \emph{frozen} function operators.

5 There are various possibilities for the equational theory \((\Sigma, E)\) such as unsorted, many-sorted, order-sorted, or even membership equational logic. However, to keep the exposition as simple as possible, we will assume that \((\Sigma, E)\) is a many-sorted equational theory.

6 See [?] for basic notation on term rewriting. Positions in a term are denoted as strings of nonzero natural numbers and represent tree positions when the term is parsed as a tree. Two useful notions are that of a subterm of a given term \( t \) at a given position \( p \), denoted \( t[p] \), and of replacement in \( t \) of such a subterm by another term \( u \) at position \( p \), denoted \( t[u]_p \). For example, in the term \( t = x + (z + 0) + y \), the subterm at position \( 2.1 \) is \( z + 0 \), and the replacement \( t[z|2.1] \) is the term \( x + (z + y) \).
In addition, to make the integration of rewriting logic and TLR\(^*\) smoother, we define the class \(R\!W\!T\!h_0\) of rewrite theories as follows.

**Definition 2.1** \(\mathcal{R} \in R\!W\!T\!h_0\) when the following condition is satisfied:

- \(\mathcal{R}\) is computable and has a sort \(\text{State}\) as its chosen sort of states.
- If \(\mathcal{R}\) has a sort named \(\text{Prop}\), then it must also have a sort named \(\text{Bool}\) with constants true and false, and an operator \(\_ \models \_ : \text{State} \times \text{Prop} \rightarrow \text{Bool}\).\(^7\)
- The subsignature \(\Pi \subseteq \Sigma\) of its state predicate symbols is the set of all operators in \(\Sigma\) of the form \(p : A_1 \times \cdots \times A_n \rightarrow \text{Prop}\), \(n \geq 0\). \(A_1, \ldots, A_n\) are called the parameter sorts of the atomic state predicate \(p\).
- \(\mathcal{R}\) is deadlock-free, that is, there are no finite sequences
  \[
  [t_1]_A \rightarrow \mathcal{R} [t_2]_A \ldots \rightarrow [t_n]_A \rightarrow \mathcal{R} [t_{n+1}]_A
  \]
such that \([t_{n+1}]_A\) cannot be further rewritten (i.e., it is a “deadlock state”). This is not at all a strong restriction, since, as explained in [?], any rewrite theory \(\mathcal{R}\) whose rules do not have rewrites in their conditions can be transformed into a semantically equivalent theory \(\hat{\mathcal{R}}\) that is deadlock-free.

**Proof Terms and Computations**

The inference rules of rewriting logic derive all concurrent computations in the system specified by \(\mathcal{R} [\text{Prop}, \text{State}]\). That is, given two states \([u], [v] \in T_{\Sigma/E/A}\), one can reach \([v]\) from \([u]\) by some possibly complex concurrent computation if and only if one can prove \(\mathcal{R} \vdash [u] \rightarrow [v]\)\(^8\). In rewriting logic any such complex computation reaching \([v]\) from \([u]\) is witnessed by a proof term \([\text{Prop}, \text{State}]\), say \(\lambda\), written \(\mathcal{R} \vdash \lambda : [u] \rightarrow [v]\).

Proof terms are identified modulo natural equations making any proof term \(\lambda\) always equivalent to an interleaving description as a sequential composition \(\gamma_1; \ldots; \gamma_k\) of one-step proof terms \(\gamma_i[\text{Prop}, \text{State}]\), which have a very simple algebraic description.

**Definition 2.2** Given rewrite proof \(\mathcal{R} \vdash \gamma : [u] \rightarrow [v]\) using a rewrite rule \(l : q \rightarrow r \in \mathcal{R}\), a one-step proof term \(\gamma\) has the form \(t[l(\phi)]_p\), where \(t \in [u]\), \(p\) is a position in \(t\) where the rule is applied, and \(\phi = \{x_1 \mapsto u_1, \ldots, x_n \mapsto u_n\}\) is a substitution such that \(t[p] = \phi(q), t[\phi(r)]_p \in [v]\), where \(x_1, \ldots, x_n\) are variables in \(q\).

In the above definition, if \(t\) is an \(E/A\)-canonical term, say \(t = \text{can}_{E/A}(u)\), we say that \(\text{can}_{E/A}(u)[l(\phi)]_p\) is a canonical one-step proof term \([\text{Prop}, \text{State}]\). Canonical one-step rewrite proofs are the key ingredient to arrive at our desired notion of computation, on which the truth of TLR\(^*\) path formulas will be evaluated. Before defining computation, let us define two useful sets of canonical proof terms. First, the set \((\text{Can}_{\Sigma/E,A})_{\text{State}}\) of all \(A\)-equivalence classes of the form \(\text{can}_{E/A}(t)_A\), where \(t\) is a ground \(\Sigma\)-term of sort \(\text{State}\). Second, we can define the set \(\text{CanPTerms}^1(\mathcal{R})\) of all one-step canonical proof terms in \(\mathcal{R}\). In a computable rewrite theory, any proof term is always semantically equivalent to a canonical one \([?]\).

---

\(^7\) Prop is the designated sort of atomic state predicates, and \(\models\) is the function defining whether a given state satisfies a given state predicate.

\(^8\) \([u] \rightarrow [v]\) denotes a combination of one or more sequential compositions of concurrent rewrites.
Definition 2.3 An infinite computation in $R \in RWTh_0$ is a pair of functions $(\pi, \gamma)$, with $\pi : \mathbb{N} \rightarrow (\text{Can}_\Sigma/E,A)_{\text{State}}$ and $\gamma : \mathbb{N} \rightarrow \text{CanTerms}^\infty(R)$ such that for all $n \in \mathbb{N}$, $\pi(n) \xrightarrow{\gamma(n)} \pi(n + 1)$ is a canonical one-step rewrite proof in $R$. Graphically, $\pi(0) \xrightarrow{\gamma(0)} \pi(1) \xrightarrow{\gamma(1)} \pi(2) \cdots \pi(n) \xrightarrow{\gamma(n)} \pi(n + 1) \cdots$

$\text{Comp}(R)^\infty$ denotes the set of infinite computations in $R$, and for each $[t] \in (\text{Can}_\Sigma/E,A)_{\text{State}}$, $\text{Comp}(R)_[t]^\infty$ denotes the infinite computations starting at $[t]$, that is, those computations $(\pi, \gamma)$ such that $\pi(0) = [t]$. Given an infinite computation $(\pi, \gamma)$ and a number $i \in \mathbb{N}$, $(\pi, \gamma)^i$ denotes the suffix of $(\pi, \gamma)$ beginning at position $i$, that is, the pair of functions $(\pi \circ s^i, \gamma \circ s^i)$ with $s$ the successor function, $s^0$ the identity function, and $s^{n+1} = s \circ s^n$.

Spatial Actions

Spatial actions are the action atoms of TLR*. They generalize one-step proof terms, which can be thought of as ground-instantiated spatial actions. Spatial actions describe patterns, which in general specify not just a single one-step proof term, but a possibly infinite set of such proof terms. Roughly speaking, we can think of spatial actions as “one-step proof terms with variables,” but they are slightly more general than that as we explain below.

Let $\Omega$ be the subsignature of constructors* and $L$ be the set of labels labeling rules in $R$, and assume that $\Omega \cap L = \emptyset$. The signature $\Omega(L)$ extends $\Omega$ by adding:

- fresh sorts $\text{Top}$ and $\text{Subst}$
- an associative and commutative operator $\_;\_ : \text{Subst} \rightarrow \text{Subst}$
- for each rewrite rule $l : q \rightarrow r$ in $R$ with $q, r$ of sort $B$, and with variables $x_1, \ldots, x_n$ in $q$ having sorts $B_1, \ldots, B_n$:
  - operators $l : \text{Subst} \rightarrow B$ and $x_1\_ : B_1 \rightarrow \text{Subst}$, $\ldots, x_n\_ : B_n \rightarrow \text{Subst}$
  - a constant $l$ of sort $B$ and an operator $\text{top} : B \rightarrow \text{Top}$

Let $X$ be a many-sorted set of variables with an infinite set of variables for each sort in $\Omega$. Consider the algebras: (i) $T_{\Omega(L)/A}(X)$ of $A$-equivalence classes of $\Omega(L)$-terms with variables in $X$; and (ii) $T_{\Omega/A}(X)$ of $A$-equivalence classes of $\Omega$-terms with variables in $X$. Also, assume that the substitution $\phi$ has the form $x_1\_u_1; \cdots; x_n\_u_n$.

Definition 2.4 $R$’s spatial action patterns $\text{SP}(\Omega, L) \subset T_{\Omega(L)/A}(X)$ are defined by:

- for each $l \in L$, $\,[l]_A, [\text{top}(l)]_A \in \text{SP}(\Omega, L)$
- $[l(\phi)]_A \in \text{SP}(\Omega, L)$ if $l \in L$, $[l(\phi)]_A \in T_{\Omega(L)/A}(X)$, $u_1, \ldots, u_n \in T_{\Omega/A}(X)$
- $[\text{top}(l(\phi))]_A \in \text{SP}(\Omega, L)$ if $l \in L$, $[\text{top}(l(\phi))]_A \in T_{\Omega(L)/A}(X)$, $u_1, \ldots, u_n \in T_{\Omega/A}(X)$
- $[v[l]]_A \in \text{SP}(\Omega, L)$ if $p$ is not the empty (top) position, $l \in L$, $[v[l]]_A \in T_{\Omega(L)/A}(X)$, and $v \in T_{\Omega/A}(X)$.
- $[v[l(\phi)]_A \in \text{SP}(\Omega, L)$ if $p$ is not the empty (top) position, $l \in L$, $[v[l(\phi)]_A \in T_{\Omega(L)/A}(X)$, and $v, u_1, \ldots, u_n \in T_{\Omega/A}(X)$.

* $\Omega \subseteq \Sigma$ is the subsignature of constructors associated with the ground confluent and terminating (modulo $A$) theory $(\Sigma, E \cup A)$, where $f \in \Omega$ iff there is a ground term $t$ s.t. $f$ is a function symbol in $\text{can}_E/A(t)$.
SP(Ω, L) defines Ω(L)-terms that are spatial action patterns. Note that CanPTerms1(ℛ) ⊆ SP(Ω, L), so that any canonical one-step proof term is a ground version of some spatial action pattern.

An action pattern of the form ℓ describes a rule labeled ℓ that can be applied anywhere. An action pattern ℓ(φ) allows ℓ to also be applied anywhere, but constrains the variable instantiation related to rule ℓ to be itself a further instance of φ. Action patterns of the form top(ℓ(φ)) are needed to cover the case where ℓ is applied at the top of the term. The most fully spatial patterns are those of the form v[ℓ(φ)]p with v a nonempty context and p a position. The instance-of relation between a spatial action pattern and a proof term captures these meanings of spatial action patterns. Let [u]A ⊑ [v]A iff there is a many-sorted substitution θ such that [u]A = [θ(v)]A10. In addition, for substitution terms, let us define [φ]A ⊑ [φ]A for all [x]A ⊑ [φ]A, there exists [x]A ⊑ [φ]A such that [u]A ⊑ [v]A.

Definition 2.5 The instance-of relation between a canonical one-step proof term γ and a spatial action pattern δ ∈ SP(Ω, L), denoted γ ⊑ δ, is a slight variant of the ⊑ relation defined as follows:

• [v[ℓ(φ)]p]A ⊑ [ℓ]A
• [ℓ]A ⊑ [top(ℓ)]A

Reflection and Metalevel Computation

Rewriting logic is reflective in a precise mathematical way [7], namely, there is a finitely presented rewrite theory ℛ that is universal in the sense that we can represent in ℛ any finitely presented rewrite theory U (including U itself) as a term Ω, any terms t, t′ in ℛ as terms ¯Ω, ¯Ω, and any pair (ℛ, t) as a term (Ω, ¯Ω), in such a way that we have the following equivalence

ℛ ⊢ t −→∗ t′ ⇔ U ⊢ (Ω, ¯Ω) −→∗ (Ω, ¯Ω)

Since ℛ is representable in itself, we can achieve a “reflective tower” with an arbitrary number of levels of reflection [7,7]:

ℛ ⊢ t −→∗ t′ ⇔ U ⊢ (Ω, ¯Ω) −→∗ (Ω, ¯Ω) ⇔ U ⊢ (U, (Ω, ¯Ω)) −→∗ (U, (Ω, ¯Ω)) . . .

Key functionality of the universal theory ℛ can be controlled by descent function[7,7]. For example, in Maude, given ℛ = (Σ, E ⊔ A, R), several key constructs of ℛ are defined by the following descent functions:

• metaReduce(Ω, ¯Ω) metarepresents the E/A-canonical form of term t in ℛ. If ℛ is computable, then [t]E/A = [′]E/A iff metaReduce(Ω, ¯Ω) = metaReduce(Ω, ¯Ω).
• metaMatch(Ω, ¯Ω) metarepresents the instance-of modulo A relation between terms: t′ ⊑ A t iff metaMatch(Ω, ¯Ω) = true.

10 This is a decidable relation by our assumption that there is an A-matching algorithm.
• $\text{metaXapply}(\mathcal{R}, t, l, m)$ \footnote{This definition of $\text{metaXapply}$ describes only part of the actual function. See [?1] for the full definition.} metarepresents the $m$-th one-step rewrite of term $t$ by rule $l$ in $\mathcal{R}$. $\mathcal{R} \vdash [v[l(\phi)]_{\mathcal{R}}]_{A} : t \rightarrow t'$ iff $\exists m$ s.t. $\text{metaXapply}(\mathcal{R}, t, l, m) = (\overline{\mathcal{F}}, \overline{\mathcal{T}}[\underline{\mathcal{T}}], \overline{\mathcal{G}})$.

