Code Generation = A* + BURS

Albert Nymeyer*, Joost-Pieter Katoen, Ymte Westra, Henk Alblas

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

Abstract. A system called BURS that is based on term rewrite systems and a search algorithm A* are combined to produce a code generator that generates optimal code. The theory underlying BURS is re-developed, formalised and explained in this work. The search algorithm uses a cost heuristic that is derived from the term rewrite system to direct the search. 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.

Key words: compiler generators, code generation, term rewrite systems, search algorithms, formal techniques

Compiler building is a time-consuming and error-prone activity. Building the front-end (i.e. scanner, parser and intermediate-code generator) is 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, remains 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 Pelegrí-Llopart and Graham [30], 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 that is poorly understood. The theory has received virtually no attention in the literature since its initial publication [29]. The only research that has been carried out in this technique has been on improved table-compression methods. Many researchers who claim to use BURS theory (e.g. [31, 14]) generally use 'weaker' tree grammars instead of term rewrite systems, or equate BURS with a system that does a static cost analysis (e.g. [13]). We argue that a static cost analysis is neither necessary nor sufficient to warrant a BURS label, and that a system that is based on tree grammars cannot be a BURS.

In this work we present an outline of formal BURS theory. Due to space restrictions, we present the full theory in [28]. This formalisation of BURS contrasts with the semi-formal work of Pelegrí-Llopart and Graham. But there are other important differences in our work. We do not, for example, use instruction costs to do static pattern selection, and we do not use dynamic programming. Instead we use a *heuristic* search algorithm that only needs to dynamically compute costs for those patterns that may contribute to

* Contact author: e-mail address is nymeyer@cs.utwente.nl

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 brief survey of the literature. In Section 2 we derive the *input* and *output* sets of an expression tree. These sets contain the patterns that match the given expression tree. The patterns are selected by the heuristic search algorithm A*. This algorithm, described in Section 3, is all-purpose—it can be used to solve all kinds of 'shortest-path' problems. In our case the search graph consists of all possible reductions of an expression tree, and we wish to find the least expensive. The A* algorithm uses a *successor function* (*algorithm*) to select patterns and apply rewrite rules. In this sense, the successor function marries A* to BURS. The successor function is presented in Section 4. 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 Section 4. Finally, in Section 5, we present our conclusions.

1 Other work

Kron [25], Hoffmann and O'Donnell [23], and Chase [7] have laid the foundations of the BUPM technique. Chase [7] implemented a BUPM by specifying patterns using a *regular tree grammar* (RTG). An 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. A formalisation of Chase's table-compression technique can be found in Hemerik and Katoen [18]. An asymptotic improvement in both space and time to Chase's algorithm is given by Cai *et al* [3].

Hatcher and Christopher [17] 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. In both the static and dynamic BUPMs, the cost analysis is usually carried out using *dynamic programming* [1, 8, 32]. For a comparison of the performance of static and dynamic BUPMs, see Henry and Damron [22, 21] and Henry [19, 20]. Two notable attempts to improve the efficiency of the dynamic (BUPM) code generator have been Emmelmann *et al* [11], who developed the BEG system, and more recently Fraser *et al* [13] with the IBURG system.

In 1990, Balachandran *et al* [2] used a RTG and techniques based on the work of Chase, Hatcher and Christopher to build a static BUPM. Very recently, Ferdinand *et al* [12] reformulated the (static) bottom-up pattern-matching algorithms (based on RTGs) in terms of finite tree automata. A subset-construction algorithm is developed that does a static cost analysis, and generalises the table-compression technique of Chase.

Pelegrí-Llopart and Graham [29, 30] 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 system called 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 'theory' that Pelegrí-Llopart and Graham develop is quite complex, however. They also compare the performance of a BURS with other techniques. They find that the tables are smaller and the code generator much faster.

Mainly theoretical research into the role of term rewrite systems in code generation has been carried out by Emmelmann [10] and Giegerich [16, 15].

In 1992, Fraser, Henry and Proebsting [14] presented a new implementation of '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 [31].

The only serious application of heuristic search techniques to code generation has been the PQCC (Production-Quality Compiler-Compiler) Project [33]. The construction of the code generator and the code-generator generator in PQCC are reported by Cattell in [4, 5, 6]. 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. The heuristic *semantic closeness* means that either the root operators of the pattern and a particular template match, or that there is a rewrite rule that rewrites the root operator of the template into the root operator of the pattern. For performance reasons, the search procedure is done mostly statically, using a set of heuristically generated pattern trees.

2 An Outline of BURS Theory

In this section we describe how the sets of patterns that match a given expression tree are computed. We will only outline the formal approach that has been used—for a full treatment, the reader is referred to [28]. For more information on term rewrite systems see [9].

A *ranked alphabet* Σ is a pair (S, r) with S a finite set of symbols and $r \in S \rightarrow \mathbb{N}$, where $\mathbb{N}$ denotes the set of natural numbers. If a is a symbol in S, then $r(a)$ is its rank. Symbols with rank 0 are called *constants*. The set of symbols with rank n, denoted Σ_n, is $\{ a \in S \mid r(a) = n \}$.

