Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

Minimal term rewriting systems

H.R. Walters and J.F.Th. Kamperman

Computer Science/Department of Software Technology

CS-R9573 1995

Report CS-R9573
ISSN 0169-118X

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

Minimal Term Rewriting Systems

H.R.Walters (pum@cwi.nl)
J.F.Th.Kamperman (jasper@cwi.nl)

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

Abstract

Formally well-founded compilation techniques for Term Rewriting Systems (TRSs) are presented. TRSs are compiled into *Minimal Term Rewriting Systems* (MTRSs), a subclass of TRSs in which all rules have an extremely simple form. A notion of simulation of (rewrite) relations is presented, under which an MTRSs can be said to simulate a TRS. The MTRS rules can be directly interpreted as instructions for an extremely simple Abstract Rewriting Machine (ARM). Favourable practical results have already been obtained with an earlier version of ARM.

CR Subject Classification (1991): D.3.4 [Programming languages]: Processors – Compilers; D.1.1 [Programming Techniques]: Applicative (Functional) Programming; D.1.6: Logic Programming.

AMS Subject Classification (1991): 68N20: Compilers and generators; 68Q05: Models of Computation; 68Q42: Rewriting Systems; 68Q65: Abstract data types; algebraic specification.

Keywords & Phrases: minimal term rewriting systems, program transformation.

Note: This report is under formal review for the Conference Proceedings of the 11th Workshop on Abstract Data Types. Partial support was received from the Foundation for Computer Science Research in the Netherlands (SION) under project 612-17-418, "Generic Tools for Program Analysis and Optimization", and from the IBM SP/1 project of department AP.

1. Introduction

Term (graph) rewriting systems (TRSs) are becoming increasingly important for the implementation of theorem provers, algebraic specifications, compiler generators, program analyzers and functional programming languages. Hence, a clear need arises for techniques enabling fast execution of TRSs. Furthermore, these techniques should be flexible with regard to extensions, such as selection of reduction strategy.

A standard technique for speeding up the execution of a program in a formal (programming) language is *compilation* into the language of a concrete machine (e.g., a microprocessor). In compiler construction (c.f. [ASU86]), it is customary to use an *abstract machine* as abstraction of the concrete machine. On the one hand, this allows hiding details of the concrete machine in a small part of the compiler, and thus an easy reimplementation on other concrete machines. On the other hand, a good design of the abstract machine enables a simple mapping from source language into abstract machine language.

A compiler consists of zero or more mappings from its source language into a restricted version of the source language, followed by a mapping to a lower-level language. This is repeated until the level of the concrete machine is reached. Because they take place in one domain, the source-to-source mappings are easier to grasp semantically than the mappings to lower levels. In this paper, we present a compilation technique for TRSs which stays entirely within the well-known source language domain; the mapping to the concrete machine level is a trivial interpretation.

We map TRSs to *Minimal Term Rewriting Systems* (MTRSs), a restriction of TRSs, and we interpret the MTRSs directly as programs for our Abstract Rewriting Machine (ARM). An example may clarify this. The TRS defining successor-zero naturals on the left side is compiled into the MTRS on the right side:

$$
\begin{array}{rcl}
plus(zero, X) & \to & X \\
plus(succ(X), Y) & \to & succ(plus(X, Y))
\end{array}
\quad \Longrightarrow \quad
\begin{array}{rcl}
zero & \to & r(zero_c) \\
succ(X') & \to & r(succ_c(X')) \\
plus(X', X'') & \to & plus^S(X', X'') \\
plus(zero_c, Z') & \to & plus\_zero(Z') \\
plus(succ_c(Y'), Z') & \to & plus\_succ(Y', Z') \\
plus\_zero(Y') & \to & Y' \\
plus\_succ(Y', Y'') & \to & succ(plus(Y', Y'')) \\
plus^S(X', X'') & \to & r(plus_c(X', X'')) \\
r(X) & \to & X
\end{array}
$$

It is easily verified that rewriting the term $plus(succ(zero), succ(zero))$ in the original system yields $succ(succ(zero))$, and rewriting in the transformed system yields $succ_c(succ_c(zero_c))$. The latter normal form can be said to *simulate* the former by assuming a *simulation map* $\mathcal{S}$ defined as $\mathcal{S}(zero_c) = zero$ and $\mathcal{S}(succ_c(X)) = succ(\mathcal{S}(X))$.

By a slight change of perspective, the MTRS above can be interpreted as a program for ARM (the resemblance to assembly code is intended):

$$
\begin{array}{ll}
zero: & \mathbf{build}(zero_c, 0); \mathbf{goto}(r) \\
succ: & \mathbf{build}(succ_c, 1); \mathbf{goto}(r) \\
plus: & \mathbf{match}(zero_c, plus\_zero); \\
 & \mathbf{match}(succ_c, plus\_succ); \\
 & \mathbf{goto}(plus^S); \\
plus\_zero: & \mathbf{recycle}; \\
plus\_succ: & \mathbf{cpush}(succ); \mathbf{goto}(plus); \\
plus^S: & \mathbf{build}(plus_c, 2); \mathbf{goto}(r); \\
r: & \mathbf{recycle},
\end{array}
$$

where the instructions are either available on common concrete machines (**goto** is always available, **recycle** corresponds to **return**, and **match** to **compare**) or can be implemented in a few instructions (**build** and **cpush**). With a precursor of ARM, we have reached favourable results for TRSs of real-world size [HF+96].

The remainder of this paper is structured as follows. First, we review basic TRS theory in Section 1.1. Then, in Section 2, we present a notion of simulation of a TRS by another TRS.

After that, in Section 3, we present MTRSs, and in Section 4, we indicate how MTRSs can simulate arbitrary TRSs. Finally, in Section 5, we show how the rules of an MTRS can be interpreted in a straightforward way as instructions for an efficient abstract machine.

1.1 Term Rewriting

We follow [Klo92]. A *signature* Σ consists of:

- A countably infinite set $\mathcal{V}$ of *variables*: $x, y, \ldots$
- A non-empty set $\mathcal{F}$ of *function symbols*: $f, g, \ldots$, each with an *arity* (≥ 0), which is the number of arguments the function requires. We denote the arity of f by $|f|$.

