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

WAN FOKKINK
University of Wales Swansea
JASPER KAMPERMAN
Cosmos Group BV
and
PUM WALTERS
Babelfish

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).

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*compilers*; *optimization*

General Terms: Languages
Additional Key Words and Phrases: Abstract machine, automata, specificity ordering, term rewriting

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. Aho et al. [1986]) it is customary to use an abstract machine as an abstraction of the concrete machine. This allows to hide details of the concrete machine in a small part of the compiler, and thus permits an easy reimplementation on other concrete machines. 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.

Authors' addresses: W. J. Fokkink, University of Wales Swansea, Department of Computer Science, Singleton Park, Swansea SA2 8PP, Wales; email: W.J.Fokkink@swan.ac.uk; J. F. Th. Kamperman, Cosmos Group BV, P.O. Box 3108, 2130 KC Hoofddorp, The Netherlands; email: jasper@research.techforce.nl; H. R. Walters, Babelfish, Korenbloemweg 23, 2403 GA Alphen a/d Rijn, The Netherlands; email: pum@babelfish.nl.

This article 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 on 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 [1982]. 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. An earlier version of ARM was introduced in Kamperman and Walters [1993], while the transformation from left-linear TRS to ARM code was first described in Kamperman and Walters [1996]. An overview is given in Kamperman [1996, Chapter 3].

The transformation of a left-linear TRS into an MTRS manipulates function symbols at the head of left-hand sides of original rewrite rules. In order to make sure that these manipulations do not affect the resulting normal forms, first we add so-called most general rules to the left-linear TRS. In the resulting left-linear TRS, which is called simply complete, function symbols that occur at the head of a left-hand side do not occur in normal forms. This transformation is an adaptation of one introduced by Thatte [1985].

For the sake of compilation, it is imperative to fix a deterministic rewriting strategy. We apply innermost rewriting, which reduces a redex that is as close as possible to the leaves of the parse tree of a term; if there are several such redexes, then we reduce the rightmost one. 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 be applied. We 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 a simply complete MTRS, with respect to our preferred rewriting strategy, is performed by the resulting ARM program as follows. First, the function symbols 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, whereby popping a function symbol f triggers the execution of the sequence of ARM instructions that belongs to rewrite rules of the form $f(t_1, \ldots, t_k) \to r$ in the MTRS, according to the deterministic rewriting strategy.

In order to compile simply complete MTRSs into efficient ARM code, we intro-

duce 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 sides of the rule indicate which argument on the left- or right-hand side contains a function symbol. If this argument is on the left-hand side, then the locus allows fast pattern matching with respect to this argument. If this argument is on 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 is 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. In innermost rewriting, each TRS can be transformed into a left-linear TRS by means of an auxiliary equality function $eq(s,t)$, which evaluates to `true` if and only if s and t are syntactically equal. Namely, 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. For example, a rewrite rule $f(x,x) \to r$ can be simulated by the following left-linear TRS:

$$\begin{aligned} f(x,y) &\to g(eq(x,y),x,y) \\ g(\texttt{true},x,y) &\to r \\ g(\texttt{false},x,y) &\to f'(x,y) \end{aligned}$$

whereby intuitively the fresh function symbol f' is a copy of f. See Kamperman [1996, p. 28] for a more elaborate example.

We do not consider conditional TRSs [Bergstra and Klop 1986], 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. Therefore, the problems involved with the implementation of conditions in rewrite rules are orthogonal to the matters that are investigated in this article. In innermost rewriting, 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. For example, a rewrite rule $x \downarrow y \Rightarrow f(x,y) \to r$ can be simulated by the left-linear TRS above.

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 [Klint 1993; van Deursen et al. 1996], which is a combination of the algebraic specification formalism [Bergstra et al. 1989] and the syntax definition formalism [Heering et al. 1989]. Van den Brand [1997] remarks that a compiler for ASF, together with an underlying formal semantics, are still missing. Our main goals are a well-structured, clearly described, and efficient implementation. The detailed description of the implementation is given in the current article, where the

reader may convince herself or 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 Hartel et al. [1996, Table 9], favorable execution times were reported concerning the equational programming language Epic [Walters and Kamperman 1996a; 1996b; Walters 1997], which has been implemented by means of the ARM methodology. Based on these experiences, and by further insights reported in this article, it can be stated that the compilation of a left-linear TRS via a stratified simply complete MTRS into an ARM program leads to an efficient implementation.

Questions on the correctness of compilation of programming languages date back to McCarthy [1963]. The question of whether our compilation strategy for left-linear TRSs is correct can be answered in several ways. First, the intuitive explanations that are given for the transformations, from left-linear TRS to stratified simply complete MTRS and then to ARM code, help to understand why the compilation is correct. Second, the technology has been tested thoroughly, in the sense that it has been implemented, and works satisfactorily, in Epic. The third and final answer is a formal correctness proof, which is presented in the electronic appendix. The proof is based on the notion of simulation [Kamperman and Walters 1996], which relates the reduction graphs of the transformed TRS to the reduction 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, in the sense that no information on normal forms in the original rewrite system is lost; see Fokkink and van de Pol [1997]. That is, there exist mappings *parse* from original to transformed terms and *print* from transformed to original terms such that for each original term t its normal forms can be computed as follows: compute the normal forms of $parse(t)$, and apply the *print* function to them. This correctness notion is based on established ideas on compiler correctness [Burstall and Landin 1969; Morris 1973].

This article 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 simply complete MTRS. In Section 4 the syntax and semantics of ARM are given, and it is shown how to obtain an ARM program. Section 5 discusses related work. Finally, the electronic appendix contains the formal correctness proof.

2. PRELIMINARIES

This section introduces some preliminaries from term rewriting; for more background see, for example, Dershowitz and Jouannaud [1990] and Klop [1992].

2.1 Term-Rewriting Systems

Definition 2.1.1. A *signature* Σ consists of

—a countably infinite set $\mathcal{V}$ of variables $u, v, w, x, y, z, \ldots$;

—a non-empty set $\mathcal{F}$ of function symbols $f, g, h, \ldots$, disjoint from $\mathcal{V}$, 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 α-conversion (i.e., modulo renaming of variables).

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

Definition 2.1.2. $\mathbb{T}(\Sigma)$ denotes the set of *terms* $\ell, p, q, r, s, t, \ldots$ over Σ, being the smallest set satisfying

—$\mathcal{V} \subset \mathbb{T}(\Sigma)$;
—if $f \in \mathcal{F}$ and $t_1, \ldots, t_{ar(f)} \in \mathbb{T}(\Sigma)$, then $f(t_1, \ldots, t_{ar(f)}) \in \mathbb{T}(\Sigma)$.

A (possibly empty) sequence $t_1, \ldots, t_k$ of terms is often abbreviated to $\vec{t}$, and $|\vec{t}|$ denotes the length k of this sequence.

Definition 2.1.3. 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 2.1.4. A *rewrite rule* is an expression $\ell \to r$ with $\ell, r \in \mathbb{T}(\Sigma)$, where

(1) the left-hand side ℓ is not a single variable;
(2) variables that occur in the right-hand side r also occur in the left-hand side ℓ.

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

Definition 2.1.5. A rewrite rule $\ell \to r$ in a TRS $\mathcal{R}$ *matches* a term t if $\sigma(\ell) = t$ for some substitution σ. In this case t is called a *redex* for $\mathcal{R}$.

Definition 2.1.6. A rewrite rule $\ell \to r$ is *left-linear* if each variable occurs no more than once in its left-hand side ℓ.
A TRS is *left-linear* if all its rules are.

Definition 2.1.7. A rewrite rule $f(x_1, \ldots, x_{ar(f)}) \to r$ with $x_1, \ldots, x_{ar(f)}$ distinct variables is *most general*.
A TRS $\mathcal{R}$ is *simply complete* if for each $f \in \mathcal{F}$ 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 define, for each left-linear TRS $\mathcal{R}$ over a signature Σ, a binary rewrite relation $\to_\mathcal{R}$ on $\mathbb{T}(\Sigma)$.

Definition 2.1.8. $t \in \mathbb{T}(\Sigma)$ is a *normal form* for a rewrite relation $\to_\mathcal{R}$ if there does not exist a $t' \in \mathbb{T}(\Sigma)$ with $t \to_\mathcal{R} t'$.
t is said to be a *normal form of* s if $s \to_\mathcal{R}^* t$ and t is a normal form for $\to_\mathcal{R}$.

2.2 Rewriting Strategy

A deterministic rewriting strategy is determined by two priorities:

(1) if a term has several redexes, then it selects which of these redexes is actually reduced;
(2) if several rewrite rules match the same redex, then it selects which rewrite rule is preferred to reduce this redex.

For the first preference we adopt innermost rewriting, which selects a subterm as close as possible to the leaves of the parse tree of the term. Furthermore, we adopt rightmost rewriting; that is, if innermost rewriting can choose between several subterms, then it selects the rightmost of these subterms.

For the second 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 Baeten et al. [1989], where the semantics of such orderings on term-rewriting systems was studied thoroughly for the first time.

First, we present the precise definition of specificity ordering.

Definition 2.2.1. The syntactic specificity ordering $<$ on terms is defined by

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

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

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

If for two left-hand sides ℓ and ℓ' of left-linear rewrite rules there exist substitutions σ and σ' such that $\sigma(\ell) = \sigma'(\ell')$, then it is easy to see that $\ell < \ell'$, $\ell' < \ell$, or $\ell =_\alpha \ell'$. So in order to avoid situations in which two left-linear rewrite rules match the same term, and neither of these rules has priority over the other, it suffices to require that a TRS does not contain two rewrite rules of which the left-hand sides are equal modulo α-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 2.2.2. Given a TRS $\mathcal{R}$ over signature Σ, 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 $t_1, \ldots, t_{ar(f)}$. Then $\to_{\mathcal{R}}$ is defined for $f(t_1, \ldots, t_{ar(f)})$ as follows.
—(*Rightmost Innermost*) If, for some $i \in \{1, \ldots, ar(f)\}$, $t_{i+1}, \ldots, t_{ar(f)}$ are normal forms for $\to_{\mathcal{R}}$, and $t_i \to_{\mathcal{R}} s$, then

$$f(t_1, \ldots, t_{ar(f)}) \to_{\mathcal{R}} f(t_1, \ldots, t_{i-1}, s, t_{i+1}, \ldots, t_{ar(f)}).$$

—(*Specificity*) Suppose that $t_1, \ldots, t_{ar(f)}$ 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 $f(t_1, \ldots, t_{ar(f)}) = \sigma(\ell)$ for a certain substitution σ, then

$$f(t_1, \ldots, t_{ar(f)}) \to_{\mathcal{R}} \sigma(r).$$

If such a rewrite rule does not exist in $\mathcal{R}$, then $f(t_1, \ldots, t_{ar(f)})$ is a normal form for $\to_{\mathcal{R}}$.

$\rightarrow_{\mathcal{R}}$ is a subrelation of the standard rewrite relation for R without a rewriting strategy. Furthermore, normal forms for $\rightarrow_{\mathcal{R}}$ are also normal forms for this standard rewrite relation. Since the subscript $\mathcal{R}$ is usually clear from the context, in general it is omitted from $\rightarrow_{\mathcal{R}}$. The overloading of $\rightarrow$ 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 are supplied with "locus" values, which are important for the efficient compilation of an MTRS 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 on 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 3.1.1. A rewrite rule is called *minimal* if it is left-linear and has one of five forms:

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

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

3.2 Locus

For the sake of the efficiency of the compilation of MTRSs, we 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. The innermost rewriting strategy implies that function symbols that occur inside the left-hand side of a rewrite rule only apply to normal forms, so that they 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 3.2.1. A *locus* for a TRS $\mathcal{R}$ over $(\mathcal{V}, \mathcal{F}, ar)$ is a function $L : \mathcal{F} \rightarrow \mathbb{N}$ 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 $\vec{t}$ 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 on 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 on 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 on the left-hand side have to be deleted.
—For minimal rules of type M5, the locus should indicate how many arguments on the left-hand side have to be deleted.

These intuitions are incorporated in the following definition.

Definition 3.2.2. 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 3.1.1 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 depends on the incorporation of such rewrite rules.

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

3.3 Simple Completeness

In the transformations described in the next sections, we 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 (see Definition 2.1.7).

Assume a left-linear TRS $\mathcal{R}$ over $\Sigma = (\mathcal{V}, \mathcal{F}, ar)$. The following procedure 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 $f(\vec{s})$, but no most general rule with left-hand side $f(\vec{v})$. Add a fresh function symbol f^c to $\mathcal{F}$, of arity $ar(f)$, and add a most general rule

$$f(\vec{v}) \ \to \ f^c(\vec{v}).$$

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

The resulting TRS $\mathcal{S}$ is still left-linear, because the new rewrite rule is. Apply the procedure "Add Most General Rules" to $\mathcal{S}$. *(End of procedure)*

In the electronic appendix it is proved that the procedure "Add Most General Rules" terminates and constitutes a correct transformation; see Section A.2. Since the procedure "Add Most General Rules" terminates, 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. This locus trivially satisfies the requirements of Definition 3.2.1.

The introduction of the most general rule $f(\vec{v}) \to f^c(\vec{v})$ influences the normal forms of terms in $\mathbb{T}(\Sigma)$. If $t \in \mathbb{T}(\Sigma)$ and t' is the normal form of t for $\mathcal{R}$, then the normal form of t for $\mathcal{S}$ is obtained by replacing all occurrences of f in t' by f^c.

The intuition behind the transformation of $\mathcal{R}$ into $\mathcal{S}$ is as follows. The new rewrite rule $f(\vec{v}) \to f^c(\vec{v})$ is the desired most general rule with left-hand side $f(\vec{v})$. Innermost rewriting and specificity ordering together ensure that this rewrite rule only replaces occurrences of f in 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.

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 procedure transforms $\mathcal{R}$ in such a way that the left-hand sides of its nonminimal rules contain only one function symbol.

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

If each nonminimal rewrite rule in $\mathcal{R}$ contains only one function symbol in its left-hand side, then output $(\mathcal{R}, L, \mathcal{F}, ar)$. Else, select a function symbol f and an index i such that there exists a left-hand side $f(\vec{w}, g(\vec{p}), \vec{q})$ of a nonminimal rewrite rule in $\mathcal{R}$ with $|\vec{w}| = i$.

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

$$f_g(\vec{w}, \vec{p}, \vec{q}) \to r. \tag{1}$$

Furthermore, for each f_g add a left-linear rule

$$f(\vec{x}, g(\vec{y}), \vec{z}) \to f_g(\vec{x}, \vec{y}, \vec{z}). \tag{2}$$

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. For each such f_g add a most general rule

$$f_g(\vec{x}, \vec{y}, \vec{z}) \to f^d(\vec{x}, g(\vec{y}), \vec{z}) \tag{3}$$

and replace all left-hand sides of rewrite rules in $\mathcal{R}$ of the form $f(\vec{w}, \vec{s})$ with $|\vec{w}| > i$

by $f^d(\vec{w}, \vec{s})$, and add a most general rule

$$f(\vec{v}) \to f^d(\vec{v}). \tag{4}$$

The resulting TRS $\mathcal{S}$ is still simply complete, because among rules (1) and (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 $\mathcal{R}$ (which exists due to simple completeness) becomes a most general rule for f^d in $\mathcal{S}$, and rule (4) is a most general rule for f. Furthermore, $\mathcal{S}$ is still left-linear, because all the new rules are. 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 $\mathcal{S}$. Apply the procedure "Minimize Left-Hand Sides" to $\mathcal{S}$. *(End of procedure)*

In the electronic appendix it is proved that the procedure "Minimize Left-Hand Sides" terminates and constitutes a correct transformation; see Section A.3. Since the procedure "Minimize Left-Hand Sides" terminates, it produces a simply complete left-linear TRS with a locus, where the left-hand sides of its nonminimal rules contain only one function symbol. The adaptation of left-hand sides does not affect the resulting normal forms, due to simple completeness. Namely, there exist most general rewrite rules for the fresh function symbols f^d and f_g, which ensure that they do not occur in normal forms.

The intuition behind the transformation of $\mathcal{R}$ into $\mathcal{S}$ is as follows. The rules (2) constitute a first step toward checking whether a rule $f(\vec{w}, g(\vec{p}), \vec{q}) \to r$ with $|\vec{w}| = i$ in $\mathcal{R}$ matches a term t. If a rule in (2) reduces t to a term t', then

—either a rule $f(\vec{w}, g(\vec{p}), \vec{w}) \to r$ with $|\vec{w}| = i$ in $\mathcal{R}$ matches t, in which case t' can be reduced to r by a rule (1);

—or such matchings all fail, in which case a rule (3) reduces t' back to t, where a superscript d is attached to f, to prevent that a rule in (2) matches t again (d is mnemonic for "duplicate").

Rewrite rules in $\mathcal{R}$ of the form $f(\vec{w}, \vec{s}) \to r$ with $|\vec{w}| > i$ only apply to terms which do not match any left-hand sides in $\mathcal{R}$ of the form $f(\vec{w}, g(\vec{p}), \vec{q})$ with $|\vec{w}| = i$, due to the specificity ordering. In particular, such rewrite rules may apply to terms which have been reduced subsequently by a rule in (2) and a rule in (3). So if there exists a rule in (3), then rewrite rules $f(\vec{w}, \vec{s}) \to r$ with $|\vec{w}| > i$ in $\mathcal{R}$ are replaced by $f^d(\vec{w}, \vec{s}) \to r$. In this case, rule (4) makes sure that the superscript d is also attached to occurrences of f in terms which cannot be reduced by any rules in (2).

Note that the choice of loci for the function symbols f^d and f_g, namely i, ensures that the minimal rules (2) and (3) are stratified.

Remark. Since $\mathcal{R}$ is simply complete, the function symbol f does not occur in normal forms. So there is no need to change occurrences of f *inside* left-hand sides of rules in $\mathcal{R}$ 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.

The procedure "Minimize Left-Hand Sides" is based on an efficient pattern-match strategy using automata similar to Hoffmann and O'Donnell [1982] (see also Walters [1991, Chapter 3]). We give an example.

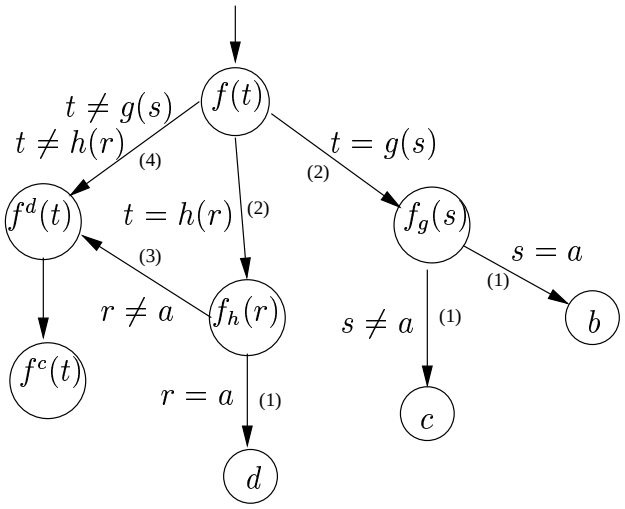


Fig. 1. A pattern-match automaton.

Example 3.4.1. 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:

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

Pattern matching of a term $f(t)$ with respect to the left-hand sides of the TRS above can be expressed by the automaton shown in Figure 1. 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 to the reducts of $f(t)$ after application of these minimal rules.

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 nonminimal rewrite rule contains only one function symbol in its left-hand side. Also assume a locus $L : \mathcal{F} \to \mathbb{N}$. The following procedure 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 symbol.

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 two most general rules:

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

Case 2
($r = h(\vec{w})$). Since $f(\vec{v}) \to h(\vec{w})$ is not minimal, in particular, it is not of type M4, so that $\vec{v} = \vec{x}, \vec{y}$ and $\vec{w} = \vec{x}, \vec{z}$, where $\vec{z}$ is non-empty, and z_1 is not the first element of $\vec{y}$. We distinguish two cases.