2.2 The Linear Temporal Logic of Rewriting

$\text{TLR}^*$ is a family of logics parameterized by the spatial actions $\text{SP}(\Omega, L)$ and the signature of atomic propositions $\Pi$. The most general of these logics is $\text{TLR}^*$, a generalization of the state-based $\text{CTL}^*$ logic that allows both spatial actions and state predicates in formulas. It contains various important sublogics of interest, which appear as special cases (see [?1]). For our model checking purposes here, where we are interested in extending with support for spatial actions the MaudeLTL model checker for rewrite theories described in [?1], we focus on $\text{TLR}$, the sublogic generalizing $\text{LTL}$, which is parameterized as $\text{LTLR}(\text{SP}(\Omega, L), \Pi)$ by the spatial actions $\text{SP}(\Omega, L)$ and the signature of state predicates $\Pi$. The following is the syntax for $\text{LTLR}$ in BNF-like style, where, instead of using $\text{CTL}^*$-like notation, we adopt the implicitly universally path quantified $\text{LTL}$ notation \footnote{We assume that all state predicate constants and function symbols are constructors, i.e., that there is a subsignature containment $\Pi \subseteq \Omega$, and then define the set $\text{Prop}(\Pi)$ of atomic propositions as the set of ground terms $\text{Prop}(\Pi) = \text{Prop}_{\text{typ}}$.}:

$$\delta : \text{SP}(\Omega, L), \ p : \text{Prop}(\Pi), \ \varphi, \varphi' : \text{LTLR}(\text{SP}(\Omega, L), \Pi)$$

$$\text{LTLR}(\text{SP}(\Omega, L), \Pi) : \delta \ | p \ | \varphi \ | \varphi' \ | \varphi \lor \varphi' \ | \varphi \land \varphi' \ | \varphi \Rightarrow \varphi' \ | \varphi \Leftrightarrow \varphi' \ | \varphi \text{W} \varphi' \ | \varphi \text{U} \varphi' \ | \varphi \text{O} \varphi \ | \varphi \text{D} \varphi$$

Smaller and useful sublogics of $\text{TLR}^*(\text{SP}(\Omega, L), \Pi)$ can be obtained by restricting the atomic propositions and/or spatial actions allowed. That is, we can define sublogics of $\text{TLR}^*$ parameterized by a subset $W \subseteq \text{SP}(\Omega, L)$ of spatial actions, and a subset $\Delta \subseteq \text{Prop}(\Pi)$ of atomic propositions. Specifically, the sublogic $\text{LTLR}(W, \Delta) \subseteq \text{LTLR}(\text{SP}(\Omega, L), \Pi)$ is defined by the set-theoretic formula $\text{LTLR}(W, \Delta) = \{ \varphi \in \text{LTLR}(\text{SP}(\Omega, L), \Pi) \ | \ \text{sp}(\varphi) \subseteq W \land \text{prop}(\varphi) \subseteq \Delta \}$, where $\text{sp}(\varphi)$ denotes the set of spatial action subformulas of $\varphi$, and $\text{prop}(\varphi)$ denotes the set of atomic proposition subformulas. In the particular, when $W = \emptyset$, and $\Delta = \text{Prop}(\Pi)$, we obtain specializations to the state-based logics $\text{LTL}$; that is, $\text{LTL}(\Pi) = \text{LTLR}(\emptyset, \Pi)$. For other specializations of this kind to various state-based and action-based logics see [?2].

The semantics of formula $\varphi \in \text{LTLR}(\text{SP}(\Omega, L), \Pi)$ is given by the satisfaction relation $\mathcal{R}, [t] \models \varphi$, where $\mathcal{R} \in \text{RWTh}_{\text{θ0}}$ has subsignatures of constructors $\Omega$ and of state predicates $\Pi$, and $[t]$ is a state (i.e., an $A$-equivalence class $[t]_{A}$ in $E$-canonical form modulo $A$ and of sort $\text{State}$, where $E \cup A$ are the equations in $\mathcal{R}$). By definition, $\mathcal{R}, [t] \models \varphi$ holds if and only if for each infinite computation $(\pi, \gamma) \in \text{Comp}(\mathcal{R})_{[t]}^\infty$ the path satisfaction relation $\mathcal{R}, ([\pi, \gamma]) \models \varphi$. Since one can express all of $\text{LTLR}(\text{SP}(\Omega, L), \Pi)$ in terms of $\text{SP}(\Omega, L)$, $\text{Prop}(\Pi)$, and the basic connectives $\top, \bot, \lor, \land, \forall$, and $\exists$, it is enough to define the semantics for the atoms and for those connectives. Since $\text{LTLR}$ generalizes $\text{LTL}$, the semantic definitions are entirely similar to those for $\text{LTL}$ (see, e.g., [?3]). The key new addition is the semantics of spatial actions; the relation $\mathcal{R}, ([\pi, \gamma]) \models \delta$ holds if and only if the proof term $\gamma(\Pi)$ of a current computation is an instance of a spatial action pattern $\delta$. 

53
The faithful mapping of tandems we seek is a mapping parametric in \( W \). Section 3.2 provides the theoretical foundations for the reflective design and implementation of the tandem RewritingLogic/TLR* to the tandem Kripke/CTL*. After briefly describing how such a mapping is mathematically defined in terms of Kripke structures in Section 3.1 (summarizing results in [?,?]), we give in Section 3.2 a new construction of the same mapping at the level of rewrite theories, as a reflective rewrite theory transformation and prove the correctness of this new construction. Section 3.2 provides the theoretical foundations for the reflective design and implementation of the Maude LTLR model checker described in Section 4.

### 3 Reflective Reduction to State-Based Temporal Logics

To make possible the use of standard CTL* (or LTL) model checkers to verify TLR* (or LTLR) properties of finite-state systems specified by rewrite theories, we need to show that there is a pair of model and formula transformations faithfully mapping the tandem RewritingLogic/TLR* to the tandem Kripke/CTL*. After briefly describing how such a mapping is mathematically defined in terms of Kripke structures in Section 3.1 (summarizing results in [?,?]), we give in Section 3.2 a new construction of the same mapping at the level of rewrite theories, as a reflective rewrite theory transformation and prove the correctness of this new construction. Section 3.2 provides the theoretical foundations for the reflective design and implementation of the Maude LTLR model checker described in Section 4.

#### 3.1 Reduction to State-Based Temporal Logics

The faithful mapping of tandems we seek is a mapping parametric in \( W \)

\[
(K_W, \circledast) : \text{RewritingLogic/TLR}^*(W, \Pi) \rightarrow \text{Kripke/CTL}^*(\Pi \cup W)
\]

with \( W \) a finite set of spatial actions in the given rewrite theory \( R \), and \( \Pi \) the subsignature of state predicates in \( R \).

**Definition 3.1** Given \( R \in RWTh_0 \) and a finite set \( W \subseteq \text{SP}(\Omega, L) \), the construction \( K_W \) maps the rewrite theory \( R \) to the following Kripke structure \( K_W(R) \):

(i) Its set of states is \( (\text{Can}_{\Sigma/E, A})_{\text{State}} \times \mathcal{P}(W) \)

(ii) Its transition relation is defined by the equivalence: \( ([t], U) \rightarrow ([t'], V) \) iff there is a canonical one-step rewrite proof \([t] \xrightarrow{\gamma} [t]\) in \( R \), and \( V \) is the set \( \text{act}_W(\gamma) = \{ \delta \in W \mid \gamma \subseteq A \delta \} \).

\(^{13}\) Since any TLR* formula \( \xi \) only involves a finite set \( \text{sp}(\xi) \) of spatial actions as subformulas, it is always sufficient to consider formulas in TLR*(\( W, \Pi \)) with \( W \) finite.

\(^{14}\) Recall that a Kripke structure on a set \( \text{AP} \) of atomic propositions is a triple \( K = (A, R, L) \), with \( A \) a set of states, \( R \subseteq A \times A \) a total transition relation, and \( L : A \rightarrow \mathcal{P}(\text{AP}) \) a labeling function assigning to each state \( a \in A \) the set \( L(a) \subseteq \text{AP} \) of the atomic propositions that hold in \( a \).
can be achieved using the \textit{metaMatch} built-in implementation of \(\mathcal{U}\)'s sorts and descent functions, this can be achieved using the \texttt{metaMatch} descent function, which meta-represents the

\begin{equation}
\text{(iii) Its set of atomic propositions is the set } \text{Prop}(\Pi) \cup W, \text{ and the labeling function maps a state } ([t], U) \text{ to the set of atomic propositions } \mathcal{L}_\mathcal{R}([t]) \cup U, \text{ where, by definition, } \mathcal{L}_\mathcal{R}([t]) = \{p \in \text{Prop}(\Pi) \mid \text{can}_{E/A}([t]) \models p = \text{true}\}.
\end{equation}

By the above definition, \(([t], U) \models \delta\) if and only if a spatial action pattern \(\delta \in U\), where \([t]\) is a state of \(\mathcal{R}\) and \(U \subseteq \mathcal{W}\). Since the condition (ii) asserts that \(U\) is the set of all spatial action patterns of which the one-step proof term of a current computation is an instance, this coincides with the semantics of \(\mathcal{R}\), \((\pi, \gamma) \models \delta\) defined in Section 2.2.

**Definition 3.2** Given a formula \(\varphi \in TLR^*(W, \Pi)\) we can map it to the formula \(\bar{\varphi} \in CTL^*(W \cup \Pi)\) by systematically replacing each occurrence of a spatial action \(\delta \in W\) in \(\varphi\) by the formula \(X\delta\).

The construction \(\mathcal{K}_W(\mathcal{R})\), with the above formula translation (\(\bar{\cdot}\)), defines a mapping of tandems \((\mathcal{K}_W, \bar{\cdot}) : \text{RewritingLogic/} TLR^*(W, \Pi) \rightarrow \text{Kripke/} CTL^*(W \cup \Pi)\). This mapping is a faithful mapping of tandems preserving the satisfaction relations \(\models \) in \(TLR^*(W, \Pi)\) and \(\models_{CTL^*}\) in \(CTL^*(W \cup \Pi)\). This is shown by the following theorem proved in detail in [?], where a complexity-theoretic analysis of the reduction is also given.

**Theorem 3.3** Given a rewrite theory \(\mathcal{R} \in RWTh_0\) and a finite \(W \subseteq SP(\Omega, L)\), for each state \([t]\) \(\in \mathcal{R}\), \(U \subseteq W\), and \(\varphi \in TLR^*(W, \Pi)\), the following equivalence holds:

\begin{equation}
\mathcal{R}, [t] \models \varphi \iff \mathcal{K}_W(\mathcal{R}), ([t], U) \models_{CTL^*} \bar{\varphi}.
\end{equation}

### 3.2 The \(\mathcal{K}_W\) Construction as a Reflective Theory Transformation

The \(\mathcal{K}_W(\mathcal{R})\) construction maps each rewrite theory in \(RWTh_0\) to a Kripke structure. However, as pointed out in [?], it is very useful to decompose the mapping \(\mathcal{R} \rightarrow \mathcal{K}_W(\mathcal{R})\) into a rewrite theory transformation \(\mathcal{R} \rightarrow \mathcal{R}_W\) followed by the general construction \(\mathcal{R} \rightarrow \mathcal{K}(\mathcal{R})\), spelled out in detail in [?, ?], which maps each \(RWTh_0\) to its underlying Kripke structure. That is, we can decompose the \(\mathcal{K}_W(\mathcal{R})\) construction as \(\mathcal{K}_W(\mathcal{R}) = \mathcal{K}(\mathcal{R}_W)\). This is particularly useful for tool building purposes, since the construction \(\mathcal{R} \rightarrow \mathcal{K}(\mathcal{R})\) is already automated in Maude’s LTL model checker [?]. So, “essentially” all we need to do is to automate the theory transformation \(\mathcal{R} \rightarrow \mathcal{R}_W\). Here is where the reflective properties of rewriting logic summarized in Section 2.1 and efficiently supported in Maude by the META-LEVEL module become extremely useful. In what follows we describe in detail and prove correct a reflective, parametric construction for the \(\mathcal{R} \rightarrow \mathcal{R}_W\) transformation in which the parameter \(\mathcal{R}\) is metarepresented as a term \(\overline{\mathcal{R}}\) in the universal rewrite theory \(\mathcal{U}\). We can then obtain the \(\mathcal{R} \rightarrow \mathcal{R}_W\) construction as a reflective parametric rewrite theory \(\mathcal{U}(\underline{\cdot})\) which, when instantiated with parameters \(\overline{W}\) and \(\overline{\mathcal{R}}\), yields a theory \(\mathcal{U}_W(\mathcal{R})\) that extends \(\mathcal{U}\) and provides a correct realization of the theory \(\mathcal{R}_W\). We explain in detail the parametric \(\mathcal{U}_W(\mathcal{R})\) construction in what follows.

Note that, in particular, the \(\mathcal{U}_W(\mathcal{R})\) construction has to metarepresent the instance-of relation \(\subseteq_{A}\), between a one-step proof term and a spatial action pattern. In the META-LEVEL built-in implementation of \(\mathcal{U}\)’s sorts and descent functions, this can be achieved using the \texttt{metaMatch} descent function, which meta-represents the
instance-of relation between terms. Since metaMatch is parametric on each rewrite theory, its metarepresentation is also parametric on the given rewrite theory. Let \( \mathcal{\Gamma}_\mathcal{R} \) denote the metarepresentation of the instance-of relation between the meta representations of a one-step proof term and a spatial action pattern in the equational theory \( (\Omega(L), A) \) associated with the rewrite theory \( \mathcal{R} = (\Sigma, E \cup A, R) \) in Section 2.1. Then, for a one-step proof term \( \gamma \) and a spatial action pattern \( \delta \), we have \( \gamma \sqsubseteq_A \delta \iff \gamma \sqsubseteq_{\mathcal{\Gamma}_\mathcal{R}} \delta \). The \( \mathcal{U}_W(\mathcal{R}) \) construction contains the universal theory \( \mathcal{U} \) and can then be defined as follows.

