
# Within ARM's Reach: Compilation of Left-Linear Rewrite Systems via Minimal Rewrite Systems

W.J. Fokkink, J.F.Th. Kamperman, H.R. Walters

Software Engineering (SEN)

**SEN-R9721 November 30, 1997**

Report SEN-R9721
ISSN 1386-369X

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

# Within ARM's Reach: Compilation of Left-Linear Rewrite Systems via Minimal Rewrite Systems

W.J. Fokkink

W.J.Fokkink@swan.ac.uk, http://www.swan.ac.uk/compsci/AllStaff/WJF.html.

*University of Wales Swansea*

*Department of Computer Science, Singleton Park, Swansea SA2 8PP, Wales*

J.F.Th. Kamperman

J.Kamperman@idr.nl, http://www.idr.nl/~jasper.

*ID Research*

*Groningenweg 6, 2803 PV Gouda, The Netherlands*

H.R. Walters

pum@babelfish.nl, http://www.babelfish.nl.

*Babelfish*

*Korenbloemweg 23, 2403 GA Alphen a/d Rijn, The Netherlands*

and

*CWI*

*P.O. Box 94079, 1090 GB Amsterdam, The Netherlands*

ABSTRACT

A new compilation technique for left-linear term rewriting systems is presented, where rewrite rules are transformed into so-called minimal rewrite rules. These minimal rules have such a simple form that they can be viewed as instructions for an abstract rewriting machine (ARM).

*1991 Mathematics Subject Classification:* 68N20: Compilers and Generators; 68Q05: Models of Computation; 68Q42: Rewriting Systems and 68Q65: Algebraic specification

*1991 Computing Reviews Classification System:* D.3.4 [Programming Languages]: Processors - Compilers; D.1.1 [Programming Techniques]: Applicative (Functional) Programming; F.3.2 [Logics and Meanings of Programs]: Semantics of Programming Languages, *Algebraic approaches to semantics*

*Keywords and Phrases:* minimal term rewriting systems, abstract machines, program transformations

*Note:* Part of this work is carried out under project SEN1.2 "Software development"

## 1. Introduction

A standard technique for speeding up the execution of a program in a formal (programming) language is compilation of the program into the language of a concrete machine (e.g., a microprocessor). In compiler construction (c.f. [1]), it is customary

to use an abstract machine as an abstraction of the concrete machine. On the one hand, this allows to hide details of the concrete machine in a small part of the compiler, and thus an easy reimplementation on other concrete machines. On the other hand, a good design of the abstract machine enables a simple mapping from source language into abstract machine language. A compiler consists of zero or more transformations in the semantic domain of its source language, followed by a mapping to a lower-level language. This procedure is repeated until the level of the concrete machine is reached.

This paper presents a compilation technique for (single-sorted, unconditional) left-linear term rewriting systems (TRSs). A rewrite rule is called left-linear if it does not contain multiple occurrences of the same variable in its left-hand side. The compilation is partly performed within the well-known source language domain: a left-linear TRS is first transformed into a so-called minimal TRS (MTRS). The basic restriction on a minimal rewrite rule is that it is not allowed to contain more than two occurrences of function symbols at each side of the rule, and no more than three occurrences of function symbols in total. Furthermore, only little difference is allowed between the variable configurations on either side of a minimal rewrite rule. The transformation of left-linear TRS into MTRS is based on a pattern match algorithm using automata, from Hoffmann and O'Donnell [22]. As a result of its simple pattern, the application of a minimal rewrite rule is an elementary operation. An MTRS can therefore be transformed into a program for an abstract rewriting machine (ARM). The instructions of ARM are such that they can be implemented efficiently on modern microprocessors. A previous version of ARM was introduced in [26], while the transformation from left-linear TRS to ARM code was first described in [27]. A more up-to-date overview was given in [25].

For the sake of compilation, it is imperative to fix a deterministic rewriting strategy. We apply rightmost innermost rewriting, which reduces the redex that is closest to the rightmost leaf of the parse tree of a term. The innermost strategy is known to be an efficient implementation method. Furthermore, we impose a specificity ordering on rewrite rules, such that if the left-hand sides of two different rewrite rules can be unified, then the rewrite rule with the more specific left-hand side has higher priority. This choice makes sense, because reversal of this priority would mean that the rewrite rule with the more specific left-hand side would never applied. We will give a precise formal definition of specificity ordering, and show that it is an efficient implementation method. Most implementations of functional languages are based on so-called textual orderings, where the priority of a rewrite rule depends on the position of this rule in the layout of the TRS. Textual orderings have an unclear semantics, and often hamper the efficiency and clarity of implementations.

The reduction of a term by the original left-linear TRS, with respect to our preferred rewriting strategy, is performed by the resulting ARM program as follows. First, the function symbols and variables of the term are collected on a so-called control stack, in a rightmost innermost fashion. The elements are popped from the control stack, one by one, to trigger the execution sequences of ARM instructions that belong to rewrite rules in the MTRS. This execution simulates the reduction of the term by the left-linear TRS, and if this reduction terminates, then the execution will produce the resulting normal form. In contrast with other compilation

techniques for functional languages, this technique also applies to open terms.

To speed up the compilation of MTRSs into ARM code, we introduce the notion of a locus, which assigns a natural number to each function symbol. Intuitively, the locus indicates the point of interest in the list of arguments of a function symbol. For example, if a minimal rule contains three function symbols, then the loci of the function symbols at the head of the left- and right-hand side of the rule indicate which argument at the left- or right-hand side contains a function symbol. If this argument is at the left-hand side, then the locus allows fast pattern-matching with respect to this argument. If this argument is at the right-hand side, then the locus allows fast continuation of innermost rewriting at this argument. If the locus of an MTRS satisfies these, and other related requirements, then it will be called a stratification.

In principle, our compilation technique can handle TRSs that are not left-linear. However, such TRSs require checks on syntactic equality, which have a complexity that is related to the sizes of the terms to be checked. An efficient way to deal with TRSs that are not left-linear is to define an equality function $eq(s,t)$, which evaluates to true if and only if $s$ and $t$ are syntactically equal. This equality function can be incorporated in the rewrite rules, to eliminate multiple occurrences of the same variable in the left-hand side of a rewrite rule; see [25, page 28] for an elaborate example.

We do not consider conditional TRSs [8], where rewrite rules are allowed to carry conditions. A sensible way to compile a conditional TRS properly, is to eliminate the conditions in its rewrite rules first. For example, conditions of the form $s \downarrow t$, i.e., $s$ and $t$ have the same normal form, can be expressed by means of an equality function. Therefore, the problems involved with the implementation of conditions in rewrite rules are orthogonal to the matters that are investigated in this paper. Bergstra and van den Brand [6] have implemented an efficient compiler which eliminates the conditions from a conditional TRS.

We consider only single-sorted signatures, because a TRS over a many-sorted signature can be treated as a TRS over a single-sorted signature, after it has been type-checked. That is, suppose that a TRS over a many-sorted signature is to rewrite a term over this signature. Then a parser should first check whether the rewrite rules and the term satisfy the syntactic restrictions that are imposed by many-sortedness. If this is the case, then the reducts of the term will all satisfy these syntactic restrictions automatically, so that types can be ignored.

This research was started with the aim to support the equational language ASF+ SDF [31, 15], which is a combination of Algebraic Specification Formalism [7] and Syntax Definition Formalism [21]. Our main goals are a well-structured, clearly described and efficient implementation. The detailed description of the implementation is given in the current paper, where the reader may convince her/himself that the implementation is well-structured. The compilation of left-linear TRS into ARM code increases the number of rewrite steps in a linear fashion. The complexity of executing a single rewrite step, however, decreases. In practice, this leads to comparable performance. In [20, Table 9], favourable execution times were reported concerning the equational programming language Epic [48, 49], which has been implemented by means of the ARM methodology. Based on these experiences, and by

further insights reported in this paper, it can be stated that the compilation of a left-linear TRS via a stratified MTRS into an ARM program leads to an efficient implementation.

Questions on the correctness of compilation of programming languages date back to McCarthy [37]. The question whether our compilation strategy for left-linear TRSs is correct can be answered in several ways. Firstly, the intuitive explanations that are given for the transformations, from left-linear TRS to stratified MTRS and then to ARM code, help to understand why the compilation is correct. Secondly, the technology has been tested thoroughly, in the sense that it has been implemented, and works satisfactorily, in Epic. An outline of a formal correctness proof for the transformation of left-linear TRS into stratified MTRS was presented in [27, 25]. That proof is based a new notion of simulation, which relates the reductions graphs of the transformed TRS to the reductions graphs of the original TRS. If such a simulation is proved to be sound, complete and termination preserving, then it can be concluded that the transformation constitutes a correct compilation step, see [16]. This technique can also be applied to prove correctness of the transformation from stratified MTRS into ARM code. However, such a correctness proof is beyond the scope of the present paper.

This paper is set up as follows. Section 2 presents the necessary preliminaries from term rewriting. In Section 3 the syntactic format for MTRSs is defined, and it is shown how to transform a left-linear TRS into a stratified MTRS. In Section 4 the syntax and semantics of ARM are given, and it is shown how to transform an MTRS into an ARM program. Finally, Section 5 discusses related work.

**Acknowledgements.** We would like to thank Jan Bergstra, Mark van den Brand, Jan Heering, Paul Klint, Jaco van de Pol, and John Tucker for their support.

## 2. Preliminaries

This section introduces some preliminaries from term rewriting; for more background see e.g. [14, 32].

### *2.1 Term Rewriting Systems*

**Definition 2.1** *A* signature $\Sigma$ *consists of:*

- *a countably infinite set $\mathcal{V}$ of variables $u, v, w, x, y, z, ...$;*
- *a non-empty set $\mathcal{F}$ of function symbols $f, g, h, ...$, disjoint with $\mathcal{V}$, where each function symbol $f$ is provided with an arity $ar(f)$, being a natural number.*

*Function symbols of arity 0 are called* constants. *In the sequel, $=$ denotes syntactic equality between terms, and $=_\alpha$ denotes syntactic equality modulo $\alpha$-conversion, i.e., modulo renaming of variables.*

In the next definitions we assume a signature $\Sigma = (\mathcal{V}, \mathcal{F}, ar)$.

**Definition 2.2** $\mathbb{T}(\Sigma)$ *denotes the set of* terms $\ell, p, q, r, s, t, ...$ *over* $\Sigma$, *being the smallest set satisfying:*

- $\mathcal{V} \subset \mathbb{T}(\Sigma)$;

- *if* $f \in \mathcal{F}$ *and* $t_1, ..., t_{ar(f)} \in \mathbb{T}(\Sigma)$, *then* $f(t_1, ..., t_{ar(f)}) \in \mathbb{T}(\Sigma)$.

A (possibly empty) sequence $t_1, ..., t_k$ of terms will sometimes be abbreviated to vector notation $\vec{t}$, and $|\vec{t}|$ will denote the length $k$ of this sequence.

**Definition 2.3** *A* substitution *is a mapping* $\sigma : \mathcal{V} \to \mathbb{T}(\Sigma)$. *Each substitution is extended to a mapping from terms to terms in the standard way.*

**Definition 2.4** *A* rewrite rule *is an expression* $\ell \to r$ *with* $\ell$ *and* $r$ *terms, where:*

1. *the left-hand side* $\ell$ *is not a single variable;*
2. *all variables that occur in the right-hand side* $r$ *also occur in the left-hand side* $\ell$.