For Σ a ranked alphabet and V a set of variable symbols, the set of terms $T_\Sigma(V)$ consists of constants, variables and $a(t_1, \ldots, t_n)$, where $a \in \Sigma_n$, and $t_1, \ldots, t_n \in T_\Sigma(V)$, $n \geq 1$. For term t, $Var(t)$ denotes the set of variables in t. The terms t for which $Var(t) = \emptyset$ are called *ground terms*.

The position of a sub-term of a term t can be indicated by a path from the root of t to the root of the sub-term. A position is represented as a string of positive naturals, separated by dots. For example, the position of the first child of the root is 1, and the second child 2. The position of the first grandchild of the root is 1·1. The root is at position ε. We define $Pos(t)$ as the set of positions of all nodes in t. The sub-term of a term t at position $p \in Pos(t)$ is denoted $t|_p$. We are now able to define a term rewrite system with costs.

Definition 2.1 *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 $T_\Sigma(V) \times T_\Sigma(V)$
- $C \in R \rightarrow \mathbb{R}^+ \cup \{0\}$, a cost function

such that, for all $(t, t') \in R$, $t' \neq t$, $t \notin V$ and $\mathrm{Var}(t') \subseteq \mathrm{Var}(t)$. □

Note that $\mathbb{R}$ denotes the set of real numbers. Elements of R, identified as r_1, r_2, and so on, are called *rewrite rules*. An element $(t, t') \in R$ is usually written as $t \longrightarrow t'$. 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 CTRS defined in the following example is a modified version of an example taken from Pelegrí-Llopart and Graham [30], and will be used as a running example throughout this section.

Example 2.2 Let $((\Sigma, V), R, C)$ be a CTRS, where $\Sigma = (S, r)$, $S = \{+, c, a, r, 0\}$, $r(+) = 2, r(c) = r(r) = r(a) = r(0) = 0$, $V = \{x, y\}$, and R defined as follows:

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

The cost function C is defined by $C(r_1) = C(r_2) = C(r_5) = 0$, $C(r_3) = C(r_6) = 3$, $C(r_4) = 5$ and $C(r_7) = C(r_8) = 1$. 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 $t|_{2\cdot 2} = +(c, y)$. □

Variables in a term t can be substituted by some term. The substitution $\sigma \in V \rightarrow T_\Sigma(V)$ in a term t is written t^σ. 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 $t_1'^\sigma = t_2'$. Thus, rewrite rules that are identical, except for variable symbols, are considered equivalent.

A rewrite rule and substitution are used to define a *rewrite step*. In a rewrite step $t_1 \xRightarrow{\langle r,p\rangle} t_2$, where $t_1, t_2 \in T_\Sigma(V)$, $r : t \longrightarrow t' \in R$ and $p \in Pos(t_1)$, the result term t_2 is obtained from t_1 by replacing $t_1|_p$ by t'^σ in t_1 and using substitution σ with $t^\sigma = t_1|_p$. We can also write $\langle r, p\rangle\, t_1 = t_2$. 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 that are applied one after another is called a *rewrite sequence*. A rewrite step is a rewrite sequence of length 1. We write $t \xRightarrow{\langle r_1,p_1\rangle \ldots \langle r_n,p_n\rangle} t'$ if and only if $\exists\, t_1,\ldots,t_{n-1} : t \xRightarrow{\langle r_1,p_1\rangle} t_1 \xRightarrow{\langle r_2,p_2\rangle} \ldots t_{n-1} \xRightarrow{\langle r_n,p_n\rangle} t'$. We can also let $S(t) = \langle r_1,p_1\rangle \ldots \langle r_n,p_n\rangle$, and write $S(t)\,t = t'$. We sometimes denote a rewrite sequence $S(t)$ by τ.

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. If a rewrite rule r occurs in a rewrite sequence τ, then we write $r \in \tau$. We assume that all rewrite sequences are acyclic.

Two rewrite sequences may also be *permutations* of each other. Permuted rewrite sequences will, of course, have the same cost, but note that corresponding rules in the two sequences may not be applied at the same positions.

Example 2.3 Consider the CTRS shown in Example 2.2, and let $t = +(0,+(r,c))$. We can write $t \xRightarrow{\langle r_1,2\rangle} t'$, 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 $\sigma(y) = \mathrm{c}$ such that $(x,y)^\sigma = t|_2$. Two derivations starting with t' are:

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

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

Given a CTRS $((\Sigma,V),R,C)$ and 2 ground terms $t,t' \in T_\Sigma$, we now wish to determine a rewrite sequence τ such that $t \xRightarrow{\tau} t'$ with minimal cost. In practice, term rewrite systems in code generation will allow many different rewrite sequences to transform t into t'. Fortunately, optimisations are possible so that we only need to consider relatively few of these rewrite sequences.

The first 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 all terms t. Hence we only need to consider rewrite sequences in normal form. It is a stipulation for our approach, and a property of a BURS, that permuted sequences also have the same cost. If a cost function 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 that we obtain by only considering the normal form will lead to legal rewrite sequences being discarded.

