BURS Automata Generation

TODD A. PROEBSTING
University of Arizona

A simple and efficient algorithm for generating bottom-up rewrite system (BURS) tables is described. A small code-generator generator implementation produces BURS tables efficiently, even for complex instruction set descriptions. The algorithm does not require novel data structures or complicated algorithmic techniques Previously published methods for on-the-fly elimination of states are generalized and simplified to create a new method, *triangle trimming*, that is employed in the algorithm. A prototype implementation, burg, generates BURS tables very efficiently

Categories and Subject Descriptors. D.3.4 [**Programming Languages**] Processors—*code generation*; *compilers*; *translator writing systems and compiler generators*

General Terms: Algorithms, Languages

Additional Key Words and Phrases Code generation, code-generator generator, dynamic programming, tree pattern matching

1. INTRODUCTION

Possibly the simplest way to visualize and understand the complex instructions and addressing modes of a processor is to view them as expression trees in which leaves represent registers, memory locations, or constant values, and internal nodes represent operations on operand values. Describing even the most complex addressing mode is simplified when such trees are used. Figure 1 gives an example of tree patterns.

Because of their expressive power, trees also serve as a natural intermediate representation (IR) to be generated by the front end of a compiler. If the same domain of trees is used to describe machine instructions as is used for the IR, instruction selection for a given IR tree becomes a matter of matching instruction patterns against the generated IR such that the IR is *covered* (parsed) with adjacent patterns. Figure 2 shows two legal covers of the same expression tree. Many techniques are known for finding such coverings efficiently (in time proportional to the size of the IR tree). Equally important, finding a least-cost covering (based on costs associated with the patterns) is also efficient.

Tree pattern matching combined with dynamic programming can be used in code generators to create locally optimal code for expression trees [Aho et al. 1989]. Code generators based on bottom-up rewrite system (BURS) theory can be extremely fast because all dynamic programming is done when the BURS automaton is built.

This work was supported by NSF Grant CCR-9122267
A preliminary version of this paper was presented at the 1992 SIGPLAN Conference on Programming Language Design and Implementation.
Author's address: Department of Computer Science, University of Arizona, Tucson, AZ 85721.

Pattern #	Label →	Pattern	Cost
1	goal →	reg	(0)
2	reg →	Reg	(0)
3	reg →	Int	(1)
4	reg →	Fetch ↓ addr	(2)
5	reg →	Plus reg reg	(2)
6	addr →	reg	(0)
7	addr →	Int	(0)
8	addr →	Plus reg Int	(0)

Fig. 1 Sample machine instruction templates.

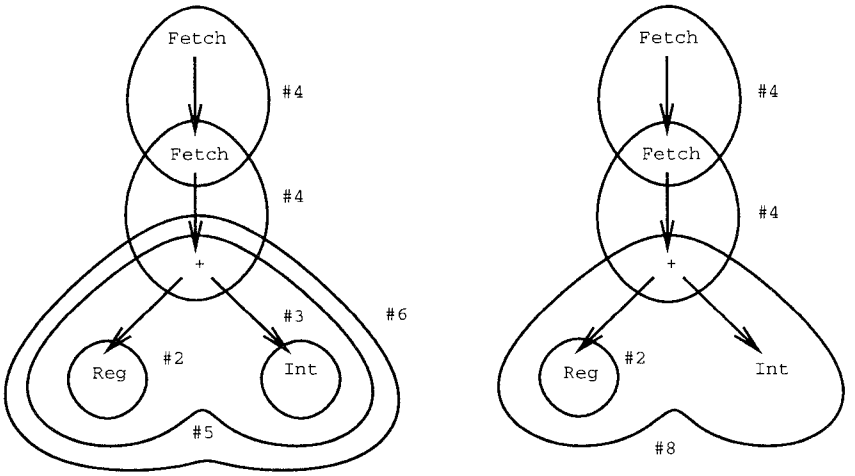

Fig 2 Sample coverings of identical trees

At compile-time, it is only necessary to make two traversals of the subject tree: one bottom-up traversal to label each node with a *state* that encodes all optimal matches and a second top-down traversal that uses these states to select and emit code. Fraser and Henry [1991] report that careful encodings can produce an automaton that executes fewer than 50 VAX instructions ($\approx$ 90 RISC instructions) per node to do both traversals.

The automaton that labels the tree is a simple state-transition machine. A

Table I. Simple Grammar and Its Normal Form

	Simple Grammar				Normal Form			
Rule#	LHS		RHS	Cost	LHS		RHS	Cost
1	goal	→	reg	(0)	goal	→	reg	(0)
2.	reg	→	Reg	(0)	reg	→	Reg	(0)
3	reg	→	Int	(1)	reg	→	Int	(1)
4.	reg	→	Fetch(addr)	(2)	reg	→	Fetch(addr)	(2)
5	reg	→	Plus(reg, reg)	(2)	reg	→	Plus(reg, reg)	(2)
6.	addr	→	reg	(0)	addr	→	reg	(0)
7	addr	→	Int	(0)	addr	→	Int	(0)
8.	addr	→	Plus(reg, Int)	(0)	addr	→	Plus(reg, n.1)	(0)
8a					n.1	→	Int	(0)

bottom-up walk of the tree is performed, and the label for any given node is determined by a table lookup given the operator at the node and the states that label each of its children. The automaton that emits code is equally simple in design. The code to be emitted is determined by the state that labels a node and by the nonterminal to which that node should be reduced—another table lookup.

Two difficulties arise in creating a BURS-style code generator: efficiently generating the states and state transition tables (because *all* potential dynamic programming decisions are done at table generation time, they must be done efficiently) and creating an efficient encoding of the automata for use in a compiler. A solution to the encoding problem is described by Fraser and Henry [1991].

This article describes a new simple and efficient table generation algorithm. The code-generator generator described in this article, `burg`, is based on tree pattern-matching technology and dynamic programming. Simplicity has increased, not decreased, efficiency. Efficiency has been enhanced, and tables sizes have been kept small, by the development of two new techniques, *chain rule trimming* and *triangle trimming*, for eliminating many redundant states. Triangle trimming is an uncomplicated optimization that, for complex grammars, can reduce both the table generation time and table sizes by over 30%. Additional optimizations take advantage of special properties of BURS states.

2. BURS MODEL

The input to a BURS code-generator generator is a set of rules. Each rule indicates a tree pattern, a cost, a replacement symbol, and an action. The set of all the rules is called the *grammar*. Table I gives a small sample grammar (without actions). The replacement symbol is a *nonterminal* on the left of the rule—the linearized tree pattern it derives is on the right. In the sample, `goal`, `reg`, and `addr` are nonterminals. In addition to nonterminals, the grammar has *operators* of varying arities. In the sample, `Reg`, `Int`, `Fetch`, and `Plus` are operators with respective arities of 0, 0, 1, and 2.

A least-cost parse can be found using dynamic programming. By trying all matching patterns at all nodes, it is possible to remember the rules that lead to the cheapest derivation from each possible nonterminal. Figure 3 applies the rules in Table I to the tree representing `Fetch(Fetch(Plus(Reg, Int)))`. Each node is labeled with the least-cost derivation from each nonterminal.