*A* term rewriting system *(TRS)* $\mathcal{R}$ *consists of a finite set of rewrite rules.*

**Definition 2.5** *A rewrite rule* $\ell \to r$ *is* left-linear *if each variable occurs no more than once in its left-hand side* $\ell$.

*A TRS is* left-linear *if all its rules are so.*

**Definition 2.6** *A rewrite rule* $f(x_1, ..., x_k) \to r$ *with* $x_1, ..., x_k$ *distinct variables is* most general.

*A TRS* $\mathcal{R}$ *is* simply complete *if for each* $f$ *for which there is a rule in* $\mathcal{R}$ *with left-hand side* $f(\vec{t})$, *there is also a most general rule in* $\mathcal{R}$ *with left-hand side* $f(\vec{x})$.

In the next section we will define for each left-linear TRS $\mathcal{R}$ over a signature $\Sigma$, a binary rewrite relation $\to_{\mathcal{R}}$ on $\mathbb{T}(\Sigma)$ which allows at most one possible reduction for each term in $\mathbb{T}(\Sigma)$.

**Definition 2.7** $t \in \mathbb{T}(\Sigma)$ *is a* normal form *for a rewrite relation* $\to_{\mathcal{R}}$ *if there does not exist a* $t' \in \mathbb{T}(\Sigma)$ *with* $t \to_{\mathcal{R}} t'$.

*2.2 Rewriting Strategy*

A deterministic rewriting strategy is determined by two priorities:

1. if the left-hand sides of several rewrite rules match with the same term, then it selects which rewrite rule is preferred to reduce this term;
2. if several subterms of a term match with left-hand sides of rewrite rules, then it selects which of these subterms is actually reduced.

For the first preference we adopt specificity ordering, meaning that if the left-hand sides of two different rewrite rules can be unified, then the rewrite rule with the most specific left-hand side has higher priority. Specificity ordering is based on ideas in [5], where the semantics of such orderings on term rewriting systems was studied thoroughly for the first time.

For the second preference we adopt rightmost innermost rewriting, which selects the subterm closest to the rightmost leaf of the parse tree of the term. In general, the innermost rewriting strategy is an efficient implementation method for rewrite systems.

First, we present the precise definition of specificity ordering.

**Definition 2.8** *The syntactic specificity ordering $<$ on terms is defined by:*

- $x < f(t_1, ..., t_{ar(f)})$;
- $f(s_1, ..., s_{ar(f)}) < f(t_1, ..., t_{ar(f)})$ *if* $s_1 =_\alpha t_1, ..., s_{i-1} =_\alpha t_{i-1}, s_i < t_i$, *for some* $i \in \{1, .., ar(f)\}$.

*The specificity ordering $\prec$ is defined on rewrite rules by*

$$\ell < \ell' \;\Rightarrow\; \ell \to r \prec \ell' \to r'.$$

If two terms $\ell$ and $\ell'$ can be unified, say $\sigma(\ell) = \sigma'(\ell')$ for certain substitutions $\sigma$ and $\sigma'$, then it is easy to see that either $\ell < \ell'$ or $\ell' < \ell$ or $\ell =_\alpha \ell'$. So in order to avoid situations in which two rewrite rules apply to the same term, and neither of these rules has priority over the other, it suffices to exclude TRSs that contain two different rules of which the left-hand sides are equal modulo $\alpha$-conversion. In practice, such ambiguities can be resolved by giving one of these rules priority over the other, in which case the rule with lower priority is never applied.

We proceed to present the precise definition of rightmost innermost rewriting, with respect to specificity ordering.

**Definition 2.9** *Given a TRS $\mathcal{R}$ over signature $\Sigma$, the binary rewrite relation $\to_\mathcal{R}$ is defined on $\mathbb{T}(\Sigma)$ inductively as follows.*

1. *All variables in $\mathcal{V}$ are normal forms for $\to_\mathcal{R}$.*
2. *Assume that we already defined $\to_\mathcal{R}$ for $t_1, ..., t_{ar(f)}$. Then $\to_\mathcal{R}$ is defined for $f(t_1, ..., t_{ar(f)})$ as follows.*
   - (RIGHTMOST INNERMOST) *If, for some $i \in \{1, ..., ar(f)\}$, $t_{i+1}, ..., t_{ar(f)}$ are normal forms for $\to_\mathcal{R}$, and $t_i \to_\mathcal{R} s$, then*
     $$f(t_1, ..., t_{ar(f)}) \to_\mathcal{R} f(t_1, ..., t_{i-1}, s, t_{i+1}, ..., t_{ar(f)}).$$
   - (SPECIFICITY) *Suppose that $t_1, ..., t_{ar(f)}$ are normal forms for $\to_\mathcal{R}$. If $\ell \to r$ is the greatest rewrite rule in $\mathcal{R}$ (with respect to the specificity ordering $\prec$) such that $f(t_1, ..., t_{ar(f)}) = \sigma(\ell)$ for a certain substitution $\sigma$, then*
     $$f(t_1, ..., t_{ar(f)}) \to_\mathcal{R} \sigma(r).$$
     *If such a rewrite rule does not exist in $\mathcal{R}$, then $f(t_1, ..., t_{ar(f)})$ is a normal form for $\to_\mathcal{R}$.*

$\to_\mathcal{R}$ is a subrelation of the standard rewrite relation for $R$ without any rewriting strategy. Furthermore, normal forms for $\to_\mathcal{R}$ are also normal forms for this standard rewrite relation.

Since the subscript $\mathcal{R}$ is usually clear from the context, in general it will be omitted from $\to_\mathcal{R}$. The overloading of $\to$ is by convention.

## 3. Minimal Term Rewriting Systems

In this section it is shown how each left-linear TRS can be transformed into a simply complete TRS which contains only 'minimal' rewrite rules. These rules have such a simple form that they can be viewed as instructions for an abstract rewriting machine. Furthermore, function symbols will be supplied with a 'locus' value, such that the minimal TRS (MTRS) is 'stratified'. These locus values will be important for the efficient compilation of a minimal TRS into an abstract rewriting machine, in Section 4.

### *3.1 Minimal Rewrite Rules*

A minimal rewrite rule is left-linear, and is not allowed to contain more than two occurrences of function symbols at each side of the rule, and no more than three occurrences of function symbols in total. Furthermore, only little difference is allowed between the variable configurations on either side of a minimal rewrite rule.

**Definition 3.1** *A rewrite rule is called* minimal *if it is left-linear and has one of the following five forms:*

$$\begin{array}{lrcll} \text{M1} & f(\vec{x}, g(\vec{y}), \vec{z}) & \to & h(\vec{x}, \vec{y}, \vec{z}) & \\ \text{M2} & f(\vec{x}, \vec{y}, \vec{z}) & \to & h(\vec{x}, g(\vec{y}), \vec{z}) & \\ \text{M3} & f(\vec{x}, \vec{y}) & \to & h(\vec{x}, z, \vec{y}) & z \in \vec{x}, \vec{y} \\ \text{M4} & f(\vec{x}, \vec{y}, \vec{z}) & \to & h(\vec{x}, \vec{z}) & \\ \text{M5} & f(\vec{x}, y) & \to & y & \end{array}$$

*A TRS is* minimal *is all its rules are so.*

### *3.2 Locus*

For the sake of the efficiency of the compilation of MTRSs, we will need a so-called locus, which maps function symbols to natural numbers. A basic characteristic of a locus is that function symbols that occur in normal forms have locus 0. Since we use the innermost rewriting strategy, this implies that function symbols that occur inside the left-hand side of a rewrite rule should have locus 0. For convenience we require the same for function symbols that occur inside the right-hand side of a rewrite rule.

**Definition 3.2** *A* locus *for a TRS $\mathcal{R}$ over $(\mathcal{V}, \mathcal{F}, ar)$ is a function $L : \mathcal{F} \to \mathbb{N}$ such that:*

1. *whenever $L(f) \neq 0$, there is a most general rule in $\mathcal{R}$ with left-hand side $f(\vec{x})$;*
2. *for each left- or right-hand side $h(\vec{t})$ of a rewrite rule in $\mathcal{R}$, the function symbols that occur in $\vec{t}$ all have locus 0.*

In the case of an MTRS, we want the locus to be a 'stratification', so that it provides valuable information for implementation purposes.

- For minimal rules of type M1, the locus should indicate which argument at the left-hand side contains a function symbol, to enable fast pattern matching with respect to such arguments.

- For minimal rules of type M2, the locus should indicate which argument at the right-hand side contains a function symbol, to enable fast continuation of (innermost) rewriting at such arguments.
- For minimal rules of type M3, the locus should indicate at which position in the right-hand side an argument from the left-hand side has to be copied.
- For minimal rules of type M4 with $|\vec{y}| \neq 0$, the locus should indicate which arguments at the left-hand side have to be deleted.

These intuitions are incorporated in the following definition.

**Definition 3.3** *Assume an MTRS $\mathcal{M}$ over $(\mathcal{V}, \mathcal{F}, ar)$, and a locus $L : \mathcal{F} \to \mathbb{N}$. The pair $(\mathcal{M}, L)$ is called* stratified *if (using the notations from Definition 3.1 for rules of types M1-5):*

1. *for each rule in $\mathcal{M}$ of type M1, $L(f) = L(h) = |\vec{x}|$;*
2. *for each rule in $\mathcal{M}$ of type M2, $L(f) = L(h) = |\vec{x}|$;*
3. *for each rule in $\mathcal{M}$ of type M3, $L(f) = L(h) = |\vec{x}|$;*
4. *for each rule in $\mathcal{M}$ of type M4 with $|\vec{y}| \neq 0$, $L(f) = L(h) = |\vec{x}|$;*
5. *for each rule in $\mathcal{M}$ of type M5, $L(f) = |\vec{x}|$.*

Note that a stratification does not impose restrictions on the loci of function symbols $f$ and $h$ for minimal rewrite rules of the form $f(\vec{x}) \to h(\vec{x})$. The construction of a stratification for an MTRS will depend on the incorporation of such rewrite rules.

We now continue to show how to transform a left-linear TRS into a stratified simply complete MTRS; in Section 3.3 the original TRS will be made simply complete, in Sections 3.4 and 3.5 the left- and right-hand sides of rewrite rules will be minimized, respectively, and finally in Section 3.6 the resulting simply complete MTRS will be stratified.

### *3.3 Simple Completeness*

In the transformations that will be described in the next sections, we are going to manipulate function symbols at the head of left-hand sides of rewrite rules. In order to make sure that these manipulations do not affect the resulting normal forms, first we add most general rules, to make the TRS simply complete. Note that in a simply complete TRS, function symbols that occur at the head of a left-hand side do not occur in normal forms. The transformation is an adaptation of a transformation introduced by Thatte [44].

Assume a left-linear TRS $\mathcal{R}$ over $\Sigma = (\mathcal{V}, \mathcal{F}, ar)$. The following program adds most general rules, to obtain a simply complete left-linear TRS.

Procedure "Add Most General Rules" applied to $(\mathcal{R}, \mathcal{F}, ar)$.

If $\mathcal{R}$ is simply complete, then output $(\mathcal{R}, \mathcal{F}, ar)$.