**Definition 3.4** The rewrite theory \( \mathcal{U}_W(\mathcal{R}) \) is the following parametric extension of the universal theory \( \mathcal{U} \), in which \( \mathcal{R} = (\Sigma, E \cup A, R) \) and \( W \subset SP(\Omega, L) \) are metarepresented as data parameters \( \overline{\mathcal{R}} \) and \( \overline{W} \):

- \( \mathcal{U} \subset \mathcal{U}_W(\mathcal{R}) \), where \( \mathcal{U}_W(\mathcal{R}) \) includes all the descent functions in the \textsc{Meta-Level} module as well as a descent function for the instance-of relation \( \gamma \sqsubseteq_{\mathcal{\Gamma}_\mathcal{R}} \delta \).
- \( \mathcal{U}_W(\mathcal{R}) \) has sorts State, Prop, and Bool with constants true and false, where the ground terms of sort State are pairs \( (\overline{t}, \overline{U}) \), with \( t \) a term of sort State in \( \mathcal{R} \), and \( U \subseteq W \) (that is, \( \overline{U} \) is a meta-term that uses a set union associative and commutative operator to represent a finite set of action patterns contained in \( W \)).
- \( \mathcal{U}_W(\mathcal{R}) \) has also an \( \text{act} \) operator with \( \text{act}(\overline{W}, \gamma) = \{ \overline{\delta} \in \overline{W} \mid \gamma \sqsubseteq_{\mathcal{\Gamma}_\mathcal{R}} \overline{\delta} \} \).
- There is a single conditional rewrite rule \( tr \) in \( \mathcal{U}_W(\mathcal{R}) \) such that \( tr : (\overline{t}, \overline{U}) \longrightarrow (\text{metaReduce}(\overline{R}, \overline{t}, \overline{V}), \overline{V}) \) iff there exists a rule label \( l \) in \( \mathcal{R} \) and a natural number \( m \) such that \( \text{metaApply}(\overline{R}, \overline{t}, l, m) = (\overline{t}, \overline{V}[p]_l, \overline{\phi}) \land \overline{V} = \text{act}(\overline{W}, \overline{t}[l(\overline{\phi})]_p) \). The rule’s condition can be equationally expressed using “matching conditions” with extra variables (see [?]). Note that in particular this means that we have a one-step rewrite proof term \( \mathcal{R} \vdash v[l(\overline{\phi})]_p : t \longrightarrow t' \).
- The ground terms of sort Prop are precisely the metarepresentations of: (i) either the atomic propositions of \( \mathcal{R} \), or (ii) the elements of \( W \).
- There is a labeling operator \( \_ \models \_ : \text{State} \times \text{Prop} \rightarrow \text{Bool} \) with conditional equations such that:
  - \( (\overline{t}, \overline{U}) \models \overline{\delta} = \text{true} \iff \delta \in U \)
  - \( (\overline{t}, \overline{U}) \models \overline{p} = \text{true} \iff \text{metaReduce}(\overline{R}, t \models p) = \overline{\text{true}} \), i.e., \( \overline{\text{can}_{E/A}}(t \models p) = \text{true} \).

The theory \( \mathcal{U}_W(\mathcal{R}) \) defined above is meta-level description for \( \mathcal{K}_W(\mathcal{R}) \), and so the definition is closely related to Definition 3.1. The correctness of the \( \mathcal{U}_W(\mathcal{R}) \) construction is expressed by the following proposition.

**Proposition 3.5** Given a rewrite theory \( \mathcal{R} \in \mathcal{RW}th_0 \) and a finite set \( W \subseteq SP(\Omega, L) \), for each state \( [t] \) in \( \mathcal{R} \) and \( U \subseteq W \), the following conditions hold:

1. states ([t], U) of \( \mathcal{K}_W(\mathcal{R}) \) are in one-to-one correspondence with ground terms of sort State of the form \( \overline{\text{can}_{E/A}(t), U} \) in \( \mathcal{U}_W(\mathcal{R}) \).
2. For each atomic proposition \( \alpha \) of \( \mathcal{K}_W(\mathcal{R}) \), ([t], U) \models \alpha \iff \overline{\text{can}_{E/A}(t), U} \models \overline{\alpha} = \text{true} \) in \( \mathcal{U}_W(\mathcal{R}) \).
3. There is a transition \( ([t], U) \rightarrow^* ([t'], V) \) in \( \mathcal{K}_W(\mathcal{R}) \) iff \( \mathcal{U}_W(\mathcal{R}) \vdash (\overline{\text{can}_{E/A}(t), U}) \rightarrow (\overline{\text{can}_{E/A}(t'), V}) \).

56
Proof (Sketch)

(i) By construction, the terms of sort State in the reflective theory $U_W(R)$ extending $U$ are pairs $(\text{can}_{E/A}(t), U)$ metarepresenting states $([t], U)$ of $K_W(R)$. This metarepresentation is one-to-one because of the computability assumptions about $R$ (which include confluence and termination of $E$ modulo $A$).

(ii) There are two cases when $([t], U) \models \alpha$ is true in $K_W(R)$; $\alpha \in U$, or $\text{can}_{E/A}([t] \models \alpha) = \text{true}$. Since $\alpha \in U$ iff $\alpha \in U$ and $\text{can}_{E/A}([t] \models \alpha) = \text{true}$ iff $\text{metaReduce}(R, \text{can}_{E/A}(t) \models \alpha) = \text{true}$, in both cases, by the above definition, $([t], U) \models \alpha$ iff $(\text{can}_{E/A}(t), U) \models \alpha$.

(iii) If a transition $([t], U) \to ([t'], U)$ exists in $K_W(R)$, then, by definition, there exists a one-step rewrite $[t] \xrightarrow{\ell} [t']$ in $R$ and $V = \text{act}_W(U)$. But, by the definition of $U_W(R)$, this is equivalent to the existence of a one-step rewrite $tr : (t, U) \to (\text{metaReduce}(R, \ell), V)$ which can happen if and only if there exists a rule label $\ell$ in $R$ and a natural number $m$ such that $\text{metaApply}(R, t, \ell, m) = (t', \sigma[p], \phi)$ and $V = \text{act}(W, v[t[\phi]])_p$.

\[ \square \]

In addition to the $U_W(R)$ construction, we need to deal with the metarepresentation of the LTL formulas $\tilde{\varphi}$ associated to LTLR formulas.

**Definition 3.6** Given a formula $\varphi$, $\tilde{\varphi}$ is the same as $\tilde{\varphi}$, except that each atomic proposition (resp. a spatial action pattern) is replaced by its meta-representation.

4 Reflective Design of an LTLR Model Checker

Maude supports rewriting based LTL model checking for any computable rewrite theory $R[?,?]$, that is, Maude automates the $R \mapsto K(R)$ construction and provides an LTL model checker for the underlying Kripke structure $K(R)$. Therefore, the reduction method described in Section 3.1 to reduce model checking of LTLR formulas to that of LTL formulas can be used, thanks to the reflective $U_W(R)$ construction described in Section 3.2, to reduce the model checking of $R$ with respect to a formula $\varphi \in LTLR(W, \Pi)$ to performing LTL model checking on the rewrite theory $U_W(R)$ with respect to the formula $\tilde{\varphi}$.

In a practical model checking system implementation, however, we want to hide all metalevel representations from the user; therefore the reflective $U_W(R)$, while being the core of the model checker, is not what the user directly interacts with. In particular, we should support model checking commands in which the LTLR formula and the initial state are specified at the object level. This requires the design of a suitable user interface in addition to the internal reflective commands that perform the actual model checking. All this can be achieved, as we explain in this section, by extending the Full Maude language [?] using reflective methods.

4.1 Constructing the LTLR Syntax with Spatial Action Patterns

In the LTLR syntax, the atomic propositions and the spatial action patterns in formulas are not fixed: they depend parametrically on the given computable rewrite
theory $\mathcal{R}$. In particular, spatial action patterns can be uniquely characterized as soon as $\mathcal{R}$ is specified. The idea is that given $\mathcal{R} = (\Sigma, E, R)$ with a subsignature of constructor $\Omega$ and rule labels $L$, we can define a map

$$\text{TLR} : \mathcal{R} \to \mathcal{R} \cup SP(\Omega, L),$$

and this map can be metarepresented as an equationally defined function

$$\text{TLR} : \mathcal{R} \to \mathcal{R} \cup SP(\Omega, L)$$

in an extension of the META-LEVEL module. The function $\text{TLR}$ can be integrated into Full Maude [?], and by extending Full Maude’s parser the user can then specify at the object level LTLR properties using the parametric LTLR syntax in $\text{TLR}(\mathcal{R})$. The remaining thing we need to do is to define $SP(\Omega, L)$ as a rewrite theory. It is not hard, since we can just use the definition of $\Omega(L)$ and Definition 2.4. The slightly tricky part is to make actions $v[l(\phi)]_p$ include only one basic action term $l(\phi)$. It is archived by adding the operators

$$o : \text{Action} - A_1 \times A_2 \times A_3 \times \cdots \times A_n \to \text{Action} - A$$

$$o : A_1 \times \text{Action} - A_2 \times A_3 \times \cdots \times A_n \to \text{Action} - A$$

$$\vdots \quad \vdots \quad \vdots$$

$$o : A_1 \times A_2 \times \cdots \times A_{n-1} \times \text{Action} - A_n \to \text{Action} - A$$

for each operator $o : A_1 \times \cdots \times A_n \to A \in \Omega$ with $n \geq 1$, where $\text{Action} - A_i$ is a related action sort for each sort $A_i$. Since to yield a spatial action pattern, operators in $\Omega$ can have only one action sub-pattern, the uniqueness of a basic action term $l(\phi)$ as a sub-term is automatically guaranteed.

4.2 Reflective LTLR Model Checker Design

Given a computable rewrite theory $\mathcal{R} \in RWTh_0$, we are now able to define LTLR properties using $\text{TLR}(\mathcal{R}) \cup \Pi$, where $\Pi$ is the set of atomic propositions in $\mathcal{R}$. The next step is to transform the LTLR properties into LTL properties. This is accomplished by the map $\overline{\varphi} \mapsto \overline{\varphi}$ associated to Definition 3.6, which can be constructed by reflection using the metarepresented theory $\text{TLR}(\mathcal{R}) \cup \Pi$. Then, when a user gives to the LTLR Model Checker system extending Full Maude the command

$$(\text{tlr check}(t, \varphi))$$

the system performs the followings tasks: (i) $t$ and $\varphi$ are parsed and metarepresented as $\overline{t}$ and $\overline{\varphi}$; (ii) the set $W = sp(\overline{\varphi})$ is computed, and the translation $\overline{\varphi} \mapsto \overline{\varphi}$ is performed; (iii) the Maude LTL model checker is invoked at the metalevel on the theory $U_W(\mathcal{R})$ with initial state $(\overline{t}, \emptyset)$ and LTL formula $\overline{\varphi}$; and (iv) the output of the model checking results (either true or a counterexample) are provided to the user at the object level using Full Maude’s meta-pretty printing features. By the results in Section 3, we are then guaranteed (assuming correctness of the Maude and Full Maude implementations and of the reflective model checker implementation) that the model checker system will answer true to a model checking command $(\text{tlr check}(t, \varphi))$ if and only if $\mathcal{R}, t \models \varphi$. 
5 The Example Revisited

This section illustrates the use of the above \textit{LTLR} model checker with Dekker’s algorithm example explained in Section 1.1. A similar definition of this parallel language and the algorithm appeared in [?,?], but since only \textit{LTL} model checking was available in [?,?], there this required manual “cooking” both the rewrite theory and the formulas, as explained in Subsection 1.1. Instead, here everything can be specified and model checked in the most natural way. The code of algorithm and the system description are the same as before. The global states are of the form \{\{I, R\} | S, M\}, with \{I,R\} a process with id I and code R, S the set of remaining processes, and M the memory. The rewrite rules defining the language’s operational semantics are all labeled with the same label \texttt{stmt}. Hence, the spatial action pattern \texttt{stmt(\textbackslash I\ p1)} asserts the execution of process \texttt{p1}. The predicates \texttt{in-crit} and \texttt{in-rem}, discussed in Section 1.1, are equationally defined as follows.

\begin{verbatim}
(mod DEKKER-CHECK is protecting TLR[DEKKER] .
  subsort MachineState < State .
  ops in-crit in-rem : Pid -> Prop .
  op exec : Pid -> Action .
  var M : Memory . vars R : Program . var S : Soup . vars J : Pid .
  eq exec(J) = [stmt(I\ J)] .
  eq {{J, crit : R} | S, M} |= in-crit(J) = true .
  eq {{J, R} | S, M} |= in-crit(J) = false [owise] .
  eq {{J, rem : R} | S, M} |= in-rem(J) = true .
  eq {{J, R} | S, M} |= in-rem(J) = false [owise] .
endm)
\end{verbatim}

We can then check that the strong fairness property fails, and the model checker returns a counterexample. The result is translated to the same format as that of the \textit{LTL} model checker.

\begin{verbatim}
Maude> (tlr check(initial, [\]<> exec(1) -> [\]<> in-crit(1)) .)
result : counterexample({{{[1,repeat 'c1 := 1 ; while 'c2 = 1 do ...}}
\end{verbatim}

Instead, the (somewhat subtle) weaker fairness property satisfied by Dekker’s algorithm can be verified as follows.

\begin{verbatim}
Maude> (tlr check(initial, [\]<> exec(1) /\ [\]<> exec(2) ->
      [\]<> ~ in-rem(1) -> [\]<> in-crit(1)) .)
result : true
\end{verbatim}

If a given property is a \textit{LTL} formula, the model checker do \textit{LTL} model checking without transformation, such as the \textit{mutual exclusion property} like the following:

\begin{verbatim}
Maude> (tlr check(initial, [\] ~ (in-crit(1) /\ in-crit(2))) .)
ltl-result : true
\end{verbatim}

\footnote{For a collection of other examples and the code of the \textit{LTLR} model checker see the web page [?].}
6 Related Work and Conclusions

There is much related work on both state-based and action-based logics (see [?]) for a more thorough discussion). Related temporal logics include: (i) state-based logics; (ii) action-based logics; and (iii) mixed logics supporting both actions and state predicates. Well-known state-based logics such as LTL, CTL, and $CTL^*$ (see, e.g., [?], [?]), are all special cases of $TLR^*$. Faithful translations to $TLR^*$ from well-known action-based logics such as Hennessy-Milner logic [?] and $A-CTL^*$ [?] are defined in detail in [?]. The mixed action-state logic $SE-LTL$ in [?,?] can also be viewed as special case of the $TLR^*$. The Spatial Logic for Concurrency of Caires and Cardelli [?,?] is a state-based spatial modal logic for process calculi in the $\pi$-calculus spirit with spatial features used only for state predicates (but see the logic in [?]), which has an action-labeled diamond). Other action-based temporal and modal logics are discussed in the survey paper [?], including the modal $\mu$-calculus [?] ($\mu L$), which is in some ways more powerful than $TLR^*$, but lacks spatial action patterns. Among other logics supporting both actions and state predicates we find several extensions of either $A-CTL^*$ or $A-CTL$ such as, e.g., [?,?.?,?.?]. Three other approaches proposing mixed logics with both state-predicates and actions are: (i) the extension of the $SE-LTL$ in [?,?] to a universally path quantified logic involving $\omega$-regular expressions [?]; (ii) the $ESTL$ logic of events and states for Petri nets of [?]; and (iii) the Kripke modal transition systems of [?], and their use in the verification of safety and liveness properties in the context of the modal $\mu$-calculus.

