ELSEVIER

Theoretical Computer Science 238 (2000) 439–464

Theoretical Computer Science

www.elsevier.com/locate/tcs

Pattern-matching algorithms based on term rewrite systems

Joost-Pieter Katoen[a], Albert Nymeyer[a,b,*]

[a] *Department of Computer Science, University of Twente, P.O. Box 217, 7500 AE Enschede, Netherlands*

[b] *Software Verification Research Center, The University of Queensland, St Lucia, QLD 4072, Australia*[1]

Received May 1997; revised April 1998
Communicated by J. Staples

Abstract

Automatic code generators often contain pattern matchers that are based on tree grammars. In this work we generalise this approach by developing pattern matchers that are based on more powerful term rewrite systems. A pattern matcher based on a 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, the vast majority of sequences are in fact redundant. This redundancy is caused by the fact that many rewrite sequences are permutations of each other. A theory and a series of algorithms are systematically developed that identify and remove two types of redundant rewrite sequences. These algorithms terminate if rewrite sequences do not diverge.

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

* Corresponding author.
E-mail addresses: katoen@cs.utwente.nl (J.-P. Katoen), anymeyer@cse.unsw.edu.au (A. Nymeyer)

[1] Part of this work was carried out while this author was on study leave at this address.

0304-3975/00/$ - see front matter
PII: S0304-3975(00)00041-4

a (sub)term that matches the term on the left-hand side of a rewrite rule. This pattern is replaced by an instantiation of 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 the 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, each resulting in a different normal form. It is not terminating because, in code-generation grammars, there are invariably rules (for example, commutativity) that lead to cycles. 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 is not described in this work. The pattern selector chooses a least-cost sequence of rewrite rules that will transform (rewrite) the given expression tree into the given goal. The goal is (invariably) a single node, and is the place where the result of the calculation that is represented by the expression tree is stored.

In a nutshell, the main problem in code generation is the typically 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 a least-cost instruction sequence.

The costs of rewrite rules and sequences only play an "indirect" role in this work as we are only interested in enumerating all the possible rewrite sequences, not on selecting a least-cost one (for this we refer the reader to Nymeyer and Katoen [20]). However, the role of costs in code generation is so important that we have included them in the definition of a term rewrite system (for example), and we occasionally refer to them. In the course of this work we make certain assumptions about the costs of rewrite sequences: for example, do two rewrite sequences that only differ in the order that rules are applied always have the same cost? The costs therefore play a role even in pattern matching.

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. In this work we will not show machine instructions, nor discuss other aspects of code generation like register and memory allocation.

In the past, there have been only two notable applications of term rewrite systems to code generation. Emmelmann [8] 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 [12, 13], who has carried out extensive, mainly theoretical research into this approach. Emmelmann's ambitious work pursued this approach further, and resulted in a large complex system that would be difficult to implement. A second, more successful application of term rewrite systems was that by Pelegri-Llopart and Graham [23, 24]. 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 [11, 14], but LR grammars were found to be too restrictive and cumbersome, and the method fell out of favour. It made way for tree grammars that use either a top-down [1, 4] or bottom-up [2, 3, 9, 10, 15–17] traversal strategy. Tree grammars using a bottom-up traversal strategy are used extensively today. 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 not only a theory of pattern matching in code generation based on term rewrite systems, but also to systematically develop an (intelligent) pattern-matching algorithm. In Section 2 we will define the basic theory, and we will define a weighted term rewrite system. For a more elaborate treatment of term rewrite systems we refer the reader to Dershowitz and Jouannaud [7]. Using the basic theory, we present in Section 3 a (naive) pattern-matching algorithm that generates all rewrite sequences for a given expression tree and goal.

In Section 4 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 *innermost normal* (IN) form.[2] Removing sequences that are not in IN form can be done after all rewrite sequences are first generated, or on-the-fly. We present algorithms that implement both techniques in Section 5. In Section 6 we see that there is another form of redundancy in the IN 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 *weak innermost normal* (WIN) rewrite sequences.[3] An algorithm that generates rewrite sequences in WIN form is given in Section 7.

[2] In earlier work [20, 21] we referred to the innermost normal form as simply the normal form.

[3] In earlier work referred to as strong normal rewrite sequences.

An important theoretical and practical property of term rewrite systems is termination. A non-terminating term rewrite system generates rewrite sequences that are of infinite length. We discuss this issue in Section 8. Finally in Section 9 we present our conclusions.

2. Weighted term rewrite systems

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

Definition 1 (*Ranked alphabet*). A ranked alphabet Σ is a pair (S,r) with S a finite set and $r \in S \to \mathbb{N}$.

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 $\mathscr{V}$ is a countably infinite universe of variable symbols, and $V \subseteq \mathscr{V}$.

Definition 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 $n \geqslant 1$.

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 paths 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 3 (*Positions*). The set of positions $Pos \in T_\Sigma(V) \to \mathscr{P}(\mathbb{N}_+^*)$ of a term t is defined as
- $Pos(t) = \{\varepsilon\}$, if $t \in \Sigma_0 \cup V$,
- $Pos(a(t_1,\ldots,t_n)) = \{\varepsilon, 1 \cdot Pos(t_1),\ldots,n \cdot Pos(t_n)\}$.

A trailing ε in a position can be omitted; for example $2 \cdot 1 \cdot \varepsilon$ is written as $2 \cdot 1$. 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 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.

Additionally, for all $(t,t') \in R$, we have the constraints: (1) $t' \neq t$, (2) $t \notin V$, (3) $\mathrm{Var}(t') \subseteq \mathrm{Var}(t)$ and (4) variables in $\mathrm{Var}(t)$ and $\mathrm{Var}(t')$ occur only once.

Elements of R are called *rewrite rules*. An element $(t,t') \in R$ is usually written as $t \to t'$ 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 $((\Sigma, V), R)$. A WTRS is called *ground* if all left-hand sides of rewrite rules are ground terms.

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 third constraint says that no new variables may be introduced by a rewrite rule, and the last constraint that the set of rewrite rules must be *linear*. The first three constraints mean that we avoid simple infinite sequences of rewrites. The importance of the last constraint will be explained in Section 6.