Else, select a function symbol $f \in \mathcal{F}$ for which there is a rule in $\mathcal{R}$ with left-hand side $f(\vec{t})$, but no most general rule with left-hand side $f(\vec{x})$. Add a fresh function symbol $f^c$ to $\mathcal{F}$, of arity $ar(f)$, and add a most general rule

$$f(\vec{x}) \;\to\; f^c(\vec{x})$$

For all left-hand sides $g(\vec{s})$ of rules in $\mathcal{R}$ (with $g = f$ as well as $g \neq f$), replace each occurrence of $f$ in $\vec{s}$ by $f^c$.

The resulting TRS is still left-linear, because the new rewrite rule is so. Apply the procedure "Add Most General Rules" to this left-linear TRS.

**Intuition** The intuition behind the transformation of $\mathcal{R}$ is as follows. The new rewrite rule $f(\vec{x}) \to f^c(\vec{x})$ is the desired most general rule with left-hand side $f(\vec{x})$. Innermost rewriting and specificity ordering together ensure that this rewrite rule is innocuous, in the sense that it only replaces occurrences of $f$ at the head of normal forms by $f^c$ ($c$ is mnemonic for 'constructor'). Since occurrences of $f$ inside left-hand sides of rewrite rules in $\mathcal{R}$ only apply to normal forms, due to innermost rewriting, such occurrences of $f$ are replaced by $f^c$.

**Termination** The program above terminates for each allowed input. Namely, each application of the procedure strictly decreases the number of function symbols $f$ for which there is a rewrite rule with left-hand side $f(\vec{t})$, but no most general rule with left-hand side $f(\vec{x})$.

Since the program "Add Most General Rules" is terminating, it produces a simply complete left-linear TRS. We provide this TRS with a locus by defining $L(f) = 0$ for all function symbols $f$. Note that this locus trivially satisfies the two requirements of Definition 3.2.

### *3.4 Minimization of Left-Hand Sides*

Assume a simply complete left-linear TRS $\mathcal{R}$ over $\Sigma = (\mathcal{V}, \mathcal{F}, ar)$, with a locus $L : \mathcal{F} \to \mathbb{N}$. The following program transforms $\mathcal{R}$ in such a way that the left-hand sides of its non-minimal rules contain only one function symbol.

Procedure "Minimize Left-Hand Sides" applied to $(\mathcal{R}, L, \mathcal{F}, ar)$.

If each non-minimal rewrite rule in $\mathcal{R}$ contains only one function symbol its left-hand side, then output $(\mathcal{R}, L, \mathcal{F}, ar)$.

Else, let $i$ be the smallest index for which there is a left-hand side $f(\vec{x}, g(\vec{t}), \vec{s})$ of a non-minimal rewrite rule in $\mathcal{R}$ with $|\vec{x}| = i$.

For each $g \in \mathcal{F}$ for which there exists a non-minimal rewrite rule of the form $f(\vec{x}, g(\vec{t}), \vec{s}) \to r$ with $|\vec{x}| = i$, introduce a fresh function symbol $f_g$, with arity $ar(f) + ar(g) - 1$ and locus $i$. Replace each such rule $f(\vec{x}, g(\vec{t}), \vec{s}) \to r$ with $|\vec{x}| = i$ by a rule

$$f_g(\vec{x}, \vec{t}, \vec{s}) \to r \tag{3.1}$$

Furthermore, for each $f_g$ add a left-linear rule

$$f(\vec{x}, g(\vec{y}), \vec{z}) \to f_g(\vec{x}, \vec{y}, \vec{z}) \tag{3.2}$$

If for some $f_g$ there is not yet a most general rule, then add a fresh function symbol $f^d$ to $\mathcal{F}$, of arity $ar(f)$ and with locus $i$ ($d$ is mnemonic for 'duplicate'). For each such $f_g$ add a most general rule

$$f_g(\vec{x},\vec{y},\vec{z}) \to f^d(\vec{x},g(\vec{y}),\vec{z}) \tag{3.3}$$

In this case, replace all left-hand sides of rewrite rules in $\mathcal{R}$ of the form $f(\vec{v},\vec{q})$ with $|\vec{v}| > i$ by $f^d(\vec{v},\vec{q})$, and add a most general rule

$$f(\vec{w}) \to f^d(\vec{w}) \tag{3.4}$$

The resulting TRS is still simply complete, because among rules (3.1) and (3.3) there is a most general rule for each $f_g$, and if $f^d$ is introduced, then the most general rule for $f$ in the original TRS (which exists due to simple completeness) becomes a most general rule for $f^d$ in the resulting TRS, and rule (3.4) is a most general rule for $f$. Furthermore, the resulting TRS is still left-linear, because all the new rules are so. Finally, the extension of $L$ to the fresh function symbols $f^d$ and $f_g$ is still a locus, because there are most general rules for all these function symbols, and they do not occur *inside* any left- or right-hand side of the resulting TRS.

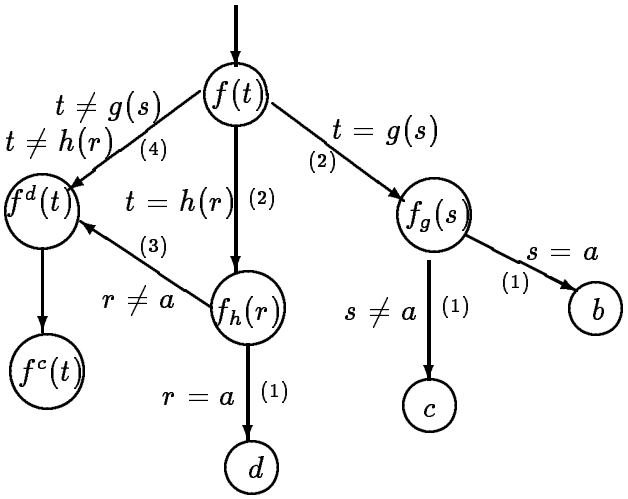
Apply the procedure "Minimize Left-Hand Sides" to the resulting TRS.

The procedure "Minimize Left-Hand Sides" is based on an efficient pattern-matching strategy using automata similar to Hoffmann and O'Donnell [22] (see also [47, Chapter 3]). We give an example.

**Example 3.4** *Let $a$, $b$, $c$ and $d$ be constants, and $f$, $f^c$, $g$ and $h$ unary functions. Consider the following simply complete left-linear TRS:*

$$\begin{array}{rcl} f(g(a)) & \longrightarrow & b \\ f(g(x)) & \longrightarrow & c \\ f(h(a)) & \longrightarrow & d \\ f(x) & \longrightarrow & f^c(x) \end{array}$$

*Pattern-matching of a term $f(t)$ with respect to the left-hand sides of this TRS can be expressed by the following automaton.*




*Note that matching with respect to the function symbol $g$ in the left-hand sides of the first and the second rule is shared. The procedure "Minimize Left-Hand Sides"*

*transforms the original TRS into a TRS that mimics this automaton. The labels attached to a transition yield the syntactic requirement under which this transition is applied, together with the number of the minimal rule in the procedure "Minimize Left-Hand Sides" that captures this transition. The state names correspond with the reducts of $f(t)$ after application of these minimal rules.*

**Intuition** The intuition behind the transformation of $\mathcal{R}$ is as follows. The rules (3.2) constitute a first step towards checking whether a term $p$ matches with a left-hand side in $\mathcal{R}$ of the form $f(\vec{x}, g(\vec{t}), \vec{s})$ with $|\vec{x}| = i$. If a rule in (3.2) reduces a term $p$ to a term $p'$, then there are two possibilities:

- either $p$ indeed matches with the left-hand side of a rule $f(\vec{x}, g(\vec{t}), \vec{s}) \to r$ with $|\vec{x}| = i$ in $\mathcal{R}$, in which case $p'$ can be reduced to $r$ by a rule (3.1);
- or such matchings all fail, in which case a rule (3.3) reduces $p'$ back to $p$, where a superscript $d$ is attached to $f$, to avoid that $p$ is matched with a rule in (3.2) again.

Rewrite rules in $\mathcal{R}$ of the form $f(\vec{v}, \vec{q}) \to r$ with $|\vec{v}| > i$ only apply to terms which do not match with any left-hand sides in $\mathcal{R}$ of the form $f(\vec{x}, g(\vec{t}), \vec{s})$ with $|\vec{x}| = i$, due to the specificity ordering. In particular, such rewrite rules may apply to terms which have been reduced subsequently by a rule in (3.2) and a rule in (3.3). So if there exists a rule in (3.3), then rewrite rules $f(\vec{v}, \vec{q}) \to r$ with $|\vec{v}| > i$ in $\mathcal{R}$ are replaced by $f^d(\vec{v}, \vec{q}) \to r$. In this case, rule (3.4) makes sure that the superscript $d$ is also attached to occurrences of $f$ in terms which cannot be reduced by any rules in (3.2).

**Remark 3.5** Since $\mathcal{R}$ is simply complete, the function symbol $f$ does not occur in any of its normal forms. Therefore, the adaptation of $\mathcal{R}$ does not influence the representations of its normal forms. So there is no need to change occurrences of $f$ *inside* left-hand sides into $f^d$; a rewrite rule with an occurrence of $f$ inside its left-hand side is never applied, owing to the innermost rewriting strategy.

Note that the choice of loci for the function symbols $f^d$ and $f_g$, namely $i$, ensures that the minimal rules (3.2) and (3.3) are stratified.

**Termination** The program "Minimize Left-Hand Sides" terminates for each allowed input. Namely, in each subsequent call of the procedure, the total number $N$ of occurrences of function symbols in left-hand sides of non-minimal rules strictly decreases. This can be seen by considering the effect of each separate step in the procedure on the value of $N$.

- Replacing occurrences of function symbol $f$ by function symbol $f^d$ does not influence $N$.
- (3.2), (3.3) and (3.4) introduce minimal rules, which again does not influence $N$.
- In (3.1), for some $f$ and $i$, rewrite rules of the form $f(\vec{x}, g(\vec{t}), \vec{s}) \to r$ with $|\vec{x}| = i$ are replaced by rules $f_g(\vec{x}, \vec{t}, \vec{s}) \to r$. Amongst the rewrite rules that are replaced, there is at least one which is non-minimal. Since the left-hand side of each replacement contains one function symbol less than its original, and minimal rules stay minimal, it follows that the replacements in (3.1) strictly decrease the value of $N$.

Since the program "Minimize Left-Hand Sides" is terminating, it produces a simply complete left-linear TRS with a locus, where the left-hand sides of its non-minimal rules contain only one function symbol.

*3.5 Minimization of Right-Hand Sides*

Assume a simply complete left-linear TRS $\mathcal{R}$ over $\Sigma = (\mathcal{V}, \mathcal{F}, ar)$, in which each non-minimal rewrite rule contains only one function symbol in its left-hand side. Also assume a locus $L : \mathcal{F} \to \mathbb{N}$. The following program minimizes the right-hand sides of rewrite rules in $\mathcal{R}$.

Procedure "Minimize Right-Hand Sides" applied to $(\mathcal{R}, L, \mathcal{F}, ar)$.

If $\mathcal{R}$ is minimal, then output $(\mathcal{R}, L, \mathcal{F}, ar)$.

Else, select a rewrite rule $f(\vec{v}) \to r$ in $\mathcal{R}$ that is not minimal. We distinguish three cases, depending on whether $r$ contains zero, one, or more than one function symbols.

CASE 1: $r = v_k$ for some $k < ar(f)$.