Case 2.1.
$\vec{y}$ is non-empty, and y_1 does not occur in $\vec{z}$. Then $\vec{y} = \vec{y}', \vec{y}''$ where
—$\vec{y}'$ is non-empty;
—$\vec{y}' \cap \vec{z}$ is empty;
—either $\vec{y}''$ is empty or $y_1'' \in \vec{z}$; and
—$\vec{y}'' \neq \vec{z}$.
Add a fresh function symbol f^d to $\mathcal{F}$, of arity $|\vec{x}| + |\vec{y}''|$ and with locus $|\vec{x}|$, and replace the rule $f(\vec{v}) \to r$ in $\mathcal{R}$ by two most general rules:

$$\begin{aligned} f(\vec{v}) &\to f^d(\vec{x}, \vec{y}'') \\ f^d(\vec{x}, \vec{y}'') &\to r. \end{aligned}$$

Case 2.2.
Either $\vec{y}$ is empty, or $y_1 \neq z_1$ and $y_1 \in \vec{z}'$, where $\vec{z} = z_1, \vec{z}'$. 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 two most general rules:

$$\begin{aligned} f(\vec{v}) &\to f^d(\vec{x}, z_1, \vec{y}) \\ f^d(\vec{x}, u, \vec{y}) &\to h(\vec{x}, u, \vec{z}'), \end{aligned}$$

where u is a fresh variable.

Case 3
($r = h(\vec{w}, g(\vec{p}), \vec{q})$). 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 two most general rules:

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

where $|\vec{x}| = |\vec{w}|$.

The resulting TRS $\mathcal{S}$ is still simply complete, because in all three 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, $\mathcal{S}$ is still left-linear, and its nonminimal 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 $\mathcal{S}$. Apply the procedure "Minimize Right-Hand Sides" to $\mathcal{S}$. *(End of procedure)*

In the electronic appendix it is proved that the procedure "Minimize Right-Hand Sides" terminates and constitutes a correct transformation; see Section A.4. Since the procedure "Minimize Right-Hand Sides" terminates, it produces a simply complete MTRS with a locus. The adaptation of right-hand sides does not affect the resulting normal forms. Namely, there exists a most general rewrite rule for the fresh function symbol f^d or h_g, so that it does not occur in normal forms.

The intuition behind the transformation of $\mathcal{R}$ into $\mathcal{S}$ is as follows. In all cases, the selected nonminimal rewrite rule $f(\vec{v}) \to r$ is replaced by two new rewrite rules, which together simulate the rewrite steps defined by $f(\vec{v}) \to r$. One of the new rules is minimal, while the other is, in a certain sense, smaller than $f(\vec{v}) \to r$ (see the termination argument for this procedure in the electronic appendix).

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.

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 procedure 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 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 $\mathcal{M}$ of the form $f(\vec{w}, \vec{s})$ with $|\vec{w}| \geq |\vec{x}|$ by $f^d(\vec{w}, \vec{s})$. Furthermore, add a stratified minimal rule

$$f(\vec{v}) \ \to \ f^d(\vec{v}).$$

Case 2. Suppose that there is a minimal rule in $\mathcal{M}$ of type M2–5, say

$$f(\vec{v}) \ \to \ r,$$

where the locus of f does not satisfy the stratification property (see Definition 3.2.2). Then add a fresh function symbol f^d, of arity $ar(f)$ and with what would have been the right locus for f, and replace this minimal rule by two minimal rules:

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

Case 3. Suppose that there is a minimal rule in $\mathcal{M}$ of type M1–4, say

$$\ell \ \to \ h(\vec{s}),$$

where the locus of h does not satisfy the stratification property (see Definition 3.2.2). Then add a fresh function symbol h^d, of arity $ar(h)$ and with what would have been the right locus for h, and replace this minimal rule by two minimal rules:

$$\begin{aligned} \ell &\to h^d(\vec{s}) \\ h^d(\vec{v}) &\to h(\vec{v}). \end{aligned}$$

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 3.2.1. Apply the procedure "Stratify" to the resulting MTRS. *(End of procedure)*

In the electronic appendix it is proved that the procedure "Stratify" terminates and constitutes a correct transformation; see Section A.5. Since the procedure "Stratify" terminates, it produces a stratified simply complete MTRS. The adaptation of the MTRS does not affect the resulting normal forms. Namely, there exists a most general rewrite rule for the fresh function symbol f^d or h^d, so that it does not occur in normal forms.

The intuition behind the transformation of $\mathcal{M}$ is as follows. In Case 1, the left-hand side $f(\vec{x}, g(\vec{y}), \vec{z})$ is stratified: a fresh function symbol f^d is introduced with locus $|\vec{x}|$, and the left-hand side is replaced by $f^d(\vec{x}, g(\vec{y}), \vec{z})$. In order to preserve the specificity ordering, all left-hand sides of the form $f(\vec{w}, \vec{s})$ with $|\vec{w}| \geq |\vec{x}|$ are replaced by $f^d(\vec{w}, \vec{s})$. Since the transformed left-hand sides only match terms of the form $f^d(\vec{t})$, a stratified minimal rule $f(\vec{v}) \to f^d(\vec{v})$ is introduced to replace occurrences of f in terms by f^d.

In Cases 2 and 3, the nonstratified 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.

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*:

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

The function symbols *zero*, *succ*, *plus*, and $plus^c$ all have locus 0.

The two original rewrite rules are not minimal. 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 the rules (2) and (1) in this procedure, respectively. Moreover, the procedure "Minimize Left-Hand Sides" replaces the second rule by

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

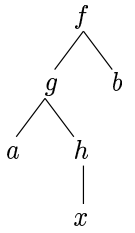


Fig. 2. Parse tree.

The fresh function symbols $plus_{zero}$ and $plus_{succ}$ both have locus 0. The resulting simply complete MTRS satisfies the stratification criteria in Definition 3.2.2.

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. Rules in the MTRS which have the same function symbol at the head of their left-hand sides, are 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 set D, the collection $Stack(D)$ consists of lists over D, whereby the elements of a list are separated by the $\cdot$ notation, and ε_D represents the empty list. Two lists S_1 and S_2 can be linked in the standard fashion, to obtain a list $S_1 \cdot S_2$. We view lists as stacks, by assuming obvious definitions for top, pop, and push. In running text we take stacks to grow from right to left; that is, 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 $\mathcal{F} \cup \mathcal{V} \cup \{\bot\}$, where $\bot$ is a special element that is always placed at the bottom of the control stack. This element is added for efficiency reasons; it prevents us from having to check for 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 4.1.1. The stacks $ri\text{-}stack(t)$ for $t \in \mathbb{T}(\Sigma)$ are defined inductively by

—$ri\text{-}stack(x) = x$;
—$ri\text{-}stack(f(t_1, \ldots, t_{ar(f)})) = ri\text{-}stack(t_{ar(f)}) \cdot \ldots \cdot ri\text{-}stack(t_1) \cdot f$.

The control stack $control(t)$ is $ri\text{-}stack(t) \cdot \bot$.

Example 4.1.2. The term $f(g(a, h(x)), b)$, of which the parse tree is depicted in Figure 2, is transformed into the control stack $b \cdot x \cdot h \cdot a \cdot g \cdot f \cdot \bot$.

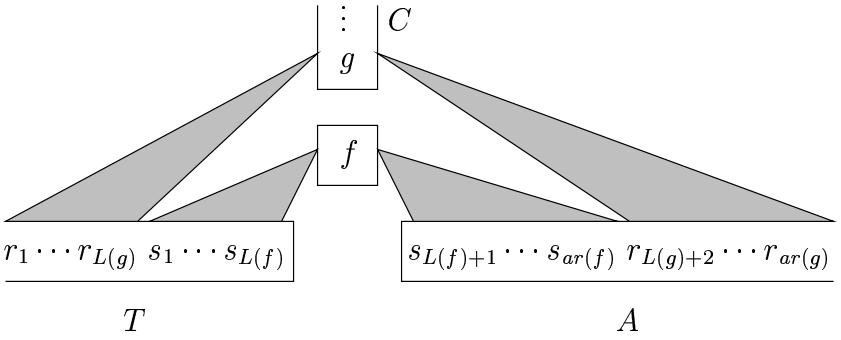


Fig. 3. Interplay between control, argument, and traversal stack.

The reduction of a term t, with respect to some stratified simply complete MTRS, is 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 mimics the rightmost innermost rewriting strategy. Intuitively, the elements on the control stack invoke the following behavior.

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 $t_1, \ldots, t_{ar(f)}$ were previously collected from the control stack, and used to produce their respective normal forms $s_1, \ldots, s_{ar(f)}$. 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: $s_{L(f)}, \ldots, s_1$ are on top of the traversal stack, while $s_{L(f)+1}, \ldots, s_{ar(f)}$ are on top of the argument stack. These terms are used to obtain the normal form of $f(s_1, \ldots, s_{ar(f)})$, by means of a number of ARM instructions (explained in Section 4.3). Subsequently, the terms $s_1, \ldots, s_{ar(f)}$ are removed from the argument and the traversal stack, and the normal form of $f(s_1, \ldots, s_{ar(f)})$ is stored on top of the argument stack.

Example 4.1.3. Figure 3 pictures a typical example of the interplay between control, argument, and traversal stack. The top elements of the three stacks are at the center 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 $s_{L(f)}, \ldots, s_1$ and $s_{L(f)+1}, \ldots, s_{ar(f)}$ are on top of T and A, respectively. Function symbol g is on top of C, with its arguments $r_{L(g)}, \ldots, r_1$ and $r_{L(g)+2}, \ldots, r_{ar(g)}$ divided over T and A, respectively. Note that the argument $r_{L(g)+1}$ of g is missing; this will be the normal form of $f(s_1, \ldots, s_{ar(f)})$.

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 2.1.4(1)).

Finally, if the bottom element $\perp$ 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 is always empty, and the argument stack always consists of 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.

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 this term is transformed into a control stack. Furthermore, the MTRS is transformed into a program, being a table of ARM instructions.

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 its executable stack is executed. This execution continues until either an instruction on the executable stack of f invokes the execution of another executable stack, or the bottom of the executable stack of f 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 modeling executable code as stacks.

Program Table. Each stratified simply complete MTRS $(\mathcal{M}, L)$ is transformed into a *program table*, denoted by $\mathit{program}(\mathcal{M}, L)$. 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 I. In the following definitions, f, g, and h range over $\mathcal{F}$, and k ranges over $\mathbb{N}$.

—**match**(g, h): If the top element of the argument stack is of the form $g(t_1, \ldots, t_k)$, then replace this term by its arguments $t_1, \ldots, t_k$, and proceed with the execution with respect to the function symbol h (i.e., replace the current executable stack by the executable stack with address h in the program table). Otherwise, ignore this instruction.

—**copya**(k): Copy the kth term of the argument stack on top of the argument stack.

—**copyt**(k): Copy the kth term of the traversal stack on top of the argument stack.

—**push**(h): Push h 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 $t_1,\ldots,t_k$ on top of the argument stack by the term $f(t_1,\ldots,t_k)$. The argument k always equals $ar(f)$.

—**goto**(h): Proceed with the execution with respect to the function symbol h.

—**recycle**: Proceed with 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, in order 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**$(h)$
	$f(\vec{x})$	$\to h(\vec{x})$	$L(f)<L(h)$	**skip**$(L(h)-L(f))\cdot$**goto**(h)
	$f(\vec{x})$	$\to h(\vec{x})$	$L(f)=L(h)$	**goto**(h)
	$f(\vec{x})$	$\to h(\vec{x})$	$L(f)>L(h)$	**retract**$(L(f)-L(h))\cdot$**goto**(h)
M5	$f(\vec{x},y)$	$\to y$	$\lvert\vec{x}\rvert\neq 0$	**tdrop**$(\lvert\vec{x}\rvert)\cdot$**recycle**
	$f(y)$	$\to y$		**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, M2, M3, M4 with $|\vec{y}|\neq 0$, and M5, we have $L(f)=L(h)=|\vec{x}|$, because $(\mathcal{M},L)$ is stratified (see Definition 3.2.2). 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 **match**(g,h) tests whether the $(|\vec{x}|+1)$th argument of the left-hand side, which is on 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 **push**(h) pushes function symbol h onto the control stack, after which instruction **goto**(g) can proceed with (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 3.2.1 (2)).

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

—For minimal rules of type M4 with $|\vec{y}| \neq 0$, instruction **adrop**$(|\vec{y}|)$ deletes the $|\vec{y}|$ arguments of the left-hand side that are on top of the argument stack, after which **goto**(h) proceeds with 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 sides may differ. If $L(f) < L(h)$, then **skip**$(L(h) - L(f))$ transfers the top $L(h) - L(f)$ terms from the argument to the traversal stack, and if $L(f) > L(h)$, then **retract**$(L(f) - L(h))$ transfers the top $L(f) - L(h)$ terms from the traversal to the argument stack. Next, **goto**(h) proceeds with the execution with respect to h.

—For minimal rules of type M5 with $|\vec{x}| \neq 0$, instruction **tdrop**$(|\vec{x}|)$ eliminates the first $|\vec{x}|$ arguments of the left-hand side, which are on top of the traversal stack. Next, **recycle** proceeds with 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 **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

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

The intuition behind this construction is as follows. The **match**(g, h) instructions on top of the executable stack for f test whether a minimal rule $f(\vec{x}, g(\vec{y}), \vec{z}) \to h(\vec{x}, \vec{y}, \vec{z})$ of type M1 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 of type M2–5 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 (1)), so that its arguments $t_1, \ldots, t_{ar(f)}$ are on top of the argument stack. Moreover, in this case $f(t_1, \ldots, t_{ar(f)})$ is a normal form. Then the instruction **build**$(f, ar(f))$ on the executable stack for f replaces the terms $t_1, \ldots, t_{ar(f)}$ on top of the argument stack by $f(t_1, \ldots, t_{ar(f)})$, after which **recycle** proceeds with the execution at the top of the control stack.

Remark. For efficiency reasons, large sequences of **match** instructions should be performed table-driven rather than iterative. That is, the **match**(g, h) instructions on top of an executable stack are implemented as a hash table. This data type is suitable in situations where the collection of used addresses is sparse in comparison with the collection of possible addresses (see e.g., Cormen et al. [1990]).

Some more efficiency can be gained as follows. If the executable stack of a function symbol f consists of the single instruction **goto**(h) with $h \neq f$, then all occurrences of f in **goto** and **push** instructions and on the right-hand side of **match** instructions in the executable stacks of other function symbols can be replaced by h. Furthermore, if the executable stack of a function symbol f consists of the single instruction **recycle**, then all occurrences of **push**(f) instructions in the executable stacks of other function symbols can be eliminated.

Finally, the stratified simply complete MTRS $(\mathcal{M}, L)$ is turned into a program table 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. The resulting table is denoted by $\mathit{program}(\mathcal{M}, L)$.

Remark. At the bottom of each executable stack there is either an instruction **goto** or an instruction **recycle**. Moreover, at the bottom of the control stack there is a special bottom element $\perp$. This observation ensures that it is unnecessary to check whether 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 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 contains a sufficient number of elements; see the electronic appendix. These observations are important for efficiency reasons; repeated checks on the emptiness of stacks would be 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 I. The states of ARM are represented by five-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 I use an auxiliary function $\mathit{get}(f, P)$, which produces the executable stack with address f in table P.

Let $\mathcal{F}_0$ consist of those function symbols in $\mathcal{F}$ that have locus 0, and let Σ_0 denote the original signature restricted to $\mathcal{F}_0$. The reduction of a term $t \in \mathbb{T}(\Sigma_0)$ by means of a stratified simply complete MTRS $(\mathcal{M}, L)$ is simulated by the expression

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

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 I 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. A formal proof of this fact is given in the electronic appendix; see Section A.6.

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 $(\mathcal{M}, L)$, which is then transformed into a table $\mathit{program}(\mathcal{M}, L)$ of ARM instructions. Furthermore, we construct the control stack of t, which is obtained by collecting the function symbols and variables of t in a rightmost innermost fashion. The

Table I. Semantics of ARM.

$$
\begin{aligned}
< P, C, \mathbf{match}(g,h) \cdot E, g(t_1,\ldots,t_k) \cdot A, T > &\rightarrow < P, C, get(h,P), t_1 \cdot \ldots \cdot t_k \cdot A, T > \\
< P, C, \mathbf{match}(g,h) \cdot E, t \cdot A, T > &\rightarrow < P, C, E, t \cdot A, T > \quad \text{if } t \neq g(t_1,\ldots,t_k) \\
< P, C, \mathbf{copya}(k) \cdot E, t_1 \cdot \ldots \cdot t_k \cdot A, T > &\rightarrow < P, C, E, t_k \cdot t_1 \cdot \ldots \cdot t_k \cdot A, T > \\
< P, C, \mathbf{copyt}(k) \cdot E, A, t_1 \cdot \ldots \cdot t_k \cdot T > &\rightarrow < P, C, E, t_k \cdot A, t_1 \cdot \ldots \cdot t_k \cdot T > \\
< P, C, \mathbf{push}(f) \cdot E, A, T > &\rightarrow < P, f \cdot C, E, A, T > \\
< P, C, \mathbf{adrop}(k) \cdot E, t_1 \cdot \ldots \cdot t_k \cdot A, T > &\rightarrow < P, C, E, A, T > \\
< P, C, \mathbf{tdrop}(k) \cdot E, A, t_1 \cdot \ldots \cdot t_k \cdot T > &\rightarrow < P, C, E, A, T > \\
< P, C, \mathbf{skip}(k) \cdot E, t_1 \cdot \ldots \cdot t_k \cdot A, T > &\rightarrow < P, C, E, A, t_k \cdot \ldots \cdot t_1 \cdot T > \\
< P, C, \mathbf{retract}(k) \cdot E, A, t_1 \cdot \ldots \cdot t_k \cdot T > &\rightarrow < P, C, E, t_k \cdot \ldots \cdot t_1 \cdot A, T > \\
< P, C, \mathbf{build}(f,k) \cdot E, t_1 \cdot \ldots \cdot t_k \cdot A, T > &\rightarrow < P, C, E, f(t_1,\ldots,t_k) \cdot A, T > \\
< P, C, \mathbf{goto}(f), A, T > &\rightarrow < P, C, get(f,P), A, T > \\
< P, f \cdot C, \mathbf{recycle}, A, T > &\rightarrow < P, C, get(f,P), A, T > \\
< P, x \cdot C, \mathbf{recycle}, A, T > &\rightarrow < P, C, \mathbf{recycle}, x \cdot A, T > \\
< P, \bot, \mathbf{recycle}, t, \varepsilon_T > &\rightarrow t
\end{aligned}
$$

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 I. Execution proceeds until the executable stack consists of a **recycle** instruction, and the control stack contains only the bottom element $\bot$. 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 stratified simply complete MTRS

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

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

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

In the sequences for $plus$ and $plus_{zero}$, 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 $plus(succ(zero),zero)$ to its normal form $succ(zero)$, by means of the state transition rules for ARM in Table I. Let P denote the program table above, and note that the control stack of $plus(succ(zero),zero)$ is $zero\cdot zero\cdot succ\cdot plus\cdot\bot$. The execution proceeds as follows, whereby in each of its 15 steps adaptations of stacks have been underlined.

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

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

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

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

because then the resulting MTRS would incorporate function symbols with locus 1. The reduction of the term $plus(zero,succ(zero))$ to its normal form $succ(zero)$ by means of the resulting ARM program would then 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 Kamperman [1996, Chapter 3]) to speed up copying of arguments, and swapping of terms between argument and traversal stack. In this article we do 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, the heap should be accessed as little as possible: "faster implementations use less heap" [Hartel et al. 1996, p. 649]. 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.

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 Cohen [1981] and Peyton Jones [1987, Chapter 17] for overviews of garbage collection techniques.

Remark. 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 [Sassa and Goto 1976] in the heap. Basically, hash-consing means that a term in the heap that consists of a function symbol f and addresses $a_1, \ldots, a_{ar(f)}$, gets as pointer $\langle f, a_1, \ldots, a_{ar(f)} \rangle$. 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 pointers 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 Hoffmann and O'Donnell [1982]. The idea to express pattern matching of TRSs in the language of TRSs itself was inspired by Pettersson [1992]. The notion of an MTRS, and the procedures to transform a TRS into an MTRS, are 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 be built on the heap.

