Comput. Lang. Vol. 15, No. 3, pp. 127–140, 1990
Printed in Great Britain. All rights reserved

0096-0551/90 $3.00 + 0.00
Copyright © 1990 Pergamon Press plc

EFFICIENT RETARGETABLE CODE GENERATION USING BOTTOM-UP TREE PATTERN MATCHING

A. BALACHANDRAN, D. M. DHAMDHERE† and S. BISWAS
Indian Institute of Technology, Bombay-400076, India

(*Received* 3 *July* 1989; *received for publication* 4 *January* 1990)

Abstract—Instruction selection is the primary task in automatic code generation. This paper proposes a practical system for performing optimal instruction selection based on tree pattern matching for expression trees. A significant feature of the system is its ability to perform code generation without requiring cost analysis at code generation time. The target machine instructions are specified as attributed production rules in a regular tree grammar augmented with cost information in Graham–Glanville style. Instruction selection is modelled as a process of determining minimum cost derivation for a given expression tree.

A matching automaton is used for instruction selection. Cost information is encoded into the states of this automaton so that cost analysis is not required at code generation time. The folding technique of table compression is extended to this automaton and two schemes of table compression based on cost information are proposed.

Compilers Retargetable code generation Code-generator Code-generator-generator Tree-pattern matching Instruction selection Table compression

1. INTRODUCTION

In the last decade, substantial efforts have been made to automate object code generation [1, 2]. In most schemes, the target machine instructions are represented as tree patterns and the intermediate representation (IR) is a sequence of expression trees created by the compiler front end. The code generator generator preprocesses the tree patterns and builds tables for a matching automaton. The matching automaton with supporting routines such as register allocation forms the code generator. It takes one expression tree at a time from IR and generates code using pattern matching and replacement scheme.

The PQCC system [3] automatically derives the patterns from the ISP like machine descriptions [4] and uses a greedy heuristic for pattern matching. In Graham–Glanville technique, [5], tree pattern matching is reduced to string pattern matching by linearising the expression and pattern trees and LR-parsing is used for string pattern matching. Since the grammar used for specifying machine instructions is highly ambiguous, shift–reduce and reduce–reduce conflicts may arise in LR-parsing. In case of ambiguity, longest pattern or the first pattern in the order of specification is selected. The main advantages of this method are: specifications are easy to write, the operation of the parser can be proved to be correct, and well understood algorithms exist for constructing the tables. However, instruction selection is not optimal and code generator may block if the conflicts are not resolved properly.

Aho *et al.* [6] associate costs with the patterns and use a top-down tree pattern matching algorithm coupled with the dynamic programming algorithm of Aho and Johnson [7] to select the optimal sequence of patterns. In addition to optimal instruction selection, the dynamic programming algorithm allows highly ambiguous grammars. Since the ambiguities are resolved using a cost measure during code generation, user need not order the patterns or worry about factoring. Hence it allows the user to specify concise and elegant description of the target machine instructions. However the algorithm requires expensive cost analysis to be performed during code generation. Hatcher and Christopher [8] have used a technique based on bottom-up tree pattern matching algorithm to perform optimal instruction selection without requiring cost analysis during code generation. However, their technique requires modifying some part of machine description to retain optimality.

†Author to whom correspondence should be addressed.

Pelegri-Llopart and Graham [9] have used a class of rewrite systems called BURS to solve the instruction selection problem. It differs from the work of other tree pattern matching techniques in its ability to encode cost information into states of the automaton (thus avoiding cost analysis at code generation time) and in handling a larger class of rewrite systems.

We propose a system in which target machine instructions are represented as attributed production rules in regular tree grammar and instruction selection is modelled as a process of finding minimum cost derivation. A variation of bottom-up tree pattern matching automaton is used to find the minimum cost derivation. The cost information is encoded into states of the matching automaton. The table compaction technique of Chase [10] is extended to this automaton.

Our technique is similar to Hatcher and Christopher's technique [8]. However it differs from their technique in the following ways.

(i) It does not require modifying machine description to retain optimality. The required context information to find the minimum cost derivation is encoded into the states of the automaton.
(ii) A formal description based on regular tree grammar as in Giegerich and Schmal [11] is adopted for specifying machine descriptions.
(iii) An extension of Chase's [10] technique is used which improves the preprocessing time and table compression. Two schemes of table compression based on cost information are proposed.