We can label, or decorate, each node in a term with a rewrite sequence. 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). A term t in which each sub-term is labelled by a (possibly empty) local rewrite sequence is called a decorated term, or *decoration*, and is denoted by $D(t)$. 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 $S_D(t)$ can be obtained by a post-order traversal of t. 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 equivalent if and only if their corresponding rewrite sequences are permutations of each other.

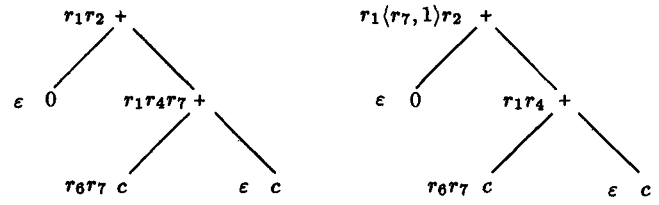

Fig. 1. Equivalent decorations $D(t)$ and $D'(t)$ of a term t

Example 2.4 Consider the CTRS shown in Example 2.2 and let $t = +(0, +(c, c))$. Two decorations $D(t)$ and $D'(t)$ of t are depicted in Figure 1, on the left and right, respectively. The corresponding rewrite sequences are:

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

The decorations $D(t)$ and $D'(t)$ are equivalent because $S_D(t)$ is a permutation of $S_{D'}(t)$. □

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 $D(t)$.

The transitive closure of $\prec$ is denoted $\prec^+$. The minimal decorations under $\prec^+$ are said to be in normal form. Normal forms need not be unique as $\prec^+$ does not need to have a least element. We let $NF(t)$ denote the set of decorations of t that are in normal form. In [28] we prove that, given a term t and a rewrite sequence τ such that $t \xrightarrow{\tau} t'$, a normal-form decoration of t always exists.

Example 2.5 In Example 2.4 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 any lower, hence $D(t)$ is in normal form. □

In a second optimisation, we reduce the number of decorations that we need to consider still further by restricting the class of normal-form decorations to *strong normal* *form*. Local rewrite sequences in this restricted class contain rewrite rules that are only applied to positions that have not previously been substituted for a variable. We say that each position in a term is either *rewriteable* or *non-rewriteable*. If a term is rewritten using a rewrite rule that does not contain a variable, then the writeability of the positions in the rewritten term do not change. If the rewrite rule does contain a variable, then the positions in the term substituted for the variable become non-rewriteable.

Let $RP_t(\tau)$ be the set of rewriteable positions in the term resulting from applying τ to t. Initially, all positions in t are rewriteable, so $RP_t(\varepsilon) = Pos(t|_\varepsilon)$. For rewrite sequence $\tau\langle t_1 \longrightarrow t_2, p\rangle$ we define:

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

where $t \xRightarrow{\tau} t' \xRightarrow{\langle t_1 \longrightarrow t_2, p\rangle} t''$, and $VP(t)$ is the set of positions in t at which a variable occurs. In the definition above, we see that the set of rewriteable positions in t'' consists of the rewriteable positions in t' (i.e. $RP_t(\tau)$), 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 ($\{Pos(t''|_{p\cdot q}) \mid q \in VP(t_2)\}$). Given a normal-form decoration, we prove in [28] that a strong-normal form always exists.

Example 2.6 We are given a TRS with $S = \{*, +, c, r, 2\}$, corresponding ranks $\{2, 2, 0, 0, 0\}$, $V = \{x\}$ and R defined as follows:

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

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 $\langle r_2, 2\rangle$ (note that this rule does not contain a variable), then we generate the term $t'' = *(2, r)$ with rewriteable positions:

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

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

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

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. □

A normal-form decoration $D(t)$ is in *strong normal form* if all rewrite rules r in local rewrite sequences $L_D(t|_p)$ are applied at rewriteable positions p, for all $p \in Pos(t)$. We let $SNF(t)$ denote the set of decorations of t that are in strong normal form.

Example 2.7 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:

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

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 (r_3 may therefore not be applied to this position). The decoration $D'(t)$ defined by $L_{D'}(t) = r_1\, \langle r_3, 2\rangle\, r_2$ and $L_{D'}(t|_1) = L_{D'}(t|_2) = \varepsilon$ is, however, in strong normal form. Note that the rewrite step $\langle r_3, 2\rangle$ is applied at the root in both decorations. □

We now use the strong normal-form decorations of a term to compute the *input* and *output sets*. These sets define the patterns that match the expression tree. Given the strong-normal-form decoration $D(t)$ such that $t \xRightarrow{S_D(t)} g$ for some given goal term g, then we define the possible inputs for each sub-term t' of t, denoted $I_D(t')$, and outputs, denoted $O_D(t')$, as follows:

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

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

$$O_D(t) = t' \text{ where } I_D(t) \xRightarrow{L_D(t)} t'$$

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 $OS_g(t)$.

$$IS_g(t) = \{\, I_D(t) \mid D(t) \in SNF(t) \ \wedge\ t \xRightarrow{S_D(t)} g \,\}$$

$$OS_g(t) = \{\, O_D(t) \mid D(t) \in SNF(t) \ \wedge\ t \xRightarrow{S_D(t)} g \,\}$$

