The Systematic Development of a Pattern-Matching Algorithm using Term Rewrite Systems

*J.-P. Katoen**, *A. Nymeyer*
University of Twente, Department of Computer Science,
P.O. Box 217, 7500 AE Enschede, The Netherlands
{*katoen,nymeyer*}*@cs.utwente.nl*

Abstract

A term rewrite system is used to specify a pattern matcher in a code generator. The pattern matcher derived from the term rewrite system computes all the sequences of rewrite rules that will reduce a given expression tree to a given goal. While the number of sequences of rewrite rules that are generated is typically enormous, many sequences are in fact redundant. A theory and accompanying algorithms are developed that identify and remove these redundant rewrite sequences. These algorithms terminate if the term rewrite system is finite.

Keywords Term rewrite systems, code generation, pattern matching, formal techniques.

1 Introduction

Term rewrite systems have traditionally been used to prove properties of abstract data types, implement functional languages and mechanise deduction systems, to name just a few areas. Term rewrite systems can also be used to specify part of the back-end of a compiler—the so-called *pattern matcher*. In a pattern matcher, rewrite rules are used to rewrite a given (input) term into a given goal term. A pattern here is simply a (sub)term that matches the term on the left-hand side of a rewrite rule. This pattern is replaced by the term on the right-hand side of the rewrite rule.

A term rewrite system defines a mapping between the intermediate representation and the machine instructions. The intermediate representation is code that is generated by the front-end (usually consisting of a scanner, parser and type-checker) of the compiler. For the purposes of this work the intermediate representation will simply consist of expression trees. Actually, the mapping between the intermediate representation and machine instructions is indirect as machine instructions are only associated with rewrite rules. During pattern matching, when we transform a given expression tree (term) using a rewrite rule, we generate the associated instruction. In effect, the semantics of a rewrite rule is the associated machine instruction.

This application of term rewrite systems is fundamentally different from traditional areas. In code generation, a term rewrite system is neither confluent nor terminating. It is not confluent because there may be many ways of rewriting a given term, and it is not terminating because there may be rules that can always be applied. Because there can be many ways of rewriting a given term, a cost is added to the rewrite rules. The resulting term rewrite system is referred to as *weighted*. However, the costs are not used by the pattern matcher, but by a subsequent phase, the pattern selector, which must choose a minimum sequence of rewrite rules that will transform (rewrite) the given expression tree into the given goal. Note, therefore, that while costs are shown and even occasionally referred to in this work, they do not play an active role.

In a nutshell, the main problem in code generation is that there is typically an enormous number (actually an infinite number) of instruction sequences that correspond to a given expression tree, where each different instruction sequence corresponds to a different sequence of rewrites of the expression tree. In code generation, we must not only deal with this large number, we must select an optimal sequence.

Having described the context of this research, we now focus on the role that term rewrite systems play in code generation, namely as a specification formalism of the pattern matcher. Note that we do not address the problem of pattern selection in this paper. We are only concerned with generating all the possible ways of rewriting an expression tree, not which way is the best (for this we refer the reader to Nymeyer et al. [16]). We will also not show machine instructions, nor discuss other aspects of code generation like register and memory allocation.

*Current affiliation is University of Erlangen-Nürnberg, Institut für Informatik VII, Martensstrasse 3, 91058 Erlangen, Germany. E-mail: *katoen@informatik.uni-erlangen.de*

Proceedings of CATS'97 (Computing: The Australasian Theory Symposium), Sydney, Australia, February 5–7 1997.

In the past, there have been two notable applications of term rewrite systems to code generation. Emmelmann [6] used a term rewrite system to specify a mapping from intermediate to target code, and a tree grammar to specify the target terms and their costs. This idea of using different formalisms to specify the target code, and the mapping from intermediate to target code originates from Giegerich [10, 11]. Emmelmann's ambitious work resulted in a complex system that would be difficult to implement. A second, more successful application of term rewrite systems was that by Pelegri-Llopart and Graham [17, 18]. Their work, in fact, forms the starting point of our work. While a number of the concepts that we use are also used by Pelegri-Llopart and Graham, their work is informal and less concise, and they devote much attention to implementation issues such as the pre-computing of tables.

Over the last two decades there have been many attempts to find the right specification formalism for code generation. The most popular have been LR grammars and tree grammars. The so-called Graham-Glanville method uses an LR grammar as a specification. This method had its heyday in the late 1970s and early 1980s (see [12, 9]). However, LR grammars were found to be too restrictive and cumbersome. The method made way for tree grammars, and code generators that use either a top-down traversal strategy (see [1, 4]), or more popularly, a bottom-up strategy ([2, 3, 7, 8, 13]). The advantage of a term rewrite system over a tree grammar is that a term rewrite system has more specification 'power'. Rules that specify algebraic properties (like commutativity) can be used in a term rewrite system but not in a tree grammar.

The aim of this work is to present a theory of pattern matching, and to systematically develop an (intelligent) pattern-matching algorithm. In Section 2 we will define a weighted term rewrite system. For a more elaborate treatment of term rewrite systems we refer the reader to [5]. As well as outlining the basic theory, we present a naive pattern-matching algorithm that generates all rewrite sequences for a given expression tree and goal. In Section 3 we show that many of the rewrite sequences generated by the naive algorithm are redundant. This redundancy is caused by the fact that many rewrite sequences are simply permutations of each other, and hence have the same cost. We eliminate permuted rewrite sequences by considering only rewrite sequences that are in *normal-form*. Two algorithms that generate normal-form sequences are presented. In Section 4 we see that there is another form of redundancy in the normal-form rewrite sequences. This redundancy results from the action during term rewriting of *variables* in the term rewrite system. We eliminate these redundant sequences by defining *strong normal form* rewrite sequences. An algorithm that generates rewrite sequences in strong normal form is given. Finally in Section 5 we present our conclusions. The proofs of theorems and lemmas given in this paper are not shown (these can be found in [16]).

2 Weighted term rewrite systems

We denote the set of naturals by $\mathbb{N}$, the set $\mathbb{N}\backslash\{0\}$ by $\mathbb{N}_+$, and the set of non-negative reals by $\mathbb{R}^+$.

Definition 2.1 *Ranked alphabet*

A ranked alphabet Σ is a pair (S, r) with S a finite set and $r \in S \to \mathbb{N}$. $\triangle$

Elements of S are called *function symbols* and $r(a)$ is called the *rank* of symbol a. Function symbols with rank 0 are called *constants.* Σ_n denotes the set of function symbols with rank n, that is, $\Sigma_n = \{\, a \in S \mid r(a) = n \,\}$. We assume $\mathcal{V}$ is an infinite universe of variable symbols, and $V \subseteq \mathcal{V}$.

Definition 2.2 *Terms*

For Σ a ranked alphabet and V a set of variable symbols, the set of terms, $T_\Sigma(V)$ is the smallest set satisfying the following:

- $V \subseteq T_\Sigma(V)$ and $\Sigma_0 \subseteq T_\Sigma(V)$
- $\forall\, a \in \Sigma_n$ and $t_1, \ldots, t_n \in T_\Sigma(V)$ implies $a(t_1, \ldots, t_n) \in T_\Sigma(V)$, for $n \geq 1$

$\triangle$

For term t, *Var*(t) denotes the set of variables in t. If *Var*$(t) = \emptyset$ then t is called a *ground term.*

A sub-term of a term can be indicated by a path, represented as a string of positive naturals separated by dots, from the outermost symbol of the term (the 'root') to the root of the sub-term. For P a set of sequences and n a natural number, let $n\cdot P$ denote $\{\, n\cdot p \mid p \in P \,\}$. The position of the root is denoted by ε.