The set $T(\Sigma)$ of terms over Σ is the smallest set satisfying

- $\mathcal{V} \subset T(\Sigma)$,
- for all $f \in \mathcal{F}$ with arity n, and $t_1, \ldots t_n \in T(\Sigma)$, we have $f(t_1, \ldots, t_n) \in T(\Sigma)$.

Occasionally, we will abbreviate a sequence $t_1, \ldots, t_n$ to $\vec{t}$, and write $|\vec{t}|$ for n. We generalize this to empty sequences, which have $|\vec{t}| = 0$.

A *context* is a 'term' containing one occurrence of a special symbol $\Box$, denoting an empty place. A context is generally denoted by $C[]$. If $t \in T(\Sigma)$ and t is substituted for $\Box$, the result is $C[t] \in T(\Sigma)$ and t is said to be a subterm of $C[t]$, notated as $C[t] \subseteq t$.

A *substitution* is a (total) map $\sigma : T(\Sigma) \mapsto T(\Sigma)$ satisfying

$$\forall f \in \mathcal{F} \; : \; \sigma(f(t_1, \ldots, t_n)) = f(\sigma(t_1), \ldots, \sigma(t_n)).$$

By convention, we often write t^σ for $\sigma(t)$.

A *rewrite rule* is a pair of terms written as $s \to t$ with $s, t \in T(\Sigma)$. It is assumed that the left-hand side s of a rule $s \to t$ is not a single variable, and that $var(t) \subseteq var(s)$.

A *term rewriting system* $\mathcal{R}$ consists of a signature Σ and a set of rewrite rules R over Σ.

A term rewriting system defines a *rewrite relation* $\to_{\mathcal{R}}$. Since the subscript $\mathcal{R}$ is usually clear from the context, it is omitted. The overloading of $\to$ is by convention.

$$s \to t \overset{\text{def}}{\Longleftrightarrow} \exists C[], \sigma, u \to v \in R : s = C[u^\sigma] \land t = C[v^\sigma]$$

The sub-term u^σ is referred to as *redex* (for reducible expression); the sub-term v^σ, as *reduct*. We write $\overset{*}{\to}$ for the transitive reflexive closure of $\to$.

The rewrite relation is closed under taking contexts, i.e., if $s \to t$, then for all $C[]$, $C[s] \to C[t]$.

A series of terms $s = s_1, s_2, \ldots$ such that $s_1 \to s_2 \to \ldots$ is called a *rewrite sequence*. A term s is said to be in *normal form* if there is no t such that $s \to t$. A function-symbol f is called a *defined* function symbol if there is a rule $f(t_1, \ldots, t_n) \to r$. A function-symbol c is called a *constructor* symbol if there is a normal form in which it occurs, and a *free constructor* if it is not a defined symbol.

A TRS is called *left-linear* if all left-hand sides are linear. A TRS is called *confluent* if, for all terms t_1, t_2, t_3, we have that $t_1 \xrightarrow{*} t_2$ and $t_1 \xrightarrow{*} t_3$ implies that there exists a term t_4 such that $t_2 \xrightarrow{*} t_4$ and $t_3 \xrightarrow{*} t_4$. A TRS is called *terminating* if there are no infinite rewrite sequences. Note that confluence and termination are generally undecidable.

In general, a term may contain many redexes. A rewriting *strategy* determines which of these is chosen. Confluence guarantees unique normal forms, regardless of the strategy. Some well-known strategies are *leftmost innermost*, *leftmost outermost*, *rightmost innermost*, *rightmost outermost*, *parallel innermost* and *parallel outermost*. For lack of space, we only consider the *rightmost innermost* strategy in this paper, which allows only rewriting of the rightmost redex that does not contain other redexes.

In *priority* rewrite systems (PRSs) [BBKW89], the rules are (partially) ordered, and a rule may be applied only if there are no applicable rules (i.e., even after reduction of subterms) with higher priority. PRSs are very expressive, but their operational semantics can be problematic. For our purposes, a weaker notion suffices, which we will call *syntactic* priority. In a TRS with syntactic priority, the decision whether a rule is applicable is made without considering reductions of sub-terms.

The ordering we will use is *syntactic specificity ordering*, where a rule $l \to r$ is called *more specific* than a rule $s \to t$, when there exists a substitition σ such that $s^\sigma = l$.

Under syntactic specificity ordering, any set of terms has a greatest lower bound (glb). We will call the glb of all terms with top-symbol f, a term of the form $f(\vec{x})$, a *most general* LHS. We will call two terms s, t (or rules with LHSs s, t) *mutually exclusive*, if they have no upper bound, i.e. if there is no term u with $u > s \wedge u > t$. We will call a rule r *maximal* if there is no rule s with $s > r$.

2. Term Rewriting Simulations

In this section, we define the notion of *simulation* of a TRS by another TRS.

In principle, a TRS $T = (\Sigma, R)$ is simulated by a TRS $T' = (\Sigma', R')$ if every rewrite sequences w.r.t. R can be related to a rewrite sequence w.r.t. R'. To this end, there must be a map from $T(\Sigma')$ to $T(\Sigma)$, which is called the simulation map.

This notion of simulation can be developed for arbitrary relations, but we will only use it in the more limited context of (minimal) term rewriting systems. In that context, as we will see, it is preferable to regard a simulating TRS of which the signature is an extension of that of the simulated TRS (i.e., $\Sigma' \supseteq \Sigma$), and for which the simulation map is identity on the common set of terms $T(\Sigma)$.

2.1 Simulation maps between terms

Let $\Sigma = (\mathcal{F}, \mathcal{V})$ and $\Sigma' = (\mathcal{F}', \mathcal{V}')$ be signatures, such that $\Sigma' \supseteq \Sigma$, and let $\mathcal{S} : \mathcal{F}' \to \mathcal{F}$ be a (partial) map, which has the following properties:

- Symbols in the original signature simulate themselves ($\forall f \in \mathcal{F} : \mathcal{S}(f) = f$).