The idea of encoding cost information into states has been mentioned in [5] and was later successfully adopted in [9]. We have also used this idea in our code generating system independent of the work of Pelegri-Llopart and Graham [9]. Our tree grammar based approach has following advantages over BURS theory [9]: It is simpler to understand and implement. As against the notions of rewrite rules and reachability, grammars and derivations are well understood. The grammar can be easily enhanced with attributes and predicates to aid the other supporting routines in code generation as in [12]. Blocking problem can be easily detected and good table compression techniques can be applied.

It is assumed that readers are familiar with tree pattern matching algorithms [10, 13] and the issues in retargetable code generation [8, 14]. The rest of the paper is organised as follows. The next section describes the instruction selection problem. In Section 3 we describe table construction and code generation algorithms. Section 4 describes experience with our code generation algorithm. Section 5 contains comparison with other work.

2. INSTRUCTION SELECTION

2.1 Tree grammar

A ranked alphabet is a finite set A of symbols, together with a function rank (or arity), such that rank(a) ≥ 0 for each $a \in A$. The symbols with rank 0 are called terminals (T) and the symbols with rank > 0 are called operators(θ).

The homogeneous tree language trees(A) over the ranked alphabet A, is defined by

—$t \in$ trees(A), if t is a terminal.
—a term op$(t_1, \ldots, t_q) \in$ trees(A), if $t_i \in$ trees(A) for $1 \leq i \leq q$, op $\in \theta$ and rank(op) $= q$.

Example 2.1.1 Let A be the alphabet $\{:=, +, \text{deref}, \text{c}, \text{sp}\}$ with ranks 2, 2, 1, 0 and 0 respectively. Elements of trees(A) are, for example: $+(\text{c}, \text{c})$, $+(\text{deref}(+(\text{c}, \text{sp})), \text{c})$.

A regular tree grammar G augmented with cost is a quadruple (N, A, P, S), where

—N is a finite set of nonterminal symbols,
—A is a ranked alphabet, $A \cap N = \emptyset$.
—P is a finite set of production rules augmented with cost. A production rule (or simply rule) is of the form $X \to t$, with $X \in N$ and $t \in$ trees$(A \cup N)$, with nonterminals given rank 0. Associated with each production rule is a cost called rule-cost. The rule-cost is a non-negative real number.
—S is a special nonterminal symbol called start symbol.

A tree pattern is a tree $p \in$ trees$(A \cup N)$ such that there exists a rule r or the form $X \to p$.

For $t, t' \in$ trees$(A \cup N)$, t immediately derives t', written $t \Rightarrow t'$, if there is a $r \in P$, say $X \to p$ such that t' results from t by replacing a leaf labeled X by p. The r used in the derivation step is

1.	$S \rightarrow$	:= deref GR	[2]
2.	$S \rightarrow$	:= deref cR	[2]
3.	$R \rightarrow$	c	[2]
4.	$G \rightarrow$	sp	[0]
5.	$G \rightarrow$	R	[0]
6.	$R \rightarrow$	G	[1]
7.	$R \rightarrow$	$+GR$	[1]
8.	$R \rightarrow$	$+RG$	[1]
9.	$R \rightarrow$	deref G	[2]
10.	$R \rightarrow$	$+Rc$	[2]
11.	$R \rightarrow$	$+cR$	[2]
12.	$R \rightarrow$	deref $+Gc$	[3]
13.	$R \rightarrow$	deref $+cG$	[3]
14.	$R \rightarrow$	+deref GR	[2]
15.	$R \rightarrow$	$+R$ deref G	[2]
16.	$R \rightarrow$	$+R$ deref $+Gc$	[3]
17.	$R \rightarrow$	$+R$ deref $+cG$	[3]
18.	$R \rightarrow$	+deref $+GcR$	[3]
19.	$R \rightarrow$	+deref $+cGR$	[3]

Fig. 1. Production rules for a target machine.

indicated by writing $\Rightarrow_r$, where relevant. The relations $\Rightarrow^+$ and $\Rightarrow^*$ are the transitive and the transitive and reflexive closure of $\Rightarrow$. As in context free grammar, we can associate a parse tree with a derivation.