The work most closely related to $TLR^*$ is that on $VLRL$ [?,?]. The $VLRL$ solution was less general and did not consider model checking aspects. Two other logics that combine actions and state-based formulas are the UNITY logic of Chandy and Misra [?], and Misra’s logic for Seuss [?]; however, actions as such do not appear in temporal logic formulas, which remain state-based. Methodologically and technically, Lamport’s Temporal Logic of Actions ($TLA$) [?] and $TLR^*$ are very different. In $TLA$, there is no division of labor between system and property specification logics: $TLA$ plays both roles simultaneously [?]. Also, actions in $TLA$ are interpreted as binary relations between states, so that one cannot distinguish between two actions having the same outcomes from a given state.

In conclusion, after reviewing background on rewriting logic and the temporal logic of rewriting we have presented the formal foundations of the Maude $LTLR$ model checker and explained its reflective implementation as an extension of the Full Maude language. We have also presented an example illustrating the use of the model checker. The $LTLR$ model checker presented here is a useful prototype that can be used for both teaching and research and that, as mentioned, is available at [?]. The theoretical complexity results given in [?] mean that in practice the growth in the number of states is reasonable. Also, thanks to the efficient implementation of reflective features in Maude, our experience with examples suggest that the performance of the $LTLR$ model checker is acceptable, so that it can be used in both teaching and experimental research. In particular, for pure $LTL$ formulas it has essentially the same performance as that of Maude’s $LTL$ model checker, which is quite good.

We believe, however, that a greater efficiency, obviating even the need for any
growth in the state space of the given rewrite theory $\mathcal{R}$, and avoiding any reflective computations can be achieved by a new, native $LTLR$ model checking algorithm. However, the design of such an algorithm and its implementation in Core Maude are both nontrivial tasks than have to be left for future research.
Neural Networks in Maude

Gustavo Santos-García
Universidad de Salamanca

Miguel Palomino
Departamento de Sistemas Informáticos y Computación, UCM

Alberto Verdejo
Departamento de Sistemas Informáticos y Computación, UCM

Abstract
In this work we study the representation of the computational model of artificial neural networks in rewriting logic, along the lines of several models of parallelism and concurrency that have already been mapped into it. We show how crucial is the right choice for the representation operations and the availability of strategies to guide the application of our rules. Finally, we also apply our specification to data used in the diagnosis of glaucoma.

Keywords: Neural networks, rewriting logic, Maude, strategies, executability.

1 Introduction
Rewriting logic [11] is a logic of concurrent change that can naturally deal with states and with highly nondeterministic concurrent computations. It has good properties as a flexible and general semantic framework for giving semantics to a wide range of languages and models of concurrency. Indeed, rewriting logic was proposed as a unifying framework in which many models of concurrency could be represented, such as labeled transition systems, phrase structure grammars, Petri nets, concurrent object-oriented programming, or CCS, to name a few. For many of these models, concrete maps have actually been defined into rewriting logic; see e.g. [15,14,12,7,16] and the references in [8].

* Research supported by Spanish project DESAFIOS TIN2006–15660–C02–01 and by Comunidad de Madrid program PROMESAS S-0506/TIC/0407.

Email: santos@usal.es
Email: miguelpt@sip.ucm.es
Email: alberto@sip.ucm.es

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs
Artificial neural networks [6] are another important model of parallel computation. This model attempts to imitate the operation of biological neural networks. In [10] it was argued that rewriting logic was also a convenient framework in which to embed artificial neural nets, and a possible representation was sketched. However, and to the best of our knowledge, no concrete map has ever been constructed either following those ideas or any others. Our goal with this paper is to fill this gap.

In our representation of artificial neural networks we consider two stages: a first one, in which we are only concerned with the evaluation of patterns by the network, and a second one, in which the output produced by those patterns is used to adapt (or train) the net so as to make it more precise (that is, so that the error between the actual network output and the target output provided by the training set is reduced). It turns out that while the representation suggested in [10] allows a straightforward implementation of the first stage, we were unable to make use of it for the training phase. Therefore, we propose an alternative representation in which the specification of both stages proceeds smoothly.

Since its conception, rewriting logic was proposed as the foundation of an efficient executable system called Maude [1]. Here we write our representation directly in Maude to be able to run our neural networks and apply them to a real case-study, that is the analysis of campimetric fields and nerve fibres of the retina for the diagnosis of glaucoma on patients from the University Hospital in Salamanca [3].

The paper is organized as follows. In Section 2 we review those aspects of Maude that will be used in the specification of artificial neural nets, mainly object-oriented modules and strategies, to make the paper (almost) self-contained. Section 3 introduces multilayer perceptron nets, their specification in Maude, and an appropriate strategy for their evaluation. The backpropagation algorithm for neural network training is presented in Section 4, together with the changes necessary in the specification of the network and a training strategy. Section 5 discusses the application of our implementation to the study of the diagnosis of glaucoma and Section 6 concludes. The Maude code, the data used for the examples, and the output can be downloaded from http://maude.sip.ucm.es/~miguelpt/.

2 Maude

Maude [1] is a high performance language and system supporting both equational and rewriting logic computation for a wide range of applications. The key novelty of Maude is that besides efficiently supporting equational computation and algebraic specification it also supports rewriting logic computation. Mathematically, a rewrite rule has the form \( l : t \rightarrow t' \) if Cond with \( t, t' \) terms of the same kind which may contain variables. Intuitively, a rule describes a local concurrent transition in a system: anywhere a substitution instance \( \sigma(t) \) is found, a local transition of that state fragment to the new local state \( \sigma(t') \) can take place.

Full Maude [1] is an extension of Maude with a richer module algebra of parameterized modules and module composition operations and with special syntax for object-oriented specifications. These object-oriented modules have been exploited for specifying artificial neural networks.
2.1 Object oriented modules

An object in a given state is represented as a term \( < O : C | a_1 : v_1, \ldots, a_n : v_n > \) where \( O \) is the object’s name, belonging to a set \( Oid \) of object identifiers, \( C \) is its class, the \( a_i \)’s are the names of the object’s attributes, and the \( v_i \)’s are their corresponding values. Messages are defined by the user for each application.

In a concurrent object-oriented system the concurrent state, which is called a configuration, has the structure of a multiset made up of objects and messages that evolves by concurrent rewriting (modulo the multiset structural axioms of associativity, commutativity, and identity) using rules that describe the effects of communication events between some objects and messages. The rewrite rules in the module specify in a declarative way the behavior associated with the messages.

The general form of such rules is

\[
M_1 \ldots M_n \langle O_1 : F_1 | atts_1 \rangle \ldots \langle O_m : F_m | atts_m \rangle \\
\rightarrow \langle O_{i_1} : F'_{i_1} | atts'_{i_1} \rangle \ldots \langle O_{i_k} : F'_{i_k} | atts'_{i_k} \rangle \\
\langle Q_1 : D_1 | atts''_1 \rangle \ldots \langle Q_p : D_p | atts''_p \rangle \\
M'_1 \ldots M'_q \\
\text{if } Cond
\]

where \( k, p, q \geq 0 \), the \( M_i \)s are message expressions, \( i_1, \ldots, i_k \) are different numbers among the original 1, \ldots, m, and \( Cond \) is a rule condition. The result of applying a rewrite rule is that the messages \( M_1, \ldots, M_n \) disappear; the state and possibly the class of the objects \( O_{i_1}, \ldots, O_{i_k} \) may change; all the other objects \( O_j \) vanish; new objects \( Q_1, \ldots, Q_p \) are created; and new messages \( M'_1, \ldots, M'_q \) are sent.

Since the above rule involves several objects and messages in its lefthand side, we say that it is a synchronous rule. It is conceptually important to distinguish the special case of rules involving at most one object and one message in their lefthand side. These rules are called asynchronous and have the form

\[
(M) \langle O : F | atts \rangle \\
\rightarrow (\langle O : F' | atts' \rangle) \\
\langle Q_1 : D_1 | atts''_1 \rangle \ldots \langle Q_p : D_p | atts''_p \rangle \\
M'_1 \ldots M'_q \\
\text{if } Cond
\]

By convention, the only object attributes made explicit in a rule are those relevant for that rule. In particular, the attributes mentioned only in the lefthand side of the rule are preserved unchanged, the original values of attributes mentioned only in the righthand side of the rule do not matter, and all attributes not explicitly mentioned are left unchanged.

63
2.2 Maude’s strategy language

Rewrite rules in rewriting logic need to be neither confluent nor terminating. This theoretical generality requires some control when the specifications become executable, because it must be ensured that the rewriting process does not go in undesired directions and eventually terminates. Maude’s strategy language can be used to control how rules are applied to rewrite a term [9,2]. Strategies are defined in a separate module and are run from the prompt through special commands.

The simplest strategies are the constants idle, which always succeeds by doing nothing, and fail, which always fails. The basic strategies consist of the application of a rule (identified by the corresponding rule label) to a given term, and with the possibility of providing a substitution for the variables in the rule. In this case a rule is applied anywhere in the term where it matches satisfying its condition. When the rule being applied is a conditional rule with rewrites in the conditions, the strategy language allows to control how the rewrite conditions are solved by means of strategies. An operation top to restrict the application of a rule just to the top of the term is also provided. Basic strategies are then combined so that strategies are applied to execution paths. Some strategy combinators are the typical regular expression constructions: concatenation (\texttt{;}), union (\texttt{|}), and iteration (* for 0 or more iterations, + for 1 or more, and ! for a ‘repeat until the end’ iteration). Another strategy combinator is a typical ‘if-then-else’, but generalized so that the first argument is also a strategy. By using this combinator, we can define many other useful strategy combinators as derived operations: for example a binary \texttt{orelse} combinator that applies the second argument strategy only if the first fails, and a unary \texttt{not} combinator that fails when its argument is successful and vice versa. The language also provides a \texttt{matchrew} combinator that allows a term to be split in subterms, and specifies how these subterms have to be rewritten. Moreover, an extended \texttt{matchrew} combinator, that is \texttt{amatchrew}, is also provided to support rewriting modulo axioms of associativity, commutativity, identity, and idempotency is considered, when declared. Recursion is also possible by giving a name to a strategy expression and using this name in the strategy expression itself or in other related strategies.

For our implementation, the full expressive power of the strategy language will not be needed and all our strategies will be expressed as combinations of the application of certain rules (possibly instantiated), concatenation (\texttt{;}), and ‘repeat until the end’ iteration (!). For efficiency reasons, we have extended the previous strategy language with a new combinator \texttt{one}(S) which, when applied to a term \(t\), returns one of the possible solutions of applying \(S\) to \(t\). However, our use of this combinator together with the ! combinator guarantees that no final solution is lost.

3 Specification of neural networks

Artificial neural network models have been extensively studied and applied in recent years in the hope of reaching human performance in different fields, including, for instance, automatic speech recognition, image processing, and biomedical applications [3,5,6,17]. Models of artificial neural networks are made up of many nodes or computational elements, called “neurons”, which work in parallel and are con-
nected to each other by “weights”—transmission coefficients—that are usually updated throughout the computation in order to increase the precision of the network diagnoses.

Among the different models of neural networks we have chosen to work with multilayer perceptrons. Perceptrons are classifiers that learn to build complex separating hyperplanes that partition the input space in decision regions based on the information provided during training; a multilayer perceptron with one hidden layer with step transfer functions can solve any problem with arbitrary decision regions [18].

3.1 Multilayer perceptron networks

A neural network is defined in mathematical terms as a graph with the following properties: (1) each node $i$, called neuron, is associated with a state variable $x_i$ storing its current output; (2) each junction between two neurons $i$ and $k$, called synapse or link, is associated with a real weight $\omega_{ik}$; (3) a real threshold $\theta_i$, called activation threshold, is associated with each neuron $i$; (4) a transfer function $f_i(n_k, \omega_{ik}, \theta_i, (k \neq i))$ is defined for each neuron, and determines the activation degree of the neuron as a function of its threshold, the weights of the input junctions and the outputs $n_k$ of the neurons connected to its input synapses. In our case, the transfer function has the form $f(\sum_k \omega_{ik}n_k - \theta_i)$, where $f(x)$ is, in our case, a sigmoidal function, defined by $f(x) = 1/(1 + e^{(\nu-x)})$, which corresponds to the continuous and derivable generalization of the step function.

Multilayer perceptrons are networks with one or more layers of nodes between the layer of input units and the layer of output nodes; Figure 1 shows a three-layer perceptron. These layers contain hidden units or nodes which obtain their input from the previous layer and output their results to the next layer, to both of which they are fully-connected. Nodes within each layer are not connected and have the same transfer function. The strength of the multilayer perceptron originates from the use of non-linear sigmoidal functions in the nodes. If the nodes were linear elements, then monolayer networks with appropriately selected weights could repeat the calculations carried out by a multilayer network [18]. A multilayer perceptron with a non-linear step function and a hidden layer can solve problems in which the decision regions are open or closed convex regions. In the case of perceptrons with one hidden layer, problems with arbitrary decision regions can be solved, but more complex regions will need a greater number of nodes in the network [4,5].
The accuracy of the multilayer perceptron depends basically on the correct learn-
ing of the connection weights between nodes. The backpropagation training algo-
rithm is an algorithm which uses a gradient descent method to minimize the mean
quadratic error between the actual output of the perceptron and the desired out-
put. In this section we focus on specifying three-layers perceptrons in Maude and
designing a strategy for evaluation; we consider the backpropagation algorithm in
Section 4.

3.2 Evaluation of perceptrons

The core of our representation of perceptrons in Maude revolves around the defini-
tion of two object-oriented classes to represent neurons and synapses as individual
objects.

```plaintext
class Neuron | x : Float, t : Float, st : Nat .
class Link | w : Float, st : Nat .
```

Each neuron object carries its current activation value \( x \), depending on its threshold
\( t \), and an attribute \( st \) that will be used to determine whether the neuron has already
fired or not, that is, whether it is still waiting for input or has already output a value.
Similarly, synapse objects store their numerical weight and contain an attribute \( st \)
to flag whether some value has already passed through them or not. A net then is
a “soup” (a multiset) of neurons and synapses.

Neurons and links are identified by a name. We define two operations that take
natural numbers as arguments and return an object identifier: for neurons, the
numbers correspond to the layer and the position within the layer; for links, the
numbers correspond to the output layer and the respective positions within each
layer of the neurons connected.

```plaintext
op neuron : Nat Nat -> Oid .
op link : Nat Nat Nat -> Oid .
```

The evaluation of the network is essentially performed by repeated application
of the following two rules:

```plaintext
rl [feedForward] :< neuron( L, I) : Neuron | x : X1 , st : 1 >
< neuron(s L, J) : Neuron | x : X2 , st : 0 >
< link(s L, I, J) : Link | w : W , st : 0 >
=> < neuron( L, I) : Neuron | x : X1 , st : 1 >
< neuron(s L, J) : Neuron | x : (X2 + (X1 * W)) , st : 0 >
< link(s L, I, J) : Link | w : W , st : 1 > .
```

```plaintext
rl [sygmoid] :
< neuron(L, I) : Neuron | x : X , t : T , st : 0 >
```

Rule \texttt{feedForward} calculates the weighted sum of the inputs to the neuron, whereas
\texttt{sygmoid} just applies the sigmoidal function \texttt{syg} (defined somewhere else in the code)
to the net input.