- $\mathcal{S}$ may be partial, and we assume the existence of a predicate $\mathcal{D}_{\mathcal{S}}$, which holds for all symbols in $\mathcal{F}'$ for which $\mathcal{S}$ is defined, because a simulating TRS may use intermediate symbols (terms) which are not a simulation of any symbol (term) in $\mathcal{F}$.

We extend $\mathcal{S}$ and $\mathcal{D}_{\mathcal{S}}$ to $T(\Sigma')$ by (partial) homomorphic extension.

As an example, consider $\mathcal{F} = \{f, a\}$ and $\mathcal{F}' = \{f, a, f_c, h\}$. In this example, f_c is a variant (a so-called constructor variant, discussed further in the sequel) of f with $\mathcal{S}(f_c) = f$, and h is an auxiliary function that has no counterpart in $\mathcal{F}$. Supposing that the arity of f is 1, and the arity of a is 0, we have (by partial homomorphic extension) that $\mathcal{S}(f(f_c(a))) = f(f(x))$, and $\mathcal{S}(f(h(a)))$ is undefined.

2.2 Simulating Relations

Using simulation maps, we will now define simulations of relations over terms. A simulation of a relation R is defined by a pair $(\mathcal{S}, R')$.

A simulation should be both *sound* and *complete*, i.e., it should simulate neither too much nor too little. The definition of these notions is somewhat complicated by the fact that $\mathcal{S}$ is partial. We define a *simulating sequence* to be a sequence $s_1 \rightarrow_{R'} s_2 \rightarrow_{R'} \ldots$ for which $\mathcal{S}$ is defined on s_1, and we call the first step of such a sequence a *simulating step*. In the figures illustrating the definitions below, dashed arrows are implied by solid arrows, closed points are universally quantified, and open points are existentially quantified.

First we consider soundness. If we have a simulating sequence sR'^*t with $\mathcal{S}$ defined on t, it is only reasonable to call such a sequence *sound* when $\mathcal{S}(s)R^*\mathcal{S}(t)$, so the image of R'^* under $\mathcal{S}$ is contained in R^* (depicted in Fig. 1a). In case $\mathcal{S}$ is *not* defined, we do not want the sequence to 'escape into undefinedness', so we demand that there is some u with tR'^*u and $\mathcal{S}$ defined on u (depicted in Fig. 1b). Formally, soundness is defined in Definition 1.

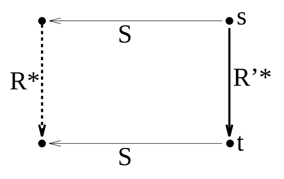


Fig. 1a.

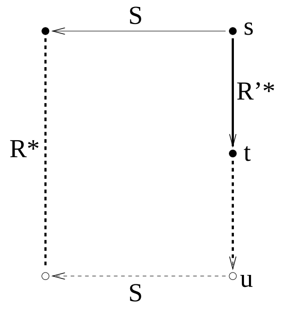
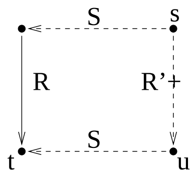


Fig. 1b.

Definition 1 *A simulation* $(\mathcal{S}, R')$ *of* R *is* sound *whenever*

$$\forall st\ (\mathcal{D}_{\mathcal{S}}(s) \wedge sR'^*t) \Longrightarrow \mathcal{S}(s)R^*\mathcal{S}(t) \vee (\neg\mathcal{D}_{\mathcal{S}}(t) \wedge \exists u \mathcal{D}_{\mathcal{S}}(u) \wedge tR'^*u)$$

A simulation is *complete*, when every step sRt in the simulated relation has as counterpart a simulating sequence sR'^+u, where $\mathcal{S}(u) = t$. This is defined formally in Definition 2, and depicted in Fig. 2.

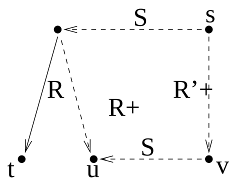


Definition 2 *A simulation* $(\mathcal{S}, R')$ *of a relation* R *is* complete *whenever*

$$\forall st\ \mathcal{D}_{\mathcal{S}}(s) \wedge \mathcal{S}(s)Rt \Longrightarrow \exists u\ sR'^{+}u \wedge \mathcal{S}(u) = t$$

Fig. 2. Completeness

For term rewriting, however, this is a rather rigid notion of completeness, because it requires that the simulation mimicks every single step in the simulated relation. If we are mainly interested in simulating the computation of normal forms, and the simulated relation is confluent, a weaker property suffices. A simulation is *weakly complete* when every step in the simulated system *does* correspond to a simulating sequence, but the endpoints of the step and the image of the simulating sequence need not agree. This is defined formally in definition 3, and depicted in Fig. 3.



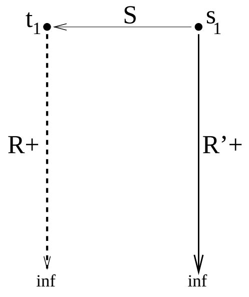
Definition 3 *A simulation* $(\mathcal{S}, R')$ *of a relation* R *is* weakly complete *whenever*

$$\forall st\ \mathcal{D}_{\mathcal{S}}(s) \wedge \mathcal{S}(s)Rt \Longrightarrow \exists u\ \mathcal{D}_{\mathcal{S}}(u) \wedge sR'^{+}u \wedge \mathcal{S}(s)R^{+}\mathcal{S}(u)$$

Fig. 3. Weak completeness

A simulation that is both sound and (weakly) complete need not conserve the termination behaviour, because there may be (cyclic) sequences in the simulating relation corresponding to zero steps in the simulated relation.

A simulation is *termination conserving* when only terms that take part in infinite sequences in the simulated system, have origins occurring in infinite sequences in the simulating system. This is defined by Definition 4, and illustrated in Fig. 4.



Definition 4 *A simulation* $(\mathcal{S}, R')$ *is* termination preserving *whenever*

$$\forall s \in \mathit{inf}(R')\ \mathcal{D}_{\mathcal{S}}(s_1) \Longrightarrow \exists t \in \mathit{inf}(R)\ \mathcal{S}(s_1) = t_1$$

where $\mathit{inf}(R)$ *is the set of* infinite *sequences in* R*, and we denote the ith term in a rewrite sequence* s *by* s_i*.*

Fig. 4. Conservation of termination

3. Minimal Term Rewriting Systems

In this section, we present *minimal term rewriting systems* (MTRSs), a syntactic restriction of TRSs that can be interpreted as the language of an abstract machine. By virtue of being a syntactic restriction, MTRSs inherit syntax and semantics of TRSs.