Let $L(X) = \{t \mid t \in \text{trees(A) and } X \Rightarrow^* t\}$, then $L(S)$ is the language of G.

The rule-cost of r is incurred each time the production rule r is used in a derivation. The cost of a derivation is the sum of rule-costs of all rules used in derivation. Minimum cost derivation for a tree t is a derivation $S \Rightarrow^* t$ such that cost of the derivation is *minimum*.

A production rule of the form $X \rightarrow Y$, with $X, Y \in N$ is a chain rule. The chain rules may lead to circular derivation of the form $X \Rightarrow^* X$.

Example 2.1.2 Let P be the set of production rules shown in Fig. 1. Let G1 be the regular tree grammar $\{(R, G), A, P, S\}$ with A as before. ":=(deref(c), +(c, c))" belongs to $L(S)$, since $S \Rightarrow^*$ ":=(deref(c), +(c, c))". The derivation steps are as follows.

$$
\begin{aligned}
S &\Rightarrow_2 :=(\text{deref(c)}, \text{R}) \\
&\Rightarrow_{10} :=(\text{deref(c)}, +(\text{R}, \text{c})) \\
&\Rightarrow_3 :=(\text{deref(c)}, +(\text{c}, \text{c}))
\end{aligned}
$$

The derivation $R \Rightarrow_6 G \Rightarrow_5 R$ is a circular derivation.

2.2 The instruction selection problem

Retargetable code generation consists of the following steps:

(i) specification of IR and target machine instructions;
(ii) building tables for a matching automaton;
(iii) generating code from IR using the matching automaton.

The target machine instructions are specified as attributed production rules in regular tree grammar augmented with cost information in Graham–Glanville style. The intermediate representation IR is represented as a sequence of trees at the semantic level of the target machine as in [5], i.e. IR trees and the machine grammar use the same alphabet. The instruction selection process can be considered as finding the minimum cost derivation for a given IR tree. There is an action routine associated with each production rule. Given a derivation, a sequence of instructions can be generated using a scheme similar to syntax-directed translation.

As an example, Fig. 2 shows an IR tree for an assignment statement $a := b + 1$ in which a is a global variable stored in memory and b is a local variable stored on stack whose run time address is given as offset c_b from the stack pointer stored in register sp. The operator 'deref' is a de-referencing operator which gives the value at the address specified by operand. Figure 3 shows action routines for some rules in Fig. 1. Figure 4 shows one possible parse tree with the code sequence generated for that parse tree by executing the action routines.

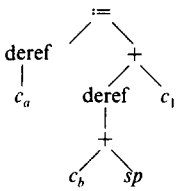

Fig. 2. Intermediate representation of $a := b + 1$

2.	$S \rightarrow :=\text{deref } c_a R_i$	{Emit ("STORE r%s, %s" i, a)}
19.	$R_i \rightarrow +\text{deref} + c_j G_k R_i$	{Emit ("ADD %s[%s], %s", j, k, i)}
3.	$R_i \rightarrow c_j$	{Allocreg(i); Emit("MOV # %s, %s", j, i)}
4.	$G_i \rightarrow sp.$	{$i := sp$}

Fig. 3. Action routines for some rules in Fig. 1.

2.3 Minimum cost derivation

A rule r of the form $X \rightarrow p$ matches a tree t, $t \in \text{trees(A)}$, if there exists a derivation such that $X \Rightarrow_r p \Rightarrow^* t$. A nonterminal X matches a tree t if there exists a rule r of the form $X \rightarrow p$ which matches t. A rule (or nonterminal) matches a tree t at node n if the rule (or the nonterminal) matches the subtree rooted at the node n.

For a tree $t \in \text{trees(A)}$, let $r: X \rightarrow p$ be a rule which matches the tree t at node n. We define the following.

(i) Cost of rule r matching t at node n.
(ii) cost of nonterminal X matching t at node n.

The cost of the rule r is minimum of the cost of all possible derivations of the form $X \Rightarrow_r p \Rightarrow^* t$. The cost of the matching nonterminal X is minimum of the cost of all rules r of the form $X \rightarrow p$ which matches t. Note that unlike the rule-cost, the cost of the rule r matching a tree t depends upon t.