As can be seen in \texttt{feedForward}, the attribute \texttt{st} of links is assumed to be 0
prior to their firing and becomes 1 once the information has been sent from one
neuron to the other. Hence, pending some kind of reset, links can only be used once. Similarly, the rule \texttt{sigmoid} sets the attribute \texttt{st} of a neuron to 1 once the sigmoidal function has been applied. Note that this last rule can be triggered before \texttt{feedForward} has considered all possible links, thus producing an incorrect result: our evaluation strategy will take care of not executing \texttt{sigmoid} while \texttt{feedForward} is still enabled.

3.3 The introduction of data

In order to have a running net we need to specify the number of layers, the neurons in each of them, the weights of all links, and the input patterns which, in general, will be multiple. Whereas the object-oriented representation presented in the previous section is very convenient for specifying their behavior, it is clear that introducing all these data directly in this form would be very cumbersome. Hence, we have decided to use matrices and vectors of values to specify thresholds and weights, defining equations and rules to transform them into the object representation.

We define two operations to construct neurons and links:

\begin{verbatim}
  op neuronGeneration : Nat FloatList FloatList Nat -> Configuration .
  op linkGeneration : Nat Matrix Nat Nat -> Configuration .
\end{verbatim}

The operation \texttt{neuronGeneration} constructs all neurons in the layer specified in its first argument and \texttt{linkGeneration} constructs all links between two consecutive layers. For that, the second and third arguments of \texttt{neuronGeneration} are vectors or lists of real numbers with the corresponding value and threshold for each neuron. Similarly, the second argument of \texttt{linkGeneration} is a matrix of reals, that is, a list of lists where each row (list) contains the weights of all links from a neuron in a certain layer to all neurons in the next layer. The third and fourth arguments are used during the process of building the object-oriented representation; essentially, they act as counters to identify neurons and links and are increased every time a value is removed from one of the lists or the matrices.

Objects for individual neurons with the correct values from the vector representation are obtained by repeated application of the following two equations:

\begin{verbatim}
  eq neuronGeneration(L, X LX, T LT, S)
  = neuronGeneration(L, LX, LT, s S)
    < neuron(L, s S) : Neuron | x : X , t : T , st : 0 > .

  eq neuronGeneration(L, X, T, S)
  = < neuron(L, s S) : Neuron | x : X , t : T , st : 0 > .
\end{verbatim}

The argument \texttt{L} in the operation \texttt{neuronGeneration} identifies the layer the data pertain to.

Similarly, individual links are obtained from the matrix representation by means of the equations:

\begin{verbatim}
  eq linkGeneration(L, ((W LW) ; MW), S, S2)
  = linkGeneration(L, (LW ; MW), s S, S2)
    < link(L, s S2, s S) : Link | w : W , st : 0 > .

  eq linkGeneration(L, (W ; MW), S, S2)
  = linkGeneration(L, MW, 0, s S2)
    < link(L, s S2, s S) : Link | w : W , st : 0 > .
\end{verbatim}
Besides the information about neurons and links, we also need a means to specify the input, that is, the patterns the neural network is going to work with and, when available, the desired output they should produce. (This output is not needed for evaluation purposes, but it is required to train the net.) Again, we will use two operations to build our object-oriented representation in which the input/output to a single neuron constitutes an individual message, from vectors storing all patterns to a layer.

\[
\text{ops inPatternConversion : Nat FloatList Nat -> Configuration .}
\]

\[
\text{op outPatternConversion : Nat FloatList Nat -> Configuration .}
\]

--- Input-output for a single neuron

\[
\text{msg inputPattern : Nat Nat Float -> Msg .}
\]

\[
\text{msg outputPattern : Nat Nat Float Nat -> Msg .}
\]

The representation of input patterns is constructed as for neurons and links:

\[
\text{eq inPatternConversion(N, X LX, S) = inPatternConversion(N, LX, s S)} \quad \text{inputPattern(N, s S, X)} .
\]

\[
\text{eq inPatternConversion(N, X, S) = inputPattern(N, s S, X)} .
\]

As for the output patterns, objects of a new class Net will also be created to hold the difference between the generated output and the desired value.

\[
\text{class Net | e : Float, r : Bool, st : Nat .}
\]

\[
\text{eq outPatternConversion(N, X LX, S) = outPatternConversion(N, LX, s S)} \quad \text{outputPattern(N, s S, X, 0)} .
\]

\[
\text{eq outPatternConversion(N, X, S) = outputPattern(N, s S, X, 0)} .
\]

\[
\text{< net(N) : Net | e : 0.0 , r : false , st : 0 > .}
\]

Evaluation of a perceptron starts by obtaining an input pattern through the rule nextPattern, which is guided by the message netStatus. A message of the form netStatus(N0, 0, 0, N1) means that the s N0-th pattern should be considered, and then the following patterns until the N1-th.

\[
\text{msg netStatus : Nat Nat Nat Nat -> Msg .}
\]

\[
\text{crl [nextPattern] : --- use next pattern}
\]

\[
\text{netStatus(N , N1 , N2 , N0) => netStatus(s N , N1 , N2 , N0) inPatternConversion(s N , inputPattern(s N) , 0) outPatternConversion(s N , outputPattern(s N) , 0) if N < N0 .}
\]

Before starting the feedforward process, the values of the neurons in the input
layer and the corresponding weights are reset:

```
rl [resetNeuron] :
  < neuron(L, I) : Neuron | x : X , st : s S > =>
  < neuron(L, I) : Neuron | x : 0.0 , st : 0 > .
```

```
rl [resetLink] :
  < link(L, I, J) : Link | w : W , st : 1 > =>
  < link(L, I, J) : Link | w : W , st : 0 > .
```

After the perceptron has been initialized, the rule introducePattern inserts
the input pattern in the neurons of the input layer and removes them from the
configuration. Note that the input layer is identified with the number 0.

```
rl [introducePattern] :
  inputPattern(N, I, X0)
  < neuron(0, I) : Neuron | x : X , st : 0 > =>
  < neuron(0, I) : Neuron | x : X0 , st : 1 > .
```

Once we are done with the evaluation of all patterns (by means of the rules feedForward
and sygmoid of the previous section), we compute the error and mark the current
object net(N) as completed.

```
rl [computeError] :
  < neuron(2, I) : Neuron | x : X0 , st : 1 >
  outputPattern(N, I, X1, 0)
  < net(N0) : Net | e : E , st : 0 > =>
  < neuron(2, I) : Neuron | x : X0 , st : 1 >
  outputPattern(N, I, X1, 1)
  < net(N0) : Net | e : (E + (-(X1, X0)) * (-(X1, X0))) , st : 0 > .
```

After the error has been computed for all neurons in the output layer (again, a
strategy will take care of that), the attribute r stores whether the result is admissible
by checking if the error is less than a previously defined constant tol.

```
rl [setNet] :
  < net(N) : Net | e : E , r : B , st : 0 > =>
  < net(N) : Net | e : E , r : (E <= tol) , st : 2 > .
```

3.4 Shepherding the perceptron: evaluation

As remarked at the end of Section 3.2, our specification is nondeterministic and not
all of its possible executions correspond to valid behaviors of a perceptron. Hence,
in order to be able to use the specification to simulate the evaluation of patterns we
need to control the order of application of the different rules by means of strategies.

The main strategy is

```
strat feedForwardStrat : Nat @ Configuration .
sd feedForwardStrat(L') :=
  ( one(feedForward[L <- L']) ; one(sigmoid[L <- s L']) ) .
```

This strategy, which takes a natural number as argument an applies to a Configuration
(that is, a perceptron), chooses a layer L’ and applies rule feedForward, at random
positions and as long as it is enabled, to compute the weighted sum of values asso-
associated to each neuron at the layer. When all sums have been calculated, it applies
the sigmoidal function to all of them by means of rule \texttt{sygmoid} which, again, is
applied at random positions and as long as it is enabled.

There are two additional auxiliary strategies:

\begin{verbatim}
strat inputPatternStrat : @ Configuration.
sd inputPatternStrat :=
    ( one(resetNeuron)! ; one(resetLink)! ; one(nextPattern) ).

strat computeOutput : @ Configuration.
sd computeOutput :=
    ( one(computeError)! ; setNet ).
\end{verbatim}

The strategy \texttt{inputPatternStrat} takes care of making the successive patterns avail-
able and of resetting the appropriate attributes of the neurons and links, whereas
\texttt{computeOutput} is invoked to compute the error once a pattern has been evaluated.

Last, all these previous strategies are combined into the evaluation strategy,
which inputs the next pattern, computes the values of the neurons in the hidden
and the output layers, and returns the error:

\begin{verbatim}
strat evaluateANN : @ Configuration.
sd evaluateANN := ( inputPatternStrat ; feedForwardStrat(0) ;
    feedForwardStrat(1) ; computeOutput ).
\end{verbatim}

Then, to force the evaluation of the first \( M \) patterns by the multilayer perceptron
the following command would be executed:

\begin{verbatim}
(srew ann netStatus(0, 0, 0, M) using one(evaluateANN) ! .)
\end{verbatim}

where \texttt{ann} would be a term of the form

\begin{verbatim}
  neuronGeneration(0, input0, threshold0, 0)
  neuronGeneration(1, input1, threshold1, 0)
  neuronGeneration(2, input2, threshold2, 0)
  linkGeneration(1, link1, 0, 0)
  linkGeneration(2, link2, 0, 0)
\end{verbatim}

and the input patterns would have been suitable defined.

4 The backpropagation algorithm

The backpropagation training algorithm is an iterative gradient algorithm designed
to minimize the mean square error between the actual and the desired network
output. We recall it here as presented in [6].

Let us consider a perceptron with \( c+1 \) layers where layer 0 is the input layer
and layer \( c \) is the output layer. Let \( N_i \) be the number of nodes in the \( i \)-th layer,
where \( i = 0, 1, \ldots, c \). Let \( x_{ij}^k \) be the input for the \( i \) pattern of node \( j \) of layer \( k \).
Let \( y_{ij}^k \) be the output for the \( i \) pattern of node \( j \) of layer \( k \). By definition, the input and
output values coincide for the nodes of the input layer.

Let \( \omega_{ij}^k \) be the weight of the connection of neuron \( j \) of layer \( k \) with neuron \( i \) of the
previous layer. By definition of the perceptron by layers, the following relationships
are fulfilled

\begin{equation}
    x_{ij}^k = \sum_l y_{il}^{k-1} \cdot \omega_{ij}^k; \quad y_{ij}^k = f(x_{ij}^k).
\end{equation}

The mean quadratic error function between the real output of the perceptron and
the desired output, for a particular pattern $i$, is defined as
\[ E_i = \frac{1}{2} \sum_{j,k} (y_{ij}^k - d_{ij}^k)^2, \]
where $d_{ij}^k$ is the desired output for pattern $i$ of node $j$ of layer $k$.

In order to minimize the error function we use the descending gradient function, considering the error function $E_p$ and the weight sequence $\omega_{ij}^k(t)$, started randomly at time $t = 0$, and adapted to successive discrete time intervals. We then have
\[ \omega_{ij}^k(t + 1) = \omega_{ij}^k(t) - \eta \frac{\partial E_i}{\partial \omega_{ij}^k}(t), \]
where $\eta$ is the so-called learning rate constant.

Learning is usually faster if a term of the moment of inertia is added and the changes of the connection weights are adapted as
\[ \omega_{ij}^k(t + 1) = \omega_{ij}^k(t) - \eta \frac{\partial E_i}{\partial \omega_{ij}^k}(t) + \alpha [\omega_{ij}^k(t) - \omega_{ij}^k(t - 1)], \]
where $\alpha$ is a value between 0 and 1 called the inertia constant.

It is now necessary to calculate the quantities $\partial E_i / \partial \omega_{ij}^k$ and substitute in the expressions (2) or (3) for each one of the learning patterns, thus optimizing the values of the connection weights of the network.

Applying the rule of the chain in $\partial E_i / \partial \omega_{ij}^k$ and using the relationships (1), we can conclude that
\[ \frac{\partial E_i}{\partial y_{ij}^k} = \sum_m \frac{\partial E_i}{\partial x_{jm}^{k+1}} \cdot \frac{\partial x_{jm}^{k+1}}{y_{ij}^k} = \sum_m \frac{\partial E_i}{\partial x_{jm}^{k+1}} \cdot \omega_{jm}^{k+1}, \]
and, substituting,
\[ \frac{\partial E_i}{\partial \omega_{ij}^k} = \frac{\partial E_i}{\partial x_{ij}^k} \cdot \frac{\partial x_{ij}^k}{\partial \omega_{ij}^k} = \left( \frac{\partial E_i}{\partial x_{ij}^k} \cdot \frac{\partial x_{ij}^k}{\partial \omega_{ij}^k} \right) \cdot \omega_{ij}^k = \left( \sum_m \frac{\partial E_i}{\partial x_{jm}^{k+1}} \cdot \omega_{jm}^{k+1} \right) \cdot f'(x_{ij}^k) \cdot y_{ji}^{k-1}. \]

We next review how it works [13,6], when particularized to the case in which the sigmoidal function is $f(x) = 1/(1 + e^{\nu-x})$.

(i) Set all weights and node states to small random values.
(ii) Introduce the input vector $x_1, \ldots, x_n$ and the desired output $d_1, \ldots, d_m$.
(iii) Use the net to evaluate the actual output $y_1, \ldots, y_m$ produced by $x_1, \ldots, x_n$.
(iv) Adjust weights according to the equation
\[ w_{ij}(t + 1) = w_{ij}(t) + \eta \delta_j x'_i. \]

In this equation, $w_{ij}(t)$ is the weight of the link connecting neuron $i$ to neuron $j$ at time $t$, $x'_i$ is the output of neuron $i$ (input and output values coincide for the neurons of the input layer), $\eta$ is a gain term or learning constant, and $\delta_j$ is an error term for node $j$. For output neurons:
\[ \delta_j = y_j(1 - y_j)(d_j - y_j). \]
If node $j$ belongs to a hidden layer, then

$$\delta_j = x'_j(1 - x'_j) \sum_k \delta_k w_{jk},$$

where $k$ ranges over all neurons in the layers above neuron $j$. Internal node
thresholds are adapted in a similar manner. (However, since training thresholds
is not necessary to optimally classify the data, we do not consider it in our
implementation.)