In MTRSs, all rules have an extremely simple form. The most conspicuous aspect is that any rule has at most three function symbols, of which at most two are found on either side. Even the SKI calculus ([Klo92]), which is minimal in the number of rules (3), and in the total number of function symbols (4: S, K, I, and $\cdot$), needs 7 function symbols in its most complicated rule ($S \cdot x \cdot y \cdot z \to (x \cdot y) \cdot (y \cdot z)$).

In order to simulate general TRSs, MTRSs must be able to express at least the basic actions of composing (building) a term from a function symbol and a sequence of terms, decomposing (matching) a term into a function symbol and a sequence of terms, duplicating some subterm, and deleting some subterm. From these basic assumptions, we arrive at a set of six forms, displayed in Fig. 5.

$$
\begin{array}{rrcll}
\mathbf{C}: & f(\vec{x}, \vec{y}, \vec{z}) & \to & h(\vec{x}, g(\vec{y}), \vec{z}) & \\
\mathbf{R}: & f(y) & \to & y & \\
\mathbf{M}: & f(\vec{x}, g(\vec{y}), \vec{z}) & \to & h(\vec{x}, \vec{y}, \vec{z}) & \\
\mathbf{A}: & f(\vec{x}, \vec{z}) & \to & h(\vec{x}, y, \vec{z}) & (y \text{ is } x_i \text{ or } z_i) \\
\mathbf{D}: & f(\vec{x}, \vec{y}, \vec{z}) & \to & h(\vec{x}, \vec{z}) & (|\,\vec{y}\,| \neq 0) \\
\mathbf{I}: & f(\vec{x}) & \to & h(\vec{x}) &
\end{array}
$$

Fig. 5. Forms of MTRS rules.

We have labeled the forms with mnemonics reminding of their basic purpose (in the context of innermost rewriting). The mnemonic **C** stands for *continuation*, in the sense that h is the continuation after the evaluation of g. Conversely, **R** stands for *return*, in the sense that control is passed to a continuation that was issued earlier, or rewriting is finished if there is no such continuation. Rules of the form **M** take apart a term, when there is a *match* of the symbol g. The forms **A**, **D** and **I** are for *addition, deletion* and *identity* on the set of variables.

Both under innermost and outermost rewriting, all forms have an independent purpose. Here, we discuss only innermost rewriting. The forms **C** and **A** have the independent purposes of introducing a new function and a new variable, respectively. When the form **M** applies, the function g is necessarily a constructor function. The form **R** removes a defined function. Therefore, forms **R** and **M** are the inverse of **C** for a defined function and a constructor function, respectively. In a similar sense, **D** is the inverse of **A**.

4. How to Obtain Simulating MTRSs

In [KW], an executable specification is presented of the translation of an arbitrary TRS into an MTRS that simulates the TRS under innermost rewriting. Furthermore, there are transformations for the simulation of outermost and lazy rewriting, given innermost rewriting with specificity ordering. Here we explain the idea underlying the transformation from TRSs into MTRSs.

We first show how pattern matching of general LHSs can be simulated by MTRS rules, using the following example:

$$f(g(X), g(X)) \to r_1(X) \tag{4.1}$$
$$f(X, h) \to r_2(X) \tag{4.2}$$

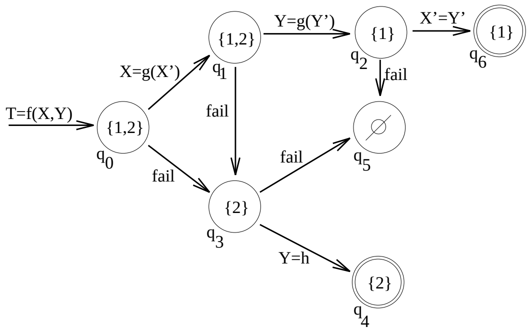


Fig. 6. A tree matching automaton

This example contains overlapping rules and a nonlinearity, thus presenting the basic problems to be addressed by a TRS pattern-match compiler.

It is well-known that we can use tree matching automata [HO82, Wal91] for determining whether a given term T matches the LHS of one (or more) of a set of rewrite rules. In Fig. 6, a matching automaton for this set of LHSs is depicted.

The states q_i of the automaton encode the set of patterns that might still match the term under consideration. Accepting states, in which it is known that T matches one or more rules, are indicated by a double circle. Based on the value of an argument position, there are success and failure transitions between states. It is understood that a failure transition is only made when no other transition is possible.

We will now show how this matching automaton is simulated by innermost rewriting with specificity of a TRS in which every rule has a minimal LHS.

There are three crucial ideas in this simulation. The first idea is that in innermost rewriting, the arguments of T are in normal form before a match with T is attempted, and when T fails to match, it is itself in normal form. Therefore, for every function symbol f, we introduce a *constructor variant* f_c which simulates f ($\mathcal{S}(f_c) = f$), and which indicates that matching has been attempted and failed. It follows that normal forms always consist entirely of constructor variants.

The second idea (found also in [Pet92]) is to encode the states of the automaton by (new) functions $q_0 \mapsto f$, $q_1 \mapsto f_g$, $q_2 \mapsto f_{gg}$, $q_3 \mapsto f_X$, $q_4 \mapsto r_2$, $q_5 \mapsto f_c$ and $q_6 \mapsto r_1$, and the transitions by rules defining these functions. The map $\mathcal{S}$ is undefined on the new functions, i.e., f_g, f_{gg} and f_X.

