Acta Informatica 34, 597–635 (1997)

Acta Informatica

© Springer-Verlag 1997

Code generation based on formal BURS theory and heuristic search

A. Nymeyer, J.-P. Katoen

University of Twente, Department of Computer Science, P.O. Box 217, 7500 AE Enschede, The Netherlands

Received 20 November 1995 / 26 June 1996

Abstract. BURS theory provides a powerful mechanism to efficiently generate pattern matches in a given expression tree. BURS, which stands for *bottom-up rewrite system*, is based on term rewrite systems, to which costs are added. We formalise the underlying theory, and derive an algorithm that computes all pattern matches. This algorithm terminates if the term rewrite system is *finite*. We couple this algorithm with the well-known search algorithm A^* that carries out pattern selection. The search algorithm is directed by a cost heuristic that estimates the minimum cost of code that has yet to be generated. The advantage of using a search algorithm is that we need to compute only those costs that may be part of an optimal rewrite sequence (and not the costs of all possible rewrite sequences as in dynamic programming). A system that implements the algorithms presented in this work has been built.

1 Introduction

Compiler building is a time-consuming and error-prone activity. Building the front-end (i.e. scanner, parser and intermediate-code generator) is relatively straightforward – the theory is well established, and there is ample tool support. The main problem lies with the back-end, namely the code generator and optimiser – there is little theory and even less tool support. Generating a code generator from an abstract specification, also called automatic code generation, in an efficient way is a very difficult problem.

Pattern matching and selection is a general class of code-generation technique that has been studied in many forms. The most successful form uses a code generator that works predominantly bottom-up; a so-called *bottom-up pattern matcher* (BUPM). A variation of this technique is based on term rewrite systems. This technique, popularised under the name BURS, and developed by Pelegri-Llopart and Graham [36], has arguably been considered the state of the art in automatic code generation. BURS, which stands for *bottom-up rewrite system*, has an underlying theory. However, to this day, BURS theory is poorly understood. Evidence of this statement are:

Correspondence to: A. Nymeyer (e-mail: nymeyer@cs.utwente.nl)

- There has been no further development of BURS theory since its initial publication [35]. Research in so-called BURS theory has been mainly concerned with improved table-compression methods.
- Researchers who claim to use BURS theory (e.g. [17, 37]) generally use 'weaker' tree grammars instead of term rewrite systems.
- Researchers often equate a BURS with a system that does a static cost analysis (e.g. [16]).

We argue that a static cost analysis is neither necessary nor sufficient to qualify as BURS, and that a system that is based on tree grammars cannot be BURS.

In this work we present a lucid but concise and formal derivation of BURS theory that is based on the (semi-formal) work of Pelegri-Llopart and Graham. However, we differ in that we do not use the instruction costs to make optimal choices statically. Instead we use a heuristic search algorithm that only needs to dynamically compute costs for those patterns that may contribute to optimal code. A result of this dynamic approach is that we do not require involved table-compression techniques. Note that we do not address register allocation in this work; we are only interested in pattern matching and selection, and optimal code generation.

We begin in the following section with a literature survey. This survey traces the development of BUPMs and BURSs, and places our research in context. In Sect. 3 we describe the heuristic search algorithm A^* that is used to select optimal code. The algorithm A^* is all-purpose – it can be used to solve all kinds of 'shortest-path' problems. In our case the search graph consists of all the possible reductions of an expression tree, and we wish to find the least expensive.

In Sect. 4 we define a *term rewrite system*, and derive an algorithm that generates the *input* and *output* sets of an expression tree. These sets contain the patterns that match the expression tree. To select the 'optimal' patterns, we use the search algorithm A^*. This algorithm uses a *successor function* (*algorithm*) to select patterns and apply rewrite rules. The successor function, which is presented in Sect. 5, provides the search algorithm with a set of possibly-optimal selections. In this sense, the successor function couples A^* and BURS. In the implementation, the algorithm that generates input and output sets, and the successor function, are modules that can be simply 'plugged' into A^* to produce a code generator. The implementation is also briefly described in Sect. 5. Finally, in Sect. 6, we present our conclusions.

2 Literature survey

In 1977 R. Glanville submitted a thesis [20] that provided a major impetus to the field of automatic code generation. Under the supervision of Susan Graham [22], he developed a technique that in the decade that followed was the subject of much scrutiny and refinement. Major players in this period include Henry, Ganapathi and Fischer (see [33] for a review). In the Graham-Glanville technique, as it has become known, the intermediate representation generated by the front-end of the compiler is specified by a context-free string grammar, and an LR-parser generator is used as a code-generator generator. Target-machine instructions are generated as a side effect of parsing the input expression. Unfortunately, while the technique is conceptually elegant, it proved unworkable in a production environment. The non-ambiguous LR formalism is inappropriate and too restrictive to specify the inherently ambiguous mapping from an intermediate representation to target code. As the popularity of

the Graham-Glanville technique waned in the mid-1980s, interest turned to another technique, called *bottom-up pattern matching*.

2.1 Bottom-up pattern matchers

In this technique, we represent the input expression and target-machine instructions as trees, where the instruction trees are referred to as patterns. Corresponding to each pattern is a result (leaf or node), a target-machine instruction and a cost. If a pattern matches a subtree of the input tree, then we can replace the subtree by its result node. During code generation, we traverse the input tree bottom-up and find all pattern matches. In a subsequent top-down traversal, we choose the least-expensive series of pattern matches that reduce the input tree into a single node. In a final bottom-up traversal, we generate the instructions that correspond to the selected patterns. The code-generator generator in this technique reads the patterns, instructions and costs, and generates a code generator that consists of a combined pattern matcher and instruction generator. This is referred to as the *static* phase. In the *dynamic* phase, the code generator reads the input expression tree and generates target-machine instructions. What makes the problem hard is that there can be very many ways of reducing the input tree. Further, the cost analysis that is necessary to determine the least-expensive reduction can be very time consuming.

Early work on this technique was done by Kron [31], and by Hoffmann and O'Donnell [29]. The latter in particular have provided the basic theory for early implementations of BUPMs. Chase [10], for example, implemented a BUPM using the theory developed by Hoffmann and O'Donnell. Chase specified the patterns using a *regular tree grammar* (RTG). A RTG is a context-free grammar with prefix notation on the right-hand sides of the productions representing trees. Chase found that the tables generated by the pattern matcher were enormous, requiring extensive use of compression techniques. Like Hoffmann and O'Donnell, Chase did not consider the problem of selecting patterns, hence he had no need for costs. A formal, concise and lucid description of Chase's table-compression technique can be found in Hemerik and Katoen [24], who formally developed naive and optimised bottom-up pattern-matching algorithms. An asymptotic improvement in both space and time to Chase's algorithm is given by Cai et al. [6].

Hatcher and Christopher [23] went further than Chase and built a complete BUPM for a VAX-11. Their work was a milestone in that they carried out *static* cost analysis, which is a cost analysis carried out at code-generator generation time. In a *dynamic* cost analysis, the code generator itself performs the cost analysis. This is a space-time trade-off. Static cost-analysis makes the code-generator generator more complex and requires a lot of space for tables. In effect, pattern selection is encoded into the tables. The resulting code generator, however, is simple and fast. This means that compilation is faster. We refer to a BUPM that does static (dynamic) cost analysis as a static (dynamic) BUPM. The approach of Hatcher and Christopher does not guarantee that the (statically) selected code will always be optimal. In those cases where this occurs, however, the compiler builder is warned.

In both the static and dynamic BUPMs, the cost analysis is usually carried out using *dynamic programming*. Dynamic programming has had a long association with the field of code generation. Early theoretical work by Aho and Johnson [2, 3] for example, considered globally optimal code generation for idealised register machines

and a restricted class of input expression trees. This work used the dynamic programming algorithm [4] to generate provably optimal code. A top-down dynamic programming algorithm was used by Aho, Ganapathi and Tjiang [1] to build a code-generator generator called *twig*, and by Christopher et al. [11], and Weisgerber and Wilhelm [39]. The advantages of this top-down technique is that it is intuitive and that it has theoretical roots. Its disadvantage is that, because dynamic programming is done during code generation, the code generator can be slow. More recently, Fraser et al. [16] have reported that *twig* has problems processing large grammars.

In a *tour de force*, Henry and Damron [28, 27] compared the static and dynamic performance of the bottom-up and top-down methods, and also the Graham-Glanville and two brute-force methods, using a system called CODEGEN [25]. They found that the code generators for both the static and dynamic BUPM produce (locally) optimal code. The static BUPM had the slowest and most complex code-generator generator, but the code generator carried out pattern matching 3 times faster than its dynamic counterpart. Henry and Damron [28] remark, however, that at the time of their research, bottom-up technology was still "immature." In [26], Henry found similar differences in performance between static and dynamic BUPMs.

In 1990, Balachandran et al. [5] used a RTG and techniques based on the work of Chase, Hatcher and Christopher to build a static BUPM. Unlike the system of Hatcher and Christopher, however, no user intervention was needed to achieve optimal code. Very recently, Ferdinand et al. [15] reformulated the (static) bottom-up pattern-matching algorithms (based on RTGs) in terms of finite tree automata. This theoretical work was based on the work of Kron [31]. To determine the patterns that match an input tree, they use a subset-construction algorithm. This algorithm, which is developed in a step-wise fashion, does a static cost analysis, and generalises the table-compression technique of Chase.

There have been a number of notable attempts to improve the efficiency of the dynamic (BUPM) code generator, namely Emmelmann et al. [14] who developed the BEG system, Fraser et al. [16] who developed the IBURG system, and very recently Gough [21] who developed the MBURG system. All used a hard-coded dynamic cost analysis. Emmelmann et al. found that their dynamic code generator outperformed the standard SUN-workstation (Modula-2) code generator by almost an order of magnitude, and generated code of comparable quality. Fraser et al. compared their dynamic code generator with a code generator from the BURG system (described below). The BURG-system code generator does not perform a cost analysis. They found that their dynamic code generator was something like an order of magnitude slower. However, the code generator's slowness was compensated by the fact that IBURG was simpler and more intuitive in structure. In fact, it was found to be useful as a 'test tool' in developing the more complex, static BURG system. Gough's MBURG system is a variant of IBURG, and more developed.

2.2 Bottom-up rewrite systems

A decade to the month after Glanville submitted his thesis, E. Pelegri-Llopart, who was also a student of Susan Graham, submitted a thesis [35] that attracted much attention. In his thesis, Pelegri-Llopart developed a code-generation system using a semi-formal approach referred to as BURS theory [36]. Pelegri-Llopart combined the static cost analysis concept from Hatcher and Christopher, the pattern-matching and table-compression techniques from Chase, and, most importantly, term rewrite systems

rather than tree grammars to develop a BURS. A BURS is, in fact, a generalisation of a BUPM, and is more powerful. The term rewrite system in a BURS consists of rewrite rules that define transformations between *terms*. A term, which is represented by a tree, consists of operators and operands (which are analogous to nonterminals and terminals in context-free grammars). However, *variables* that can match any tree are also allowed. The advantage of using a term rewrite system is that, as well as the usual rewrite rules that reduce the expression tree, we can use rules that transform the expression tree. Algebraic properties of terms can therefore be incorporated into the code-generation process. The BURS 'theory' that Pelegri-Llopart and Graham developed is quite complex, however.

Pelegri-Llopart and Graham [36] compared the performance of a BURS with other techniques as part of an early (1987) implementation of Henry and Damron's CODEGEN system (see above). They found that the tables were smaller and the code generator much faster. Henry [26] also compared a BURS with a static BUPM using the CODEGEN system. He found that the static BUPM code-generator generator was $2\frac{1}{2}$ times faster but used a surprising 4 times as much space.

In 1991 Emmelmann [13] 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 [19, 18], 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 is unlikely ever to be completely implementable. It demonstrates that power of specification must be weighed up against practicality.

The difficulty researchers had with BURS theory is reflected in the work of Balachandran et al., described above, who conceded that term rewrite systems are more powerful, but argued that RTGs are simpler, and more easily understood and implemented than term rewrite systems, and that better table-compression techniques could be applied.

In 1992, Fraser, Henry and Proebsting [17] presented a new implementation of so-called 'BURS technology'. Their system, called BURG, accepts a tree grammar (and not a term rewrite system) and generates a 'BURS'. The algorithm for generating the 'BURS' tables is described by Proebsting in [37]. Proebsting compares this algorithm with a table-generation algorithm from Henry [26], described above, and reports an impressive performance gain (an order of magnitude). It is not clear, however, what the relationship is between the table-generation algorithms in BURG and Pelegri-Llopart's BURS, given that they are based on different formalisms.

2.3 Heuristic search techniques

The PQCC (Production-Quality Compiler Compiler) Project [40] was an early and ambitious project that aimed at automating the process of generating high-performance compilers. This work paid particular attention to the problem of *phase ordering*. (The phase-ordering problem relates to the inter-dependence and order of the various code-generation activities like storage allocation, common-subexpression elimination, instruction selection and scheduling, (peephole) optimisation and register allocation.) Together with the Graham-Glanville and pattern-matching techniques, it was also one of the first works that separated the target-machine description from the code-generation algorithm.