The speed of convergence to a solution and the propensity to fall into local
minima both depend heavily on the learning constant. Standard backpropagation
is very sensitive to the initial learning rate chosen for a given learning task. In
general the optimum value of the learning constant depends on the problem being
solved: there is no single value suitable for different training cases.

### 4.1 Backpropagation in Maude

For training the net we need neurons and links to hold additional information,
namely the error terms $\delta_j$ and the adjusted weights. Since evaluation is part
of backpropagation, we define `NeuronTR` and `LinkTR` as subclasses of `Neuron` and `Link`
with an additional attribute to store the extra information.

```plaintext
class LinkTR | w+1 : Float . subclass LinkTR < Link .
class NeuronTR | dt : Float . subclass NeuronTR < Neuron .
```

Note that the rules for evaluating a net also apply to these new objects; the new
attributes are simply ignored. Hence, we already have the code for the first three
steps of the algorithm.

The next step demands the evaluation of the error terms $\delta_j$ before adjusting the
weights. The calculation of these $\delta_j$ depends on whether we are working with the
output layer or not. For the output layer, the corresponding rule is straightforward:

```plaintext
rl [delta2] :
< neuron(2, I) : NeuronTR | x : X , dt : DT , st : 2 >
outputPattern(N, I, D, 1)
=>
< neuron(2, I) : NeuronTR | x : X ,
  dt : (X * ((-_(1.0, X)) * (_-_(D, X))) ) , st : 3 > .
```

The case for the remaining layers is a bit more involved and is split in three phases:
the operation `delta1A` initializes $dt$ to zero, `delta1B` below takes care of calculating
the sum of the weights multiplied by the corresponding error term, and `delta1C`
computes the final product.

```plaintext
rl [delta1A] :
< neuron(1, I) : Neuron | dt : DT , st : 1 > ==>  
< neuron(1, I) : Neuron | dt : 0.0 , st : 2 > .  
rl [delta1B] :
< neuron(1, J) : Neuron | dt : DT1 , st : 2 >
< neuron(2, K) : Neuron | dt : DT2 , st : 2 >
< link(2, J, K) : Link | w : W , st : 2 > 
=>
< neuron(1, J) : Neuron | dt : (DT1 + (DT2 * W)) , st : 2 >
< neuron(2, K) : Neuron | dt : DT2 , st : 2 >
< link(2, J, K) : Link | w : W , st : 3 > .
```

72
Again, in all these rules the status attribute \( st \) is correspondingly updated.

Once the error terms are available, the updated weights can be calculated: rule \( link1 \) does it for the hidden layer and \( link2 \) for the output layer.

\[ \text{rl} \{ \text{link1} \} : \]
\[ \begin{align*}
&< 	ext{neuron}(0, I) : \text{Neuron} \mid x : X_1, st : 1 > \\
&< 	ext{neuron}(1, J) : \text{Neuron} \mid dt : DT, st : 3 > \\
&< \text{link}(1, I, J) : \text{Link} \mid w : W, w+1 : W_1, st : 1 > \\
&=> \]
\[ < 	ext{neuron}(0, I) : \text{Neuron} \mid x : X_1, st : 1 > \\
< 	ext{neuron}(1, J) : \text{Neuron} \mid dt : DT, st : 3 > \\
< \text{link}(1, I, J) : \text{Link} \mid w : W, w+1 : (W + (\eta * (DT * X_1))), st : 3 > .
\]

\[ \text{rl} \{ \text{link2} \} : \]
\[ \begin{align*}
&< \text{link}(2, I, J) : \text{Link} \mid w : W, w+1 : W_1, st : 1 > \\
&< \text{neuron}(1, I) : \text{Neuron} \mid x : X_1, st : 2 > \\
&< \text{neuron}(2, J) : \text{Neuron} \mid dt : DT, st : 1 > \\
&=> \]
\[ < \text{link}(2, I, J) : \text{Link} \mid w : W, w+1 : (W + (\eta * (DT * X_1))), st : 1 > \\
< \text{neuron}(1, I) : \text{Neuron} \mid x : X_1, st : 2 > \\
< \text{neuron}(2, J) : \text{Neuron} \mid dt : DT, st : 2 > .
\]

Last, the old weights are replaced by the adjusted ones.

\[ \text{rl} \{ \text{switchLink} \} : \]
\[ \begin{align*}
&< \text{link}(N, I, J) : \text{Link} \mid w : W, w+1 : W_1, st : s S > \\
&< \text{neuron}(N, I, J) : \text{Neuron} \mid x : X, w+1 : W_1, st : 0 > .
\]

The reason why this rule (and a fortiori, the attribute \( w+1 \)) is needed, instead of simply updating the value of the attribute \( w \) in the \( \text{link} \) rules, is because the old weights are used in the computation of the error terms \( \delta \) and could be lost otherwise.

### 4.2 Shepherding the perceptron: training

As we did for evaluation, we need to define an appropriate strategy for training the perceptron. Assuming we have already calculated the output associated to a pattern, we next must calculate the error terms \( \delta \), use them to obtain the adjusted weights, and transfer them to the right attribute. That can be easily done by applying the rules defined in the previous section in the right order.

\[ \text{strat backpropagateANN} : \emptyset \text{ Configuration} . \]
\[ \text{sd backpropagateANN} := \]
\[ ( \text{one(delta2)} ! ; \text{one(link2)} ! ; \text{one(delta1A)} ! ; \text{one(delta1B)} ! ; \text{one(delta1C)} ! ; \text{one(link1)} ! ; \text{one(switchLink)} ! ) . \]

Finally, training a net consists in evaluating a pattern, with the strategy defined in Section 3.4, and then adjusting the weights accordingly.

\[ \text{strat stratANN} : \emptyset \text{ Configuration} . \]
\[ \text{sd stratANN} := ( \text{evaluateANN} ; \text{backpropagateANN} ) . \]
5 Example: Diagnosis of glaucoma

For the diagnosis of glaucoma, Gustavo Santos-García participated in a project that proposed the use of an intelligent system that employs artificial neural networks and integrates the analysis of the nerve fibers of the retina from the study with scanning laser polarimetry (NFAII:GDx), perimetry and clinical data [3]. The resulting multilayer perceptron was trained using MatLab.

We used the data from that project as a test bed for our specification of the backpropagation algorithm in Maude. Our results coincided and the success rate was of 100% but the execution time of our implementation lagged far behind, which motivated us to optimize our code. Since equations are executed much faster than rules by Maude and, in addition, do not give rise to branching but linear computations, easily handled by strategies, we simplified rules as much as possible. The technique used was the same in all cases and is illustrated here with feedForward:

```
rl [feedForward] : C => feedForward(C).
op feedForward : Configuration -> Configuration.
eq feedForward(C < link(s L, I, J) : Link | w : W, st : 0 >
   < neuron( L, I) : Neuron | x : X1 , st : 1 >
   < neuron(s L, J) : Neuron | x : X2 , st : 0 >) =
   feedForward(C < link(s L, I, J) : Link | w : W, st : 1 >
   < neuron( L, I) : Neuron | >
   < neuron(s L, J) : Neuron | x : (X2 + (X1 * W)) >).
eq feedForward(C) = C [owise].
```

The evaluation and training strategies had to be correspondingly modified since the combinator ! was no longer needed. The resulting specification is obviously less natural, but more efficient; however, it is still not competitive with MatLab.

6 Conclusions

We have presented in this paper a specification of a class of artificial neural networks, namely multilayer perceptrons, in a two step fashion. First we have shown how to use rewrite rules guided by strategies to simulate the evaluation of patterns by a perceptron, and then we have enhanced the specification to make the training of the net possible.

The evaluation process is straightforward, essentially amounting to the repeated application of two rules, feedForward and sigmoid, which further does credit to the suitability of rewriting logic as a framework for (yet another model of) concurrency. The training algorithm requires more rules, but the strategy is also rather simple.

However, the simplicity of the resulting specification should be put in perspective. First of all, the election of our concrete representation in which neurons and links are individual entities and which, at first sight, might not strike as the most appropriate, is of paramount importance. Indeed, our first attempts at specifying perceptrons made use of a vector representation like the one we have used here for inputting the data and similar to that proposed in [10]. Such representation was actually suitable for the evaluation of patterns but proved unmanageable when considering the training algorithm.

In addition to the representation, the availability of strategies turned out to be crucial. While with the vector representation layers could be considered as a whole
and there was no much room for nondeterminism, the change to the object-oriented representation gave rise, as we have observed, to the possible interleaving of rules in an incorrect order. It then became essential the use of the strategy language to guide the rewriting process in the right direction.

As a result, our specification is the happy crossbreed of an object-oriented representation and the use of strategies: without the first the resulting specification would have been much more obscure, whereas without the availability of the strategy language, its interest would have been purely theoretical.

References


A Rewriting Logic Approach to Static Checking of Units of Measurement in C \(^1\)

Mark Hills\(^2\) Feng Chen\(^3\) Grigore Roșu\(^4\)

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL, US

Abstract

Many C programs assume the use of implicit domain-specific information. A common example is units of measurement, where values can have both a standard C type and an associated unit. However, since there is no way in the C language to represent this additional information, violations of domain-specific policies, such as unit safety violations, can be difficult to detect. In this paper we present a static analysis, based on the use of an abstract C semantics defined using rewriting logic, for the detection of unit violations in C programs. In contrast to typed approaches, the analysis makes use of annotations present in C comments on function headers and in function bodies, leaving the C language unchanged. Initial evaluation results show that performance scales well, and that errors can be detected without imposing a heavy annotation burden.

Keywords: Unit safety, rewriting logic, abstract semantics, static analysis.

1 Introduction

Many programs make use of domain-specific information. A common example, often occurring in scientific and engineering applications, is the use of units of measurement. Units are associated with specific values or variables; unit rules then determine how operations in the language (addition, multiplication, etc) change and combine units, and also determine when this is safe. In many languages, including C, this information on units is implicit: instead of having a program-level representation, values are assumed by the programmer to have specific units, which may be documented informally in source comments. Unfortunately, the implicit nature of this information means that it cannot be used to automatically ensure that unit manipulations are safe, i.e., that operators are always applied to operands with

\(^1\) Supported by NSF CCF-0448501 and NSF CNS-0509321.
\(^2\) Email: mhmills@cs.uiuc.edu
\(^3\) Email: fengchen@cs.uiuc.edu
\(^4\) Email: grosu@cs.uiuc.edu

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs
typedef struct {
  double atomicWeight;
  double atomicNumber;
} Element;

//@ pre(UNITS): unit(material->atomicWeight) = kg
//@ pre(UNITS): unit(material->atomicNumber) = noUnit
//@ post(UNITS): unit(@result) = m ^ 2 kg ^ -1
double radiationLength(Element * material) {
  double A = material->atomicWeight;
  double Z = material->atomicNumber;
  double L = log( 184.15 / pow(Z, 1.0/3.0) );
  double Lp = log( 1194.0 / pow(Z, 2.0/3.0) );
  return ( 4.0 * alpha * re * re) * ( NA / A ) *
    ( Z * Z * L + Z * Lp );
}

//@ pre(UNITS): unit(material->atomicWeight) = kg
//@ pre(UNITS): unit(material->atomicNumber) = noUnit
//@ pre(UNITS): unit(density) = kg m ^ -3
//@ pre(UNITS): unit(thick) = m
//@ pre(UNITS): unit(initEnergy) = kg m ^ 2 s ^ -2
double finalEnergy(Element * material, double density,
                     double thick, double initEnergy) {
  double X0 = radiationLength(material);
  return initEnergy / exp ( thick / X0 );
}

Fig. 1. Electron Energy Example, in C

compatible units. The burden to ensure this falls directly on the programmer. This
leaves open the possibility that serious domain-specific errors will go undetected.

The possibility of serious errors is not just theoretical. On September 30, 1999,
NASA’s Mars Climate Orbiter spacecraft crashed into Mars’ atmosphere due to a
software navigation error, caused by one team using English units while another
used metric units in a key Orbiter operation [2]. Roughly 15 years before this, the
space shuttle Discovery flipped over mid-flight in an attempt to point a mirror at
a spot 10,023 feet above sea level; the software interpreted this figure as 10,023
nautical miles, or roughly 60,900,905 feet [27].

Checking by hand may be an option for small programs, but does not scale to
large programs. Even in small programs, some calculations can be very complex
and can depend on non-local information, like the contents of global variables and
the results of function calls, making manual checking challenging. For instance, a
portion of a program used to calculate the final energy of an electron \(^5\) is shown
in Figure 1. Without a method to record expected units and check for correctness,

\(^5\) This example was borrowed from Jiang and Su’s work on Osprey [18], which in turn borrowed it from
Brown’s work on SIUNITS [7].
it is not obvious whether the code is, or is not, unit-safe. In fact, line 26 will report a unit error: the unit returned by the \texttt{radiationLength} calculation will be \texttt{meter}^2\texttt{kilogram}^{-1}, and \texttt{thick} has unit \texttt{meter}, so \texttt{thick} divided by \texttt{X0} will have unit \texttt{meter}^{-1}\texttt{kilogram}. However, \texttt{exp} expects a unitless argument, meaning either the annotations are incorrect or \texttt{radiationLength} is not being used correctly.

Many approaches have been proposed to enforce unit safety in programs, a number of which are discussed in Section 2. In this paper, we propose a new solution for the C language, CPF[UNITS]. CPF[UNITS], based on the CPF framework [16], is a significant extension of the ideas introduced in the proof-of-concept C-UNITS system [29]. CPF[UNITS] allows unit-specific annotations to be added to C programs in the form of function preconditions, function postconditions, assertions, and assumptions. These annotations are then checked for validity using a combination of the abstract rewriting logic semantics of C, part of CPF, and the UNITS policy, a collection of unit-specific semantics for certain language features and the combination of an annotation language and annotation semantics. Hence the name CPF[UNITS], for CPF parameterized by the UNITS policy.

The remainder of this paper is organized as follows. We first present related work in Section 2, including the earlier C-UNITS system and approaches based on types. We then provide introductory details on the abstract rewriting logic semantics of C in Section 3, assuming familiarity with term rewriting and a basic familiarity with equational or rewriting logic. An introduction to units of measurement is presented in Section 4, followed by details of the CPF[UNITS] unit safety checker in Section 5. Section 6 presents initial evaluation results, with Section 7 presenting possible future work and concluding. Our website provides downloads of all tools and examples described in this paper, along with a web-based interface for experimentation [1].