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

Example 2.8 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 Figure 1. This decoration is also in strong normal form. The inputs $I_D(t)$ and outputs $O_D(t)$ of this decoration for goal term r are depicted on the left in Figure 2, where the 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 on the right in Figure 2. □

An algorithm to calculate input and output sets for terms t and g, and the corresponding local rewrite sequences, consists of 2 passes. In the first, bottom-up pass, sets of *triples* are computed for all possible goal terms. A triple, written $\langle t, \tau, t'\rangle$, consists of an input t, rewrite sequence τ, and output t' such that $t \xRightarrow{\tau} t'$. In the second, top-down pass, these sets of triples are 'trimmed' using the given goal term g. These trimmed sets of triples, denoted by $V(t)$, consist of the input and output sets, and the associated local rewrite sequences. For space reasons, the algorithm to compute $V(t)$ is not shown, but can be found in [28].

Example 2.9 Below we show the set of triples $V(t)$ for our running example with expression tree $t = +(0, +(c, c))$.

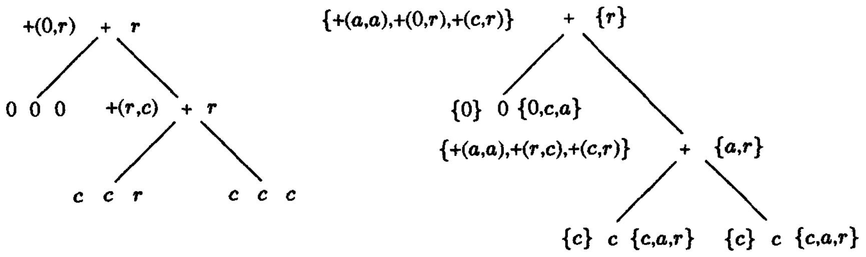

Fig. 2. The inputs, outputs, input sets and output sets of the term $+(0, +(c, c))$

$$
\begin{aligned}
t|_\varepsilon &= \{ \langle +(a,a), r_3, r\rangle, \langle +(0,r), r_1r_2, r\rangle, \langle +(c,r), r_4r_7, r\rangle \} \\
t|_1 &= \{ \langle 0, \varepsilon, 0\rangle, \langle 0, r_5, c\rangle, \langle 0, r_5r_6, a\rangle \} \\
t|_2 &= \{ \langle +(a,a), r_3, r\rangle, \langle +(a,a), r_3r_8, a\rangle, \langle +(r,c), r_1r_4, a\rangle, \langle +(r,c), r_1r_4r_7, r\rangle, \\
&\quad \langle +(c,r), r_4, a\rangle, \langle +(c,r), r_4r_7, r\rangle \} \\
t|_{2\cdot 1} &= \{ \langle c, \varepsilon, c\rangle, \langle c, r_6, a\rangle, \langle c, r_6r_7, r\rangle \} \\
t|_{2\cdot 2} &= \{ \langle c, \varepsilon, c\rangle, \langle c, r_6, a\rangle, \langle c, r_6r_7, r\rangle \}
\end{aligned}
$$

Note that all rewrite rules are applied at the root. □

To guarantee termination of this algorithm the length of each local rewrite sequence must be bounded. That is, for all $t \in T_\Sigma(V)$ and $D(t) \in SNF(t)$ there exists some natural number k such that $\forall p \in Pos(t) : \mid L_D(t|_p) \mid \leq k$. This is referred to as the *BURS property*. The BURS property is necessary because we can have terms that contain variables on the right-hand side of rewrite rules in our rewrite system. Rewrite sequences can therefore continue indefinitely, and terms can 'explode' if the property does not hold. Our running example, by the way, is BURS with $k = 3$.

The work of Pelegrí-Llopart and Graham

Pelegrí-Llopart and Graham[30] (PLG) first define a normal-form rewrite sequence, and then a local rewrite sequence and assignment. We have reversed this order, and we have been more formal. For example, we characterise normal-form decorations by using the ordering relation $\prec$. Our rewriteable positions are related to PLG's *touched positions*, which PLG define only informally and unclearly. PLG do not explicitly define a strong normal form. While we directly encode the inputs, outputs and local rewrite sequences into the expression tree, PLG use *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.

3 Heuristic-Search Methods

Search techniques are used extensively in artificial intelligence [24, 27] 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. A *search graph* G is a quadruple (N, E, n_0, N_g) with a set of nodes N, a set of directed edges $E \subseteq N \times N$, each labelled with a cost $C(n, m) \in \mathbb{R}$, $(n, m) \in E$, an initial node $n_0 \in N$, and a set of goal nodes $N_g \subseteq N$. Furthermore, G is connected, $N_g \neq \emptyset$ and $\forall (n, m) \in E : n \notin N_g$.