A BURS pattern matcher finds a least-cost parse of a subject tree for the grammar that reduces to the *goal* nonterminal. Each tree node will be labeled with a state

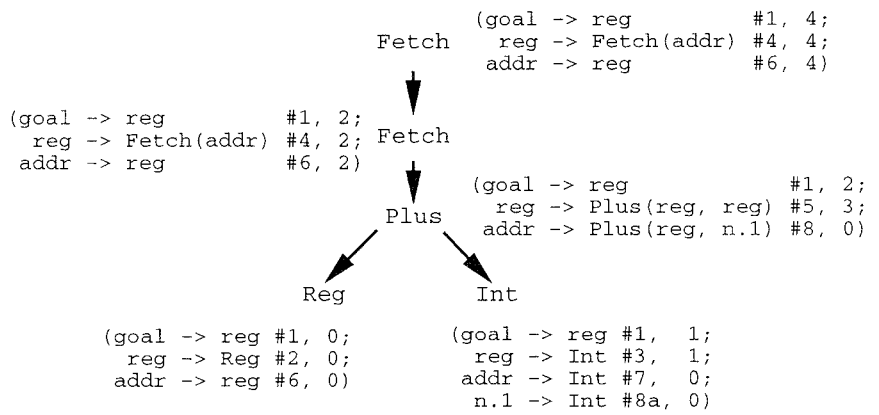

Fig 3 Dynamic programming applied to example tree, each node labeled with "(Rule, Cost) "

that encodes which rule is to be used when that node is to be reduced to a given nonterminal. These states encode the information given explicitly in Figure 3. For example, it is possible to derive the leaf node, `Int`, from all the nonterminals. `Int` can be directly derived from the nonterminals `reg`, `addr`, and `n.1`, by directly applying the rules #3, #7, and #8a, respectively. The costs associated with each derivation is the cost of that particular rule. The derivation from `goal` utilizes the rule, "`goal` → `reg`," that will require that `Int` be subsequently derived from `reg`. Therefore, while the cost associated with rule #1 is 0, the cost of the derivation is 1 — the sum of the costs of complete derivation of `Int` from `goal`.

2.1 BURS Automata

A BURS pattern matcher operates in two passes over the subject tree: the first pass labels the tree nodes with states during a bottom-up tree walk, and the second pass reduces the tree to the goal nonterminal top-down. *Match*() controls this process in Figure 4.

In a BURS pattern matcher, both passes are completely table driven. The labeler, *Label*() in Figure 4, utilizes *Transition*, a state transition table created by the BURS automata generator. *Transition* is indexed by the node's operator and the states of all its children.

Trees are subsequently reduced top-down to a goal nonterminal. Encoded into each state is the rule to apply for each possible nonterminal. (Table *RuleTable* decodes this relationship.) After determining the rule to apply at a given node from the state/nonterminal combination, subsequent reductions are applied recursively down the tree. The shape of the applied rule governs where these recursive applications take place and to which nonterminal those subtrees should be reduced. In Figure 4, *Reduce* gives the simplest possible BURS reducer.

Normally, the reducer would fire some action(s) associated with *Rule*. In a top-down reduction, these actions would precede the loop over descendent children; in a bottom-up reduction, actions follow the loop. The construction of the reducer is irrelevant to the generation of BURS automata.

Fraser et al. [1992b] gives the details of `burg`'s concrete rules and syntax.

```
procedure Match(Root)
  Label(Root)
  Reduce(Root, GoalNonterminal)
end procedure

procedure Label(Node)
  ∀ i ∈ 1..Node.Arity do
    Label(Node.Child[i])
  end ∀
  Node.state= Transition(Node.op, Node.Child[1], ..., Node.Child[Node.Arity])
end procedure

procedure Reduce(Node, Goal)
  Rule = RuleTable(Node.state, Goal)
  ∀ i ∈ the number of nonterminals on RHS of Rule do
    n = ith nonterminal of Rule
    p = node reached from Node corresponding to ith nonterminal of Rule
    Reduce(p, n)
  end ∀
end procedure
```

Fig. 4. Skeletal BURS pattern matcher.

Table II Normal-Form Expansion of "reg → Fetch(Plus(Reg,Reg))"

Original Grammar				Normal-Form			
LHS		RHS	Cost	LHS		RHS	Cost
reg	→	Fetch(Plus(Reg,Reg))	(2)	reg	→	Fetch(n.1)	(2)
				n.1	→	Plus(n.2, n.2)	(0)
				n.2	→	Reg	(0)

2.2 Normal-Form Patterns

To simplify the generation of BURS tables, all patterns are put into the *normal form* introduced in Balachandran et al. [1990]. This form requires that all patterns be of the form "$n \rightarrow m$" where both n and m are nonterminals, or of the form "$n_0 \rightarrow op(n_1, \ldots, n_k)$" where n_i are all nonterminals, $k \geq 0$, and op is an operator. This normal form does not reduce the expressiveness of the grammars—any set of rules not in normal form can be put into normal form by introducing new nonterminals that replace embedded tree patterns. The nested patterns become new nonterminals, and new rules are added that define the new nonterminals in terms of the nested patterns. The rewrite of the original rule maintains the same cost; new rules defining the new nonterminals have a cost of 0. For instance, Table II gives a simple normal-form expansion of "reg → Fetch(Plus(Reg,Reg))":

```
procedure Main()
  States = ∅
  WorkList = ∅
  ComputeLeafStates()
  while WorkList ≠ ∅ do
    state = Pop(WorkList)
    ∀ op ∈ Operators do
      ComputeTransitions(op, state)
    end ∀
  end while
end procedure
```

Fig. 5 BURS automata generation.

3. ALGORITHM TO GENERATE BURS TABLES

The new method of computing the states and state transition tables is an uncomplicated work-list algorithm. *Main*() outlines this algorithm in Figure 5. Initially, the states corresponding to each leaf operator (arity = 0) are computed and are added to the set of known states, *States*, and to the list of states to be processed, *WorkList*. One by one, states are removed from *WorkList* and processed. For each operator with arity greater than 0, the state must be examined to determine what transitions are induced by that state when combined with each of the already processed states. These transitions may create new states to be added to the *WorkList*. Figure 5 gives the pseudocode.

3.1 Data Structures Used to Generate BURS Tables

The set of known states, *States*, is a table that maintains a one-to-one mapping from individual states to nonnegative integers. These integers are used as indices into state transition tables via index maps.

States in a BURS code generator encode three pieces of information at any node in a subject tree: the nonterminals derived from patterns that match a rule at that node, the relative costs of those nonterminals, and which rules generated each nonterminal (at a minimal cost). Such triples are called *items*, and a collection of items describing a particular state is called an *item set*. Item sets are implemented as arrays of { *rule, cost* } pairs that are indexed by a nonterminal. Item sets are, therefore, states. A cost of infinity (∞) indicates that, in this state, no rule derives the given nonterminal. The empty state ($\emptyset$) has all costs equal to infinity.

The relative costs are called *delta costs* and are always normalized so that the nonterminal with the lowest cost derivation has a delta cost of 0. Figure 6 gives the results of dynamic programming on the tree in Figure 3 after the grammar has been put into normal form and the relative costs normalized. Note that the states for the two different `Fetch` nodes are identical—normalization of costs caused this to happen.

Without cost normalization there would be infinitely many states for this grammar. This can be easily seen by imagining a chain of N `Fetch` nodes rooting the tree in Figure 3. The costs representing the state the root node of the tree with N `Fetch` nodes would be $\{N+2, N+2, N+2\}$. Since dynamic programming must

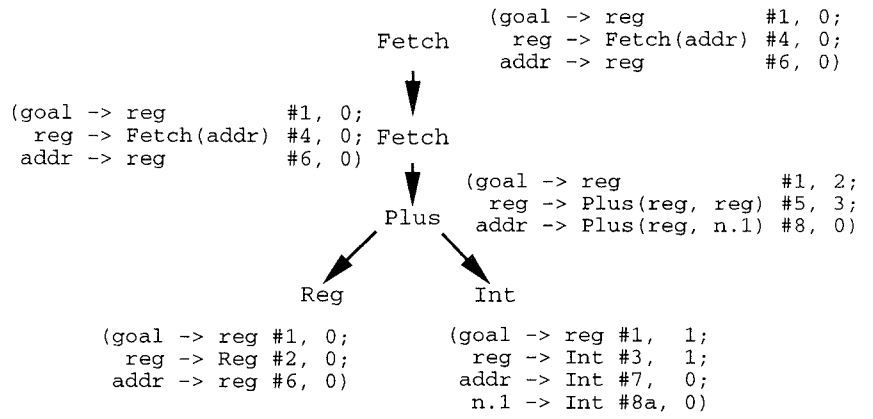

Fig. 6. Dynamic programming with delta costs, each node labeled with "(Rule, Delta Cost)."

```
procedure NormalizeCosts(state)
  delta = min_∀i {state[i].cost}
  ∀ n ∈ Nonterminals do
    state[n].cost = state[n].cost − delta
  end ∀
end procedure
```

Fig 7. Cost normalization.

compute all the states for all possible trees, this sequence of trees would generate an infinite set of states.