2 Related Work

Related work on unit safety tends to fall into one of three categories: library-based solutions, where libraries which manipulate units are added to a language; language and type system extensions, where new language syntax or typing rules are added to support unit checking in a type checking context; and annotation-based solutions, where user-provided annotations assist in unit checking.

Library-based solutions have been proposed for several languages, including Ada [15,23], Eiffel [19], and C++ [7]. The Mission Data Systems team at NASA’s JPL developed a significant library, written in C++, which includes several hundred classes representing typical units, like \texttt{MeterSecond}, with appropriately typed methods for arithmetic operations. An obvious disadvantage of such an explicit approach is that the units supported by the library are fixed: adding new units requires extending the library with new classes and potentially adding or modifying existing methods to ensure the new classes are properly supported.

Solutions based around language and type system extensions work by introducing units into the type system and potentially into the language syntax, allowing expressions to be checked for unit correctness by a compiler or interpreter using extended type checking algorithms. MetaGen [5], an extension of the MixGen [4] extension of Java, provides language features which allow the specification of dimen-
sion and unit information for object-oriented programs. Other approaches making use of language and type system extensions have targeted ML [21,20], Pascal [13,17], and Ada [14].

A newer tool, Osprey [18], also uses a typed approach to checking unit safety, allowing type annotations in C programs (such as `#meter int`) using a modified version of CQUAL [12]. These annotations can then be checked using a combination of several tools, including the annotation processor, a constraint solver, a union/find engine, and a Gaussian elimination engine (the latter two used to reduce the number of different constraints and properly handle the Osprey representation of unit types as matrices). One limitation of Osprey is that there is no way to express relationships between the units of function parameters and return values, something possible with a richer annotation language:

```c
//@ post(UNITS): unit(@result)^2 = unit(x)
double sqrt(double x) { ... }
```

Instead, this type of relationship has to be added by hand-editing files generated during processing. Osprey also checks dimensions (i.e., length), not units (i.e., meters or feet), instead converting all units in a single dimension into a canonical unit. This can mask potential errors: for instance, it is not an error to pass a variable declared with unit meter to a function expecting feet. On the other hand, Osprey includes functionality to check explicit conversions for correctness, catching common conversion errors such as using the wrong conversion factor.

Annotation-based systems, including JML [8] and Spec# [6], have been applied to many problem domains, but not specifically to units. Systems for unit safety based on annotations include the C-UNITS system [29], which used concepts about abstract semantics and annotations that first appeared in the context of BC, a small calculator language [9]. CPF[UNITS] was inspired by the work on C-UNITS, and takes a similar approach, with a focus on using abstract semantics and annotations. However, CPF[UNITS] has extended this approach in three significant ways. First, CPF[UNITS] has been designed to be modular: the abstract semantics of C have been completely redefined using concepts developed over the last several years as part of the rewriting logic semantics project [25]. The semantics are divided into core modules, shared by all CPF policies, and units modules, specific to CPF[UNITS]. This allows improvements in the core modules to be shared by all policies, simplifies the unit checking logic, and greatly improves the ease with which the semantics can be understood and modified. Second, CPF[UNITS] has been designed to cover a much larger portion of C. C-UNITS was designed as a prototype, and left out a number of important C features, with minimal or no support for structures, pointers, casts, switch/case statements, gotos, or recursive function calls. Support for expressions was also limited, with the main focus on commonly-used expressions, and more complex memory scenarios (structures with pointers, arrays of pointers, etc) were ignored. CPF[UNITS] supports all these features, and makes use of a more advanced parser to handle a larger class of C programs. Finally, CPF[UNITS] has been designed to be more scalable. While C-UNITS requires a complete program for analysis, CPF[UNITS] analyzes individual functions, leading to smaller individual verification tasks.
The technique used by CPF[UNITS], like most (if not all) static analyses, could be framed in terms of abstract interpretation [11], where the domain of interpretation is the algebra of units of measurement. However, CPF[UNITS] makes intensive use of recently developed rewriting logic language definitional techniques based on representations of abstract syntax trees as continuations; establishing the relationships between rewriting logic semantics and abstract interpretation is an interesting subject in and of itself, but it goes beyond our purposes in this paper.

3 Abstract Rewriting Logic Semantics of C

The abstract semantics of C is defined using Maude [10], a high-performance language and engine for rewriting logic. The current program is represented as a “soup” (multiset) of nested terms representing the current computation, environment (mapping names to abstract values and other information), analysis results, bookkeeping information, and analysis-specific information. The most important piece of information is the Computation, named k, which is a first-order representation of the current computation, made up of a list of instructions separated by ->. The Computation can be seen as a stack, with the current instruction at the left and the remainder of the computation at the right. This methodology is described in more detail in papers about the rewriting logic semantics project [24,25]. To simplify the presentation of the rules in the CPF semantics, we use K notation [28], which includes a number of simplifying conventions.

Figures 2 and 3 show an example of a semantic rule included in the abstract C semantics used in the CPF[UNITS] tool (see Section 5), first in K notation, then in Maude. The rule represents a memory lookup operation. Here, if identifier X is being looked up, and the environment contains an item named X with location L and unit value U, a location value pair lvp containing L and U, lvp(L,U), is returned in place of the lookup operation, while the environment remains unchanged. The K version uses three K conventions: > is used in place of ) to represent “and everything else”, expanded into -> K in the Maude version; < is used for set matching (“everything else to either side”), which requires Env to represent “everything else” in Maude; and replacement is represented by underlining the portion of the term that has changed, allowing unchanged portions of the term to be mentioned without the need to be repeated. K also does not need the wrapper u(), which is used to wrap units and turn them into values.

---

6 This is slightly simplified from the actual semantics, where the environment is made up of 5-tuples instead of triples, but is otherwise the same.
op _^_ : Unit Rat -> Unit.
op __ : Unit Unit -> Unit [assoc comm].
eq U ^ 0 = noUnit .
eq U ^ 1 = U .
eq U U = U ^ 2 .
eq U (U ^ Q) = U ^ (Q + 1) .
eq (U ^ Q) (U ^ P) = U ^ (Q + P) .
eq (U U') ^ Q = (U ^ Q) (U' ^ Q) .
eq (U ^ Q) ^ P = U ^ (Q * P) .
ops noUnit any fail cons : -> Unit .
ops meter m feet f : -> Unit .

Fig. 4. Units of Measurement, in Maude

4 Units of Measurement

In the International System of Units (SI), there are seven base dimensions, including length, mass, and time [3]. Each base dimension includes a standard base unit, such as meters for length or seconds for time. Other units can be defined for each dimension in terms of the base unit – feet or centimeters for length, for instance. Units can also be combined to form derived units, such as area (meters squared, or meter meter) and velocity (meters per second).

Technically, the algebraic structure of units forms an Abelian group. This provides several important properties which need to be modeled during unit checking. First, as mentioned above, units can be combined to form new units – for any two units $A$ and $B$, $AB$ is also a unit ($AB$ is the product of units $A$ and $B$). Units are also associative (given $C$ is also a unit, $(AB)C$ is the same as $A(BC)$), commutative ($AB$ is the same as $BA$), and have inverses and identities. Generally, products of the same unit are represented with exponents, i.e. $AA$ is the same as the more commonly used $A^2$, but both forms are acceptable and should be usable.

Our equational definition of the units domain is shown in Figure 4. The first two operands (defined with op) specify that a unit can have a rational exponent and that the product of two units is a new unit. The following seven equations (defined with eq) are used to simplify units, putting them into a canonical form, with $P$ and $Q$ representing rational numbers. The next two operand lists define some actual units: meters and feet, along with short forms, plus special units: noUnit, any, fail, and cons. noUnit represents the unit of values that have none, like the result of a bitwise computation. any means a value can be considered to be of any unit, which is similar to cons, the unit given to constants (cons is used internally as the default unit of constants, while any can be used in annotations). Finally, fail represents a unit failure, and is represented as a unit so it can be easily propagated. Additional equations, not shown here, are also provided to allow for canonical forms of predefined units, for instance ensuring that $m$ and meter are recognized as the same unit.
5 CPF[UNITS]: Checking Unit Safety of C Programs

In this section we present CPF[UNITS], a tool for checking the unit safety of C programs. In CPF[UNITS], users specify units on C objects\(^7\) that hold numerical values by providing annotations in comments in the source code. Annotations indicate function preconditions and postconditions, assertions, and assumptions. The annotated code is converted into a formal representation based on a Maude-defined abstract C semantics, and then checked function by function, ensuring that the size of the verification task does not (except in some pathological conditions, such as with deeply nested conditionals where each branch makes different changes to units) grow too large. The use of conditionals and looping constructs can cause multiple units to be associated with a single object\(^8\). Techniques to handle this, while still maintaining precision, are discussed below.

CPF includes logic to add annotations to C programs, parse these programs, generate verification tasks, and process most C statements. The portions of CPF not specific to the UNITS policy are described below at a high level; additional coverage, including detailed information about the CPF, a sample not-null policy, and a high-level introduction to the units policy can be found in a companion technical report [16].

5.1 Code Annotations

In CPF[UNITS], code annotations are included directly in the C source code as comments, starting either with /*@ (for block comments) or //@ (for line comments). Examples of annotations can be found in Figure 1. Note that both functions, such as radiationLength and finalEnergy, and function prototypes can be annotated, allowing annotations to easily be added to library functions. @result is a special token used to refer to the return value of the function. In general, a function can have multiple, or no, preconditions or postconditions.

5.2 Generating Verification Tasks

The CPF[UNITS] semantics assume that one verification task will be generated for each function. The tasks are generated using a combination of a Perl script for processing annotations and a modified version of the CIL tool for C [26], which provides parsing, analysis, and code transformation capabilities. After parsing the program, CIL first performs a CPF-specific three-address transform, moving expression computations out of function calls and return statements. CIL then performs a CPF-specific inlining step, where function call sites are replaced by the function preconditions and postconditions of the called function, with preconditions becoming asserts and postconditions becoming assumes. The preconditions and postconditions for each function are then moved into the function body, with preconditions becoming assumptions at the start of the body and postconditions placed before each return statement as assertions. The latter operation is safe because the trans-

---

\(^7\) In C, an object is a memory region that can be read from or written to.

\(^8\) Recursive calls do not require similar treatment; since functions are checked individually, call sites are handled without descending into the called function.
form moves any computation out of the return statement, ensuring that a return will not also modify units referenced in the postcondition. Finally, CIL also generates the verification tasks as part of a CPF-specific pretty-printing step; instead of printing out modified C code, which is the standard CIL behavior, verification tasks for each function, given in Maude using the CPF abstract C syntax, are generated. Figure 5 illustrates the process of checking annotated code using CPF. More information about this process can be found in the CPF technical report [16].

5.3 Checking Unit Safety

Once the verification tasks for each function have been generated, each task is checked using Maude. The executable nature of rewriting logic specifications is leveraged to symbolically execute programs using an abstract rewriting logic semantics. This semantics is made up of the CPF core semantics and, for unit safety, the unit-specific semantics included in the UNITS policy. The CPF semantics includes an abstract syntax of C and semantics for C statements, high-level definitions of concepts such as “value” and “annotation language”, and a number of hooks which allow policy-specific behavior to be added. CPF[UNITS] extends this with semantics for declarations, assertions, assumptions, and expressions, a definition of the UNITS annotation language used in annotations, and a policy-specific concept of unit values. Here, we focus on those parts specific to CPF[UNITS], with the CPF semantics described here at a high level of detail; as mentioned above, more detail about CPF is available in a technical report [16] and on the CPF website [1].

5.3.1 CPF: Shared Semantics

The bulk of the shared CPF semantics is focused on the semantics of C statements, defined over an abstract semantics for C language constructs. In CPF, statements are executed in an environment, which provides information on the names, values, and types of C objects which will be used in a statement’s constituent expressions. In some cases, such as with conditionals that make different changes to the environment on different paths, it is possible for a statement to start with one environment but return multiple environments as the result of execution\(^9\). Part of the program state is thus a set of environments, with the framework executing each statement

---

\(^9\) This makes our analysis path-sensitive, although we make no attempt to track which conditions led to which path.
Hills and Chen and Roșu

```c
int x,y,z;
//@ assume(UNITS): unit(x) = unit(y) = m
if (b) {
  y = 3; //@ assume(UNITS): unit(y) = f
  x = y;
}
z = x + y;
```

Fig. 6. Path-Sensitive Unit Assignment, in C

```
op _^_ : Unit CInt -> Unit .
op u : Unit -> Value .
op ptr : Location -> Value .
op arr : Location -> Value .
op struct : Identifier SFieldSet -> Value .
op union : Identifier SFieldSet -> Value .
```

Fig. 7. CPF[UNITS] Values, in Maude

once in each environment, and gathering the resulting environments together as the environment set to use for the following statement(s). An example where this could occur is shown in Figure 6, where the unit assigned to both \( x \) and \( y \) starts off as \( m \), but can be either \( m \) or \( f \) for both at the end of the if. If, instead of using environment sets, sets of values were associated with each object, a false positive would be generated on line 7, since \( x \) and \( y \) could both either be \( f \) or \( m \), leading to an invalid combination of one of each. One disadvantage to this use of environment sets is that the analysis can be potentially more expensive, especially in certain pathological cases where the environment keeps splitting (deeply nested conditionals, for instance). In practice this does not appear to happen often, since units do not often change once they are initially assigned; as a precaution, CPF allows the use of a high-water mark on the size of the environment set, which when crossed causes some environments (selected randomly) in the set to be discarded and a warning message to be issued.

CPF defines semantics for all C statements, such as conditionals and gotos, and also includes semantics to “break down” expressions into their constituent pieces: \( E + E' \) into evaluations of \( E \) and \( E' \), for instance. The policy-specific semantics specifies the abstract values to which \( E \) and \( E' \) can evaluate.

5.3.2 CPF[UNITS]: Abstract Values

The CPF[UNITS] values are a combination of unit values and values representing C pointers, structures, unions, arrays, and enums (treated as constants). The unit values are those defined in the theory of units shown in Figure 4, augmented with C-specific unit values that include simple C expressions, such as constant integers in the exponent. Pointers are represented as locations; dereferencing accesses the actual value at that location, such as a unit, a structure, or another pointer (for multiple levels of indirection). Arrays have a similar representation, which allows them to be used like pointers, but limits them to containing only a single value,
so all array elements are considered to have the same unit. Structures and unions contain the name of the structure or union type (anonymous structures and unions are given names by CIL) and a set of field/location pairs to indicate where the value of each field is stored. Function pointers are represented with a special value, with alias analysis used to determine which function is invoked by an indirect call through the pointer; warnings are issued if it is not possible to determine a unique function at a call site. A subset of the value definitions used by the CPF[UNITS] policy is shown in Figure 7.