Specificity ordering is also important for the efficiency of pattern matching. First, it enables us to share several matchings in the minimal rule (2) in Section 3.4. Second, specificity ordering causes **match** instructions to be executed in sequence, which makes it worthwile to combine such sequences in hash tables.

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. For example, 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 $\vec{y}$ between the arguments $\vec{x}$ and $\vec{z}$. If all arguments had been stored on the argument stack, this splice operation would take $O(|\vec{x}, \vec{y}|)$. 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 last $|\vec{z}| + 1$ arguments are on top of the argument stack. Therefore, the splice operation takes only $O(|\vec{y}|)$. While the function symbol g always stems from the original signature, in most cases the function symbol f has been introduced during the minimization process. This means that in general $|\vec{x}|$ is 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 the SECD machine [Landin 1964], which is utilized 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 [Cardelli 1984] and the categorical abstract machine [Cousineau et al. 1987], respectively. In contrast, the abstract rewriting machine in this article 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 [Turner 1979], the G-machine [Johnsson 1984], the three-instruction machine [Wray and Fairbairn 1989], and the spineless tagless G-machine [Peyton Jones and Salkild 1989]. 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 Kamperman and Walters [1995] and Kamperman [1996, Chapter 6]. A similar observation was made for the categorical abstract machine [Cousineau et al. 1987, Section 4].

Fradet and Le Métayer [1991] 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 [1994] 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 also geared more toward provability than toward efficiency, because environments are built explicitly on the heap, instead of on the cheaper stack. Klaeren and Indermark [1987] 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 Lester [1987] and Cousineau et al. [1987].

5.3 Functional Languages

Backus [1978] propagated the use of functional programming languages, and since then, several such languages have been implemented. The eager first-order equational programming language Epic [Walters and Kamperman 1996a; 1996b; Walters 1997] has been implemented by means of the ARM technology. Other first-order languages with an eager implementation are ASF+SDF [van Deursen et al. 1996] and Sisal [Cann 1992]. Neither language is compiled via an abstract machine.

There is a long tradition of eager higher-order functional languages, which dates back to the implementation of Lisp [McCarthy 1960], and Landin's proposal ISWIM [Landin 1966]. Lisp was succeeded by Scheme [Rees and Clinger 1986], and ISWIM was an inspiration for ML [Gordon et al. 1978], which in turn was succeeded by Standard ML [Milner et al. 1990]. Other eager higher-order languages include Caml [Weis and Leroy 1993], which was implemented using the categorical abstract machine, and Trafola [Alt et al. 1993], which was implemented by means of an abstract machine called Trama. More recently, several lazy higher-order functional languages have been implemented, notably: Miranda [Turner 1985] using the S-K reduction machine, Lazy ML [Augustsson and Johnsson 1989] by means of the G-machine, and Haskell [Hudak et al. 1992].

In Hartel et al. [1996] the efficiency of several functional programming languages is compared, 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" [Hartel et al. 1996, p. 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 noninterpreted 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)" [Hartel et al. 1996, p. 649]. With respect to higher-order languages, it is remarked that "higher-order functions are generally expensive to implement" [Hartel et al. 1996, p. 636]. 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.

ONLINE-ONLY APPENDIX

An appendix to this article is available in electronic form (PostScript™). Any of the following methods may be used to obtain it; or see the inside back cover of a current issue for up-to-date instructions.

—By anonymous ftp from **acm.org**, file [**pubs.journals.toplas.append**]**p1916.ps**

—Send electronic mail to **mailserve@acm.org** containing the line **send** [**anonymous.pubs.journals.toplas.append**]**p1916.ps**

—By *Gopher* from **acm.org**

—By anonymous ftp from **ftp.cs.princeton.edu**, file **pub/toplas/append/p1916.ps**

—Hardcopy from *Article Express*, for a fee: phone 800-238-3458, fax +1-516-997-0890, or write 469 Union Avenue, Westbury NY 11550; and request ACM-TOPLAS-APPENDIX-1916.

ACKNOWLEDGMENTS

We would like to thank Jan Bergstra, Mark van den Brand, Jan Friso Groote, Jan Heering, Paul Klint, Bas Luttik, Pieter Olivier, Jaco van de Pol, and John Tucker for their support. A considerable part of this research was carried out at the CWI in Amsterdam.

REFERENCES

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

Alt, M., Fecht, C., Ferdinand, C., and Wilhelm, R. 1993. TrafoLa-H subsystem. In *Program Development by Specification and Transformation: The PROSPECTRA Methodology, Language Family, and System*, B. Hoffmann and B. Krieg-Brückner, Eds. Lecture Notes in Computer Science, vol. 680. Springer-Verlag, Berlin, 539–576.

Augustsson, L. and Johnsson, T. 1989. The Chalmers Lazy-ML compiler. *Comput. J. 32,* 2, 127–141.

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

Baeten, J. C. M., Bergstra, J. A., Klop, J. W., and Weijland, W. P. 1989. Term-rewriting systems with rule priorities. *Theor. Comput. Sci. 67,* 2/3, 283–301.

Bergstra, J. A., Heering, J., and Klint, P., Eds. 1989. *Algebraic Specification.* ACM Frontier Series. ACM/Addison-Wesley, New York.

Bergstra, J. A. and Klop, J. W. 1986. Conditional rewrite rules: Confluence and termination. *J. Comput. Syst. Sci. 32,* 3, 323–362.

Burstall, R. M. and Landin, P. J. 1969. Programs and their proofs: An algebraic approach. In *Proceedings of the 4th Workshop on Machine Intelligence*, B. Melzer and D. Michie, Eds. Edinburgh University Press, Edinburgh, 17–43.

Cann, D. C. 1992. The optimizing SISAL compiler: Version 12.0. Manual UCRL-MA-110080, Lawrence Livermore National Laboratory, Livermore, Calif. Available via ftp as sisal.llnl.gov/pub/sisal/OSC-manual.ps.

Cardelli, L. 1984. Compiling a functional language. In *Proceedings of the ACM Symposium on Lisp and Functional Programming.* ACM, New York, 208–226.

Cohen, J. 1981. Garbage collection of linked data structures. *ACM Comput. Surv. 13,* 3, 341–367.

Cormen, T. H., Leiserson, C. E., and Rivest, R. L. 1990. *Introduction to Algorithms.* MIT Press, Cambridge, Mass.

Cousineau, G., Curien, P.-L., and Mauny, M. 1987. The categorical abstract machine. *Sci. Comput. Program. 8,* 2, 173–202.

Dershowitz, N. and Jouannaud, J.-P. 1990. Rewrite systems. In *Handbook of Theoretical Computer Science*, J. van Leeuwen, Ed. Vol. B. Elsevier, Amsterdam, 243–320.

Fokkink, W. J. and van de Pol, J. C. 1996. Correct implementation of rewrite systems for implementation purposes. Logic Group Preprint Series 164, Utrecht University, Utrecht. Available at http://www.phil.ruu.nl/preprints.html.

Fokkink, W. J. and van de Pol, J. C. 1997. Simulation as a correct transformation of rewrite systems. In *Proceedings of the 22nd Symposium on Mathematical Foundations of Computer Science*, I. Prívara and P. Ružička, Eds. Lecture Notes in Computer Science, vol. 1295. Springer-Verlag, Berlin, 249–258.

Fradet, P. and Le Métayer, D. 1991. Compilation of functional languages by program transformation. *ACM Trans. Program. Lang. Syst. 13,* 1, 21–51.

Gordon, M. J. C., Milner, R., Morris, L., Newey, M. C., and Wadworth, C. P. 1978. A metalanguage for interactive proof in LCF. In *Proceedings of the 5th Symposium on Principles of Programming Languages*. ACM, New York, 119–130.

Hamel, L. H. and Goguen, J. A. 1994. Towards a provably correct compiler for OBJ3. In *Proceedings of the 6th Symposium on Programming Language Implementation and Logic Programming*, M. V. Hermenegildo and J. Penjam, Eds. Lecture Notes in Computer Science, vol. 844. Springer-Verlag, Berlin, 132–146.

Hartel, P. H., Feeley, M., Alt, M., Augustsson, L., Baumann, P., Beemster, M., Chailloux, E., Flood, C. H., Grieskamp, W., van Groningen, J. H. G., Hammond, K., Hausman, B., Ivory, M. Y., Jones, R. E., Kamperman, J. F. Th., Lee, P., Leroy, X., Lins, R. D., Loosemore, S., Rjemo, N., Serrano, M., Talpin, J.-P., Thackray, J., Thomas, S., Walters, H. R., Weis, P., and Wentworth, P. 1996. Benchmarking implementations of functional languages with "pseudoknot", a float-intensive benchmark. *J. Funct. Program. 6,* 4, 621–655.

Heering, J., Hendriks, P. R. H., Klint, P., and Rekers, J. 1989. The syntax definition formalism SDF—reference manual. *ACM SIGPLAN Not. 24,* 11, 43–75.

Hoffmann, C. M. and O'Donnell, M. J. 1982. Pattern matching in trees. *J. ACM 29,* 1, 68–95.

Hudak, P., Peyton Jones, S. L., and Wadler, P. L., Eds. 1992. Report on the programming language Haskell—a non-strict purely functional language, version 1.2. *ACM SIGPLAN Not. 27,* 5, R1–R164.

Johnsson, T. 1984. Efficient compilation of lazy evaluation. In *Proceedings of the ACM Symposium on Compiler Construction. ACM SIGPLAN Not. 19,* 6, 58–69.

Kamperman, J. F. Th. 1996. Compilation of term rewriting systems. Ph.D. thesis, University of Amsterdam, Amsterdam. Available at http://www.babelfish.nl.

Kamperman, J. F. Th. and Walters, H. R. 1993. ARM—abstract rewriting machinery. In *Proceedings of Computer Science in the Netherlands*. Stichting Mathematisch Centrum, Amsterdam, 193–204.

Kamperman, J. F. Th. and Walters, H. R. 1995. Lazy rewriting on eager machinery. In *Proceedings of the 6th Conference on Rewriting Techniques and Applications*, J. Hsiang, Ed. Lecture Notes in Computer Science, vol. 914. Springer-Verlag, Berlin, 147–162.

Kamperman, J. F. Th. and Walters, H. R. 1996. Minimal term rewriting systems. In *Proceedings of the 11th Workshop on Specification of Abstract Data Types*, M. Haveraaen, O. Owe, and O.-J. Dahl, Eds. Lecture Notes in Computer Science, vol. 1130. Springer-Verlag, Berlin, 274–290.

Klaeren, H. and Indermark, K. 1987. Efficient implementation of an algebraic specification language. In *Proceedings of the Workshop on Algebraic Methods: Theory, Tools and Applications*, M. Wirsing and J. A. Bergstra, Eds. Lecture Notes in Computer Science, vol. 394. Springer-Verlag, Berlin, 69–90.

Klint, P. 1993. A meta-environment for generating programming environments. *ACM Trans. Softw. Eng. Method. 2,* 2, 176–201.

Klop, J. W. 1992. Term rewriting systems. In *Handbook of Logic in Computer Science*, S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, Eds. Vol. I. Oxford University Press, New York, 1–116.

Landin, P. 1964. The mechanical evaluation of expressions. *Comput. J. 6,* 4, 308–320.

Landin, P. 1966. The next 700 programming languages. *Commun. ACM 9,* 3, 157–166.

Lester, D. 1987. The G-machine as a representation of stack semantics. In *Proceedings of the 3rd Conference on Functional Programming Languages and Computer Architecture*, G. Kahn, Ed. Lecture Notes in Computer Science, vol. 274. Springer-Verlag, Berlin, 46–59.

McCarthy, J. 1960. Recursive functions of symbolic expressions and their computation by machine: Part I. *Commun. ACM 3,* 4, 184–195.

McCarthy, J. 1963. Towards a mathematical science of computation. In *Proceedings of Information Processing '62*. North-Holland, Amsterdam, 21–28.

Milner, R., Tofte, M., and Harper, R. 1990. *The Definition of Standard ML.* MIT Press, Cambridge, Mass.

Morris, F. L. 1973. Advice on structuring compilers and proving them correct. In *Proceedings of the 1st Symposium on Principles of Programming Languages.* ACM, New York, 144–152.

Pettersson, M. 1992. A term pattern-match compiler inspired by finite automata theory. In *Proceedings of the 4th Workshop on Compiler Construction*, U. Kastens and P. Pfahler, Eds. Lecture Notes in Computer Science, vol. 641. Springer-Verlag, Berlin, 258–270.

Peyton Jones, S. L. 1987. *The Implementation of Functional Programming Languages.* Prentice-Hall, Englewood Cliffs, N.J.

Peyton Jones, S. L. and Salkild, J. 1989. The spineless tagless G-machine. In *Proceedings of the 4th Conference on Functional Programming Languages and Computer Architecture*, D. B. MacQueen, Ed. Addison-Wesley, Reading, Mass., 184–201.

Rees, J. A. and Clinger, W., Eds. 1986. Revised^3 report on the algorithmic language Scheme. *ACM SIGPLAN Not. 21,* 12, 37–79.

Sassa, M. and Goto, E. 1976. A hashing method for fast set operations. *Inf. Process. Lett. 5,* 2, 31–34.

Thatte, S. R. 1985. On the correspondence between two classes of reduction systems. *Inf. Process. Lett. 20,* 2, 83–85.

Turner, D. A. 1979. A new implementation technique for applicative languages. *Softw. Pract. Exper. 9,* 1, 31–49.

Turner, D. A. 1985. Miranda: A non-strict functional language with polymorphic types. In *Proceedings of the 2nd Conference on Functional Programming Languages and Computer Architecture*, J.-P. Jouannaud, Ed. Lecture Notes in Computer Science, vol. 201. Springer-Verlag, Berlin, 1–16.

van den Brand, M. G. J. 1997. The missing links. In *Liber Amicorum Paul Klint.* CWI, Amsterdam, 55–60.

van Deursen, A., Heering, J., and Klint, P., Eds. 1996. *Language Prototyping: An Algebraic Specification Approach.* AMAST Series in Computing, vol. 5. World Scientific, Singapore.

Walters, H. R. 1991. On equal terms, implementing algebraic specifications. Ph.D. thesis, University of Amsterdam, Amsterdam. Available at http://www.babelfish.nl.

Walters, H. R. 1997. Epic and ARM—user's guide. Tech. Rep. SEN-R9724, CWI, Amsterdam. Report and tool set available at http://www.babelfish.nl.

Walters, H. R. and Kamperman, J. F. Th. 1996a. EPIC 1.0 (unconditional), an equational programming language. Tech. Rep. CS-R9604, CWI, Amsterdam. Available at http://www.babelfish.nl.

Walters, H. R. and Kamperman, J. F. Th. 1996b. EPIC: An equational language—abstract machine and supporting tools. In *Proceedings of the 7th Conference on Rewriting Techniques and Applications*, H. Ganzinger, Ed. Lecture Notes in Computer Science, vol. 1103. Springer-Verlag, Berlin, 424–427.

Weis, P. and Leroy, X. 1993. *Le Langage Caml.* InterÉditions, Paris.

Wray, S. and Fairbairn, J. 1989. Non-strict languages—programming and implementation. *Comput. J. 32,* 2, 142–151.

Received July 1997; revised January 1998; accepted March 1998

THIS DOCUMENT IS THE APPENDIX TO THE FOLLOWING PAPER:

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

WAN FOKKINK
University of Wales Swansea
JASPER KAMPERMAN
Cosmos Group BV
and
PUM WALTERS
Babelfish

The body of this article is to appear in *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Vol. 20, No. 3, May 1998, Pages 679–706.

In this appendix we present formal correctness proofs for the separate transformations that together make up our compilation technique. This leads to the conclusion that the compilation of left-linear TRSs into ARM programs is correct.

A.1 Simulation

In Kamperman and Walters [1996] and Kamperman [1996] a notion of *simulation* of one rewrite system by another rewrite system is introduced, to prove correctness of the transformation of a left-linear TRS into an MTRS, as described in Section 3 of this article. A simulation basically consists of a partially defined, surjective mapping ϕ, which relates terms in the simulating rewrite system to terms in the original rewrite system. We focus on *deterministic* rewrite systems, in which each term can do no more than one rewrite step.

We say that a simulation is *sound* if for each rewrite step of a term t to a term t' in the simulating rewrite system, t' can be rewritten to a term t'' in the simulating rewrite system, such that $\phi(t)$ can be rewritten to $\phi(t'')$ in zero or one steps in the original rewrite system. Furthermore, a simulation is *complete* if for each normal form t for the simulating rewrite system, $\phi(t)$ is a normal form for the original rewrite system. Finally, a simulation *preserves termination* if for each term t for which the original rewrite system is terminating for $\phi(t)$, the simulating rewrite system is terminating for t.

Fokkink and van de Pol [1997] showed that if a simulation is sound, complete, and termination preserving, then no information on normal forms in the original rewrite system is lost. That is, there exist mappings *parse* from original to transformed terms and *print* from transformed to original terms such that for each original

---

term t its normal forms can be computed as follows: compute the normal forms of $parse(t)$, and apply the *print* function to them. In Fokkink and van de Pol [1997] correctness was proved for fully defined simulations, but this proof easily extends to partially defined simulations; see Fokkink and van de Pol [1996].

The notion of simulation constitutes a useful tool for proving correctness of compilation of rewrite systems. Namely, such a compilation may involve a chain of transformations of rewrite systems. In order to prove correctness of one such transformation, it is sufficient to find a simulation relation that is sound, complete, and termination preserving. The intuitive link between the two rewrite systems before and after a transformation can be materialized into an explicit simulation relation. Simulation and its properties soundness, completeness, and termination preservation, are all conserved under composition, so it suffices to determine these properties for each consecutive step of a transformation chain. We use simulations to show that the separate transformations in our compilation technique are correct.

We proceed to present the formal definitions and properties for simulations, for the general notion of an abstract reduction system.

Definition A.1.1. An *abstract reduction system (ARS)* consists of a collection $\mathbb{A}$ of elements, together with a binary reduction relation R between elements in $\mathbb{A}$.

R^+ denotes the transitive closure of R, and R^* denotes the reflexive transitive closure of R. Normal forms are defined as before (see Definition 2.1.8): $a \in \mathbb{A}$ is a normal form for R if there does not exist an $a' \in \mathbb{A}$ with aRa'.

Definition A.1.2. A *simulation* of an ARS $(\mathbb{A}, R)$ by an ARS $(\mathbb{B}, S)$ is surjective mapping $\phi : D(\phi) \to \mathbb{A}$ with $D(\phi) \subseteq \mathbb{B}$.

In the remainder of this section we focus on deterministic ARSs $(\mathbb{A}, R)$, in which each element $a \in \mathbb{A}$ can do an R-step to no more than one element a'. In the next definitions we assume that the deterministic ARS $(\mathbb{A}, R)$ is simulated by the deterministic ARS $(\mathbb{B}, S)$ by means of the surjective mapping $\phi : D(\phi) \to \mathbb{A}$.

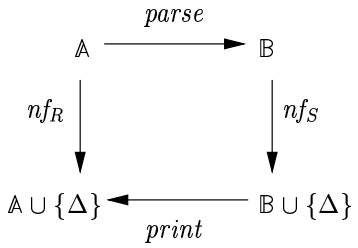
Definition A.1.3. The simulation ϕ is *sound* if, for each $b \in D(\phi)$ and $b' \in \mathbb{B}$ with bSb', there is a $b'' \in D(\phi)$ with $b'S^*b''$ and either $\phi(b) = \phi(b'')$ or $\phi(b)R\phi(b'')$.

Definition A.1.4. A simulation is *complete* if, for each normal form $b \in \mathbb{B}$ for S, $b \in D(\phi)$ and $\phi(b)$ is a normal form for R.

Definition A.1.5. A simulation *preserves termination* if for each $b_0 \in D(\phi)$ that induces an infinite S-reduction $b_0Sb_1Sb_2S\cdots$, there is a $k \geq 1$ such that $b_k \in D(\phi)$ and $\phi(b_0)R\phi(b_k)$.