Costs within an item set are normalized by the routine *NormalizeCosts*() given in Figure 7.

3.2 Chain Rules

Item sets are computed in a two-step process. *ComputeTransitions*() applies rules of the form "$n \rightarrow op(\ldots)$" to generate nonterminals in the initial item set. Next, the algorithm computes the closure of this set by applying *chain rules*. Chain rules are rules of the form "$n \rightarrow m$" where both n and m are nonterminals. These rules may introduce new nonterminals into an item set, or they may introduce cheaper ways of deriving nonterminals already in the set. Finding the closure of the set is done by iteratively trying all the chain rules and repeatedly applying those that add new or cheaper nonterminals, until no changes are made. *Closure*(), given in Figure 8, implements this procedure. Because all costs are nonnegative, and because a change is made only if a strictly less expensive derivation is found, this process must terminate.

One nonterminal may be derived from another by zero or more chain rule applications. The least-cost derivation is denoted "$n \stackrel{*}{\Rightarrow} m$." A shortest-path algorithm can efficiently compute the cost of such least-cost derivations, "$Cost(n \stackrel{*}{\Rightarrow} m)$."

```
procedure Closure(state)
  repeat
    ∀ r : n → m such that m ∈ Nonterminals do
      cost = r.cost + state[m].cost
      if cost < state[n].cost then
        state[n] = { r, cost }
      end if
    end ∀
  until no changes to state
end procedure
```

Fig 8 Closing a state with chain rules.

```
procedure ComputeLeafStates()
  ∀ leaf ∈ Leaves do
    state = ∅    // state[n].cost = ∞, ∀ n ∈ Nonterminals
    ∀ r : n → leaf do
      if r.cost < state[n].cost then
        state[n] = { r, r.cost }
      end if
    end ∀
    NormalizeCosts(state)
    Closure(state)
    WorkList = Append(WorkList, state)
    States = States ∪ {state}
    leaf.state = state
  end ∀
end procedure
```

Fig 9. *ComputeLeafStates*()

3.3 Computing States and Transitions

The computation of the states and the state transition tables begins by generating a state for each leaf operator (with arity of 0) in the routine *ComputeLeafStates*(). These leaf states must be combined as children of each nonleaf operator, and new states will be created. Each new state is added to the *WorkList* and will be subsequently processed to determine what transitions it induces.

Computing the state to label each leaf is straightforward. Rules with a right-hand side of the given leaf operator generate nonterminals directly into the item set. Normalizing the costs and finding the closure of the item set completes the computation of the state corresponding to the leaf operator. Figure 9 illustrates *ComputeLeafStates*().

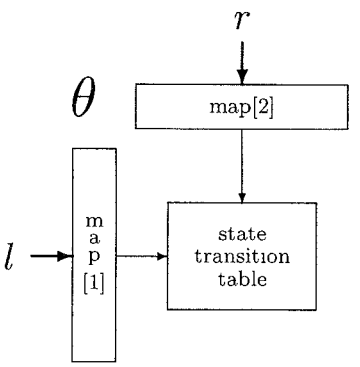

Fig 10 Computing transitions for $\theta(l, r)$ using index maps

For each dimension of a nonleaf operator,[1] an index map of *representer states* is maintained. Representer states are constructed from an item set by retaining only those nonterminals that may contribute to a match in the given dimension for the given operator [Balachandran et al. 1990; Chase 1987]. Suppose that, for a given grammar, there is no rule with a tree pattern for the binary operator, θ, that has a left child of nonterminal n. In this case, the algorithm projects n out of any state when that state is to be examined as a possible left child (in the 1st dimension) of θ. Thus, representer states model the behavior of an equivalence class of states in the restricted context of being the nth child of θ. Chase noted that these equivalence classes produce a much smaller automaton and yield a much faster state generation algorithm.

Project() will retain only those nonterminals in a given state that may be used in determining the transitions that may be induced by that state as a given child of a particular operator. A representer state also discards the *rule* field of each item because that information does not affect transitions (only reductions). After useless nonterminals and all rules are discarded, costs are renormalized. For each dimension, d, a table of representer states, $op.reps[d]$, is maintained that encodes a one-to-one mapping between those states and nonnegative integers. Each dimension's $op.map[d]$ table maintains a mapping from global states to representer states ($op.map[d][s]$ is the representer state to which s maps in the dth dimension of op).

Figure 10 illustrates the relationship between index maps and transition tables. Given the states l and r for the children of binary operator θ, an indirection is used to look up the state transition for the θ node. Figure 11 describes the computation of the relevant nonterminals.

Transition tables are computed based on representer states, not on the original states. This reduces transition table size because many states may map to the same representer state. At tree-matching time the cost of using this technique is the extra level of indirection necessary to compute transitions. In Figure 12, *ComputeTransitions*() finds all the transitions that each new state induces when used in combination with other known states for a given operator.

Each representer state is checked to see if it has already been processed. If the representer state has been previously processed, then no additional work must be

[1]Each operator of arity n has a transition table of n dimensions.

```
function Project(op, i, state)
  pState = ∅
  ∀ n ∈ Nonterminals do
    if ∃ r : m → op(n1,...,ni−1, n, ni+1,...,nop arity) then
      // Nonterminal n may be used in the ith dimension of op.
      pState[n].cost = state[n].cost
    end if
  end ∀
  NormalizeCosts(pState)
  return pState
end procedure
```

Fig 11 *Project*().

```
procedure ComputeTransitions(op, state)
  ∀ i ∈ 1..op.arity do
    pState = Project(op, i, state)
    op.map[i][state] = pState
    if pState ∉ op.reps[i] then
      op reps[i] = op.reps[i] ∪ {pState}
      ∀ (s1, . . , si−1, pState, si+1, ..., sop arity) such that ∀j ≠ i, sj ∈ op.reps[j] do
        — Enumerate all possible combinations of representer states, sk
        result = ∅
        ∀ r : n → op(m1,...,mop arity) do
          — Enumerate all possible rules for op. mk are nonterminals.
          cost = r.cost + pState[mi].cost + Σj≠i sj[mj].cost
          if cost < result[n].cost then
            result[n] = { r, cost }
          end if
        end ∀
        Trim(result)
        NormalizeCosts(result)
        if result ∉ States then
          Closure(result)
          WorkList = Append(WorkList, result)
          States = States ∪ {result}
        end if
        op.transition[s1, ..., si−1, pState, si+1, ..., sop arity] = result
      end ∀
    end if
  end ∀
end procedure
```

Fig 12. *ComputeTransitions*().

Table III

LHS		RHS	Cost
reg	→	Reg	(1)
int	→	Int	(1)
reg	→	int	(1)
reg	→	Plus(reg,int)	(2)
reg	→	Plus(reg,reg)	(2)

done. If the representer state is new, the transition table must be extended along the given dimension for all possible combinations of the representer states of other dimensions (along with this representer state). This is done by generating all such combinations and then searching for all applicable rules. Once these rules have been applied, the delta costs are normalized, and the item set is closed. If the generated state is new, then it is added to *States* and *WorkList*.

The postponement of *Closure*() until after the check for the state's existence in *States* is an optimization justified in Section 5.3. *Trim*(), the routine responsible for reducing the number of states produced, is discussed in Section 3.4.