To determine minimum cost derivation for a tree t, all derivations need not be found. Instead it is sufficient to determine all matching rules and nonterminals and their costs at each node of the tree. Minimum cost derivation is a sequence of rules, starting with a minimum cost rule corresponding to start symbol, say $r: S \rightarrow p$ such that $S \Rightarrow_r p \Rightarrow^* t$. The rule r matches the tree at the root and the successive rules in the derivation can be obtained by a top-down traversal of the tree.

2.4 Computing minimum cost derivation

Let p be any tree pattern. $\sigma(p)$ is a substitution in which each nonterminal X_i in p is replaced by a tree t_i, $t_i \in \text{trees(A)}$. The node at which the tree t_i is rooted is called touching position of X_i. A tree pattern matching algorithm can be used to find whether there exists a substitution such that $\sigma(p) = t$. A substitution is valid, if each X_i matches t_i. A pattern matches t, if there is a valid substitution $\sigma(p) = t$. A variation of bottom-up tree pattern matching algorithm can be used to find all valid substitutions (or matching patterns) at each node of a tree.

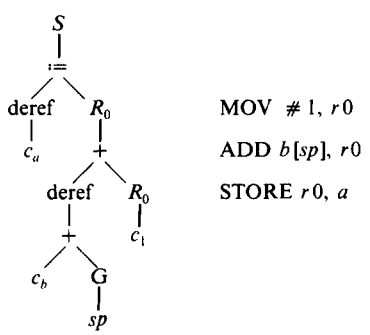

Fig. 4. The parse tree for the IR in Fig. 2. Bold lines indicate the parse tree relation.

If a term $op(t_1 \ldots t_q)$ is a pattern, then the terms $t_1, \ldots, t_q$ are called subpatterns. The crux of the bottom-up tree pattern matching algorithm is that given the patterns and subpatterns at the sons of a node n, the patterns and subpatterns matching at node n can be computed. The possible combinations of the matching patterns and subpatterns can be precomputed and a matching automaton can be built using these combinations. The automaton is appropriately modified so that the matching rules can be computed instead of matching patterns.

The cost of a pattern p matching t at node n is equal to the sum of the cost of the nonterminals in p matching the subtrees at their touching positions. The cost of the pattern which does not contain any nonterminal is defined to be zero. The cost of the rule r: $X \rightarrow p$ is equal to sum of the cost of pattern p and rule-cost(r). Instead of evaluating the cost at parsing time (i.e. at code generation time), the cost information can also be encoded into states of the matching automaton. Now the state is not just the set of rules, but a set of pairs (rule, delta-cost) called items. The delta-cost is defined by subtracting from the cost associated with each rule, the smallest cost associated with any rule in the set. The number of item sets are finite for real machines (see [9] for a discussion). They can be enumerated and an automaton can be built to determine the item sets.

Now to find the minimum cost derivation for a tree t, we proceed as follows: Tranverse the tree t "bottom-up" and compute the itemsets at each node using the automaton. Minimum cost rule corresponding to start symbol matching at root node can be determined from the itemset. Traversing the tree top-down, the minimum cost derivation can be determined.

2.5 Blocking and looping

Given a tree t, $t \in$ trees(A), the automaton will always find a derivation if it exists. Otherwise it will go to an error state. However one may want to know whether a given target machine specification grammar G (of language $L(S_G)$) is complete in the sense that any tree t in IR language $L(S_{IR})$ would have a derivation in grammar G or alternatively $L(S_{IR}) \subseteq L(S_G)$. Though this problem is decidable [11], no purely automatic method is used in practice. This problem can be avoided easily. We have to just describe IR specification as a subset of grammar G. Even if the grammar is complete, the code generator may get blocked due to "semantic blocking". This can be avoided by having a default action for every rule.

Looping can occur only due to circular derivations. Since they are computed using closure function, the problem does not occur.