The third idea is that failure transitions correspond to most general rules, so when a term is rewritten innermost, with (syntactic) specificity ordering according to the MTRS[1] below, rewriting in the TRS above is simulated.

$$h \rightarrow h_c \tag{4.3}$$
$$g(X) \rightarrow g_c(X) \tag{4.4}$$
$$f(g_c(X), Y) \rightarrow f_g(X, Y) \tag{4.5}$$

[1]It is an MTRS because the RHSs are chosen judiciously. See Section 4.1 for a transformation to remedy non-minimal RHSs.

$$\begin{aligned} f(X,Y) &\rightarrow f_X(X,Y) && (4.6)\\ f_g(X,g_c(Y)) &\rightarrow f_{gg}(X,Y) && (4.7)\\ f_g(X,Y) &\rightarrow f_X(g_c(X),Y) && (4.8)\\ f_{gg}(X,Y) &\rightarrow f_{gge}(eq(X,Y),X,Y) && (4.9)\\ f_{gge}(\text{true},X,Y) &\rightarrow r_1(X,Y) && (4.10)\\ f_{gge}(B,X,Y) &\rightarrow f_c(g_c(X),g_c(Y)) && (4.11)\\ f_X(X,h_c) &\rightarrow r_2(X) && (4.12)\\ f_X(X,Y) &\rightarrow f_c(X,Y) && (4.13) \end{aligned}$$

Note that in rules (4.8) and (4.11), previously deconstructed terms are reconstructed. At the cost of introducing extra variables, the cost of reconstruction can be avoided. The function `eq`, which is used in rule 4.9 to test (syntactic) equality of its arguments, can easily be defined by a TRS if the signature is known and innermost rewriting is assumed. For innermost rewriting, this simulation is sound, complete, and termination conserving.

4.1 Transforming Complicated RHSs

Here we present a transformation that will transform a TRS N, which may have RHSs that do not conform to the RHSs found in MTRSs, into a simulating TRS M, whose RHSs are minimal. Any rule with a minimal LHS and a non-minimal RHS has the form $l(\vec{x},\vec{y},\vec{z}) \rightarrow h(\vec{x},\vec{t},u,\vec{z})$, where u is either a variable or a term $g(\vec{u})$, and $\vec{x}$ and $\vec{z}$ contain only variables, and are taken of maximal length. The goal is to reduce the non-compliant segment $\vec{t},u$.

In case u is a variable, we replace the rule by the following rules:

$$\begin{aligned} l(\vec{x},\vec{y},\vec{z}) &\rightarrow h^R(\vec{x},\vec{y},u,\vec{z}) && (4.14)\\ h^R(\vec{x},\vec{y},u,\vec{z}) &\rightarrow h(\vec{x},\vec{t},u,\vec{z}) && (4.15) \end{aligned}$$

Rule (4.14) is an instance of **A**, and rule (4.15) has a shorter non-compliant segment $\vec{t}$.

In case u is a non-variable ($g(\vec{u})$), we replace the rule by the following rules:

$$\begin{aligned} l(\vec{x},\vec{y},\vec{z}) &\rightarrow h^R(\vec{x},\vec{t},\vec{u},\vec{z}) && (4.16)\\ h^R(\vec{x'},\vec{y'},\vec{z}) &\rightarrow h(\vec{x'},g(\vec{y'}),\vec{z}) && (4.17) \end{aligned}$$

where $|\vec{x'}| = |\vec{x}| + |\vec{t}|$, $|\vec{y'}| = |\vec{u}|$, and h^R is a fresh function symbol which did not already occur in the TRS, and $\vec{x'}$ and $\vec{y'}$ consist entirely of fresh variables.

Rule (4.17) is an instance of **C**, and Rule (4.16) has one function symbol less on the RHS than the original rule. Therefore, the number of transformation steps is bounded by the total number of nested function symbols in RHSs of the original TRS.

We take the simulation map $\mathcal{S}$ to be undefined for h^R. It is not very hard to see that $(\mathcal{S},M)$ is sound, complete and termination preserving, we show the vital ingredient of the proof, only for the case that u is nonvariable. Let s be l^σ. According to rule (4.16) of M,

s rewrites to the term $t^R = h^R(\vec{x}, \vec{t}, \vec{u}, \vec{z})^\sigma$. Under the substitution $\vec{x'} \mapsto \vec{x}\,\vec{t}, \vec{y} \mapsto \vec{u}, \vec{z'} \mapsto \vec{z}$, rule (4.17) of M rewrites t^R to $h(\vec{x}, \vec{t}, g(\vec{u}), \vec{z})^\sigma$, which is the original RHS, instantiated by σ.

5. An Abstract Machine View on MTRSs

$$\begin{array}{rcl}
\langle \mathbf{goto}(l) \cdot p, P, C, T, A\rangle & = & \langle \mathrm{get}(l, P), P, C, T, A\rangle \\
\langle \mathbf{cpush}(l) \cdot p, P, C, T, A\rangle & = & \langle p, P, l \cdot C, T, A\rangle \\
\langle \mathbf{recycle} \cdot p, P, l \cdot C, T, A\rangle & = & \langle \mathrm{get}(l, P), P, C, T, A\rangle \\
\langle \mathbf{recycle} \cdot p, P, \epsilon, \epsilon, a \cdot \epsilon\rangle & = & \mathrm{nf}(a) \\
\langle \mathbf{build}(c, n) \cdot p, P, C, T, t_1 \cdots t_n \cdot A\rangle & = & \langle p, P, C, T, c(t_1, \ldots, t_n) \cdot A\rangle \\
\langle \mathbf{match}(c, n, l) \cdot p, P, C, T, c(t_1, \ldots, t_n) \cdot A\rangle & = & \langle \mathrm{get}(l, P), P, C, T, t_1 \cdots t_n \cdot A\rangle \\
\langle \mathbf{match}(c, n, l) \cdot p, P, C, T, c'(t_1, \ldots, t_m) \cdot A\rangle & = & \langle p, P, C, T, c'(t_1, \ldots, t_m) \cdot A\rangle \\
 & & \text{when } c \neq c' \\
\langle \mathbf{tpusha}(i) \cdot p, P, C, T, A\rangle & = & \langle p, P, C, T, \mathrm{top}(i, T) \cdot A\rangle \\
\langle \mathbf{apusha}(i) \cdot p, P, C, T, A\rangle & = & \langle p, P, C, T, \mathrm{top}(i, A) \cdot A\rangle \\
\langle \mathbf{tdrop}(n) \cdot p, P, C, t_1 \cdots t_n \cdot T, A\rangle & = & \langle p, P, C, T, A\rangle \\
\langle \mathbf{skip}(s(n)) \cdot p, P, C, T, a \cdot A\rangle & = & \langle \mathbf{skip}(n) \cdot p, P, C, a \cdot T, A\rangle \\
\langle \mathbf{skip}(0) \cdot p, P, C, T, A\rangle & = & \langle p, P, C, T, A\rangle \\
\langle \mathbf{retract}(s(n)) \cdot p, P, C, t \cdot T, A\rangle & = & \langle \mathbf{retract}(n) \cdot p, P, C, T, t \cdot A\rangle \\
\langle \mathbf{retract}(0) \cdot p, P, C, T, A\rangle & = & \langle p, P, C, T, A\rangle \\
\mathrm{top}(0, a \cdot T) & = & a \\
\mathrm{top}(\mathrm{s}(n), a \cdot T) & = & \mathrm{top}(n, T)
\end{array}$$