We only consider finite WTRSs in this work because instruction sets of machines are finite. A finite WTRS contains finite sets of function symbols, variables and rewrite rules. The WTRS shown below will be used as a running example in this work.

Example 5. Let $((\Sigma, V), R, C)$ be a WTRS, where $\Sigma = (S, r)$, $V = \{x, y\}$, $S = \{+, i, c, d, r\}$, $r(+) = 2$, $r(i) = 1$ and $r(c) = r(d) = r(r) = 0$. 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 as follows:

$$
\begin{aligned}
R = \{\ & r_1 : +(d,c) \to i(d) \\
& r_2 : +(d,r) \to r \\
& r_3 : +(x,y) \to +(y,x) \\
& r_4 : i(r) \to r \\
& r_5 : d \to r \\
& r_6 : c \to d \\
& r_7 : r \to d\ \}
\end{aligned}
$$

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 elements of R are given in Fig 1. Examples of 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 $t|_{1\cdot 2\cdot 1} = d$.

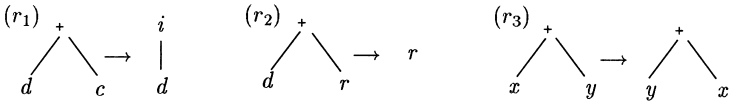


Fig. 1. The tree representations of some term rewrite rules.

Definition 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

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

$$a(t_1, \ldots, t_n)^\sigma = a(t_1^\sigma, \ldots, t_n^\sigma).$$

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 7 (*Rewrite rule equivalence*). Rewrite rules $r_1 : t_1 \to t_1'$ and $r_2 : t_2 \to t_2'$ are equivalent iff there is a bijection $\sigma \in Var(t_1) \to Var(t_2)$ such that $t_1^\sigma = t_2$ and $t_1'^\sigma = t_2'$.

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

Definition 8 (*Rewrite step*). Given the TRS $((\Sigma, V), R)$, $r : t \to t' \in R$, $t_1, t_2 \in T_\Sigma(V)$ 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 a substitution σ with $t^\sigma = t_1|_p$. We can also write $\langle r, p\rangle\, t_1 = t_2$.

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 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 $t \xrightarrow{\langle r_1,p_1\rangle} t_1 \xrightarrow{\langle r_2,p_2\rangle} \ldots t_{n-1} \xrightarrow{\langle r_n,p_n\rangle} t'$. The corresponding rewrite sequence of t is $S(t) = \langle r_1, p_1\rangle \ldots \langle r_n, p_n\rangle$. We can also write $S(t)\, t = t'$.

When convenient, we denote a rewrite sequence $S(t)$ by τ. Furthermore, 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.

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 $t \xrightarrow{\tau_2} t'$. Although rules like r_3 in Example 5 that specify commutativity lead naturally to cyclical rewrite sequences, in this work we only consider acyclic rewrite sequences. In Section 8 we consider in more detail termination properties of TRSs.

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, $\tau\backslash r$ denotes sequence τ with r deleted,[4] and $r \in \tau$ denotes that r occurs in τ. 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 10. Consider the WTRS shown in Example 5, and let $t = +(c,d)$. We can write $t \xrightarrow{\langle r_3,\varepsilon\rangle} t'$, with $t' = +(d,c)$. We can also write $\langle r_3, \varepsilon\rangle\, t = t'$. 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 11. An example of a derivation for $t = +(c, i(d))$ of length 4 is

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

and two sequences of length 7 are:

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

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

In Definition 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 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 $L_D(t|_p)$.

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

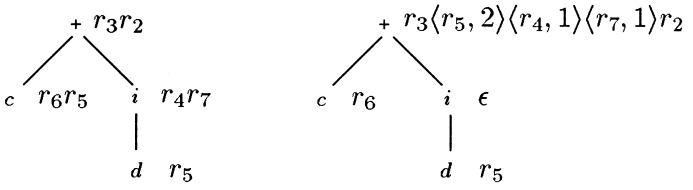


Fig. 2. Decorations $D(t)$ and $D'(t)$ of the term $+(c, i(d))$.

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

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

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

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

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.

Definition 15 (*Inputs 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 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 $1 \leqslant i \leqslant n$.

Definition 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 $I_D(t) \xrightarrow{L_D(t)} t'$.

3. An algorithm that computes all decorations

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 term t and goal term g, and the corresponding decorations, is given in Fig. 3. For convenience, in the algorithm we refer to the type T_Σ as Term and $\mathscr{P}$(Term) as SetOfTerms. This algorithm, which we refer to as the *naive* algorithm, is in fact a generalisation of bottom-up tree pattern matching methods (see, for example, Hemerik and Katoen [17]). The algorithm consists of two passes.

In the first, bottom-up pass, carried out by the recursive function *Generate*, sets of *triples*, denoted by $Z(t)$, are computed for all possible goal terms. A triple is written $\langle it, D(t), ot\rangle$, and 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 in the algorithm is defined as Term $\times$ Decoration $\times$ Term. $\mathscr{P}$(Triple) is denoted by SetOfTriples.

In the function *Generate*, the inputs and outputs at each leaf node a in t are initialised 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 initialised to $a(t_{k_1}, \ldots, t_{k_n})$, where t_{k_i} is an output of the ith child. The decoration $D_1 \oplus \cdots \oplus D_n$ 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. Note that the rewrite sequence $S_D\langle r, p\rangle$ is defined to be acyclic: this is necessary to ensure termination.

Now consider the triple do-loop in the algorithm. 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 as 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 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 $V(t)$. Under some circumstances it may be possible to trim the nodes in t while they are being generated (see, for example, [25]). We do not consider that aspect further here however.