One of the best known search techniques is the A* algorithm ([26, 27]). 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 A* algorithm computes the least-cost *path* from the initial node to a goal node. The algorithm begins by initialising sets of *open* nodes $N_o \subseteq N$ to $\{n_0\}$, *closed* nodes $N_c \subseteq N$ to $\emptyset$, and the path and cost of the initial node n_0. As long as we have not found a goal node, we carry out the following procedure. We use a cost function to compute N_s, which is the set of nodes in N_o with lowest cost. If this set contains a goal node, then we are 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. The *successor nodes* of a given node are those nodes that can be reached with a path of length 1 from the node. 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. The algorithm terminates when we find a successor node that is a goal node.

The cost of a node n, denoted $f^*(n)$, is the sum of the minimum cost of a path from n_0 to n, denoted $g(n)$, and the *estimated* cost from n to a goal node, denoted $h^*(n)$. The estimated cost is obtained by using heuristic domain knowledge. This heuristic knowledge allows us to avoid searching some unnecessary parts of the search graph. The search technique therefore needs to try fewer paths in an attempt to find a goal node. Note that the *actual* cost of reaching a goal node from n is denoted $h(n)$. 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.

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.

In our description of the A* algorithm we have used successor nodes and paths. Given a search graph $G = (N, E, n_0, N_g)$, the set of successor nodes *Successor*$(n) \in \mathcal{P}(N)$ of a node $n \in N$, where $\mathcal{P}(N)$ is the power set of N, can be defined as *Successor*$(n) = \{m \in N \mid (n, m) \in E\}$. Note that if $n \in N_g$ then *Successor*$(n) = \emptyset$. Furthermore, the path *Path*$(n) \in N^*$ to a node $n \in N$, where N^* denotes sequences of elements from N, is a string of nodes $n_0 n_1 \ldots n_k$ such that $\forall 1 \leq i \leq k : n_i \in \mathit{Successor}(n_{i-1}) \ \wedge \ n_k = n, k \geq 0$. Note that there may be more than 1 path that leads to a node. If *Path*$(n) = n_0 n_1 \ldots n_k$ and $m \in \mathit{Successor}(n)$ then we can append the node m to the path *Path*(n) using the append operator $\oplus$. We write $\mathit{Path}(m) = \mathit{Path}(n) \oplus m = n_0 n_1 \ldots n_k m$.

Conceptually the A* algorithm can be quite straightforwardly applied to code generation. 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 is the 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.

4 Coupling A* to BURS

In practice, the major problem in coupling A* and BURS is determining the successor nodes of a given node (in the search graph). In other words, given some term (expression tree), at what positions may we apply rewrite rules? We note that all rewrite rules that we apply must be *correct*, of course. A rewrite rule is correct if there is a path in the search graph from the resulting term (node) to a goal term (node). In this section we describe how a search graph for a BURS is initialised, and how successor nodes are computed.

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, r, t') where t is the current term, p is the current position in that term, r 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 4.1 Consider our running example (Example 2.2). 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 Figure 3. In this function we use the functions *Next*, *Parent* and *Child* to position ourselves in the search graph. Given a position $p \in \mathit{Pos}(t) \setminus \{\epsilon\}$ in a term t, $\mathit{Next}(p, t) \in \mathbb{N}_+^*$ is the next position in a post-order traversal of t. Note $\mathbb{N}_+$ is $\mathbb{N} \setminus \{0\}$. The function $\mathit{Parent}(p, t) \in \mathbb{N}_+^*$ is the position of the parent of p in t, and $\mathit{Child}(p, t) \in \mathbb{N}_+$ 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(ϵ,t)* = *Child(ϵ,t)* = ϵ, but *Next(ϵ,t)* is undefined, for any t. Note that p = *Parent(p,t)*$\cdot$*Child(p,t)*.

Example 4.2 In the term $t = +(0, +(c, c))$, we have *Next*(1,t) = 2·1, *Next*(2·1,t) = 2·2, *Next*(2·2,t) = 2 and *Next*(2,t) = ϵ. Furthermore, *Parent*(2·1,t) = 2 and *Child*(2·1,t) = 1. □

```
|[ con $((\Sigma, V), R)$ : TRS;
     $t, g : T_\Sigma$;
     $V(t) : \mathcal{P}(T_\Sigma \times (R \times \mathbb{N}_+^*)^* \times T_\Sigma)$;

  func Successor $(t : T_\Sigma,\ p : \mathbb{N}_+^*,\ \tau : (R \times \mathbb{N}_+^*)^*,\ it : T_\Sigma)$
                 $: \mathcal{P}(T_\Sigma \times \mathbb{N}_+^* \times (R \times \mathbb{N}_+^*)^* \times T_\Sigma)$
  |[ var $S : \mathcal{P}(T_\Sigma \times \mathbb{N}_+^* \times (R \times \mathbb{N}_+^*)^* \times T_\Sigma)$;

    func Match$(p' : \mathbb{N}_+^*,\ t' : T_\Sigma)$ : boolean
    |[ var $Z : \mathcal{P}(T_\Sigma)$;
          $it' : T_\Sigma$;
          $b$ : boolean ;
      $b := (p' = \epsilon)$;
      $Z(t) := \{ it \mid \langle it, \tau, ot\rangle \in V(t|_{Parent(p')}) \ \wedge\ it|_{Child(p')} = t' \}$;
      do $Z \neq \emptyset \ \wedge\ \neg b$ $\longrightarrow$|[ choose $it' \in Z$;
                                $Z := Z \setminus \{it'\}$;
                                $b := (\forall\, 1 \le i < Child(p') : it'|_i = t|_{Parent(p')\cdot i})$
                              ]|
      od;
      return $b$
    ]|;

    if $(p = \epsilon)\ \vee\ \neg Match(p,\ t|_p)$ $\longrightarrow$ $S := \emptyset$
    ▯ $(p \neq \epsilon)\ \wedge\ Match(p,\ t|_p)$ $\longrightarrow$ $S := Successor(t,\ Next(p),\ \epsilon,\ t|_{Next(p)})$
    fi ;
    for all $r \in R$
    do for all $p' \in Pos(it)$
       do for all $\langle it,\ \tau\langle r, p'\rangle\tau',\ ot\rangle \in V(t|_p)$   (* this is a loop over $\tau'$ and $ot$ *)
          do if $\neg Match(p,\ ot)$ $\longrightarrow$ skip
             ▯ $Match(p,\ ot)$ $\longrightarrow$ $S := S \cup \{ (\langle r, p'\rangle t,\ p,\ \tau\langle r, p'\rangle,\ it) \}$
             fi
          od
       od
    od;
    return $S$
  ]|
]|.
```