Then add a fresh function symbol $f^d$ to $\mathcal{F}$, of arity $k$ and with locus $k$, and replace the rule $f(\vec{v}) \to r$ in $\mathcal{R}$ by the following two most general rules:

$$\begin{aligned} f(\vec{v}) &\to f^d(v_1, ..., v_k) \\ f^d(v_1, ..., v_k) &\to v_k \end{aligned}$$

CASE 2: $r = h(\vec{w})$.

Since $f(\vec{v}) \to h(\vec{w})$ is not minimal, $\vec{v} = \vec{x}, \vec{y}, \vec{z}$ and $\vec{w} = \vec{x}, \vec{u}, \vec{z}$ where $\vec{u}$ has length $l > 0$, and the first and last element of $\vec{u}$ are not the first and last element of $\vec{y}$, respectively. We call $\vec{u}$ the *non-compliant segment* of $f(\vec{v}) \to h(\vec{w})$.

Then add a fresh function symbol $f^d$ to $\mathcal{F}$, of arity $ar(f) + 1$ and with locus $|\vec{x}|$, and replace the rule $f(\vec{v}) \to r$ in $\mathcal{R}$ by the following two most general rules:

$$\begin{aligned} f(\vec{x}, \vec{y}, \vec{z}) &\to f^d(\vec{x}, u_1, \vec{y}, \vec{z}) \\ f^d(\vec{x}, u', \vec{y}, \vec{z}) &\to h(\vec{x}, u', u_2, ..., u_l, \vec{z}) \end{aligned}$$

where $u'$ is a fresh variable.

CASE 3: $r = h(\vec{w}, g(\vec{s}), \vec{t})$.

Then add a fresh function symbol $h_g$ to $\mathcal{F}$, of arity $ar(h) + ar(g) - 1$ and with locus $|\vec{w}|$, and replace the rule $f(\vec{v}) \to r$ in $\mathcal{R}$ by the following two most general rules:

$$\begin{aligned} f(\vec{v}) &\to h_g(\vec{w}, \vec{s}, \vec{t}) \\ h_g(\vec{x}, \vec{y}, \vec{z}) &\to h(\vec{x}, g(\vec{y}), \vec{z}) \end{aligned}$$

The resulting TRS is still simply complete, because in all cases a most general rule is introduced for the fresh function symbol $f^d$ or $h_g$, and the most general rule $f(\vec{v}) \to r$ is replaced by another most general rule for $f$. Furthermore, the resulting TRS is still left-linear, and its non-minimal rules still contain only

one function symbol in their left-hand sides, because all the new rules satisfy these properties. Finally, the extension of $L$ to the fresh function symbol $f^d$ or $h_g$ is still a locus, because there is a most general rule for each of these function symbols, and they do not occur inside any left- or right-hand side of the resulting TRS.

Apply the procedure "Minimize Right-Hand Sides" to the resulting TRS.

**Intuition** In all cases, the selected non-minimal rewrite rule $f(\vec{v}) \to r$ is replaced by two new rewrite rules, which together are able to simulate the rewrite steps defined by $f(\vec{v}) \to r$. One of these new rules is minimal, while the other is, in a certain sense, "smaller" than the original rule $f(\vec{v}) \to r$ (see the termination argument below).

Note that in all cases the locus of the fresh function symbol $f^d$ or $h_g$ is chosen in such a way that (one of) the minimal rewrite rule(s) that is introduced is stratified.

**Termination** The program above terminates for each allowed input. This can be seen by considering the separate cases in the procedure.

- In case 1, the number of non-minimal rules is decreased by one. Moreover, this number is not increased in the cases 2 and 3. Hence, case 1 can only be applied a finite number of times.
- In case 3, the total number of occurrences of function symbols in right-hand sides of non-minimal rules strictly decreases. Moreover, this number is not increased in case 2. Hence, case 3 can only be applied a finite number of times.
- In case 2, the total sum of lengths of non-compliant segments in right-hand sides of non-minimal rules strictly decreases. Hence, this case can only be applied a finite number of times.

Since the program "Minimize Right-Hand Sides" is terminating, it produces a simply complete MTRS with a locus.

### *3.6 Stratification of Minimal Rules*

Assume a simply complete MTRS $\mathcal{M}$ over $\Sigma = (\mathcal{V}, \mathcal{F}, ar)$, together with a locus $L : \mathcal{F} \to \mathbb{N}$. The following program stratifies $\mathcal{M}$.

Procedure "Stratify" applied to $(\mathcal{M}, L, \mathcal{F}, ar)$.

If $(\mathcal{M}, L)$ is stratified, then output $(\mathcal{M}, L, \mathcal{F}, ar)$.

Else, apply one of the following cases.

CASE 1: Suppose that there is a minimal rule in $\mathcal{M}$ of type M1, say

$$f(\vec{x}, g(\vec{y}), \vec{z}) \ \to \ r$$

with $L(f) \neq |\vec{x}|$. Then select such a rule with $|\vec{x}|$ as small as possible. Add a fresh function symbol $f^d$ to $\mathcal{F}$, of arity $ar(f)$ and with locus $|\vec{x}|$. Replace all left-hand sides of rewrite rules in $\mathcal{R}$ of the form $f(\vec{v}, \vec{q})$ with $|\vec{v}| \geq |\vec{x}|$ by $f^d(\vec{v}, \vec{q})$. Furthermore, add a stratified minimal rule

$$f(\vec{w}) \;\to\; f^d(\vec{w})$$

CASE 2: Suppose that there is a minimal rule in $\mathcal{M}$ of type M2-5, say

$$f(\vec{x}) \;\to\; r$$

where the locus of $f$ does not satisfy the stratification property (see Definition 3.3). Then add a fresh function symbol $f^d$, of arity $ar(f)$ and with the right locus, and replace this minimal rule by the following two minimal rules:

$$\begin{array}{rcl} f(\vec{x}) & \to & f^d(\vec{x}) \\ f^d(\vec{x}) & \to & r \end{array}$$

CASE 3: Suppose that there is a minimal rule in $\mathcal{M}$ of type M1-4, say

$$\ell \;\to\; h(\vec{t})$$

where the locus of $h$ does not satisfy the stratification property (see Definition 3.3). Then add a fresh function symbol $h^d$, of arity $ar(h)$ and with the right locus, and replace this minimal rule by the following two minimal rules:

$$\begin{array}{rcl} \ell & \to & h^d(\vec{t}) \\ h^d(\vec{y}) & \to & h(\vec{y}) \end{array}$$

It is not hard to see that in all three cases the resulting MTRS is still simply complete, and that $L$ extended to the fresh function symbol $f^d$ or $h^d$ still satisfies the two requirements of Definition 3.2.

Apply the procedure "Stratify" to the resulting MTRS.

**Intuition** In case 1, left-hand sides $f(\vec{x}, g(\vec{y}), z)$ with $L(f) \neq |\vec{x}|$ are stratified by introducing a fresh function symbol $f^d$ with locus $|\vec{x}|$, and replacing the occurrences of $f$ at the head of these left-hand sides by $f^d$. In order to preserve the specificity ordering, occurrences of $f$ at the head of left-hand sides of the form $f(\vec{v}, \vec{q})$ with $|\vec{v}| > |\vec{x}|$ are also replaced by $f^d$. Since the transformed left-hand sides only match with terms of the form $f^d(\vec{t})$, a stratified minimal rule $f(\vec{w}) \to f^d(\vec{w})$ is introduced to replace occurrences of $f$ in terms by $f^d$.

**Remark 3.6** Since $\mathcal{M}$ is simply complete, the function symbol $f$ does not occur in any of its normal forms. Therefore, the adaptation of $\mathcal{M}$ does not influence the representations of its normal forms. So there is no need to change occurrences of $f$ inside left-hand sides into $f^d$; a rewrite rule with an occurrence of $f$ inside its left-hand side is never applied, owing to the innermost rewriting strategy.

In the cases 2 and 3, the non-stratified left- or right-hand side of a minimal rule is stratified, respectively. Moreover, a stratified minimal rule is added, such that the two new minimal rules together are able to simulate the rewrite steps of the original minimal rule.

**Termination** The program above terminates for each allowed input. This can be seen by considering the separate cases in the procedure.

- Let $N$ be the number of minimal rules $\ell \to r$ in $\mathcal{M}$ of type M1 such that:

- either $\ell \to r$ has a non-stratified left-hand side;
- or there is a minimal rule $\ell' \to r'$ in $\mathcal{M}$ with a non-stratified left-hand side and $\ell \to r \prec \ell' \to r'$.

In case 1, the number $N$ strictly decreases. Moreover, this number is not increased in the cases 2 and 3. Hence, case 1 can only be applied a finite number of times.

- In the cases 2 and 3, the number of minimal rules with a non-stratified left- or right-hand side strictly decreases, respectively. Hence, these cases can only be applied a finite number of times.

Since the program "Stratify" is terminating, it produces a stratified simply complete MTRS.

*3.7 Example*
We give a toy example of a transformation of a specific left-linear TRS into a stratified simply complete MTRS. Assume the natural numbers, constructed from the constant *zero* and the unary successor function *succ*. The following two left-linear rewrite rules constitute a standard specification of the binary addition function *plus* on the natural numbers:

$$\begin{aligned} plus(zero, y) &\to y \\ plus(succ(x), y) &\to succ(plus(x, y)) \end{aligned}$$

In order to transform this left-linear TRS into a stratified simply complete MTRS, first it is made simply complete by adding a most general rule for *plus*:

$$plus(x, y) \to plus^c(x, y)$$

Next, the locus $L$ is introduced: the function symbols *zero*, *succ*, *plus* and $plus^c$ all have locus 0.

Note that the two original rewrite rules are not minimal. In order to obtain a simply complete MTRS, the procedure "Minimize Left-Hand Sides" replaces the first rule by two rules:

$$\begin{aligned} plus(zero, y) &\to plus_{zero}(y) \\ plus_{zero}(y) &\to y \end{aligned}$$

which relate to rule (3.2) and (3.1), respectively, in this procedure. Even so, the second rule is replaced by:

$$\begin{aligned} plus(succ(x), y) &\to plus_{succ}(x, y) \\ plus_{succ}(x, y) &\to succ(plus(x, y)) \end{aligned}$$

The fresh function symbols $plus_{zero}$ and $plus_{succ}$ both have locus 0. Note that the resulting simply complete MTRS satisfies the stratification criteria from Definition 3.3.

## 4. Abstract Rewriting Machine

We define the syntax and semantics for a simple abstract rewriting machine (ARM), and show how to transform a stratified simply complete MTRS into an ARM program. Each set of rules in the MTRS with the same function symbol at the head of their left-hand sides, is transformed into a sequence of ARM instructions. For brevity of presentation, this sequence is represented as a so-called executable stack. The executable stacks are collected in a table to obtain an ARM program. This program manipulates the elements of three separate stacks, called control, argument and traversal stack, guided by the instructions on the executable stack.

### *4.1 Control, Argument and Traversal Stack*

Given a non-empty set $D$, the collection $Stack(D)$ of stacks over $D$ is the smallest fixpoint of:

- $\varepsilon_D \in Stack(D)$
- $D \subset Stack(D)$
- if $S_1, S_2 \in Stack(D)$, then $S_1 \cdot S_2 \in Stack(D)$

Here, $\varepsilon_D$ represents the empty stack. Table 0.1 presents an axiomatization for stacks; in the remainder of this paper, stacks are considered modulo these axioms.

| | | | |
|---|---|---|---|
| St1 | $(x \cdot y) \cdot z$ | $=$ | $x \cdot (y \cdot z)$ |
| St2 | $\varepsilon_D \cdot x$ | $=$ | $x$ |
| St3 | $x \cdot \varepsilon_D$ | $=$ | $x$ |

Table 0.1: Axioms for Stacks

In running text, we take stacks to grow from right to left, i.e., the leftmost element is the top and the rightmost element is the bottom.