Example 17. Given the input term $+(c, i(d))$, the sets of (trimmed) triples $V(t)$ that are generated are shown in the table in Fig. 4. Notice that the cardinality of the set $V(\text{'}+\text{'})$ is 2775. The cardinality of the corresponding set of untrimmed triples,

```
|[ con ((Σ, V), R) : TermRewriteSystem;
       t, g : Term;
  var W(t), V(t) : SetOfTriples;
  func Generate(t : Term) : SetOfTriples
   |[ var H, Z(t) : SetOfTriples; i : ℕ;
     H := Z(t) := ∅;
     if t :: a ⟶ Z(t) := {⟨t, D_ε, t⟩};
      [] t :: a(t_1, ..., t_n) ⟶
           |[ for all 1 ⩽ i ⩽ n do Z(t_i) := Generate(t_i) od;
              (* Let O(t_i) = {ot_{k_i} | ⟨it_{k_i}, D_{k_i}, ot_{k_i}⟩ ∈ Z(t_i)} *)
              for all (t_{k_1}, ..., t_{k_n}) ∈ O(t_1) × ··· × O(t_n)
              do Z(t) := Z(t) ∪ {⟨a(t_{k_1}, ..., t_{k_n}), D_{k_1} ⊕ ··· ⊕ D_{k_n}, a(t_{k_1}, ..., t_{k_n})⟩} od
           ]|
     fi;
     do H ≠ Z(t) ⟶ |[ H := Z(t);
                      for all ⟨it, D, ot⟩ ∈ Z(t)
                      do for all p ∈ Pos(t)
                         do for all r ∈ R ∧ S_D⟨r, p⟩ is acyclic
                            do if ot ⟹/⟨r,p⟩ ⟶ skip
                                 [] ot ⟹⟨r,p⟩ ot' ⟶ Z(t) := Z(t) ∪ {⟨it, D ⊗ ⟨r, p⟩, ot'⟩}
                               fi
                            od
                         od
                      od
                   ]|
     od;
     return Z(t)
   ]|;
  func Trim(t : Term, t_g : SetOfTerms) : SetOfTriples
   |[ var Z(t) : SetOfTriples; i : ℕ;
     Z(t) := {⟨it, D, ot⟩ ∈ W(t) | ot ∈ t_g};
     if t :: a ⟶ skip
      [] t :: a(t_1, ..., t_n) ⟶ for all 1 ⩽ i ⩽ n do Z(t_i) := Trim(t_i, {it|_i | ⟨it, D, ot⟩ ∈ Z(t)}) od
     fi ;
     return Z(t)
   ]|;
  (* main program *)
  W(t) := Generate(t);
  V(t) := Trim(t, {g})
]|.
```

Fig. 3. An algorithm that generates all the local rewrite sequences of an input term.

$W(\text{`}+\text{'})$, by the way, is 19 033! An example of the longest local rewrite sequence that is generated (the length is 16) is shown below:

$$+(c,i(r)) \xrightarrow{r_3} +(i(r),r) \xrightarrow{\langle r_7,1\rangle} +(i(d),c) \xrightarrow{r_3} +(c,i(d)) \xrightarrow{\langle r_6,1\rangle}$$

$$+(d,i(d)) \xrightarrow{r_3} +(i(d),d) \xrightarrow{\langle r_5,2\rangle} +(i(d),r) \xrightarrow{r_3} +(r,i(d)) \xrightarrow{\langle r_5,2\cdot 1\rangle}$$

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	$\{\langle i(d),\varepsilon,i(d)\rangle,\ \langle i(r),\varepsilon,i(r)\rangle,\ \langle i(d),\langle r_5,1\rangle,i(r)\rangle,$ $\langle i(r),\langle r_7,1\rangle,i(d)\rangle,\ \langle i(d),\langle r_5,1\rangle r_4,r\rangle,\ \langle i(r),r_4,r\rangle,$ $\langle i(d),\langle r_5,1\rangle r_4r_7,d\rangle,\ \langle i(r),r_4r_7,d\rangle\}$	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

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

$$+(r,i(r)) \xrightarrow{\langle r_7,1\rangle} +(d,i(r)) \xrightarrow{r_3} +(i(r),d) \xrightarrow{\langle r_5,2\rangle} +(i(r),r) \xrightarrow{\langle r_4,1\rangle}$$

$$+(r,r) \xrightarrow{\langle r_7,2\rangle} +(r,d) \xrightarrow{\langle r_7,1\rangle} +(d,d) \xrightarrow{\langle r_5,2\rangle} +(d,r) \xrightarrow{r_2} r$$

4. Innermost normal decorations

We saw in the previous section that the number of rewrite sequences (triples) that are generated by 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 *innermost normal* (IN) form. [5]

Definition 18 (*Permutations*). Given a term t, rewrite sequences τ and τ' are permutations of each other, 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 terms t'.

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 (for example, if the cost of an instruction includes the number of registers that are free at a given moment), then only considering IN rewrite sequences will lead to valid rewrite sequences being discarded. This property is therefore a restriction on the cost function and is necessary to keep the number of rewrite sequences manageable. Because permuted rewrite sequences yield the same result for term t (cf. Definition 18), and they have the same costs, we only need to consider rewrite sequences in IN form. IN rewrite

[5] Rewrite strategies for term rewriting systems are usually studied to prove confluency or termination. These strategies involve selecting particular subterms to rewrite next. The terms innermost and outermost are used to refer to the lowest and highest positions (resp.) in the term that can be rewritten. [7, 22].

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 19 (*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. $S_D(t) \cong_t S_{D'}(t)$.

Example 20. The decorations $D(t)$ and $D'(t)$ shown in Fig. 2 are equivalent because $S_D(t) \cong_t S_{D'}(t)$ (see also Example 14).

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 $D(t)$. We formally state this in the next definition.

Definition 21 (*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:

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

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 IN form. IN forms need not be unique as $\prec^+$ does not need to have a least element.

Definition 22 (*IN decoration*). A decoration $D(t)$ of a term t is in IN form iff $\neg(\exists D'(t) : D'(t) \prec^+ D(t))$.

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

Example 23. In Fig. 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 of t in $D(t)$. 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 IN form.

The following theorem allows us to consider only IN decorations of a term t, and not the entire universe of decorations of t.

Theorem 24 (IN decoration existence). *Given a rewrite sequence* τ *and term* t, *we have that* $t \stackrel{\tau}{\Longrightarrow} \Rightarrow (\exists D(t) \in IN(t)\colon S_D(t) \cong_t \tau)$.