3. ALGORITHM FOR CODE GENERATION

Let Match_rule and Match_NT be two functions which give the set of matching rules and matching nonterminals of a tree. An itemset $I(t)$ matching a tree t, $t \in$ trees(A) is a defined as follows:

$$I(t) = \{(r, \Delta_r) | r \in \text{Match\_rule}(t)\}$$

where Δ_r is difference between cost of r and the smallest cost associated with any rule in $I(t)$. The set of nonterminals matching t and the set of the respective Δ-costs can be obtained from $I(t)$ using the two functions π_n and π_c. We view these sets as ordered lists. $\pi_n(I(t))$ is the set of nonterminals matching t. $\pi_n^j(I(t))$ gives the jth element in the set. $\pi_c^j(I(t))$ is the delta-cost of the nonterminal $\pi_n^j(I(t))$. A function π_r is defined which gives the minimum cost rule corresponding to the nonterminal. $\pi_r^j(I(t))$ is defined to be a rule r: $n \rightarrow p$ such that delta-cost of the rule r is $\pi_c^j(I(t))$ and $n = \pi_n^j(I(t))$. There may be more than one rule having the minimum cost (i.e. $\pi_c^j(I(t))$), in which case we choose the rule occurring earlier in the specification.

In the next section we describe a table driven algorithm to determine itemsets. The procedure requires the production rules to be in normal form.

3.1 Normal form production rules

To treat pattern and subpattern information uniformly, the production rules are written in normal form. A rule is defined to be in **normal form** if it is either a chain rule or a rule of the form $X \rightarrow y$ or $X \rightarrow op(X_1, \ldots, X_q)$ where $X, X_1, \ldots, X_q$ are nonterminals, y is a terminal and op is an

1.	$S \rightarrow := YR$	[3]
2.	$S \rightarrow := WR$	[3]
3.	$Y \rightarrow$ deref G	[0]
4.	$W \rightarrow$ deref C	[0]
5.	$R \rightarrow C$	[2]
6.	$G \rightarrow SP$	[0]
7.	$G \rightarrow R$	[0]
8.	$R \rightarrow G$	[1]
9.	$R \rightarrow +GR$	[1]
10.	$R \rightarrow +RG$	[1]
11.	$R \rightarrow$ deref G	[3]
12.	$R \rightarrow +RC$	[2]
13.	$R \rightarrow +CR$	[2]
14.	$X \rightarrow +GC$	[0]
15.	$X \rightarrow +CG$	[0]
16.	$R \rightarrow$ deref X	[3]
17.	$R \rightarrow +YR$	[3]
18.	$R \rightarrow +RY$	[3]
19.	$Z \rightarrow$ deref X	[0]
20.	$R \rightarrow +ZR$	[3]
21.	$R \rightarrow +RZ$	[3]
22.	$C \rightarrow c$	[0]
23.	$SP \rightarrow sp$	[0]

Fig. 5. Normal form production rules for the target machine in Fig. 1.

operator of arity q. Any rule can be converted into normal form by introducing new nonterminals and rules. Rule-costs of the introduced rules are defined to be zero.

Figure 5 shows the production rules of grammar G1 (see Fig. 1) in normal form. The terminals c and sp are replaced by nonterminals C and SP. The nonterminals W, X, Y and Z are introduced to represent subpatterns. Representing subpatterns by nonterminals simplifies the computation of matching patterns at a node. The conversion of rules into normal form does not put any undue restrictions on the user. Further this process can be automated easily.

Note that the nonterminal may be used for factoring also. For example, X is used for representing the two patterns $+GC$ and $+CG$. Factoring reduces the number of rules in specification and helps in minimising table size.

3.2 Computing itemsets

Given the itemsets at sons of a node, computation of the itemset matching at the node involves following steps. First the matching rules at a node are computed from the matching nonterminals at the sons. Then the matching nonterminals at that node are found from the matching rules. From the matching nonterminals the matching chain rules are determined using Closure function and they are added to the set of matching rules. Then the Δ component is calculated for each matching rule.

For the sake of brevity, we assume

$r, r_1, r_2, \ldots, \in P$ (the set of production rules),
$n, n_1, n_2, \ldots, \in N$ (the set of nonterminals),
ntc, ntc_1, $\text{ntc}_2, \ldots, \in$ powerset(N) and
op, op_1, $\text{op}_2, \ldots, \in \theta$ (the set of operators).