Control Stack: Assume a signature $\Sigma = (\mathcal{V}, \mathcal{F}, ar)$. A *control stack* $C$ is a stack over $\mathcal{F} \cup \mathcal{V} \cup \{bottom\}$, where *bottom* is a special element that is always placed at the bottom of the control stack. This element is added for efficiency reasons; it will avoid having to check on possible emptiness of control stacks.

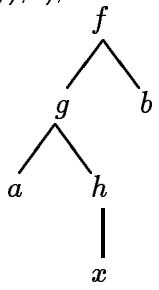
In order to rewrite a specific term $t$ by means of an ARM program, the function symbols and variables of this term are collected on a control stack $control(t)$ in a rightmost innermost fashion.

**Definition 4.1** *The stacks ri-stack$(t)$ for $t \in \mathbb{T}(\Sigma)$ are defined inductively as follows:*

- *ri-stack$(x) = x$*
- *ri-stack$(f(t_1, ..., t_k)) =$ ri-stack$(t_k) \cdot ... \cdot$ ri-stack$(t_1) \cdot f$*

*The control stack $control(t)$ is ri-stack$(t) \cdot bottom$.*

**Example 4.2** *The term $f(g(a,h(x)),b)$, which has the following parse tree,*




*is transformed into the control stack $b \cdot x \cdot h \cdot a \cdot g \cdot f \cdot bottom$.*

The reduction of a term $t$, with respect to some stratified simply complete MTRS, will be performed by popping the function symbols and variables from the control stack of $t$, one by one, and executing instructions related to these elements. Since the parse tree of $t$ was collected on its control stack in a rightmost innermost fashion, this procedure will mimic the rightmost innermost rewriting strategy.

ARGUMENT AND TRAVERSAL STACK: If a function symbol $f$ is popped from the control stack, then the function symbols and variables of its original arguments $t_1, ..., t_{ar(f)}$ were previously collected from this control stack, and used to produce their respective normal forms $s_1, ..., s_{ar(f)}$. These normal forms were subsequently stored on two stacks over $\mathbb{T}(\Sigma)$, called the *argument stack* $A$ and the *traversal stack* $T$. The locus of $f$ tells exactly how the normal forms are divided over these two stacks: $s_{L(f)}, ..., s_1$ are on top of the traversal stack, while $s_{L(f)+1}, ..., s_{ar(f)}$ are on top of the argument stack. These terms are used to obtain the normal form of $f(s_1, ..., s_{ar(f)})$, by means of a number of ARM instructions (see Section 4.3). Subsequently, the terms $s_1, ..., s_{ar(f)}$ are removed from the argument and the traversal stack, and the normal form of $f(s_1, ..., s_{ar(f)})$ is stored on the top of the argument stack.

**Example 4.3** *Figure 0.1 pictures a typical example of the interplay between control, argument and traversal stack. The top elements of the three stacks are at the centre of the figure, while their respective bottom elements are at the edge of the figure. The function symbol $f$ has been popped from $C$, and its arguments $s_{L(f)}, ..., s_1$ and $s_{L(f)+1}, ..., s_{ar(f)}$ are on top of $T$ and $A$, respectively. Function symbol $g$ is on the top of $C$, with its arguments $r_{L(g)}, ..., r_1$ and $r_{L(g)+2}, ..., r_{ar(g)}$ divided over $T$ and $A$, respectively. Note that the argument $r_{L(g)+1}$ of $g$ is yet missing; this will be the normal form of $f(s_1, ..., s_{ar(f)})$.*

When a variable $x$ is popped from the control stack, it is simply pushed onto the argument stack. Namely, a single variable is always a normal form (according to Definition 2.4.1).

Finally, if the element *bottom* is popped from the control stack, then it is concluded that there are no elements left on the control stack. In this situation, the traversal stack will always be empty, and the argument stack will always contain exactly one element, being the normal form of the original term (which was stored on the control stack), with respect to rightmost innermost rewriting and specificity ordering. Then the procedure terminates, producing the term on the argument stack as output.

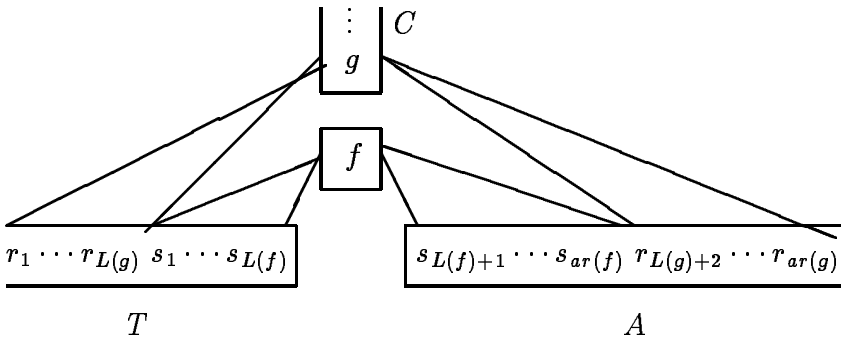



Figure 0.1: Interplay between Control, Argument and Traversal Stack

*4.2 Executable Stack and Program Table*

Suppose that we want to reduce a term by a stratified simply complete MTRS. We already saw that the term is transformed into a control stack. At the same time, the MTRS is transformed into a program, being a table of ARM instructions, as follows.

EXECUTABLE STACK: Each minimal rule in a stratified simply complete MTRS $(\mathcal{M}, L)$ is interpreted as a sequence of ARM instructions (see Section 4.4). For each function symbol $f$, the sequences of machine instructions for minimal rules of the form $f(\vec{s}) \to r$ in $\mathcal{M}$ are gathered on an *executable stack* $E$. These sequences are put in order of priority, with respect to specificity ordering, so as to ensure that machine instructions that belong to a minimal rule with a high priority are executed first.

If a function symbol $f$ is popped from the control stack, then this means that its executable stack is executed. This execution continues until either an instruction on the executable stack of $f$ invokes the execution of another execution stack, or the bottom of the executable stack is reached, in which case a next element is popped from the control stack.

A formalization of executable code falls beyond the scope of this article. However, the two key operations – 'what is the next instruction' and 'what is the remainder of the code after the next instruction' – are very similar to the two common stack operations top and pop, respectively. Hence we ask the reader to indulge us in our simplification of modelling executable code as stacks.

PROGRAM TABLE: Each stratified simply complete MTRS $(\mathcal{M}, L)$ is transformed into a *program table*, denoted by $\mathit{program}(\mathcal{M})$. This table is obtained by collecting the separate executable stacks for all function symbols $f$, where each such stack is provided with the address $f$. (To be more precise, the address is a number related to $f$.)

*4.3 ARM Instructions*

The manipulation of elements on the control, argument and traversal stack is performed by means of a limited number of ARM instructions. We proceed to present these instructions, together with their intuitive meaning. A formal semantics for

ARM is presented in Section 4.5, in Table 0.2. In the following definitions, $f$ and $g$ range over $\mathcal{F}$, and $k$ over $\mathbb{N}$.

- **match**$(f,g)$: If the top element of the argument stack is of the form $f(t_1,...,t_k)$, then replace this term by its arguments $t_1,...,t_k$, and proceed the execution with respect to the function symbol $g$. Otherwise, ignore this instruction.
- **copya**$(k)$: Copy the $k$th term of the argument stack on the top of the argument stack.
- **copyt**$(k)$: Copy the $k$th term of the traversal stack on the top of the argument stack.
- **push**$(f)$: Push $f$ onto the control stack.
- **adrop**$(k)$: Delete the top $k$ terms from the argument stack.
- **tdrop**$(k)$: Delete the top $k$ terms from the traversal stack.
- **skip**$(k)$: Transfer the top $k$ terms from the argument to the traversal stack.
- **retract**$(k)$: Transfer the top $k$ terms from the traversal to the argument stack.
- **build**$(f,k)$: Replace the terms $t_1,...,t_{ar(f)}$ on top of the argument stack by $f(t_1,...,t_{ar(f)})$. The argument $k$ always equals $ar(f)$.
- **goto**$(f)$: Proceed the execution with respect to the function symbol $f$.
- **recycle**: Proceed the execution with respect to the top element on the control stack.

The arity of the function symbol $f$ is provided to the **build** instruction as an explicit second argument to support efficient implementation: if the arity is at hand it need not be looked up in a table.

*4.4 Transformation of MTRS into ARM Code*

The minimal rewrite rules of a stratified simply complete MTRS $(\mathcal{M},L)$ are transformed into sequences of ARM instructions as follows.

| | | | | | |
|---|---|---|---|---|---|
| M1 | $f(\vec{x},g(\vec{y}),\vec{z})$ | $\to$ | $h(\vec{x},\vec{y},\vec{z})$ | | **match**$(g,h)$ |
| M2 | $f(\vec{x},\vec{y},\vec{z})$ | $\to$ | $h(\vec{x},g(\vec{y}),\vec{z})$ | | **push**$(h)\cdot$**goto**$(g)$ |
| M3 | $f(\vec{x},\vec{y})$ | $\to$ | $h(\vec{x},x_k,\vec{y})$ | | **copyt**$(\lvert\vec{x}\rvert-k+1)\cdot$**goto**$(h)$ |
| | $f(\vec{x},\vec{y})$ | $\to$ | $h(\vec{x},y_k,\vec{y})$ | | **copya**$(k)\cdot$**goto**$(h)$ |
| M4 | $f(\vec{x},\vec{y},\vec{z})$ | $\to$ | $h(\vec{x},\vec{z})$ | $\lvert\vec{y}\rvert\neq 0$ | **adrop**$(\lvert\vec{y}\rvert)\cdot$**goto**$(h)$ |
| | $f(\vec{x})$ | $\to$ | $h(\vec{x})$ | $L(f)\leq L(h)$ | **skip**$(L(h)-L(f))\cdot$**goto**$(h)$ |
| | $f(\vec{x})$ | $\to$ | $h(\vec{x})$ | $L(f)>L(h)$ | **retract**$(L(f)-L(h))\cdot$**goto**$(h)$ |
| M5 | $f(\vec{x},y)$ | $\to$ | $y$ | | **tdrop**$(\lvert\vec{x}\rvert)\cdot$**recycle** |

This transformation of minimal rules into sequences of ARM instructions makes clear why loci of function symbols, and the auxiliary traversal stack, enhance the efficiency of our compilation technique. For minimal rules of type M1,2,3,5, and

M4 with $|\vec{y}| \neq 0$, we have $L(f) = L(h) = |\vec{x}|$, because $(\mathcal{M}, L)$ is stratified (see Definition 3.3). So if the executable stack of function symbol $f$ or $h$ is executed, then the first $|\vec{x}|$ arguments of $f$ or $h$ are on top of the traversal stack, while its remaining arguments are on top of the argument stack. This information is used in the transformation of minimal rules to ARM instructions as follows.

- For minimal rules of type M1, instruction $\mathbf{match}(g, h)$ tests whether the $(|\vec{x}|+1)$th argument of the left-hand side, which is on the top of the argument stack, has outermost function symbol $g$. If so, then this term is replaced by its arguments, and the execution proceeds with respect to function symbol $h$.