Assume a deterministic ARS $(\mathbb{A}, R)$. Then $nf_R : \mathbb{A} \to \mathbb{A} \cup \{\Delta\}$ maps each $a \in \mathbb{A}$ to its normal form for R; elements that do not have a normal form, because they induce an infinite R-reduction, are mapped to Δ (which represents 'divergence').

Definition A.1.6. A deterministic ARS $(\mathbb{B}, S)$ is a *correct transformation* of a deterministic ARS $(\mathbb{A}, R)$ if there exist mappings $parse : \mathbb{A} \to \mathbb{B}$ and $print : \mathbb{B} \to \mathbb{A}$ such that the diagram below commutes:



where $print(\Delta) = \Delta$. In other words, $print(nf_S(parse(a))) = nf_R(a)$ for $a \in \mathbb{A}$.

We note that the composition of two correct transformations is again a correct transformation.

Theorem A.1.7. If a simulation between two deterministic ARSs is sound, complete, and termination preserving, then it is a correct transformation.

PROOF. Let $(\mathbb{A}, R)$ and $(\mathbb{B}, S)$ be deterministic ARSs, and let $\phi : D(\phi) \to \mathbb{A}$ be a simulation of $(\mathbb{A}, R)$ by $(\mathbb{B}, S)$ which is sound, complete, and termination preserving. We extend ϕ to Δ by defining $\phi(\Delta) = \Delta$.

Fix a $b \in D(\phi)$. Owing to completeness of ϕ, and the fact that ϕ is defined for Δ, it follows that ϕ is defined for $nf_S(b)$. We show that

$$\phi(nf_S(b)) = nf_R(\phi(b)). \tag{5}$$

We distinguish two cases.

— *Case* 1: $nf_S(b) = \Delta$.
Then b induces an infinite S-reduction $bSb_1Sb_2S\cdots$, so by termination preservation $\phi(b)$ induces an infinite R-reduction $\phi(b)R\phi(b_{i_1})R\phi(b_{i_2})R\cdots$. Hence, $\phi(nf_S(b)) = \phi(\Delta) = \Delta = nf_R(\phi(b))$.

— *Case* 2: $nf_S(b) = b' \in \mathbb{B}$.
We derive the desired equality by the length of the derivation bS^*b'.
— bS^0b'. In other words, $b = b'$.
Then b is a normal form for S, so completeness of ϕ yields that $\phi(b)$ is a normal form for R. Hence, $\phi(nf_S(b)) = \phi(b) = nf_R(\phi(b))$.
— $bS^{n+1}b'$, whereby we have already derived the desired equality for derivations of length $\leq n$.
Let bSb_0S^nb'. Since ϕ is sound and bSb_0, it follows that $b_0S^*b_1$ with $b_1 \in D(\phi)$ and $\phi(b)R^*\phi(b_1)$.
⋆ $bSb_0S^*b_1$ implies $nf_S(b) = nf_S(b_1)$.
⋆ $\phi(b)R^*\phi(b_1)$ implies $nf_R(\phi(b)) = nf_R(\phi(b_1))$.
⋆ Since bS^+b_1, the derivation b_1S^*b' has length $\leq n$. So by induction $\phi(nf_S(b_1)) = nf_R(\phi(b_1))$.
Hence,

$$\phi(nf_S(b)) \;=\; \phi(nf_S(b_1)) \;=\; nf_R(\phi(b_1)) \;=\; nf_R(\phi(b))$$

which finishes the proof of equation (5).

Let *print* be any extension of ϕ (i.e., $print(b) = \phi(b)$ for each $b \in D(\phi) \cup \{\Delta\}$), and let *parse* be any inverse of ϕ (i.e., $parse(a) \in \phi^{-1}(a)$ for each $a \in \mathbb{A}$). Equation

(5) implies that

$$print(nf_S(parse(a))) \;=\; nf_R(\phi(parse(a))) \;=\; nf_R(a)$$

for $a \in \mathbb{A}$. □

A.2 Simple Completeness

We start the correctness proof of the compilation of left-linear TRSs into ARM by showing that the procedure "Add Most General Rules" in Section 3.3 is correct. Recall that $\mathcal{R}$ denotes the original left-linear TRS, and that $\Sigma = (\mathcal{V}, \mathcal{F}, ar)$ denotes the original signature. Moreover, $\mathcal{S}$ denotes the TRS that results after adding a most general rule to $\mathcal{R}$, as described in the procedure "Add Most General Rules". $\mathcal{S}$ consists of

$$\mathcal{S}_0 \;=\; \{g(\vec{q}) \to r \mid g(\vec{p}) \to r \in \mathcal{R},\ \vec{q} \text{ obtained by replacing each } f \text{ in } \vec{p} \text{ by } f^c\}$$
$$\mathcal{S}_1 \;=\; \{f(\vec{v}) \to f^c(\vec{v})\}.$$

Let Σ' denote the original signature Σ extended with f^c, and let

$$\mathbb{B} \;=\; \{t \in \mathbb{T}(\Sigma') \mid \exists s \in \mathbb{T}(\Sigma)\,(s \to_{\mathcal{S}}^* t)\}.$$

We prove that the transformation of $(\mathbb{T}(\Sigma), \to_{\mathcal{R}})$ into $(\mathbb{B}, \to_{\mathcal{S}})$ is correct.

SIMULATION: The mapping $\phi : \mathbb{B} \to \mathbb{T}(\Sigma)$ is defined as follows: $\phi(t)$ is obtained by replacing all occurrences of the function symbol f^c in t by f. Note that ϕ is surjective, because $\mathbb{T}(\Sigma) \subseteq \mathbb{B}$ and ϕ is the identity mapping on $\mathbb{T}(\Sigma)$. Hence, ϕ constitutes a (fully defined) simulation of $(\mathbb{T}(\Sigma), \to_{\mathcal{R}})$ by $(\mathbb{B}, \to_{\mathcal{S}})$.

We proceed to prove that ϕ is complete, sound, and termination preserving.

COMPLETENESS: If $t \in \mathbb{B}$ is a normal form for $\mathcal{S}$, then $\phi(t)$ is a normal form for $\mathcal{R}$.

PROOF. We apply induction with respect to the size of t. The base case where t is a variable is trivial. We focus on the inductive case $t = g(\vec{q})$; since the $\vec{q}$ are normal forms for $\mathcal{S}$, induction implies that the $\phi(\vec{q})$ are normal forms for $\mathcal{R}$.

Let $h = \phi(g)$ (i.e., $h = f$ if $g = f^c$, and $h = g$ if $g \in \mathcal{F}$). First we prove that $h(\vec{q})$ is a normal form for $\mathcal{S}_0$. The case $g \in \mathcal{F}$ is trivial, because then $h(\vec{q}) = t$ is a normal form for $\mathcal{S}$. We focus on the case $g = f^c$. Since $t \in \mathbb{B}$, $s \to_{\mathcal{S}}^* t$ for some $s \in \mathbb{T}(\Sigma)$. Due to the innermost rewriting strategy, and the fact that f^c does not occur in the right-hand sides of rules in $\mathcal{S}_0$, the f^c is introduced by application of $\mathcal{S}_1$; that is, $f(\vec{q}) \to_{\mathcal{S}} f^c(\vec{q})$. So due to the specificity ordering, $f(\vec{q})$ must be a normal form for $\mathcal{S}_0$.

Assume, toward a contradiction, that $\phi(t) = h(\phi(\vec{q}))$ is *not* a normal form for $\mathcal{R}$. Since the $\phi(\vec{q})$ are normal forms for $\mathcal{R}$, it follows that $h(\phi(\vec{q}))$ is an $\mathcal{R}$-redex (see Definition 2.1.5). So there exists a rule $h(\vec{p}) \to r$ in $\mathcal{S}_0$ such that the corresponding rule $h(\phi(\vec{p})) \to r$ in $\mathcal{R}$ matches $h(\phi(\vec{q}))$; that is, $\sigma(\phi(\vec{p})) = \phi(\vec{q})$ for some substitution σ. Note that $\vec{p}$ does not contain any occurrences of f, because the procedure "Add Most General Rules" replaced all occurrences of f inside left-hand sides of $\mathcal{S}$ by f^c. Moreover, since the $\vec{q}$ are normal forms for $\mathcal{S}$, they do not contain occurrences of f, due to the rewrite rule $\mathcal{S}_1$. Then $\sigma(\vec{p}) = \sigma(\phi(\vec{p})) = \phi(\vec{q}) = \vec{q}$, so the rule $h(\vec{p}) \to r$ in $\mathcal{S}_0$ matches $h(\vec{q})$. This contradicts the fact that $h(\vec{q})$ is

a normal form for $\mathcal{S}_0$. Hence, our assumption was false: $\phi(t)$ is a normal form for $\mathcal{R}$. □

We state an observation on specificity or rewrite rules. It is left to the reader to verify this simple fact.

A. If $g(\vec{q}) \to r \in \mathcal{S}_0$ is the most specific rule in $\mathcal{S}$ that matches $\sigma(g(\vec{q}))$, then $g(\phi(\vec{q})) \to r \in \mathcal{R}$ is the most specific rule in $\mathcal{R}$ that matches $\sigma'(g(\phi(\vec{q})))$, where $\sigma' = \phi \circ \sigma$; that is, $\sigma'(x) = \phi(\sigma(x))$ for variables x.

In the sequel, a context $Con[]$ is an expression with one occurrence of the symbol [], such that replacing this symbol by a term t results in a term $Con[t]$.

SOUNDNESS: If $t \in \mathbb{B}$ and $t \to_{\mathcal{S}} t'$, then either $\phi(t) = \phi(t')$ or $\phi(t) \to_{\mathcal{R}} \phi(t')$.

PROOF. We distinguish two cases.

— *Case* 1: $t \to_{\mathcal{S}} t'$ is the result of an application of the rewrite rule $\mathcal{S}_1$.
Then clearly $\phi(t) = \phi(t')$, so that we are done.

— *Case* 2: $t \to_{\mathcal{S}} t'$ is the result of an application of a rule $g(\vec{q}) \to r$ in $\mathcal{S}_0$.
Then $g \in \mathcal{F}$, and $g(\phi(\vec{q})) \to r$ is a rewrite rule in $\mathcal{R}$. Furthermore, $t = Con[\sigma(g(\vec{q}))]$ and $t' = Con[\sigma(r)]$ for some context $Con[]$ and some substitution σ, whereby $\sigma(g(\vec{q}))$ is the rightmost innermost $\mathcal{S}$-redex of t, and $g(\vec{q}) \to r$ is the most specific rule in $\mathcal{S}$ that matches this redex.
Since $g \in \mathcal{F}$, $\phi(t) = Con'[\sigma'(g(\phi(\vec{q})))]$ with $Con'[] = \phi(Con[])$ and $\sigma' = \phi \circ \sigma$. Since $\sigma(g(\vec{q}))$ is the rightmost innermost $\mathcal{S}$-redex of $t \in \mathbb{B}$, completeness of ϕ implies that $\sigma'(g(\phi(\vec{q})))$ is the rightmost innermost $\mathcal{R}$-redex of $\phi(t)$. Moreover, $g(\vec{q}) \to r$ is the most specific rule in $\mathcal{S}$ that matches $\sigma(g(\vec{q}))$, so observation A implies that $g(\phi(\vec{q})) \to r$ is the most specific rule in $\mathcal{R}$ that matches $\sigma'(g(\phi(\vec{q})))$. Hence, $\phi(t) = Con'[\sigma'(g(\phi(\vec{q})))] \to_{\mathcal{R}} Con'[\sigma'(r)]$. Furthermore, $r \in \mathbb{T}(\Sigma)$ so $\phi(r) = r$, and so $\phi(t') = \phi(Con[\sigma(r)]) = Con'[\sigma'(r)]$. Hence, $\phi(t) \to_{\mathcal{R}} \phi(t')$.

□

TERMINATION PRESERVATION: If $t_0 \in \mathbb{B}$ allows an infinite $\mathcal{S}$-reduction $t_0 \to_{\mathcal{S}} t_1 \to_{\mathcal{S}} t_2 \to_{\mathcal{S}} \cdots$, then there is a $k \geq 1$ such that $\phi(t_0) \to_{\mathcal{R}} \phi(t_k)$.

PROOF. Since t_0 contains only finitely many occurrences of the function symbol f, there can only be a finite number of applications of the rule $\mathcal{S}_1$ in a row with respect to t_0. Hence, there exists a smallest $k \geq 1$ such that $t_{k-1} \to_{\mathcal{S}} t_k$ is the result of an application of a rule in $\mathcal{S}_0$.

—$t_{i-1} \to_{\mathcal{S}} t_i$ for $i = 1, \ldots, k-1$ is the result of an application of the rewrite rule $\mathcal{S}_1$, so clearly $\phi(t_{i-1}) = \phi(t_i)$.

—Since $t_{k-1} \to_{\mathcal{S}} t_k$ is the result of an application of a rewrite rule in $\mathcal{S}_0$, we saw in Case 2 of the soundness proof that $\phi(t_{k-1}) \to_{\mathcal{R}} \phi(t_k)$.

So $\phi(t_0) = \phi(t_{k-1}) \to_{\mathcal{R}} \phi(t_k)$. □

Hence, the simulation $\phi : \mathbb{B} \to \mathbb{T}(\Sigma)$ is sound, complete, and termination preserving. So the transformation of $(\mathbb{T}(\Sigma), \to_{\mathcal{R}})$ into $(\mathbb{B}, \to_{\mathcal{S}})$ is correct. Finally, we show that the procedure "Add Most General Rules" is terminating.

TERMINATION: "Add Most General Rules" terminates for each allowed input.

PROOF. Each subsequent call of the procedure "Add Most General Rules" strictly decreases the number of function symbols f for which there is a rewrite rule with left-hand side $f(\vec{s})$, but no most general rule with left-hand side $f(\vec{v})$. □

A.3 Minimization of Left-Hand Sides

In this section we prove that the transformation "Minimize Left-Hand Sides" in Section 3.4 is correct. Recall that $\mathcal{R}$ denotes the original simply complete left-linear TRS, and that $\Sigma = (\mathcal{V}, \mathcal{F}, ar)$ denotes the original signature. Moreover, $\mathcal{S}$ denotes the TRS that results after minimizing the left-hand side of a nonminimal rule in $\mathcal{R}$, as described in the procedure "Minimize Left-Hand Sides". $\mathcal{S}$ is defined by the following six transformation rules, whereby f and i are fixed.

$$\begin{aligned}
\mathcal{S}_0 &= \{f(\vec{w}, g(\vec{p}), \vec{q}) \to r \in \mathcal{R} \mid |\vec{w}| < i\} \cup \{h(\vec{s}) \to r \in \mathcal{R} \mid h \neq f\} \\
\mathcal{S}_1 &= \{f(\vec{x}, g(\vec{y}), \vec{z}) \to f_g(\vec{x}, \vec{y}, \vec{z}) \mid \exists f(\vec{w}, g(\vec{p}), \vec{q}) \to r \in \mathcal{R}\,(|\vec{w}| = i)\} \\
\mathcal{S}_2 &= \{f_g(\vec{w}, \vec{p}, \vec{q}) \to r \mid f(\vec{w}, g(\vec{p}), \vec{q}) \to r \in \mathcal{R} \wedge |\vec{w}| = i\} \\
\mathcal{S}_3 &= \{f_g(\vec{x}, \vec{y}, \vec{z}) \to f^d(\vec{x}, g(\vec{y}), \vec{z}) \mid \exists f(\vec{w}, g(\vec{p}), \vec{q}) \to r \in \mathcal{R}\,(|\vec{w}| = i) \\
&\qquad \text{and there is no most general rule for } f_g \text{ in } \mathcal{S}_2\} \\
\mathcal{S}_4 &= \{f(\vec{v}) \to f^d(\vec{v})\} \\
\mathcal{S}_5 &= \{f^d(\vec{w}, \vec{s}) \to r \mid f(\vec{w}, \vec{s}) \to r \in \mathcal{R} \wedge |\vec{w}| > i\}.
\end{aligned}$$

In the definition of $\mathcal{S}$ we assume that $\mathcal{S}_3$ is non-empty. In the case that $\mathcal{S}_3$ is empty, "Minimize Left-Hand Sides" does not introduce the fresh function symbol f^d. Moreover, in this case $\mathcal{S}_4$ and $\mathcal{S}_5$ are not added to $\mathcal{S}$, while rules in $\mathcal{R}$ of the form $f(\vec{w}, \vec{s}) \to r$ with $|\vec{w}| > i$ are added to $\mathcal{S}_0$. The correctness of the procedure "Minimize Left-Hand Sides" in the case that $\mathcal{S}_3 = \emptyset$ can be derived in the same way as in the case that $\mathcal{S}_3 \neq \emptyset$, with some cases for $\mathcal{S}_3$, $\mathcal{S}_4$, and $\mathcal{S}_5$ omitted.

Note that $\mathcal{R}$ can be partitioned into the following three disjoint subsets:

$$\begin{aligned}
&\mathcal{S}_0 \\
\mathcal{R}_1 &= \{f(\vec{w}, g(\vec{p}), \vec{q}) \to r \in \mathcal{R} \mid |\vec{w}| = i\} \\
\mathcal{R}_2 &= \{f(\vec{w}, \vec{s}) \to r \in \mathcal{R} \mid |\vec{w}| > i\}.
\end{aligned}$$

Let Σ' denote the original signature Σ extended with f_g and f^d. We prove that the transformation of $(\mathbb{T}(\Sigma), \to_{\mathcal{R}})$ into $(\mathbb{T}(\Sigma'), \to_{\mathcal{S}})$ is correct.

SIMULATION: Let $\phi : \mathbb{T}(\Sigma) \to \mathbb{T}(\Sigma)$ be the identity mapping. Since ϕ is surjective, it constitutes a (partially defined) simulation of $(\mathbb{T}(\Sigma), \to_{\mathcal{R}})$ by $(\mathbb{T}(\Sigma'), \to_{\mathcal{S}})$.

We proceed to prove that ϕ is complete, sound, and termination preserving.

COMPLETENESS: If $t \in \mathbb{T}(\Sigma')$ is a normal form for $\mathcal{S}$, then $t \in \mathbb{T}(\Sigma)$ and t is a normal form for $\mathcal{R}$.

PROOF. There are most general rules for the function symbols f_g and f^d in $\mathcal{S}$, so since t is a normal form for $\mathcal{S}$ we have $t \in \mathbb{T}(\Sigma)$. If there is a rule $h(\vec{p}) \to r \in \mathcal{R}$, then there is a rule $h(\vec{q}) \to r' \in \mathcal{S}$. Since $\mathcal{S}$ is simply complete, it follows that if a term $h(\vec{s})$ in $\mathbb{T}(\Sigma)$ is an $\mathcal{R}$-redex, then it is an $\mathcal{S}$-redex. □

We state six observations on specificity of rewrite rules. It is left to the reader to verify these simple facts.

A. No rule in $\mathcal{R}\backslash\mathcal{S}_0$ is more specific than any rule in $\mathcal{S}_0$.

B. If $\mathcal{S}_4$ is the most specific rule in $\mathcal{S}$ that matches $t \in \mathbb{T}(\Sigma)$, then some rule in $\mathcal{R}_2$ is the most specific rule in $\mathcal{R}$ that matches t.

C. If some rule in $\mathcal{S}_1$ is the most specific rule in $\mathcal{S}$ that matches $t \in \mathbb{T}(\Sigma)$, then some rule in $\mathcal{R}_1 \cup \mathcal{R}_2$ is the most specific rule in $\mathcal{R}$ that matches t.

D. If $f(\vec{w}, \vec{s}) \to r \in \mathcal{R}_2$ with $|\vec{w}| > i$ is the most specific rule in $\mathcal{R}$ that matches $f(\vec{t}) \in \mathbb{T}(\Sigma)$, then $f^d(\vec{w}, \vec{s}) \to r \in \mathcal{S}_5$ is the most specific rule in $\mathcal{S}$ that matches $f^d(\vec{t})$.