Definition 2.3 *Positions*

The set of positions $Pos \in T_\Sigma(V) \to \mathcal{P}(\mathbb{N}_+^*)$ of a term t is defined as:

- $Pos(t) = \{\, \varepsilon \,\}$, if $t \in \Sigma_0 \cup V$
- $Pos(a(t_1, \ldots, t_n)) = \{\, \varepsilon, 1\cdot Pos(t_1), \ldots, n\cdot Pos(t_n) \,\}$

$\triangle$

A trailing ε in a position can be omitted; for example $2\cdot1\cdot\varepsilon$ is written as $2\cdot1$. By definition, $Pos(t)$ is prefix-closed for all terms t. Position q is 'higher than' p if q is a proper prefix of p. The sub-term of a term t at position $p \in Pos(t)$ is denoted $t|_p$.

Definition 2.4 *Weighted term rewrite system*

A weighted term rewrite system (WTRS) is a triple $((\Sigma, V), R, C)$, where

- Σ, a non-empty ranked alphabet
- V, a finite set of variables
- R, a non-empty, finite subset of $T_\Sigma(V) \times T_\Sigma(V)$
- $C \in R \to \mathbb{R}^+$, a cost function

with the constraints $t' \neq t$, $t \notin V$ and $\mathrm{Var}(t') \subseteq \mathrm{Var}(t)$, for all $(t, t') \in R$. $\triangle$

Elements of R are called *rewrite rules*. An element $(t, t') \in R$ is usually written as $t \longrightarrow t'$ where t is called the left-hand side, and t' the right-hand side of the rewrite rule. Elements of R are usually uniquely identified as r_1, r_2, and so on. The cost function C assigns to each rewrite rule a non-negative cost. This cost reflects the cost of the instruction associated with the rewrite rule and may take into account, for instance, the number of instruction cycles, or the number of memory accesses. When C is irrelevant it is omitted from the WTRS. A term rewrite system (TRS) is in that case a tuple $((\Sigma, V), R)$.

The first constraint in the above definition says that R should be irreflexive, and the second constraint that the left-hand side of a rewrite rule may not consist of a single variable. The last constraint says that no new variables may be introduced by a rewrite rule. A WTRS is called *ground* if all left-hand sides of rewrite rules are ground terms.
The WTRS shown below will be used as a running example in this work.

Example 2.5 Let $((\Sigma, V), R, C)$ be a WTRS, where $\Sigma = (S, r)$, $S = \{+, i, c, d, r\}$, $r(+) = 2, r(i) = 1, r(c) = r(d) = r(r) = 0$, and $V = \{x, y\}$. Here c represents the constant 1, d represents a data register, r represents a general register, and i stands for increment. The set R of rules is defined by:

$$\begin{array}{ll} \{\, r_1 : +(d,c) \longrightarrow i(d), & r_2 : +(d,r) \longrightarrow r, \\ \;\; r_3 : +(x,y) \longrightarrow +(y,x), & r_4 : i(r) \longrightarrow r, \\ \;\; r_5 : d \longrightarrow r, & r_6 : c \longrightarrow d, \\ \;\; r_7 : r \longrightarrow d \,\} & \end{array}$$

The cost function C is defined as $C(r_1)$=4, $C(r_2)$=5, $C(r_3)$=0, $C(r_4)$=2, $C(r_5)$=1, $C(r_6)$=3 and $C(r_7)$=1. Alternative representations of the first three rules are given in Figure 1.

Some example terms are $+(c, d)$ and $i(+(c, i(d)))$. If $t = i(+(c, i(d)))$ then $Pos(t) = \{\varepsilon, 1, 1\cdot 1, 1\cdot 2, 1\cdot 2\cdot 1\}$. Some sub-terms of t are $t|_\varepsilon = t$, $t|_1 = +(c, i(d))$, and $t|_{1\cdot 2\cdot 1} = d$. $\triangle$

Definition 2.6 *Substitution*

Let $\sigma \in V \to T_\Sigma(V)$. For $t \in T_\Sigma(V)$, t under substitution σ, denoted t^σ, is defined as:

- $t^\sigma = \begin{cases} t, & \text{if } t \in \Sigma_0 \\ \sigma(t), & \text{if } t \in V \end{cases}$
- $a(t_1, \ldots, t_n)^\sigma = a(t_1^\sigma, \ldots, t_n^\sigma)$

$\triangle$

Rewrite rules that are identical, except for variable symbols, are considered to be the same. In this work we consider rewrite rules modulo rewrite rule equivalence.

Definition 2.7 *Rewrite rule equivalence*

Rewrite rules $r_1 : t_1 \longrightarrow t_1'$ and $r_2 : t_2 \longrightarrow t_2'$ are equivalent iff there is a bijection $\sigma \in \mathrm{Var}(t_1) \to \mathrm{Var}(t_2)$ such that $t_1^\sigma = t_2$ and $t_1'^\sigma = t_2'$. $\triangle$

For our purposes it suffices to informally define the notion of a rewrite step.

Definition 2.8 *Rewrite step*

Given the TRS $((\Sigma, V), R)$, $r : t \longrightarrow t' \in R$, $t_1, t_2 \in T_\Sigma(V)$ and $p \in Pos(t_1)$, then $t_1 \xRightarrow{\langle r,p\rangle} t_2$ iff t_2 can be obtained from t_1 by replacing $t_1|_p$ by t'^σ in t_1, and using substitution σ with $t^\sigma = t_1|_p$. We can also write $\langle r, p\rangle\, t_1 = t_2$. $\triangle$

A rewrite rule r that is applied at the root position, i.e. $\langle r, \varepsilon\rangle$, is usually abbreviated to r. A sequence of rewrite steps, called a *rewrite sequence*, consists of rewrite steps that are applied one after another.

Definition 2.9 *Rewrite sequence*

Let $t \xRightarrow{\langle r_1,p_1\rangle \ldots \langle r_n,p_n\rangle} t'$ iff $\exists\, t_1, \ldots, t_{n-1}$ such that $t \xRightarrow{\langle r_1,p_1\rangle} t_1 \xRightarrow{\langle r_2,p_2\rangle} \ldots t_{n-1} \xRightarrow{\langle r_n,p_n\rangle} t'$.
$S(t) = \langle r_1, p_1\rangle \ldots \langle r_n, p_n\rangle$ is called a *rewrite sequence* of t. We can also write $S(t)\, t = t'$. $\triangle$

When convenient, we denote a rewrite sequence $S(t)$ by τ. Further, we write $t \xRightarrow{\tau}$ if and only if $\exists t' : t \xRightarrow{\tau} t'$. The empty rewrite sequence is denoted ε, hence $t \xRightarrow{\varepsilon} t$ for all terms t.

The cost of a rewrite sequence τ is defined as the sum of the costs of the rewrite rules in τ. The length of τ is denoted $|\tau|$ and indicates the number of rewrite rules in τ. A rewrite step is a rewrite sequence of length 1. For rewrite sequence τ and rewrite rule r, $\tau \setminus r$ denotes sequence τ with r deleted[1], and $r \in \tau$ denotes that r occurs in τ.

A rewrite sequence τ_1 is called *cyclic* if it contains a proper prefix τ_2 such that for some term t, $t \xRightarrow{\tau_1} t'$ and $t \xRightarrow{\tau_2} t'$. In the rest of this paper we

[1]This operation is only used when r can be uniquely identified in τ.