- For minimal rules of type M2, instruction $\mathbf{push}(h)$ pushes function symbol $h$ onto the control stack, after which instruction $\mathbf{goto}(g)$ can proceed (innermost) rewriting with respect to function symbol $g$ immediately, because the $|\vec{y}|$ arguments of $g$ are on top of the argument stack, which agrees with the fact that the locus of $g$ is 0 (by Definition 3.2.2).

- For minimal rules of type M3, an instruction **copyt** or **copya** copies the $k$th or $(|\vec{x}| + k)$th argument of the left-hand side, which is on the traversal stack or the argument stack, respectively, on the top of the argument stack. Then instruction $\mathbf{goto}(h)$ proceeds the execution with respect to function symbol $h$.

- For minimal rules of type M4 with $|\vec{y}| \neq 0$, instruction $\mathbf{adrop}(|\vec{y}|)$ deletes the $|\vec{y}|$ arguments of the left-hand side that are on top of the argument stack, after which $\mathbf{goto}(h)$ proceeds the execution with respect to $h$.

- For minimal rules of type M4 with $|\vec{y}| = 0$, the loci of the function symbols $f$ and $h$ at the head of the left- and right-hand side may differ. If $L(f) \leq L(h)$, then $\mathbf{skip}(L(h) - L(f))$ transfers the top $L(h) - L(f)$ terms from the argument to the traversal stack, and otherwise, $\mathbf{retract}(L(f) - L(h))$ transfers the top $L(f) - L(h)$ terms from the traversal to the argument stack. Next, $\mathbf{goto}(h)$ proceeds the execution with respect to $h$.

- For minimal rules of type M5, instruction $\mathbf{tdrop}(|\vec{x}|)$ removes the first $|\vec{x}|$ arguments of the left-hand side, which are on top of the traversal stack. Next, **recycle** proceeds the execution at the top of the control stack.

For each function symbol $f$, an executable stack is constructed as follows:

1. If there is a minimal rule for $f$ in $\mathcal{M}$, then there is exactly one such rule of type M2-5 for $f$ in $\mathcal{M}$ (due to simple completeness). In this case, the sequence of machine instructions that belongs to this most general rule is stored at the bottom of the executable stack for $f$. Next, the $\mathbf{match}(g, h)$ instructions that belong to the minimal rules of type M1 for $f$ in $\mathcal{M}$ (there is never more than one such rule for each $g$) are stored on top of the executable stack for $f$, in arbitrary order.

2. If there is no minimal rule for $f$ in $\mathcal{M}$, then its executable stack consists of the two instructions

$$\mathbf{build}(f, ar(f)) \cdot \mathbf{recycle}$$

The intuition behind this construction is as follows. The $\mathbf{match}(g, h)$ instructions on top of the executable stack for $f$ test whether minimal rules of the form $f(\vec{x}, g(\vec{y}), \vec{z}) \to h(\vec{x}, \vec{y}, \vec{z})$ can be applied. If these instructions all fail, then the instructions at the bottom of the executable stack for $f$ make sure that the most general rule for $f$ is applied. In the case that there are no minimal rules for $f$, the locus of $f$ is 0 (by Definition 3.2.1), so that its arguments $t_1, ..., t_{ar(f)}$ are on top of the argument stack. Moreover, in this case $f(t_1, ..., t_{ar(f)})$ is a normal form. Then the instruction $\mathbf{build}(f, ar(f))$ on the executable stack for $f$ replaces the terms $t_1, ..., t_{ar(f)}$ on top of the argument stack by $f(t_1, ..., t_{ar(f)})$, after which **recycle** proceeds the execution at the top of the control stack.

**Remark 4.4** For efficiency reasons, large sequences of **match** instructions should be implemented table driven rather than iterative. That is, the $\mathbf{match}(g, h)$ instructions on top of an executable stack should be implemented as a hash table (see e.g. [12, Chapter 12]), where $g$ is an address for the executable stack of $h$. A hash table is a data type that is suitable in situations where the collection of used addresses is sparse in comparison with the collection of possible addresses.

Finally, the stratified simply complete MTRS $(\mathcal{M}, L)$ is turned into a program table $\mathit{program}(\mathcal{M})$ of executable stacks as follows. For each function symbol $f$, its executable stack is paired with the address $f$, and these pairs are all gathered in a table. Redundant instructions of the form $\mathbf{adrop}(0)$ and $\mathbf{tdrop}(0)$ and $\mathbf{skip}(0)$ and $\mathbf{retract}(0)$ are eliminated, to gain efficiency. Moreover, if after this elimination the executable stack of a function symbol $f$ consists of the single instruction $\mathbf{goto}(h)$, then all occurrences of $f$ in **goto** instructions and at the right-hand side of **match** instructions in other executable stacks are replaced by $h$. The resulting table is denoted by $\mathit{program}(\mathcal{M})$.

**Remark 4.5** At the bottom of each executable stack there is always either an instruction **goto** or an instruction **recycle**. Moreover, at the bottom of the control stack there is always a *bottom* element. This observation ensures that it is unnecessary to check whether or not an executable or control stack is empty. Furthermore, in the case that a **copya**, **adrop**, **skip** or **build** instruction is executed, it is guaranteed that the argument stack always contains a sufficient number of elements, and in the case that a **copyt**, **tdrop** or **retract** instruction is executed, it is guaranteed that the traversal stack always contains a sufficient number of elements. These observations are important for efficiency reasons; repeated checks on the emptiness of stacks are expensive.

### *4.5 Semantics of ARM*

The intuitive meaning of ARM instructions, when popped from the executable stack, was presented in Section 4.3. This semantics is made precise in Table 0.2. The states of ARM are represented by 5-tuples $\langle P, C, E, A, T \rangle$, where $P$ is a program table, and $C$, $E$, $A$ and $T$ are a control, executable, argument and traversal stack, respectively. The state transition rules that are presented in Table 0.2 use an auxiliary function $\mathit{get}(f, P)$, which produces the executable stack with address $f$ in table $P$.

The reduction of a term $t$ by means of a stratified simply complete MTRS $\mathcal{M}$ is simulated by the expression

$$< \mathit{program}(\mathcal{M}), \mathit{control}(t), \mathbf{recycle}, \varepsilon_A, \varepsilon_T >$$

$$
\begin{array}{rcl}
< P, C, \mathbf{match}(f,g) \cdot E, f(t_1,...,t_k) \cdot A, T > & \rightarrow & < P, C, get(g,P), t_1 \cdot ... \cdot t_k \cdot A, T > \\
< P, C, \mathbf{match}(f,g) \cdot E, t \cdot A, T > & \rightarrow & < P, C, E, t \cdot A, T > \\
 & & \text{if } t \neq f(t_1,...,t_k) \\
< P, C, \mathbf{copya}(k) \cdot E, t_1 \cdot ... \cdot t_k \cdot A, T > & \rightarrow & < P, C, E, t_k \cdot t_1 \cdot ... \cdot t_k \cdot A, T > \\
< P, C, \mathbf{copyt}(k) \cdot E, A, t_1 \cdot ... \cdot t_k \cdot T > & \rightarrow & < P, C, E, t_k \cdot A, t_1 \cdot ... \cdot t_k \cdot T > \\
< P, C, \mathbf{push}(f) \cdot E, A, T > & \rightarrow & < P, f \cdot C, E, A, T > \\
< P, C, \mathbf{adrop}(k) \cdot E, t_1 \cdot ... \cdot t_k \cdot A, T > & \rightarrow & < P, C, E, A, T > \\
< P, C, \mathbf{tdrop}(k) \cdot E, A, t_1 \cdot ... \cdot t_k \cdot T > & \rightarrow & < P, C, E, A, T > \\
< P, C, \mathbf{skip}(k) \cdot E, t_1 \cdot ... \cdot t_k \cdot A, T > & \rightarrow & < P, C, E, A, t_k \cdot ... \cdot t_1 \cdot T > \\
< P, C, \mathbf{retract}(k) \cdot E, A, t_1 \cdot ... \cdot t_k \cdot T > & \rightarrow & < P, C, E, t_k \cdot ... \cdot t_1 \cdot A, T > \\
< P, C, \mathbf{build}(f, ar(f)) \cdot E, t_1 \cdot ... \cdot t_{ar(f)} \cdot A, T > & \rightarrow & < P, C, E, f(t_1,...,t_{ar(f)}) \cdot A, T > \\
< P, C, \mathbf{goto}(f), A, T > & \rightarrow & < P, C, get(f,P), A, T > \\
< P, f \cdot C, \mathbf{recycle}, A, T > & \rightarrow & < P, C, get(f,P), A, T > \\
< P, x \cdot C, \mathbf{recycle}, A, T > & \rightarrow & < P, C, \mathbf{recycle}, x \cdot A, T > \\
< P, bottom, \mathbf{recycle}, t, \varepsilon_T > & \rightarrow & t
\end{array}
$$

Table 0.2: Semantics of ARM

where the **recycle** instruction starts up the reduction process by popping the top element from the control stack. When the state transition rules for ARM in Table 0.2 reduce this expression to a term $s$, then this is the normal form of $t$ with respect to $\mathcal{M}$, using rightmost innermost rewriting with specificity ordering.

Finally, we give an overview of the complete transformation, from left-linear TRS to ARM code. Suppose that we want to reduce a term $t$ with respect to a left-linear TRS $\mathcal{R}$. First we transform $\mathcal{R}$ into a stratified simply complete MTRS $\mathcal{M}$, which is then transformed into a table $\mathit{program}(\mathcal{M})$ of ARM instructions. Furthermore, we construct the control stack of $t$, that is obtained by collecting the function symbols and variables of $t$ in a rightmost innermost fashion. The executable stack consists of the instruction **recycle**, while the argument and the traversal stack are both empty. The resulting ARM expression, which combines these elements, is executed according to the semantics of ARM in Table 0.2. Execution proceeds until the execution stack consists of a **recycle** instruction, and the control stack contains only the *bottom* element. In that case the traversal stack will be empty, and the argument stack will contain exactly one term, being the normal form of $t$ with respect to $\mathcal{M}$, where function symbols in this normal form may carry auxiliary superscripts $c$, which were introduced in "Add Most General Rules". A final execution step produces this normal form.

*4.6 Example*

In Section 3.7, the standard specification for addition on natural numbers was transformed into the following stratified simply complete MTRS:

$$\begin{aligned}
\mathit{plus}(\mathit{zero}, y) &\to \mathit{plus}_{\mathit{zero}}(y) \\
\mathit{plus}_{\mathit{zero}}(y) &\to y \\
\mathit{plus}(\mathit{succ}(x), y) &\to \mathit{plus}_{\mathit{succ}}(x, y) \\
\mathit{plus}_{\mathit{succ}}(x, y) &\to \mathit{succ}(\mathit{plus}(x, y)) \\
\mathit{plus}(x, y) &\to \mathit{plus}^{c}(x, y)
\end{aligned}$$

whereby all function symbols involved have locus 0. The transformation described in Section 4.4 turns this MTRS into the following ARM program:

$$\begin{aligned}
\mathit{zero} &: \mathbf{build}(\mathit{zero}, 0) \cdot \mathbf{recycle} \\
\mathit{succ} &: \mathbf{build}(\mathit{succ}, 1) \cdot \mathbf{recycle} \\
\mathit{plus} &: \mathbf{match}(\mathit{zero}, \mathit{plus}_{\mathit{zero}}) \cdot \mathbf{match}(\mathit{succ}, \mathit{plus}_{\mathit{succ}}) \cdot \mathbf{goto}(\mathit{plus}^{c}) \\
\mathit{plus}_{\mathit{zero}} &: \mathbf{recycle} \\
\mathit{plus}_{\mathit{succ}} &: \mathbf{push}(\mathit{succ}) \cdot \mathbf{goto}(\mathit{plus}) \\
\mathit{plus}^{c} &: \mathbf{build}(\mathit{plus}^{c}, 2) \cdot \mathbf{recycle}
\end{aligned}$$

In the sequences for $\mathit{plus}$ and $\mathit{plus}_{\mathit{zero}}$, the redundant instructions $\mathbf{skip}(0)$ and $\mathbf{tdrop}(0)$ have been omitted, respectively.