Proof. Let τ be an arbitrary rewrite sequence and t some term such that $t \stackrel{\tau}{\Longrightarrow}$. A simple decoration $D_0(t)$ corresponding to τ can be obtained by decorating the root of t with τ and all other sub-terms with the empty rewrite sequence. Suppose $D_0(t)$ is not in IN form. We informally describe a procedure to obtain from $D_0(t)$ an equivalent decoration that is in IN form. The decoration $D_0(t)$ can be modified into $D_1(t)$ by moving a single rewrite rule from a higher position in t to a lower position in t, so that $S_{D_0}(t) \cong_t S_{D_1}(t)$. This procedure can be repeated, until no rewrite rules can be moved to a lower position. The procedure must terminate successfully as τ is finite, at which time there cannot be a decoration $D'(t)$ such that $D'(t) \prec D_{n+1}(t)$. The result is a chain of decorations $D_0(t), D_1(t), D_2(t)$, etc. so that $D_{n+1}(t) \prec D_n(t)$, for all $n \geqslant 0$. By construction, the last obtained decoration is a minimal element under $\prec^+$. □

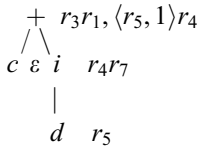
The consequence of the existence of an IN decoration is that the local rewrite sequence at each position must always begin with a rewrite step that is applied to the root of the subtree rooted at that position.

Lemma 25. *For all decorations* $D(t) \in IN(t)$ *and* $p \in Pos(t)$, *if the local rewrite sequence* $L_D(t|_p) \neq \varepsilon$ *then* $L_D(t|_p) = \langle r, \varepsilon \rangle \tau$, *for some* $r \in R$ *and rewrite sequence* τ.

Proof (*by contradiction*). Let us assume $D(t) \in IN(t)$ and for some $p \in Pos(t)$, $L_D(t|_p) = \langle r, q \rangle \tau$ with $q = n.q'$, $n \in \mathbb{N}_+$. Let $D(t)$ be identical to $D'(t)$ with the exception that $L_{D'}(t|_p) = \tau$ and $L_{D'}(t|_{p.n}) = L_D(t|_{p.n})\langle r, q' \rangle$. By construction $D'(t) \prec D(t)$, contradicting that $D(t) \in IN(t)$. □

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

Example 26. Continuing on with our running example, the term $+(c, i(d))$ has the following IN decoration:



The definitions of the inputs and outputs of IN decorations are the same as Definitions 15 and 16 (resp.), with the extra restriction $D(t) \in IN(t)$.

```
|[ 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 := Checkin(Z\{⟨it, D, ot⟩}, ⟨it, D, ot⟩) od;
     return Z
   ]|;
  func Checkin(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
   ]|;
  (* main program *)
  W(t) := Filter(Generate(t));
  V(t) := Trim(t, {g})
]| .
```

Fig. 5. An algorithm that generates all rewrite sequences, and filters out those that are not in IN form.

5. Algorithms that compute IN decorations

We now give two algorithms that compute the sets of triples consisting of inputs, decorations and outputs, of an input term, where the decorations are in IN form.

