Optimal Code Generation for Expression Trees: An Application of BURS Theory†

Eduardo Pelegrí-Llopart

Susan L. Graham

Computer Science Division
EECS Department
University of California, Berkeley

Summary

A ***Rewrite System*** is a collection of ***rewrite rules*** of the form $\alpha \rightarrow \beta$ where α and β are tree patterns. A rewrite system can be extended by associating a cost with each rewrite rule, and by defining the cost of a rewrite sequence as the sum of the costs of all the rewrite rules in the sequence. The REACHABILITY problem for a rewrite system $\boldsymbol{R}$ is, given an input tree T and a fixed ***goal*** tree G, to determine if there exists a rewrite sequence in $\boldsymbol{R}$, rewriting T into G and, if so, to obtain one such sequence. The C-REACHABILITY problem is similar except that the obtained sequence must have minimal cost among all those sequences rewriting T into G.

This paper introduces a class of rewrite systems called Bottom-Up Rewrite Systems (BURS), and a table-driven algorithm to solve REACHABILITY for members of the class. This algorithm is then modified to solve C-REACHABILITY and specialized for a subclass of BURS so that all cost manipulation is encoded into the tables and is not performed explicitly at solving time. The subclass extends the ***simple machine grammars*** [AGH84], rewrite systems used to describe target machine architectures for code generation, by allowing additional types of rewrite rules such as commutativity transformations.

A table-driven code generator based on solving C-REACHABILITY has been implemented and tested with several machine descriptions. The code generator solves C-REACHABILITY faster than a comparable solver based on Graham-Glanville techniques [AGH84] (a non-optimal technique), yet requires only slightly larger tables. The code generator runs much faster than recent proposals to solve C-REACHABILITY that use pattern matching and deal with costs explicitly at solving time [AGT86, HeD87, WeW86]. The BURS theory generalizes and unifies the bottom-up approaches of Henry/Damron [HeD87] and Weisgerber/Wilhelm [WeW86].

1. Introduction

Trees are convenient representations for many applications because of their hierarchical structure and the ease with which they can be manipulated. Frequently this manipulation corresponds to transformations between different tree representations. In this paper we study a mechanism to describe tree transformations and rewrite systems, together with a specific tree transformation problem, REACHABILITY, and its application to the generation of optimal code for expression trees.

In this paper, ***trees*** are denoted either either by graphs (as in Figure 1.1) or by a prefix linearization. For example, $op(T_1,T_2)$ denotes the tree with root op and subtrees T_1 and T_2. The node labels are taken from an alphabet ***Op*** of operators and all operators are assumed to have fixed arity. ***Patterns*** are trees over an alphabet that has been extended with new symbols with arity 0 called ***variables***. In the examples, variables are represented by X, Y, or X_i $(i \geq 0)$, and all other symbols stand for operators. If σ is a value assignment for variables present in a pattern ρ, $\sigma(\rho)$ denotes the replacement of the variables by the values associated by σ. ρ ***matches*** at a tree T if there is an assignment of values to the variables in the pattern, σ, such that $\sigma(\rho)$ is T. Thus, the pattern $+(X,Y)$ matches at any tree rooted with + and having two subtrees, corresponding to X and Y respectively. Two patterns ρ_1 and ρ_2 are said to be ***equivalent*** if they are identical up to a systematic

† This research was partially sponsored by Defense Advance Research Projects Agency (DoD) Arpa Order No. 4871, monitored by Naval Electronic Systems Command under Contract No. N00039-84-C-0089.

Proceedings of the Fifteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, San Diego, California (January 1988)

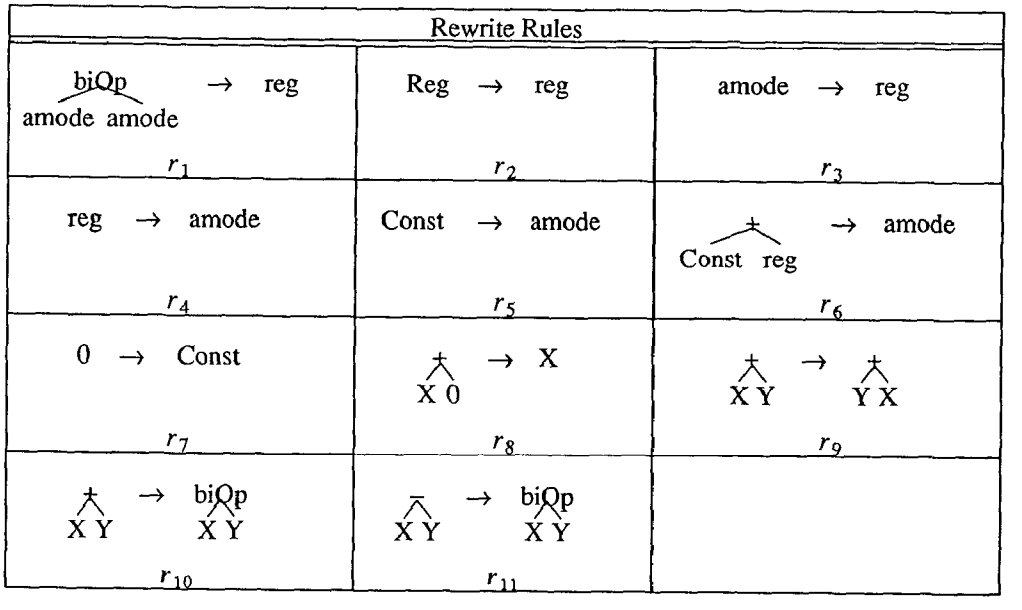
renaming of the variables. All patterns in this paper are ***linear***, i.e., every variable appears at most once. A ***rewrite rule*** is of the form $\alpha \rightarrow \beta$, where α and β are ***patterns***; α is called the ***input pattern*** and β the ***output pattern***, and all the variables in β must appear in α. A ***rewrite system*** is just a collection of rewrite rules. Figure 1.1 shows a ***rewrite system*** that we will use in our examples.

A ***position*** in a tree is a sequence of integers (separated by $\cdot$ for readability) representing a "path" from the root of the tree to a node in the tree. If p is a position in T, the subtree of T rooted at p is denoted by $T_{@p}$. The root position in a tree is designated by the empty sequence ε; each integer corresponds to an index from left to right commencing with 1. If $k \cdot s$ is a sequence with head an integer k and tail a sequence s, and $\triangleq$ is read as "is defined as", positions and subtrees are related as follows:

$$T_{@\varepsilon} \triangleq T$$
$$op(T_1, \ldots, T_n)_{@k \cdot s} \triangleq (T_k)_{@s}, \text{ if } 1 \le k \le n.$$