3.3.1 *Example State Computation.* State computation for the normal-form grammar in Table III illustrates the preceding ideas.[2]

State enumeration begins by computing the states corresponding to the leaf operators in the grammar, Reg and Int, in routine *ComputeLeafStates*(). The states are item sets of { rule, cost } pairs indexed by nonterminals — the nonterminals on the left-hand side of the corresponding rule. For each leaf operator, all rules with that operator on the right are matched, and dynamic programming keeps the least-cost rules for each left-hand nonterminal. After this step, the state for Int is

$$\left\{ \begin{array}{llll} \texttt{int} & \rightarrow & \texttt{Int}, & \text{cost} = (1) \\ \texttt{reg} & \rightarrow & \text{—}, & \text{cost} = (\infty) \end{array} \right\},$$

and the state for Reg is

$$\left\{ \begin{array}{llll} \texttt{int} & \rightarrow & \text{—}, & \text{cost} = (\infty) \\ \texttt{reg} & \rightarrow & \texttt{Reg}, & \text{cost} = (1) \end{array} \right\}.$$

Subsequently, the states must be closed with chain rules. Applying reg → int to Int's state gives

$$\left\{ \begin{array}{llll} \texttt{int} & \rightarrow & \texttt{Int}, & \text{cost} = (1) \\ \texttt{reg} & \rightarrow & \texttt{int}, & \text{cost} = (2) \end{array} \right\}.$$

Closure does not change Reg's state.

After closure, a state's cost is normalized so that the lowest-cost nonterminal is zero. After normalization, Int's state becomes

$$\left\{ \begin{array}{llll} \texttt{int} & \rightarrow & \texttt{Int}, & \text{cost} = (0) \\ \texttt{reg} & \rightarrow & \texttt{int}, & \text{cost} = (1) \end{array} \right\},$$

[2]The costs are slightly contrived to illustrate cost normalization.

and Reg's state becomes

$$\left\{ \begin{array}{llll} \texttt{int} & \rightarrow & \text{—}\ , & \text{cost} = (\infty) \\ \texttt{reg} & \rightarrow & \texttt{Reg}, & \text{cost} = (0) \end{array} \right\}.$$

These are the first two states in the BURS state transition automaton. Call the states for Int and Reg, #1 and #2, respectively.

Computing the states for nonleaf operators is more complicated. For each child of each operator, the algorithm maintains a mapping from states to representer states. This mapping discards unnecessary information with respect to dynamic programming and pattern matching. Representer states discard nonterminals that are not usable for that child, and they discard rule information. After this, costs are renormalized. Plus is this example's only nonleaf operator. Since Plus's first child must be a reg, the representer states for Plus's first child can discard any entry for the int nonterminal. The representer state corresponding to both states #1 and #2 for Plus's first child is

$$\left\{ \begin{array}{llll} \texttt{int} & \rightarrow & \text{—}, & \text{cost} = (\infty) \\ \texttt{reg} & \rightarrow & \text{—}, & \text{cost} = (0) \end{array} \right\}.$$

Call this representer state Plus.1.A (i.e., Plus's first child's first representer state).

Because both int and reg can be used as a second child of Plus, neither can be discarded when computing those representer states. For state #1, the representer state for Plus's second child is

$$\left\{ \begin{array}{llll} \texttt{int} & \rightarrow & \text{—}, & \text{cost} = (0) \\ \texttt{reg} & \rightarrow & \text{—}, & \text{cost} = (1) \end{array} \right\},$$

and the representer for state #2 is

$$\left\{ \begin{array}{llll} \texttt{int} & \rightarrow & \text{—}, & \text{cost} = (\infty) \\ \texttt{reg} & \rightarrow & \text{—}, & \text{cost} = (0) \end{array} \right\}.$$

Call these representer states Plus.2.A and Plus.2.B, respectively.

Combining Plus's representer states produces transitions, and possibly new states. First, all rules with Plus on the right side are examined, and dynamic programming picks the the cheapest match for each nonterminal. Combining Plus.1.A and Plus.2.A gives the following state.

$$\left\{ \begin{array}{llll} \texttt{int} & \rightarrow & \text{—} & , \quad \text{cost} = (\infty) \\ \texttt{reg} & \rightarrow & \texttt{Plus(reg,int)}, & \text{cost} = (2) \end{array} \right\}$$

Note that the rule reg → Plus(reg,reg) also matched, but at a greater cost of 3, so dynamic programming picked the cheaper alternative. Closure does not change that state. Cost normalization yields

$$\left\{ \begin{array}{llll} \texttt{int} & \rightarrow & \text{—} & , \quad \text{cost} = (\infty) \\ \texttt{reg} & \rightarrow & \texttt{Plus(reg,int)}, & \text{cost} = (0) \end{array} \right\}.$$

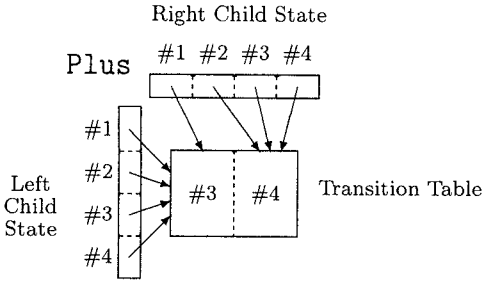

Fig. 13. Transitions for Plus node

This is new state #3.

After closure and cost normalization, combining Plus.1.A and Plus.2.B gives state #4,

$$\left\{ \begin{array}{lcll} \texttt{int} & \rightarrow & \text{—} & , \quad \text{cost} = \quad (\infty) \\ \texttt{reg} & \rightarrow & \texttt{Plus(reg,reg)}, & \text{cost} = \quad (0) \end{array} \right\}$$

Because states #3 and #4 are new, they must be fed back into transition computations. Again, they must be mapped to representer states for Plus. Because both map to Plus.1.A as Plus's first child, and both map to Plus.2.B as Plus's second child, no more work must actually be done — transitions are computed on representer states, and #3 and #4 produce no new representer states.

Thus, this grammar has a transition automaton with four states. All Reg nodes are labeled with state #1, and all Int nodes are labeled with #2. The transitions for Plus nodes are illustrated in Figure 13.

3.4 State Trimming

Many of the states created by the *ComputeTransitions*() are nearly identical. The state generation algorithm will run faster if it can increase the likelihood that two created states will be identical. Two states can often be made identical by trimming *unessential* nonterminals from the item set. A nonterminal is unessential (in a particular state) if it can be proven that it will never be needed to produce a least-cost cover of any subject tree. Henry [1989] devised two ad hoc techniques, "sibling" and "demand" trimming, to identify when one "{ *rule, cost* }" item (representing a nonterminal) can be safely removed from a state because another item subsumes it.

3.4.1 *Triangle Trimming.* Triangle trimming is a new method for safely removing unessential nonterminals from an item set, that generalizes Henry's trimming techniques. Triangle trimming considers all pairs of nonterminals in a particular item set and determines if, given their respective costs, one of the nonterminals can be removed. A nonterminal can be removed if, in all dimensions of all rules where it is applicable, the other nonterminal can be used in a different rule to generate the same resulting nonterminal at no greater cost. Informally, a nonterminal, i, can be removed from an item set if it can be shown that everywhere i can lead to a pattern match, another nonterminal, j, in the item set can also lead to a comparable pattern match at no greater cost.

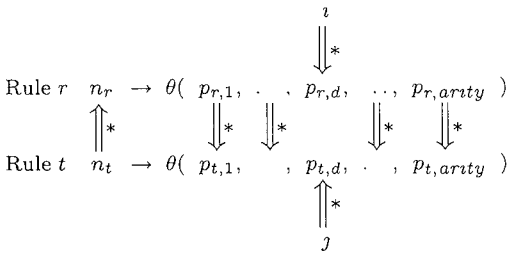

Fig. 14 Triangle-trimming relationship (for j to subsume i)

$$
\begin{aligned}
state[i].cost \;+\; r.cost \;+\; Cost(p_{r,d} \stackrel{*}{\Rightarrow} i) \;&\geq \\
state[j].cost \;+\; t.cost \;+\; Cost(p_{t,d} \stackrel{*}{\Rightarrow} j) \;&+\; Cost(n_r \stackrel{*}{\Rightarrow} n_t) \\
&+\; \textstyle\sum_{k \neq d} Cost(p_{t,k} \stackrel{*}{\Rightarrow} p_{r,k})
\end{aligned}
$$

Fig. 15. Inequality that must hold for i to be removed if j is present.