E. If some rule in $\mathcal{R}_2$ is the most specific rule in $\mathcal{R}$ that matches $f(\vec{\ell}, g(\vec{s}), \vec{t}) \in \mathbb{T}(\Sigma)$ with $|\vec{\ell}| = i$, then no rule in $\mathcal{S}_2$ matches $f_g(\vec{\ell}, \vec{s}, \vec{t})$.

F. If $f(\vec{w}, g(\vec{p}), \vec{q}) \to r \in \mathcal{R}_1$ with $|\vec{w}| = i$ is the most specific rule in $\mathcal{R}$ that matches $f(\vec{\ell}, g(\vec{s}), \vec{t}) \in \mathbb{T}(\Sigma)$ with $|\vec{\ell}| = i$, then $f_g(\vec{w}, \vec{p}, \vec{q}) \to r \in \mathcal{S}_2$ is the most specific rule in $\mathcal{S}$ that matches $f_g(\vec{\ell}, \vec{s}, \vec{t})$.

Note that observations B and C use simple completeness of $\mathcal{R}$.

SOUNDNESS: If $t \in \mathbb{T}(\Sigma)$ and $t \to_{\mathcal{S}} t'$, then there is a $t'' \in \mathbb{T}(\Sigma)$ with $t' \to_{\mathcal{S}}^* t''$ and $t \to_{\mathcal{R}} t''$.

PROOF. Since $t \in \mathbb{T}(\Sigma)$, we can distinguish three cases.

— *Case* 1: $t \to_{\mathcal{S}} t'$ is the result of an application of a rule $h(\vec{s}) \to r \in \mathcal{S}_0$.
Then $t = Con[\sigma(h(\vec{s}))]$, $t' = Con[\sigma(r)]$, $\sigma(h(\vec{s}))$ is the rightmost innermost $\mathcal{S}$-redex of t, and $h(\vec{s}) \to r$ is the most specific rule in $\mathcal{S}$ that matches this redex. Completeness of ϕ implies that $\sigma(h(\vec{s}))$ is also the rightmost innermost $\mathcal{R}$-redex of t. Furthermore, observation A implies that $h(\vec{s}) \to r$ is the most specific rule in $\mathcal{R}$ that matches $\sigma(h(\vec{s}))$. Hence, $t \to_{\mathcal{R}} t'$, so we can take $t'' = t'$.

— *Case* 2: $t \to_{\mathcal{S}} t'$ is the result of an application of rule $\mathcal{S}_4$.
Then $t = Con[\sigma(f(\vec{v}))]$, $t' = Con[\sigma(f^d(\vec{v}))]$, $\sigma(f(\vec{v}))$ is the rightmost innermost $\mathcal{S}$-redex of t, and $\mathcal{S}_4$ is the most specific rule in $\mathcal{S}$ that matches this redex. Completeness of ϕ implies that $\sigma(f(\vec{v}))$ is also the rightmost innermost $\mathcal{R}$-redex of t. Moreover, $\sigma(f^d(\vec{v}))$ is clearly the rightmost innermost $\mathcal{S}$-redex of t'.
By observation B, some rule $f(\vec{w}, \vec{s}) \to r \in \mathcal{R}_2$ with $|\vec{w}| > i$ is the most specific rule in $\mathcal{R}$ that matches $\sigma(f(\vec{v}))$. Hence,

$$t \to_{\mathcal{R}} Con[\sigma'(r)]$$

with $\sigma'(\vec{w}, \vec{s}) = \sigma(\vec{v})$. By observation D, $f^d(\vec{w}, \vec{s}) \to r \in \mathcal{S}_5$ is the most specific rule in $\mathcal{S}$ that matches $\sigma(f^d(\vec{v}))$. Hence,

$$t' \to_{\mathcal{S}} Con[\sigma'(r)].$$

So we can take $t'' = Con[\sigma'(r)]$.

— *Case* 3: $t \to_{\mathcal{S}} t'$ is the result of an application of a rule $f(\vec{x}, g(\vec{y}), \vec{z}) \to f_g(\vec{x}, \vec{y}, \vec{z}) \in \mathcal{S}_1$.
Then $t = Con[\sigma(f(\vec{x}, g(\vec{y}), \vec{z}))]$, $t' = Con[\sigma(f_g(\vec{x}, \vec{y}, \vec{z}))]$, $\sigma(f(\vec{x}, g(\vec{y}), \vec{z}))$ is the rightmost innermost $\mathcal{S}$-redex of t, and $f(\vec{x}, g(\vec{y}), \vec{z})$ is the most specific left-hand side in $\mathcal{S}$ that matches this redex. Completeness of ϕ implies that $\sigma(f(\vec{x}, g(\vec{y}), \vec{z}))$

is also the rightmost innermost $\mathcal{R}$-redex of t. Furthermore, $\sigma(f_g(\vec{x},\vec{y},\vec{z}))$ is clearly the rightmost innermost $\mathcal{S}$-redex of t'.
By observation C we can distinguish two cases.
— *Case* 3.1: Some rule $f(\vec{w},g(\vec{p}),\vec{q}) \to r \in \mathcal{R}_1$ with $|\vec{w}| = i$ is the most specific rule in $\mathcal{R}$ that matches $\sigma(f(\vec{x},g(\vec{y}),\vec{z}))$. Hence,

$$t \to_{\mathcal{R}} Con[\sigma'(r)]$$

with $\sigma'(\vec{w},\vec{p},\vec{q}) = \sigma(\vec{x},\vec{y},\vec{z})$. By observation F, $f_g(\vec{w},\vec{p},\vec{q}) \to r \in \mathcal{S}_2$ is the most specific rule in $\mathcal{S}$ that matches $\sigma(f_g(\vec{x},\vec{y},\vec{z}))$. Hence,

$$t' \to_{\mathcal{S}} Con[\sigma'(r)].$$

So we can take $t'' = Con[\sigma'(r)]$.
— *Case* 3.2: Some rule $f(\vec{w},\vec{s}) \to r \in \mathcal{R}_2$ with $|\vec{w}| > i$ is the most specific rule in $\mathcal{R}$ that matches $\sigma(f(\vec{x},g(\vec{y}),\vec{z}))$. Hence,

$$t \to_{\mathcal{R}} Con[\sigma'(r)]$$

with $\sigma'(\vec{w},\vec{s}) = \sigma(\vec{x},g(\vec{y}),\vec{z})$. By observation E, $f_g(\vec{x},\vec{y},\vec{z}) \to f^d(\vec{x},g(\vec{y}),\vec{z}) \in \mathcal{S}_3$ is the most specific rule in $\mathcal{S}$ that matches $\sigma(f_g(\vec{x},\vec{y},\vec{z}))$. Hence,

$$t' \to_{\mathcal{S}} Con[\sigma(f^d(\vec{x},g(\vec{y}),\vec{z}))].$$

It is easy to see that $\sigma(f^d(\vec{x},g(\vec{y}),\vec{z}))$ is the rightmost innermost $\mathcal{S}$-redex of the term $Con[\sigma(f^d(\vec{x},g(\vec{y}),\vec{z}))]$. Furthermore, by observation D, $f^d(\vec{w},\vec{s}) \to r \in \mathcal{S}_5$ is the most specific rule in $\mathcal{S}$ that matches this redex. Hence,

$$Con[\sigma(f^d(\vec{x},g(\vec{y}),\vec{z}))] \to_{\mathcal{S}} Con[\sigma'(r)].$$

So we can take $t'' = Con[\sigma'(r)]$.

□

TERMINATION PRESERVATION: If $t_0 \in \mathbb{T}(\Sigma)$ allows an infinite $\mathcal{S}$-reduction $t_0 \to_{\mathcal{S}} t_1 \to_{\mathcal{S}} t_2 \to_{\mathcal{S}} \cdots$, then there is a $k \geq 1$ such that $t_k \in \mathbb{T}(\Sigma)$ and $t_0 \to_{\mathcal{R}} t_k$.

PROOF. Since $t_0 \to_{\mathcal{S}} t_1$, the soundness property that we derived for ϕ induces that there exists a $k \geq 1$ with $t_0 \to_{\mathcal{R}} t_k$. □

Hence, the simulation $\phi : \mathbb{T}(\Sigma) \to \mathbb{T}(\Sigma)$ is sound, complete, and termination preserving. So the transformation of $(\mathbb{T}(\Sigma), \to_{\mathcal{R}})$ into $(\mathbb{T}(\Sigma'), \to_{\mathcal{S}})$ is correct. Finally, we show that the procedure "Minimize Left-Hand Sides" is terminating.

TERMINATION: "Minimize Left-Hand Sides" terminates for each allowed input.

PROOF. In each subsequent call of the procedure 'Minimize Left-Hand Sides" the total number N of occurrences of function symbols in left-hand sides of nonminimal rules strictly decreases. This can be seen by considering the effect of the separate transformation rules $\mathcal{S}_1$-$\mathcal{S}_5$ on the value of N.

— Replacing occurrences of f by f^d, in $\mathcal{S}_5$, does not influence N.

— $\mathcal{S}_1$, $\mathcal{S}_3$, and $\mathcal{S}_4$ introduce minimal rules, which again does not influence N.

— In $\mathcal{S}_2$, for some f and i, rules of the form $f(\vec{w}, g(\vec{p}), \vec{q}) \to r$ with $|\vec{w}| = i$ are replaced by $f_g(\vec{w}, \vec{p}, \vec{q}) \to r$. Amongst the rules that are replaced, at least one is nonminimal. The left-hand side of each replacement contains one function symbol less than its original, and minimal rules stay minimal, so $\mathcal{S}_2$ strictly decreases N.

□

A.4 Minimization of Right-Hand Sides

In this section we prove that the transformation "Minimize Right-Hand Sides" in Section 3.5 is correct. Recall that $\mathcal{R}$ denotes the original simply complete left-linear TRS, in which each nonminimal rule contains only one function symbol in its left-hand side, and that $\Sigma = (\mathcal{V}, \mathcal{F}, ar)$ denotes the original signature. Moreover, $\mathcal{S}$ denotes the TRS that results after minimizing the right-hand side of a nonminimal rule $f(\vec{v}) \to r$ in $\mathcal{R}$, as described in the procedure "Minimize Right-Hand Sides". $\mathcal{S}$ consists of the collection

$$\mathcal{S}_0 \;=\; \mathcal{R} \backslash \{f(\vec{v}) \to r\}$$

together with two more rules, which depend on the following four possibilities.

— $r = v_k$ for some $k < ar(f)$. Then $\mathcal{S}_0$ is extended with

$$\mathcal{S}_1 \;=\; \{f(\vec{v}) \to f^d(v_1, \ldots, v_k)\} \cup \{f^d(v_1, \ldots, v_k) \to v_k\}.$$

— $\vec{v} = \vec{x}, \vec{y}', \vec{y}''$ and $r = h(\vec{x}, \vec{z})$, where $\vec{y}'$ is non-empty, $\vec{y}' \cap \vec{z}$ is empty, either $\vec{y}''$ is empty or $y_1'' \in \vec{z}$, and $\vec{y}'' \neq \vec{z}$. Then $\mathcal{S}_0$ is extended with

$$\mathcal{S}_2 \;=\; \{f(\vec{v}) \to f^d(\vec{x}, \vec{y}'')\} \cup \{f^d(\vec{x}, \vec{y}'') \to h(\vec{x}, \vec{z})\}.$$

— $\vec{v} = \vec{x}, \vec{y}$ and $r = h(\vec{x}, z_1, \vec{z}')$, where either $\vec{y}$ is empty, or $y_1 \neq z_1$ and $y_1 \in \vec{z}'$. Then $\mathcal{S}_0$ is extended with

$$\mathcal{S}_3 \;=\; \{f(\vec{v}) \to f^d(\vec{x}, z_1, \vec{y})\} \cup \{f^d(\vec{x}, u, \vec{y}) \to h(\vec{x}, u, \vec{z}')\}.$$

— $r = h(\vec{w}, g(\vec{p}), \vec{q})$. Then $\mathcal{S}_0$ is extended with

$$\mathcal{S}_4 \;=\; \{f(\vec{v}) \to h_g(\vec{w}, \vec{p}, \vec{q})\} \cup \{h_g(\vec{x}, \vec{y}, \vec{z}) \to h(\vec{x}, g(\vec{y}), \vec{z})\}.$$

If $\mathcal{S} = \mathcal{S}_0 \cup \mathcal{S}_i$ with $i = 1, 2, 3$, then it is easy to see that the transformation of $(\mathbb{T}(\Sigma), \to_{\mathcal{R}})$ into $(\mathbb{T}(\Sigma'), \to_{\mathcal{S}})$, with Σ' the original signature Σ extended with f^d, is correct. Namely, in these three cases, the identity mapping $\phi : \mathbb{T}(\Sigma) \to \mathbb{T}(\Sigma)$ constitutes a (partially defined) sound and complete simulation of $(\mathbb{T}(\Sigma), \to_{\mathcal{R}})$ by $(\mathbb{T}(\Sigma'), \to_{\mathcal{S}})$ that preserves termination: the rewrite rule $f(\vec{v}) \to r$ is simulated by subsequent applications of the two rewrite rules in $\mathcal{S}_i$, and vice versa.

We focus on the fourth case, where the rule $f(\vec{v}) \to h(\vec{w}, g(\vec{p}), \vec{q})$ in $\mathcal{R}$ is replaced by $\mathcal{S}_4$. The simulation of $f(\vec{v}) \to h(\vec{w}, g(\vec{p}), \vec{q})$ by the two rewrite rules in $\mathcal{S}_4$ is not so straightforward. Namely, if the first rule $f(\vec{v}) \to h_g(\vec{w}, \vec{p}, \vec{q})$ in $\mathcal{S}_4$ is applied to some term $Con[\sigma(f(\vec{v}))]$, then the arguments $\sigma(\vec{p}, \vec{q})$ in the reduct may contain subterms that are not normal forms for $\mathcal{S} = \mathcal{S}_0 \cup \mathcal{S}_4$. Such subterms are first reduced to normal form (and this reduction may not even terminate), before the second rule in $\mathcal{S}_4$ is applied to eliminate h_g.

Let Σ' denote the original signature Σ extended with h_g, and let

$$\mathbb{B} \;=\; \{t \in \mathbb{T}(\Sigma') \mid \exists s \in \mathbb{T}(\Sigma)\, (s \to_{\mathcal{S}}^* t)\}.$$

We prove that the transformation of $(\mathbb{T}(\Sigma), \to_{\mathcal{R}})$ into $(\mathbb{B}, \to_{\mathcal{S}})$ is correct.

SIMULATION: The mapping $\phi : \mathbb{B} \to \mathbb{T}(\Sigma)$ is defined inductively as follows:

$$\begin{array}{rcll} \phi(x) & = & x & \\ \phi(f'(\vec{t})) & = & f'(\phi(\vec{t})) & f' \in \mathcal{F} \\ \phi(h_g(\vec{t},\vec{t}',\vec{t}'')) & = & h(\phi(\vec{t}), g(\phi(\vec{t}')), \phi(\vec{t}'')) & |\vec{t}| = |\vec{w}|,\ |\vec{t}'| = |\vec{p}|,\ |\vec{t}''| = |\vec{q}|. \end{array}$$

ϕ is surjective, because it is the identity mapping on $\mathbb{T}(\Sigma) \subseteq \mathbb{B}$, so it constitutes a (fully defined) simulation of $(\mathbb{T}(\Sigma), \to_{\mathcal{R}})$ by $(\mathbb{B}, \to_{\mathcal{S}})$.

We proceed to prove that ϕ is complete, sound, and termination preserving.

COMPLETENESS: If $t \in \mathbb{B}$ is a normal form for $\mathcal{S}$, then $\phi(t)$ is a normal form for $\mathcal{R}$.

PROOF. t is a normal form for $\mathcal{S}$, so it does not contain occurrences of the function symbols h_g and f, because there are most general rules for these function symbols in $\mathcal{S}$. Since $t \in \mathbb{T}(\Sigma)$ we have $\phi(t) = t$, so $\phi(t)$ is a normal form for $\mathcal{S}$. Furthermore, since f does not occur in $\phi(t)$, the only rule in $\mathcal{R}\backslash\mathcal{S}$, $f(\vec{v}) \to h(\vec{w}, g(\vec{p}), \vec{q})$, does not apply to $\phi(t)$. Hence, $\phi(t)$ is a normal form for $\mathcal{R}$. □

The following observation is used in the soundness proof.

A. If t' is the rightmost innermost $\mathcal{S}$-redex of $t \in \mathbb{B}$, then there is no $\mathcal{R}$-redex rightmost innermost of $\phi(t')$ in $\phi(t)$.

PROOF. Assume toward a contradiction that there exists an $\mathcal{R}$-redex rightmost innermost of $\phi(t')$ in $\phi(t)$. Since t' is the rightmost innermost $\mathcal{S}$-redex of t, completeness of ϕ implies that this $\mathcal{R}$-redex of $\phi(t)$ must be of the form $g(\vec{s}')$, where $t = C[h_g(\vec{\ell}, \vec{s}, \vec{s}')]$ with $|\vec{\ell}| = |\vec{w}|$, the $\vec{s}, \vec{s}'$ normal forms for $\mathcal{S}$, and t' a subterm in $\vec{\ell}$. Since $t \in \mathbb{B}$, there is a $t_0 \in \mathbb{T}(\Sigma)$ with $t_0 \to_{\mathcal{S}}^* t$, so the occurrence of h_g in t was introduced by an application of the rule $f(\vec{v}) \to h_g(\vec{w}, \vec{p}, \vec{q})$ in $\mathcal{S}_4$. Since the $\vec{w}$ are variables, and we apply innermost rewriting, and h_g only occurs in the left-hand side of the second rule of $\mathcal{S}_4$, this implies that the $\vec{\ell}$ are normal forms for $\mathcal{S}$. This contradicts the fact that t' is a subterm in $\vec{\ell}$. Hence, our assumption was false: there does not exist an $\mathcal{R}$-redex rightmost innermost of $\phi(t')$ in $\phi(t)$. □

We state two more observations, on specificity of rewrite rules. It is left to the reader to verify these simple facts.

B. If $f(\vec{v}) \to h_g(\vec{w}, \vec{p}, \vec{q}) \in \mathcal{S}_4$ is the most specific rule in $\mathcal{S}$ that matches t, then $f(\vec{v}) \to h(\vec{w}, g(\vec{p}), \vec{q})$ is the most specific rule in $\mathcal{R}$ that matches $\phi(t)$.

C. If $f'(\vec{s}) \to r \in \mathcal{S}_0$ is the most specific rule in $\mathcal{S}$ that matches t, then $f'(\vec{s}) \to r$ is the most specific rule in $\mathcal{R}$ that matches $\phi(t)$.

SOUNDNESS: If $t \in \mathbb{B}$ and $t \to_{\mathcal{S}} t'$, then either $\phi(t) = \phi(t')$ or $\phi(t) \to_{\mathcal{R}} \phi(t')$.

PROOF. We distinguish three cases.

— *Case* 1: $t \to_{\mathcal{S}} t'$ is the result of an application of $h_g(\vec{x}, \vec{y}, \vec{z}) \to h(\vec{x}, g(\vec{y}), \vec{z}) \in \mathcal{S}_4$.
Then clearly $\phi(t) = \phi(t')$, so that we are done.

— *Case* 2: $t \to_{\mathcal{S}} t'$ is the result of an application of $f(\vec{v}) \to h_g(\vec{w}, \vec{p}, \vec{q}) \in \mathcal{S}_4$. Then $t = Con[\sigma(f(\vec{v}))]$, $t' = Con[\sigma(h_g(\vec{w}, \vec{p}, \vec{q}))]$, $\sigma(f(\vec{v}))$ is the rightmost innermost $\mathcal{S}$-redex of t, and $f(\vec{v})$ is the most specific left-hand side in $\mathcal{S}$ that matches this redex.
$\phi(t) = Con'[\sigma'(f(\vec{v}))]$ with $Con'[] = \phi(Con[])$ and $\sigma' = \phi \circ \sigma$. Statement A implies that $\sigma'(f(\vec{v}))$ is the rightmost innermost $\mathcal{R}$-redex of $\phi(t)$, and by observation B $f(\vec{v}) \to h(\vec{w}, g(\vec{p}), \vec{q})$ is the most specific rule in $\mathcal{R}$ that matches this redex. Hence, $\phi(t) \to_{\mathcal{R}} Con'[\sigma'(h(\vec{w}, g(\vec{p}), \vec{q}))]$. Furthermore, $\phi(t') = \phi(Con[\sigma(h_g(\vec{w}, \vec{p}, \vec{q}))]) = Con'[\sigma'(h(\vec{w}, g(\vec{p}), \vec{q}))]$. So $\phi(t) \to_{\mathcal{R}} \phi(t')$.