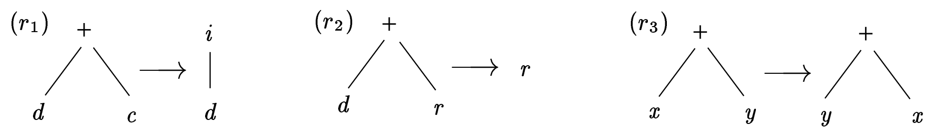


Figure 1: The tree representations of some term rewrite rules

assume all rewrite sequences to be acyclic. If $\tau = \langle r_1, p_1\rangle \ldots \langle r_n, p_n\rangle$ then we define $\bar{\tau} = \{ r_1, \ldots, r_n \}$, i.e. $\bar{\tau}$ is the set of rewrite rules in τ. Actually, $\bar{\tau}$ is a multiset as the same rewrite rule may (and often does) occur more than once in $\bar{\tau}$.

Example 2.10 Consider the WTRS shown in Example 2.5. If $t = +(c,d)$ then we can write $t \xRightarrow{\langle r_3,\epsilon\rangle} t'$, with $t' = +(d,c)$. We can also write $\langle r_3, \epsilon\rangle\, t = t'$. The term t' is obtained from t by replacing $t|_\varepsilon$ by $+(y,x)^\sigma$ in t, using substitution σ with $\sigma(x) = c$ and $\sigma(y) = d$ such that $(x,y)^\sigma = t|_\varepsilon$. △

Example 2.11 An example of a derivation for $t = +(c, i(d))$ of length 4 is:

$$+(c,i(d)) \xRightarrow{\langle r_6,1\rangle} +(d,i(d)) \xRightarrow{\langle r_5,21\rangle} +(d,i(r))$$
$$\xRightarrow{\langle r_4,2\rangle} +(d,r) \xRightarrow{r_2} r$$

and two sequences of length 7 are:

$$+(c,i(d)) \xRightarrow{\langle r_6,1\rangle} +(d,i(d)) \xRightarrow{\langle r_5,21\rangle} +(d,i(r))$$
$$\xRightarrow{r_3} +(i(r),d) \xRightarrow{\langle r_5,2\rangle} +(i(r),r) \xRightarrow{\langle r_4,1\rangle} +(r,r)$$
$$\xRightarrow{\langle r_7,1\rangle} +(d,r) \xRightarrow{r_2} r$$

and

$$+(c,i(d)) \xRightarrow{\langle r_6,1\rangle} +(d,i(d)) \xRightarrow{\langle r_5,1\rangle} +(r,i(d))$$
$$\xRightarrow{\langle r_5,21\rangle} +(r,i(r)) \xRightarrow{\langle r_4,2\rangle} +(r,r) \xRightarrow{\langle r_7,2\rangle} +(r,d)$$
$$\xRightarrow{r_3} +(d,r) \xRightarrow{r_2} r$$

△

In Definition 2.9 we defined the rewrite sequence $S(t)$ of a term t. We now go a step further and label, or decorate, a term with rewrite sequences. Such a rewrite sequence is called a *local rewrite sequence*, and is denoted by $L(t\,|_p)$, where $t\,|_p$ is the sub-term of t at position p at which the local rewrite sequence occurs. Of course, p may be ε (denoting the root). Note that all the positions in the local rewrite sequence $L(t|_p)$ are relative to p.

A term in which each sub-term is labelled by a (possibly empty) local rewrite sequence is called a decorated term, or *decoration*. From now on all terms that we consider will be *ground* terms.

Definition 2.12 *Decoration*

A decoration $D(t)$ is a term in which each sub-term of t at position $p \in Pos(t)$ is labelled with a local rewrite sequence $L(t|_p)$. △

We can usually decorate a given term in many ways. If we wish to differentiate between the rewrite sequences in different decorations, then we use the notation $L_D(t|_p)$.

Given a decoration $D(t)$ of a term t, the corresponding rewrite sequence $S(t)$ can be obtained by a post-order traversal of t. Again, different decorations may lead to different rewrite sequences, so we denote the rewrite sequence of a decoration D by $S_D(t)$.

Definition 2.13 *Rewrite sequence corresponding to a decoration*

The rewrite sequence $S_D(t)$ corresponding to a decoration $D(t)$ is defined as:

$$S_D(t) = \begin{cases} L_D(t|_\varepsilon), & \text{if } t \in \Sigma_0 \\ (1{\cdot}S_D(t_1)\ldots n{\cdot}S_D(t_n))\, L_D(t|_\varepsilon), & \\ \quad \text{if } t = a(t_1,\ldots,t_n) & \end{cases}$$

△

Here, $n \cdot \tau$ for rewrite sequence τ and (positive) natural number n denotes τ where each position p_i in τ is prefixed with $n\cdot$.

Example 2.14 Consider our running example again, and let $t = +(c, i(d))$. Two decorations $D(t)$ and $D'(t)$ are depicted in Figure 2, on the left and right, respectively. The corresponding rewrite sequences are:

$$\begin{aligned} S_D(t) &= \langle r_6,1\rangle\langle r_5,1\rangle\langle r_5,2{\cdot}1\rangle\langle r_4,2\rangle\langle r_7,2\rangle r_3 r_2 \\ S_{D'}(t) &= \langle r_6,1\rangle\langle r_5,2{\cdot}1\rangle r_3\langle r_5,2\rangle\langle r_4,1\rangle\langle r_7,1\rangle r_2 \end{aligned}$$

△

Sets of patterns, called *input* and *output sets*, can now be computed from the decorations of t. These sets define the patterns that match the expression tree. The algorithm to generate these sets is a generalisation of bottom-up tree pattern matching methods (see e.g., Hemerik and Katoen [14]). We begin by defining the inputs and outputs of a decoration.

Definition 2.15 *Inputs of a decoration*

Let $D(t)$ be a decoration such that, for some

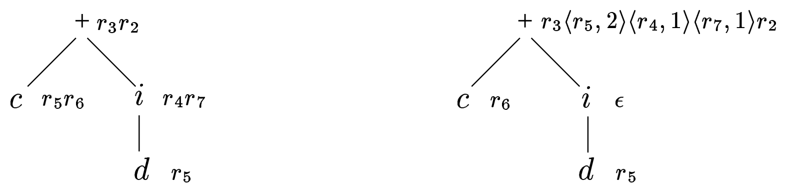


Figure 2: Decorations $D(t)$ and $D'(t)$ of the term $+(c, i(d))$

given goal term g, $t \xRightarrow{S_D(t)} g$. For each sub-term t' of t, the possible inputs, denoted $I_D(t')$, are defined as follows:

$$I_D(t) = \begin{cases} t, & \text{if } t \in \Sigma_0 \\ a(t'_1, \ldots, t'_n) & \text{if } t = a(t_1, \ldots, t_n) \end{cases}$$

where $I_D(t_i) \xRightarrow{L_D(t_i)} t'_i$, for $1 \leq i \leq n$. △

Definition 2.16 *Outputs of a decoration*

Let $D(t)$ be a decoration such that, for some given goal term g, $t \xRightarrow{S_D(t)} g$. For each sub-term t' of t, the possible outputs are defined as $O_D(t) = t'$ with $I_D(t) \xRightarrow{L_D(t)} t'$. △

Given a WTRS $((\Sigma, V), R, C)$ and two ground terms $t, t' \in T_\Sigma$, then, we wish to determine all rewrite sequences τ such that $t \xRightarrow{\tau} t'$. An algorithm to calculate the inputs and outputs for terms t and g, and the corresponding decorations, is given in Figure 3. For convenience, in the algorithm we refer to the type T_Σ as Term and $\mathcal{P}$(Term) as SetOfTerms. This algorithm, which we refer to as the *naive* algorithm, consists of two passes.