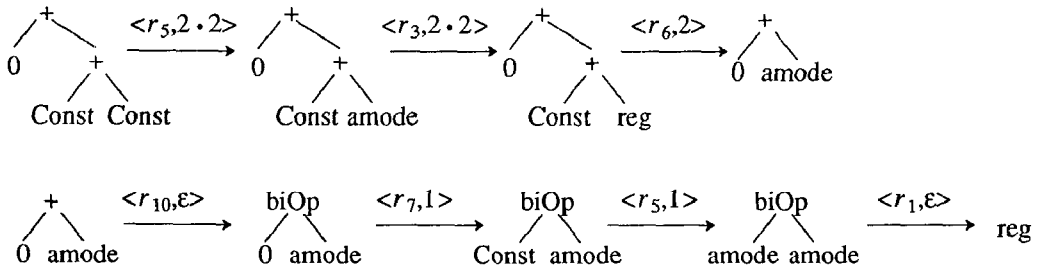
A rewrite rule $r: \alpha \rightarrow \beta$ is ***applicable*** to a tree T at a position p if α matches at $T_{@p}$. If r is applicable to T at p with variable assignment σ, the ***application*** of r to T at p is a new tree, identical to T except that the subtree $T_{@p}$ is replaced by $\sigma(\beta)$. A rewrite sequence is just a sequence of applications of rewrite rules:

Definition 1.1 *A **rewrite application** for a rewrite system **R** is a pair $<r,p>$ where r is a rule in **R** and p is a position. A **rewrite sequence** for **R** is a sequence τ of rewrite applications. If $\tau = <r_0,p_0> \cdots <r_n,p_n>$ is a rewrite sequence, then τ is **applicable** to a tree T if r_0 is applicable to $T_{@p_0}$ and its application yields T_1, and for $1 \le i < n$, r_i is applicable to $(T_i)_{@p_i}$ and its application is T_{i+1}. The application of τ to T is denoted $\tau(T)$ and is T_{n+1}. We say that a rewrite sequence is **valid** if there is some tree to*



Example of a Rewrite System

Figure 1.1



A Rewrite Sequence

Figure 1.2

which it is applicable. The length of a rewrite sequence is the number of rewrite applications in it. The ***composition*** *of a rewrite sequence is a rewrite rule (possibly not in* ***R****) that is applicable whenever the rewrite sequence is applicable and always yields the same result as the sequence.*

If τ *is a rewrite sequence for* T *such that all the applications in* τ *have positions below* p *(that is, position* p *is an initial sequence of all the application positions), the* ***restriction*** *of* τ *to* p, $\tau_{@p}$, *is the sequence of applications identical to* τ *except that every position is stripped of the initial sequence corresponding to* p.

Not every rewrite sequence has a composition. A rewrite system R defines transformations between sets of trees through its rewrite sequences: a tree T can be mapped into a tree T' if there exists a rewrite sequence in R taking T into T'. The transformation is, in general, many-to-many.

The problems studied in this paper are the following:

Problem REACHABILITY *Let* ***R*** *be a rewrite system over an alphabet* ***Op****, and let* L_i *and* L_o *be two sets of trees over* ***Op****. The* REACHABILITY *problem for* ***R****,* L_i, *and* L_o *is, given any* $T \in L_i$ *and any* $T' \in L_o$, *determine if there is a rewrite sequence* τ *for* ***R*** *applicable to* T *such that* $\tau(T) \equiv T'$, *and, if so, to produce one such sequence.*

If L_o *is a singleton* $\{G\}$, *then the* REACHABILITY *problem is called the* ***fixed goal*** REACHABILITY *problem, and* G *is called the* ***goal****. The* BLOCKING *problem for* ***R****,* L_i, *and goal* G *is to determine if there exists a tree* $T \in L_i$ *that cannot be rewritten into* G *by* ***R****.*

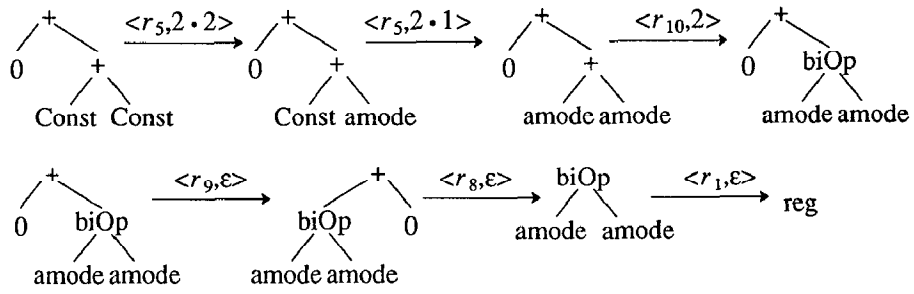
This paper is only concerned with the fixed-goal version of REACHABILITY, and with BLOCKING; see [Pel87] for some considerations on general REACHABILITY. In our example, given the input tree +(0,+(*Const*,*Const*)), one solution to REACHABILITY for goal *reg* is the sequence of Figure 1.2. If L_i consists only of trees with labels *Reg*, *Const*, 0, +, and –, the rewrite system of Figure 1.1, never blocks.

A rewrite system can be extended by assigning a cost to each rewrite rule. The cost of a rewrite sequence for an extended rewrite system can then be defined as the sum of the costs of all the rewrite rules in the sequence. This leads to a variation of the REACHABILITY problem, called C-REACHABILITY, where the objective is not only to provide a rewrite sequence but to provide one with minimum cost.

Returning to our example, if the cost of each of the rewrite rules of Figure 1.1 were defined to be 1, then the cost of the sequence of Figure 1.2 would be 7. That rewrite sequence is not a solution to C-REACHABILITY; the smallest possible cost is 6, and may be obtained by the sequence of Figure 1.3.

The rewrite system of Figure 1.1 is chosen so that applicable rewrite sequences correspond to instructions for some (hypothetical) target machine[1]. Hence, the rewrite sequences in Figure 1.2 and Figure 1.3 correspond to instruction sequences for the given input tree and for the target machine. If the rewrite system accurately describes the target machine, a solution to REACHABILITY provides a

[1] r_2 might be used to generate a register-to-register move if the input register could not be modified.



A Better Rewrite Sequence

Figure 1.3

correct sequence of instructions for the input tree. If, in addition, the costs of the rewrite rules correctly represent the desired properties of the target machine, a solution to C-REACHABILITY provides a locally optimal instruction sequence. BLOCKING corresponds to detecting the existence of an input tree for which code cannot be generated. Examples of typical cost metrics are the number of cycles, and the number of bytes referenced. In the presence of features like pipelines and caches, the number of cycles will be only a static approximation of the execution costs. Previous research (notably [GrH84, Hen84]) has shown how to write target machine descriptions using a variety of techniques, and allows us to conclude that although some features, like constraints on the number of registers, must be handled outside the framework of C-REACHABILITY, an efficient algorithm to solve C-REACHABILITY can be used to provide an efficient algorithm for locally optimal code generation.

The rest of this paper is organized as follows. Section 2 shows how to solve REACHABILITY for a special class of rewrite systems, then Section 3 modifies the techniques and applies them to solving C-REACHABILITY. Finally, Section 4 discusses a code generator generator implemented following the theory of the previous sections. The paper concludes with a discussion of related work.

2. Solving REACHABILITY

REACHABILITY is solved by characterizing all the possible rewrite sequences with a bottom-up tree automaton [Tha73]. We use two notions of state: local rewrite graphs (LR graphs) and uniquely invertible local rewrite graphs (UI LR graphs). Without loss of generality, we assume that the goal tree for fixed-goal REACHABILITY is a leaf labeled with a distinguished nullary operator which appears only as an output pattern in a rewrite rule.