5.3.3 CPF[UNITS]: C Declarations

The CPF[UNITS] rules for handling declarations provide an initial symbolic representation of memory for the global variables, formal parameters, and local variables of a function. Different allocators are used for each C object type, allocating initial values appropriate to the object. For instance, the allocator for scalars initially associates a “fresh” unit, unique to that declaration, with the scalar, while the allocator for pointers associates a reference to a new memory location. Declarations for structure and union variables allocate field/location maps based on the fields contained in the structure or union declaration. Allocation is recursive; structure allocation also allocates the fields of the structure, while allocation of arrays and pointers allocates the base type of the array or pointer as well, with unions currently represented like structures (i.e., we do not attempt to just allocate one location shared by all fields in the union). One challenging but common case to represent is structures which contain pointers to other structures. It is not possible to allocate the entire memory representation up front, since this could be (in theory, at least) infinite. Pointers inside structures are instead created with an allocation trigger, which will allocate the pointer’s target on the first attempt to access it. This allows the memory representation to grow sensibly, modeling just what is needed to perform unit checking.

After processing all declarations in the function body/verification task (CIL moves all declarations to the top of a function, using renaming to model shadowing), initial values are given to local variables using a combination of assertions (from annotations) and assignments. For instance, an assertion may indicate that variable \( x \) has unit \texttt{meter}; a declaration like \texttt{int y = x;} would then associate \texttt{meter} with \( y \) as well. Initial units for formal parameters and global variables are based just on the function preconditions. If a precondition or assignment does not indicate the initial unit of a variable, it keeps its assigned fresh (unique) unit, which will allow detection of errors from misuse of the variable in expressions. After all initial assignments are complete, a locking process locks certain memory locations to make sure they cannot be changed in ways that are not reflected in the annotations. For instance, it is not possible to write a new unit through a pointer given as a formal parameter. This ensures that changes visible outside the function but not included in the preconditions and postconditions are prevented, allowing checking to be truly modular\(^{10}\). Finally, a “checkpoint” is taken, saving the original assigned values.

\(^{10}\)It is possible to override this locking behavior using annotations, but this will generate a warning message to alert the user to the potential unsoundness created by doing so. We are working on incorporating alias analysis results into the locking process to ease this restriction.
[1] \( U \cdot U' = U \cdot U' \)

[2] \( U + U' = \text{mergeUnits}(U,U') \rightarrow \text{checkForFail}('+') \)

[3] \( U > U' = \text{mergeUnits}(U,U') \rightarrow \text{checkForFail}('>') \rightarrow \text{discard} \rightarrow \text{noUnit} \)

[4] \((lvp(L,V) = V') = V' \rightarrow \text{assign}(L)\)

[5] \((lvp(L,U) += U') = \text{mergeUnits}(U,U') \rightarrow \text{checkForFail}('=') \rightarrow \text{assign}(L)\)

[6] \(*\text{(lvp}(L,\text{ptr}(L'))) = \text{llookup}(L')\)

[7] \text{lvp}(L,\text{struct}(X', (\text{sfield}(X,L') \_))) . \ X = \text{llookup}(L')

Fig. 8. CPF[UNITS] Expression Rules, in K

before any changes are made in the function body. This allows these original values to be accessed later during unit checking, such as when checking the assertions added to represent the function postconditions.

5.3.4 CPF[UNITS]: C Expressions

To evaluate expressions in CPF[UNITS] the semantic rules need to properly modify, combine, and propagate abstract values representing units and C objects (pointers, structures, etc). Expressions, along with assert statements, are also the point where unit safety violations are discovered, so semantics for expressions which can produce failures need to ensure that the failures are properly handled. Figure 8 includes rules for a representative set of expressions, illustrating how abstract values are propagated and failures are detected.

The first rule models the multiplication operation. Here, given two unit values \( U \) and \( U' \), the result is their product, \( U \cdot U' \). The second rule, for addition, is structured similarly to the first, but must also check that the combination of the units is correct. This is done by merging the units with \text{mergeUnits}. In merging, if one unit is \text{any} or \text{cons}, both of which can be treated as being of any unit, the other unit is returned. Otherwise, the two units must match, with no automatic conversions between units performed. If the units do not match, or one of the units is \text{fail}, \text{fail} is returned to indicate a unit safety violation. This enforces the unit rule for addition – the units of both operands must be the same. To detect the failure and issue a warning, \text{checkForFail} is used, which will print an error message if the result unit is \text{fail}.

The third rule handles the greater-than relational operation. Here, the rule is the same as for addition: to compare two values they must have the same unit. Beyond this, the returned unit is \text{noUnit}, since it does not make sense to assign a unit to the result of the comparison. The fourth rule is used for assignment. Here, the lvalue evaluates to an \text{lvp}, or location-value pair, with the location and current value of the object being assigned into. The value of the right-hand expression is assigned over the current value to the same location. While this works for units, it also works for other C entities, such as the representations for pointers and structures. The
Unit \( U ::= \) \( \text{unit}(E) \mid \text{unit}(E) \land Q \mid BU \mid U U \)

UnitExp \( UE ::= \) \( U \mid U = UE \mid UE \text{ and } UE \mid UE \text{ or } UE \mid UE \text{ implies } UE \mid \text{not } UE \)

Fig. 9. Units Annotation Language

fifth rule, for the \( \bowtie \) operand, is a combination of the rules for \( + \) and assignment, performing both the check for failure and the assignment to the location of the lvalue. In this case, the values should be units, since it is necessary to compare them to verify the operation is safe; a different rule would be needed for pointer arithmetic. In both the fourth and fifth rule, parens have been added to clearly distinguish the \( K = \) from the C assignment \( = \).

Finally, rules six and seven show how some aspects of pointers and structures are handled. A pointer is represented as a location – a pointer to location \( L \) has the value \( \text{ptr}(L) \). On dereference, the location held in the pointer is looked up to retrieve its value. A structure is represented as a tuple containing the name of the structure type and the aforementioned finite map from field names to locations; the unneeded part of the finite map is represented as \( _{-} \). When field \( X \) is looked up in a structure, like \( S.X \), the location of \( X \) is retrieved using the map and then looked up to bring back the value assigned to the field.

5.3.5 CPF[UNITS]: Annotation Language, Asserts, and Assumes

The unit annotation language is shown in Figure 9. \( \text{Unit} \) includes an operation, \( \text{unit} \), to check the unit of an expression; unit exponents, where \( Q \) can be a rational number; basic units, such as meters or kilograms; and unit products, specifying a new unit. \( \text{UnitExp} \) includes units, tests for unit equality, and a number of logical connectives. Logical operators have their standard precedences, not reflected in the simplified grammar shown here.

The unit annotation language can be used inside annotations tagged as \text{UNITS} annotations. These annotations are changed into custom \text{assert} and \text{assume} statements, with policy-specific handling. CPF[UNITS] will check the \text{asserts} and \text{assumes} by first determining the units of any expressions, based on the current environment. \text{assumes} are then treated like unit assignments, with assignment going from right to left – \( \text{unit}(x) = \text{meter} \) assigns the unit \text{meter} to variable \( x \), while \( \text{unit}(x) = \text{unit}(y) \) assigns the unit of \( y \) to \( x \). By comparison, \text{asserts} are treated as logical checks, with unit comparisons performed using a combination of the units theory from Figure 4, to determine when units are equal, and the concept of unit compatibility used during unit merging when checking expressions. Unannotated functions and objects are treated conservatively, with functions given default annotations and objects assigned fresh units (described above in Section 5.3.3) that allow incorrect uses to be detected.
<table>
<thead>
<tr>
<th>Test</th>
<th>LOC</th>
<th>x100</th>
<th>x400</th>
<th>x4000</th>
<th>x100</th>
<th>x400</th>
<th>x4000</th>
</tr>
</thead>
<tbody>
<tr>
<td>straight</td>
<td>25</td>
<td>6.39</td>
<td>23.00</td>
<td>229.80</td>
<td>0.06</td>
<td>0.06</td>
<td>0.06</td>
</tr>
<tr>
<td>ann</td>
<td>27</td>
<td>8.62</td>
<td>31.27</td>
<td>307.54</td>
<td>0.09</td>
<td>0.08</td>
<td>0.08</td>
</tr>
<tr>
<td>nosplit</td>
<td>69</td>
<td>12.71</td>
<td>46.08</td>
<td>467.89</td>
<td>0.13</td>
<td>0.12</td>
<td>0.12</td>
</tr>
<tr>
<td>split</td>
<td>69</td>
<td>27.40</td>
<td>106.55</td>
<td>1095.34</td>
<td>0.27</td>
<td>0.27</td>
<td>0.27</td>
</tr>
</tbody>
</table>

Times in seconds. All times averaged over three runs of each test. LOC (lines of code) are per function, with 100, 400, or 4000 identical functions in a source file.

Fig. 10. CPF[UNITS] Performance

5.4 Running CPF[UNITS]

As an example of the use of CPF[UNITS], Figure 1 shows a portion of a C program that uses units. As mentioned in Section 1, it is not obvious that this code contains a unit error. By adding the annotations shown in the figure, CPF[UNITS] can check the program for unit errors. This gives the following output:

Function finalEnergy: ERROR on line 26(1): Assert failed!

This message shows that the code actually has a unit error, in this case on line 26. The potential cause of this error was explained in Section 1. Any assertion failures are similarly reported to the user, with additional information provided where this is possible. For instance, errors triggered by addition operations will report the line number and the fact that the error is caused by an invalid addition.

6 Evaluation

Evaluation was performed using two sets of experiments. All tests were performed on the same computer, a Pentium 4 3.40 GHz with 2 GB RAM running Gentoo Linux and Maude 2.3. In the first, the focus was on performance, ensuring that using a per-function analysis would scale well as desired. The results are shown in Figure 10. Here, each test performs a series of numerical calculations: straight includes straight-line code; ann includes the same code as straight with a number of added unit annotations; nosplit includes a number of nested conditionals that change units on variables uniformly, leaving just one environment; and split includes nested conditionals that change variable units non-uniformly in different branches, yielding eight different environments in which statements will need to be evaluated. LOC gives the lines of code count, derived using the CCCC tool [22], for each function, with the same function repeated 100, 400, or 4000 times.

As shown in Figure 10, performance scales almost linearly: quadrupling the number of functions to check roughly quadruples the total processing time. Per-function processing time is small, making CPF[UNITS] a realistic option for checking individual functions during development, something not possible in some other solutions (such as C-UNITS) that require the entire program be checked at once. Splitting environments increases the execution time, but not prohibitively: with
eight environments, the time per function to process \texttt{split} is roughly double, not eight times, that to process \texttt{nosplit}, which has just one environment. Finally, processing annotations in the units annotation language seems to add little overhead; annotations are treated as statements during processing, so in some sense just add additional “hidden” lines of code.

The second set of experiments compares against some of the same examples used by Osprey, some of which were originally from C-UNITS, with the results shown in Figure 11. \texttt{fe.c} is the example shown in Figure 1; \texttt{coil.c} is part of an electrical inductance calculator; \texttt{projectile.c} calculates the launch angle for a projectile; and \texttt{projectile-bad.c} does the same, but with an intentionally-introduced unit error. \texttt{big0.c}, \texttt{big1.c}, and \texttt{big2.c} include a repeated series of arithmetic operations and are designed to test the size of function that CPF[UNITS] can handle, with \texttt{big2.c} included as an especially unrealistic example.

Overall, Figure 11 shows that the annotation burden is not heavy: assumptions on variable declarations are sometimes needed, while preconditions and postconditions are often needed, with the number of annotations needed by Osprey being similar (although those used by Osprey are generally smaller). \texttt{big0.c}, \texttt{big1.c}, and \texttt{big2.c} require no annotations, while \texttt{coil.c} requires 14, including on function prototypes. \texttt{fe.c} requires 9 annotations, with \texttt{ex18.c} requiring 10. The \texttt{projectile.c} example is particularly interesting: the use of a more flexible annotation language allows a more general version of the program to be checked than in some other systems (as discussed in Section 2), maintaining unit polymorphism, while \texttt{projectile-bad.c} includes an error not caught by Osprey, since the error involves using a variable with a different unit (pounds versus kilograms) in the same dimension. Overall, only 16 annotations are needed across 5 functions and 2 prototypes in both \texttt{projectile.c} and \texttt{projectile-bad.c}. \texttt{coil.c} shows a disadvantage of the CPF[UNITS] approach: one of the \texttt{goto} statements never stabilizes, meaning the units keep changing with each iteration. This raises an error in the program, which in this case appears to be a false positive.

7 Future Work and Conclusions

This paper presented CPF[UNITS], a static analysis tool based on an abstract rewriting logic semantics of C, designed for checking the unit safety of C programs. This tool provides a modular, scalable method of detecting unit violations. Unlike
many type or library based approaches, CPF[UNITS] requires no changes to the base language, and can support relationships between the units of formal parameters, local variables, and function return values via annotations. Finally, the use of a modular framework, the C Policy Framework, and an underlying abstract semantics in rewriting logic allow for the rapid testing of new features and extensions, such as extensions to the annotation language.

There are several areas where CPF[UNITS] could be extended. First, some C code cannot yet be safely analyzed. This includes code that uses features that are not type-safe, such as pointer arithmetic and unions, as well as code that uses ambiguous function pointers. Extending the CPF[UNITS] definition, while using additional analysis information from CIL, should make it possible to safely handle more of these cases. Second, a number of conservative assumptions around aliasing and global variables preserve correctness but can generate warnings; additional analysis information from CIL should also be useful in these cases, to sharpen the analysis capabilities without losing correctness. Third, annotations on global variables and structure definitions would allow assumptions about units associated with globals or instances of structures to be stated once, instead of stating them in functions which use them; these are currently being added. Fourth, error messages are being improved. Finally, there are some useful annotations that cannot yet be properly handled, including unit annotations that depend on variables in the exponent (such as saying that, given an integer $n$, variable $x$ has unit meter$^n$). Extending the capabilities of the annotation language would increase the power of CPF[UNITS], allowing it to handle more complex cases.

References

Hills and Chen and Roșu