In the first algorithm, shown in Fig. 5, all triples are generated, and then "filtered" to remove those that contain decorations that are not in IN form. The filtering is carried out by the function *Checkin*. This function simply checks every triple against every other triple. If the two triples contain equivalent decorations, then the decoration with the highest precedence is discarded. The resulting sets of triples, $W(t)$, contains all rewrite sequences of the input term that correspond to IN 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.

```
|[ con ((Σ, V), R) : TermRewriteSystem;
      t, g : Term;

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

  func Generate(t : Term) : SetOfTriples
   |[ var H, Z(t) : SetOfTriples; i : ℕ;
    H := Z(t) := ∅;
    if t :: a ⟶ Z(t) := {⟨t, D_ε, t⟩};
     [] t :: a(t_1, ..., t_n) ⟶
          |[ for all 1 ⩽ i ⩽ n do Z(t_i) := Generate(t_i) od;
            (* Let O(t_i) = {ot_{k_i} | ⟨it_{k_i}, D_{k_i}, ot_{k_i}⟩ ∈ Z(t_i)} *)
            for all (t_{k_1}, ..., t_{k_n}) ∈ O(t_1) × ··· × O(t_n)
            do Z(t) := | Checkin(Z(t), ⟨a(t_{k_1}, ..., t_{k_n}), D_{k_1} ⊕ ··· ⊕ D_{k_n}, a(t_{k_1}, ..., t_{k_n})⟩) | od
          ]|
    fi;
    do H ≠ Z(t) ⟶
          |[ H := Z(t);
            for all ⟨it, D, ot⟩ ∈ Z(t)
            do for all p ∈ Pos(t) ∧ | (L_D(t|_ε) = ε ⇒ p = ε) |
              do for all r ∈ R ∧ S_D⟨r, p⟩ is acyclic
                do if ot ⟹/⟶ ⟨r,p⟩ ⟶ skip
                   [] ot ⟹⟶ ⟨r,p⟩ ot' ⟶ Z(t) := | Checkin(Z(t), ⟨it, D ⊗ ⟨r, p⟩, ot'⟩) |
                   fi
                od
              od
            od
          ]|
    od;
    return Z(t)
   ]|;

  func Checkin(Z : SetOfTriples, ⟨it, D, ot⟩ : Triple) : sEToFtRIPLES
    ..... as before .....

  func Trim(t : Term, t_g : SetOfTerms) : SetOfTriples
    ..... as before .....

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

Fig. 6. An algorithm that computes the IN rewrite sequences on-the-fly.

In the second algorithm, shown in Fig. 6, we check whether a triple contains an IN decoration on-the-fly. This check (as in the first algorithm we use the function *Checkin*) occurs in two places (indicated by the first and third boxes) in the function *Generate*; once when we initialise, and once when we have found a rewrite step that

rewrites the output. We also add a check (indicated by the second box) to ensure that a local rewrite sequence of a subtree begins with a rewrite rule applied to the root of the subtree. In the worst case, checking whether decorations are in IN form is quadratic in the number of triples in both algorithms.

6. Weak innermost normal decorations

The idea behind the *weak innermost normal* (WIN) form is to identify permutations of rewrite sequences that arise because of the substitution of variables. In the WIN 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 weakly non-rewriteable, or non-rewriteable for short. By avoiding rewriting these positions, we (again) avoid generating local rewrite sequences that are permutations of each other. All definitions in this section are with respect to a WTRS $((\Sigma, V), R, C)$. We begin by defining the set of positions in a term at which a variable occurs.

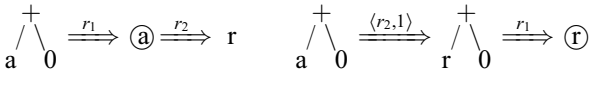
Definition 27 (*Variable positions*). The set VP of variable positions of a term $t \in T_\Sigma(V)$ is defined as the set of positions at which a variable occurs. In other words, $VP(t) = \{ p \in Pos(t) \mid t|_p \in V \}$.

A position in a term is either *rewriteable* or *non-rewriteable*. A rewriteable position in a term is a position 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 contains a variable, then the positions in the term substituted for the variable become non-rewriteable. Otherwise, the rewriteability of the positions in the rewritten term do not change.

Example 28. Consider the TRS with $S = \{+, a, 0\}$, corresponding ranks $\{2, 0, 0\}$, variable $V = \{x\}$, input term $+(a, 0)$, goal term r, and rules:

$$r_1 : +(x, 0) \to x,$$
$$r_2 : a \to r.$$

The rewrite sequence shown on the left below is *not* in WIN 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 sequence on the right is in WIN form



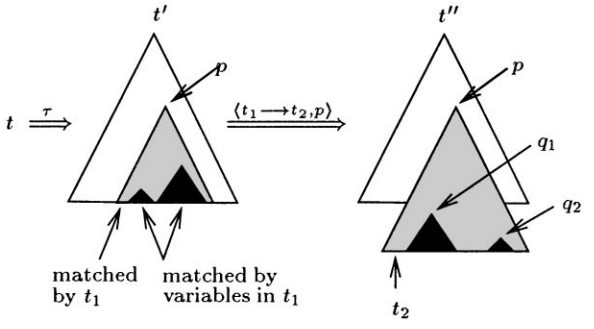


Fig. 7. Computing the rewriteable positions in a term after the application of $\langle t_1 \to t_2, p\rangle$.

Definition 29 (*Rewriteable positions*). The set RP_t of rewriteable positions in a term t after the application of the rewrite sequence τ and rewrite step $\langle t_1 \to t_2, p\rangle$ is defined as

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

where $t \stackrel{\tau}{\Longrightarrow} t' \stackrel{\langle t_1 \to t_2, p\rangle}{\Longrightarrow} t''$.

In Fig. 7 we depict how rewriteable positions are computed. Assume that we have some rewrite sequence $t \stackrel{\tau}{\Longrightarrow} t'$. If the left-hand side of the rule $t_1 \to t_2$ matches $t'|_p$, 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 Fig. 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 these variables in t_2 (the matching sub-terms are shown in black in t' and t'').

In Definition 29, 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 30. 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

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

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

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

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

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

In an IN rewrite sequence, a commutativity rule (here r_3) cannot be applied until the 'children' have first been rewritten. In this case, this means:

$$+(c,r) \xrightarrow{\langle r_7,2\rangle} +(c,d) \xrightarrow{r_3} +(d,c) \xrightarrow{r_1} i(d) \xrightarrow{\langle r_5,1\rangle} i(r) \xrightarrow{r_4} r$$

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 31 (*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,

- $Permitted_t(\varepsilon) = true$,
- $Permitted_t(\tau\langle r, p\rangle) = p \in RP_t(\tau) \wedge Permitted_t(\tau)$.

Definition 32 (*Weak innermost normal decoration*). An innermost normal decoration $D(t)$ is in weak innermost normal form iff $Permitted_t(L_D(t|_p))$ is true, for all $p \in Pos(t)$.

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

The following theorem means that we only need to consider WIN decorations of a term t, and not the entire universe of IN decorations of t. This theorem is analogous to Theorem 24.

Theorem 33 (Weak innermost normal decoration existence). *Given a rewrite sequence τ and term t, we have that $t \xrightarrow{\tau} \Rightarrow (\exists D(t) \in WIN(t) : S_D(t) \cong_t \tau)$.*

Proof (*outline*)**.** Let τ be an arbitrary rewrite sequence and t some term such that $t \xrightarrow{\tau}$. From Theorem 24 it follows that there exists a IN decoration $D(t)$ correspond-

ing to τ. If $D(t)$ is not in WIN form, then there is some p such that $\neg\, Permitted_t(L_D(t|_p))$. This means that we can write:

$$L_D(t|_p) = \cdots t' \xrightarrow{\langle t_1 \to t_2, p' \rangle} t'' \xrightarrow{\tau_1} t''' \xrightarrow{\langle t_a \to t_b, p''' \rangle} \cdots$$

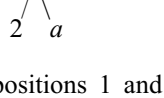
where t_1 (and t_2) contain at least one variable v, and t_a is a sub-term of the sub-term t_v of t' that matches v. In the proof, we move a rewrite step ($\xrightarrow{\langle t_a \to t_b, p''' \rangle}$ above) that is applied to a sub-term that is non-rewriteable to before the rewrite step ($\xrightarrow{\langle t_1 \to t_2, p' \rangle}$ above) that made the sub-term non-rewriteable in the first place. Applying this procedure repeatedly will result in a local rewrite sequence $L_D(t|_p)$ that is permissible. Given an IN decoration $D(t)$, therefore, we can now make each local rewrite sequence $L_D(t|_p)$, for all $p \in Pos(t)$, permissible. This results in a WIN decoration. For space reasons, the full proof is not shown, and the reader is referred to [20]. □

Note that if the TRS is non-linear (see the 4th constraint in Definition 4) then there may be rewrite sequences (decorations) that have no corresponding WIN form. To reach the goal in this situation you are forced to rewrite a non-rewriteable position. Consider the following example. (Note that from now on we will dispense with explicitly declaring the alphabet and variables in TRSs, and adopt the convention that x is always a variable, and the rest of the symbols that occur in the rewrite rules constitute the alphabet. Furthermore, the goal term is always r.)

Example 34. A TRS consisting of the rules

$$r_1 : *(2,x) \to +(x,x),$$
$$r_2 : a \to b,$$
$$r_3 : +(b,a) \to r$$

is non-linear because the variable x appears twice on the right-hand side of rule r_1. Consider the IN decoration



The positions 1 and 2 are non-rewriteable after the application of r_1. But we cannot swap the application of r_1 and r_2 without losing the possibility of performing r_3, which is the (only) rule that leads to the goal term. There is therefore no WIN form.

To better understand the role that the WIN form plays, it is instructive to consider IN decorations that are not in WIN form. If the underlying TRS does not contain any variables, then quite obviously, all IN decorations are also in WIN form (because all positions are always rewriteable). More interestingly, an example of an IN decoration that is not in WIN form is the following.

Example 35. Consider the TRS with variable x:

$$r_1 : a(b) \to c(d),$$
$$r_2 : c(x) \to x,$$
$$r_3 : d \to r$$

and the decorations:

$$\begin{array}{ll} D_1 & D_2 \\ \begin{array}{c} a\ r_1 r_2 r_3 \\ | \\ b \end{array} & \begin{array}{c} a\ r_1\langle r_3, 1\rangle r_2 \\ | \\ b \end{array} \end{array}$$

The rewrite sequences S_{D_1} and S_{D_2} are permutations of each other (as are $L_{D_1}(t|_\varepsilon)$ and $L_{D_2}(t|_\varepsilon)$), and both decorations are in IN form. However, unlike D_2, the decoration D_1 is not in WIN form because the rule r_2 contains a variable x that makes the root position non-rewriteable (r_3 may therefore not be applied).

Note that many decorations that fail to satisfy the WIN property also fail to satisfy the IN property. In effect, both properties require that rules be applied as low as possible, which means as early as possible when you traverse the expression tree bottom-up. The worth of the WIN property lies predominantly in excluding rewrite sequences in which the permutation exists at the local rewrite sequence level.

The definitions of the inputs and outputs of WIN decorations are the same as Definitions 15 and 16 (resp.), with the extra restriction $D(t) \in WIN(t)$.

7. An algorithm that computes WIN decorations

We can now give an algorithm that computes the WIN triples of an input term. This algorithm is shown in Fig. 8. Actually, this algorithm is almost identical to the 'IN' algorithm (shown in Fig. 6). The only difference is the extra check for the 'weak' condition (i.e. the check whether a position is rewriteable). This check is indicated by a box in Fig. 8.

Example 36. 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 Fig. 9. Notice that in this table there are 3 WIN rewrite sequences that rewrite the root node $+$. This should be compared to the number of 'naive' rewrite sequences, which is 2775 (see Example 17).

8. Terminating term rewrite systems

A TRS that has the property that all rewrite sequences are finite in length is referred to as *terminating*. More formally, a TRS $((\Sigma, V), R)$ is terminating for a set of terms