The first step in defining LR graphs is to restrict attention to rewrite sequences in a **normal form** that rewrites the input tree bottom-up. A rewrite sequence can be put in normal form by "reordering" the rewrite applications.

Definition 2.1 *Let r_0 and r_1 be two rewrite rules in $\mathbf{R}$, and let $\tau \equiv <r_0,p_0><r_1,p_1>$ be a valid rewrite sequence. An **exchange** of the two applications is a new rewrite sequence τ' of the form $<r_1,p_2><r_0,p_3>$ such that for all T, τ is applicable to T if and only if τ' is applicable to T, and when so, $\tau(T) \equiv \tau'(T)$.*

*If τ_1 and τ_2 are two rewrite sequences in $\mathbf{R}$, τ_1 is a **permutation** of τ_2 if τ_1 can be obtained from τ_2 through a sequence of exchanges.*

A rewrite sequence τ_1 is said to "loop" if it contains a proper prefix subsequence τ_2 such that, for some tree T, $\tau_1(T) \equiv \tau_2(T)$. All non-looping rewrite sequences can be reordered so that they proceed in a "bottom-up fashion", namely, so that any rewrite application at a position p is preceded by all rewrite applications having positions below p which can be reordered in that way. For example, the sequence of Figure 1.2 can be placed into bottom-up form by reordering the subsequence $<r_{10},\varepsilon><r_7,1><r_5,1>$ as $<r_7,1><r_5,1><r_{10},\varepsilon>$. The rewrite sequence of Figure 1.3 is already in bottom-up form. The notion is formalized as follows:

Definition 2.2 *Let $T \equiv op(T_1, \ldots, T_n)$ be a tree in some input set L, let τ be a rewrite sequence without loops transforming T into a tree T'. τ is in* ***normal form at*** *T, if it is of the form $\tau_1 \cdots \tau_n \tau_0$, and, (1) for $1 \le i \le n$, τ_i only contains applications in positions in the subtree T_i, and the restriction of τ_i to the position i, $(\tau_i)_{@i}(T_i)$, is T'_i; (2) τ_0 applied to $op(T'_1, \ldots, T'_n)$ yields the output tree T'; and (3) there is no permutation of τ satisfying (1) and (2) and in which some rewrites from τ_0 have been moved into τ_i for $1 \le i \le n$. τ is in* ***normal form*** *(everywhere) if it is in normal form at T, and, (4) for $1 \le i \le n$, $(\tau_i)_{@i}$ is in normal form.*

A normal form rewrite sequence for an input tree assigns to each position in the tree a "local" rewrite sequence: those rewrites done at that position. Formally:

Definition 2.3 *Let τ be a normal form rewrite sequence for T of the form $\tau_1 \cdots \tau_n \tau_0$. The* ***local rewrite sequence*** *assigned by τ to a position p in T is defined by $F(T, \tau, p)$, where (1) $F(T, \tau, \varepsilon)$ is τ_0, and (2) if p is of the form $i \cdot q$ and T is of the form $op(T_1, \ldots, T_n)$, then $F(T, \tau, p)$ is $F(T_i, \tau_i, q)$. The* ***local rewrite assignment*** *of τ and T is the function assigning to each position in T its local rewrite sequence.*

For example, the local rewrite sequence assigned by the rewrite sequence of Figure 1.3 at the root of the input tree is $\langle r_9, \varepsilon \rangle \langle r_8, \varepsilon \rangle \langle r_1, \varepsilon \rangle$.

We can now define the k-BURS and BURS properties.

Definition 2.4 *Let k be a positive integer, and let τ be a rewrite sequence in normal form for some input tree T. τ is in* ***k-normal form*** *if it is in normal form and each of the local rewrite sequences assigned by τ to the nodes of T is of length at most k.*

Let $\boldsymbol{R}$ be a rewrite system over $\boldsymbol{Op}$, let $\boldsymbol{L}_i$ and $\boldsymbol{L}_o$ be sets of trees over $\boldsymbol{Op}$, and let k be a positive integer. The triple $\langle \boldsymbol{R}, \boldsymbol{L}_i, \boldsymbol{L}_o \rangle$ is said to have the ***k-BURS property*** *if for any two trees $T \in \boldsymbol{L}_i$ and $T' \in \boldsymbol{L}_o$ and any sequence τ in $\boldsymbol{R}$, with $\tau(T) = T'$, there is a permutation of τ which is in k-normal form. The class* ***BURS*** *is composed of those triples $\langle \boldsymbol{R}, \boldsymbol{L}_i, \boldsymbol{L}_o \rangle$ satisfying the k-BURS property for some positive integer k.*

Since we are considering only fixed-goal problems in this paper, $\boldsymbol{L}_o$ is normally understood to be $\{goal\}$. If $\boldsymbol{L}_i$ is not specified, it is understood to be the set of all trees over the given set of symbols, $\boldsymbol{Op}$, which we denote as $\boldsymbol{L}_{Op}$.

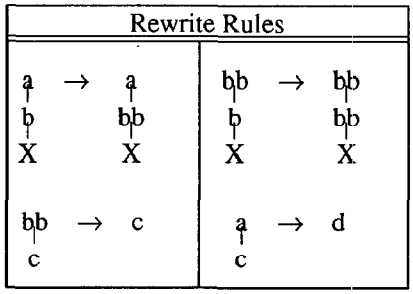
There are rewrite systems and sets of trees not in BURS. The rewrite system of Figure 2.1 with goal d is one example: the local rewrite sequence of the (unique) normal form rewrite sequence rewriting the input tree $a(b(b(\cdots b(c))))$ into d has length dependent on the height of the input tree. In contrast, the example of Figure 1.1 with goal *reg* satisfies the BURS property for $k=3$.

Testing membership in k-BURS is easy when both $\boldsymbol{L}_i$ and $\boldsymbol{L}_o$ are $\boldsymbol{L}_{Op}$.

Proposition 2.1 *Let $\boldsymbol{R}$ be a rewrite system over a set of operators $\boldsymbol{Op}$, and let k be a positive integer. There is an algorithm that will determine whether $\langle \boldsymbol{R}, \boldsymbol{L}_{Op}, \boldsymbol{L}_{Op} \rangle$ is in k-BURS.*

Proof

We can characterize the form of any local rewrite sequence at some position p in a tree. The first observation is that it must start with a rewrite application at position p because, otherwise, this first rewrite application would be assigned to the local rewrite sequence of a position below p. The second observation is similar but requires an additional notion. If ρ is a pattern, let $T(\rho)$ denote the set of positions in ρ that do not correspond to a variable; we call these the positions "touched" by the pattern. If $\langle r, p \rangle$ is a rewrite application and r is $\alpha \rightarrow \beta$, we define $T(\langle r, p \rangle)$ to be the union of those positions of the form $p \cdot q$ where q is in $T(\alpha) \cup T(\beta)$. Finally, if τ is a



Example of a Rewrite System not in BURS

Figure 2.1

rewrite sequence, define $T(\tau)$ as the union all the sets $T(<r,p>)$ for rewrite applications $<r,p>$ in τ. The second observation now states that, if τ_0 is a local rewrite sequence then, for any prefix of τ_0 of the form $\tau \cdot <r,p>$, the position p must be in $T(\tau)$. As before, the reason is that otherwise $<r,p>$ could be moved ahead and would belong to the local rewrite sequence of a position below p.