Determining if j subsumes i requires comparisons that have a *triangular* shape (see Figure 14). For a given operator, θ, and in a given dimension, d, two rules must be found such that both rules represent patterns for θ; and one rule, r, can employ i as its dth child, and the other rule, t, can employ j as its dth child. (It is not necessary that these rules use i and j directly—they may use nonterminals that are derived from i and j via chain rules.)

Since rule r reduces to nonterminal n_r, it must be shown that t can also produce n_r at no greater cost. By assuming that rule r has matched, it can be determined if rule t can also match. Rule t can also match if its children in dimensions other than d can be derived via chain rules from the corresponding children of rule r. (Since the initial assumption is only that r matches, determining if $p_{t,k}$ exists for a match of rule t depends on whether $p_{r,k}$ derives $p_{t,k}$ via chain rules.)

Figure 14 shows how i and j, and the rules r and t, must relate for j to subsume i. Once rule r is found to use i to derive n_r, a rule must be found that can employ j and derive n_r. Notice that for any rule r that employs i, it is only necessary to find one such rule t employing j for j to subsume i.

Subsumption is based not only on feasibility, but also on costs. A nonterminal cannot be removed if its removal would force more-expensive reductions to be found than had it been retained. For the pair of rules, r and t, in Figure 14, it is possible to remove i from the item set containing j if the inequality in Figure 15 holds. The cost of using r is the sum of the cost of i, the cost of deriving $p_{r,d}$ from i, and the cost of r. Since the premise is only that rule r matches and that i and j are present in some item set, the computation of the cost of using t with j to produce n_r indirectly will require not only the costs of t, j, and $p_{t,d} \stackrel{*}{\Rightarrow} j$, but will also require the costs of deriving the other $p_{t,k}$ from $p_{r,k}$ and the cost of deriving n_r from n_t.

The inequality in Figure 15 is the basis for finding the *minimal* cost difference between two nonterminals to allow one of them to be removed for a given rule. In general, to remove i safely, it is necessary to examine *all* contexts in which i can

```
// Compute C, such that if state[i].cost ≥ state[j].cost + C
// then i can safely be removed from state.
function Triangle(i, j)
  if i = Goal then
    return ∞        // Do not remove the goal nonterminal.
  end if
  Max = −∞
  ∀ n ∈ Nonterminals−{i} do
    if Max < Cost(n ⇒* j) − Cost(n ⇒* i) then
      Max = Cost(n ⇒* j) − Cost(n ⇒* i)
    end if
  end ∀
  ∀ op ∈ Operators do
    ∀ d ∈ 1..op.arity do
      ∀ r : n_r → op(p_{r,1},...,p_{r,op.arity}) do
        C_i = Cost(p_{r,d} ⇒* i)
        if C_i < ∞ then
          LocalMin = ∞
          ∀ t : n_t → op(p_{t,1},...,p_{t,op.arity}) do
            C_{r,t} = Cost(n_r ⇒* n_t)
            C_j = Cost(p_{t,d} ⇒* j)
            C_k = Σ_{k≠d} Cost(p_{t,k} ⇒* p_{r,k})
            C = C_{r,t} + C_j + C_k + t.cost − r.cost − C_i
            if C < LocalMin then
              LocalMin = C
            end if
          end ∀
          if LocalMin > Max then
            Max = LocalMin
          end if
        end if
      end ∀
    end ∀
  end ∀
  return Max
end procedure
```

Fig. 16 *Triangle*().

be used and find the cost difference that is sufficient to guarantee that i can be removed based on the relative costs of i and j. In Figure 16, *Triangle*(), calculates this minimal difference for any pair of nonterminals. (When it is impossible for nonterminal j to be used in place of i, regardless of their respective costs, *Triangle*() returns ∞.)

3.4.2 *Triangle-Trimming Example.* The simple grammar in Table IV illustrates triangle trimming. The example will demonstrate the necessary relative costs of nonterminals X and Z to remove X from a state.

Table IV

LHS		RHS	Cost
A	$\rightarrow$	$\theta(X, Q)$	(4)
A	$\rightarrow$	B	(1)
B	$\rightarrow$	$\theta(Y, R)$	(1)
Y	$\rightarrow$	Z	(1)
R	$\rightarrow$	Q	(1)

Table V. Two Derivations for Triangle Trimming

Derivation			Total Cost	Rewriting Step
A	$\Longrightarrow$	$\theta(X, Q)$	$4+Cost(X)+Cost(Q)$	Apply $A \rightarrow \theta(X, Q)$ directly
A	$\Longrightarrow$	B	$1+Cost(B)$	Apply $A \rightarrow B$ directly
	$\Longrightarrow$	$\theta(Y, R)$	$2+Cost(Y)+Cost(R)$	Apply $B \rightarrow \theta(Y, R)$.
	$\Longrightarrow$	$\theta(Z, R)$	$3+Cost(Z)+Cost(R)$	Apply $Y \rightarrow Z$.
	$\Longrightarrow$	$\theta(Z, Q)$	$4+Cost(Z)+Cost(Q)$	Apply $R \rightarrow Q$.

```
procedure Trim(state)
  ∀ n ∈ state do
    ∀ m ∈ state (m≠ n) do
      C = Cost(n ⇒* m)
      if state[n].cost ≥ state[m].cost + C then
        state[n] = { ⊥, ∞ } // Remove n from state.
      end if
    end ∀
  end ∀
  ∀ n ∈ state do
    ∀ m ∈ state (m≠ n) do
      C = Triangle(n, m)
      if state[n].cost ≥ state[m].cost + C then
        state[n] = { ⊥, ∞ } // Remove n from state.
      end if
    end ∀
  end ∀
end procedure
```

Fig. 17 *Trim*()

Note that both rules for θ can be used to reduce to A. Consider the two derivations in Table V. Triangle-trimming notes that these two reductions to A relate the costs of X and Z in any state that represents the left child of a θ node. If the $Cost(X)$ exceeds (or equals) the $Cost(Z)$ in such a state, X is not necessary to find a minimum cost reduction to A because the alternative path (using the second θ rule) can always be employed. Furthermore, if in all contexts (i.e., for all children of every operator) X is unnecessary if $Cost(X) \geq Cost(Z)$, then we can eliminate X from all states in which that condition holds. Because many of these states may have differed only in X — either from differing costs or deriving rules — eliminating

X will cause them to become a single state.

3.4.3 *Chain Rule Trimming.* Two states are identical if they represent the same nonterminals at the same costs with each respective nonterminal generated by the same rule. Triangle-trimming removes nonterminals from states whenever possible, thereby eliminating the possibility that two states differ on the particular costs or rules involving those nonterminals. To minimize the number of states further, it is necessary to bias the algorithm toward using the same rules whenever possible. Biasing the algorithm toward using chain rules whenever possible increases the likelihood that two states will have used the same rules to derive a given nonterminal. This bias can be forced by removing nonterminal entries from an item set prior to closure when it can be determined that *Closure*() will restore those nonterminals at an equal or lesser cost using chain rules.

In Figure 17, *Trim*(), uses both triangle and chain rule trimming to prune nonterminals from item sets so that they will be more likely to be identical, thereby reducing the size of the generated tables and the table generation time.

3.4.4 *Fully General Trimming.* Nonterminal trimming does not need to be constrained to looking at pairs of nonterminals as it was in triangle and chain rule trimming. While it may not be the case that a single nonterminal in an item set subsumes any other, it may be the case that some *set* of nonterminals subsumes another.

One can ask the simple question, "Given *all* the nonterminals, n_i, in state N, can one safely remove nonterminal n_k?" This could be answered by attempting, one by one, to remove nonterminals from the item set and determining by analysis similar to triangle trimming if that removal would force more-costly matches to be found. If not, the nonterminal can be safely removed.