```
|[ con ((Σ, V), R): TermRewriteSystem;
       t, g : Term;

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

  func Generate(t : Term): SetOfTriples
   |[ var H, Z(t): SetOfTriples; i : ℕ;
     H := Z(t) := ∅;
     if t :: a ⟶ Z(t) := {⟨t, D_ε, t⟩};
      [] t :: a(t_1, ..., t_n) ⟶
           |[ for all 1 ⩽ i ⩽ n do Z(t_i) := Generate(t_i) od;
              (* Let O(t_i) = {ot_{k_i} | ⟨it_{k_i}, D_{k_i}, ot_{k_i}⟩ ∈ Z(t_i)} *)
              for all (t_{k_1}, ···, t_{k_n}) ∈ O(t_1) × ··· × O(t_n)
              do Z(t) := Checkin(Z(t), ⟨a(t_{k_1}, ..., t_{k_n}), D_{k_1} ⊕ ··· ⊕ D_{k_n}, a(t_{k_1}, ..., t_{k_n})⟩) od
           ]|
     fi;
     do H ≠ Z(t) ⟶ |[ H := Z(t);
                      for all ⟨it, D, ot⟩ ∈ Z(t)
                      do for all [ p ∈ RP_t(S_D) ] ∧ (L_D(t|_ε) = ε ⇒ p = ε)
                         do for all r ∈ R ∧ S_D⟨r, p⟩ is acyclic
                            do if ot ⟹/ ⟨r,p⟩ ⟶ skip
                                 [] ot ⟹ ⟨r,p⟩ ot' ⟶ Z(t) := Checkin(Z(t), ⟨it, D ⊗ ⟨r, p⟩, ot'⟩)
                               fi
                            od
                         od
                      od
                   ]|
     od;
     return Z(t)
   ]|;

  func Checkin(Z :SetOfTriples, ⟨it, D, ot⟩: Triple): SetOfTriples
   ..... as before .....

  func Trim(t : Term, t_g : SetOfTerms): SetOfTriples
   ..... as before .....

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

Fig. 8. An algorithm that computes the WIN rewrite sequences.

$T_\Sigma(V)$ if no infinite sequence of terms $t_i \in T_\Sigma(V)$ exists such that $t_1 \Longrightarrow t_2 \Longrightarrow t_3 \Longrightarrow \cdots$. A system is *non-terminating* if such a sequence exists.

There may be two causes of non-termination: cyclical rewrite sequences and divergent rewrite sequences:

Cyclical: A TRS that contains rewrite rules that express commutativity or associativity, for example, is cyclical. Furthermore, TRSs that contain a rewrite rules of the form

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

Fig. 9. The sets of trimmed WIN triples $V(t)$ for each node in the term $+(c,i(d))$.

$f(a)\to f(b)$, for some function f and $a,b\in\Sigma_0$, even simply $a\to b$, will be cyclical if there exists some rewrite sequence $b\Longrightarrow\cdots\Longrightarrow a$.

Divergent: Intuitively, diverging rewrite sequences can occur because the right-hand side of a rewrite rule can be more complex than the left-hand side. In that case, terms in a rewrite sequence can continue to grow in size indefinitely.