The construction of the code generator and the code-generator generator in PQCC are reported by Cattell in [7, 8, 9]. There are 2 aspects of Cattell's work that are relevant to our work.

- Cattell uses a means-ends analysis to determine an optimal code match. This involves selecting a set of instruction templates that are *semantically close* to a given pattern in the input expression tree. For each of these templates, rewrite rules (Cattell calls them *axioms*) are used to transform the template recursively into the pattern. The least-expensive template is chosen from those that are successful. The heuristic *semantic closeness* means that either the root operators match, or that there is an axiom that rewrites the root operator of the template into the root operator of the pattern.
  Note that the search procedure is not bounded – both the depth of the recursion and the number of semantically-close templates need to be restricted.
- For performance reasons, the 'major part' of the search procedure is carried out statically on a set of heuristically generated pattern trees that the code generator is likely to encounter. All the templates generated by the code-generator generator, together with the associated instruction sequences, are stored in a table for use by the code generator.
  The ('minor') part of the search procedure carried out by the code generator involves only 1 rule, called 'fetch decomposition' by Cattell. This rule basically allocates temporary storage (or registers) to those operands that are not matched.

2.4 Summing up

Much recent research in automatic code generation has been aimed at improving the performance of the code generator by doing a static cost analysis. This research is often carried out under the name BURS, in spite of the fact that an underlying tree-grammar formalism is used. Term rewrite systems, and the 'theory' developed by Pelegri-Llopart, has received scant attention in the literature. Furthermore, researchers have encountered the following problems with a static cost analysis:

1. A static cost analysis requires extensive table-compression techniques.
2. The resulting code-generator generator is complex.
3. Costs cannot depend on run-time (dynamic) parameters.
4. The cost analysis can fail (due to *cost divergence*, see for example [17]).

However, the overriding benefit of doing a static cost analysis is that it results in a simpler and faster code generator. To the authors' knowledge, the only application of search techniques to code generation to date has been the work of Cattell in the PQCC project. Cattell's technique, however, is related to dynamic programming, and his use of heuristics is different.

3 Heuristic-search methods

Search techniques are used extensively in artificial intelligence [34, 30] where data is dynamically generated. In a search technique, we represent a given state in a system by a node. The system begins in an initial state. Under some action, the state can change – this is represented by an edge. Associated with an action (or edge) is a cost.

By carrying out a sequence of actions, the system will eventually reach a certain goal state. The aim of the search technique is to find the least-cost series of actions from the initial state to one of the goal states. In most problems of practical interest, the number of states in the system is very large. The representation of the system in terms of nodes, edges and costs is called the search graph.

Definition 1 (Search graphs). *A search graph G is defined by a quadruple (N, E, n_0, N_g) with:*

- *N, a set of nodes*
- *$E \subseteq N \times N$, a set of directed edges each labelled with a cost $C(n,m) \in \mathbb{R}$, $(n,m) \in E$*
- *$n_0 \in N$, an initial node*
- *$N_g \subseteq N$, a set of goal nodes*

such that the following conditions are satisfied:

- *G is connected*
- *$N_g \neq \emptyset$*
- *$\forall (n,m) \in E : n \notin N_g$*

Note that $\mathbb{R}$ denotes the set of real numbers.

Traditionally, the search graph is drawn 'top-down', i.e., with the initial node n_0 at the top and the set of goal nodes N_g at the bottom. We adopt this convention. All edges in an acyclic search graph can therefore be assumed to point 'down'. Given a search graph, the aim is to find the least-cost path from n_0 to a node in N_g.

A brute-force technique that acts as a basis for finding such a least-cost path is called *breadth-first search*. Actually, this technique determines a shortest path from n_0 to a node in N_g. In this technique, we initialise a set of nodes to $\{n_0\}$. At each step we compute the *successor nodes* of all the nodes in the set. The successor nodes of a given node are those nodes that can be reached with a path of length one from the node. The algorithm terminates when we find a successor node that is a goal node. This algorithm computes all the nodes in the search graph at a certain depth, before proceeding further. At some point we will reach a depth that contains a goal node.

We can use the breadth-first search technique to determine the *least-cost* path by computing the successor nodes of only the 'cheapest' node in our set at each step. We determine the cheapest node by determining the costs of the paths from n_0 to each node in our set. The successors of the chosen node are then added to our set. This technique is called *best-first search*. We can improve this technique even more if we include the *estimated* cost to the goal in the cost of a node. This estimated cost is obtained by using heuristic domain knowledge that is available during traversal of the search graph. By using this heuristic knowledge, we can avoid searching some unnecessary parts of the search graph. Careful choice of the heuristic can therefore reduce the number of paths that the search technique tries in an attempt to find a goal node.

The best known search technique that uses this technique is the A* algorithm. The letter 'A' here stands for 'additive' (an additive cost function is used), and the asterisk signifies that a heuristic is used in the algorithm. The algorithm uses the following two cost functions to direct the search:

- $g(n)$, which is the minimum cost of reaching the node n from the initial node n_0, and

– $h^*(n)$, which is the *estimated* minimum cost of reaching a goal node from node n.

Associated with each node n is a cost $f^*(n) = g(n) + h^*(n)$. The actual cost of reaching a goal node from n is called $h(n)$. The relationship between $h^*(n)$ and $h(n)$ is important. We consider the following cases:

1. $h^*(n) = 0$ If we do not use a heuristic, then the search will only be directed by the costs on the edges. This is called a *best-first* search.
2. $0 < h^*(n) < h(n)$ If we always underestimate the actual cost, then the algorithm will always find a minimal path (if there is one). A search algorithm with this property is said to be *admissible*.
3. $h^*(n) = h(n)$ If the actual and estimated costs are the same, then the algorithm will always choose correctly. As we do not need to choose between nodes, no search is necessary.
4. $h^*(n) > h(n)$ If the heuristic can overestimate the actual cost to a goal node, then the A* algorithm may settle on a path that is not minimal.

Example 1. In this example we show what can happen when a heuristic that overestimates the actual cost is used. Consider the search graph in Fig. 1. At node B the value of the heuristic is 3. This overestimates the actual cost to a goal node (C), which is 1. At nodes D and E the values of the heuristic happen to be correct. Starting at the initial node A, if we always choose the node with the lowest value of $f^*(n)$, then, since $f^*(D) < f^*(B)$ and $f^*(E) < f^*(B)$, we will determine that the minimum path is, incorrectly, $ADEF$.

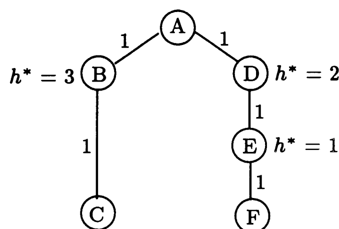

Fig. 1. A search graph including the value of some heuristic at each node

Note, however, that in some applications (code generation, for example) it may not be important that we find a path that is not (quite) minimal. It may be the case, for example, that a heuristic that occasionally overestimates the actual cost has superior performance than a heuristic that always plays safe. Furthermore, a heuristic that occasionally overestimates may only generate a non-minimum path in a very small number of cases.

The A* algorithm is shown in Fig. 2. The algorithm computes the minimum path from the initial node to a goal node. In this algorithm, we maintain two sets of nodes; open nodes $N_o \subseteq N$ and closed nodes $N_c \subseteq N$. The algorithm begins by initialising N_o to $\{n_0\}$, and N_c to $\emptyset$. Further, the path and cost of the initial node n_0 are initialised. We execute the main loop as long as we have not found a goal node. In the main loop, we use the cost function $f^*(n) = g(n) + h^*(n)$ to compute N_s, which is the set of nodes in N_o with smallest cost. If this set contains a goal node, then we are

```
|[ con G = (N, E, n_0, N_g): SearchGraph;

  func A_star : N*
   |[ var N_o, N_c, N_s : 𝒫(N);
         n, m : N;
         g, h* : N → ℝ;
         C : N² → ℝ;
         Path : N → N*;
         Successor : N → 𝒫(N);

    proc Propagate(p : N, q : N)
     |[ var r : N;
       if g(p) + C(p, q) ≥ g(q) ⟶ skip
        [] g(p) + C(p, q) < g(q) ⟶|[ Path(q) := Path(p) ⊕ q;
                                      g(q) := g(p) + C(p, q);
                                      for all r ∈ Successor(q) ∩ (N_o ∪ N_c)
                                      do if Path(r) ≠ Path(q) ⊕ r ⟶ skip
                                           [] Path(r) = Path(q) ⊕ r ⟶ Propagate(q, r)
                                          fi
                                      od
                                    ]|
       fi
    ]|;

    N_c := ∅;
    N_o := { n_0 };
    Path(n_0) := n_0;
    g(n_0) := 0.0;
    N_s := N_o;
    do  (N_s ∩ N_g = ∅) ⟶
        |[ choose n ∈ N_s;
           N_o := N_o − { n };
           N_c := N_c ∪ { n };
           for all m ∈ Successor(n)
           do if m ∉ N_o ∪ N_c ⟶|[ N_o := N_o ∪ { m };
                                    Path(m) := Path(n) ⊕ m;
                                    g(m) := g(n) + C(n, m)
                                  ]|
               [] m ∈ N_o ∪ N_c ⟶ Propagate(n, m)
              fi
           od;
           N_s := { n ∈ N_o | ∀m ∈ N_o : g(n) + h*(n) ≤ g(m) + h*(m) }
        ]|
    od;
    choose n ∈ (N_s ∩ N_g);
    return Path(n)
  ]|
]|.
```

Fig. 2. The A* algorithm

finished, and we return the path of this node. Otherwise we choose a node out of N_s, remove it from N_o, add it to N_c, and compute its successors. If a successor, m say, is neither in N_o nor N_c, then we add m to N_o, and compute its path and cost. If we have visited m before, and the 'new' cost of m is less than the cost on the previous visit, then we will need to 'propagate' the new cost. This involves visiting all nodes on paths emanating from m and recomputing the cost (this is done by a recursive call to *Propagate*).

The algorithm uses the functions *Successor* and *Path*. These functions are defined below.

Definition 2 (Successor nodes). *Given a search graph* $G = (N, E, n_0, N_g)$*, we define the set of successor nodes* $Successor(n) \in \mathscr{P}(N)$ *of a node* $n \in N$ *as*

$$Successor(n) = \{m \in N \mid (n, m) \in E\}$$

Note that if $n \in N_g$ *then* $Successor(n) = \emptyset$.

Definition 3 (Paths). *Given a search graph* $G = (N, E, n_0, N_g)$*, we define a path* $Path(n) \in N^*$ *to a node* $n \in N$ *as a string of nodes in the following way:*

$$Path(n) = n_0 n_1 \ldots n_k$$

such that $\forall 1 \leq i \leq k : n_i \in Successor(n_{i-1}) \ \wedge \ n_k = n, k \geq 0$.

Note that there may be more than one path that leads to a node. Furthermore, if $Path(n) = n_0 n_1 \ldots n_k$ and $m \in Successor(n)$ then we can append the node m to the path $Path(n)$ using the append operator $\oplus$. We write

$$\begin{aligned} Path(m) &= Path(n) \oplus m \\ &= n_0 n_1 \ldots n_k m \end{aligned}$$

Example 2. To demonstrate the A* algorithm, and the effect of the heuristic, we have considered the 8-puzzle. The 8-puzzle is a game consisting of a 3×3 board, and 8 tiles numbered 1 to 8. Initially the tiles are placed in some (presumably arbitrary) configuration on the board. The aim is to re-arrange the tiles until some final configuration is reached, making as few moves as possible. The only room we have to move a tile is the vacant square on the board. The initial and goal configurations that we use are shown below (on the left and right respectively).

1	6	
7	2	5
4	3	8

1	2	3
8		4
7	6	5

Notice that tile 1 is already in its correct square. To reach the goal configuration, we could first move tile 6 to the vacant square, and then move tile 2 to the square just vacated by tile 6. We would then have tile 2 in its correct position. Continuing on in this way with the other tiles we would eventually reach the goal configuration. Note that at each step, we will have to choose between at least two tiles to move to the vacant square.

For simplicity, we let the cost of moving a tile be 1, so the cost of a configuration is the number of moves required to reach the configuration. Note that tiles are not moved to the (vacant) square from which they came from in the previous move. Such a move would be redundant. A node in the search graph represents a configuration,

and an edge represents the action of moving a tile to the vacant square on the board. We apply the A* algorithm and the following two heuristics:

1. $h_0(n) = 0$. This corresponds to a best-first search, and because the costs of all edges are all equal to 1, it also corresponds to a breadth-first search.
2. $h_1(n) = \sum_{i=1}^{8} \mid p_x^i - g_x^i \mid + \mid p_y^i - g_y^i \mid$, where p_x^i is the x-coordinate of tile i in the present configuration (similarly p_y^i), and g_x^i is the x-coordinate of tile i in the goal configuration (similarly g_y^i). This heuristic, called the Manhattan distance, computes the number of moves that each of the eight tiles needs to make to reach its goal square, assuming no other tiles stand in the way. It usually underestimates the actual number of moves that will be required.