Auxiliary functions

First_rule:

$$\text{First\_rule}(\text{op}) = \{r \mid r \text{ is } n \rightarrow \text{op}(n_1 \ldots n_q)\}$$

Father_rule functions:

$$\text{Father\_rule}_i(\text{n}) = \{r \mid r \text{ is } n' \rightarrow \text{op}(n_1 \ldots n_i \ldots n_q), \; n_i = n)\}$$

Closure_NT, Closure_rule:

$$\text{Closure\_NT}(\text{n}) = \{n_1 \mid n_1 \Rightarrow^+ n\}$$

$$\text{Closure\_rule}(\text{n}) = \{r \mid r \text{ is } n_1 \rightarrow n_2, n_1 \Rightarrow_r n_2 \Rightarrow^* n\}$$

The definitions of Father_rule functions and closure functions are extended to a nonterminal combination by taking unions.

$$\text{Father\_rule}_i(\text{ntc}) = \bigcup_{n \in \text{ntc}} \text{Father\_rule}_i(n)$$

$$\text{Closure\_rule}(\text{ntc}) = \bigcup_{n \in \text{ntc}} \text{Closure\_rule}(n)$$

$$\text{Closure\_NT}(\text{ntc}) = \bigcup_{n \in \text{ntc}} \text{Closure\_NT}(n)$$

Example 3.2.1 $\text{First\_rule}(\text{deref}) = \{Y \rightarrow \text{deref } G,\ W \rightarrow \text{deref } C,\ R \rightarrow \text{deref } G,$
$R \rightarrow \text{deref } X,\ Z \rightarrow \text{deref } X\}$

$\text{Father\_rule}_1(\text{Y}) = \{R \rightarrow +YR,\ S \rightarrow :=YR\}$

$\text{Father\_rule}_2(\text{C}) = \{R \rightarrow +RC,\ X \rightarrow +GC\}$

$\text{Closure\_NT}(\text{G}) = \{R, G\}$

$\text{Closure\_rule}(\text{G}) = \{R \rightarrow G,\ G \rightarrow R\}$

Algorithm 3.2.1 Computing itemset

PROCEDURE $I(t)$
BEGIN
  IF t is a terminal THEN
    BEGIN
      LET RULE $= \{r \mid r \text{ is } n \rightarrow t\}$
      LET NONTERM $= \{n \mid \exists r: n \rightarrow t, r \in \text{RULE}\}$
      For each $r \in$ RULE, LET $C_r = 0$
    END
  ELSE
    BEGIN LET t be a term of the form $\text{op}(t_1, \ldots, t_q)$
      where q is the arity of the operator 'op'.
    LET $\text{ntc}_i = \text{Match\_NT}(t_i)$, $1 \leq i \leq q$.
    LET RULE $=$ First_rule(op) $\cap$ $\text{Father\_rule}_1(\text{ntc}_1) \cap \ldots \cap$
                $\text{Father\_rule}_q(\text{ntc}_q)$.
    LET NONTERM $= \{n \mid \exists r: n \rightarrow p, r \in \text{RULE}\}$
      For each $r: n \rightarrow p, r \in$ RULE,
      LET p be a term of the form $\text{op}(n_{j(1)}, \ldots, n_{j(q)})$
        where $j(i)$ is an index such that $\pi_n^{j(i)}(I(t_i)) = n_{j(i)}$
      LET $C_r = \pi_c^{j(1)}(I(t_1)) + \cdots + \pi_c^{j(q)}(I(t_q))$
    END
  Match_NT(t) = NONTERM $\cup$ Closure_NT(NONTERM)
  Match_rule(t) = RULE $\cup$ Closure_rule(NONTERM)
    Initialise $\Delta_r = \infty$, for each $r \in$ Match_rule(t)
    Initialise $\text{DCOSTNT}_n = \infty$, for each $n \in$ Match_NT(t)
      For each $r, r \in$ RULE
        LET $\text{COST}_r = C_r +$ rule-cost(r).
      LET $\text{COST}_{\min} = \min\{\text{COST}_r \mid r \in \text{RULE}\}$.
      For each $r \in$ RULE,
        LET $\Delta_r = \text{COST}_r - \text{COST}_{\min}$.
      For each $n, n \in$ NONTERM
        $\text{DCOSTNT}_n = \min\{\Delta_r \mid \exists r: n \rightarrow p,\ r \in \text{RULE}\}$
      WHILE change of value of DCOSTNT_n or Δ_r occurs do
        BEGIN