Fig. 7. An algebraic specification of ARM instructions.

The rules of MTRSs can be viewed as (short sequences of) instructions for an abstract machine with three stacks C (control), A (arguments) and T (traversal), a program counter p and a program P, visualized as a tuple $\langle p, P, C, T, A\rangle$. In Fig. 7, we give an algebraic specification of this machine, which we will now explain in text.

The program counter p denotes the fragment of the program P which is currently being executed. The **goto** instruction replaces the current fragment by a fragment of P, which is obtained as $\mathrm{get}(l, P)$, where l is a unique label identifying the fragment.

The **cpush** instruction pushes a label onto the C (mnemonic for *control*) stack, whence it may be removed by a **recycle** instruction, which causes execution to continue at the label obtained from C.

A **build** instruction builds a term from a function symbol and a number of terms from the A (mnemonic for *argument*) stack.

The **match** instruction matches the term on top of the A stack against a certain function symbol. On success, it decomposes the topmost term on the A stack, leaving the arguments on the A stack, and continues at a specified label. On failure, the next instruction of the current fragment is executed. Strictly speaking, the arity argument of **match** is redundant, because the number of arguments may be obtained by inspecting the term on top of the A

stack.

The instruction **tpusha** pushes an argument from the T (mnemonic for *traversal*) stack onto the A stack, the **apusha** pushes an argument from the A stack onto the A stack, the **skip** instruction moves a number of terms from the A stack to the T stack, **retract** does this in the reverse direction, and the **tdrop** instruction removes a number of terms from the T stack. We assume the programs to be such that top is never applied to an empty stack.

As shown, the rule for **tdrop** is actually short for two rules, $\langle \mathbf{tdrop}(0) \cdot p, P, C, T, A\rangle = \langle p, P, C, T, A\rangle$ and $\langle \mathbf{tdrop}(\mathrm{s}(n)) \cdot p, P, C, t \cdot T, A\rangle = \langle \mathbf{tdrop}(n) \cdot p, P, C, T, A\rangle$ when $n \neq 0$. It is written as it is because $\mathbf{tdrop}(n)$ is best understood as a single abstract machine instruction. For the same reason, we have written the **build** instruction in a similar way.

5.1 Some Further Constraints on MTRSs

The actual interpretation of MTRS rules as instructions for ARM presupposes some further constraints, which can be met by two more transformations. For other mappings from MTRSs to machine code (say, via Pascal, Lisp or C), these additional constraints may not be essential, which is the reason we deal with them in this section.

1. Defined function symbols must always have a most general rule (i.e., one of the forms **C, R, A, D** or **I**). This constraint is satisfied by MTRSs produced by the transformation in [KW].

2. All **M** rules for a given function symbol must be mutually exclusive. This constraint is satisfied by MTRSs produced by the transformation in [KW].

3. Constructors should not occur as the outermost function symbol of a RHS. This can be remedied by adding a new rule $r(y) \rightarrow y$, where r does not already occur in the MTRS, replacing all RHSs s with an outermost constructor symbol by $r(s)$, and applying the RHS transformation of Section 4.1 until we have again an MTRS.

4. When any MTRS rule[2] $f(\vec{x}, \vec{t}) \rightarrow g(\vec{x}, \vec{t'})$ applies, the terms corresponding to $\vec{x}$ should be on the traversal stack. In the transformation in [KW], the original function symbols have all arguments on the argument stack; **I** rules move them gradually to the traversal stack. Newly introduced function symbols are annotated with the number of arguments that are already on the traversal stack, as in $f^{|\vec{x}|}(\vec{x}, g(\vec{y}), \vec{z}) \rightarrow h^{|\vec{x}|}(\vec{x}, \vec{y}, \vec{z})$.

Constraint 1 ensures that normal forms consist entirely of free constructors.

Interestingly enough, only rules of the form **M** may fail to apply to a term with the defined topsymbol f, since all other rules are unconditional (given the topsymbol). From this and constraint 2, it is easily concluded that the rewrite relation, restricted to innermost rewriting with syntactic specificity, is deterministic.

In the transformations in [KW] and Section 4.1, it is clear where constraints 3 and 4 are violated, and the necessary additional rules can more easily be produced by a slightly modified version of these transformations.

[2]An **R** rule does not conform to the format that follows. For **R** rules we take $|\vec{x}| = 0$.