From the characterization, it follows that there are a finite number of local rewrite sequences of length no larger than k, and they can be generated. Given a rewrite system R, $<R,L_{Op},L_{Op}>$ is in k-BURS if and only if there is not a rewrite sequence of length $k+1$, which can be found by generating and testing all the candidates. □

It follows from the characterization used in the proof that every local rewrite sequence has a composition (which may not be in R).

The rewrite systems used to describe target machines are BURS.

Definition 2.5 *Let $\alpha \rightarrow \beta$ be a rewrite rule. We say that the rule is: an* ***instruction fragment*** *rule if α is a tree without variables and β is a (0-ary) symbol; a* ***generic operator*** *rule if α and β are $op(X_1,\ldots,X_n)$ and $op'(X_1,\ldots,X_n)$, for some n-ary symbols op and op'; a* ***commutativity*** *rule if α and β are $op(X_1,\ldots,X_n)$ and $op(X_{\pi(1)},\ldots,X_{\pi(n)})$, for some n-ary operator op and some permutation π; and an* ***identity*** *rule if α and β are $op(X,T)$ and X, for some tree T that has no variables.*

A ***simple machine grammar*** *is a rewrite system with only instruction fragment and generic operator rewrites*[2].

In Figure 1.1 rules r_1 to r_7 are instruction fragment rules, and rules r_{10} and r_{11} are generic operator rules. Rule r_9 is a commutativity rule, while rule r_8 is a identity rule. The proof of Proposition 2.1 can be used to show:

Proposition 2.2 *Simple machine grammars are in BURS. Machine grammars extended with commutativity and identity rules are in BURS.*

A local rewrite assignment provides a decomposition of the original rewrite sequence: the concatenation of the local rewrite sequences of the input tree in post-order traversal order yields a permutation of the original rewrite sequence. This decomposition can be used to define our first notion of a state for solving REACHABILITY. The ***local rewrite graph*** (LR graph) of a tree T represents the local rewrite sequences of all normal form rewrite sequences applicable to T. For a rewrite system R and a goal G, we consider two sets of patterns: $I_{R,G}$ are the patterns of interest at the beginning of local rewrite sequences, and $O_{R,G}$ are the patterns of interest at the end of the local rewrite sequences and are used to construct members of I higher in the tree. $EF_{R,G}$ is their union.

Definition 2.6 *If R is a rewrite system, and G is the fixed goal, the* ***extended pattern set*** *of R and G, $EF_{R,G}$, is the union of the sets $I_{R,G}$ (the inputs), and $O_{R,G}$ (the outputs), defined constructively below.*

(1) G belongs to $O_{R,G}$.

(2) For some input tree T, position p, and some normal form rewrite sequence, let τ be a local rewrite sequence with composition $\alpha_\tau \rightarrow \beta_\tau$. Let ρ be a pattern in $O_{R,G}$ with variables renamed, if necessary, to be distinct from those in β_τ. If there is a substitution σ such that $\sigma(\beta_\tau)=\sigma(\rho)$, then $\sigma(\beta_\tau)$ is in $O_{R,G}$, $\sigma(\alpha_\tau)$ is in $I_{R,G}$, and all the proper subtrees of $\sigma(\alpha_\tau)$ belong to $O_{R,G}$.

Now we can define LR-graphs.

Definition 2.7 *Let R be a rewrite system in BURS. The* ***LR graph*** *associated with a tree T is a graph $G=(V,E)$ defined as follows.*

Let A be the set of pairs $<T_{in},\tau_0>$ such that there is a normal form rewrite sequence for T of the form $\tau_1\tau_2\cdots\tau_n\tau_0$ and $\tau_n(\cdots\tau_2(\tau_1(T))\cdots)$ is T_{in} and $\tau_0(T_{in})$ is T_{out}. For every local rewrite sequence τ such that there is a $<T_{in},\tau>\in A$, let $\alpha_\tau \rightarrow \beta_\tau$ be its composition. If τ has n rewrite applications, let $pre(\tau,1)$,..., $pre(\tau,n)\equiv\tau$ be the prefix subsequences of τ. Let B be the set of trees of the form $\sigma(\beta_\tau)$ where, for some T_i, $<T_i,\tau>$ is in A, $\sigma(\alpha_\tau)$ matches at T_i, and $\sigma(\beta_\tau)=\sigma(\rho)$ for some ρ in O_R. Finally, define B' as a set of representatives of B under the equivalence relation between patterns.

For every pair $<T_{in},\tau>\in A$ and every substitution σ with $\sigma(\beta_\tau)\in B'$, $\sigma(\alpha_\tau)$, $\sigma(\beta_{pre(\tau,1)})$,..., $\sigma(\beta_{pre(\tau,n)})$ are nodes in V and there is an edge in E between each successive pair of them. There are no other nodes in V or edges in

[2] For example those used by Henry in [Hen84]. Henry handles commutativity explicitly by adding patterns. Identity rules are recognized by a peephole optimizer or prior to instruction selection.

E.

The nodes corresponding to $\sigma(\alpha_\tau)$ *are called the* ***input nodes****, and those corresponding to* $\sigma(\beta_\tau)$ *(those in* B'*) are called the* ***output nodes****. The remaining nodes, corresponding to* $\sigma(\beta_{pre(\tau,j)})$ *for* $1 \le j < n$*, are called the* ***intermediate*** *nodes.*

Note that the union of all the input nodes in all the LR graphs gives $I_{R,G}$, the union of all the output nodes gives $O_{R,G}$, and the union of the input and the output nodes gives $EF_{R,G}$. Figure 2.2 shows an input tree and its associated LR graph for our rewrite system. The input trees are shown inside broken circles, while the output trees are in complete circles. The goal is *reg*.

The notion of an LR graph leads to the following procedure for solving REACHABILITY for rewrite systems in k-BURS:

1 Compute the LR graphs of all the subtrees of the input tree T.

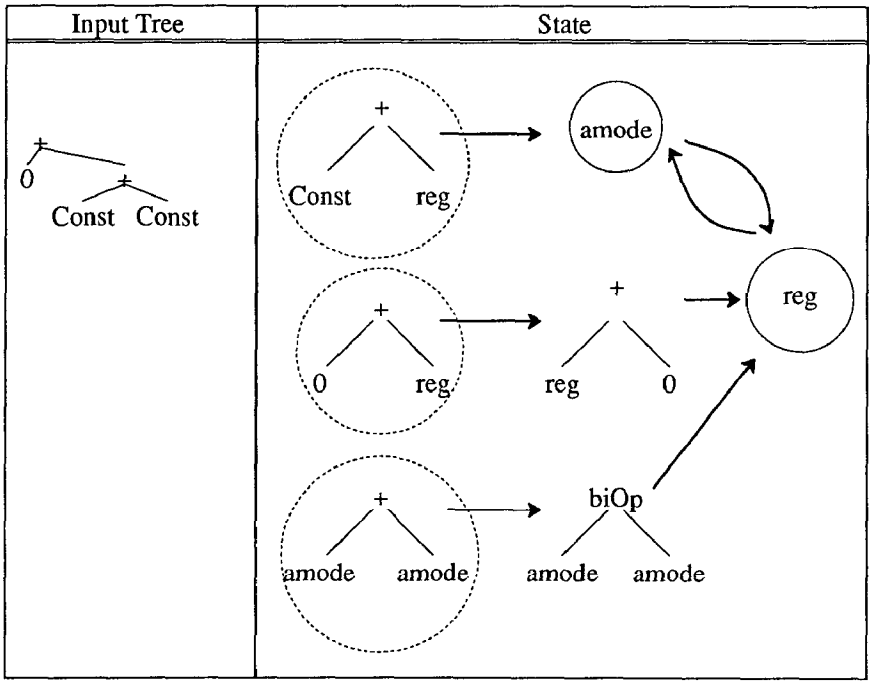
2 If the goal G does not appear in the LR graph of T, then there is no rewrite sequence from T into G; i.e., T "blocks" [GlG78]. If G does appear, assign to each position of T a local rewrite sequence by applying steps 3 to 5 recursively starting with T_{in} being T and T_{out} being G.