For each $r: n \rightarrow n_1$ such that $r \in$ Closure_Rule(Match_NT(t))
BEGIN
DCOSTNT$_n$ = min(DCOSTNT$_n$, DCOSTNT$_{n_1}$ + rule-cost(r))
Δ_r = min(Δ_r, DCOSTNT$_{n_1}$ + rule-cost(r))
END
END
RETURN $\{(r, \Delta_r) | r \in$ Match_rule(t)$\}$
END

Example 3.2.2

Consider a tree $+(cc)$.

RULE at terminal $c = \{C \rightarrow c\}$
NONTERM at terminal $c = \{C\}$
Match_NT(c) = $\{R, G, C\}$
Match_rule(c) = $\{C \rightarrow c, R \rightarrow C, G \rightarrow R, R \rightarrow G\}$.
$I(c) = \{C \rightarrow c, 0), (R \rightarrow C, 2), (G \rightarrow R, 2), (R \rightarrow G, 3)\}$
$\pi_n(I(c)) = [C, R, G]$
$\pi_c(I(c)) = [0, 2, 2]$
$\pi_r(I(c)) = [C \rightarrow c, R \rightarrow C, G \rightarrow R]$.
RULE at root of the tree$(+(c, c))$
$= \{R \rightarrow RC, R \rightarrow +GR, R \rightarrow +RG,$
$R \rightarrow +CR, X \rightarrow +GC, X \rightarrow +CG\}$
NONTERM at root of the tree $(+(c, c)) = \{R, X\}$
Match_NT$(+(c, c)) = \{R, X, G\}$
Match_rule$(+(c, c)) = \{R \rightarrow +RC, R \rightarrow +CR,$
$R \rightarrow +RG, R \rightarrow +GR,$
$X \rightarrow +GC, R \rightarrow +CG,$
$R \rightarrow G, G \rightarrow R\}$
$I(+cc) = \{(R \rightarrow +RC, 2), (R \rightarrow CR, 2),$
$(R \rightarrow +GR, 3), (R \rightarrow +RG, 3),$
$(X \rightarrow +GC, 0), (X \rightarrow +CG, 0),$
$(R \rightarrow G, 3), (G \rightarrow R, 2)\}$
$\pi_n(I(+(c, c))) = [R, X, G]$
$\pi_c(I(+(c, c))) = [2, 0, 2]$
$\pi_r(I(+(c, c))) = [R \rightarrow +RC, X \rightarrow +GC, G \rightarrow R]$.

3.3 Table generation

The itemsets are enumerated and an integer is used as index for each itemset. As in the bottom-up tree pattern matching algorithm, tables are built for each operator. Given the indices of the matching itemsets at the sons, the index of the matching itemset at a node can be determined by table look-up. The required enumeration of itemsets and tables may be generated by a simple closure strategy which starts with itemsets for the terminals and repeatedly generates itemsets for trees with increasing heights, till no more itemsets are generated. Then the following table driven algorithm can be used to find the index $I(t)$ of the itemset matching a tree t.

Algorithm 3.3.1 Table driven algorithm for computing itemset

$I(t)$ = If t is a terminal A then TABLE$_A$
else TABLE$_A(I(t_1), \ldots, I(t_q))$
where t is a term $A(t_1, \ldots, t_q)$.

Where TABLE$_A$ is the table for each operator and terminal.

Chase [10] has used equivalence classes of match sets to compress tables and speed up preprocessing. In a two dimensional table, say for an operator "+", duplicate rows and columns represent the equivalence classes. They are called duplicate subtables in higher dimensions. It is possible to compress the interior of the table by "folding" duplicate subtables using "index maps". This idea can be extended to itemsets as follows.

LET N_{Aj} be the set of all nonterminals occurring in jth position in at least one pattern p of the form $A(n_1, \ldots, n_j, \ldots, n_q)$.