$$
\begin{array}{rrcll}
\mathbf{C}: & f^{|\vec{x}|}(\vec{x},\vec{y},\vec{z}) & \rightarrow & h^{|\vec{x}|}(\vec{x},g(\vec{y}),\vec{z}) & \begin{cases} (g \text{ defined}) & \mathbf{f}: \mathbf{cpush}(h); \mathbf{goto}(g) \\ (g \text{ free}) & \mathbf{f}: \mathbf{build}(g, |\,\vec{y}\,|); \mathbf{goto}(h) \end{cases} \\
\mathbf{M}: & f^{|\vec{x}|}(\vec{x},c(\vec{y}),\vec{z}) & \rightarrow & h^{|\vec{x}|}(\vec{x},\vec{y},\vec{z}) & \mathbf{f}: \mathbf{match}(c,h) \\
\mathbf{A}: & f^{|\vec{x}|}(\vec{x},\vec{z}) & \rightarrow & h^{|\vec{x}|}(\vec{x},x_k,\vec{z}) & \mathbf{f}: \mathbf{tpusha}(|\,\vec{x}\,| - k); \mathbf{goto}(h) \\
\mathbf{A}: & f^{|\vec{x}|}(\vec{x},\vec{z}) & \rightarrow & h^{|\vec{x}|}(\vec{x},z_k,\vec{z}) & \mathbf{f}: \mathbf{apusha}(k-1); \mathbf{goto}(h) \\
\mathbf{D}: & f^{|\vec{x}|}(\vec{x},\vec{y},\vec{z}) & \rightarrow & h^{|\vec{x}|}(\vec{x},\vec{z}) & \mathbf{f}: \mathbf{tdrop}(|\,\vec{y}\,|); \mathbf{goto}(h) \\
\mathbf{I}: & f^{n}(\vec{x}) & \rightarrow & h^{m}(\vec{x}) & \begin{cases} (m \geq n) & \mathbf{f}: \mathbf{skip}(m-n); \mathbf{goto}(h) \\ (n > m) & \mathbf{f}: \mathbf{retract}(n-m); \mathbf{goto}(h) \end{cases} \\
\mathbf{R}: & f^{1}(x) & \rightarrow & x & \mathbf{f}: \mathbf{recycle}
\end{array}
$$

Fig. 8. The instruction mapping

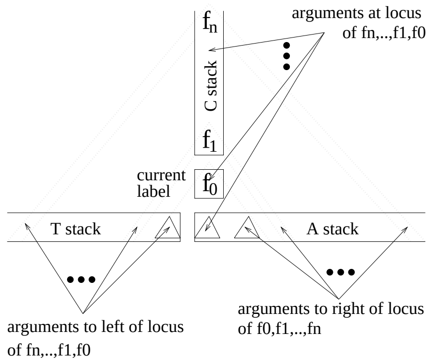

f$_n$
arguments at locus of fn,..,f1,f0
C stack
f$_1$
current label
f$_0$
T stack
A stack
arguments to left of locus of fn,..,f1,f0
arguments to right of locus of f0,f1,..,fn

Fig. 9. The invariant maintained by the mapping

5.2 Interpreting MTRS Forms as Instruction Sequences

In Fig. 8, we now show the straightforward mapping of MTRS forms onto instruction sequences. First, we view defined symbols as labels in the machine program. A rule with f as outermost function symbol on the LHS *defines* the instructions at label f, and a rule with h as outermost function symbol on the RHS *uses* the label h, i.e. it causes execution to continue at label h. (for rules with the same label, we simply concatenate the code, taking care that the code for the least specific rule is put at the end).

Form **C** has two labels on the RHS, of which the innermost label g is interpreted as the label where execution should continue, whereas the outermost label h is pushed on the C stack for future reference.

Form **R** has no labels on the RHS, which is taken to mean that execution should continue at a label popped from the C stack.

Second, the similarity of the variable configurations on LHS and RHS is exploited by consistently mapping the left part of the arguments $t_0, \ldots, t_l$ (with l the *locus*) to the T stack (t_l on top), and mapping the right part of the arguments $t_{l+1}, \ldots, t_{n-1}$ to the A stack (t_{l+1} on top). Given constraint 4 on page 11, this ensures that only the top of either stack is changed by any rule.

Every function symbol in an MTRS corresponds with a label, and the machine is initialized by traversing the input term in pre-order, pushing all encountered function symbols on the C

stack. Then the machine is started with the **recycle** instruction. When the machine halts, it has the normal form of the input term on the A stack.

It is easily verified that this interpretation of an MTRS implements rightmost innermost rewriting, by checking that the machine instructions associated with a particular MTRS rule satisfy the invariant depicted in Fig. 8.

The top-symbol f_n of the entire term being rewritten, is at the bottom of the C stack (the C stack is shown upside-down), all arguments left of the locus of f_n are on the T stack, all arguments right of the locus of f_n are on the A stack, and, recursively, the arguments at and below the locus (with top-symbol $f_{n-1} \dots f_1$) are represented less deep than f_n on the control-stack.

The symbol f_0 is the current label, which does not reside on the stack, but is expressed in the current state of the machine (i.e., the program counter p in the tuple $\langle p, P, C, T, A \rangle$). The initial state satisfies this invariant, because the function symbols of the input term have locus 0.

5.3 The Naturals Revisited

After the transformation to satisfy the additional requirements, and the replacement of most **goto** instructions by the code at their destination label, the instructions generated for the example in the introduction:

$$\begin{array}{ll}
zero: & \mathbf{build}(zero_c, 0); \mathbf{recycle} \\
succ: & \mathbf{build}(succ_c, 1); \mathbf{recycle} \\
plus: & \mathbf{match}(zero_c, plus\_zero); \\
 & \mathbf{match}(succ_c, plus\_succ); \\
 & \mathbf{build}(plus_c, 2); \mathbf{recycle}; \\
plus\_zero: & \mathbf{recycle}; \\
plus\_succ: & \mathbf{cpush}(succ); \mathbf{goto}(plus);
\end{array}$$

It is readily verified, that the state $\langle \mathbf{recycle}, P, zero \cdot succ \cdot zero \cdot succ \cdot plus \cdot \epsilon, \epsilon, \epsilon \rangle$ with P the program above, is transformed into the state $\langle \epsilon, P, \epsilon, \epsilon, succ_c(succ_c(zero_c)) \cdot \epsilon$ by the algebraic semantics given in Fig. 7.

5.4 Efficiency and Complexity of ARM

All instructions of the ARM machine can be implemented efficiently on modern microprocessors. Usually, **goto** and **recycle** are available as native instructions, **cpush**, **match**, **apush**, and **tdrop** can be implemented in one or a few native instructions, and **build** and **skip** can be implemented in $k|n|$ instructions, where k is a small factor, and n is the parameter of the instruction.

Furthermore, only the implementation of **build** requires write-access to global (heap) storage, and only **match** requires read-access to such storage. The other instructions only access relatively cheap local (stack) storage.

We believe that this set is the minimal set for which efficiency can be conserved in the translation from general TRSs. In [HF$^+$96], concrete execution times are reported concerning an implementation based on ARM technology.