3 Select a local rewrite sequence for $T_{in}=op(T_1, \ldots, T_n)$ by selecting any path in the LR graph corresponding to a local rewrite sequence τ_0 from some input tree $op(T'_1, \ldots, T'_n)$ into an output tree ρ such that there is a substitution σ with $\sigma(\rho)=\sigma(T_{out})$.

4 Recursively apply (3) to input T_1 and goal T'_1,..., and to input T_n and goal T'_n.

5 Combine all the local rewrites in post-order to yield a normal-form rewrite sequence for T_{in} into T_{out}.

The procedure is non-deterministic since any path can be chosen in Step 3. The first step in the procedure



Example of a BURS-state

Figure 2.2

and the test in the second step determine whether there is an appropriate rewrite sequence. The remainder of the procedure produces one such sequence.

The first step assumes that it is possible to compute the LR graphs. An important special case when this is possible is when the extended pattern set is finite. In this case the collection of all the LR graphs that may be assigned to any subtree of any input tree is finite. We call this subclass of BURS *finite BURS*.

A semi-decision procedure for membership in finite k-BURS in the case that the input set L_i is L_{Op} first tests k-BURS and generates all the local rewrite sequences, using the characterization used in the proof of Proposition 2.1, and then tries to generate the extended pattern set, following Definition 2.6. If the procedure terminates, then the extended pattern set can be used to obtain the collection of all the LR graphs.

It is not difficult to see that the extended pattern set of machine grammars as defined in Definition 2.5 is finite. Thus:

Proposition 2.3 *Every simple machine grammar is finite BURS. Machine grammars extended with commutativity and identity rewrites are also finite BURS.*

If the rewrite system is finite BURS, then each one of the individual steps of the algorithm for solving fixed REACHABILITY can be precomputed, stored into a table, and replaced, at REACHABILITY-solving time, by a table lookup. This leads to a typical "table-generator plus solver" approach to REACHABILITY. The table generator computes all the possible LR graphs, and stores their interactions into tables. The solver then consults them. By moving computation into the table generator, the solver can proceed very rapidly.

If the rewrite system is finite BURS, it is also possible to solve BLOCKING efficiently. If the input set is L_{Op}, there will be a blocking tree if there is an LR graph that does not contain the goal G as a node. If the rewrite system has the property that $<R,L_{Op},L_{Op}>$ is in finite BURS, and S is a recognizable [Tha73] subset of L_{Op}, we can find the LR graphs that are useful for trees in S, and we can find whether there is a tree in S for which R blocks. Both problems are solved by constructing the bottom-up tree automaton recognizing S and "running it against" the automaton computing LR graphs; see [Pel87] for details.

In general, an LR graph contains more than one path within the state which leads to an output tree. But it is only necessary to keep one alternative for solving REACHABILITY. Consequently, it is possible to use the same REACHABILITY algorithm and to replace the LR graph by any subgraph of it such that (i) it contains all the output nodes, (ii) every node has at most one entering edge, and (iii) for every output node there is at least one directed path with all its nodes in the subgraph from an input node to the output node. (Since all input nodes are reachable, all but one of them can be omitted). Such a graph is called a ***uniquely invertible local graph*** (UI LR graph), and is the second notion of state used to solve REACHABILITY.

Since the same UI LR graph may be a subgraph of several different LR graphs, choosing the UI LR graphs carefully may allow a reduction in the number of states needed. For example, in Figure 2.2 there are many different ways of obtaining *reg*; any one of them is good enough. If the path starting from +(*amode*,*amode*) is selected, this state could also be used for many other trees including, for example, +(0,*Reg*) and +(*Reg*,0). Unfortunately, selecting the UI LR graphs so as to minimize the total number required is a complex problem.

Proposition 2.4 *Given a rewrite system R over Op, and a set of trees L_o over Op, with $<R,L_{Op},L_o>\in$ finite BURS, the MINIMUM UI LR GRAPH problem consists of assigning to each LR graph a valid UI LR graph such that the number of UI LR graphs used is minimum. MINIMUM UI LR GRAPH is NP-complete.*

Proof by reduction of MINIMUM COVER [GaJ80]; quite straight-forward, see [Pel87]. □

The selection process is further complicated because selecting some paths in an LR graph may make some trees in the graph "useless" for solving REACHABILITY, which may open new opportunities for making graphs equivalent. Section 4 below describes a heuristic used to select the UI LR graphs, as well as the table representation used by the solver. Detection of useless nodes in the graphs can be done by a simple iteration process.

REACHABILITY problems can be used in several applications, the rest of this paper shows how to modify the algorithm to solve the C-REACHABILITY problem.

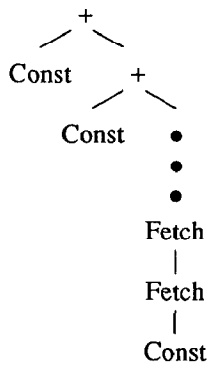
3. Solving C-REACHABILITY

A first approach to solving C-REACHABILITY would be to enrich the notion of an LR graph by using a graph where the nodes are not patterns but pairs (ρ,*cost*) where *cost* represents the minimum cost to reach pattern ρ, and where the edges correspond to rewrite applications along paths of minimal cost. Such an approach works correctly but leads to an unbounded number of states and thus to costly solver-time operations. A better solution is to store, instead of the total cost needed to reach ρ, only the ***delta cost***. The delta cost is defined by substracting from the cost associated with each pattern, the smallest cost associated with any pattern in the LR graph. Since the cost of a sequence is the sum of the costs of all the rewrites in the sequence, choosing a rewrite sequence based in the delta cost yields the same solution as choosing one based on full costs, yet the number of states will be smaller. The resulting notion is called a ***δ-LR graph***.

The delta costs can be computed without first computing the full minimal cost for each pattern. The delta costs of all patterns in the graph can be computed from the delta costs of the input nodes, which are determined by the delta costs of the output nodes of other states.

There is no guarantee that a rewrite system that is in finite-BURS will, when extended with costs, have a finite number of δ-LR graphs. Consider, for example, the rewrite system of Figure 3.1, where the cost of each rule is shown below it.