In the first, bottom-up pass, carried out by the function *Generate*, sets of *triples*, denoted by $Z(t)$, are computed for all possible goal terms. A triple, written $\langle it, D(t), ot\rangle$, consists of an input it, decoration $D(t)$, and output ot such that $t \xRightarrow{S_D} ot$, and $it \xRightarrow{L_D(t|_\varepsilon)} ot$. The type Triple is defined as Term×Decoration×Term. $\mathcal{P}$(Triple) is denoted by SetOfTriples. In the recursive function *Generate*, the inputs and outputs at each leaf node a in t are initialized to a accompanied with decoration D_ε of t that associates an empty rewrite sequence to each position in t. The inputs and outputs of $a(t_1, \ldots, t_n)$ are initialized to $a(t_{k_1}, \ldots, t_{k_n})$, where t_{k_i} is an output of the i-th child. The decoration $D_1 \oplus \ldots \oplus D_n$ is obtained by decorating the root a with an empty rewrite sequence (i.e., $L_D(t|_\varepsilon) = \varepsilon$), and $L_D(t|_{n\cdot p}) = L_{D_n}(t_n|_p)$ for non-root positions. In the triple do-loop, given some node t in our input term, for each triple $\langle it, D, ot\rangle$, and at each position p in the sub-term rooted at that node, we apply each of the rewrite rules r to the output ot in the triple. If ot is rewritten into ot', then the new triple $\langle it, D \otimes \langle r, p\rangle, ot'\rangle$ is added to the set of triples at the node t. $D \otimes \langle r, p\rangle$ is obtained by appending $\langle r, p\rangle$ to the local rewrite sequence at the root (i.e., $L_D(t|_\varepsilon)$). The remaining local rewrite sequences in D are unaffected. We continue this process until we can add no more triples and we have reached the root node. The resulting set of triples is denoted $W(t)$. The outputs in the set of triples at t are all the terms that can be generated by sequences of rewrite rules applied to the sub-term rooted at t.

In the second, top-down pass, carried out by the function *Trim*, we 'trim' the triples generated in the first pass. At the root position, each triple whose output is not identical to the goal term is removed. Other nodes in the t are trimmed by removing each triple whose output is not identical to an input of its parent node. The resulting trimmed sets of triples are denoted by $V(t)$.

Example 2.17 Given the input term $+(c, i(d))$, the sets of (trimmed) triples $V(t)$ that are generated are shown in the table in Figure 4. Notice that the cardinality of the set V('+') is 2775. The cardinality of the corresponding set of untrimmed triples, W('+'), is 19033! An example of the longest local rewrite sequence that is generated (the length is 16) is shown below:

$$\begin{aligned}
&+(c, i(r)) \xRightarrow{r_3} +(i(r), r) \xRightarrow{\langle r_7,1\rangle} +(i(d), c) \\
&\quad \xRightarrow{r_3} +(c, i(d)) \xRightarrow{\langle r_6,1\rangle} +(d, i(d)) \\
&\quad \xRightarrow{r_3} +(i(d), d) \xRightarrow{\langle r_5,2\rangle} +(i(d), r) \\
&\quad \xRightarrow{r_3} +(r, i(d)) \xRightarrow{\langle r_5,2\cdot1\rangle} +(r, i(r)) \\
&\quad \xRightarrow{\langle r_7,1\rangle} +(d, i(r)) \xRightarrow{r_3} +(i(r), d) \\
&\quad \xRightarrow{\langle r_5,2\rangle} +(i(r), r) \xRightarrow{\langle r_4,1\rangle} +(r, r) \\
&\quad \xRightarrow{\langle r_7,2\rangle} +(r, d) \xRightarrow{\langle r_7,1\rangle} +(d, d) \\
&\quad \xRightarrow{\langle r_5,2\rangle} +(d, r) \xRightarrow{r_2} r
\end{aligned}$$

△

3 Normal-form decorations

We saw in the previous section that the number of rewrite sequences (triples) that are generated by

```
|[ con $((\Sigma, V), R)$: TermRewriteSystem;
        $t, g$ : Term;
  var $W(t), V(t)$: SetOfTriples;
  func $Generate\,(t$ : Term): SetOfTriples
   |[ var $H, Z(t)$: SetOfTriples; $i : \mathbb{N}$;
    $H := Z(t) := \emptyset$;
    if $t :: a \longrightarrow Z(t) := \{\, \langle t, D_\varepsilon, t\rangle \,\}$;
    ▯ $t :: a(t_1, \ldots, t_n) \longrightarrow$|[ for all $1 \le i \le n$ do $Z(t_i) := Generate(t_i)$ od;
                                   (* Let $O(t_i) = \{\, ot_{k_i} \mid \langle it_{k_i}, D_{k_i}, ot_{k_i}\rangle \in Z(t_i) \,\}$ *)
                                   for all $(t_{k_1}, \ldots, t_{k_n}) \in O(t_1) \times \ldots \times O(t_n)$
                                   do $Z(t) := Z(t) \cup \{\, \langle a(t_{k_1}, \ldots, t_{k_n}), D_{k_1} \oplus \ldots \oplus D_{k_n}, a(t_{k_1}, \ldots, t_{k_n})\rangle \,\}$ od
                                 ]|
    fi ;
    do $H \neq Z(t) \longrightarrow$|[ $H := Z(t)$;
                          for all $\langle it, D, ot\rangle \in Z(t)$
                          do for all $p \in Pos(t)$
                              do for all $r \in R \;\wedge\; S_D\langle r, p\rangle$ is acyclic
                                  do if $ot \overset{\langle r,p\rangle}{\not\Longrightarrow} \longrightarrow$ skip
                                      ▯ $ot \overset{\langle r,p\rangle}{\Longrightarrow} ot' \longrightarrow Z(t) := Z(t) \cup \{\, \langle it, D \otimes \langle r, p\rangle, ot'\rangle \,\}$
                                      fi
                                  od
                              od
                          od
                        ]|
    od;
    return $Z(t)$
   ]|;
  func $Trim\,(t$ : Term, $t_g$ : SetOfTerms): SetOfTriples
   |[ var $Z(t)$: SetOfTriples; $i : \mathbb{N}$;
    $Z(t) := \{\, \langle it, D, ot\rangle \in W(t) \mid ot \in t_g \,\}$;
    if $t :: a \longrightarrow$ skip
    ▯ $t :: a(t_1, \ldots, t_n) \longrightarrow$ for all $1 \le i \le n$ do $Z(t_i) := Trim(t_i, \{\, it|_i \mid \langle it, D, ot\rangle \in Z(t) \,\})$ od
    fi ;
    return $Z(t)$
   ]|;
  $W(t) := Generate(t)$;   (* main program *)
  $V(t) := Trim(t, \{g\})$
]|.
```

Figure 3: A naive algorithm for generating the local rewrite sequences of an input term