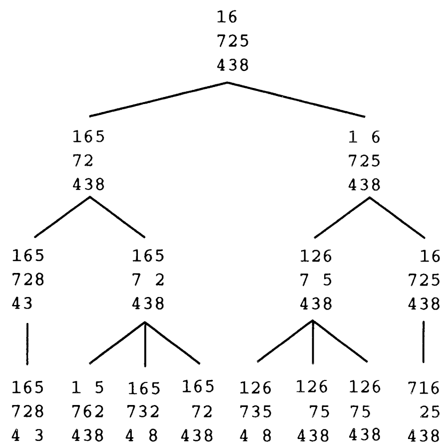

Fig. 3. Part of the search graph for the 8-puzzle

Part of the resulting breadth-first (i.e. corresponding to heuristic $h_0(n)$) search graph for the 8-puzzle is depicted in Fig. 3. The initial configuration is shown as the root, and we show all configurations up to three moves. We can read in this figure the number of nodes at depths 1, 2 and 3, namely 2, 4 and 8 (resp.). Ultimately, 22 moves are needed before the goal configuration is found (the goal is configuration number 8271 at depth 22). The total number of nodes at each depth ($\leq$ 22) is shown in Table 1 (column h_0). In total, the breadth-first search generated 103309 (open) nodes.

The performance with the heuristic $h_1(n)$ was a very different story. If we compare the two columns in Table 1 we see that at first there is little difference between the number of nodes at each depth. By depth 6, however, the ratio is approximately 2 to 1. From depth 10, the heuristic begins to home in on the goal node. This is most dramatic at depth 19, at which time it 'knows' the path to the goal. In total, this heuristic generated 655 (open) nodes. Both h_0 and h_1 generated the same path, by the way.

Other heuristics were also tried. The coarser (i.e. less accurate) a heuristic is, the more nodes that are generated, and the longer it takes to find the goal. Even a very coarse heuristic, however, is an improvement on the breadth-first search in this application. Heuristics that occasionally overestimate the actual cost were also tried.

Table 1. The number of closed nodes in the search graph for the 2 heuristics in the 8-puzzle

Depth	h_0	h_1
1	2	2
2	4	3
3	8	7
4	16	13
5	20	12
6	39	19
7	62	25
8	116	40
9	152	34
10	286	44
11	396	41

Depth	h_0	h_1
12	748	39
13	1024	29
14	1893	26
15	2512	23
16	4485	18
17	5638	11
18	9529	4
19	10878	1
20	16993	1
21	17110	1
22	8271	1

These also performed well, and interestingly, these generated different paths (but with the same cost, 22) from h_0, h_1 and other heuristics that underestimated the cost.

Code generation considered as template or pattern matching also lends itself to the A* technique. The transformations in code generation are specified by rewrite rules. Each rule consists of a match pattern, result pattern, cost and an associated machine instruction. A node n is an expression tree. The initial node consists of a given expression tree. From a given node, we can compute successor nodes by transforming sub-trees that are matched by match patterns. If a match occurs, we rewrite the matched sub-tree by the corresponding result pattern. The aim is to rewrite the expression tree (node) into a goal using the least-expensive sequence of rules. The associated sequence of machine instructions forms the code that corresponds to the expression tree. In the following example we illustrate this process.

Example 3. Consider the following set of rewrite rules, with corresponding costs and machine instructions.

rule	cost	machine instruction
$(r_1)\ c \longrightarrow r$	1	`load #c, r`
$(r_2)\ m(r_i) \longrightarrow r_j$	4	`load (r`$_i$`), r`$_j$
$(r_3)\ m(+(c, r_i)) \longrightarrow r_j$	7	`load #c(r`$_i$`), r`$_j$
$(r_4)\ r_i \longrightarrow m(r_j)$	4	`store r`$_i$`, (r`$_j$`)`
$(r_5)\ +(r_i, r_j) \longrightarrow r_j$	2	`add r`$_i$`, r`$_j$
$(r_6)\ +(c, r) \longrightarrow r$	3	`add #c, r`
$(r_7)\ +(c, m(r)) \longrightarrow m(r)$	9	`add #c, (r)`
$(r_8)\ +(x, y) \longrightarrow +(y, x)$	0	

The machine instructions consist of 3 load instructions, a store instruction and 3 add instructions. The addressing modes are register, immediate, register deferred and index. The letter c stands for constant, m for memory access, and r for register. The memory access takes one argument, and addition takes two arguments. The last rule, which expresses commutativity, contains the *variables* x and y. A variable may be substituted by any pattern representing an expression tree. Note that the match and result patterns are written in prefix notation, and that we differentiate between different instances of the same symbol in a rule by using the subscripts i and j. (We only do this in this example – in the rest of this paper we omit the subscripts.)

Let the initial expression tree be $+(m(+(c_1, c_2)), +(m(r_1), c_3))$, and the goal be r. We can rewrite this tree using the above rules in the following way:

$$\begin{aligned}
+(m(+(c_1, c_2)), +(m(r_1), c_3)) &\xrightarrow{r_1} +(m(+(c_1, r_2)), +(m(r_1), c_3)) \\
&\xrightarrow{r_3} +(r_3, +(m(r_1), c_3)) \\
&\xrightarrow{r_2} +(r_3, +(r_4, c_3)) \\
&\xrightarrow{r_8} +(r_3, +(c_3, r_4)) \\
&\xrightarrow{r_6} +(r_3, r_4) \\
&\xrightarrow{r_5} r_4
\end{aligned}$$

The cost of this rewrite sequence is 17. We could also have rewritten the tree in (very) many other ways. We now apply the A* algorithm. The heuristic that we use is the following:

$$h^*(n) = 4 * \ |m| \ + 2 * \ |+| \ + \ |c|$$

where $|s|$ denotes the number of times the symbol s appears in the expression tree at n. The heuristic is derived from the costs of the instructions. Instructions that access memory (m) are deemed to be 'expensive' (contributing a factor 4); an addition (+) is also moderately expensive (a factor 2); and finally a constant (c), which uses the immediate addressing mode, also contributes (a factor 1).

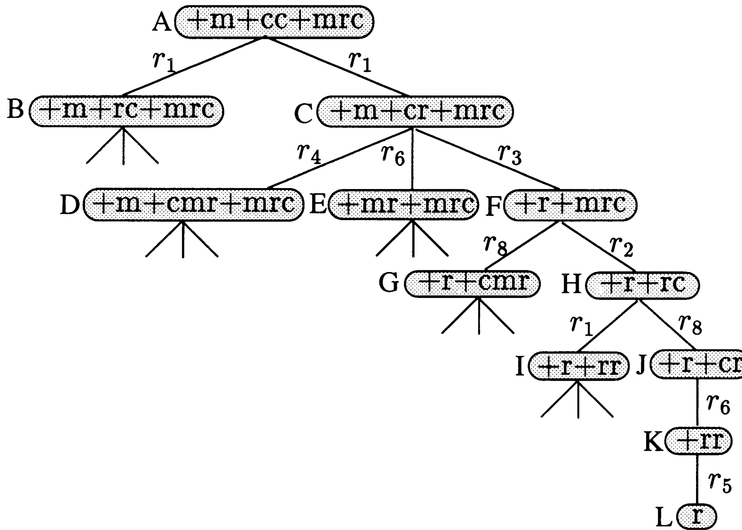

Fig. 4. The nodes in the heuristic search graph for the expression tree $+(m(+(c_1, c_2)), +(m(r_1), c_3))$

The steps that the A* algorithm takes to reduce the initial expression tree are shown in Table 2. The steps are also indicated in Fig. 4, by shaded boxes. Each edge in the graph is labelled by the rule that was applied. To make identification easier, each node is labelled by a letter.

In step one of the algorithm, we initialise the set N_o to $\{A_{0+17}\}$, where $g(A) = 0$ and $h^*(A) = 2 * 4 + 3 * 2 + 3 = 17$, and N_c to $\emptyset$. We choose the node A, add its successors B_{1+16} and C_{1+16} to the set N_o, and move A to N_c. In step two, the nodes

Table 2. The steps that the A* procedure takes to reduce $+(m(+(c_1,c_2)), +(m(r_1),c_3))$. The subscripts are the costs $g+h^*$ at the nodes

Step	N_o	N_c	Choose
1	A_{0+17}	ϵ	A
2	$B_{1+16}C_{1+16}$	A	C
3	$B_{1+16}D_{5+20}E_{4+13}F_{8+9}$	AC	F
4	$B_{1+16}D_{5+20}E_{4+13}G_{8+9}H_{12+5}$	ACF	H
5	$B_{1+16}D_{5+20}E_{4+13}G_{8+9}I_{13+4}J_{12+5}$	$ACFH$	J
6	$B_{1+16}D_{5+20}E_{4+13}G_{8+9}I_{13+4}K_{15+2}$	$ACFHJ$	K
7	$B_{1+16}D_{5+20}E_{4+13}G_{8+9}I_{13+4}L_{17+0}$	$ACFHJK$	

in N_o have the same cost, so we must choose between them. We adopt the policy that, when there is more than one node with the same (minimum) cost, we choose the last computed node. In this case that is C. We add the successors of C to N_o, move C to N_c and again choose the last, least expensive node in N_o. The process continues until step seven when we encounter the goal node L. The optimal path A, C, F, H, J, K and L is returned by the algorithm. This path corresponds to the rewrite sequence shown above. The code that is emitted by this sequence is the following:

```
load #c2,r2
load #c1(r2),r3
load (r1),r4
add #c3,r4
add r3,r4
```

4 BURS theory

In this section we derive the theory of a BURS. We first define some basic concepts, and we define a costed term rewrite system. For a more elaborate treatment of (term) rewrite systems we refer to [12].

Given a term rewrite system and an expression tree, we can define rewrite sequences and permutations of rewrite sequences. Typically, the total number of rewrite sequences for a given expression tree is enormous. In Sect. 4.2 and 4.3 we show how the number of rewrite sequences that need to be considered can be reduced. We do this in Sect. 4.2 by defining normal-form decorations of the expression tree. In a decoration, we label each node in the tree with a (possibly empty) rewrite sequence. Such a rewrite sequence is called a local rewrite sequence. A decoration is in normal form if rewrite rules are applied as low as possible in the expression tree. Rewrite sequences that correspond to decorations that are not in normal form do not need to be considered.

In Sect. 4.3, we define strong normal-form decorations. These decorations have the extra property that nodes in sub-terms that match variables must be rewritten before substitution takes place. By considering only strong normal-form decorations we attempt to contain the explosion of possible rewrite sequences that can occur due to the action of variables.

Finally, in Sect. 4.4, we present an algorithm that determines the strong normal-form decorations of an expression tree. In this algorithm, we compute the input and output sets of each node in the given expression tree. The input set of a node lists all patterns that match the sub-term rooted at that node. The output set lists the results

of matching the sub-term with each of the input patterns. We also highlight in this section the similarities and differences between the theory that we have developed and the work of Pelegri-Llopart and Graham [36]. We refer to their work using the abbreviation PLG.

4.1 Costed term rewrite systems

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

Definition 4 (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.[1] 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 an infinite universe of variable symbols, and $V \subseteq \mathscr{V}$.

Definition 5 (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:*

- $V \subseteq T_\Sigma(V) \;\wedge\; \Sigma_0 \subseteq T_\Sigma(V)$
- $\forall\, a \in \Sigma_n : t_1, \ldots, t_n \in T_\Sigma(V) \;\Rightarrow\; a(t_1, \ldots, t_n) \in T_\Sigma(V)$*, for* $n \geq 1$

For term t, *Var*(t) denotes the set of variables in t. Terms t for which *Var*$(t) = \emptyset$ are called *ground terms*.

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 6 (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 is usually 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 7 (Costed term rewrite system). *A costed term rewrite system (CTRS) is a triple $((\Sigma, V), R, C)$ with*

- *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*

such that, for all $(t, t') \in R$, the following conditions are satisfied:

- $t' \neq t$

[1] The term *arity* is sometimes used instead of rank.

– $t \notin V$
– $Var(t') \subseteq Var(t)$

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 CTRS. A term rewrite system (TRS) is in that case a tuple $((\Sigma, V), R)$.

The first constraint in Definition 7 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 CTRS is called *ground* if all left-hand sides of rewrite rules are ground terms.

The CTRS defined in the following example is a slightly modified version of an example taken from PLG, and will be used as a running example throughout this section.

Example 4. Let $((\Sigma, V), R, C)$ be a CTRS, where $\Sigma = (S, r)$, $S = \{ +, c, a, r, 0 \}$, $r(+) = 2, r(c) = r(r) = r(a) = r(0) = 0$, and $V = \{ x, y \}$. Here c represents a constant, a represents an address register and r represents a general register. The set of rules R is defined as follows:

$$
\begin{array}{lll}
R = \{ & (r_1) & +(x, y) \longrightarrow +(y, x) \\
 & (r_2) & +(x, 0) \longrightarrow x \\
 & (r_3) & +(a, a) \longrightarrow r \\
 & (r_4) & +(c, r) \longrightarrow a \\
 & (r_5) & 0 \longrightarrow c \\
 & (r_6) & c \longrightarrow a \\
 & (r_7) & a \longrightarrow r \\
 & (r_8) & r \longrightarrow a \; \}
\end{array}
$$

An alternative representation of the first four elements of R is given in Fig. 5. The cost function C is defined as follows: $C(r_1) = C(r_2) = C(r_5) = 0$, $C(r_3) = C(r_6) = 3$, $C(r_7) = C(r_8) = 1$ and $C(r_4) = 5$.