This rewrite system contains two separate sets of rewrite rules: those involving "imodes" and those involving "amodes". Now consider an input tree of the form:



Whether "imode" rewrites or "amode" rewrites are cheaper depends on the relationship between the number of "+" operators and the number of "Fetch" operators in the tree. Recording this information requires an unbounded number of states.

Rewrite Rules	
Fetch → amode Const 2	Fetch → amode amode 2
+ → amode Const amode 1	amode → goal 0
Const → imode 1	Fetch → imode imode 1
+ → imode Const imode 2	imode → goal 0

An Unbounded Number of δ-LR Graphs

Figure 3.1

Fortunately, the above situation is uncharacteristic of "real" machine descriptions. Real machine descriptions have a symbol, which corresponds to the notion of a "register", that plays a central role: all trees (except maybe a few) can be rewritten into "register" in a short number of rewrites and "register" can rewritten into all trees (except maybe a few) also in a short number of rewrites. This provides a "triangular inequality" that forces together the delta costs associated with the trees in the LR graph and leads to a finite number of δ-LR graphs. See [Pel87] for one possible formalization of this argument.

Testing if there is a finite number of δ-LR graphs can be done as part of the generation of the graphs. Conceptually, the procedure can be understood as first generating the LR graphs and then annotating them with costs and generating more δ-LR graphs until no new graph is found. This procedure will terminate if there is a finite number of δ-LR graphs, but will fail to do so if there is an infinite

number of them.

If there is a finite number of states, then it is possible to apply the same algorithm used for REACHABILITY to solve C-REACHABILITY very efficiently. Unfortunately, a single LR graph may be replaced by several δ-LR graphs, which may lead to a substantially larger number of states. The number of states needed can be reduced in a way similar to that of the previous section by using δ-UI LR graphs instead of δ-LR graphs. In addition, one can observe that the costs associated with the δ-LR graphs (and the δ-UI LR graphs) are not used in solving C-REACHABILITY: they are used to compute the states but, after that, only the graph structure is used, without the cost information. Since two different δ-(UI) LR graphs may have the same structure, it is possible that two different states may be equivalent. The minimal number of states needed can be computed by a variation of the standard algorithm to minimize a bottom-up tree automaton which is, in turn, a variation of the minimization of a finite-state automaton.

4. A Code Generator Generator

We have implemented a code generator generator that works by solving C-REACHABILITY using BURS theory. The table generator implementation is stand-alone and emphasizes generating small tables, with no great effort spent in trying to generate them fast. It has been running since late 1986. The implementation is based on the theory presented in the previous two sections with some modifications. The δ-LR graphs are first generated using an extension of David Chase's algorithm for bottom-up pattern matchers [Cha87]; then useless information is removed and δ-UI LR graphs are selected. Since optimal selection of δ-UI LR graphs is difficult, the selection is done by a process which starts from δ-LR graphs and attempts to make graphs identical by removing some alternatives, determining which nodes in the graphs are useless, removing them, and repeating the process. In our experiments, the number of states stabilizes in two or three iterations.

The modified version of Chase's algorithm generates a representation of the bottom-up tree automaton computing the states (representing the δ-UI LR graphs) as a collection of "folded" tables, one for each n-ary operator, where identical $n-1$-hyperplanes have been found and shared. Our implementation accepts only 0, 1, and 2-ary operators. In the case of binary operators, the representation is a table where identical rows and columns have been found; we call the 1-dimensional arrays indicating identical rows and columns, ***restrictors***. We bit-encode the restrictors and share them across different tables.

The code generator uses the automaton tables and also a second set of tables that encodes each δ-UI LR graph; the costs are not stored since they are unused. This second set of tables is encoded using a technique similar to that in YACC [Joh78], by overlaying rows of information. The table generator uses a few simple heuristics to reduce the table size of these tables, see [Pel87] for details.

A final modification from the theory of the previous section is that the problem that the code generator really wants solved is not C-REACHABILITY. Each rewrite rule of the rewrite system given to the table generator has, in addition to a cost, a call to a semantic routine. What the code generator uses is not a rewrite sequence of minimum cost, but its associated sequence of semantic routine calls. We call this problem UCODE. The minimum number of states needed to solve UCODE can be found using a minimization method like the one mentioned in the previous section.

The BURS code generator has been operative since early 1987, integrated into UW-CODEGEN [HeD87], a testbed for table-driven code generators developed by Robert Henry at the University of Washington. UW-CODEGEN does temporary and register management and includes the following code generators:

- GG — A code generator based on Graham-Glanville technology [GlG78];
- BU — A locally optimal code generator based on bottom-up pattern matching, manipulating states similar to LR graphs but with costs represented explicitly and computed with a dynamic programming algorithm; and
- TD — A locally optimal code generator based on top-down pattern matching technology and manipulating costs explicitly with a dynamic programming algorithm. The trees in a state are listed explicitly.

TD and BU were implemented by Damron and Henry, respectively, and were developed independently of the BURS-based code generator presented in this paper. The theory behind TD is similar to that used in *twig* [AGT86]

and in the top-down algorithms of Weisgerber/Wilhelm [WeW86]. The theory behind BU is related to BURS and to the bottom-up algorithm described by Weisgerber/Wilhelm [WeW86]. The big advantage of the UW-CODEGEN testbed is that it facilitates meaningful code generator comparisons.

We have tested the table constructor with several machine descriptions that were developed at UC Berkeley as part of the CODEGEN effort [AGH84]. This paper only reports on two machine descriptions that were made available by Robert Henry: a Vax-11 description and a Motorola MC68000 description; for technical reasons, they are the only ones that we can use to generate code with UW-CODEGEN. The machine descriptions used are machine grammars without generic operator rewrite rules. The cost assigned to each rule is a 4-tuple indicating the numbers of memory bytes referenced, instructions issued, side effects issued, and operands in the instruction fragment represented by the rule. The tuple leads to 6 "natural" costs: a constant cost, each of the 4 elements considered separately, and a lexicographic ordering on the full tuple. We will denote the 6 costs as K, M, I, S, O, and L, respectively. The GG implementation disregards the cost information; the BU and TD implementations always use full lexicographic cost. BURS currently can use any of the 6 costs except L.

The two principal measures of interest are table size and code generation speed. Table size is related to the number of states needed to solve UCODE, which depends on the cost function used and the method of state construction. Figure 4.1 shows, for the two machine descriptions mentioned, the number of δ-LR graphs that are generated initially and the final number of states needed. The table shows the big variation in the number of states needed: the constant cost function (K) requires few states while the function that counts the memory references (M) requires many. The lexicographic cost would produce a larger number of states but, due to implementation restrictions in our exploratory implementation of the table generator, the tables cannot be generated. An approximation to the lexicographic cost produces tables slightly larger than the largest using a single component. The table also shows that in this example our heuristic to reduce the number of states obtains a significant reduction; we have obtained larger reductions with other machine descriptions.