It is not known how much, if any, general trimming would reduce the number of states. The general approach to state trimming was not attempted because it is significantly more expensive than triangle trimming. Because triangle trimming tests pairs of nonterminals, the relative costs necessary for subsumption can be cached for reuse easily (see Section 5.2). There is no simple relationship based on a set of nonterminals that can be so easily stored and accessed.

4. DIVERGING GRAMMARS

Because all dynamic programming is done at compile-compile time, it is necessary to anticipate *all* possible trees and generate states that can label the nodes of those trees. To do this, there must be only a finite number of states. Grammars that do not produce a finite number of states are said to *diverge* [Pelegri-Llopart 1988].

A grammar diverges when it is possible for the derivation costs of a pair of nonterminals in the same state to become arbitrarily distant. To prevent the BURS table generation algorithm from attempting to enumerate an infinite set of states for diverging grammars, a simple threshold test is used. A test is inserted into the normalization procedure (*NormalizeCosts*()) to determine the greatest cost differential between nonterminals in any given state. If that differential is above the threshold value, the grammar is rejected as "probably diverging."

Fortunately, code generation grammars do not typically diverge. This is because the nonterminals usually describe data values (*e.g.*, registers, data, address-

ing modes) that can be interchanged at a bounded cost. For instance, it is unlikely that the cost of computing a value into memory could be arbitrarily more expensive than computing a value into a register since there almost certainly is a store instruction of fixed cost.

5 SPEED-OPTIMIZING TECHNIQUES

The previous routines provide many opportunities for speed optimization. Some of the improvements are general techniques not specific to BURS table generation; other improvements rely on subtle knowledge of BURS table generation.

5.1 Attempt Cheaper Alternatives First

It may appear that the two sets of nested loops in *Trim*() could be jammed into a single pair of nested loops for improved efficiency. Both loops have the intended side-effect of removing nonterminals from the states. Since the loops iterate over only the nonterminals that remain in the state, the second set of loops will normally iterate fewer times than the first set. Because triangle trimming is an expensive operation relative to chain rule trimming, it is more efficient to remove all possible nonterminals via chain rule trimming and then attempt triangle trimming only on the remaining nonterminals.

5.2 Precomputing and Caching Values

In the previous routines, many situations exist where values can be computed once and used many times. For instance, *Project*() requires the knowledge of which nonterminals can appear in the ith dimension of operator op. Because this list is invariant for a given rule set, it can be computed once and used repeatedly. Efficiency is also enhanced if the list of rules is partitioned by the operator of the pattern, so that *ComputeTransitions*() will only iterate over the list of applicable rules. The cost of transitive closure rules ($Cost(n \stackrel{*}{\Rightarrow} m)$) is precomputed advantageously since it is used often by *Trim*() and *Triangle*().

There are $O(N^2)$ possible pairs of nonterminals that may be used in a call to *Triangle*(), but in practice only very few pairs are ever used. Originally, *Triangle*() was implemented via a precomputed table. This precomputation consumed over 75% of the execution time generating tables for a VAX grammar, and yet fewer than 4% of the values were ever accessed. Changing the program to compute (and cache) those values by need increased the speed tremendously

5.3 Defer Closure

If two item sets are equal before closure, then they must be equal after closure. Because two item sets are chain rule trimmed before closure, it is also the case that if two item sets are equal after closure, they must have been equal before closure. By maintaining both preclosure and postclosure copies of an item set in a table, the algorithm can check for the existence of an item set in the table by comparing their preclosure representations. This allows the closure computation to be deferred until it is known that the state is indeed new and must be added to the table.

5.4 Item Set Equivalence

Determining whether an item set is already in a table of states is an expensive operation, and this test is done for every entry in every transition table. The integer subset 68000 grammar required over 425,000 calls to determine item set equivalence. For two item sets to be equal, they must be equal for all of their items. Fortunately, two observations make testing for equivalence much more efficient: two item sets created as members of transition tables for different operators can never be equal, and for any given operator it is only necessary to compare the entries corresponding to the left-hand sides of the rules for that operator.

The same routines are used to implement the global *States* table and each of the local *op.reps*[] tables. These tables are implemented as hash tables. Computing the hash function is also made more efficient by examining only the relevant nonterminals. Calling *NormalizeCosts*() after *Trim*(), but before *Closure*(), allows it to limit the nonterminals it must inspect. Again, the same nonterminals that are relevant to determining item set equivalence are those that must be normalized prior to a call to *Closure*().

5.5 Specialize Memory Allocation

`burg` allocates and deallocates an enormous amount of memory to compute item sets and transition tables. The primary source of allocation and deallocation of memory is the tentative allocation of item sets by *ComputeTransitions*() and *Project*(). Only after an item set is allocated and computed can it be determined if an equivalent state has already been seen, thereby allowing the deallocation of the item set. Redundant item sets really *must* be deallocated—for a 68000 grammar the program computed over 100,000 redundant item sets.

Fortunately, fixed allocation/deallocation patterns of particular data can lead to very efficient memory management [Hanson 1990]. Item sets, after allocation, are computed and then either retained forever or immediately released. Therefore, two item set deallocations can never occur sequentially without an intervening allocation. This allows the creation of specialized deallocation and allocation routines for item sets. The deallocation routine simply maintains a reference to the last discarded item set and does not return the space to the heap. Allocation checks this reference, and if the reference is not null, it returns the reference to the previously deallocated value (and clears the reference); only if the reference is null does the allocator request space from the heap.

5.6 Minimize Space

The single biggest user of memory is the item set representation. Item sets must be kept as small as possible to avoid overconsumption of RAM by minimizing the number of nonterminals in the normal-form grammar. A naive translation of a grammar into normal form may produce too many nonterminals if it creates different nonterminals that represent identical patterns. It is important (and easy) to reuse previously created nonterminals.

5.7 Unprofitable Optimization

When an item set is normalized, the relative costs of all the nonterminals are retained. This is unduly conservative because certain nonterminals can never be

Table VI. Code Size for burg

Function	Lines (C/Yacc)
Table Generation	1981
Front End	633
Table Output	1345
Total	3959

used in the same context, and could, therefore, be independently normalized within the same item set. It is possible to partition the set of nonterminals based on whether they can be used in the same context (i.e., in the ith dimension of a given operator or as part of the same chain rule). Once the nonterminals are partitioned, each partition can be independently normalized. The hope was that this would cause more identical states to be found because differences between elements of different partitions would now be irrelevant. Unfortunately, only the VAX grammar showed any reduction in the number of states—two states were eliminated from over 500.

6. OUTPUT

The table generator must output two sets of data: the state transition tables for labeling the subject tree and a mapping from (states×nonterminals) to rules for reducing the matched tree and emitting code.

For the transition tables, it is necessary to output both the n-dimensional transition tables (*op.transition*) and the mappings from states to representer states for each dimension (*op.reps*[d]) since the transition tables are indexed by representer states. For leaf nodes, it is only necessary to give the mapping from the node to its unique state (*leaf.state*).

The reduction mapping is a table of all the states (*States*) and the *rule* fields that correspond to each nonterminal. These fields indicate which rule produces the given nonterminal. There is no need for the *cost* field at compile-time.

7. IMPLEMENTATION RESULTS

The new algorithm has been implemented in a system called `burg` [Fraser et al. 1992b]. The input has two parts: a description of the operators (including the arity and identifying value of each) and a list of grammar rules. The operators are limited to being nullary (leaf), unary, or binary. (The arity was limited because the intended application required only nullary, unary, and binary operators.) Each rule includes an arbitrarily complex pattern, the nonterminal the pattern derives, its cost, and a unique external rule number (for identification). The front end of the table generator puts the rules into normal form.

As output, the program creates C routines and tables for labeling and reducing a subject tree. The program can output either a simple table-driven tree labeler and reducer or a hard-coded labeler and reducer. The hard-coded routines incorporate the time- and space-saving techniques in Fraser and Henry [1991].