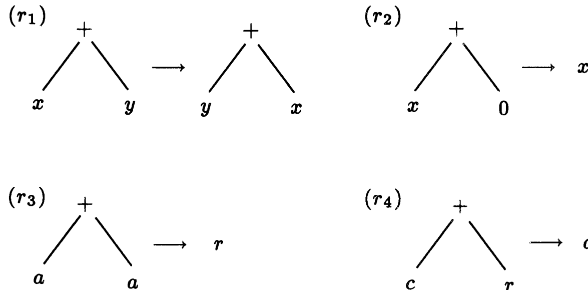

Fig. 5. The tree representation of some term rewrite rules

Some example terms are $+(0, +(c, c))$, a, and $+(x, +(0, +(c, y)))$. For $t = +(x, +(0, +(c, y)))$ we have that $Pos(t) = \{\varepsilon, 1, 2, 2\cdot 1, 2\cdot 2, 2\cdot 2\cdot 1, 2\cdot 2\cdot 2\}$. Some sub-terms of t are $t|_\varepsilon = t, t|_1 = x$, and $t|_{22} = +(c, y)$.

Definition 8 (Substitution). *Let $\sigma \in V \rightarrow T_\Sigma(V)$. 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.

Definition 9 (Rewrite rule equivalence). *Rewrite rules $r_1 : t_1 \longrightarrow t_1'$ and $r_2 : t_2 \longrightarrow t_2'$ are equivalent if and only if there is a bijection $\sigma \in Var(t_1) \rightarrow Var(t_2)$ such that $t_1^\sigma = t_2$ and $t_1'^\sigma = t_2'$.*

In this work we consider rewrite rules modulo rewrite equivalence.

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

Definition 10 (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 $t_1 \xrightarrow{\langle r,p \rangle} t_2$ if and only if t_2 can be obtained from t_1 by replacing $t_1|_p$ by t'^σ in t_1, and using the substitution σ with $t^\sigma = t_1|_p$. We can also write $\langle r, p\rangle\, t_1 = t_2$.*

(PLG refer to a rewrite step as a rewrite application.) 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 11 (Rewrite sequence). *Let $t \xrightarrow{\langle r_1,p_1\rangle \ldots \langle r_n,p_n\rangle} t'$ if and only if there 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 $S(t)\, t = t'$.*

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[2], 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 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 $\bar{\tau} = \{ r_1, \ldots, r_n \}$, that is, $\bar{\tau}$ 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 $\bar{\tau}$.

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

Definition 12 (Permutations). *Given a term t, rewrite sequences τ and τ' are permutations of each other, denoted $\tau \cong_t \tau'$, if and only if all elements in $\bar{\tau}$ and $\bar{\tau}'$ have the same cardinality, and $t \stackrel{\tau}{\Longrightarrow} t' \iff t \stackrel{\tau'}{\Longrightarrow} t'$ for all terms t'.*

Example 5. Consider the CTRS shown in Example 4, and let $t = +(0, +(r, c))$. We can write $t \stackrel{\langle r_1, 2\rangle}{\Longrightarrow} t'$, with $t' = +(0, +(c, r))$. We can also write $\langle r_1, 2\rangle\, t = t'$. The term t' is obtained from t by replacing $t|_2$ by $+(y, x)^\sigma$ in t, using substitution σ with $\sigma(x) = r$ and $\sigma(y) = c$ such that $(x, y)^\sigma = t|_2$. Two derivations starting with t' are:

1. $+(0, +(c, r)) \stackrel{\langle r_4, 2\rangle}{\Longrightarrow} +(0, a) \stackrel{\langle r_7, 2\rangle}{\Longrightarrow} +(0, r) \stackrel{\langle r_1, \varepsilon\rangle}{\Longrightarrow} +(r, 0) \stackrel{\langle r_2, \varepsilon\rangle}{\Longrightarrow} r$
2. $+(0, +(c, r)) \stackrel{\langle r_4, 2\rangle}{\Longrightarrow} +(0, a) \stackrel{\langle r_1, \varepsilon\rangle}{\Longrightarrow} +(a, 0) \stackrel{\langle r_7, 1\rangle}{\Longrightarrow} +(r, 0) \stackrel{\langle r_2, \varepsilon\rangle}{\Longrightarrow} r$

These rewrite sequences are permutations of each other and both have cost 6.

A permutation defines an equivalence relation on rewrite sequences. In the next section we will use this fact to reduce the number of rewrite sequences that we need to consider. Note that all permutations of a rewrite sequence have the same cost. This is a stipulation for our approach, and a property of a BURS. 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 the reduction, or optimisation, that we consider in the next section will lead to legal 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.

4.2 Normal-form decorations

Given a CTRS $((\Sigma, V), R, C)$ and two ground terms $t, t' \in T_\Sigma$, we now wish to determine a rewrite sequence τ such that $t \stackrel{\tau}{\Longrightarrow} t'$ with minimal cost. If we assume that such a rewrite sequence exists, then PLG refer to this as the C-REACHABILITY problem. In practice, term rewrite systems in code generation are rather extensive and allow for many possible rewrite sequences to transform t into t'. Fortunately, an optimisation is possible so that we do not need to consider all possible rewrite sequences.

This optimisation 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*. Permuted rewrite sequences yield the same result for term t (cf. Definition 12), and they have identical costs, hence 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.

In Definition 11 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 13 (Decorations). *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 14 (The 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 $n\cdot$.

Decorations are considered to be the same if and only if their corresponding rewrite sequences are permutations of each other.

Definition 15 (Decoration equivalence). *The decorations $D(t)$ and $D'(t)$ are equivalent, denoted by $D(t) \equiv D'(t)$, if and only if $S_D(t)$ and $S_{D'}(t)$ are permutations of each other, i.e. $S_D(t) \cong_t S_{D'}(t)$.*

Example 6. Consider our running example again, and let $t = +(0, +(c, c))$. Two decorations $D(t)$ and $D'(t)$ of t are depicted in Fig. 6, on the left and right, respectively. The corresponding rewrite sequences are as follows:

$$\begin{aligned} S_D(t) &= \langle r_6, 2\cdot 1\rangle\langle r_7, 2\cdot 1\rangle\langle r_1, 2\rangle\langle r_4, 2\rangle\langle r_7, 2\rangle\langle r_1, \varepsilon\rangle\langle r_2, \varepsilon\rangle \\ S_{D'}(t) &= \langle r_6, 2\cdot 1\rangle\langle r_7, 2\cdot 1\rangle\langle r_1, 2\rangle\langle r_4, 2\rangle\langle r_1, \varepsilon\rangle\langle r_7, 1\rangle\langle r_2, \varepsilon\rangle \end{aligned}$$

The decorations $D(t)$ and $D'(t)$ are equivalent because $S_D(t) \cong_t S_{D'}(t)$.

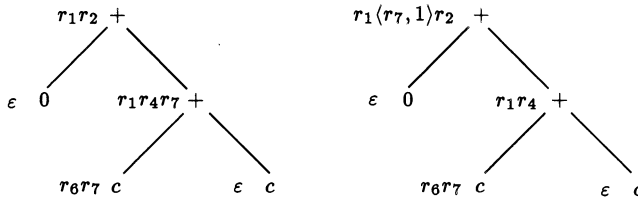

Fig. 6. Equivalent decorations $D(t)$ and $D'(t)$ of a term t

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

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

$\prec^+$ is the transitive closure of $\prec$. 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 an interesting 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 17 (Normal-form decoration). *A decoration $D(t)$ of a term t is in normal form if and only if $\neg\, (\exists\, D'(t) : D'(t) \prec^+ D(t))$.*

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

Example 7. In Example 6 we have $D(t) \prec D'(t)$ because rewrite rule r_7 associated with the root position of t in $D'(t)$ can be moved to a lower position of t in $D(t)$. As all local rewrite rules in $D(t)$ are applied to the root position of the sub-term with which they are associated, they cannot be moved to a lower position, hence $D(t)$ is in normal form.

Example 8. The two decorations shown in Fig. 7 are equivalent, and are both in normal form. This illustrates that normal forms need not be unique.

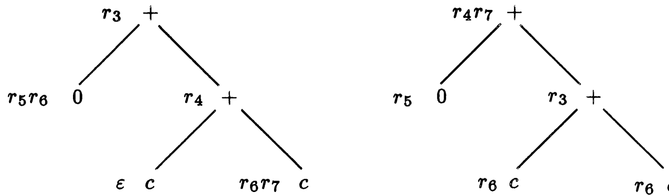

Fig. 7. Two equivalent, normal-form decorations of a term t

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

Theorem 1 (Normal-form existence). *Given a rewrite sequence τ and term t, we have that*

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

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 normal form. We informally describe a procedure to obtain from $D_0(t)$ an equivalent decoration which is in normal 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 $n \geq 0$. By construction, the last obtained decoration is a minimal element under $\prec^+$.

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 4.1

For all $D(t) \in NF(t)$, 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.

Proof. By contradiction. Let us assume $D(t) \in NF(t)$ 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 $D(t) \in NF(t)$.

The approach that has been used in this section is different from that of PLG. PLG first define a normal-form rewrite sequence, and then a local rewrite sequence and assignment. Their local rewrite assignment is the same as our decoration. We have reversed this order. Further, our approach is more formal and concise. In particular, the explicit use of the ordering relation $\prec$ is very helpful in characterising normal-form decorations.

4.3 Strong normal-form decorations

The idea behind a strong normal form is to reduce the number of local rewrite sequences that we need to consider. 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 CTRS $((\Sigma, V), R, C)$. We begin by defining the set of positions in a term at which a variable occurs.

Definition 18 (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. This leads us to the following definition.

Definition 19 (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:*

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

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

In Fig. 8 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 $t_1 \longrightarrow t_2$ 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 Fig. 8) 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'').

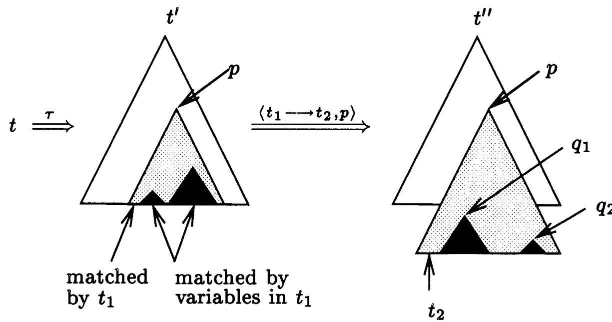

Fig. 8. Computing the rewriteable positions in a term after the application of $\langle t_1 \longrightarrow t_2, p\rangle$

In the definition above, we see that 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 9. We are given a TRS with $S = \{*, +, c, r, 2\}$, corresponding ranks $\{2, 2, 0, 0, 0\}$, $V = \{x\}$ and R defined as follows:

$$R = \{ \quad (r_1) \quad *(2, x) \longrightarrow +(x, x)$$
$$(r_2) \quad +(c, c) \longrightarrow r$$
$$(r_3) \quad +(r, r) \longrightarrow r \; \}$$

Assume that we have some term $t = *(2, +(c, c))$. Initially, the rewriteable positions in t are given by $RP_t(\varepsilon) = \{\varepsilon, 1, 2, 2{\cdot}1, 2{\cdot}2\}$. If we now apply the rewrite rule $\langle r_2, 2\rangle$, then we generate the term $t'' = *(2, r)$ with rewriteable positions:

$$\begin{array}{rcl} RP_t(\langle r_2, 2\rangle) & = & (RP_t(\varepsilon) - Pos(t|_2)) \cup Pos(t''|_2) - \emptyset \\ & = & (\{\varepsilon, 1, 2, 2{\cdot}1, 2{\cdot}2\} - \{2, 2{\cdot}1, 2{\cdot}2\}) \cup \{2\} \\ & = & \{\varepsilon, 1, 2\} \end{array}$$

In other words, each of the positions in the term $*(2, r)$ is rewriteable. Note that the rule r_2 does not contain a variable.