Costs		Vax-11 orig.	Vax-11 final	Mc-68000 orig.	Mc-68000 final
Constant	K	182	95	190	167
Mem. Refs	M	1733	652	1089	576
Instructions	I	417	270	190	167
Side Effects	S	417	268	213	194
Operands	O	182	95	537	374

Number of BURS States

Figure 4.1

Henry and Damron report in detail on the table sizes for GG, BU, and TD [HeD87]. Figure 4.2 shows the table size for BURS for the different cost metrics. For each machine description and each cost function there are three numbers, listed from the top: the space used to represent the bottom-up tree automaton, the space used to represent the states themselves, i.e. their internal nodes and edges, and the total space. Note that the major variation is in the size of the bottom-up tree automaton. The bottom of the figure shows the influence of the representation of the restrictors on the table size. The three columns indicate the restrictor size, the automaton size, and the total table size. Sharing identical restrictors is a very simple optimization and a big win; bit-encoding the restrictors does not seem to significantly slow the UCODE solver.

Figure 4.3 compares the table sizes for several code generators in UW-CODEGEN. The values for BU, TD, and GG are taken from [HeD87]; the line labelled "states" is the space for the patterns, replacements, costs, and actions; the line labelled "fsa" corresponds to different notions of automaton. The UW-CODEGEN values are estimated from bar charts[3]. The values for BURS are for the M cost function, which is the one requiring the largest tables. BURS-l and BURS-f represent different versions of the table generator: BURS-l is an approximation to L, (note that M is the first component), while BURS-f uses M as cost function but tries to generate the tables fast rather than spending too much time generating small tables. Again there are three numbers per combination of machine description and cost function. They are the size of the automaton, the states,

Automaton, States, and Total Table Size					
	K	M	I	S	O
Vax					
B-fsa	1722	26388	5760	6390	1722
States	7698	15980	12320	12294	7682
Total	9420	42368	18080	18684	9404
Mot					
B-fsa	3652	22500	3652	4542	10320
States	12588	20788	12588	13978	18692
Total	16240	43288	16240	18520	29012

Restrictor Encoding and Size (Vax using M)			
Mode	Restrictor	B-fsa	Total
No Share	77604	94424	110404
Share	30888	47708	63688
No share/Bit	18564	34810	50790
Share/Bit	10142	26388	42368

Table Sizes (bytes)

Figure 4.2

and their sum. Chase's technology would provide a substantially smaller B-fsa than the one used in BU. According to Chase [Cha87], a reasonable value is in the vicinity of 23K; this would place the total table size very close to GG and BURS.

	GG	TD	BU	BURS	BURS-1	BURS-f
Vax						
Fsa	33.7	20.1	56.4	26.3	29.1	46.9
States	8.8	18.2	18.6	15.9	15.8	18.5
Total	42.5	38.3	75.0	42.3	44.9	65.5
Mot						
Fsa	33.0	18.6	50.2	22.5	26.9	35.3
States	8.7	18.2	18.2	20.7	22.3	23.5
Total	41.7	36.8	68.4	43.3	49.3	58.8

Table Sizes (Kbytes)

Figure 4.3

We use the same set of 6 programs used in [HeD87] to measure the performance of the code generator. These are C programs ranging in size from 100 to 1200 lines. Figure 4.4 shows, for each target, three values averaged over the 6 programs: the time spent solving UCODE normalized to GG, the percentage of code generation time spent solving UCODE, and the total code generation time normalized to UCODE. (All measurements were made on a Vax 8600; only "user" time is considered).

BURS is substantially faster than TD and BU because manipulating costs is expensive: they have to be combined, computed, and compared. It is more surprising that BURS is even faster than GG. A careful comparison of the respective portions of code implementing UCODE showed several causes for the difference in speeds. Probably the biggest contribution lies in the representation of the automaton: GG uses a tight encoding and a cache, which loses in speed against the more efficient table folding. In addition, GG uses the normal technique (for parsing technology) of default transitions, which is slower than a simple lookup. Another contributor is that the relationship between the parser used in GG and the traversal of the tree providing the prefix traversal is not as simple as the tree traversal used by BURS. Finally, GG stores states and other information in a stack (the parse stack), while BURS uses (pre-allocated) slots associated with the tree; the stack requires extra checks for overflow and the like. GG also uses a few more indirect routine calls than BURS. Despite the difficulty in comparing the methods in the presence of these differences in implementation strategy, we think that the evidence shows that BURS is, at least, comparable in speed to GG. To reduce effects caused by compilation of the algorithms, the values shown in Figure 4.4 correspond to GG compiled using the peephole optimizer, and BURS without it; the values are more favorable to BURS otherwise.

[3] We are uncertain of the accuracy of some of these numbers.

Vax	GG	TD	BU	BURS
UCODE (rel.)	1.00	3.60	3.10	0.63
% in UCODE	14.04	32.97	31.13	8.74
Total CG	1.00	1.34	1.27	0.93
Mot	GG	TD	BU	BURS
UCODE (rel.)	1.00	3.82	3.22	0.55
% in UCODE	16.61	40.77	37.29	8.81
Total CG	1.00	1.45	1.36	0.94

Code Generation Time

Figure 4.4

The quality of the generated code is measured statically using the same metric that we have discussed earlier: the 4-tuple of values. Figure 4.5 shows the average cost, normalized to 100 for BU and TD. BU and TD have a small error that shows very infrequently and which causes some normalized values to be under 100.00. The quality of the code generated by BURS-1 is quite close to the lexicographic optimum.

The time spent generating the BURS tables depends on the cost function and on the effort spent trying to generate small tables. The top of Figure 4.6 shows times in seconds on a Sun-3/75 with 12 MB of main memory and no local disk. The bottom of the figure reproduces information from [HeD87] comparing the performance of the different table generators in UW-CODEGEN; values are in seconds on a DEC Microvax-II. There are two columns for BU: the first column corresponds to the generation of tables without any effort to use cost information at table-generation time to reduce the number of alternatives to consider at code generation time; the second column corresponds to the tables used in our other comparisons, in which some elimination of alternatives is done based on costs. We want to emphasize that the current implementation of the table generator for BURS was written with no special effort to generate tables fast.

Vax	M	I	S	O
GG	103.90	105.38	99.97	103.53
BURS	100.00	102.77	103.40	100.00
BURS-1	100.00	100.17	101.02	100.00
BURS-f	100.00	101.72	103.42	100.00
Mot	M	I	S	O
GG	102.43	100.00	103.28	98.05
BURS	100.00	100.00	100.00	103.57
BURS-1	100.00	100.00	100.00	99.58
BURS-f	100.00	100.00	100.00	103.57

Quality of the Generated Code

(100 is optimal)

Figure 4.5

5. Other Related Work and Conclusions

The idea behind the algorithm for REACHABILITY has been around for a while; maybe the earliest references are the dynamic programming algorithms of [AUJ77] and [Rip77]. BURS theory differs from these early proposals in that it is based on rewrite systems, it can handle a larger class of rewrite systems, and it emphasizes the computability of the states by a bottom-up finite state automaton. Our theory was developed independently of the work of

Vax	K	M	I	S	O
BURS	132.6	2361.3	398.8	368.4	148.1
BURS-f	94.1	921.3	207.7	196.5	111.7
BURS-1	204.7	4016.3	451.2	391.5	178.2
Mot	K	M	I	S	O
BURS	282.84	2582.	281.5	324.7	841.1
BURS-f	172.4	1757.9	170.7	194.6	518.9
BURS-1	216.7	19984.3	217.6	236.4	1158.3

(Sun 3/75 seconds)

Machine	GG	TD	BU	BU-cost
Vax	204.7	58.0	242.1	625.7
Mot	194.8	61.0	442.9	1753.1

(μVax-II seconds)

Table Generation Times

Figure 4.6

Weisgerber/Wilhelm [WeW86], and Henry/Damron [HeD87]; it differs from the work of those researchers in its ability to encode cost information into the δ-BURS states and in handling a larger class of rewrite systems. Our work on optimal code generation yields results similar to those claimed by Hatcher and Christopher [HaC] [Hat85] but while the Hatcher/Christopher technique requires modifying some parts of the machine description to retain optimality, the approach described here will always be optimal, provided that a finite number of states exist. We suspect that the Hatcher/Christopher technique can be explained as a simplification of BURS-theory.