The entire program is fewer than 4000 lines of code that splits evenly between table generation routines and input/output routines. Table VI gives the number of lines of code used to implement the table generator.

`burg` runs quickly on both simple and complex inputs. `burg` is compared to

Table VII. Timings

Grammar		Time (sec)		Ratio
Machine	Rules	Henry's	burg	
vax	291	467 7	14.4	32
MIPS	136	21.4	0.6	36
vax.bwl	524	146.8	15.5	9
mot.bwl	462	251.5	14.4	14

Table VIII. Number of States

Grammar		States			Ratio
Machine	Rules	Henry's	burg	burg w/ Trim	(Trim/No Trim)
vax	291	1017	1017	1015	0.998
MIPS	138	125	125	125	1.000
vax.bwl	524	493	946	610	0.645
mot.bwl	462	499	1295	835	0.645

Table IX. Size of Matchers

Grammar	Data Size (bytes)		Ratio
Machine	burg	burg w/ Trim	(Trim/No Trim)
vax	46160	41200	0.893
MIPS	2336	2336	1.000
vax.bwl	57,008	41,040	0.720
mot.bwl	133,968	64,736	0.483

Henry's [1989] table generator, which was derived from the CODEGEN system. Table VII gives a description of four sample input grammars and the execution times for each system on each grammar. The first two grammars (used to generate code generators for `lcc` [Fraser and Hanson 1995]) are for the VAX and the MIPS R3000 RISC processor. Two others that were developed as part of the CODEGEN project are integer (byte, word, and long) subsets of the VAX and Motorola 68000 processors. The number of rules is for the normalized grammar. The timings were taken on a DECstation 5000 with 96MB of RAM—the timings are *more* favorable toward `burg` on machines with limited amounts of RAM.

Table VIII indicates the number of states generated for each grammar, with and without triangle trimming. The differences in the number of generated states between the `burg` and Henry's system for the CODEGEN grammars can be attributed to the presence of a state minimization postpass in Henry's system that is not present in `burg`. State minimization for BURS is similar to DFA state minimization. It is possible to eliminate states after they and the transition tables have been generated by isolating and removing states that differ only in the respective costs of each constituent nonterminal.[3] The space savings did not seem worth the additional complexity or time, and therefore, `burg` does not have a state minimization pass.

Table IX indicates the effectiveness of triangle trimming at reducing the size of

[3]Because state minimization is a postpass, it cannot make the program faster—it must make it slower. Henry [1989] found that the additional time for the postpass was negligible (< 1%) in his system.

Table X. Inference Rules for Pascal's "+"

Rule #	LHS		Pattern	Cost
1	integer	→	ADD(integer, integer)	(0)
2.	real	→	ADD(real, real)	(0)
3	set	→	ADD(set, set)	(0)
4.	real	→	integer	(1)

Table XI Type Rules for `lcc`'s Intermediate Representation

Rule #	LHS		Pattern	Cost
1.	int	→	ADDI(int, int)	(0)
2.	double	→	ADDD(double, double)	(0)
3.	double	→	CVID(int)	(0)

the tables for encoding the BURS pattern matchers.

8. OTHER APPLICATIONS OF BURS

BURS technology has applications outside of instruction selection. For instance, BURS can be used to do simple type inferencing, data structure auditing, and tree simplification.

8.1 Simple Type Inferencing

Waite has used `burg` to automate simple type inferencing (personal communication. W. Waite, 1991). In many Algol-like languages, arithmetic operators are overloaded and may operate on different types. For example, in Pascal, "+" may operate on sets, reals, or integers—but both operands must be the same type. To add a real and an integer, the compiler must realize that the integer must be converted to a real before the addition. However, the compiler must not convert two integer operands to reals.

BURS simplifies the process of determining which instance of the overloaded operator must be used and what conversions (if any) must be performed. The BURS grammar for this simple inference system is given in Table X. Given this grammar, a BURS tree matcher will try to find a least-cost match of the tree. The patterns chosen will indicate which type of "addition" must be used. If the rule "real → integer" is used, then an integer must be converted to a real at that location. Because the conversion rule has a cost greater than zero, it will not be used unless necessary to find a legal parse of the tree.

8.2 Data Structure Auditing

BURS pattern matchers can also be used to determine if a tree has *any* legal parses. If the underlying grammar defines all *legal* tree structures, this can be used to audit trees quickly to ensure that they are not malformed.

In an experiment, a BURS pattern matcher audited the intermediate representation generated by the front end of `lcc` [Fraser and Hanson 1995], an ANSI C compiler. The IR trees were tested to see if they were correctly formed with respect to basic types. A small portion of the grammar is given in Table XI. If the matcher finds a parse, then the expression tree is legal; otherwise it is malformed.

After running the experiment on about 10,000 lines of C, a few trees were found

Table XII. Simplification Rules for lcc's IR

Rule #	LHS		Pattern	Cost
1.	int	→	ADDI(int, int)	(1)
2.	int	→	NEGI(int)	(1)
3.	int	→	ADDI(int, NEGI(int))	(0)
4.	int	→	ADDI(NEGI(int), int)	(0)

that did not parse. The problem was not in lcc; it was in the IR documentation. The documentation did not fully describe all the legal combinations of intermediate operators and types. The experiment was intended to search for bugs in the implementation and succeeded in finding omissions in the documentation.

8.3 Tree Simplification

Tree pattern matching can also be used to find opportunities for tree simplification in a compiler. Patterns can be used to represent opportunities for tree modifications that will result in simpler or more efficient code.

A simple modification would be to substitute subtraction for the addition of a negated integer. The example in Table XII gives rules for parsing the ADDI and NEGI operators. When a BURS pattern matcher finds a parse for an expression tree, it will choose to use rules #3 and #4 whenever possible since they have a lower cost than the composition of rules #1 and #2. A subsequent simplification pass would isolate these rules and perform the necessary tree modifications. Since the matchers are automatically generated, it is a simple matter to build incrementally the patterns that lead to simplifications.

9. RELATED WORK

9.1 BURS Technology

Bottom-up tree pattern matching was developed by Hoffmann and O'Donnell [1982]. Bottom-up pattern matching is theoretically optimal—relying on a single bottom-up tree walk with a simple table lookup at each node in a tree to do the matching. BURS technology relies on this technology for much of its speed.

Naively generating BURS states and state transition tables fails because the tables become too large. (The same is also true for simple Hoffman-O'Donnell bottom-up matchers *without* dynamic programming.) A typical CISC machine description will generate over 1000 states.[4] Directly encoding the transition table for a *single* binary operator would, therefore, require over 1,000,000 entries.

Fortunately, many of the rows (and columns) of bottom-up pattern-matching transition tables are identical. To exploit this redundancy, *index maps* can be used to encode much smaller tables. Index maps are vectors that map states of the automaton to *representer states* for indexing a transition table. States may share a given row or column of a transition table through a single indirection. Chase [1987] demonstrated that these maps can be produced on-the-fly during table generation thus avoiding superfluous work.

Pelegri-Llopart, the originator of BURS theory [Pelegri-Llopart 1988; Pelegri-Llopart and Graham 1988], incorporated Chase's ideas into a system that added

[4]The integer *subset* of a Motorola 68000 grammar has over 800 states (Table VII).

cost information for dynamic programming at table generation time. In addition to recognizing that dynamic programming could be done prior to compile-time, he developed the theoretical foundation for showing that the process is theoretically feasible for typical machine grammars. Pelegri-Llopart's technique is not limited to finding least-cost parses of an input tree; his BURS theory also incorporated tree rewrites. A specification could include grammar rules that allowed a matched tree to be rewritten for subsequent matches. This allowed the specification of commutativity transformations, for instance.

Subsequent BURS systems, including the techniques described here, do not allow general rewrites, but instead defer that responsibility to another phase of the compilation process. Balachandran, Dhamdhere, and Biswas [Balachandran et al 1990] simplified Pelegri's model by disallowing rewrite rules, and they generalized Chase's ideas to use cost information.