As an example, we show how this program derives $1+0=1$, or, in other words, how it reduces $\mathit{plus}(\mathit{succ}(\mathit{zero}), \mathit{zero})$ to its normal form $\mathit{succ}(\mathit{zero})$, by means of the state transition rules for ARM in Table 0.2. Let $P$ denote the program table above, and note that the control stack of $\mathit{plus}(\mathit{succ}(\mathit{zero}), \mathit{zero})$ is $\mathit{zero} \cdot \mathit{zero} \cdot \mathit{succ} \cdot \mathit{plus} \cdot \mathit{bottom}$. The execution proceeds as follows, whereby in each of its fifteen steps adaptations of stacks have been underlined:

$$
\begin{aligned}
& \langle P, zero \cdot zero \cdot succ \cdot plus \cdot bottom, \mathbf{recycle}, \varepsilon_A, \varepsilon_T \rangle \\
\rightarrow\ & \langle P, zero \cdot succ \cdot plus \cdot bottom, \underline{\mathbf{build}(zero, 0) \cdot \mathbf{recycle}}, \varepsilon_A, \varepsilon_T \rangle \\
\rightarrow\ & \langle P, zero \cdot succ \cdot plus \cdot bottom, \mathbf{recycle}, \underline{zero}, \varepsilon_T \rangle \\
\rightarrow\ & \langle P, succ \cdot plus \cdot bottom, \underline{\mathbf{build}(zero, 0) \cdot \mathbf{recycle}}, zero, \varepsilon_T \rangle \\
\rightarrow\ & \langle P, succ \cdot plus \cdot bottom, \mathbf{recycle}, \underline{zero} \cdot zero, \varepsilon_T \rangle \\
\rightarrow\ & \langle P, plus \cdot bottom, \underline{\mathbf{build}(succ, 1) \cdot \mathbf{recycle}}, zero \cdot zero, \varepsilon_T \rangle \\
\rightarrow\ & \langle P, plus \cdot bottom, \mathbf{recycle}, \underline{succ(zero)} \cdot zero, \varepsilon_T \rangle \\
\rightarrow\ & \langle P, bottom, \underline{\mathbf{match}(zero, plus_{zero}) \cdot \mathbf{match}(succ, plus_{succ}) \cdot \mathbf{goto}(plus^c)}, succ(zero) \cdot zero, \varepsilon_T \rangle \\
\rightarrow\ & \langle P, bottom, \mathbf{match}(succ, plus_{succ}) \cdot \mathbf{goto}(plus^c), succ(zero) \cdot zero, \varepsilon_T \rangle \\
\rightarrow\ & \langle P, bottom, \underline{\mathbf{push}(succ) \cdot \mathbf{goto}(plus)}, zero \cdot zero, \varepsilon_T \rangle \\
\rightarrow\ & \langle P, \underline{succ} \cdot bottom, \mathbf{goto}(plus), zero \cdot zero, \varepsilon_T \rangle \\
\rightarrow\ & \langle P, succ \cdot bottom, \underline{\mathbf{match}(zero, plus_{zero}) \cdot \mathbf{match}(succ, plus_{succ}) \cdot \mathbf{goto}(plus^c)}, zero \cdot zero, \varepsilon_T \rangle \\
\rightarrow\ & \langle P, succ \cdot bottom, \underline{\mathbf{recycle}}, zero, \varepsilon_T \rangle \\
\rightarrow\ & \langle P, bottom, \underline{\mathbf{build}(succ, 1) \cdot \mathbf{recycle}}, zero, \varepsilon_T \rangle \\
\rightarrow\ & \langle P, bottom, \mathbf{recycle}, \underline{succ(zero)}, \varepsilon_T \rangle \\
\rightarrow\ & succ(zero)
\end{aligned}
$$

If the two **match** instructions for *plus* are implemented as a hash table (see Remark 4.4), then this reduction takes one step less, because in that case step eight becomes redundant.

In this example, the traversal stack is never used, simply because the function symbols in the MTRS all have locus 0. This would change if in Section 3.7 we had started from the following specification for addition:

$$
\begin{aligned}
plus(x, zero) &\rightarrow x \\
plus(x, succ(y)) &\rightarrow succ(plus(x, y))
\end{aligned}
$$

because in that case the resulting MTRS would incorporate function symbols with locus 1. Then the reduction of the term $plus(zero, succ(zero))$ to its normal form $succ(zero)$ by means of the resulting ARM program would take eight extra steps, due to the swapping of terms between argument and traversal stack. This distinction is caused by our choice of specificity ordering, which enforces that arguments are considered from left to right.

### *4.7 Heap*

A heap is an abstract data type suitable for storing graphs representing terms (and for recycling memory that is no longer referenced). A graph is stored as a collection of structures with addresses (called 'pointers' in implementor's jargon), and all system components other than the heap represent a term by a single pointer into the heap. Heaps are implemented such that given a pointer, the related term can be found in $O(1)$ time. The actual implementation of ARM uses a heap, see [25, Chapter 3], to speed up copying of arguments, and swapping of terms between argument and traversal stack. In this paper we did not go into the role of the heap, because it is not vital for the ideas behind the ARM methodology, and it is somewhat obscuring when trying to get these ideas across to the reader.

For efficiency reasons it is important to access the heap as little as possible: "faster implementations use less heap" [20, page 649]. We note that of the ARM instructions, only **build** requires write access to heap storage, and only **match** requires read access to such storage, while the other instructions only access stack storage.

In order to avoid waste of memory space, terms in the heap with no pointers to

them are to be reclaimed by means of a so-called garbage collector. See e.g. [11] and [40, Chapter 17] for overviews of garbage collection techniques.

**Remark 4.6** In principle, the problem of pattern matching with respect to non-linear left-hand sides, which was discussed in the introduction, can be tackled by using so-called hash-consing [43] in the heap. Basically, hash-consing means that a term in the heap that consists of a function symbol $f$ and addresses $a_1, ..., a_{ar(f)}$, gets as pointer $\langle f, a_1, ..., a_{ar(f)} \rangle$. With this addressing technique, checks on syntactic equality can be performed in $O(1)$. However, serious drawbacks of hash-consing are that the construction of addresses is expensive, and that it combines badly with garbage collection.

## 5. Related Work

### *5.1 Innovations*

In this section we discuss some advantages of the proposed compilation technique.

Unlike most functional languages, we do not need to distinguish 'defined' function symbols (which occur at the head of the left-hand side of some rewrite rule) from 'constructors' (which occur in some normal form). Namely, owing to the transformation into a simply complete TRS in Section 3.3, this distinction is obtained automatically.

The technique of pattern matching using tree automata stems from [22]. The idea to express pattern matching of TRSs in the language of TRSs itself was inspired by Pettersson [39]. The notion of an MTRS, and the procedures in Sections 3.4 and 3.5, to transform a TRS into an MTRS, however, are entirely new.

Since we use an innermost rewriting strategy, pattern matching with respect to a term only involves the syntactic structure of subterms that are in normal form. We exploited this phenomenon, namely, only reducts in normal form are built on the heap, by the **build** instruction, while outermost function symbols of other reducts are pushed onto the control stack for future reference, by the **push** instruction, see Section 4.4. As was mentioned in Section 4.7, economical use of the heap is important for efficiency reasons. In contrast, pattern matching with respect to outermost rewriting uses the full syntactic structure of a term, so the outermost strategy would require that all reducts are built on the heap.

Specificity ordering is also important for the efficiency of pattern matching. Firstly, it enables to share several matchings in the minimal rule (3.2) in Section 3.4. Secondly, specificity ordering causes that **match** instructions are executed in sequence, which makes it worth wile to combine such sequences in hash tables, see Remark 4.4.

Several abstract machines from the literature contain some form of control stack, and most of them contain some form of argument stack. However, the traversal stack, and the notion of a locus, are new. These two related concepts are essential for the efficiency of pattern matching on the level of ARM. Namely, consider a minimal rule of type M1:

$$f(\vec{x}, g(\vec{y}), \vec{z}) \ \rightarrow \ h(\vec{x}, \vec{y}, \vec{z})$$

This rule is expressed in ARM as a $\mathbf{match}(g, h)$ instruction (see Section 4.4), which splices the arguments $\vec{y}$ in between the arguments $\vec{x}$ and $\vec{z}$. If all arguments had been stored on the argument stack, this splice operation would have taken $O(|\vec{x}, \vec{y}|)$. However, using the locus function we are able to ensure that the first $|\vec{x}|$ arguments are on top of the traversal stack, while the remaining $|\vec{z}| + 1$ arguments are on

top of the argument stack. Therefore, in practice this splice operation takes only $O(|\vec{y}|)$. While the function symbol $g$ will always stem from the original signature, in most cases the function symbol $f$ will have been introduced during the minimization process. This means that in general $|\vec{x}|$ will be considerably larger than $|\vec{y}|$. Hence, the traversal stack and the locus have a positive impact on the time complexity of the **match** instructions.

### *5.2 Abstract Machines*

An early abstract machine for the implementation of a functional language is Landin's SECD machine [33], which he utilizes for the eager (i.e., innermost) evaluation of higher-order function application. Two implementations that are related to Landin's approach are by means of the functional abstract machine [10] and the categorical abstract machine [13], respectively. In contrast, the abstract rewriting machine in this paper involves the eager evaluation of first-order terms.

Several abstract machines have been used for lazy evaluation of higher-order function application, notably: the S-K reduction machine [45], the G-machine [24], the three instruction machine [51], and the spineless tagless G-machine [41]. Basically, a lazy rewriting strategy postpones (innermost) rewriting of certain so-called non-strict arguments, in order to improve termination properties. Although ARM was designed purely for innermost rewriting, laziness can be incorporated by means of a source-to-source transformation, given one extra ARM instruction to capture graph rewriting, see [28] and [25, Chapter 6]. A similar observation was made for the categorical abstract machine [13, Section 4].

Fradet and Le Métayer [17] present a compilation technique of higher-order function application into an abstract machine that leaves the reduction graphs intact. Namely, the states of the machine are the reducts themselves. They state that "performance considerations are not the main topic", and show that their approach leads to simple correctness proofs. Hamel and Goguen [19] give a formal correctness proof for their eager implementation of a higher-order algebraic specification language, using the tiny rewrite instruction machine. Their approach is more geared towards provability than towards efficiency, because environments are built explicitly on the heap, instead of on the cheaper stack. Klaeren and Indermark [30] give a formal correctness proof for their eager implementation of an algebraic specification language with recursive functions, using the abstract stack machine. Further correctness proofs for abstract machines are presented in [35, 13].