Fig. 3. The successor function that computes a set of new search nodes.

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. 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 $t \in N_g$.

Example 4.3 Consider our running example again. Let us compute the successor nodes of the initial node, i.e. we compute $Successor(+(0, +(c, c)), 1, \epsilon, 0)$. Because $p \neq \epsilon$ and $Match(1,\ t|_1) = true$, we recursively call the function again with the next position, $p = 2{\cdot}1$. That is, we call $Successor(+(0, +(c, c)), 2{\cdot}1, \epsilon, c)$. Again, because $p \neq \epsilon$ and $Match(2{\cdot}1,\ t|_{2\cdot 1}) = true$, we recursively call $Successor(+(0, +(c, c)), 2{\cdot}2, \epsilon, c)$. The recursion now stops because $Match(2{\cdot}2,\ t|_{2\cdot 2}) = false$. We therefore let $S := \emptyset$, and inspect all the triples of $V(t|_{2\cdot 2})$ (see Example 2.9). The triple $\langle c, r_6 r_7, r \rangle$ satisfies the loop condition, and since $Match(2{\cdot}2, r) = true$, 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, $V(t|_{2\cdot 1})$. The triple $\langle c, r_6 r_7, r \rangle$ (again) satisfies the loop condition, $Match(2{\cdot}1, r) = true$, and we generate the search node $(+(0, +(a, c)), 2{\cdot}1, r_6, c)$. The call of *Successor* for $p = 2{\cdot}1$ is also now complete. Inspecting the triples associated with the initial position, $V(t|_1)$, we find that triple $\langle 0, r_5 r_6, a \rangle$ satisfies the loop condition, and that $Match(1, a) = true$. We therefore generate the search node $(+(c, +(c, c)), 1, r_5, 0)$. The result of the above computations is that we have generated 3 new search nodes from the initial node, namely:

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

We can continue computing successors until all the goal nodes have been found. In Figure 4 we see the search graph for the expression tree $+(0, +(c, c))$. The nodes in the graph are the expression trees, and the edges are labelled with the rewrite steps and the positions at which they are applied. Note that there are a total of 11 paths leading from the initial node (the root node) to a goal node in Figure 4. □

In the example above, we have shown how the successor function shown in Figure 3 can be used to compute the complete search graph. However, 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. The A* algorithm will compute successors of *only* those nodes that potentially lie on a least-cost path from the initial node to a goal node. The cost $g(n)$ of a node n is simply the sum of the costs of the rewrite rules applied along the path to n. But what is the value of the heuristic cost $h^*(n)$? 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.