— *Case* 3: $t \to_{\mathcal{S}} t'$ is the result of an application of a rule $f'(\vec{s}) \to r \in \mathcal{S}_0$. Then $t = Con[\sigma(f'(\vec{s}))]$, $t' = Con[\sigma(r)]$, $\sigma(f'(\vec{s}))$ is the rightmost innermost $\mathcal{S}$-redex of t, and $f'(\vec{s}) \to r$ is the most specific rule in $\mathcal{S}$ that matches this redex.
$\phi(t) = Con'[\sigma'(f'(\vec{s}))]$ with $Con'[] = \phi(Con[])$ and $\sigma' = \phi \circ \sigma$. Statement A implies that $\sigma'(f'(\vec{s}))$ is the rightmost innermost $\mathcal{R}$-redex of $\phi(t)$, and by observation C $f'(\vec{s}) \to r$ is the most specific rule in $\mathcal{R}$ that matches this redex. Hence, $\phi(t) \to_{\mathcal{R}} Con'[\sigma'(r)]$. Furthermore, since $r \in \mathbb{T}(\Sigma)$, $\phi(t') = \phi(Con[\sigma(r)]) = Con'[\sigma'(r)]$. So $\phi(t) \to_{\mathcal{R}} \phi(t')$.

□

TERMINATION PRESERVATION: If $t_0 \in \mathbb{B}$ allows an infinite $\mathcal{S}$-reduction $t_0 \to_{\mathcal{S}} t_1 \to_{\mathcal{S}} t_2 \to_{\mathcal{S}} \cdots$, then there is a $k \geq 1$ such that $\phi(t_0) \to_{\mathcal{R}} \phi(t_k)$.

PROOF. Since t_0 contains only finitely many occurrences of the function symbol h_g, there can only be a finite number of applications of the rule $h_g(\vec{x}, \vec{y}, \vec{z}) \to h(\vec{x}, g(\vec{y}), \vec{z})$ in a row with respect to t_0. Hence, there exists a smallest $k \geq 1$ such that $t_{k-1} \to_{\mathcal{S}} t_k$ is the result of an application of a rule in $\mathcal{S}_0 \cup \{f(\vec{v}) \to h_g(\vec{w}, \vec{p}, \vec{q})\}$. The following two cases follow from the soundness proof.

—$t_{i-1} \to_{\mathcal{S}} t_i$ for $i = 1, \ldots, k-1$ is the result of an application of $h_g(\vec{x}, \vec{y}, \vec{z}) \to h(\vec{x}, g(\vec{y}), \vec{z}) \in \mathcal{S}_4$, so clearly $\phi(t_{i-1}) = \phi(t_i)$.

—Since $t_{k-1} \to_{\mathcal{S}} t_k$ is the result of an application of a rule in $\mathcal{S}_0 \cup \{f(\vec{v}) \to h_g(\vec{w}, \vec{p}, \vec{q})\}$, we saw in the Cases 2 and 3 of the soundness proof that $\phi(t_{k-1}) \to_{\mathcal{R}} \phi(t_k)$.

So $\phi(t_0) = \phi(t_{k-1}) \to_{\mathcal{R}} \phi(t_k)$. □

Hence, the simulation $\phi : \mathbb{B} \to \mathbb{T}(\Sigma)$ is sound, complete, and termination preserving. So the transformation of $(\mathbb{T}(\Sigma), \to_{\mathcal{R}})$ into $(\mathbb{B}, \to_{\mathcal{S}})$ is correct. Finally, we show that the procedure "Minimize Right-Hand Sides" is terminating.

TERMINATION: "Minimize Right-Hand Sides" terminates for each allowed input.

PROOF. We consider the separate transformation rules $\mathcal{S}_1$-$\mathcal{S}_4$ in each subsequent call of the procedure "Minimize Right-Hand Sides".

— $\mathcal{S}_1$ decreases the number of nonminimal rules by one. Moreover, this number is not increased by the other transformation rules. Hence, $\mathcal{S}_1$ can only be applied a finite number of times.

— $\mathcal{S}_4$ strictly decreases the total number of occurrences of function symbols in right-hand sides of nonminimal rules. Moreover, this number is not increased by $\mathcal{S}_2$ and $\mathcal{S}_3$. Hence, $\mathcal{S}_4$ can only be applied a finite number of times.

— $\mathcal{S}_2$ and $\mathcal{S}_3$ strictly decrease the total sum of lengths of non-compliant segments in the arguments of left- and right-hand sides of nonminimal rules. Hence, these transformation rules can only be applied a finite number of times.

□

A.5 Stratification

In this section we prove that the transformation "Stratify" in Section 3.6 is correct. Recall that $\mathcal{M}$ denotes the original simply complete MTRS, and that $\Sigma = (\mathcal{V}, \mathcal{F}, ar)$ denotes the original signature. Moreover, $\mathcal{S}$ denotes the TRS that results after stratifying a nonstratified rule $f(\vec{p}) \to r$ in $\mathcal{M}$, as described in the procedure "Stratify". The definition of $\mathcal{S}$ depends on the following three possibilities.

— $\vec{p} = \vec{x}, g(\vec{y}), \vec{z}$, $L(f) \neq |\vec{x}|$, and $\mathcal{S}$ consists of

$$\begin{aligned} \mathcal{S}_0 &= \mathcal{M} \backslash \{f(\vec{w}, \vec{s}) \to r' \in \mathcal{M} \mid |\vec{w}| \geq |\vec{x}|\} \\ \mathcal{S}_1 &= \{f(\vec{v}) \to f^d(\vec{v})\} \\ \mathcal{S}_2 &= \{f^d(\vec{w}, \vec{s}) \to r' \mid f(\vec{w}, \vec{s}) \to r' \in \mathcal{M} \wedge |\vec{w}| \geq |\vec{x}|\}. \end{aligned}$$

— $\vec{p}$ is a string of variables, the locus of f does not satisfy the stratification property, and $\mathcal{S}$ consists of

$$\begin{aligned} \mathcal{S}_3 &= \mathcal{M} \backslash \{f(\vec{p}) \to r\} \\ \mathcal{S}_4 &= \{f(\vec{v}) \to f^d(\vec{v})\} \cup \{f^d(\vec{v}) \to r\}. \end{aligned}$$

— $r = h(\vec{s})$, where the locus of h does not satisfy the stratification property, and $\mathcal{S}$ consists of

$$\begin{aligned} \mathcal{S}_3 &= \mathcal{M} \backslash \{f(\vec{p}) \to r\} \\ \mathcal{S}_5 &= \{f(\vec{p}) \to h^d(\vec{s})\} \cup \{h^d(\vec{v}) \to h(\vec{v})\}. \end{aligned}$$

In the last two cases, where the rule $f(\vec{p}) \to r$ in $\mathcal{M}$ is replaced by $\mathcal{S}_i$ with $i = 4, 5$, it is easy to see that the transformation of $(\mathbb{T}(\Sigma), \to_{\mathcal{M}})$ into $(\mathbb{T}(\Sigma'), \to_{\mathcal{S}})$ is correct, where Σ' is the original signature Σ extended with f^d or h^d. Namely, in these two cases the identity mapping $\phi : \mathbb{T}(\Sigma) \to \mathbb{T}(\Sigma)$ constitutes a (partially defined) sound and complete simulation that preserves termination; the rewrite rule $f(\vec{p}) \to r$ is simulated by consecutive applications of the two rewrite rules in $\mathcal{S}_i$, and vice versa.

We focus on the first case, where the rules $f(\vec{w}, \vec{s}) \to r'$ in $\mathcal{M}$ with $|\vec{w}| \geq |\vec{x}|$ are replaced by $\mathcal{S}_1 \cup \mathcal{S}_2$. Basically, the correctness proof for this case is a simplified version of the correctness proof for the procedure "Minimize Left-Hand Sides". Let Σ' denote the original signature extended with the fresh function symbol f^d. We prove that the transformation of $(\mathbb{T}(\Sigma), \to_{\mathcal{M}})$ into $(\mathbb{T}(\Sigma'), \to_{\mathcal{S}})$ is correct.

SIMULATION: Let $\phi : \mathbb{T}(\Sigma) \to \mathbb{T}(\Sigma)$ be the identity mapping. Since ϕ is surjective, it constitutes a (partially defined) simulation of $(\mathbb{T}(\Sigma), \to_{\mathcal{M}})$ by $(\mathbb{T}(\Sigma'), \to_{\mathcal{S}})$.

We proceed to prove that ϕ is complete, sound, and termination preserving.

COMPLETENESS: If $t \in \mathbb{T}(\Sigma')$ is a normal form for $\mathcal{S}$, then $t \in \mathbb{T}(\Sigma)$ and t is a normal form for $\mathcal{M}$.

PROOF. There is a most general rule for f^d in $\mathcal{S}$, so since t is a normal form for $\mathcal{S}$ we have $t \in \mathbb{T}(\Sigma)$. If there is a rule for $g \in \mathcal{F}$ in $\mathcal{M}$, then there is a rule for g in $\mathcal{S}$. Since $\mathcal{S}$ is simply complete, it follows that if a term $g(\vec{s})$ in $\mathbb{T}(\Sigma)$ is an $\mathcal{M}$-redex, then it is an $\mathcal{S}$-redex. □

We state three observations on specificity of rewrite rules. It is left to the reader to verify these simple facts.

A. No rule in $\mathcal{M}\backslash\mathcal{S}_0$ is more specific than any rule in $\mathcal{S}_0$.

B. If $\mathcal{S}_1$ is the most specific rule in $\mathcal{S}$ that matches $t \in \mathbb{T}(\Sigma)$, then some rule $f(\vec{w}, \vec{s}) \to r'$ with $|\vec{w}| \geq |\vec{x}|$ is the most specific rule in $\mathcal{M}$ that matches t.

C. If $f(\vec{w}, \vec{s}) \to r'$ with $|\vec{w}| \geq |\vec{x}|$ is the most specific rule in $\mathcal{M}$ that matches $f(\vec{q}) \in \mathbb{T}(\Sigma)$, then $f^d(\vec{w}, \vec{s}) \to r' \in \mathcal{S}_2$ is the most specific rule in $\mathcal{S}$ that matches $f^d(\vec{q})$.

Note that observation B uses simple completeness of $\mathcal{M}$.

SOUNDNESS: If $t \in \mathbb{T}(\Sigma)$ and $t \to_{\mathcal{S}} t'$, then there is a $t'' \in \mathbb{T}(\Sigma)$ with $t' \to_{\mathcal{S}}^* t''$ and $t \to_{\mathcal{M}} t''$.

PROOF. Since $t \in \mathbb{T}(\Sigma)$, we can distinguish two cases.

— *Case* 1: $t \to_{\mathcal{S}} t'$ is the result of an application of a rule $g(\vec{q}) \to r' \in \mathcal{S}_0$.
Then $t = Con[\sigma(g(\vec{q}))]$, $t' = Con[\sigma(r')]$, $\sigma(g(\vec{q}))$ is the rightmost innermost $\mathcal{S}$-redex of t, and $g(\vec{q}) \to r'$ is the most specific rule in $\mathcal{S}$ that matches this redex. Completeness of ϕ implies that $\sigma(g(\vec{q}))$ is also the rightmost innermost $\mathcal{M}$-redex of t. Furthermore, by observation A $g(\vec{q}) \to r'$ is also the most specific rule in $\mathcal{M}$ that matches $\sigma(g(\vec{q}))$. Hence, $t \to_{\mathcal{M}} t'$, so we can take $t'' = t'$.

— *Case* 2: $t \to_{\mathcal{S}} t'$ is the result of an application of rule $\mathcal{S}_1$.
Then $t = Con[\sigma(f(\vec{v}))]$, $t' = Con[\sigma(f^d(\vec{v}))]$, $\sigma(f(\vec{v}))$ is the rightmost innermost $\mathcal{S}$-redex of t, and $\mathcal{S}_1$ is the most specific rule in $\mathcal{S}$ that matches this redex. Completeness of ϕ implies that $\sigma(f(\vec{v}))$ is also the rightmost innermost $\mathcal{M}$-redex of t. Furthermore, $\sigma(f^d(\vec{v}))$ is clearly the rightmost innermost $\mathcal{S}$-redex of t'.
Observation B yields that some rule $f(\vec{w}, \vec{s}) \to r' \in \mathcal{M}$ with $|\vec{w}| \geq |\vec{x}|$ is the most specific rule in $\mathcal{M}$ that matches $\sigma(f(\vec{v}))$. Hence,

$$t \to_{\mathcal{M}} Con[\sigma'(r')]$$

with $\sigma'(\vec{w}, \vec{s}) = \sigma(\vec{v})$. Then by observation C $f^d(\vec{w}, \vec{s}) \to r' \in \mathcal{S}_2$ is the most specific rule in $\mathcal{S}$ that matches $\sigma(f^d(\vec{v}))$. Hence,

$$t' \to_{\mathcal{S}} Con[\sigma'(r')].$$

So we can take $t'' = Con[\sigma'(r')]$.

□

TERMINATION PRESERVATION: If $t_0 \in \mathbb{T}(\Sigma)$ allows an infinite $\mathcal{S}$-reduction $t_0 \to_{\mathcal{S}} t_1 \to_{\mathcal{S}} t_2 \to_{\mathcal{S}} \cdots$, then there is a $k \geq 1$ such that $t_k \in \mathbb{T}(\Sigma)$ and $t_0 \to_{\mathcal{M}} t_k$.

PROOF. Since $t_0 \to_{\mathcal{S}} t_1$, the soundness property that we derived for ϕ induces that there is a $k \geq 1$ with $t_0 \to_{\mathcal{M}} t_k$. □

Hence, the simulation $\phi : \mathbb{T}(\Sigma) \to \mathbb{T}(\Sigma)$ is sound, complete, and termination preserving. So the transformation of $(\mathbb{T}(\Sigma), \to_{\mathcal{M}})$ into $(\mathbb{T}(\Sigma'), \to_{\mathcal{S}})$ is correct. Finally, we show that the procedure "Stratify" is terminating.

TERMINATION: "Stratify" terminates for each allowed input.

PROOF. We consider the separate transformation rules in each subsequent call of the procedure "Stratify".

— Let N be the number of minimal rules $\ell \to r$ in $\mathcal{M}$ of type M1 such that:
(1) either $\ell \to r$ has a nonstratified left-hand side;
(2) or there is a minimal rule $\ell' \to r'$ in $\mathcal{M}$ with a nonstratified left-hand side and $\ell \to r \prec \ell' \to r'$.

If $f(\vec{p}) \to t$ is replaced by $\mathcal{S}_1 \cup \mathcal{S}_2$, then the number N strictly decreases. Moreover, this number is not increased by $\mathcal{S}_4$ and $\mathcal{S}_5$. Hence, $\mathcal{S}_1 \cup \mathcal{S}_2$ can only be applied a finite number of times.

— $\mathcal{S}_4$ strictly decreases the number of minimal rules with a nonstratified left-hand side. Moreover, this number is not increased by $\mathcal{S}_5$. Hence, $\mathcal{S}_4$ can only be applied a finite number of times.

— $\mathcal{S}_5$ strictly decreases the number of minimal rules with a nonstratified right-hand side. Hence, it can only be applied a finite number of times.

□

A.6 Abstract Rewriting Machine

In this section we prove that the transformation of MTRSs into ARM programs, and of terms into five-tuples, as described in Section 4, is correct. Recall that $(\mathcal{M}, L)$ denotes the original stratified simply complete MTRS, that $\Sigma = (\mathcal{V}, \mathcal{F}, ar)$ denotes the original signature, and that L denotes the locus function on $\mathcal{F}$. Furthermore, $\mathcal{F}_0$ consists of those function symbols in $\mathcal{F}$ that have locus 0, and Σ_0 denotes the original signature restricted to $\mathcal{F}_0$. Since L is a locus, it follows that all normal forms for $\mathcal{M}$ are in $\mathbb{T}(\Sigma_0)$ (see Definition 3.2.1 (1)).

The program table $\mathit{program}(\mathcal{M}, L)$ denotes the result of transforming $(\mathcal{M}, L)$ into an ARM program, as described in Section 4.4. For convenience of notation, $\mathit{program}(\mathcal{M}, L)$ is abbreviated to P. Furthermore, to each term t belongs a so-called control stack $\mathit{control}(t)$, which is obtained by collecting the function symbols and variables of t onto a stack in a rightmost innermost fashion, whereby a special element $\perp$ is placed at the bottom of the stack; see Definition 4.1.1. Each term $t \in \mathbb{T}(\Sigma_0)$ is transformed into a five-tuple

$$\langle P, \mathit{control}(t), i : \mathbf{recycle}, \varepsilon_A, \varepsilon_T \rangle$$

whereby i represents an auxiliary initial label. The state transition rules for ARM in Table I enable to reduce such a five-tuple to other five-tuples of the form $\langle P, C, E, A, T \rangle$, with C a control stack of function symbols, E an executable stack of ARM instructions, and A and T an argument and traversal stack of terms, and

finally to a term again. We prove that this reduction produces the normal form of t for $\mathcal{M}$.

$get(f, P)$ denotes the executable stack in program table P with address f. For the sake of the correctness proof, we label executable stacks $get(f, P)$ in five-tuples with their address f. This information is essential in the definition of the simulation mapping ϕ from five-tuples to terms in $\mathbb{T}(\Sigma)$.

Let the original ARS consist of the collection $\mathbb{A}$ of terms t in $\mathbb{T}(\Sigma)$ for which there exist a term $s \in \mathbb{T}(\Sigma_0)$ with

$$s \to_{\mathcal{M}}^* t.$$

Furthermore, let the simulating ARS consist of the collection $\mathbb{B}$ of

—five-tuples $\langle P, C, E, A, T\rangle$ for which these exists a term $t \in \mathbb{T}(\Sigma_0)$ with

$$\langle P, \mathit{control}(t), i\!:\!\mathbf{recycle}, \varepsilon_A, \varepsilon_T\rangle \to_{\mathcal{S}}^* \langle P, C, E, A, T\rangle;$$

—normal forms for $\mathcal{M}$;

The reduction relation $\to_{\mathcal{S}}$ on $\mathbb{B}$ is defined by the state transition rules for ARM as formulated in Table I. The simulation mapping $\phi : D(\phi) \to \mathbb{A}$ is defined for the following elements in $\mathbb{B}$:

—five-tuples $\langle P, \mathit{control}(t), i\!:\!\mathbf{recycle}, \varepsilon_A, \varepsilon_T\rangle$ for terms $t \in \mathbb{T}(\Sigma_0)$;

—five-tuples $\langle P, C, f\!:\!\mathit{get}(f, P), A, T\rangle$, with $\mathit{get}(f, P)$ the executable stack for f in P;

—normal forms for $\mathcal{M}$.

ϕ is defined inductively on $D(\phi)$ as follows, using an auxiliary function ψ.

$$\begin{aligned}
\phi(\langle P, C, i\!:\!\mathbf{recycle}, \varepsilon_A, \varepsilon_T\rangle) &= \psi(C, \varepsilon_A, \varepsilon_T)\\
\phi(\langle P, C, f\!:\!\mathit{get}(f, P), A, T\rangle) &= \psi(f \cdot C, A, T)\\
\phi(t) &= t
\end{aligned}$$

$$\begin{aligned}
\psi(f \cdot C, t_{L(f)+1} \cdot \ldots \cdot t_{ar(f)} \cdot A, t_{L(f)} \cdot \ldots \cdot t_1 \cdot T) &= \psi(C, f(t_1, \ldots, t_{ar(f)}) \cdot A, T)\\
\psi(\bot, t, \varepsilon_T) &= t.
\end{aligned}$$

We proceed to prove that ϕ is well-defined; that is, $\phi(b)$ reduces to a term in $\mathbb{A}$ for each $b \in D(\phi)$. Furthermore, we show that ϕ is a complete and sound simulation that preserves termination. The forthcoming proofs repeatedly refer to the state transition rules for ARM in Table I, and to the inductive definition of ϕ.