Henry's BUILD/CODEGEN system could build many different tree pattern-matching systems for comparison of system build times and system run-times [Henry and Damron 1989]. The system compared naive, Graham-Glanville, top-down and bottom-up tree matchers with dynamic programming done both at build-time and run-time. It also compared these with greedy-match disambiguation. The performance comparison provides an enormous amount of information with respect to the various tradeoffs.

Henry [1989] developed optimization techniques to limit the number of BURS states produced during table generation. His system was derived from his earlier CODEGEN/BUILD systems [Henry and Damron 1989]. With fewer states, a smaller automaton is produced more quickly. Henry's techniques are much more aggressive than Chase's simple index map techniques, but at the cost of increased complexity. Henry [1989] states "The table builder uses space and time voraciously, even though it uses very complex algorithms designed to minimize these resources." The algorithm described in this article generalizes and simplifies his work. `burg` and Henry's system share the same specification language and therefore can be directly compared on a variety of machine specifications. `burg` routinely shows a factor of 10 to 30 improvement in generation speed. However, Henry's system undoubtedly suffers from its CODEGEN/BUILD roots as a more general research tool for comparing different code generation techniques.

To decrease the size of the generated tables further, both Henry and Pelegri-Llopart incorporate an additional postpass to minimize the number of states. Such a pass is essentially a DFA state minimization pass that reduces the number of states by finding states that do not differ with respect to the matches they encode or the transitions they induce. They only differ with respect to the costs that they encode—information that is not needed at compile-time.

9.2 Non-BURS Pattern-Matching Systems

Other code generation systems based on tree pattern matching and dynamic programming have been developed. They differ primarily in what technology they use to do tree pattern matching and in the fact that they do dynamic programming at compile-time rather than compile-compile time.

Aho, Ganapathi, and Tjiang [Aho et al. 1989] created a tree manipulation language and system called *Twig*. Given a specification of tree patterns and associated

costs, Twig generates a tree automaton that will find the least-cost cover of a subject tree. Twig uses fast top-down Hoffmann-O'Donnell [Hoffmann and O'Donnell 1982] pattern matching in parallel with dynamic programming to find the least-cost cover in $O(patno \times |tree|)$ time (where *patno* is the number of patterns in the grammar, and $|tree|$ is the size of the tree to be parsed).

The costs associated with patterns in Twig are more general than those afforded by any BURS system. Twig may compute the cost of a pattern dynamically—depending on semantic information available at compile-time. This flexibility further allows Twig to *abort* certain matches if semantic predicates are not satisfied. Thus, the applicability of Twig's patterns is *context sensitive*. BURS cannot have this flexibility since all costs must be compile-compile time constants to precompute dynamic programming decisions.

A code-generator generator based on tree pattern matching was developed by Emmelmann et al. [1989]. The Back End Generator (BEG) uses naive pattern matching to find pattern matches within the tree IR to do instruction selection. The least-cost cover of the tree is found using dynamic programming techniques that are essentially identical to Twig's. Like Twig, BEG can guard patterns with semantic predicates.

A BEG specification, in addition to having instruction patterns, includes a description of the register set of the target machine. This specification is used to generate the register allocator. Two different types of register allocators may be generated: a simple on-the-fly allocator and a more complex postpass allocator that processes the cover tree prior to emitting instructions. They found the code quality and code generation times to be comparable to their handwritten CGs.

Both Twig and BEG have the advantage over BURS of being able to incorporate semantic information into pattern matching and dynamic programming. However, they generate pattern matchers that are significantly slower than pattern matchers based on BURS technology. This is because (1) they use slower pattern-matching technology (either top-down or naive) *and* (2) they do dynamic programming at compile-time.

Fraser, Hanson, and Proebsting developed a code-generator generator based on naive pattern matching and dynamic programming [Fraser et al. 1992a]. This system, `iburg`, maintains the same interface as `burg` and can therefore be directly compared. Although engineered for efficiency, the resulting matchers were still 6-12 times slower than `burg`'s.

10. CONCLUSION

The BURS table generation algorithm presented is a simple and efficient method of producing BURS tables. Based on all available information, `burg` is significantly faster than any other BURS system. The prototype implementation required fewer than 2000 lines of C code for producing the BURS automata. It was able produce these tables over 30 times more quickly than the previous "state-of-the-art" optimizing system. `burg` does not sacrifice table compaction optimizations to achieve this speed—to the contrary, the compaction techniques increase the overall speed of the implementation by reducing the number of states that must be examined.

The algorithm employs only simple data structures and routines to generate these tables quickly. To a large degree, this design simplicity increases efficiency.

To increase speed further, optimizations that exploit the specific nature of BURS table generation were isolated and are described here.

To reduce the number of created states a new technique of trimming states, triangle trimming, has been developed to isolate nonterminals that can be removed from a state. This trimming provides a many-fold reduction in the number of states and a commensurate speedup in table generation.

ACKNOWLEDGMENTS

Robert Henry made his system available for comparison and explained much of his early work in BURS table generation. Chris Fraser tested `burg` and provided valuable implementation advice. Charles Fischer provided helpful comments on early drafts of this article.

REFERENCES

AHO, A. V., GANAPATHI, M., AND TJIANG, S. W. K. 1989. Code generation using tree matching and dynamic programming *ACM Trans. Program. Lang. Syst. 11,* 4 (Oct.), 491–516

BALACHANDRAN, A., DHAMDHERE, D. M., AND BISWAS, S. 1990. Efficient retargetable code generation using bottom-up tree pattern matching. *Comput. Lang. 15,* 3, 127–140

CHASE, D. R. 1987. An improvement to bottom-up tree pattern matching. In *Proceedings of the 14th Annual Symposium on Principles of Programming Languages.* ACM, New York, 168–177

EMMELMANN, H., SCHROER, F.-W., AND LANDWEHR, R. 1989. BEG—a generator for efficient back ends In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation.* ACM, New York, 227–237

FRASER, C. W. AND HANSON, D. R. 1995 *A Retargetable C Compiler: Design and Implementation* Benjamin/Cummings, Redwood City, Calif

FRASER, C. W. AND HENRY, R. R. 1991. Hard-coding bottom-up code generation tables to save time and space. *Softw. Pract. Exper. 21,* 1 (Jan.), 1–12.

FRASER, C. W., HANSON, D. R., AND PROEBSTING, T. A. 1992a. Engineering a simple, efficient code-generator generator. *ACM Lett. Program. Lang. Syst. 1,* 3 (Sept.), 213–226

FRASER, C. W., HENRY, R. R., AND PROEBSTING, T. A. 1992b. BURG — fast optimal instruction selection and tree parsing. *SIGPLAN Not. 27,* 4 (Apr.), 68–76.

HANSON, D. R. 1990. Fast allocation and deallocation of memory based on object lifetimes. *Softw. Pract. Exper. 20,* 1 (Jan.), 5–12.

HENRY, R. R. 1989. Encoding optimal pattern selection in a table-driven bottom-up tree-pattern matcher. Tech. Rep. 89-02-04, Univ. of Washington, Seattle, Wash.

HENRY, R. R. AND DAMRON, P. C. 1989. Performance of table-driven code generators using tree-pattern matching. Tech. Rep. 89-02-02, Univ. of Washington, Seattle, Wash

HOFFMANN, C. M. AND O'DONNELL, M. J. 1982 Pattern matching in trees. *J. ACM 29,* 1 (Jan.), 68–95.

PELEGRI-LLOPART, E. 1988. Rewrite systems, pattern matching, and code generation. Ph. D. thesis, Computer Science Division, Univ. of California, Berkeley

PELEGRI-LLOPART, E. AND GRAHAM, S. L. 1988. Optimal code generation for expression trees: An application of BURS theory. In *Proceedings of the 15th Annual Symposium on Principles of Programming Languages* ACM, New York, 294–308.

PROEBSTING, T. A. 1992. Simple and efficient BURS table generation. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation.* ACM, New York, 331–340.

Received December 1993; revised February 1995; accepted April 1995