We now apply the rewrite rule $\langle r_1, \varepsilon\rangle$ and generate $t'' = +(r, r)$. Note that we are allowed to do this because the position ε is rewriteable, and that $t' = *(2, r)$. The rewriteable positions in this new term are:

$$\begin{array}{rcl} RP_t(\langle r_2, 2\rangle\langle r_1, \varepsilon\rangle) & = & (RP_t(\langle r_2, 2\rangle) - Pos(t'|_\varepsilon)) \cup Pos(t''|_\varepsilon) \\ & & \qquad -\{Pos(t''|_q) \mid q = 1, 2\} \\ & = & (\{\varepsilon, 1, 2\} - \{\varepsilon, 1, 2\}) \cup \{\varepsilon, 1, 2\} - \{1, 2\} \\ & = & \{\varepsilon\} \end{array}$$

Because the root position in the term $+(r, r)$ is rewriteable, we can now apply the rewrite rule $\langle r_3, \varepsilon\rangle$ and generate the goal term r. Summing up, we have reduced the term t using the following sequence:

$$*(2, +(c, c)) \xRightarrow{\langle r_2, 2\rangle} *(2, r) \xRightarrow{\langle r_1, \varepsilon\rangle} +(r, r) \xRightarrow{\langle r_3, \varepsilon\rangle} r$$

Example 10. Instead of beginning with the rule $\langle r_2, 2\rangle$ in the above example, we could have begun with the rule $\langle r_1, \varepsilon\rangle$. This results in

$$*(2, +(c, c)) \xRightarrow{\langle r_1, \varepsilon\rangle} +(+(c, c), +(c, c))$$

The only rewriteable position in the new term is the root position. To reduce the term further we need to reduce the sub-terms $+(c, c)$ at positions 1 and 2. These positions, however, are non-rewriteable, hence we cannot proceed any further. Intuitively, we say that the term $+(c, c)$ should have been rewritten before it was substituted for a variable. (This is the strong-normal-form property.)

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 20 (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,*

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

Definition 21 (Strong-normal-form decoration). *A normal-form decoration $D(t)$ is in strong normal form if and only if $Permitted_t(L_D(t|_p))$ 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.

Example 11. Let $((\Sigma, V), R)$ be a TRS with $S = \{ *, a, b, c, d, e, f \}$, $r(*) = 2$ and all others with rank 0, $V = \{ x \}$, and R defined as follows:

$$R = \{ \quad \begin{array}{ll} (r_1) & *(a,b) \longrightarrow *(c,d) \\ (r_2) & *(c,x) \longrightarrow *(e,x) \\ (r_3) & d \longrightarrow f \ \} \end{array}$$

Let $t = *(a,b)$, and define a decoration $D(t)$ by local rewrite sequences $L_D(t) = r_1\, r_2\, \langle r_3, 2\rangle$ and $L_D(t|_1) = L_D(t|_2) = \varepsilon$. The decoration $D(t)$ is in normal form, but not in strong normal form, because r_2 makes position 2 non-rewriteable. Rule r_3 may not be applied to this position. The value of $\mathit{Permitted}_t(r_1\, r_2\, \langle r_3, 2\rangle)$ is therefore false.

Note, however, that the decoration $D'(t)$ with $L_{D'}(t) = r_1\, \langle r_3, 2\rangle\, r_2$ and $L_{D'}(t|_1) = L_{D'}(t|_2) = \varepsilon$ is in strong normal form.

The following theorem means that we only need to consider strong-normal-form decorations of a term t, and not the entire universe of normal-form decorations of t. This theorem is analogous to Theorem 1.

Theorem 2 (Strong normal-form existence). *Given a rewrite sequence τ and term t, we have that*

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

Proof. Let τ be an arbitrary rewrite sequence and t some term such that $t \xRightarrow{\tau}$. From Theorem 1 it follows that there exists a normal-form decoration $D(t)$ corresponding to τ. If $D(t)$ is not in strong normal form, then there is some p such that $\neg\,\mathit{Permitted}_t(L_D(t|_p))$. This means that we can write

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

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. We depict the above rewrite sequence in Fig. 9. In this figure, $t' = t|_p$. The application of the rewrite rule $t_a \longrightarrow t_b$ at position p''' in the term t''' in this figure is not permitted because the (earlier) application of the rule $t_1 \longrightarrow t_2$ at position p' in t' resulted in all the positions in t_v, including t_a, becoming non-rewriteable in t''. In Fig. 9, the terms t_1 and t_2 are shown lightly shaded, t_v is shown heavily shaded, and t_a is shown in black. Without loss of generality, we assume that the positions in t_a are rewriteable before the application of $\langle t_1 \longrightarrow t_2, p'\rangle$ (note that, by definition, all positions are initially rewriteable), and that $\mathit{Permitted}_{t''}(\tau_1)$ is true.

We now move the instance of the rewrite rule $t_a \longrightarrow t_b$ to *before* the rule $t_1 \longrightarrow t_2$. This results in the rewrite sequence shown in Fig. 10. In this figure we see that the rule $t_a \longrightarrow t_b$ is applied at position p^o in term t^o, before the rewrite step $\langle t_1 \longrightarrow t_2, p'\rangle$. If we now apply the rule $t_1 \longrightarrow t_2$, we find that the sub-term t_v that matches variable v will contain t_b (instead of t_a). Note that the rule $t_a \longrightarrow t_b$ is applied at position p''' in the sequence in Fig. 9, and at position p^o in the sequence in Fig. 10. The new rewrite sequence in Fig. 10 is quite obviously a permutation of the sequence in Fig. 9.

The above procedure moves a rewrite step that is applied to a sub-term that is non-rewriteable to before the rewrite step that made the sub-term non-rewriteable. Applying this procedure repeatedly will result in a local rewrite sequence $L_D(t|_p)$ that is permissible.

Given a normal-form decoration $D(t)$, therefore, we can now make each local rewrite sequence $L_D(t|_p)$, for all $p \in \mathit{Pos}(t)$, permissible. This results in a strong-normal-form decoration. This completes the proof.

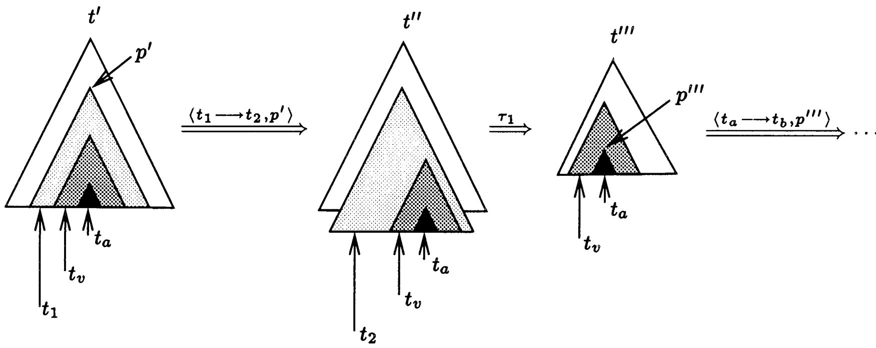

Fig. 9. A sequence that is not in strong normal form

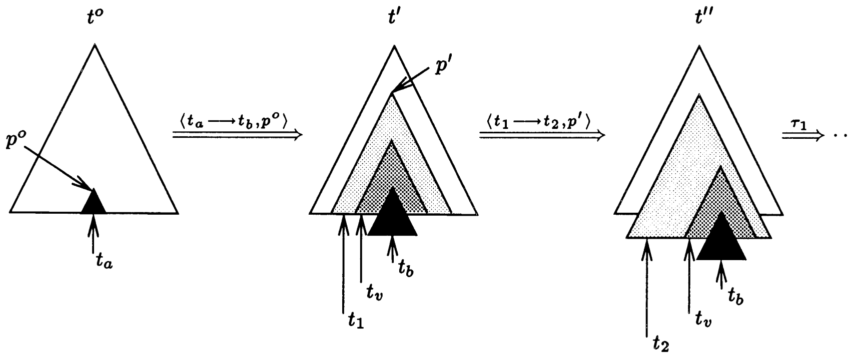

Fig. 10. A sequence that is in strong normal form

Example 12. Consider the TRS $((\Sigma, V), R)$ with $S = \{+, c, r, a\}$, $r(+) = 2$ and all others with rank 0, $V = \{x, y\}$, and R defined as follows:

$$
\begin{aligned}
R = \{ \quad & (r_1) \quad +(c, +(c, x)) \longrightarrow +(c, x) \\
& (r_2) \quad +(x, y) \longrightarrow +(y, x) \\
& (r_3) \quad a \longrightarrow r \\
& (r_4) \quad +(r, r) \longrightarrow r \\
& (r_5) \quad +(c, r) \longrightarrow r \ \}
\end{aligned}
$$

Let $t' = +(+(c, +(c, +(a, r))), c)$, and consider the sequence shown in Fig. 11. This sequence, which is *not* in strong normal form, rewrites the term t' into the goal term r. In the first step in this sequence, t' is rewritten into t'' by $\langle r_1, 1 \rangle$. In this step, the sub-term t_v (indicated in the figure) in t' is matched by the variable x in t_1, and as a result, all the corresponding positions in the term t'', including t_a, have become non-rewriteable (indicated by the circled nodes). The rewrite steps $r_2 r_1$ are then applied, resulting in the term t''', where all positions are non-rewriteable except the root. The next rewrite step, $\langle r_3, 2{\cdot}1 \rangle$, is not permitted because position 2·1, which is the position of term t_a, is non-rewriteable.

The node corresponding to position 2·1 in t''' became non-rewriteable as a result of the first step, $\langle r_1, 1 \rangle$. Following the strategy outlined in the proof of Theorem 2, we

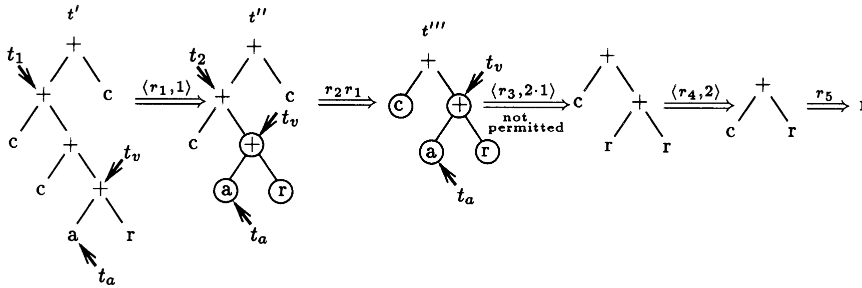

Fig. 11. A sequence that is *not* in strong normal form

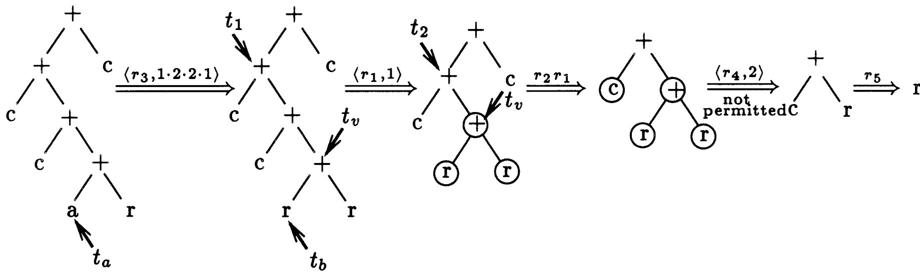

Fig. 12. Another sequence that is *not* in strong normal form

now move the rule r_3 that we were not permitted to apply above, to before this step. The resulting sequence is shown in Fig. 12. In this sequence we begin by applying the rewrite step $\langle r_3, 1{\cdot}2{\cdot}2{\cdot}1\rangle$. This step rewrites t_a into t_b. We then apply rewrite step $\langle r_1, 1\rangle$, which results in the positions in the sub-term t_v becoming non-rewriteable. We can next apply steps r_2r_1, but we cannot apply rule r_4 at position 2 because it is non-rewriteable. As before, this node has became non-rewriteable as a result of the earlier step $\langle r_1, 1\rangle$. Repeating the above procedure, and moving the application of rule r_4 to before this step, results in the sequence shown in Fig. 13. This sequence is in strong normal form. Notice that in moving the rules r_3 and r_4 forward, the positions at which these rules are applied change.

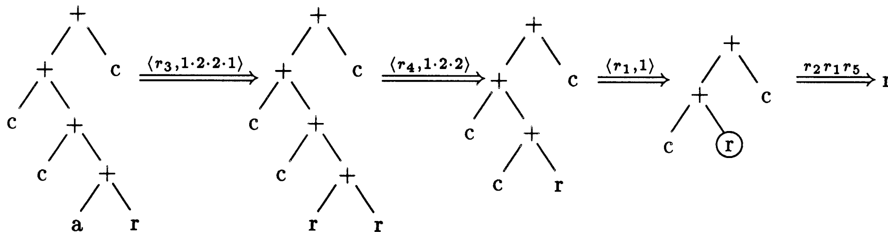

Fig. 13. This sequence is in strong normal form

In this section we have formalised the concept of rewriteable positions and strong-normal-form decorations. Rewriteable positions are related to PLG's *touched positions*, which PLG only treats cursorily. PLG does not explicitly define a strong normal form.

4.4 Input and output sets

As a direct generalisation of bottom-up tree pattern matching methods (see e.g. [24]), sets of patterns, called input and output sets, can be computed from the strong-normal-form decorations of t. These sets define the patterns that match the expression tree. We begin by defining the inputs and outputs of a decoration.

Definition 22 (Inputs of a decoration). *Let $D(t) \in SNF(t)$ 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:*

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

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

Definition 23 (Outputs of a decoration). *Let $D(t) \in SNF(t)$ 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 *$I_D(t) \xRightarrow{L_D(t)} t'$.*

Using the inputs and outputs, we can now define the *input set* and *output set* of a term t for some goal term g. The input set $IS_g(t)$ is the union of all possible inputs for all strong-normal-form decorations of t. Similarly for the *output set* $OS_g(t)$. More formally:

$$\begin{aligned} IS_g(t) &= \{\, I_D(t) \mid D(t) \in SNF(t) \ \wedge \ t \xRightarrow{S_D(t)} g \,\} \\ OS_g(t) &= \{\, O_D(t) \mid D(t) \in SNF(t) \ \wedge \ t \xRightarrow{S_D(t)} g \,\} \end{aligned}$$

Note that the sets are defined for a specific goal term g.