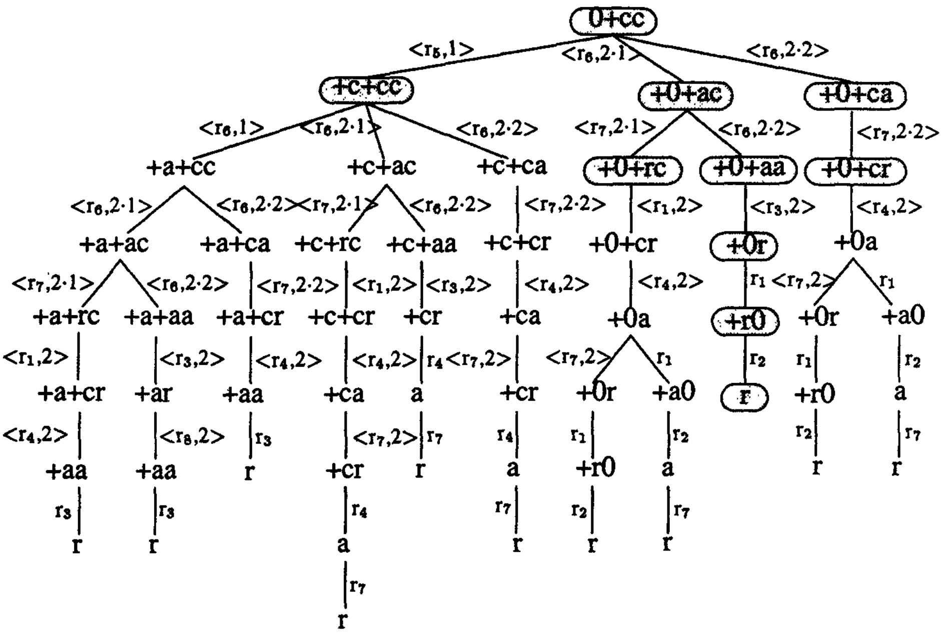

Fig. 4. A complete search graph, and heuristic search graph (in shaded boxes)

Example 4.4 The heuristic function that we use to 'predict' the cost for our running example is:

$$h^*(n) = 3 * (|+(x,y)|_t + |c|_t), \quad x \neq 0, \quad y \neq 0$$

where $n = (t, p, r, t')$, and $|s|_t$ denotes the number of sub-terms in t that match s. This heuristic function, which we obtain by inspection, predicts a cost that under-estimates, or is equal to, the actual cost. For example, $h^* = 0$ for $t = a$ (the actual cost is 1), $h^* = 3$ for $t = +(0, c)$ (actual cost 4) and $h^* = 6$ for $t = +(c, a)$ (actual cost 6). □

Example 4.5 We now apply the A* search algorithm with the cost function $f^*(n) = g(n) + h^*(n)$ to our running example. We begin by generating the (3) successors of the initial node, as shown in Example 4.3. The costs of these nodes are $0 + 15$, $3 + 6$ and $3 + 6$ (resp.). The second and third nodes are the cheapest; we choose the second, and compute its successors using the successor function. This results in $+(0, +(r, c))$ and $+(0, +(a, a))$, using rewrite steps $\langle r_7, 2\cdot 1\rangle$ and $\langle r_6, 2\cdot 2\rangle$. These nodes have costs $4 + 6$ and $6 + 3$. Continuing on in this way we find a goal node in just 6 steps, having visited (computed) just 10 nodes in total. The resulting *heuristic* search graph is shown using shaded boxes in Figure 4. The cost of the path to the goal node is 9. □

Implementation

The A* algorithm was straightforward to implement. The pattern-matching and successor-function algorithms proved more difficult, requiring many intricate tree-manipulation routines to be written. In total, the system comprises approximately 3000 lines of C code. The implementation has revealed the 'strength' of the theory. For example, we saw in Example 2.9 that there are only 3 local rewrite sequences at the root of the term $+(0, +(c, c))$. Before trimming, there are in fact 101 sequences. If we remove the restriction that sequences must be in strong normal form (in other words, we allow rewrite rules to be applied at all positions, not just rewriteable ones), then the number of (untrimmed) sequences is too large ($\gg 10^6$) to be computed. The strong-normal-form restriction is therefore extremely powerful. TRSs for real machines have not yet been developed.

5 Conclusions

In this work we have reformulated BURS theory, and we have shown how this theory can be used to solve the pattern-matching problem in code generation. This is our first major result. The task of selecting optimal patterns is carried out by the A* algorithm. The interface between the BURS algorithm that generates patterns, and the A* algorithm that selects them, is provided by the successor algorithm. This important algorithm builds the search space. Combining BURS theory A* is our second major result.

Term rewrite systems are a more powerful formalism than the popular regular tree grammars on which most code-generator-generator systems are based. The term rewrite system that underlies the BURS system is used to deduce a heuristic cost function. This cost function speeds up the search process. Optimality is guaranteed if the cost heuristic never over-estimates the actual cost of generating code.

Future work will be mainly concerned with the development of term rewrite systems that describe real machines, and a systematic technique to construct the heuristic cost function.

Acknowledgements: Gert Veldhuyzen van Zanten contributed many formative ideas.

References

1. A. V. Aho, M. Ganapathi, and S. W. K. Tjiang. Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems*, 11(4):491–516, October 1989.
2. A. Balachandran, D. M. Dhamdhere, and S. Biswas. Efficient retargetable code generation using bottom-up tree pattern matching. *Computer Languages*, 15(3):127–140, 1990.
3. J. Cai, R. Paige, and R. Tarjan. More efficient bottom-up multi-pattern matching in trees. *Theoretical Computer Science*, 106:21–60, 1992.
4. R. G. G. Cattell. Code generation in a machine-independent compiler. *Proceedings of the ACM SIGPLAN 1979 Symposium on Compiler Construction, ACM SIGPLAN Notices*, 14(8):65–75, August 1979.
5. R. G. G. Cattell. Automatic derivation of code generators from machine descriptions. *ACM Transactions on Programming Languages and Systems*, 2(2):173–190, April 1980.
6. R. G. G. Cattell. *Formalization and Automatic Derivation of Code Generators.* UMI Research Press, Ann Arbor, Michigan, 1982.