The TRSs that we consider in this work are cyclical. In the setting of code generation, chain rules (see e.g. rules r_5 and r_7 in Example 5) that can cause cycles are prevalent because there is often a great deal of data transparency in computer architectures – the same data may be stored in many 'different' ways, and moving from one form to another is at no cost (no instruction is needed). Approximately 20% of the rules for even a complex instruction-set machine like the Intel are chain rules, for example, which means the potential for cycles is large. In general, it is characteristic of code-generation TRSs that they are massively non-confluent, and cycles are commonplace.

We explicitly check that the rewrite sequences in the algorithms that we present are acyclic, otherwise the algorithms would not terminate. We do this in the function *Generate* in Figs. 3, 6 and 8.

We do not make any attempt to detect the divergence of rewrite sequences in the algorithms, however. The problem is that, as we traverse the term in a bottom-up fashion, we do not know which output terms of a given node will contribute to pattern matches at the node's parent. The output terms can also be arbitrarily large. Hence, if we observe that an output term of a node appears to be growing inordinately large, we still cannot artificially terminate rewriting because we do not know for sure whether we are witnessing divergence or not. Consider the following example.

Example 37. Consider the TRS with rewrite rules:

$$r_1 : c\to m(c),$$
$$r_2 : m(c)\to a,$$
$$r_3 : m(a)\to r.$$

The TRS is non-terminating because we can generate a divergent rewrite sequence consisting of repeated applications of the rule r_1 for input term c:

$$c \xrightarrow{r_1} m(c) \xrightarrow{\langle r_1,1\rangle} m(m(c)) \xrightarrow{\langle r_1,1\cdot 1\rangle} m(m(m(c))) \xrightarrow{\langle r_1,1\cdot 1\cdot 1\rangle} \cdots$$

In fact, a successful rewrite sequence for this input term and goal term r involves (only) two applications of rule r_1, as shown below:

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

Note that the maximum length of a local rewrite sequence in a terminating TRS may not be bounded, but will depend on the input term.

Example 38. Consider the TRS with rewrite rules:

$$r_1 : m(+(c,x)) \to m(x),$$
$$r_2 : m(r) \to r.$$

Local rewrite sequences for this TRS will be finite in length, but unbounded. Examples of rewrite sequences for three different input terms are:

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

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

$$m(+(c,+(c,+(c,r)))) \xrightarrow{r_1} m(+(c,+(c,r))) \xrightarrow{r_1} m(+(c,r)) \xrightarrow{r_1} m(r) \xrightarrow{r_2} r.$$

In general, determining whether a given TRS is terminating is, of course, undecidable. This is also true for the class of finite, linear TRSs considered in this work. The best that one can hope for, therefore, is to find under which conditions a TRS is guaranteed to be terminating.

Pelegri-Llopart and Graham state in [23, 24] that a TRS $((\Sigma, V), R)$ is terminating, if for all $(t,t') \in R$ and $a,b \in \Sigma_n$, one of the following conditions holds:

(i) $Var(t) = \emptyset$ and $t' \in \Sigma_0$,
(ii) $t = a(t_1,\ldots,t_n)$ and $t' = b(t_1,\ldots,t_n)$ for $n \geqslant 0$,
(iii) $t = a(t_1,\ldots,t_n)$ and $t' = a(t_{\pi(1)},\ldots,t_{\pi(n)})$ with π a permutation on $[1,n]$,
(iv) $t = a(t_1,\ldots,t_n)$ and $t' = t_i$ for $1 \leqslant i \leqslant n$.

They refer to this as the *BURS property*. The four conditions are referred to as the (i) *instruction fragment* rule, (ii) *generic operator* rule, (iii) *commutativity* rule, and (iv) *identity* rule.[6] Pelegri-Llopart and Graham state that these conditions (rules) are sufficient to construct term rewrite systems for real machines. Kurtz [18] has shown that BURS systems are equivalent to finite linear term rewrite systems, and has applied the BURS property to these systems as well.

[6] They formulate the identity rule as "$t = a(t_1,t_2)$ and $t' = t_1$ with $Var(t_2) = \emptyset$", which is just a special case of our formulation.

Example 39. If we apply Pelegri-Llopart and Graham's characterisation to the TRS shown in Example 37, for example, then we find that r_1 does not satisfy any of the conditions, and hence we can conclude that the TRS is non-terminating.

Unfortunately, however, there are two shortcomings with Pelegri-Llopart and Graham's BURS property:

- The BURS property does not characterise cyclic TRSs. For example, the TRS consisting of the single rule $+(x,y) \rightarrow +(y,x)$ clearly does not terminate, but satisfies the BURS property.
- The BURS property is very weak – the conditions are much too stringent. For example, any terminating TRS that includes a simple rule like $m(a) \rightarrow m(b)$ violates the BURS property.

The first shortcoming is not really a problem in our setting because we are interested in cyclical TRSs. The second, however, is a serious limitation. In spite of these shortcomings, however, there is as yet no better characterisation of divergence of TRSs than the BURS property.

In the literature, various sufficient conditions on TRSs that guarantee termination exist; for an overview see Dershowitz [5]. Generally, approaches that guarantee termination are based on verifying that the rewrite relation $\Longrightarrow$ is included in a partial-order relation $>$ on terms. If this relation is well-founded then there are no sequences of infinite length.

Example 40. Consider the TRS shown in Example 37, and the rewrite sequence $c \Longrightarrow m(c) \Longrightarrow m(m(c)) \cdots$. Although the ordering $m^k(c) > m^{k+1}(c)$ for $k \geqslant 0$ contains $\Longrightarrow$, it is not well-founded. The TRS is therefore non-terminating.

In principle, we must consider all rewrite sequences to determine $>$. However, in the case of *simplification orderings*, which are orders that possess the property that terms are always larger (under $>$) than their proper subterms, this is avoided. These orderings have been intensively investigated [19]. Steinbach [26] provides an overview and comparison of several simplification orderings (with status). A status determines the order in which subterms are compared and is necessary to order commutativity and associativity rewrite rules, for example. But note that the weakest simplification orderings are stronger than the BURS property. Which simplification ordering is most appropriate to code generation is an open question. An approach that uses forward closures (see [6]) is applicable to linear TRSs and is therefore relevant to our work. Other (more powerful) approaches to prove termination of TRSs such as the semantic-labelling method of Zantema [27] are less suited for our purpose since automation seems more complicated.