In [KW], we present the theoretical result that the transformation from TRSs into MTRS increases the cost of rewriting with at most a linear factor of 4.5. In practice, however, this constant appears to be close to 1. The plausibility of this statement follows from inspection of the ARM code, which is close to the machine code that efficient compilers for (eager) functional languages generate for comparable programs.

6. Relation to Other Abstract Machines

The abstract machine presented in Section 5 is much less complex than ARM [KW93].

In [HG94], a provably correct compiler for term rewriting systems is described, using an abstract machine TRIM, which bears some similarity to ARM. The approach seems to be geared more towards provability than towards efficiency, because environments are built explicitly on the heap (whereas the 'environments' of ARM are on a, cheaper, stack).

In the context of (lazy) functional languages, many different abstract machines are used, notably SKIM [Tur79], the Categorial Abstract Machine (CAM, [CCM85]), the Three Instruction Machine (TIM, [FW87]), the G-machine [PJ87], its successor, the spineless tagless G-machine (STG, [JS89]), and the ABC machine [PvE93]. These machines address lazy graph rewriting of curried higher-order function applications (CAM is basically innermost, but supports lazy evaluation). In contrast, ARM is designed for first-order innermost reduction on stacks, where the graph structure is only explicit in the normal forms, pointers to wich reside on the C and A stacks (see Fig. 8). Laziness can be added as a source-to-source transformation, given one extra ARM instruction [KW95]. Higher-order functions as they appear in implementations of functional programming languages can be implemented by *applicative term rewriting systems* [Tur79].

Because lazy graph rewriting is expensive, and most of the time not needed, most of the lazy functional abstract machines have add-ons for innermost (strict) rewriting, making them more complicated than ARM.

It is our experience that comparisons of (implementations of) abstract machines tend to be hard to interpret and are often misleading. A somewhat specific comparison is presented in [HF$^+$96]. Based on that and other experiences ARM technology can be said to lead to efficient implementations.

7. Conclusions and Future Work

We have presented minimal term rewriting systems (MTRSs) and a notion of simulation, and shown that under this notion of simulation, MTRSs can be used to efficiently simulate innermost rewriting of an arbitrary TRS.

Furthermore, an MTRS can directly be interpreted as a program for the Abstract Rewriting Machine (ARM), which has a straightforward, efficient implementation on conventional hardware.

Thus, a transformation that takes TRSs into simulating MTRSs can be used as a TRS compiler that produces efficient code. The resulting code turns out to be comparable with the code generated by conventional technology [HF$^+$96].

The most interesting point of our technique is that the compilation from TRSs into MTRSs takes place entirely in the theoretically attractive realm of TRSs. We expect to exploit this fact for proving the correctness of our compiler rigorously. Next to proving correctness of the compiler, it seems an interesting project to investigate how MTRSs and our notion of simulation can be applied in the general study of TRSs, e.g., for the simplification of termination proofs, as suggested by Hans Zantema.

We would like to thank Jan Bergstra, Jan Heering, Paul Klint and Bas Luttik for reading and commenting on drafts of this paper.

References

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers. Principles, Techniques and Tools.* Addison-Wesley, 1986.

[BBKW89] J.C.M. Baeten, J.A. Bergstra, J.W. Klop, and W.P. Weijland. Term-rewriting systems with rule priorities. *Theoretical Computer Science*, 67(1):283–301, 1989.

[CCM85] G. Cousineau, P.-L. Curien, and M. Mauny. The categorical abstract machine. In J.-P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 50–64. Springer-Verlag, 1985.

[FW87] Jon Fairbairn and Stuart Wray. Tim: A simple, lazy abstract machine to execute supercombinators. In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 34–45. Springer-Verlag, 1987.

[HF$^+$96] Pieter H. Hartel, Marc Feeley, et al. Benchmarking implementations of functional languages with "pseudoknot", a float-intensive benchmark. *Journal of Functional Programming*, 1996. Accepted for publication.

[HG94] Lutz H. Hamel and Joseph A. Goguen. Towards a provably correct compiler for OBJ3. In *Proceedings of the International Conference on Programming Language Implementation and Logic Programming, PLILP '94*, 1994.

[HO82] C.M. Hoffmann and M.J. O'Donnell. Pattern matching in trees. *Journal of the ACM*, 29(1):68–95, 1982.

[JS89] Simon L Peyton Jones and Jon Salkild. The Spineless Tagless G-machine. In *Functional Programming and Computer Architecture*, pages 184–201. ACM, 1989.

[Klo92] J.W. Klop. Term rewriting systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science, Volume 2.*, pages 1–116. Oxford University Press, 1992.

[KW] J.F.Th. Kamperman and H.R. Walters. A compiler for term rewriting systems. CWI Technical Report in preparation.

[KW93] J.F.Th. Kamperman and H.R. Walters. ARM – Abstract Rewriting Machine. In H.A. Wijshoff, editor, *Computing Science in the Netherlands*, pages 193–204, 1993.

[KW95] J.F.Th. Kamperman and H.R. Walters. Lazy rewriting and eager machinery. In Jieh Hsiang, editor, *Rewriting Techniques and Applications*, number 914 in

Lecture Notes in Computer Science, pages 147–162. Springer-Verlag, 1995.

[Pet92] Mikael Pettersson. A term pattern-match compiler inspired by finite automata theory. In U. Kastens and P. Pfahler, editors, *Proceedings of the Fourth International Conference on Compiler Construction*, number 641 in Lecture Notes in Computer Science, pages 258–270. Springer-Verlag, 1992.

[PJ87] Simon L. Peyton-Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.

[PvE93] M J. Plasmeijer and M C J D. van Eekelen. *Functional Programming and Parallel Graph Rewriting.* Addison Wesley, 1993.

[Tur79] D.A. Turner. A new implementation technique for applicative languages. *Software Practice and Experience*, 9:31–49, 1979.

[Wal91] H.R. Walters. *On Equal Terms, Implementing Algebraic Specifications.* PhD thesis, University of Amsterdam, 1991. Available by *ftp* from ftp.cwi.nl:/pub/gipe/reports as Wal91.ps.Z.