In order to limit the number of cases in the correctness proof, we focus on closed terms. Variables can be considered as constants that are in normal form. The state transition rule for popping a variable from the control stack, in Table I, agrees with the execution of $\mathbf{build}(a) \cdot \mathbf{recycle}$ with a a constant.

The following two observations are used in the forthcoming proofs.

A. If $\langle P, C, E, A, T\rangle \in \mathbb{B}$, then all terms on A and T are normal forms for $\mathcal{M}$.

PROOF. We apply induction with respect to a shortest derivation

$$\langle P, \mathit{control}(t), i\!:\!\mathbf{recycle}, \varepsilon_A, \varepsilon_T\rangle \to_{\mathcal{S}}^* \langle P, C, E, A, T\rangle$$

for a term $t \in \mathbb{T}(\Sigma_0)$. In the base case, where the length of this derivation is 0, both A and T are empty, so that observation A holds trivially. In the inductive case, we can assume by the induction hypothesis that

$$\langle P, C', E', A', T' \rangle \rightarrow_{\mathcal{S}} \langle P, C, E, A, T \rangle$$

where all terms on A' and T' are normal forms for $\mathcal{M}$. The semantics of ARM, in Table I, incorporates two state transition rules, for **match** and **build**, that introduce new terms on A and T. We show that in both cases these new terms are normal forms for $\mathcal{M}$.

—Application of a $\mathbf{match}(g, h)$ instruction can replace the element $g(t_1, \ldots, t_{ar(g)})$ on top of A' by $t_1, \ldots, t_{ar(g)}$. Since $g(t_1, \ldots, t_{ar(g)})$ is a normal form for $\mathcal{M}$, the same holds for $t_1, \ldots, t_{ar(g)}$.

—Application of a $\mathbf{build}(f, ar(f))$ instruction replaces the elements $t_1, \ldots, t_{ar(f)}$ on top of A' by $f(t_1, \ldots, t_{ar(f)})$. The existence of a $\mathbf{build}(f, ar(f))$ instruction in P implies that there is no rewrite rule for f in $\mathcal{M}$. So since $t_1, \ldots, t_{ar(f)}$ are normal forms for $\mathcal{M}$, it follows that $f(t_1, \ldots, t_{ar(f)})$ is a normal form for $\mathcal{M}$.

□

B. If $t \in \mathbb{T}(\Sigma_0)$, then $\phi(\langle P, \mathit{control}(t), i{:}\mathbf{recycle}, \varepsilon_A, \varepsilon_T \rangle) = t$.

PROOF. First we show, by induction on the size of $t \in \mathbb{T}(\Sigma_0)$, that

$$\psi(\mathit{ri\text{-}stack}(t) \cdot C, A, T) = \psi(C, t \cdot A, T). \tag{6}$$

Let t be of the form $f(t_1, \ldots, t_{ar(f)})$. (Recall that only closed terms are considered, so the case that t is a variable is omitted.) Since t is a term over Σ_0, we have $L(f) = 0$. By Definition 4.1.1 $\mathit{ri\text{-}stack}(t) = \mathit{ri\text{-}stack}(t_{ar(f)}) \cdot \ldots \cdot \mathit{ri\text{-}stack}(t_1) \cdot f$. Induction yields

$$\begin{aligned} & \psi(\mathit{ri\text{-}stack}(t_{ar(f)}) \cdot \ldots \cdot \mathit{ri\text{-}stack}(t_1) \cdot f \cdot C, A, T) \\ = \ & \psi(\mathit{ri\text{-}stack}(t_{ar(f)-1}) \cdot \ldots \cdot \mathit{ri\text{-}stack}(t_1) \cdot f \cdot C, t_{ar(f)} \cdot A, T) \\ = \ & \cdots \\ = \ & \psi(f \cdot C, t_1 \cdot \ldots \cdot t_{ar(f)} \cdot A, T) \\ = \ & \psi(C, f(t_1 \cdot \ldots \cdot t_{ar(f)}) \cdot A, T). \end{aligned}$$

In the last step we use that $L(f) = 0$. This finishes the proof of equation (6).

Next, we prove that observation B holds for terms $t \in \mathbb{T}(\Sigma_0)$. By Definition 4.1.1 we have $\mathit{control}(t) = \mathit{ri\text{-}stack}(t) \cdot \bot$. The definition of ϕ, together with equation (6), yields

$$\begin{aligned} \phi(\langle P, \mathit{ri\text{-}stack}(t) \cdot \bot, i{:}\mathbf{recycle}, \varepsilon_A, \varepsilon_T \rangle) &= \psi(\mathit{ri\text{-}stack}(t) \cdot \bot, \varepsilon_A, \varepsilon_T) \\ &= \psi(\bot, t, \varepsilon_T) \\ &= t. \end{aligned}$$

□

The following observation implies that ϕ is well-defined, surjective, sound, complete, and termination preserving.

C. If $b \in D(\phi)$ is a five-tuple with $\phi(b)$ well-defined, then $b \rightarrow_{\mathcal{S}}^{+} b'$ with $b' \in D(\phi)$ and $\phi(b')$ well-defined, and either $\phi(b) = \phi(b')$ or $\phi(b) \rightarrow_{\mathcal{M}} \phi(b')$.

PROOF. Since the five-tuple b is in $D(\phi)$, we can distinguish two cases.

- *Case* 1: $b = \langle P, \mathit{control}(t), i : \mathbf{recycle}, \varepsilon_A, \varepsilon_T \rangle$ for some $t \in \mathbb{T}(\Sigma_0)$.
  $\mathit{control}(t) = a \cdot C$ for some constant a. The definition of ϕ yields $\phi(b) = \psi(a \cdot C, \varepsilon_A, \varepsilon_T)$. Let

$$b' = \langle P, C, a : \mathit{get}(a, P), \varepsilon_A, \varepsilon_T \rangle.$$

  By the first state transition rule for **recycle**, $b \to_{\mathcal{S}} b'$. Furthermore, by the definition of ϕ,

$$\phi(b') \;=\; \psi(a \cdot C, \varepsilon_A, \varepsilon_T\rangle \;=\; \phi(b).$$

- *Case* 2: $b = \langle P, C, f : \mathit{get}(f, P), A, T \rangle$ for some $f \in \mathcal{F}$.
  Since $\phi(b)$ is well-defined, $A = t_{L(f)+1} \cdot \ldots \cdot t_{ar(f)} \cdot A'$ and $T = t_{L(f)} \cdot \ldots \cdot t_1 \cdot T'$, and by definition of ϕ,

$$\phi(b) \;=\; \psi(f \cdot C, A, T) \;=\; \mathit{Con}[f(t_1, \ldots, t_{ar(f)})]$$

  where $\mathit{Con}[] = \psi(C, [] \cdot A', T')$. Since the function symbols on C were collected in a rightmost innermost fashion, and since observation A implies that the terms on A and T are normal forms for $\mathcal{M}$, there is no $\mathcal{M}$-redex in $\phi(b)$ rightmost innermost of $f(t_1, \ldots, t_{ar(f)})$. We distinguish two cases, depending on whether there exists a rule for f in $\mathcal{M}$.

  ⋆ *Case* 2.1: There does not exist a rule for f in $\mathcal{M}$. Then

$$\mathit{get}(f, P) = \mathbf{build}(f, ar(f)) \cdot \mathbf{recycle}.$$

  Since L is a locus, Definition 3.2.1 (1) ensures that $L(f) = 0$, so $A = t_1 \cdot \ldots \cdot t_{ar(f)} \cdot A'$ and $T = T'$. By the state transition rule for **build**,

$$b \to_{\mathcal{S}} \langle P, C, \mathbf{recycle}, f(t_1, \ldots, t_{ar(f)}) \cdot A', T' \rangle.$$

  We distinguish two cases, depending on whether $C = \bot$.

  ∘ *Case* 2.1.1: $C = \bot$.
  Then well-definedness of $\phi(b)$ implies $A' = \varepsilon_A$ and $T' = \varepsilon_T$, so that $\mathit{Con}[] = \psi(\bot, [], \epsilon_T) = []$, and so $\phi(b) = f(t_1, \ldots, t_{ar(f)})$. We take $b' = f(t_1, \ldots, t_{ar(f)})$. The third state transition rule for **recycle** yields

$$\langle P, \bot, \mathbf{recycle}, f(t_1, \ldots, t_{ar(f)}), \epsilon_T \rangle \to_{\mathcal{S}} b'.$$

  $b' \in D(\phi)$, because it is a normal form for $\mathcal{M}$. Moreover, $\phi(b') = b' = \phi(b)$.

  ∘ *Case* 2.1.2: $C = g \cdot C'$.
  Then the first state transition rule for **recycle** yields

$$\langle P, g \cdot C', \mathbf{recycle}, f(t_1, \ldots, t_{ar(f)}) \cdot A', T' \rangle \to_{\mathcal{S}} b'$$

  with

$$b' = \langle P, C', g : \mathit{get}(g, P), f(t_1, \ldots, t_{ar(f)}) \cdot A', T' \rangle.$$

  $b \to_{\mathcal{S}}^{+} b'$ implies $b' \in \mathbb{B}$, and so $b' \in D(\phi)$. The definition of ϕ yields

$$\phi(b') \;=\; \psi(g \cdot C', f(t_1, \ldots, t_{ar(f)}) \cdot A', T') \;=\; \phi(b).$$

⋆ *Case* 2.2: There exists a rule for f in $\mathcal{M}$. Then

$$get(f,P) = \mathbf{match}(g_1,h_1)\cdot\ldots\cdot\mathbf{match}(g_n,h_n)\cdot E$$

where E does not contain **match** instructions.

$g_1,\ldots,g_n$ are distinct function symbols, and there are rules in $\mathcal{M}$ of the form

$$f(\vec{x},g_i(\vec{y}),\vec{z}) \to h_i(\vec{x},\vec{y},\vec{z})$$

for $i=1,\ldots,n$. So $ar(h_i) = ar(f)+ar(g_i)-1$, and since $\mathcal{M}$ is stratified we have $L(h_i)=L(f)=|\vec{x}|$ and $L(g_i)=0$. We distinguish two cases, depending on whether the top element of A has outermost function symbol g_i for some $i\in\{1,\ldots,n\}$.

∘ *Case* 2.2.1: $t_{L(f)+1} = g_i(s_1,\ldots,s_{ar(g_i)})$ for some $i\in\{1,\ldots,n\}$.

Then by the first state transition rule for **match**, $b \to_{\mathcal{S}}^* b_0$ where

$$b_0 = \langle P,C,\mathbf{match}(g_i,h_i)\cdot\ldots\cdot\mathbf{match}(g_n,h_n)\cdot E,A,T\rangle$$

and by the second state transition rule for **match**, $b_0 \to_{\mathcal{S}} b'$ where

$$b' = \langle P,C,h_i\!:get(h_i,P),s_1\cdot\ldots\cdot s_{ar(g_i)}\cdot t_{L(f)+2}\cdot\ldots\cdot t_{ar(f)}\cdot A', t_{L(f)}\cdot\ldots\cdot t_1\cdot T'\rangle.$$

$b\to_{\mathcal{S}}^+ b'$ implies $b'\in\mathbb{B}$, and so $b'\in D(\phi)$. Since $ar(h_i)=ar(f)+ar(g_i)-1$ and $L(h_i)=L(f)$, the definition of ϕ together with $Con[] = \psi(C,[]\cdot A',T')$ yield

$$\begin{aligned}\phi(b') &= \psi(h_i\cdot C, s_1\cdot\ldots\cdot s_{ar(g_i)}\cdot t_{L(f)+2}\cdot\ldots\cdot t_{ar(f)}\cdot A', t_{L(f)}\cdot\ldots\cdot t_1\cdot T'\rangle\\ &= \psi(C,h_i(t_1,\ldots,t_{L(f)},s_1,\ldots,s_{ar(g_i)},t_{L(f)+2}\cdot\ldots\cdot t_{ar(f)})\cdot A',T')\\ &= Con[h_i(t_1,\ldots,t_{L(f)},s_1,\ldots,s_{ar(g_i)},t_{L(f)+2}\cdot\ldots\cdot t_{ar(f)})].\end{aligned}$$

Recall that $\phi(b) = Con[f(t_1,\ldots,t_{ar(f)})]$, where $\phi(b)$ does not have an $\mathcal{M}$-redex rightmost innermost of $f(t_1,\ldots,t_{ar(f)})$. So, since $t_{L(f)+1} = g_i(s_1,\ldots,s_{ar(g_i)})$ and $|\vec{x}|=L(f)$, an application of the rule

$$f(\vec{x},g_i(\vec{y}),\vec{z}) \to h_i(\vec{x},\vec{y},\vec{z})$$

in $\mathcal{M}$ yields

$$\phi(b) \to_{\mathcal{M}} Con[h_i(t_1,\ldots,t_{L(f)},s_1,\ldots,s_{ar(g_i)},t_{L(f)+2}\cdot\ldots\cdot t_{ar(f)})] = \phi(b').$$

∘ *Case* 2.2.2: Either A is empty, or its top element is not of the form $g_i(s_1,\ldots,s_{ar(g_i)})$ for $i=1,\ldots,n$.

Then by the second state transition rules for **match**,

$$b\to_{\mathcal{S}}^* \langle P,C,E,A,T\rangle.$$

We distinguish the nine possible forms of E, which are based on the possible forms of the string of executions that belongs to a most general rule of the type M2-5 (see Section 4.4), whereby we use that $\mathcal{M}$ is stratified (see Definition 3.2.2). In each of these cases we show that $\langle P,C,E,A,T\rangle \to_{\mathcal{S}}^+ b'$ with $\phi(b)\to_{\mathcal{M}}\phi(b')$.

• *Case* 2.2.2.1: $E = \mathbf{push}(h)\cdot\mathbf{goto}(g)$.

Then the most general rule for f in $\mathcal{M}$ is of the form

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

with $ar(h)=ar(f)-ar(g)+1$, $L(h)=L(f)=|\vec{x}|$, and $L(g)=0$. Let

$$b' = \langle P,h\cdot C,g\!:get(g,P),A,T\rangle.$$

The state transition rules for **push** and **goto** yield

$$\langle P, C, \mathbf{push}(h) \cdot \mathbf{goto}(g), A, T\rangle \rightarrow_{\mathcal{S}} \langle P, h \cdot C, \mathbf{goto}(g), A, T\rangle \rightarrow_{\mathcal{S}} b'.$$

$b \rightarrow_{\mathcal{S}}^{+} b'$ implies $b' \in \mathbb{B}$, and so $b' \in D(\phi)$. Since $A = t_{L(f)+1} \cdot \ldots \cdot t_{ar(f)} \cdot A'$, $T = t_{L(f)} \cdot \ldots \cdot t_1 \cdot T'$, $L(g) = 0$, $L(h) = L(f)$, $ar(h) = ar(f) - ar(g) + 1$, and $Con[] = \psi(C, [] \cdot A', T')$, the definition of ϕ yields

$$\begin{aligned}
&\phi(b') \\
&= \psi(g \cdot h \cdot C, A, T) \\
&= \psi(h \cdot C, g(t_{L(f)+1}, \ldots, t_{L(f)+ar(g)}) \cdot t_{L(f)+ar(g)+1} \cdot \ldots \cdot t_{ar(f)} \cdot A', T) \\
&= \psi(C, h(t_1, \ldots, t_{L(f)}, g(t_{L(f)+1}, \ldots, t_{L(f)+ar(g)}), t_{L(f)+ar(g)+1}, \ldots, t_{ar(f)}) \cdot A', T') \\
&= Con[h(t_1, \ldots, t_{L(f)}, g(t_{L(f)+1}, \ldots, t_{L(f)+ar(g)}), t_{L(f)+ar(g)+1}, \ldots, t_{ar(f)})].
\end{aligned}$$

Finally, since $|\vec{x}| = L(f)$, an application of the rule

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

in $\mathcal{M}$ with respect to $\phi(b) = Con[f(t_1, \ldots, t_{ar(f)})]$ yields $\phi(b) \rightarrow_{\mathcal{M}} \phi(b')$.

- *Case* 2.2.2.2: $E = \mathbf{copyt}(k) \cdot \mathbf{goto}(h)$.
  Then the most general rule for f in $\mathcal{M}$ is of the form

$$f(\vec{x}, \vec{y}) \rightarrow h(\vec{x}, x_{L(f)-k+1}, \vec{y})$$

with $ar(h) = ar(f) + 1$ and $L(h) = L(f) = |\vec{x}|$. Let

$$b' = \langle P, C, h : get(h, P), t_{L(f)-k+1} \cdot A, T\rangle.$$

Since the kth argument on $T = t_{L(f)} \cdot \ldots \cdot t_1 \cdot T'$ is $t_{L(f)-k+1}$, the state transition rules for **copyt** and **goto** yield

$$\langle P, C, \mathbf{copyt}(k) \cdot \mathbf{goto}(h), A, T\rangle \rightarrow_{\mathcal{S}} \langle P, C, \mathbf{goto}(h), t_{L(f)-k+1} \cdot A, T\rangle \rightarrow_{\mathcal{S}} b'.$$

$b \rightarrow_{\mathcal{S}}^{+} b'$ implies $b' \in \mathbb{B}$, and so $b' \in D(\phi)$. Since $L(h) = L(f)$ and $ar(h) = ar(f)+1$, the definition of ϕ yields

$$\begin{aligned}
\phi(b') &= \psi(h \cdot C, t_{L(f)-k+1} \cdot A, T) \\
&= \psi(C, h(t_1, \ldots, t_{L(f)}, t_{L(f)-k+1}, t_{L(f)+1}, \ldots, t_{ar(f)}) \cdot A', T') \\
&= Con[h(t_1, \ldots, t_{L(f)}, t_{L(f)-k+1}, t_{L(f)+1}, \ldots, t_{ar(f)})].
\end{aligned}$$

Finally, since $|\vec{x}| = L(f)$, an application of the rule

$$f(\vec{x}, \vec{y}) \rightarrow h(\vec{x}, x_{L(f)-k+1}, \vec{y})$$

in $\mathcal{M}$ with respect to $\phi(b) = Con[f(t_1, \ldots, t_{ar(f)})]$ yields $\phi(b) \rightarrow_{\mathcal{M}} \phi(b')$.

- *Case* 2.2.2.3: $E = \mathbf{copya}(k) \cdot \mathbf{goto}(h)$.
  Then the most general rule for f in $\mathcal{M}$ is of the form

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

with $ar(h) = ar(f) + 1$ and $L(h) = L(f) = |\vec{x}|$. Let

$$b' = \langle P, C, h : get(h, P), t_{L(f)+k} \cdot A, T\rangle.$$

Since $t_{L(f)+k}$ is the kth argument on $A = t_{L(f)+1} \cdot \ldots \cdot t_{ar(f)} \cdot A'$, the state transition rules for **copya** and **goto** yield

$$\langle P, C, \mathbf{copya}(k) \cdot \mathbf{goto}(h), A, T\rangle \rightarrow_{\mathcal{S}} \langle P, C, \mathbf{goto}(h), t_{L(f)+k} \cdot A, T\rangle \rightarrow_{\mathcal{S}} b'.$$

$b \to_{\mathcal{S}}^{+} b'$ implies $b' \in \mathbb{B}$, and so $b' \in D(\phi)$. Since $L(h) = L(f)$ and $ar(h) = ar(f)+1$, the definition of ϕ yields

$$\begin{aligned} \phi(b') &= \psi(h \cdot C, t_{L(f)+k} \cdot A, T) \\ &= \psi(C, h(t_1, \ldots, t_{L(f)}, t_{L(f)+k}, t_{L(f)+1}, \ldots, t_{ar(f)}) \cdot A', T') \\ &= Con[h(t_1, \ldots, t_{L(f)}, t_{L(f)+k}, t_{L(f)+1}, \ldots, t_{ar(f)})]. \end{aligned}$$

Finally, since $|\vec{x}| = L(f)$, an application of the rule

$$f(\vec{x}, \vec{y}) \to h(\vec{x}, y_k, \vec{y})$$

in $\mathcal{M}$ with respect to $\phi(b) = Con[f(t_1, \ldots, t_{ar(f)})]$ yields $\phi(b) \to_{\mathcal{M}} \phi(b')$.