### *5.3 Functional Languages*

Backus [4] propagated the use of functional programming languages, and since then, several of such languages have been implemented. The eager first-order equational programming language Epic [48, 49] has been implemented by means of the ARM technology. Other first-order languages with an eager implementation are ASF+SDF [15] and Sisal [9]. Both languages are not compiled via an abstract machine.

There is a long tradition of eager higher-order functional languages, which dates back to the implementation of McCarthy's Lisp [36], and Landin's proposal ISWIM [34]. Lisp was succeeded by Scheme [42], and ISWIM was an inspiration for ML [18], which in turn was succeeded by Standard ML [38]. Other eager higher-order languages include Caml [50], which was implemented using the categorical abstract

machine, and Trafola [2], which was implemented by means of an abstract machine called Trama. More recently, several lazy higher-order functional languages have been implemented, notably: Miranda [46] using the S-K reduction machine, Lazy ML [3] by means of the G-machine, and Haskell [23].

Hartel, Feeley et al. [20] compare the efficiency of several functional programming languages, including a prototype of Epic. The comparison of interpreted and non-interpreted languages, which are compiled into an abstract or a concrete machine, respectively, leads to the conclusion that "interpretive systems yield the worst performance" [20, page 649]. This is not surprising, because interpreted systems have to perform one more compilation step to reach the level of the concrete machine. Furthermore, it is easier to import 'smart' programming tricks and extra features for non-interpreted languages. However, compilation into a concrete machine does lead to programs that are more difficult to maintain and document, which hampers their development in the long run.

As for the comparison of lazy and eager evaluation, it is concluded that "non-strict compilers do not achieve [...] the performance of eager implementations", and that "interpreters for strict languages (Caml Light, Epic) do seem on the whole to be faster than interpreters for non-strict languages (NHC, Gofer, RUFLI, Miranda)" [20, page 649]. With respect to higher-order languages, it is remarked that "higher-order functions are generally expensive to implement" [20, page 636]. In spite of these observations, the achievements of for instance the lazy higher-order language Haskell, implemented on a concrete machine, are impressive. There seems to be room for a well-structured implementation of an eager first-order language, which, owing to the diminished overhead, should be able to perform even better than current lazy and/or higher-order languages.

REFERENCES

1. A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers. Principles, Techniques and Tools*. Addison-Wesley, 1986.

2. M. Alt, C. Fecht, C. Ferdinand, and R. Wilhelm. TrafoLa-H subsystem. In B. Hoffmann and B. Krieg-Brückner, eds., *Program Development by Specification and Transformation: the PROSPECTRA Methodology, Language Family, and System*, LNCS 680, pp. 539–576. Springer, 1993.

3. L. Augustsson and T. Johnsson. The Chalmers Lazy-ML compiler. *The Computer Journal*, 32(2):127–141, 1989.

4. J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, 1978.

5. J.C.M. Baeten, J.A. Bergstra, J.W. Klop, and W.P. Weijland. Term-rewriting systems with rule priorities. *Theoretical Computer Science*, 67(2/3):283–301, 1989.

6. J.A Bergstra and M.G.J. van den Brand. A proposal for the $\mu$ASF family of specification languages: subequational programming with innermost reduction. In preparation.

7. J.A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification*. ACM

Frontier Series, ACM Press in cooperation with Addison-Wesley, 1989.

8. J.A. Bergstra and J.W. Klop. Conditional rewrite rules: confluence and termination. *Journal of Computer and System Sciences*, 32(3):323–362, 1986.
9. D.C. Cann. The optimizing SISAL compiler: version 12.0. Manual UCRL-MA-110080, Lawrence Livermore National Laboratory, 1992. Available via ftp as sisal.llnl.gov/pub/sisal/OSC-manual.ps.
10. L. Cardelli. Compiling a functional language. In *Proceedings ACM Symposium on Lisp and Functional Programming*, pp. 208–226. ACM, 1984.
11. J. Cohen. Garbage collection of linked data structures. *ACM Computing Surveys*, 13(3):341–367, 1981.
12. T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.
13. G. Cousineau, P.-L. Curien, and M. Mauny. The categorical abstract machine. *Science of Computer Programming*, 8(2):173–202, 1987.
14. N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, ed., *Handbook of Theoretical Computer Science, Volume B*, pp. 243–320. Elsevier, 1990.
15. A. van Deursen, J. Heering, and P. Klint, eds. *Language Prototyping: An Algebraic Specification Approach*. AMAST Series in Computing 5, World Scientific, 1996.
16. W.J. Fokkink and J.C. van de Pol. Simulation as a correct transformation of rewrite systems. In I. Prívara and P. Ružička, eds., *Proceedings 22nd Symposium on Mathematical Foundations of Computer Science (MFCS'97)*, Bratislava, LNCS 1295, pp. 249–258, 1997.
17. P. Fradet and D. Le Métayer. Compilation of functional languages by program transformation. *ACM Transactions on Programming Languages and Systems*, 13(1):21–51, 1991.
18. M.J.C. Gordon, R. Milner, L. Morris, M.C. Newey, and C.P. Wadworth. A metalanguage for interactive proof in LCF. In *Proceedings 5th Symposium on Principles of Programming Languages (POPL'78)*, Tucson, pp. 119–130, 1978.
19. L.H. Hamel and J.A. Goguen. Towards a provably correct compiler for OBJ3. In M.V. Hermenegildo and J. Penjam, eds., *Proceedings 6th Symposium on Programming Language Implementation and Logic Programming (PLILP'94)*, Madrid, LNCS 844, pp. 132–146. Springer, 1994.
20. P.H. Hartel, M. Feeley, M. Alt, L. Augustsson, P. Baumann, M. Beemster, E. Chailloux, C.H. Flood, W. Grieskamp, J.H.G. van Groningen, K. Hammond, B. Hausman, M.Y. Ivory, R.E. Jones, J.F.Th. Kamperman, P. Lee, X. Leroy, R.D. Lins, S. Loosemore, N. Rjemo, M. Serrano, J.-P. Talpin, J. Thackray, S. Thomas, H.R. Walters, P. Weis, and P. Wentworth. Benchmarking implementations of functional languages with "pseudoknot", a float-intensive benchmark. *Journal of Functional Programming*, 6(4):621–655, 1996.
21. J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. The syntax definition

formalism SDF – reference manual. *ACM SIGPLAN Notices*, 24(11):43–75, 1989.

22. C.M. Hoffmann and M.J. O'Donnell. Pattern matching in trees. *Journal of the ACM*, 29(1):68–95, 1982.
23. P. Hudak, S.L. Peyton Jones, and P.L. Wadler, editors. Report on the programming language Haskell – a non-strict purely functional language, version 1.2. *ACM SIGPLAN Notices*, 27(5):R1–R164, 1992.
24. T. Johnsson. Efficient compilation of lazy evaluation. In *Proceedings ACM Symposium on Compiler Construction*, Montreal, *ACM SIGPLAN Notices*, 19(6):58–69, 1984.
25. J.F.Th. Kamperman. *Compilation of Term Rewriting Systems.* PhD thesis, University of Amsterdam, 1996. Available at http://www.cwi.nl/~jasper.
26. J.F.Th. Kamperman and H.R. Walters. ARM – abstract rewriting machinery. In *Proceedings Computer Science in the Netherlands (CSN'93)*, pp. 193–204. Stichting Mathematisch Centrum, 1993.
27. J.F.Th. Kamperman and H.R. Walters. Minimal term rewriting systems. In M. Haveraaen, O. Owe, and O.-J. Dahl, eds., *Proceedings 11th Workshop on Specification of Abstract Data Types*, Oslo, LNCS 1130, pp. 274–290. Springer, 1995.
28. J.F.Th. Kamperman and H.R. Walters. Lazy rewriting on eager machinery. In J. Hsiang, ed., *6th Conference on Rewriting Techniques and Applications (RTA'95)*, Kaiserslautern, LNCS 914, pp. 147–162. Springer, 1995.
29. J.F.Th. Kamperman and H.R. Walters. Simulating TRSs by minimal TRSs: a simple, efficient, and correct compilation technique. Technical Report CS-R9605, CWI, 1996. Available at http://www.cwi.nl/epic.
30. H. Klaeren and K. Indermark. Efficient implementation of an algebraic specification language. In M. Wirsing and J.A. Bergstra, eds., *Proceedings Workshop on Algebraic Methods: Theory, Tools and Applications*, Passau, LNCS 394, pp. 69–90. Springer, 1987.
31. P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering Methodology*, 2(2):176–201, 1993.
32. J.W. Klop. Term rewriting systems. In S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, eds., *Handbook of Logic in Computer Science, Volume I, Background: Computational Structures*, pp. 1–116. Oxford University Press, 1992.
33. P.J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.
34. P.J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, 1966.
35. D. Lester. The G-machine as a representation of stack semantics. In G. Kahn, ed., *Proceedings 3rd Conference on Functional Programming Languages and Computer Architecture*, Portland, LNCS 274, pp. 46–59. Springer, 1987.
36. J. McCarthy. Recursive functions of symbolic expressions and their computation

by machine: part I. *Communications of the ACM*, 3(4):184–195, 1960.

37. J. McCarthy. Towards a mathematical science of computation. In *Proceedings Information Processing '62*, pp. 21–28. North-Holland, 1963.
38. R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.
39. M. Pettersson. A term pattern-match compiler inspired by finite automata theory. In U. Kastens and P. Pfahler, eds., *Proceedings 4th Workshop on Compiler Construction (CC'92)*, Paderborn, LNCS 641, pp. 258–270. Springer, 1992.
40. S.L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.
41. S.L. Peyton Jones and J. Salkild. The spineless tagless G-machine. In D.B. MacQueen, ed., *Proceedings 4th Conference on Functional Programming Languages and Computer Architecture*, pp. 184–201. Addison-Wesley, 1989.
42. J.A. Rees and W. Clinger, editors. Revised$^3$ report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 21(12):37–79, 1986.
43. M. Sassa and E. Goto. A hashing method for fast set operations. *Information Processing Letters*, 5(2):31–34, 1976.
44. S.R. Thatte. On the correspondence between two classes of reduction systems. *Information Processing Letters*, 20(2):83–85, 1985.
45. D.A. Turner. A new implementation technique for applicative languages. *Software – Practice and Experience*, 9(1):31–49, 1979.
46. D.A. Turner. Miranda: a non-strict functional language with polymorphic types. In J.-P. Jouannaud, ed., *Proceedings 2nd Conference on Functional Programming Languages and Computer Architecture*, Nancy, LNCS 201, pp. 1–16. Springer, 1985.
47. H.R. Walters. *On Equal Terms, Implementing Algebraic Specifications.* PhD thesis, University of Amsterdam, 1991. Available at http://www.cwi.nl/epic.
48. H.R. Walters and J.F.Th. Kamperman. EPIC: an equational language – abstract machine and supporting tools. In H. Ganzinger, ed., *Proceedings 7th Conference on Rewriting Techniques and Applications (RTA'96),* New Jersey, LNCS 1103, pp. 424–427. Springer, 1996.
49. H.R. Walters and J.F.Th. Kamperman. EPIC 1.0 (unconditional), an equational programming language. Technical Report CS-R9604, CWI, Amsterdam, 1996. Available at http://www.cwi.nl/epic.
50. P. Weis and X. Leroy. *Le Langage Caml.* InterÉditions, 1993.
51. S.C. Wray and J. Fairbairn. Non-strict languages – programming and implementation. *The Computer Journal*, 32(2):142–151, 1989.