Example 13. Consider again our running example and the term t given by $+(0, +(c, c))$. A normal-form decoration $D(t)$ for this term is shown on the left in Fig. 6. The inputs $I_D(t)$ and outputs $O_D(t)$ of this decoration for goal term r are depicted in Fig. 14a. In this figure, inputs and outputs are given on the left and right side (resp.) of each node.

The input sets $IS_r(t)$ and output sets $OS_r(t)$ of this term t for goal term r are shown in Fig. 14b, on the left and right side (resp.) of each node.

An algorithm to calculate input and output sets for terms t and g, and the corresponding local rewrite sequences is given in Fig. 15. This algorithm consists of two passes. In the first, bottom-up pass (see the function *Generate*) sets of *triples*, denoted by $W(t)$, are computed for all possible goal terms. A triple, written $\langle it, D(t), ot\rangle$,

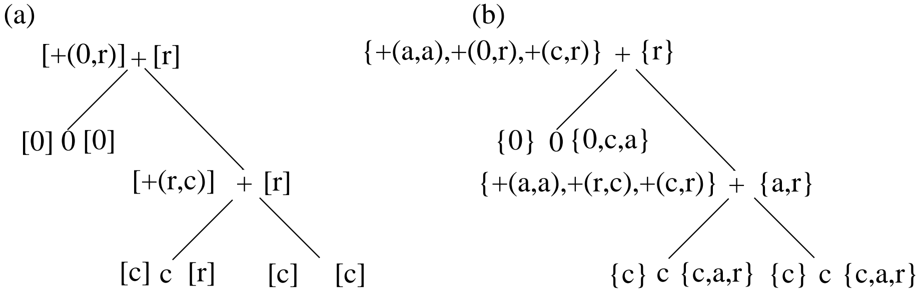

Fig. 14. a Inputs and outputs of a decoration for the term $+(0,+(c,c))$. **b** The input and output sets of this term

consists of an input it, decoration $D(t)$, and output ot such that $t \xRightarrow{S_D} ot$, and $it \xRightarrow{L_D(t|_\varepsilon)} ot$. For convenience, we use the type Term to denote T_Σ, SetOfTerms to denote $\mathscr{P}$(Term), Triple to denote Term×Decoration×Term, and SetOfTriples to denote $\mathscr{P}$(Triple).

In the second, top-down pass (the function *Trim*), these sets of triples are 'trimmed' using the desired goal term g. The root node is trimmed by removing each triple whose output term is not identical to the goal term. Other nodes in the expression tree are trimmed by removing each triple whose output term is not identical to an input term of its parent node. The resulting trimmed sets of triples, denoted by $V(t)$, consist of the input and output sets, and the associated decorations. Under some circumstances it may be possible to trim the nodes in the expression tree while they are being generated (see for example [38]). We do not consider that aspect further here however.

Example 14. Let us apply the algorithm shown in Fig. 15 to our running example. The set of triples $V(t)$ for $t = +(0,+(c,c))$ is shown below. Note that all rewrite rules are applied at the root.

$$
\begin{aligned}
V(t|_1) &= \{\langle 0,\varepsilon,0\rangle,\ \langle 0,r_5,c\rangle,\ \langle 0,r_5r_6,a\rangle\}\\
V(t|_{21}) &= \{\langle c,\varepsilon,c\rangle,\ \langle c,r_6,a\rangle,\ \langle c,r_6r_7,r\rangle\}\\
V(t|_{22}) &= \{\langle c,\varepsilon,c\rangle,\ \langle c,r_6,a\rangle,\ \langle c,r_6r_7,r\rangle\}\\
V(t|_2) &= \{\langle +(a,a),r_3,r\rangle,\ \langle +(a,a),r_3r_8,a\rangle,\ \langle +(r,c),r_1r_4,a\rangle,\\
&\qquad \langle +(r,c),r_1r_4r_7,r\rangle,\ \langle +(c,r),r_4,a\rangle,\ \langle +(c,r),r_4r_7,r\rangle\}\\
V(t|_\varepsilon) &= \{\langle +(a,a),r_3,r\rangle,\ \langle +(0,r),r_1r_2,r\rangle,\ \langle +(c,r),r_4r_7,r\rangle\}
\end{aligned}
$$

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 $D(t) \in \mathit{SNF}(t)$, $t \in T_\Sigma(V)$ and $p \in \mathit{Pos}(t)$.

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.

```
|[ 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) := Checknf(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) := Checknf(Z(t), ⟨it, D ⊗ ⟨r, p⟩, ot'⟩)
                               fi
                            od
                         od
                      od
                    ]|
    od;
    return Z(t)
   ]|;
  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
   ]|;
  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)
   ]|;
  W(t) := Generate(t);
  V(t) := Trim(t, {g})
]|.
```

Fig. 15. A two-pass algorithm to calculate the input sets, decorations and output sets

Example 15. Consider the TRS with rules:

$$
\begin{array}{ll}
(r_1) & c \longrightarrow m(c) \\
(r_2) & m(c) \longrightarrow a \\
(r_3) & m(a) \longrightarrow r
\end{array}
$$

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

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

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

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

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 16. Consider the TRS with rules:

$$
\begin{array}{ll}
(r_1) & m(+(c, X)) \longrightarrow m(X) \\
(r_2) & m(r) \longrightarrow r
\end{array}
$$

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

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

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

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

If the length of local rewrite sequences for a given TRS is bounded, by k say, then PLG say that the TRS satisfies the BURS property. (Note that our running example satisfies the BURS property with $k = 3$.)

PLG define the following sufficient syntactic condition for a finite TRS:

Theorem 3. *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* $a, b \in \Sigma_n$
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 [32]. It would be interesting to identify a coarser classification of finite TRSs.

PLG define inputs and outputs, as we do, and use these to build *local rewrite graphs* for each sub-term of the given expression tree. These graphs represent the local rewrite sequences of all 'normal-form rewrite sequences' that are applicable. We directly encode the inputs, outputs and local rewrite sequences into the expression tree (in the form of triples attached to each node).

5 Coupling A* and BURS

The search graph $G = (N, E, n_0, N_g)$ consists of a set of nodes N, edges E and goal nodes N_g, and an initial node n_0. A node represents a state of the system, and is denoted by a quadruple (t, p, τ, t') where t is the current term, p is the current position in that term, τ the local rewrite sequence applied at p, and t' the (chosen) input tree at p.

The initial node n_0 is given by the quadruple $(t_I, p_0, \epsilon, t_I|_{p_0})$. The term t_I is the input expression tree for which we want to generate code. The initial position p_0 is the lowest left-most position in this tree, and is of the form $1 \cdot 1 \cdot \ldots$

Example 17. Consider our running example (Example 4). The initial node is the quadruple $(+(0, +(c, c)), 1, \epsilon, 0)$. The lowest left-most position in $t_I = +(0, +(c, c))$ is 1, and $t_I|_1$ is 0. The set of goal terms is the singleton set $\{r\}$.

To determine the search graph, we need to compute the successor nodes of a given node. This is carried out by the function *Successor*, which is shown in Fig. 16. As in Fig. 15, we use the type SetOfTriples. We also denote the type $(R \times \mathbb{N}_+^*)^*$ by RewriteSequence, and $\mathscr{P}(T_\Sigma \times \mathbb{N}_+^* \times (R \times \mathbb{N}_+^*)^* \times T_\Sigma)$ by SetOfQuadruples.

The (standard) functions *Next*, *Parent* and *Child* are used to position ourselves in the search graph. These functions are defined below.

Definition 24 (Next, Parent and Child). *Given a position $p \in Pos(t) \setminus \{\epsilon\}$ in a term t:*

- *$Next(p, t) \in \mathbb{N}_+^*$ is the next position of p in a post-order traversal of t.*
- *$Parent(p, t) \in \mathbb{N}_+^*$ is the position of the parent of p in t.*
- *$Child(p, t) \in \mathbb{N}_+$ is the child-number of p in t.*

If a position p in tree t has children $p{\cdot}1, \ldots, p{\cdot}n$ then the *child-number* of position $p{\cdot}i$ is i. Further, $Parent(\epsilon,t) = Child(\epsilon,t) = \epsilon$, but $Next(\epsilon,t)$ is undefined, for any t. Note that $p = Parent(p,t){\cdot}Child(p,t)$.

Example 18. In the term $t = +(0, +(c, c))$, we have $Next(1,t) = 2{\cdot}1$, $Next(2{\cdot}1,t) = 2{\cdot}2$, $Next(2{\cdot}2,t) = 2$ and $Next(2,t) = \epsilon$. Furthermore, $Parent(2{\cdot}1,t) = 2$ and $Child(2{\cdot}1,t) = 1$.

The basic idea behind the successor function is the following. If we can add a rewrite step ($\langle r, p'\rangle$ in the algorithm) to a local rewrite sequence (τ) at the current position (p), and there exists a rewrite sequence ($\tau\langle r, p'\rangle\tau'$) whose output tree (*ot*) matches a corresponding child of an input tree (*it*$'$) of the parent (of p), and all the *younger siblings* of the current position also match corresponding children of the same input tree, then we have found a successor node. If position p is the i-th child (of a node) then the younger siblings are the children 1 until $i - 1$. The first child has no younger siblings of course. The function *Successor* is called recursively, using the next post-order position, for as long as the sub-term at the current position, and all the younger siblings of the current position, match corresponding children of an input tree of the parent. The function *Match* carries out the task of matching a node (sub-tree) and its siblings with the children of an input tree of the parent.

When the algorithm reaches the root position, $p = \epsilon$, the recursion will stop, and the function *Match* will always yield true. The algorithm will return with the empty set when it reaches the root position and the term $t \in N_g$.

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

  func Successor(t : T_Σ, p : IN*_+, τ : RewriteSequence, it : T_Σ): SetOfQuadruples
   |[ var S: SetOfQuadruples;

    func Match(p' : IN*_+, t' : T_Σ): Bool
     |[ var Z: SetOfTerms ;
           it': Term ;
           b: Bool ;
      b := (p' = ε);
      Z(t) := { it | ⟨it, D, ot⟩ ∈ V(t|_Parent(p')) ∧ it|_Child(p') = t' };
      (* if p' is the i-th child, then Z is the set of input trees of its
         parent such that the i-th child of each term in Z equals t' *)
      do Z ≠ ∅ ∧ ¬b ⟶|[ choose it' ∈ Z;
                              Z := Z \ {it'};
                              b := (∀ 1 ≤ i < Child(p') : it'|_i = t|_Parent(p')·i)
                           ]|
      od;
      return b
     ]|;