9. 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 an IN 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 WIN 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. There are a number of directions for future research:

- Analyse the performance and complexity of the pattern-matching algorithms. Preliminary performance measurements of a bottom-up pattern matcher based on our theory have shown that it still generates too many redundant rewrite sequences. Further optimisations are being looked at.
- Develop term rewrite systems for real machines.
- Investigate whether certain 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.

Acknowledgements

Many thanks to J.E. Jonker (University of Groningen, The Netherlands) who brought Example 34 to the attention of the authors, and to the referees for constructive criticism that resulted in an article of greater clarity.

References

[1] A.V. Aho, M. Ganapathi, S.W.K. Tjiang, Code generation using tree matching and dynamic programming, ACM Trans. Programming Languages and Systems 11 (4) (1989) 491–516.
[2] A. Balachandran, D.M. Dhamdhere, S. Biswas, Efficient retargetable code generation using bottom-up tree pattern matching, Comput. Languages 15 (3) (1990) 127–140.
[3] D.R. Chase, An improvement to bottom-up tree pattern matching, Proc. 14th Annual ACM Symp. on Principles of Programming Languages, Munich, Germany, January 1987, pp. 168–177.
[4] T.W. Christopher, P.J. Hatcher, R.C. Kukuk, Using dynamic programming to generate optimised code in a Graham-Glanville style code generator, Proc. ACM SIGPLAN 1984 Symp. on Compiler Construction; ACM SIGPLAN Notices 19 (6) (1984) 25–36.
[5] N. Dershowitz, Termination, in: J.-P. Jouannaud (Ed.), Rewriting Techniques and Applications, Lecture Notes in Computer Sciences, Vol. 202, Springer, Berlin, 1985, pp. 180–224.
[6] N. Dershowitz, Termination of rewriting, J. Symbolic Comput. 3 (1987) 69–116.
[7] N. Dershowitz, J.-P. Jouannaud, Rewrite systems, in: J. Van Leeuwen (Ed.), Handbook of Theoretical Computer Science, Vol. B: Formal Models and Semantics, Elsevier, Amsterdam, 1990, pp. 245–320 (Chapter 6).

[8] H. Emmelmann, Code selection by regularly controlled term rewriting, in: R. Giegerich, S.L. Graham (Eds.), Code generation – Concepts, Tools, Techniques, Workshops in Computing Series, Springer, Berlin, 1991, pp. 3–29.
[9] H. Emmelmann, F.W. Schröer, R. Landwehr, BEG – a generator for efficient back ends, ACM SIGPLAN Notices 24 (7) (1989) 246–257.
[10] C.W. Fraser, D.R. Hanson, T.A. Proebsting, Engineering a simple, efficient code-generator generator, ACM Letters on Programming Languages and Systems 1 (3) (1992) 213–226.
[11] M. Ganapathi, C.N. Fischer, Affix grammar driven code generation, ACM Transactions on Programming Languages and Systems 7 (4) (1985) 560–599.
[12] R. Giegerich, Code selection by inversion of order-sorted derivors, Theoret. Comput. Sci. 73 (1990) 177–211.
[13] R. Giegerich, K. Schmal, Code selection techniques: pattern matching, tree parsing, and inversion of derivors, in: H. Ganzinger (Ed.), Proc. 2nd European Symp. on Programming, Lecture Notes in Computer Science, Vol. 300, Springer, Berlin, 1988, pp. 247–268.
[14] R.S. Glanville, S.L. Graham, A new method for compiler code generation, Conf. Record 5th Annual ACM Symp. on Principles of Programming Languages, Tucson, AZ, January 1978, pp. 231–240.
[15] K.J. Gough, Bottom-up tree rewriting tool MBURG, ACM SIGPLAN Notices 31 (1) (1996) 28–31.
[16] P.J. Hatcher, T.W. Christopher, High-quality code generation via bottom-up tree pattern matching, Proc. 13th Annual ACM Symp. on Principles of Programming Languages, Tampa Bay, FL, January 1986, pp. 119–130.
[17] C. Hemerik, J.-P. Katoen, Bottom-up tree acceptors, Sci. Comput. Programming 13 (1990) 51–72.
[18] S. Kurtz, Narrowing and basic forward closures, Technical Report 5, Technische Fakultät, Universität Bielefeld, Germany, 1992.
[19] A. Middeldorp, H. Zantema, Simple termination of rewrite systems, Theoret. Comput. Sci. 175 (1997) 127–158.
[20] A. Nymeyer, J.-P. Katoen, Code generation based on formal BURS theory and heuristic search, Acta Inform. 34 (8) (1997) 597–635.
[21] A. Nymeyer, J.-P. Katoen, Y. Westra, H. Alblas, Code generation $=A^*+BURS$, in: T. Gyimóthy (Ed.), Compiler Construction, Lecture Notes in Computer Sciences, Vol. 1060, Springer, Berlin, April 1996, pp. 160–176.
[22] M.J. O'Donnel, Computing in Systems Described by Equations, Lecture Notes in Computer Science, Vol. 58, Springer, Berlin, 1977.
[23] E. Pelegrí-Llopart, Rewrite systems, pattern matching, and code generation, Ph.D. Thesis, University of California, Berkeley, December 1987.
[24] E. Pelegrí-Llopart, S.L. Graham, Optimal code generation for expression trees: An application of BURS theory, Proc. 15th Annual ACM Symp. on Principles of Programming Lanuguages, San Diego, CA, January 1998, pp. 294–308.
[25] T.A. Proebsting, B.R. Whaley, One-pass, optimal tree parsing – with or without trees, in: T. Gyimóthy (Ed.), Compiler Construction, Lecture Notes in Computer Science, Springer, Berlin, April 1996, pp. 294–308.
[26] J. Steinbach, Extensions and comparison of simplification orderings, in: N. Dershowitz (Ed.), Proc. 3rd Internat. Conf. on Rewriting Techniques and Applications, Lecture Notes in Computer Science, Vol. 355, Springer-Verlag, 1989, pp. 434–448.
[27] H. Zantema, Termination of term rewriting by semantic labelling, Fund. Inform. 24 (1994) 89–105.