Node	Triples	Cardinality
c	$\{\langle c, \varepsilon, c\rangle,\ \langle c, r_6, d\rangle,\ \langle c, r_6 r_5, r\rangle\}$	3
d	$\{\langle d, \varepsilon, d\rangle,\ \langle d, r_5, r\rangle\}$	2
i	$\{\langle i(d), \varepsilon, i(d)\rangle,\ \langle i(r), \varepsilon, i(r)\rangle,\ \langle i(d), \langle r_5, 1\rangle, i(r)\rangle,\ \langle i(r), \langle r_7, 1\rangle, i(d)\rangle,$ $\langle i(d), \langle r_5, 1\rangle r_4, r\rangle,\ \langle i(r), r_4, r\rangle,\ \langle i(d), \langle r_5, 1\rangle r_4 r_7, d\rangle,\ \langle i(r), r_4 r_7, d\rangle\}$	8
$+$	$\{\langle +(d, r), r_2, r\rangle,\ \langle +(c, r), \langle r_6, 1\rangle r_2, r\rangle,\ \langle +(d, d), \langle r_5, 2\rangle r_2, r\rangle,$ $\langle +(d, i(r)), \langle r_4, 2\rangle r_2, r\rangle,\ \ldots\ \}$	2775

Figure 4: The sets of (trimmed) triples $W(t)$ for each node in the term $+(c, i(d))$

the naive algorithm can be enormous, even for a simple rewrite system and input tree. Fortunately, a reduction in the number of rewrite sequences that need to be considered is possible. This reduction is based on an equivalence relation on rewrite sequences. The equivalence relation is based on the observation that rewrite sequences can be transformed into *permuted* sequences of a certain form, called *normal form.*

Definition 3.1 *Permutations*

Rewrite sequences τ and τ' are permutations of each other for term t, denoted $\tau \cong_t \tau'$, iff all elements in $\bar{\tau}$ and $\bar{\tau}'$ have the same cardinality, and $t \xrightarrow{\tau} t' \iff t \xrightarrow{\tau'} t'$ for all t'. $\triangle$

A permutation defines an equivalence relation on rewrite sequences. Note that all permutations of a rewrite sequence have the same cost. This is a stipulation for our approach. If we use a cost function that does not satisfy this property (e.g., if the cost of an instruction includes the number of registers that are free at a given moment), then only considering normal-form rewrite sequences will lead to valid rewrite sequences being discarded. This property is therefore a restriction on the cost function. Because all permuted rewrite sequences yield the same result for term t (cf. Definition 3.1) and have the same cost, we only need to consider rewrite sequences in normal form. Normal-form rewrite sequences consist of consecutive subsequences such that each subsequence can be applied to a sub-term of t. We use the concept of permutations to determine whether decorations are equivalent or not. Decorations are said to be equivalent if and only if their corresponding rewrite sequences are permutations of each other.

Definition 3.2 *Decoration equivalence*

The decorations $D(t)$ and $D'(t)$ are equivalent, denoted by $D(t) \equiv D'(t)$, iff $S_D(t)$ and $S_{D'}(t)$ are permutations of each other, i.e. $S_D(t) \cong_t S_{D'}(t)$. $\triangle$

Example 3.3 The decorations $D(t)$ and $D'(t)$ shown in Figure 2 are equivalent because $S_D(t) \cong_t S_{D'}(t)$. (See also Example 2.14.) $\triangle$

We can define an ordering relation $\prec$ on equivalent decorations. The intuitive idea behind this ordering is that $D(t) \prec D'(t)$ for equivalent decorations $D(t)$ and $D'(t)$ if their associated local rewrite sequences for t are identical, except for one rewrite rule r that can be moved from a higher position q in $D'(t)$ to a lower position p in $D(t)$.

Definition 3.4 *Precedence relation*

For term t and equivalent decorations $D(t)$ and $D'(t)$ the precedence relation $\prec$ is defined as $D(t) \prec D'(t)$ iff $\exists\, p, q \in Pos(t)$, such that q is a proper prefix of p, and the following holds:

- $\forall\, s \neq p, q : L_D(t|_s) = L_{D'}(t|_s)$
- $\exists\, r \in L_D(t|_p) \cap L_{D'}(t|_q) : L_D(t|_p) \setminus r = L_{D'}(t|_p) \ \wedge\ L_D(t|_q) = L_{D'}(t|_q) \setminus r$

$\triangle$

The transitive closure of $\prec$ is denoted $\prec^+$. It follows quite easily that $\prec^+$ is a strict partial order (i.e. irreflexive, anti-symmetric and transitive) on equivalent decorations (under $\equiv$). The minimal elements under $\prec^+$ constitute a special class of decorations. These decorations are said to be in normal form. Normal forms need not be unique as $\prec^+$ does not need to have a least element.

Definition 3.5 *Normal-form decoration*

A decoration $D(t)$ of a term t is in normal form iff $\neg\,(\exists\, D'(t) : D'(t) \prec^+ D(t))$. $\triangle$

We let $NF(t)$ denote the set of decorations of t that are in normal form.

Example 3.6 In Figure 2, we have $D(t) \prec D'(t)$ because rewrite rule r_5, as well as r_4 and r_7, associated with the root position of t in $D'(t)$ can be moved to lower positions. In contrast, the local rewrite rules in $D(t)$ are all applied to the root position of the sub-term with which they are associated. Hence they cannot be moved to lower positions, and $D(t)$ is in normal form. $\triangle$

The following theorem allows us to consider only normal-form decorations of a term t.

Theorem 3.7 *Normal-form existence*

For rewrite sequence τ and term t, we have

$$t \xrightarrow{\tau} \ \Rightarrow\ (\exists\, D(t) \in NF(t) : S_D(t) \cong_t \tau)$$

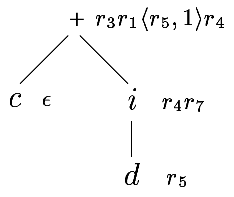
The consequence of the existence of a normal-form decoration is that the local write sequence at each position must always begin with a rewrite step that is applied to the root of the subtree rooted at that position.

Lemma 3.8

For all decorations $D(t) \in NF(t)$ and $p \in Pos(t) : L_D(t\,|_p) \neq \varepsilon \Rightarrow L_D(t\,|_p) = \langle r, \varepsilon \rangle\, \tau$, for some $r \in R$ and rewrite sequence τ.

Of course, with the exception of the leading rewrite step, local rewrite sequences in normal-form decorations may contain rewrite steps that are applied to positions other than the root.

Example 3.9 Continuing on with our running example, the term $+(c, i(d))$ has the following normal-form decoration:



△

The definitions of the inputs and outputs of normal-form decorations are the same as Definitions 2.15 and 2.16 (resp.), with the extra restriction that $D(t) \in NF(t)$. As before, we now give an algorithm that will compute all the sets of triples (containing inputs, decorations and outputs) of an input term. Instead of using all decorations, however, we now consider only normal-form decorations. Actually, we give two algorithms.

In the first algorithm, shown in Figure 5, all triples are generated, and then 'filtered' to remove those that contain decorations that are not in normal form. The filtering is carried out by the function *Checknf*. This function simply checks every triple with every other triple. If the two triples contain equivalent decorations, then the decoration with the highest precedence is discarded. The resulting set of triples, $W(t)$, contains all rewrite sequences of the input term that correspond to normal-form decorations. After all the triples have been filtered, the input term is trimmed as before. The obvious drawback of this approach is that we first generate all the triples, which for reasons of space, is just what we wish to avoid.

In the second algorithm, shown in Figure 6, we check whether a triple contains a normal-form decoration on-the-fly. This check is carried out in two places in the function *Generate*; once when we initialize, and once when we have found a rewrite step that rewrites the output. As in the first algorithm, the function *Checknf* is used to remove the unwanted triple. In the worst case, checking whether decorations are in normal form is quadratic in the number of triples in both algorithms. The line in Figure 6 adorned with asterisks will be referred to in the next section.