Probably the best-known implementation for locally optimal code generation is the one used for *twig* [AGT86]. The theory behind that implementation is quite similar to the one used in TD with two differences. The first difference is that the implementation of *twig* reported in [AGT86] does more computation at solving time than TD. Thus, *twig* has smaller tables and smaller table generation times, but larger code generation times. The second difference is in the phase organization. Both *twig* and UW-CODEGEN perform two types of transformations: some transformations are for normalization and simplification, like the mapping of short-circuit booleans into compare and jumps, the others are the ones discussed in this paper and correspond to the machine instructions. *Twig* deals with both types of transformations together in a single mechanism, but the interaction of the machine rewrite applications with the simplification routines allows looping and non-optimal transformations to occur. UW-CODEGEN first performs the normalization and simplification and then the machine rewrite applications, but allows the simplification routines to query the machine description to make decisions. The current implementation of the simplification routines in UW-CODEGEN is pattern-driven and a bit inefficient. A new version recently written by Henry [Hen87] is faster and seems easier to program. Although we don't have specific measures comparing our approach and *twig*, it is safe to say that BURS-based code generation is substantially faster than one based on *twig*. The results of Henry/Damron [HeD87] also suggest that, if one were to model the code generation in a way similar to the one used in *twig*, a bottom-up pattern matcher could be faster than the currently used top-down pattern matcher; the work of Chase [Cha87] and our own shows that the space penalty is manageable.

We have shown the potential for BURS-based fast optimal code generation for expression trees. The main advantage of optimality is that as long as the machine description is accurate, there is no need for the machine description writer to understand the theory used to generate the code generator. A non-optimal technique like GG generates optimal code for a uniform instruction set such as those found on RISC machines [Pat85]. It can generate quite good code otherwise (see Figure 4.5) if the machine description is carefully written [Hen84].

REACHABILITY problems can be used in several other applications. Projection Systems [Pel87] are a descriptive mechanism for tree transformation that is similar to tree-to-tree grammars [KMP84], and can be used, for instance, to describe the mapping between parse trees and abstract syntax trees. Forward and backward applications of projection systems can be reduced to REACHABILITY problems. X-patterns [Pel87] are an extension of traditional patterns to describe non-local conditions. Pattern matching of X-patterns can be reduced to a REACHABILITY problem.

Our current research in the area includes exploring faster algorithms for the table generation, and testing of k-BURS for any recognizable input set. We are also working in other applications of REACHABILITY.

Acknowledgements

We thank Robert Henry for his development of UW-CODEGEN, for several of the machine descriptions used in the experiment, and for many fruitful discussions. David Chase helped us by providing his bottom-up pattern matcher. We gratefully acknowledge the members of our research group for useful conversations and constructive feedback.

6. Bibliography

[AUJ77] A. V. Aho, J. D. Ullman and S. C. Johnson, "Code Generation for Expressions with Common Subexpressions", *Journal of the ACM 24*, 1 (Jan 1977), 146-160.

[AGT86] A. V. Aho, M. Ganapathi and S. W. K. Tjiang, "Code Generation Using Tree Matching and Dynamic Programming", Submitted to ACM Transactions on Programming Languages and Systems., January 1986.

[AGH84] P. Aigrain, S. L. Graham, R. R. Henry, M. K. McKusick and E. Pelegri-Llopart, "Experience with a Graham-Glanville Style Code Generator", *Proceedings of the ACM SIGPLAN 1984 Symposium on Compiler Construction, SIGPLAN Notices 19* , 6 (June 1984).

[Cha87] D. R. Chase, "An Improvement to Bottom-up Tree Pattern Matching", in *POPL87*, Munich, Germany, January 1987.

[GaJ80] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Co., 1980.

[GlG78] R. S. Glanville and S. L. Graham, "A New Method for Compiler Code Generation (Extended Abstract)", *Conference Record of the Fifth ACM Symposium on Principles of Programming Languages*, Tucson, AZ, January 1978, 231-240.

[GrH84] S. L. Graham and R. R. Henry, "Machine Descriptions for Compiler Code Generation: Experience Since JCIT-3", *IEEE 1984 Proceedings of the 4th Jerusalem Conference on Information Technology (JCIT)*, Jerusalem, Israel, 1984.

[Hat85] P. J. Hatcher, *A Tool for High-Quality Code Generation*, PhD Dissertation, Illinois Institute of Technology, Chicago, Illinois, December 1985.

[HaC] P. J. Hatcher and T. W. Christopher, "High-Quality Code Generation Via Bottom-Up Tree Pattern Matching", *POPL86*, .

[Hen84] R. R. Henry, *Graham-Glanville Code Generators*, PhD Dissertation Computer Science Division, EECS, University of California, Berkeley, CA, May 1984.

[Hen87] R. R. Henry, Personal Communication, August 1987.

[HeD87] R. R. Henry and P. C. Damron, "Code Generation Using Tree Pattern Matchers", Technical Report 87-02-04, University of Washington, February 10, 1987.

[Joh78] S. C. Johnson, *YACC: Yet Another Compiler-Compiler*, Bell Laboratories, Murray Hill, NJ, July 1978.

[KMP84] S. E. Keller, S. P. Mardinly, T. F. Payton and J. A. Perkins, "Tree Transformation Techniques and Experiences", *Proceedings of the ACM SIGPLAN 1984 Symposium on Compiler Construction, SIGPLAN Notices 19* , 6 (June 1984).

[Pat85] D. A. Patterson, "Reduced Instruction Set Computers", *Communications of the ACM 28*, 1 (January 1985), 8-21.

[Pel87] E. Pelegri-Llopart, *Tree Transformation Systems in Compiler Systems (tentative title)*, PhD Dissertation, EECS-University of California, Berkeley, December 1987.

[Rip77] K. Ripken, "Formale Beschreibung von Maschinen, Implementieurungen und optimierender Maschinencodeerzeugung aus attributierten Programmgraphen", PhD Dissertation, Technische Universitat Munchen, Munich, West Germany, July 1977.

[Tha73] J. W. Thatcher, "Tree Automata: An Informal Survey", in *Currents in the Theory of Computing*, A. V. Aho (editor), Prentice Hall, Englewood Cliffs, NJ, 1973, 143-172.

[WeW86] B. Weisgerber and R. Wilhelm, *Two Tree Pattern Matcher for Code Selection (Including Targeting)*, Technical Report, Universitat des Saarlandes, Saarbrucken, W. Germany, February 1986.