7. D. R. Chase. An improvement to bottom-up tree pattern matching. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 168–177, Munich, Germany, January 1987.
8. T. W. Christopher, P. J. Hatcher, and R. C. Kukuk. Using dynamic programming to generate optimised code in a Graham-Glanville style code generator. *Proceedings of the ACM SIGPLAN 1984 Symposium on Compiler Construction, ACM SIGPLAN Notices*, 19(6):25–36, June 1984.
9. N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science (Vol. B: Formal Models and Semantics)*, chapter 6, pages 245–320. Elsevier Science Publishers B.V., 1990.
10. H. Emmelmann. Code selection by regularly controlled term rewriting. In R. Giegerich and S. L. Graham, editors, *Code generation—concepts, tools, techniques*, Workshops in Computing Series, pages 3–29. Springer-Verlag, New York–Heidelberg–Berlin, 1991.
11. H. Emmelmann, F. W. Schröer, and R. Landwehr. BEG—a generator for efficient back ends. *ACM SIGPLAN Notices*, 24(7):246–257, July 1989.
12. C. Ferdinand, H. Seidl, and R. Wilhelm. Tree automata for code selection. *Acta Informatica*, 31(8):741–760, 1994.
13. C. W. Fraser, D. R. Hanson, and T. A. Proebsting. Engineering a simple, efficient code-generator generator. *ACM Letters on Programming Languages and Systems*, 1(3):213–226, September 1992.
14. C. W. Fraser, R. R. Henry, and T. A. Proebsting. BURG—fast optimal instruction selection and tree parsing. *ACM SIGPLAN Notices*, 27(4):68–76, July 1992.
15. R. Giegerich. Code selection by inversion of order-sorted derivors. *Theoretical Computer Science*, 73:177–211, 1990.
16. R. Giegerich and K. Schmal. Code selection techniques: pattern matching, tree parsing, and inversion of derivors. In H. Ganzinger, editor, *Proc. 2nd European Symp. on Programming*, volume 300 of *Lecture Notes in Computer Science*, pages 247–268. Springer-Verlag, New York–Heidelberg–Berlin, 1988.
17. P. J. Hatcher and T. W. Christopher. High-quality code generation via bottom-up tree pattern matching. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 119–130, Tampa Bay, Florida, January 1986.
18. C. Hemerik and J.-P. Katoen. Bottom-up tree acceptors. *Science of Computer Programming*, 13:51–72, January 1990.
19. R. R. Henry. The codegen user's manual. Technical report 87-08-04, Computer Science Department, University of Washington, Seattle, Washington, October 1988.
20. R. R. Henry. Encoding optimal pattern selection in a table-driven bottom-up tree-pattern matcher. Technical report 89-02-04, Computer Science Department, University of Washington, Seattle, Washington, February 1989.
21. R. R. Henry and P. C. Damron. Algorithms for table-driven generators using tree-pattern matching. Technical report 89-02-03, Computer Science Department, University of Washington, Seattle, Washington, February 1989.
22. R. R. Henry and P. C. Damron. Performance of table-driven code generators using tree-pattern matching. Technical report 89-02-02, Computer Science Department, University of Washington, Seattle, Washington, February 1989.
23. C. M. Hoffmann and M. J. O'Donnell. Pattern matching in trees. *Journal of the ACM*, 29(1):68–95, January 1982.
24. L. Kanal and V. Kumar, editors. *Search in Artificial Intelligence*. Springer, 1988.
25. H. Kron. *Tree Templates and Subtree Transformational Grammars*. PhD thesis, Information Sciences Department, University of California, Santa Cruz, CA, 1975.

26. N. J. Nilsson. *Problem-solving methods in artificial intelligence.* McGraw-Hill, New York, 1971.
27. N. J. Nilsson. *Principles of artificial intelligence.* Morgan Kaufmann Publishers, Palo Alto, CA, 1980.
28. A. Nymeyer and J.-P. Katoen. Code generation based on formal BURS theory and heuristic search. Technical report 95–42, Department of Computer Science, University of Twente, Enschede, The Netherlands, November 1995.
29. E. Pelegrí-Llopart. *Rewrite systems, pattern matching, and code generation.* PhD thesis, University of California, Berkeley, December 1987. (Also as Technical Report CSD-88-423).
30. E. Pelegrí-Llopart and S. L. Graham. Optimal code generation for expression trees: An application of BURS theory. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 294–308, San Diego, CA, January 1988.
31. T. A. Proebsting. BURS automata generation. *ACM Transactions on Programming Languages and Systems*, 3(17):461–486, 1995.
32. B. Weisgerber and R. Wilhelm. Two tree pattern matchers for code selection. In D. Hammer, editor, *Compiler compilers and high speed compilation*, volume 371 of *Lecture Notes in Computer Science*, pages 215–229. Springer-Verlag, New York–Heidelberg–Berlin, October 1989.
33. W. A. Wulf, B. W. Leverett, R. G. G. Cattell, S. O. Hobbs, J. M. Newcomer, A. H. Reiner, and B. R. Schatz. An overview of the production-quality compiler compiler project. *IEEE Computer*, 13(8):38–49, August 1980.