    (* body of function Successor *)
    if (p = ε) ∨ ¬Match(p, t|_p) ⟶ S := ∅
     [] (p ≠ ε) ∧ Match(p, t|_p) ⟶ S := Successor(t, Next(p), ε, t|_Next(p))
    fi ;
    for all r ∈ R
    do for all p' ∈ Pos(it)
        do for all ⟨it, D, ot⟩ ∈ V(t|_p) ∧ S_D = τ⟨r, p'⟩τ'   (* loop over τ' and ot *)
            do if ¬Match(p, ot) ⟶ skip
                 [] Match(p, ot) ⟶ S := S ∪ { (⟨r, p'⟩t, p, τ⟨r, p'⟩, it) }
               fi
            od
        od
    od;
    return S
  ]|
]|.
```

Fig. 16. The successor function that computes a set of new search nodes

Example 19. Consider our running example again. Let us compute the successor nodes of the initial node, i.e. we compute $\mathit{Successor}(+(0,+(c,c)),1,\epsilon,0)$. Because $p \neq \epsilon$ and $\mathit{Match}(1,\ t|_1) = \mathit{true}$, we recursively call the function again with the next position, $p = 2{\cdot}1$. That is, we call $\mathit{Successor}(+(0,+(c,c)),2{\cdot}1,\epsilon,c)$ where the last argument $c = t|_{2\cdot 1}$. Again, $p \neq \epsilon$ and $\mathit{Match}(2{\cdot}1,\ t|_{2\cdot 1}) = \mathit{true}$, so we recursively call *Successor*, this time with $p = 2{\cdot}2$. That is, we call $\mathit{Successor}(+(0,+(c,c)),2{\cdot}2,\epsilon,c)$ where the last argument $c = t|_{2\cdot 2}$. The recursion now stops because $\mathit{Match}(2{\cdot}2,\ t|_{2\cdot 2}) = \mathit{false}$ (there is no input tree it' of $t|_2$ in which $it'|_1 = c \ \wedge\ it'|_2 = c$). We therefore let $S := \emptyset$, and inspect all the triples of $V(t\,|_{2\cdot 2})$. The triples at each position in t were computed in Example 14. The triple $\langle c, r_6 r_7, r\rangle$ satisfies the loop condition, with $it = c$, $\tau = \epsilon$, $r = r_6$, $\tau' = r_7$ and $ot = r$. We also find $\mathit{Match}(2{\cdot}2, r) = \mathit{true}$ (for $it' = +(c,r)$), hence we generate the search node $(+(0,+(c,a)),2{\cdot}2,r_6,c)$. The call of *Successor* for $p = 2{\cdot}2$ is now complete, so we need to inspect the triples associated

with the previous position, $V(t|_{2\cdot1})$. The triple $\langle c, r_6r_7, r\rangle$ (again) satisfies the loop condition, $Match(2\cdot1, r) = true$ (this time for $it' = +(r, c)$), and we generate the search node $(+(0, +(a, c)), 2\cdot1, r_6, c)$ The call of *Successor* for $p = 2\cdot1$ is also now complete. Inspecting the triples associated with the initial position, $V(t|_1)$, we find that triple $\langle 0, r_5r_6, a\rangle$ satisfies the loop condition, and that $Match(1, a) = true$ (for $it' = +(a, a)$). We therefore generate the search node $(+(c, +(c, c)), 1, r_5, 0)$. The result of the above computation is that we have generated the following set of search nodes:

$$\{(+(0, +(c, a)),\ 2\cdot2,\ r_6,\ c),\ (+(0, +(a, c)),\ 2\cdot1,\ r_6,\ c),\ (+(c, +(c, c)),\ 1,\ r_5,\ 0)\}$$

In Fig. 17 we see the complete search graph for the expression tree $+(0, +(c, c))$. Note that instead of the lengthy quadruple notation, we have used the first argument of the quadruple (the term) as node name, and we have labelled the edges with the rule number concatenated with the position at which the rule must be applied. For the sake of convenience, we have also named some of the nodes. For example, if we are at node A, which is the term $+(0, +(c, c))$, and apply the rule r_5 at position 1, then we generate node B, which is $+(c, +(c, c))$.

We can construct more nodes in the search graph by computing the successors of B, C and D. We will consider just the node B here, and compute $Successor(+(c, +(c, c)),\ 1,\ r_5,\ 0)$. Because $p \neq \epsilon$ and $Match(1,\ t|_1) = true$, we must first recursively call $Successor(+(c, +(c, c)),\ 2\cdot1,\ \epsilon,\ c)$. Again, $p \neq \epsilon$ and $Match(2\cdot1,\ t|_{2\cdot1}) = true$, so we call $Successor(+(c, +(c, c)),\ 2\cdot2,\ \epsilon,\ c)$. The triple $\langle c, r_6, a\rangle \in V(t|_{2\cdot2})$ hence this last call generates the node $(+(c, +(c, a)),\ 2\cdot2,\ r_6,\ a)$. The same triple results in the previous call generating the node $(+(c, +(a, c)),\ 2\cdot1,\ r_6,\ a)$. This leaves us with only the initial call of *Successor*. For $it = 0$, $\tau = r_5$, $r = r_6$, $\tau' = \epsilon$ and $ot = a$ we find that the triple $\langle 0, r_5r_6, a\rangle \in V(t|_1)$, and hence generate the node $(+(a, +(c, c)),\ 2\cdot1,\ r_5r_6,\ 0)$. Note that $\tau \neq \epsilon$ here – this is the first time that we have built-on a rewrite sequence. The end result is that the set of successor nodes of $(+(c, +(c, c)),\ 1,\ r_5,\ 0)$ is:

$$\{(+(c, +(c, a)),\ 2\cdot2,\ r_6,\ c),\ (+(c, +(a, c)),\ 2\cdot1,\ r_6,\ c),\ (+(a, +(c, c)),\ 1,\ r_5r_6,\ 0)\}$$

These nodes correspond to the nodes G, F and E in Fig. 17, respectively.

We arrive at a goal node when a node consists of a goal term. A goal node has no successor nodes. Note that there are a total of 11 paths leading from the initial node to a goal node in Fig. 17.

Table 3. The initial steps that the A* procedure takes to reduce $+(0, +(c, c))$ using a best-first search strategy (the subscripts are the costs g)

Step	N_o	N_c	Choose
1	A_0	ϵ	A
2	$B_0C_3D_3$	A	B
3	$C_3D_3E_3F_3G_3$	AB	G
4	$C_3D_3E_3F_3O_4$	ABG	F
5	$C_3D_3E_3O_4M_4N_6$	$ABGF$	E
6	$C_3D_3O_4M_4N_6K_6L_6$	$ABGFE$	D
7	$C_3O_4M_4N_6K_6L_6J_4$	$ABGFED$	C
8	$O_4M_4N_6K_6L_6J_4H_4I_6$	$ABGFEDC$	H
9	$O_4M_4N_6K_6L_6J_4I_6P_4$	$ABGFEDCH$	
10	...	...	

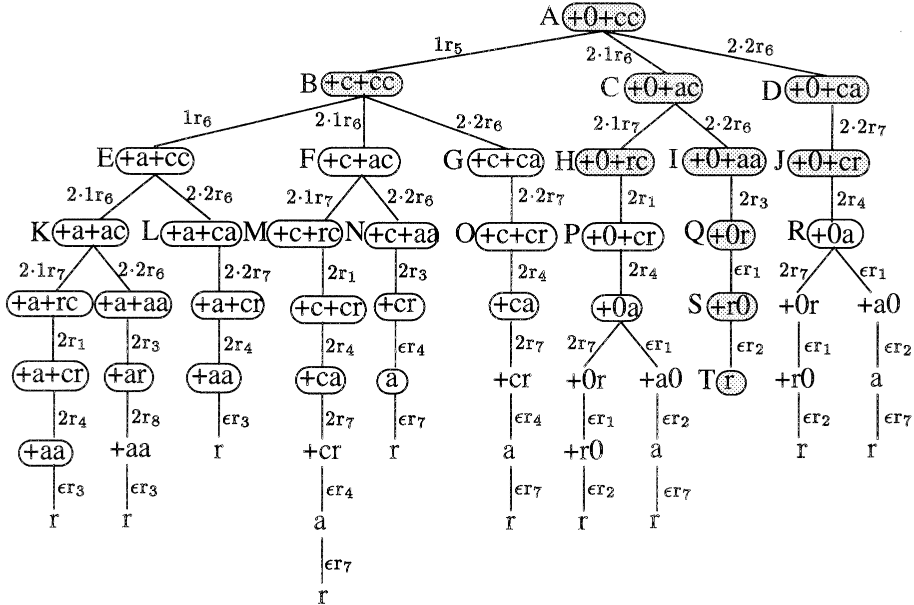

Fig. 17. For the expression tree $+(0, +(c, c))$, the (a) complete search graph, (b) best-first search graph (in boxes), and (c) heuristic search graph (in shaded boxes)

In the example above, we have shown how the successor function shown in Fig. 16 can be used to compute the complete search graph for a given rewrite system and expression tree. Calling the successor function for each and every newly created node can result in a very large tree, and is wasteful as we only wish to find one least-cost path. We could instead call the successor function from the A* search algorithm, shown in Fig. 2. The A* algorithm will compute successors for only those nodes that potentially lie on a least-cost path from the initial node n_0 to some goal node. The cost $g(n)$ of a path from n_0 to some node is simply the sum of the costs of the rewrite rules applied along the path. For the moment we let the heuristic cost function $h^*(n) = 0$, hence the cost that A* uses, $f^*(n) = g(n)$. This corresponds to a *best-first* search.

Example 20. We now apply the best-first search algorithm to our running example. We begin by initialising the sets N_c to $\emptyset$ and N_o to the initial node $\{A\}$. The successors of A are B_0, C_3 and D_3, where the subscripts are the values of the costs of the nodes. Hence, in step 2, $N_o = \{B_0, C_3, D_3\}$ and we move A to N_c. The node B is the least expensive, so we compute its successors, which are nodes E_3, F_3 and G_3, add them to N_o, and move B to N_c. The first 9 steps in this process are shown in Table 3. Note that we always choose the last computed least-expensive node. The resulting best-first search graph is shown in Fig. 17. The rewrite sequence associated with the optimal path to T is $\langle 2{\cdot}1, r_6\rangle\langle 2{\cdot}2, r_6\rangle\langle 2, r_3\rangle\langle \epsilon, r_1\rangle\langle \epsilon, r_2\rangle$. This sequence rewrites the expression tree $+(0, +(c, c))$ into r for a total cost of 9.

In the example above, we still had to compute a large part of the search tree to determine a least-cost path. We can do better by using the A* algorithm with a non-zero cost heuristic $h^*(n)$. In principle, of course, we cannot predict how much it will cost to rewrite a given node to a goal node. However, we can provide an (under) estimate of the cost. In particular we are interested in predicting when 'expensive' rewrite rules will be necessary to rewrite a term.

Example 21. Let us deduce a heuristic function that will 'predict' the cost for our running example. For convenience, the rewrite rules are shown again below. The column on the right are the costs.

$$
\begin{array}{llr}
(r_1) & +(x,y) \longrightarrow +(y,x) & 0 \\
(r_2) & +(x,0) \longrightarrow x & 0 \\
(r_3) & +(a,a) \longrightarrow r & 3 \\
(r_4) & +(c,r) \longrightarrow a & 5 \\
(r_5) & 0 \longrightarrow c & 0 \\
(r_6) & c \longrightarrow a & 3 \\
(r_7) & a \longrightarrow r & 1 \\
(r_8) & r \longrightarrow a & 1
\end{array}
$$

The heuristic cost is a function of the term t in a node n. The first observation that we make is that a +-node that does not have a 0-node as child will cost at least 3 to rewrite to our goal r. The second observation is that a node c will also cost at least 3 to rewrite. Note the special case $+(c,r)$ satisfies both conditions and costs 5+1 to rewrite to r. Combining these observations, we produce the following heuristic:

$$h^*(n) = 3 \ * \ (|+_0| \ + \ |c|)$$

where $n = (t, p, \tau, t')$, $|+_0|$ denotes the number of nodes labelled + and with no children labelled 0, and $|c|$ denotes the number of nodes labelled c. This heuristic cost under-estimates, or is equal to, the actual cost. For example, $h^* = 0$ for $t = a$ (actual cost is 1), $h^* = 3$ for $t = +(0,c)$ (actual cost 4) and $h^* = 6$ for $t = +(c,a)$ (actual cost 6).

Table 4. The steps that the A* procedure takes to reduce $+(0, +(c, c))$ using a heuristic search (the subscripts are the costs $g + h^*$)

Step	N_o	N_c	Choose
1	A_{0+9}	ϵ	A
2	$B_{0+15}C_{3+6}D_{3+6}$	A	D
3	$B_{0+15}C_{3+6}J_{4+6}$	AD	C
4	$B_{0+15}J_{4+6}H_{4+6}I_{6+3}$	ADC	I
5	$B_{0+15}J_{4+6}H_{4+6}Q_{9+0}$	$ADCI$	Q
6	$B_{0+15}J_{4+6}H_{4+6}S_{9+0}$	$ADCIQ$	S
7	$B_{0+15}J_{4+6}H_{4+6}T_{9+0}$	$ADCIQS$	goal

We now apply the A* search algorithm with this heuristic to our running example. The steps that the algorithm takes are shown in Table 4. The nodes in N_o this time have subscripts $g + h^*$. The goal node T, with a (minimum) cost of $g = 9$, is found in 7 steps. The resulting heuristic search graph is shown in Fig. 17. In total, only 10 nodes needed to be visited before the optimal path was discovered.

Implementation. The A*, pattern-matching and successor-function algorithms have been implemented in C. The A* algorithm is almost completely application-independent. An application can be the 8-puzzle, or the BURS pattern matcher, for example.

The A* algorithm calls a) a routine to initialise the application, b) the successor function to determine new nodes, and c) a simple cost function that returns with the cost of an 'edge' (i.e. rewrite rule). These 3 routines comprise the interface between A* and the application. The A* algorithm (i.e. Algorithm 2) required approximately 500 lines of code.

Implementing BURS (Algorithm 15) and the successor function (Algorithm 16) was an involved task. It required approximately 2500 lines of code, and consists of mainly intricate tree-manipulation routines. TRSs for real machines have not, as yet, been developed, hence meaningful performance figures cannot be given. However, the implementation has revealed the strength and validity of the theory. Consider, for example, the role that the strong normal form plays in reducing the number of rewrite sequences that need to be generated in the BURS. In a rewrite sequence that is in strong normal form, rewrite steps are not applied at positions that result from the substitution of a variable. Without this restriction, the number of (local) rewrite sequences can grow exponentially. This growth is caused by the rewrite rules that contain variables.