4 Strong normal-form decorations

The idea behind *strong* normal form is to identify permutations of rewrite sequences that arise because of the substitution of variables. In the strong normal form, we do not permit positions in sub-terms of the expression tree that have matched variables in an applied rewrite rule to be rewritten again. These positions are said to have become non-rewriteable. By avoiding rewriting these positions, we avoid generating local rewrite sequences that are simply permutations of each other. All definitions in this section are with respect to a WTRS $((\Sigma, V), R, C)$.

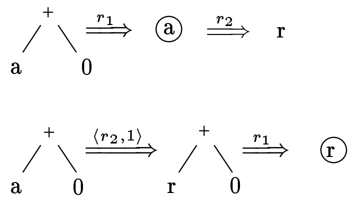
Definition 4.1 *Variable positions*

The set *VP* of variable positions of a term $t \in T_\Sigma(V)$ is defined as the set of positions at which a variable occurs. In other words, $VP(t) = \{\, p \in Pos(t) \mid t|_p \in V \,\}$. △

We say that each position in a term is either *rewriteable* or *non-rewriteable*. A rewriteable position is a position in a term at which a rewrite rule may be applied. A rewrite rule may not be applied to a non-rewriteable position. If a term is rewritten using a rewrite rule that does not contain a variable, then the rewriteability of the positions in the rewritten term does not change. If the rewrite rule does contain a variable, then the positions in the term substituted for the variable become non-rewriteable.

Example 4.2 Consider the TRS with symbols $\{+, a, 0\}$, corresponding ranks $\{2, 0, 0\}$, variable $\{x\}$, input term $+(a, 0)$, goal term r, and rules $\{r_1\colon +(x, 0) \longrightarrow x,\ r_2\colon a \longrightarrow r\}$.

The first rewrite sequence shown below is *not* in strong normal form because rule r_2 is applied to a non-rewriteable position (indicated by a circled node). This non-rewriteable position is caused by the application of rule r_1. The second rewrite sequence is in strong normal form.



△

Definition 4.3 *Rewriteable positions*

The set RP_t of rewriteable positions in a term t after the application of the rewrite sequence τ and rewrite step $\langle t_1 \longrightarrow t_2, p\rangle$ is defined as:

- $RP_t(\varepsilon) = Pos(t)$
- $RP_t(\tau\langle t_1 \longrightarrow t_2, p\rangle) =$
$$(RP_t(\tau) - Pos(t'|_p)) \cup Pos(t''|_p) - \{Pos(t''|_{p\cdot q}) \mid q \in VP(t_2)\}$$

where $t \overset{\tau}{\Longrightarrow} t' \xrightarrow{\langle t_1 \longrightarrow t_2, p\rangle} t''$. △

In Figure 7 we depict how rewriteable positions are computed. Assume that we have some rewrite sequence $t \overset{\tau}{\Longrightarrow} t'$. If the left-hand side of the rule $t_1 \longrightarrow t_2$ matches a sub-term at position p in t', then we can rewrite t' into t''. We do this by replacing the matched sub-term in t' (shown lightly shaded in the term t' in Figure 7) by the right-hand side t_2 (shown lightly shaded in the term t''). If t_1 also contains variables, then we must substitute for

```
|[ con ((Σ, V), R): TermRewriteSystem;
       t, g : Term;
   var W(t), V(t): SetOfTriples;
   func Generate(t : Term): SetOfTriples   (* as before *)
   func Trim(t : Term, t_g : SetOfTerms): SetOfTriples   (* as before *)
   func Filter(Z :SetOfTriples): SetOfTriples
    |[ for all ⟨it, D, ot⟩ ∈ Z
      do Z := Checknf(Z \ { ⟨it, D, ot⟩ }, ⟨it, D, ot⟩) od;
      return Z
    ]|;
   func Checknf(Z :SetOfTriples, ⟨it, D, ot⟩ : Triple): SetOfTriples
    |[ var exit: Bool;
      exit := false;
      for all ⟨it′, D′, ot′⟩ ∈ Z ∧ ¬ exit
      do if D ≡ D′ ∧ D ≺ D′ ⟶|[ exit := true;
                                 Z := (Z \ { ⟨it′, D′, ot′⟩ }) ∪ { ⟨it, D, ot⟩ }
                               ]|
          ▯ D ≡ D′ ∧ D′ ≺ D ⟶ exit := true
          ▯ D ≢ D′ ⟶ skip
         fi
      od;
      if ¬ exit ⟶ Z := Z ∪ { ⟨it, D, ot⟩ } ▯ exit ⟶ skip fi ;
      return Z
    ]|;
   W(t) := Filter(Generate(t));   (* main program *)
   V(t) := Trim(t, {g})
]| .
```

Figure 5: An algorithm that filters out non-normal-form rewrite sequences

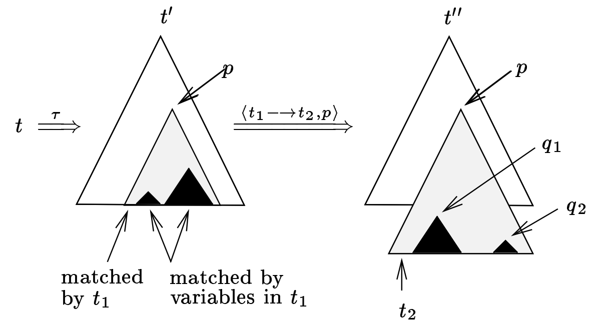


Figure 7: Computing the rewriteable positions in a term after the application of $\langle t_1 \longrightarrow t_2, p\rangle$

these variables in t_2 (the matching sub-terms are shown in black in t' and t'').

In Definition 4.3, the set of rewriteable positions in t'' consists of the rewriteable positions in t' (given by $RP_t(\tau)$), minus the positions in the sub-term that has been matched by t_1 ($Pos(t'|_p)$), plus the positions in the sub-term t_2 that replaced t_1 ($Pos(t''|_p)$), and minus the positions in the sub-terms that are substituted for the variables (if any) in t_2 ($\{Pos(t''|_{p\cdot q}) \mid q \in VP(t_2)\}$).

Example 4.4 Consider the running example again, and let the input term be $+(c, i(r))$. Initially, the rewriteable positions in t are given by $RP_t(\varepsilon) = \{\varepsilon, 1, 2, 2{\cdot}1\}$. If we now apply the rewrite rule $\langle r_4, 2\rangle$, which does not involve a variable, then we generate the term $t'' = +(c, r)$ with rewriteable positions:

$$\begin{aligned} RP_t(\langle r_4, 2\rangle) &= (RP_t(\varepsilon) - Pos(t|_2)) \cup Pos(t''|_2) - \emptyset \\ &= (\{\varepsilon, 1, 2, 2{\cdot}1\} - \{2, 2{\cdot}1\}) \cup \{2\} \\ &= \{\varepsilon, 1, 2\} \end{aligned}$$

This says that all the positions in the new term $+(c, r)$ are rewriteable.