- *Case* 2.2.2.4: $E = \mathbf{adrop}(k) \cdot \mathbf{goto}(h)$.
  Then the most general rule for f in $\mathcal{M}$ is of the form

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

with $ar(h) = ar(f) - k$ and $L(h) = L(f) = |\vec{x}|$ and $|\vec{y}| = k$. Let

$$b' = \langle P, C, h\!:\!get(h, P), t_{L(f)+k+1} \cdot \ldots \cdot t_{ar(f)} \cdot A', T \rangle.$$

The state transition rules for **adrop** and **goto** yield

$$\langle P, C, \mathbf{adrop}(k) \cdot \mathbf{goto}(h), A, T \rangle \to_{\mathcal{S}}^{+} b'.$$

$b \to_{\mathcal{S}}^{+} b'$ implies $b' \in \mathbb{B}$, and so $b' \in D(\phi)$. Since $L(h) = L(f)$ and $ar(h) = ar(f)-k$, the definition of ϕ yields

$$\begin{aligned} \phi(b') &= \psi(h \cdot C, t_{L(f)+k+1} \cdot \ldots \cdot t_{ar(f)} \cdot A', T) \\ &= \psi(C, h(t_1, \ldots, t_{L(f)}, t_{L(f)+k+1}, \ldots, t_{ar(f)}) \cdot A', T') \\ &= Con[h(t_1, \ldots, t_{L(f)}, t_{L(f)+k+1}, \ldots, t_{ar(f)})]. \end{aligned}$$

Finally, since $|\vec{x}| = L(f)$ and $|\vec{y}| = k$, an application of the rule

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

in $\mathcal{M}$ with respect to $\phi(b) = Con[f(t_1, \ldots, t_{ar(f)})]$ yields $\phi(b) \to_{\mathcal{M}} \phi(b')$.

- *Case* 2.2.2.5: $E = \mathbf{skip}(k) \cdot \mathbf{goto}(h)$.
  Then the most general rule for f in $\mathcal{M}$ is of the form

$$f(\vec{x}) \to h(\vec{x})$$

with $ar(h) = ar(f)$ and $L(h) = L(f) + k$. Let

$$b' = \langle P, C, h\!:\!get(h, P), t_{L(f)+k+1} \cdot \ldots \cdot t_{ar(f)} \cdot A', t_{L(f)+k} \cdot \ldots \cdot t_1 \cdot T' \rangle.$$

The state transition rules for **skip** and **goto** yield

$$\langle P, C, \mathbf{skip}(k) \cdot \mathbf{goto}(h), A, T \rangle \to_{\mathcal{S}}^{+} b'.$$

$b \to_{\mathcal{S}}^{+} b'$ implies $b' \in \mathbb{B}$, and so $b' \in D(\phi)$. Since $L(h) = L(f)+k$ and $ar(h) = ar(f)$, the definition of ϕ yields

$$\begin{aligned} \phi(b') &= \psi(h \cdot C, t_{L(f)+k+1} \cdot \ldots \cdot t_{ar(f)} \cdot A', t_{L(f)+k} \cdot \ldots \cdot t_1 \cdot T') \\ &= \psi(C, h(t_1, \ldots, t_{ar(f)}) \cdot A', T') \\ &= Con[h(t_1, \ldots, t_{ar(f)})]. \end{aligned}$$

Finally, an application of the rule

$$f(\vec{x}\,) \to h(\vec{x}\,)$$

in $\mathcal{M}$ with respect to $\phi(b) = Con[f(t_1, \ldots, t_{ar(f)})]$ yields $\phi(b) \to_{\mathcal{M}} \phi(b')$.

- *Case* 2.2.2.6: $E = \mathbf{goto}(h)$.
  Then the most general rule for f in $\mathcal{M}$ is of the form

$$f(\vec{x}\,) \to h(\vec{x}\,)$$

with $ar(h) = ar(f)$ and $L(h) = L(f)$. Let

$$b' = \langle P, C, h : get(h, P), A, T\rangle.$$

The state transition rule for **goto** yields

$$\langle P, C, \mathbf{goto}(h), A, T\rangle \to_{\mathcal{S}} b'.$$

$b \to_{\mathcal{S}}^{+} b'$ implies $b' \in \mathbb{B}$, and so $b' \in D(\phi)$. Since $L(h) = L(f)$ and $ar(h) = ar(f)$, the definition of ϕ yields

$$\begin{aligned} \phi(b') &= \psi(h \cdot C, A, T) \\ &= \psi(C, h(t_1, \ldots, t_{ar(f)}) \cdot A', T') \\ &= Con[h(t_1, \ldots, t_{ar(f)})]. \end{aligned}$$

Finally, an application of the rule

$$f(\vec{x}\,) \to h(\vec{x}\,)$$

in $\mathcal{M}$ with respect to $\phi(b) = Con[f(t_1, \ldots, t_{ar(f)})]$ yields $\phi(b) \to_{\mathcal{M}} \phi(b')$.

- *Case* 2.2.2.7: $E = \mathbf{retract}(k) \cdot \mathbf{goto}(h)$.
  Then the most general rule for f in $\mathcal{M}$ is of the form

$$f(\vec{x}\,) \to h(\vec{x}\,)$$

with $ar(h) = ar(f)$ and $L(h) = L(f) - k$. Let

$$b' = \langle P, C, h : get(h, P), t_{L(f)-k+1} \cdot \ldots \cdot t_{ar(f)} \cdot A', t_{L(f)-k} \cdot \ldots \cdot t_1 \cdot T'\rangle.$$

The state transition rules for **retract** and **goto** yield

$$\langle P, C, \mathbf{retract}(k) \cdot \mathbf{goto}(h), A, T\rangle \to_{\mathcal{S}}^{+} b'.$$

$b \to_{\mathcal{S}}^{+} b'$ implies $b' \in \mathbb{B}$, and so $b' \in D(\phi)$. Since $L(h) = L(f) - k$ and $ar(h) = ar(f)$, the definition of ϕ yields

$$\begin{aligned} \phi(b') &= \psi(h \cdot C, t_{L(f)-k+1} \cdot \ldots \cdot t_{ar(f)} \cdot A', t_{L(f)-k} \cdot \ldots \cdot t_1 \cdot T') \\ &= \psi(C, h(t_1, \ldots, t_{ar(f)}) \cdot A', T') \\ &= Con[h(t_1, \ldots, t_{ar(f)})]. \end{aligned}$$

Finally, an application of the rule

$$f(\vec{x}\,) \to h(\vec{x}\,)$$

in $\mathcal{M}$ with respect to $\phi(b) = Con[f(t_1, \ldots, t_{ar(f)})]$ yields $\phi(b) \to_{\mathcal{M}} \phi(b')$.

- *Case* 2.2.2.8: $E = \mathbf{tdrop}(k) \cdot \mathbf{recycle}$.
  Then the most general rule for f in $\mathcal{M}$ is of the form
  $$f(\vec{x}, y) \to y$$
  with $ar(f) = k+1$ and $L(f) = |\vec{x}| = k$. We distinguish two cases, depending on whether $C = \perp$.

◇ *Case* 2.2.2.8.1: $C = \perp$.
As in Case 2.1.1, well-definedness of $\phi(b)$ implies $A' = \varepsilon_A$ and $T' = \varepsilon_T$, so that $Con[] = \psi(\perp, [], \epsilon_T) = []$, and so $\phi(b) = f(t_1, \ldots, t_{k+1})$.
We take $b' = t_{k+1}$. According to observation A, t_{k+1} is a normal form for $\mathcal{M}$, so $t_{k+1} \in D(\phi)$. The state transition rule for **tdrop** and the third state transition rule for **recycle** yield
$$\langle P, \perp, \mathbf{tdrop}(k) \cdot \mathbf{recycle}, t_{k+1}, t_k \cdot \ldots \cdot t_1 \rangle \to_{\mathcal{S}} \langle P, \perp, \mathbf{recycle}, t_{k+1}, \varepsilon_T \rangle \to_{\mathcal{S}} b'$$
and so $b \to_{\mathcal{S}}^{+} b'$. Since $\phi(b') = t_{k+1}$ and $|\vec{x}| = k$, an application of the rule
$$f(\vec{x}, y) \to y$$
in $\mathcal{M}$ with respect to $\phi(b) = f(t_1, \ldots, t_{k+1})$ yields $\phi(b) \to_{\mathcal{M}} \phi(b')$.

◇ *Case* 2.2.2.8.2: $C = h' \cdot C'$.
Let
$$b' = \langle P, C', get(h', P), t_{k+1} \cdot A', T' \rangle.$$
The state transition rule for **tdrop** and the first state transition rule for **recycle** yield
$$\begin{array}{ll} & \langle P, h' \cdot C', \mathbf{tdrop}(k) \cdot \mathbf{recycle}, t_{k+1} \cdot A', t_k \cdot \ldots \cdot t_1 \cdot T' \rangle \\ \to_{\mathcal{S}} & \langle P, h' \cdot C', \mathbf{recycle}, t_{k+1} \cdot A', T' \rangle \\ \to_{\mathcal{S}} & b'. \end{array}$$
$b \to_{\mathcal{S}}^{+} b'$ implies $b' \in \mathbb{B}$, and so $b' \in D(\phi)$. Furthermore, by the definition of ϕ,
$$\phi(b') \;=\; \psi(h' \cdot C', t_{k+1} \cdot A', T') \;=\; Con[t_{k+1}].$$
Since $|\vec{x}| = k$, an application of the rule
$$f(\vec{x}, y) \to y$$
in $\mathcal{M}$ with respect to $\phi(b) = Con[f(t_1, \ldots, t_{k+1})]$ yields $\phi(b) \to_{\mathcal{M}} \phi(b')$.

- *Case* 2.2.2.9: $E = \mathbf{recycle}$.
  Then the most general rule for f in $\mathcal{M}$ is of the form
  $$f(y) \to y$$
  with $ar(f) = 1$ and $L(f) = 0$. We distinguish two cases, depending on whether $C = \perp$.

◇ *Case* 2.2.2.9.1: $C = \perp$.
Well-definedness of $\phi(b)$ implies $A' = \varepsilon_A$, $T' = \varepsilon_T$, and $Con[] = \psi(\perp, [], \epsilon_T) = []$, so $\phi(b) = f(t_1)$.
We take $b' = t_1$. According to observation A, t_1 is a normal form for $\mathcal{M}$, so $b' \in D(\phi)$. The third state transition rule for **recycle** yields
$$\langle P, \perp, \mathbf{recycle}, t_1, \varepsilon_T \rangle \to_{\mathcal{S}} b'$$

and so $b \to_{\mathcal{S}}^{+} b'$. Since $\phi(b') = t_1$, an application of the rule

$$f(y) \to y$$

in $\mathcal{M}$ with respect to $\phi(b) = f(t_1)$ yields $\phi(b) \to_{\mathcal{M}} \phi(b')$.

◇ *Case* 2.2.2.9.2: $C = h' \cdot C'$.

Let

$$b' = \langle P, C', get(h', P), t_1 \cdot A', T' \rangle.$$

The first state transition rule for **recycle** yields

$$\langle P, h' \cdot C', \mathbf{recycle}, t_1 \cdot A', T' \rangle \to_{\mathcal{S}} b'.$$

$b \to_{\mathcal{S}}^{+} b'$ implies $b' \in \mathbb{B}$, and so $b' \in D(\phi)$. According to the definition of ϕ,

$$\phi(b') \;=\; \psi(h' \cdot C', t_1 \cdot A', T') \;=\; Con[t_1].$$

Finally, an application of the rule

$$f(y) \to y$$

in $\mathcal{M}$ with respect to $\phi(b) = Con[f(t_1)]$ yields $\phi(b) \to_{\mathcal{M}} \phi(b')$. □

This finishes the proof of observation C, which we recall below.

C. *If $b \in D(\phi)$ is a five-tuple with $\phi(b)$ well-defined, then $b \to_{\mathcal{S}}^{+} b'$ with $b' \in D(\phi)$ and $\phi(b')$ well-defined, and either $\phi(b) = \phi(b')$ or $\phi(b) \to_{\mathcal{M}} \phi(b')$.*

It is easy to see, in the proof of observation C, that the b' was selected such that if a derivation $b \to_{\mathcal{S}}^{+} b''$ is shorter than $b \to_{\mathcal{S}}^{+} b'$, then $b'' \notin D(\phi)$.

We show how observation C implies that ϕ is well-defined, complete, sound, termination preserving, and surjective.

Well-Definedness: If $b' \in D(\phi)$, then $\phi(b') \in \mathbb{A}$.

Proof. We show that $\phi(b') \in \mathbb{A}$ by induction with respect a shortest derivation $\langle P, control(t), i : \mathbf{recycle}, \varepsilon_A, \varepsilon_T \rangle \to_{\mathcal{S}}^{*} b'$ for some $t \in \mathbb{T}(\Sigma_0)$.

— In the base case, where the length of this derivation is 0, observation B implies that $\phi(b') = \langle P, control(t), i : \mathbf{recycle}, \varepsilon_A, \varepsilon_T \rangle = t \in \mathbb{T}(\Sigma_0) \subseteq \mathbb{A}$.

— In the inductive case, there exists a five-tuple $b \in D(\phi)$ with $\phi(b) \in \mathbb{A}$ and $b \to_{\mathcal{S}}^{+} b'$. Moreover, we can select b such that for all derivations $b \to_{\mathcal{S}}^{+} b''$ that are shorter than $b \to_{\mathcal{S}}^{+} b'$ we have $b'' \notin D(\phi)$. Then observation C implies either $\phi(b) = \phi(b')$ or $\phi(b) \to_{\mathcal{M}} \phi(b')$, so $\phi(b') \in \mathbb{A}$.

□

Completeness: If $b \in \mathbb{B}$ is a normal form for $\mathcal{S}$, then $b \in D(\phi)$ and $\phi(b)$ is a normal form for $\mathcal{M}$.

Proof. Observation C implies that b is not a five-tuple, so b is a normal form for $\mathcal{M}$. Then $b \in D(\phi)$ and $\phi(b) = b$ is a normal form for $\mathcal{M}$. □

Soundness: If $b \in D(\phi)$ with $b \to_{\mathcal{S}} b'$, then $b' \to_{\mathcal{S}}^{*} b''$ with $b'' \in D(\phi)$ and either $\phi(b) = \phi(b'')$ or $\phi(b) \to_{\mathcal{M}} \phi(b'')$.

PROOF. b is a five-tuple, so soundness follows immediately from observation C. □

TERMINATION PRESERVATION: If $b_0 \in D(\phi)$ allows an infinite $\mathcal{S}$-reduction $b_0 \to_{\mathcal{S}} b_1 \to_{\mathcal{S}} b_2 \to_{\mathcal{S}} \cdots$, then there is a $k \geq 1$ such that $b_k \in D(\phi)$ and $\phi(b_0) \to_{\mathcal{M}} \phi(b_k)$.

PROOF. Clearly b_0 is a five-tuple. If b_0 is an initial five-tuple, then according to Case 1 of observation C, $b_1 \in D(\phi)$ is a non-initial five-tuple with $\phi(b_0) = \phi(b_1)$.

Consider the non-initial five-tuple b_0 or b_1. The control stack of b_0 or b_1 is finite, and each application of an executable stack $\mathbf{build}(f) \cdot \mathbf{recycle}$ decreases the size of the control stack by one, so there can only be a finite number of applications of such executable stacks $\mathbf{build}(f) \cdot \mathbf{recycle}$ in a row, with respect to b_0 or b_1. Hence, there exists a smallest $\ell \geq 1$ such that $b_\ell \in D(\phi)$ is a non-initial five-tuple, and the executable stack of b_ℓ is *not* of the form $\mathbf{build}(f) \cdot \mathbf{recycle}$.

— $b_{i-1} \to_{\mathcal{S}} b_i \to_{\mathcal{S}} b_{i+1}$ for $i = 1, 3, \ldots, \ell - 1$ (or $i = 2, 4, \ldots, \ell - 1$, if b_0 is initial) is the result of applications of the state transition rule for **build** and the first state transition rule for **recycle**. Hence, we saw in Case 2.1 of the proof of observation C that $\phi(b_{i-1}) = \phi(b_{i+1})$.

— $b_\ell \in D(\phi)$ is non-initial, and the executable stack of b_ℓ is *not* of the form $\mathbf{build}(f) \cdot \mathbf{recycle}$. Hence, we saw in the Cases 2.2-2.9 in the proof of observation C that there is a $k > \ell$ with $b_k \in D(\phi)$ and $\phi(b_\ell) \to_{\mathcal{M}} \phi(b_k)$.

So $\phi(b_0) = \cdots = \phi(b_\ell) \to_{\mathcal{M}} \phi(b_k)$. □

SURJECTIVITY: If $t \in \mathbb{A}$, then there is a $b \in D(\phi)$ with $\phi(b) = t$.

PROOF. By induction on the length of a shortest derivation $s \to_{\mathcal{M}}^* t$ with $s \in \mathbb{T}(\Sigma_0)$.

— In the base case, where the length of this derivation is 0, we have $t = s \in \mathbb{T}(\Sigma_0)$. Observation B yields $\phi(\langle P, \mathit{control}(t), i : \mathbf{recycle}, \varepsilon_A, \varepsilon_T \rangle) = t$.

— In the inductive case, there exists a $t_0 \in \mathbb{A}$ such that $t_0 \to_{\mathcal{M}} t$, and there is a $b_0 \in D(\phi)$ with $\phi(b_0) = t_0$. Assume, toward a contradiction, that $t \notin \phi(D(\phi))$. Since t_0 is not a normal form for $\mathcal{M}$, completeness of ϕ yields that b_0 is not a normal form for $\mathcal{S}$. Then soundness yields $b_0 \to_{\mathcal{S}}^+ b_1 \in D(\phi)$ with either $\phi(b_1) = t_0$ or $\phi(b_1) = t$. The assumption $t \notin \phi(D(\phi))$ enforces that $\phi(b_1) = t_0$. Again, since t_0 is not a normal form for $\mathcal{M}$, completeness yields that b_1 is not a normal form for $\mathcal{S}$. Then soundness yields $b_1 \to_{\mathcal{S}}^+ b_2 \in D(\phi)$ with $\phi(b_2) = t_0$ or $\phi(b_2) = t$. The assumption $t \notin \phi(D(\phi))$ enforces that $\phi(b_2) = t_0$, etcetera. Thus, we construct an infinite reduction $b_0 \to_{\mathcal{S}}^+ b_1 \to_{\mathcal{S}}^+ b_2 \to_{\mathcal{S}}^+ \cdots$. Termination preservation implies that there is a $b \in D(\phi)$ in this reduction with $\phi(b_0) \to_{\mathcal{M}} \phi(b)$, and so $\phi(b) = t$. This contradicts the assumption $t \notin \phi(D(\phi))$. Hence, we conclude that $t \in \phi(D(\phi))$.

□

A.7 Conclusion

We have shown that the separate transformation steps that make up the compilation of a left-linear TRS into an ARM program are all correct. Hence, we conclude that the compilation technique is correct, in the following sense. Let $\mathcal{R}$ denote the

original left-linear TRS, and let P denote the resulting ARM program. Furthermore, let t be a term over the original signature; the function symbols in t all have locus 0. $\mathcal{R}$ reduces t to a normal form s, by means of rightmost innermost rewriting with specificity ordering, if and only if P, together with the state transition rules for ARM, reduce $\langle P, \mathit{control}(t), \mathbf{recycle}, \varepsilon_A, \varepsilon_T \rangle$ to a normal form s'. In this case, s' can be transformed into s by renaming function symbols f^c, which have been introduced in the procedure "Add Most General Rules", into f.