Example 22. Consider the term $+(r, +(a, a))$, and the rewrite rules from our running example. The number of local rewrite sequences in strong normal form that can be applied at the root of this term is 1, and the length of this sequence is also 1. The sequence is $+(r, +(a, a)) \stackrel{r_1}{\Longrightarrow} +(+(a, a), r)$. We cannot apply any more rewrite steps to the output term here because the sub-terms $+(a, a)$ and r have become non-rewriteable.

Now consider the local rewrite sequences that are not in strong normal form (SNF). With no restriction on where we apply rewrite steps, we can generate many rewrite sequences. For example, we could apply $\langle r_7, 1{\cdot}1\rangle$, or $\langle r_7, 1{\cdot}2\rangle$, or $\langle r_8, 2\rangle$ to the output term above. If we continue this process, we will quickly find that a combinatorial explosion ensues. In fact, the total number of non-SNF local rewrite sequences turns out to be 335,481! The lengths of these rewrite sequences range between 1 and 21. In Fig. 18 we show an example of one of the longest sequences. Notice that the rewrite rule r_1 is applied a total of 8 times in this sequence.

Note that the sequences that we referred to in the previous example have not been *trimmed*. In other words, and to be more specific, these are the sequences τ in $W(t)$ that are generated by the routine *Generate()* in Algorithm 15, where the non-SNF sequences have been generated by not enforcing the restriction $p \in RP_t(\tau)$.

We chose the term $+(r, +(a, a))$ in the previous example because the number of non-SNF rewrite sequences for our 'running' term $+(0, +(c, c))$ is too large to be easily computed. In the table below we show the number of sequences, both strong and non-strong, for each node in the term $+(0, +(c, c))$. For completeness, we also show the number of sequences after trimming (c.f. Example 14).

position	non-trimmed		trimmed
	SNF	non-SNF	
$t\|_1$	4	4	3
$t\|_{2{\cdot}1}$	3	3	3
$t\|_{2{\cdot}2}$	3	3	3
$t\|_2$	21	215	6
$t\|_\varepsilon$	101	$\gg 10^6$	3

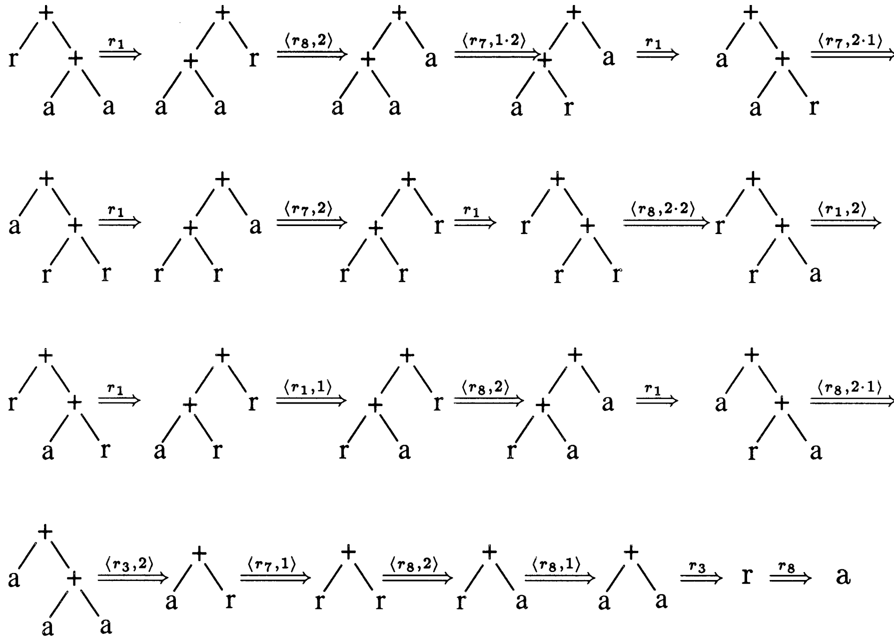

Fig. 18. A rewrite sequence of length 21

6 Conclusions

In this work we have derived BURS theory, and used this theory to construct an algorithm that determines all the pattern matches (in the form of input and output sets) of a given expression tree. BURS theory is based on term rewrite systems, which provide a more powerful formalism in the field of code generation than the more popular regular tree grammars. Given the input and output sets, the A* search algorithm is used to select patterns. Instead of computing the cost of all possible matches, the A* algorithm uses a heuristic best-first technique that applies only those rewrite rules that may form part of an optimal rewrite sequence. The cost criterion that is used is based on the costs of the rewrite rules and a heuristic that estimates the cost of rewriting a term into a goal term.

The main contributions of this work are threefold:

- We have provided a theoretical framework for BURS and presented an algorithm for pattern matching based on this framework.
- We have coupled this pattern-matching algorithm with a search algorithm to produce a code generator that generates optimal code.
- We have introduced the novel concept of a heuristic that predicts the minimum (future) cost of rewriting a term into a goal term.

The algorithms presented in this work have all been implemented. This has demonstrated the correctness of the approach, and allowed experimentation with the heuristic cost function. Note that, like the term rewrite system itself, the cost heuristic is determined by the compiler writer. We should emphasise that only when the heuristic

cost under-estimates the actual cost is optimality guaranteed. In that case the search algorithm is said to be admissible (see Sect. 3).

There are a number of directions for future research:

- Develop term rewrite systems for real machines, and test the performance of the prototype.
- Develop a systematic technique of constructing a heuristic cost function. Further, determine the sufficient and necessary conditions under which a given heuristic will not over-estimate the real cost.
- Investigate whether code optimisation and register allocation can be expressed in terms of a term rewrite system.
- Investigate whether certain parts of the pattern-matching algorithm can be done statically.
- Consider how to determine *a priori* whether a term rewrite system is finite.

Acknowledgement. Ymte Westra and Henk Alblas were involved in the initial phase of this work. Many thanks to the referees for their helpful suggestions.

References

1. Aho, A.V., Ganapathi, M., Tjiang, S.W.K.: Code generation using tree matching and dynamic programming. ACM Trans. on Prog. Lang. and Sys. **11**(4), 491–516 (1989)
2. Aho, A.V., Johnson, S.C.: Optimal code generation for expression trees. J. ACM **23**(3), 488–501 (1976)
3. Aho, A.V., Johnson, S.C., Ullman, J.D.: Code generation for machines with multiregister operations. In Proc. of the Fourth Ann. ACM Symp. on Principles of Progr. Lang., 21–28 (1977)
4. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques and Tools. Reading, MA: Addison-Wesley (1986)
5. Balachandran, A., Dhamdhere D.M., Biswas, S.: Efficient retargetable code generation using bottom-up tree pattern matching. Comput. Lang. **15**(3), 127–140 (1990)
6. Cai, J., Paige, R., Tarjan, R.: More efficient bottom-up multi-pattern matching in trees. Theoret. Comput. Sci. **106**, 21–60 (1992)
7. Cattell, R.G.G.: Code generation in a machine-independent compiler. Proc. of the ACM SIGPLAN 1979 Symp. on Compiler Construction, ACM SIGPLAN Notices **14**(8), 65–75 (1979)
8. Cattell, R.G.G.: Automatic derivation of code generators from machine descriptions. ACM Trans. on Prog. Lang. and Sys. **2**(2), 173–190 (1980)
9. Cattell, R.G.G.: Formalization and Automatic Derivation of Code Generators. UMI Research Press, Ann Arbor, Michigan (1982)
10. Chase, D.R.: An improvement to bottom-up tree pattern matching. In Proc. of the Fourteenth Ann. ACM Symp. on Principles of Progr. Lang., 168–177 (1987)
11. Christopher, T.W., Hatcher, P.J., Kukuk, R.C.: Using dynamic programming to generate optimised code in a Graham-Glanville style code generator. Proc. of the ACM SIGPLAN 1984 Symp. on Compiler Construction, ACM SIGPLAN Notices **19**(6), 25–36 (1984)
12. Dershowitz, N., Jouannaud, J.-P.: Rewrite systems. Handbook of Theoretical Computer Science (Vol. B: Formal Models and Semantics), Leeuwen, J. van (ed), 245–320. Amsterdam: Elsevier (1990)
13. Emmelmann, H.: Code selection by regularly controlled term rewriting. Code generation – concepts, tools, techniques, Giegerich, R., Graham, S.L. (eds), (Workshops in Computing Series, 3–29) Berlin, Heidelberg, New York: Springer 1991
14. Emmelmann, H., Schröer, F.W., Landwehr, R.: BEG – a generator for efficient back ends. ACM SIGPLAN Notices **24**(7), 246–257 (1989)
15. Ferdinand, C., Seidl, H., Wilhelm, R.: Tree automata for code selection. Acta Informatica **31**(8), 741–760 (1994)
16. Fraser, C.W., Hanson, D.R., Proebsting, T.A.: Engineering a simple, efficient code-generator generator. ACM Letters on Progr. Lang. and Sys. **1**(3), 213–226 (1992)

17. Fraser, C.W., Henry, R.R., Proebsting, T.A.: BURG – fast optimal instruction selection and tree parsing. ACM SIGPLAN Notices **27**(4), 68–76 (1992)
18. Giegerich, R.: Code selection by inversion of order-sorted derivors. Theoret. Comput. Sci. **73**, 177–211 (1990)
19. Giegerich, R., Schmal, K.: Code selection techniques: pattern matching, tree parsing, and inversion of derivors. Proc. 2nd European Symp. on Programming, Ganzinger, H. (ed) (Lect. Notes in Comput. Sci. vol. 300, 247–268). Berlin, Heidelberg, New York: Springer 1988
20. Glanville, R.S.: A machine independent algorithm for code generation and its use in retargetable compilers. Ph.D. thesis, University of California, Berkeley (1977)
21. Gough, K.J.: Bottom-up tree rewriting tool MBURG. ACM Sigplan Notices **31**(1), 28–31 (1996)
22. Glanville, R.S., Graham, S.L.: A new method for compiler code generation. Proc. of the Fifth Ann. ACM Symp. on Principles of Progr. Lang., 231–240 (1978)
23. Hatcher, P.J., Christopher, T.W.: High-quality code generation via bottom-up tree pattern matching. Proc. of the Thirteenth Ann. ACM Symp. on Principles of Progr. Lang., 119–130 (1986)
24. Hemerik, C., Katoen, J.P.: Bottom-up tree acceptors. Sci. of Comput. Progr. **13**, 51–72 (1990)
25. Henry, R.R.: The CODEGEN user's manual. Technical report 87-08-04, Computer Science Department, University of Washington (1988)
26. Henry, R.R.: Encoding optimal pattern selection in a table-driven bottom-up tree-pattern matcher. Technical Report 89-02-04, Computer Science Department, University of Washington (1989)
27. Henry, R.R, Damron, P.C.: Algorithms for table-driven generators using tree-pattern matching. Technical Report 89-02-03, Computer Science Department, University of Washington (1989)
28. Henry, R.R, Damron, P.C.: Performance of table-driven code generators using tree-pattern matching. Technical Report 89-02-02, Computer Science Department, University of Washington (1989)
29. Hoffmann, C.M., O'Donnell, M.J.: Pattern matching in trees. J. ACM **29**(1), 68–95 (1982)
30. Kanal, L., Kumar, V., (eds): Search in Artificial Intelligence. Berlin, Heidelberg, New York: Springer (1988)
31. Kron, H.: Tree templates and subtree transformational grammars. Ph.D. thesis, Information Sciences Department, University of California at Santa Cruz (1975)
32. Kurtz, S: Narrowing and Basic Forward Closures. Technical Report 5, Technische Fakultät, Universität Bielefeld (1992)
33. Nijmeijer, A: Review of the Graham-Glanville code-generation scheme. Technical Report 88-61, Department of Computer Science, University of Twente (1988)
34. Nilsson, N.: Principles of Artificial Intelligence. Palo Alto: Morgan Kaufmann (1980)
35. Pelegrí-Llopart, E.: Rewrite systems, pattern matching, and code generation. Ph.D. thesis, University of California, Berkeley (1987) (also as Technical Report CSD-88-423)
36. Pelegrí-Llopart, E., Graham, S.L.: Optimal code generation for expression trees: An application of BURS theory. Proc. of the Fifteenth Ann. ACM Symp. on Principles of Progr. Lang., 294–308 (1988)
37. Proebsting, T.A.: BURS automata generation. ACM Trans. on Prog. Lang. and Sys. **3**(17), 461–486 (1995)
38. Proebsting, T.A., Whaley, B.R.: One-pass, optimal tree parsing – with or without trees. Compiler construction, Gyimóthy, T. (ed) (Lect. Notes in Comput. Sci. vol. 1060, 294–308) Berlin, Heidelberg, New York: Springer 1996
39. Weisgerber, B., Wilhelm, R.: Two tree pattern matchers for code selection. Compiler compilers and high speed compilation, Hammer, D. (ed) (Lect. Notes in Comput. Sci. vol. 371, 215–229) Berlin, Heidelberg, New York: Springer 1989
40. Wulf, W.A., Leverett, B.W., Cattell, R.G.G., Hobbs, S.O., Newcomer, J.M., Reiner, A.H., Schatz, B.R.: An overview of the production-quality compiler compiler project. IEEE Computer **13**(8), 38–49 (1980)

Note added in proof. In the function *Generate* shown in Fig. 15, 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 $L_D(t\,|_{n \cdot p}) = L_{D_n}(t\,|_p)$ elsewhere. Furthermore, $D \otimes \langle r, p \rangle$ 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.

The function *Checknf* eliminates triples (from Z) that contain decorations that are not in normal form (these decorations have a higher precedence). The strong-normal-form property is imposed by considering positions $p \in RP_t(S_D)$ only.