We now apply the rewrite rule $\langle r_3, \varepsilon\rangle$ and generate $t'' = +(r, c)$. Note that $t' = +(c, r)$ here. The rewriteable positions in the term t'' are:

$$\begin{aligned} RP_t(\langle r_4, 2\rangle\langle r_3, \varepsilon\rangle) &= (RP_t(\langle r_4, 2\rangle) - Pos(t'|_\varepsilon)) \cup \\ &\quad Pos(t''|_\varepsilon) - \\ &\quad \{Pos(t''|_q) \mid q = 1, 2\} \\ &= (\{\varepsilon, 1, 2\} - \{\varepsilon, 1, 2\}) \cup \\ &\quad \{\varepsilon, 1, 2\} - \{1, 2\} \\ &= \{\varepsilon\} \end{aligned}$$

Because only the root position in the term $+(r, c)$ is rewriteable, and $+(r, c)$ does not correspond to the left-hand side of any rule, we can proceed no further. We cannot therefore reach the goal term. The rewrite sequence that we were not permitted to generate could have been:

```
|[ con $((\Sigma, V), R)$: TermRewriteSystem;
       $t, g$ : Term;

  var $W(t), V(t)$: SetOfTriples;

  func $Generate\,(t$ : Term): SetOfTriples
   |[ var $H, Z(t)$: SetOfTriples; $i : \mathbb{N}$;
    $H := Z(t) := \emptyset$;
    if $t :: a \longrightarrow Z(t) := \{\, \langle t, D_\varepsilon, t\rangle \,\}$;
     $[\!]$ $t :: a(t_1, \ldots, t_n) \longrightarrow$|[ for all $1 \le i \le n$ do $Z(t_i) := Generate(t_i)$ od;
                                   (* Let $O(t_i) = \{\, ot_{k_i} \mid \langle it_{k_i}, D_{k_i}, ot_{k_i}\rangle \in Z(t_i) \,\}$ *)
                                   for all $(t_{k_1}, \ldots, t_{k_n}) \in O(t_1) \times \ldots \times O(t_n)$
                                   do $Z(t) := Checknf(Z(t), \langle a(t_{k_1}, \ldots, t_{k_n}), D_{k_1} \oplus \ldots \oplus D_{k_n}, a(t_{k_1}, \ldots, t_{k_n})\rangle)$ od
                                  ]|
    fi ;
    do $H \neq Z(t) \longrightarrow$|[ $H := Z(t)$;
                        for all $\langle it, D, ot\rangle \in Z(t)$
        (* * * *)        do for all $p \in Pos(t) \;\wedge\; (L_D(t|_\varepsilon) = \varepsilon \;\Rightarrow\; p = \varepsilon)$        (* * * *)
                            do for all $r \in R \;\wedge\; S_D\langle r, p\rangle$ is acyclic
                                do if $ot \overset{\langle r,p\rangle}{\not\Longrightarrow} \longrightarrow$ skip
                                   $[\!]$ $ot \overset{\langle r,p\rangle}{\Longrightarrow} ot' \longrightarrow Z(t) := Checknf(Z(t), \langle it, D \otimes \langle r,p\rangle, ot'\rangle)$
                                   fi
                                od
                            od
                        od
                       ]|
    od;
    return $Z(t)$
   ]|;

  func $Checknf(Z$ :SetOfTriples, $\langle it, D, ot\rangle$ : Triple): SetOfTriples   (* as before *)

  func $Trim\,(t$ : Term, $t_g$ : SetOfTerms): SetOfTriples   (* as before *)

  $W(t) := Generate(t)$;   (* main program *)
  $V(t) := Trim(t, \{g\})$
]| .
```

Figure 6: An algorithm that computes the normal-form rewrite sequences

$$+(c,r) \xRightarrow{r_3} +(r,c) \xRightarrow{\langle r_7,1\rangle} +(d,c) \xRightarrow{r_1} i(d)$$
$$\xRightarrow{\langle r_5,1\rangle} i(r) \xRightarrow{r_4} r$$

Normal-form rewrite sequences do not apply the commutativity rule until first the 'children' have been rewritten. In this case, this means:

$$+(c,r) \xRightarrow{\langle r_7,2\rangle} +(c,d) \xRightarrow{r_3} +(d,c)\ldots \textit{ as before}$$

△

As a convenience, we now define a boolean function Permitted_t that determines whether rules in a rewrite sequence are only applied at rewriteable positions in a term t.

Definition 4.5 *Permitted*

Given the rewrite sequence τ and term t, the predicate Permitted_t is true if each rewrite rule r in τ is applied at a rewriteable position p, and false otherwise. Formally,

- $\text{Permitted}_t(\varepsilon) = \text{true}$
- $\text{Permitted}_t(\tau\langle r,p\rangle) = p \in RP(\tau) \ \wedge \ \text{Permitted}_t(\tau)$

△

Definition 4.6 *Strong-normal-form decoration*

A normal-form decoration $D(t)$ is in strong normal form iff $\text{Permitted}_t(L_D(t\,|_p))$ is true, for all $p \in Pos(t)$. △

We let $SNF(t)$ denote the set of decorations of t that are in strong normal form.

The following theorem means that we only need to consider strong-normal-form decorations of t.

Theorem 4.7 *Strong normal-form existence*

For rewrite sequence τ and term t, we have

$$t \xRightarrow{\tau} \ \Rightarrow\ (\exists\, D(t) \in SNF(t) : S_D(t) \cong_t \tau)$$

The definitions of the inputs and outputs of strong-normal-form decorations are the same as Definitions 2.15 and 2.16 (resp.), with the extra restriction that $D(t) \in SNF(t)$.

We can now give an algorithm that will compute the strong-normal-form triples of an input term. Actually, this algorithm is almost identical to the 'normal-form' algorithm shown in Figure 6. The only difference is the extra check for the 'strong' condition (i.e. whether a position is rewriteable). This extra check is carried out in the line adorned with asterisks in Figure 6. The new check is shown below.

$$p \in RP_t(S_D) \ \wedge \ (L_D(t|_\varepsilon) = \varepsilon \ \Rightarrow\ p = \varepsilon)$$

Example 4.8 Completing our running example, we can now give the sets of trimmed triples, $V(t)$, containing only permissible (local) rewrite sequences for the input term $+(c, i(d))$. This is shown in the table in Figure 8. Notice that in this table that there are 3 strong-normal-form rewrite sequences that rewrite the root node $+$. This should be compared to the number of 'naive' rewrite sequences, which is 2775 (see Example 2.17). △

To guarantee termination of this algorithm the length of each local rewrite sequence must be finite. A TRS that has this property is referred to as *finite*, and one that does not as *infinite*. More specifically, a TRS is finite if and only if $L_D(t|_p)$ is finite for all $D(t) \in SNF(t)$, $t \in T_\Sigma(V)$ and $p \in Pos(t)$.

Intuitively, infinite TRSs occur because the right-hand side of a rewrite rule can be more complex than the left-hand side. In that case, sequences can continue indefinitely.

Example 4.9 Consider the TRS with rules:

$$\{\ r_1 : c \longrightarrow m(c),\quad r_2 : m(c) \longrightarrow a,\quad r_3 : m(a) \longrightarrow r\ \}$$

The TRS is infinite because we can generate the following local rewrite sequence for the term c:

$$c \xRightarrow{r_1} m(c) \xRightarrow{\langle r_1,1\rangle} m(m(c)) \xRightarrow{\langle r_1,1\cdot1\rangle} \cdots$$

A successful rewrite sequence for this input term involves only 2 applications of rule 1, as shown below:

$$c \xRightarrow{r_1} m(c) \xRightarrow{\langle r_1,1\rangle} m(m(c)) \xRightarrow{\langle r_2,1\rangle} m(a) \xRightarrow{r_3} r$$

△

The maximum length of local rewrite sequences in a finite TRS may not be bounded. In that case the length will be dependent on the input term.

Example 4.10 Consider the TRS with rules:

$$\{\ r_1 : m(+(c,X)) \longrightarrow m(X),\quad r_2 : m(r) \longrightarrow r\ \}$$

where r is the goal term. Local rewrite sequences for this TRS will be finite in length, but unbounded. For example:

- $m(+(c,r)) \xRightarrow{r_1} m(r) \xRightarrow{r_2} r$
- $m(+(c,+(c,r))) \xRightarrow{r_1} m(+(c,r)) \xRightarrow{r_1} m(r) \xRightarrow{r_2} r$
- $m(+(c,+(c,+(c,r)))) \xRightarrow{r_1} m(+(c,+(c,r))) \xRightarrow{r_1} m(+(c,r)) \xRightarrow{r_1} m(r) \xRightarrow{r_2} r$

△

Pelegri-Llopart and Graham found in [18] that a TRS $((\Sigma, V), R)$ is finite if for all $(t, t') \in R$ one of the following conditions holds:

1. $Var(t) = \emptyset$ and $t' \in \Sigma_0$
2. $t = a(t_1, \ldots, t_n)$ and $t' = b(t_1, \ldots, t_n)$ for $n \geq 0$ and $a, b \in \Sigma_n$

Node	Triples	Cardinality
c	$\{\langle c,\varepsilon,c\rangle,\ \langle c,r_6,d\rangle,\ \langle c,r_6r_5,r\rangle\}$	3
d	$\{\langle d,\varepsilon,d\rangle,\ \langle d,r_5,r\rangle\}$	2
i	$\{\langle i(d),\varepsilon,i(d)\rangle,\ \langle i(r),\varepsilon,i(r)\rangle,\ \langle i(r),r_4,r\rangle,\ \langle i(r),r_4r_7,d\rangle\}$	4
$+$	$\{\langle +(d,r),r_2,r\rangle,\ \langle +(r,d),r_3r_2,r\rangle,\ \langle +(c,d),r_3r_1\langle r_5,1\rangle r_4,r\rangle\}$	3

Figure 8: The sets of trimmed strong-normal-form triples $V(t)$ for each node in the term $+(c,i(d))$

3. $t = a(t_1,\ldots,t_n)$ and $t' = a(t_{\pi(1)},\ldots,t_{\pi(n)})$ with π a permutation on $[1,n]$

4. $t = f(x,t'')$ and $t' = x$ with $Var(t'') = \emptyset$

This result has been confirmed, in a somewhat different context, by Kurtz [15].

5 Conclusions

In this work we have described how term rewrite systems can be applied to code generation. We have provided a theoretical framework for a pattern matcher in a code generator, and we have developed in a systematic way pattern-matching algorithms that rewrite a given input term into a given goal term. In code generation, these rewrite sequences correspond to code that is generated.

We began with a naive algorithm that determines all possible rewrite sequences. By defining a normal form, we could improve the algorithm and avoid generating many redundant permutations of the rewrite sequences (the on-the-fly version of the algorithm). A second improvement involved recognising strong normal form rewrite sequences. This improvement meant that permutations caused by the action of variables in the term rewrite system could also be eliminated.

Term rewrite systems provide a more powerful formalism in code generation than the more popular tree grammars. The naive algorithm has been implemented, and work is nearing completion on the normal-form and strong-normal form algorithms. This has demonstrated the correctness of the approach, and has allowed experimentation. There are a number of directions for future research:

- Analyse the performance and complexity of the pattern-matching algorithms.
- Determine *a priori* whether a term rewrite system is finite.
- Develop term rewrite systems for real machines.
- Investigate which parts of the pattern-matching algorithm can be pre-computed.
- Investigate whether code optimisation and register allocation can be expressed in terms of a term rewrite system.

References

[1] A. V. Aho, M. Ganapathi and S. W. K. Tjiang. Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems*, Volume 11, Number 4, pages 491–516, October 1989.

[2] A. Balachandran, D. M. Dhamdhere and S. Biswas. Efficient retargetable code generation using bottom-up tree pattern matching. *Computer Languages*, Volume 15, Number 3, pages 127–140, 1990.

[3] D. R. Chase. An improvement to bottom-up tree pattern matching. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 168–177, Munich, Germany, January 1987.

[4] T. W. Christopher, P. J. Hatcher and R. C. Kukuk. Using dynamic programming to generate optimised code in a Graham-Glanville style code generator. *Proceedings of the ACM SIGPLAN 1984 Symposium on Compiler Construction, ACM SIGPLAN Notices*, Volume 19, Number 6, pages 25–36, June 1984.

[5] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen (editor), *Handbook of Theoretical Computer Science (Vol. B: Formal Models and Semantics)*, Chapter 6, pages 245–320. Elsevier Science Publishers B.V., 1990.

[6] H. Emmelmann. Code selection by regularly controlled term rewriting. In R. Giegerich and S. L. Graham (editors), *Code generation—concepts, tools, techniques*, Workshops in Computing Series, pages 3–29. Springer-Verlag, New York–Heidelberg–Berlin, 1991.

[7] H. Emmelmann, F. W. Schröer and R. Landwehr. BEG—a generator for efficient back ends. *ACM SIGPLAN Notices*, Volume 24, Number 7, pages 246–257, July 1989.

[8] C. W. Fraser, D. R. Hanson and T. A. Proebsting. Engineering a simple, efficient code-

generator generator. *ACM Letters on Programming Languages and Systems*, Volume 1, Number 3, pages 213–226, September 1992.

[9] M. Ganapathi and C. N. Fischer. Affix grammar driven code generation. *ACM Transactions on Programming Languages and Systems*, Volume 7, Number 4, pages 560–599, October 1985.

[10] R. Giegerich. Code selection by inversion of order-sorted derivors. *Theoretical Computer Science*, Volume 73, pages 177–211, 1990.

[11] R. Giegerich and K. Schmal. Code selection techniques: pattern matching, tree parsing, and inversion of derivors. In H. Ganzinger (editor), *Proc. 2nd European Symp. on Programming*, Volume 300 of *Lecture Notes in Computer Science*, pages 247–268. Springer-Verlag, New York–Heidelberg–Berlin, 1988.

[12] R. S. Glanville and S. L. Graham. A new method for compiler code generation. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 231–240, Tucson, Arizona, January 1978.

[13] K. J. Gough. Bottom-up tree rewriting tool MBURG. *ACM SIGPLAN Notices*, Volume 31, Number 1, pages 28–31, January 1996.

[14] C. Hemerik and J.-P. Katoen. Bottom-up tree acceptors. *Science of Computer Programming*, Volume 13, pages 51–72, January 1990.

[15] S. Kurtz. Narrowing and basic forward closures. Technical report 5, Technische Fakultät, Universität Bielefeld, Germany, 1992.

[16] A. Nymeyer, J.-P. Katoen, Y. Westra and H. Alblas. Code generation = $A^* +$ BURS. In T. Gyimóthy (editor), *Compiler Construction*, Volume 1060 of *Lecture Notes in Computer Science*, pages 160–176. Springer-Verlag, New York–Heidelberg–Berlin, April 1996.

[17] E. Pelegrí-Llopart. *Rewrite systems, pattern matching, and code generation.* Ph.D. thesis, University of California, Berkeley, December 1987.

[18] E. Pelegrí-Llopart and S. L. Graham. Optimal code generation for expression trees: An application of BURS theory. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 294–308, San Diego, CA, January 1988.