Two itemsets $i1$ and $i2$ are Aj-equivalent if

$\pi_n(i1) \cap N_{Aj} = \pi_n(i2) \cap N_{Aj} = \text{NTSET(say)}$
and C_n is constant over all $n \in \text{NTSET}$.

Where $C_n = \pi_c^{k1}(i1) - \pi_c^{k2}(i2), \pi_n^{k1}(i1) = \pi_n^{k2}(i2) = n$. (i.e. The respective cost differences of the nonterminals in NTSET are also same.)

It can be easily verified that if $i1$ and $i2$ are Aj-equivalent, then the table for the operator A of arity q contains a duplicate subtable at position j. As in [10], an eqivalence class of sets is represented by a member of that class. This member is called representer set for that class. Let X_i be the set of itemsets generated after processing trees of height i. Let S_{Aj_i} be the set of representer sets for a given A, j and X_i. To determine the set of itemsets X_{i+1}, we proceed as follows. Initialise $X_{i+1} = X_i$. Then construct all possible trees for each operator A from the representer set S_{Aj_i} and determine the matching itemset for each tree. Check whether any new itemset is formed. Add the new itemset to X_{i+1}. Repeat this until no more itemsets are generated.

Algorithm 3.3.2 Generating all itemsets

X: indexed set of itemset;
BEGIN
$X_0 := \cup \{I(A)\}$
$A \in$ terminal set
Generate S_{Aj_0} for each A and j.
REPEAT

$$X_{i+1} = X_i \cup \left[\bigcup_{A \in \text{op-set}} \left[\bigcup_{(s_1, \ldots, s_q) \in \prod_{j=1}^{j=q} S_{Aj_i}} I(A(s_1, \ldots, s_q)) \right] \right]$$

Generate $S_{Aj_{(i+1)}}$ for each A and j, $1 \leq j \leq \text{arity(A)}$.
UNTIL $\forall j\, S_{Aj_{(i+1)}} = S_{Aj_i}$;
END;

Algorithm 3.3.2 can be suitably modified to build the tables also. Let the index tables be represented by M_{Aj} and the interior of the tables by T_A. Then the modified table driven algorithm to compute the itemset $I(t)$ matching a tree t is given as follows.

Algorithm 3.3.3 Table driven algorithm for computing itemset

$I(t) =$ If t is a terminal A then T_A
else $T_A(M_{A1}(I(t_1)), \ldots, M_{Aq}(I(t_q)))$
where t is a term $A(t_1, \ldots, T_q)$.

The index map tables and tables for unary operators occupy considerable space. This can be reduced by using one more level of indexing by exploiting the property that itemset matching at a node depends only on the matching nonterminals and their cost (given by π_n and π_c) at the sons of the node rather than on complete information about itemsets at the sons. This property is called β-equivalence. Two itemsets $i1$ and $i2$ are said to be β-equivalent, if

$\pi_n(i1) = \pi_n(i2)$ and $\pi_c(i1) = \pi_c(i2)$.

Example 3.3.1

The itemsets

$i3 = \{(R \rightarrow +RC, 2), (X \rightarrow +GC, 0), (R \rightarrow G, 3), (G \rightarrow R, 2),$
$(R \rightarrow +GR, 3), (R \rightarrow +RG, 3)\}$

and

$$i4 = \{(R \rightarrow +CR, 2), (X \rightarrow +CG, 0), (R \rightarrow G, 3), (G \rightarrow R, 2),$$
$$(R \rightarrow +GR, 3), (R \rightarrow +RG, 3)\}$$

are β-equivalent,

since $\pi_n(i3) = \pi_n(i4) = [R, X, G]$ and

$\pi_c(i3) = \pi_c(i4) = [2, 0, 2]$.

β-equivalence implies Aj-equivalence for all A and j. Hence there will not be any reduction in the interior of the table (T-tables of Algorithm 3.3.3) for the operators of arity more than one. But the index map tables and tables for unary operators are reduced considerably. The preprocessing time for unary operators is also reduced.

Further the information required in the itemset, to get the minimum cost derivation is merely the set of nonterminals matching at each node and the corresponding minimum cost rule for each nonterminal. This fact can be used to reduce itemsets. Two itemsets $i1$ and $i2$ are said to be α-equivalent, if

$$\pi_n(i1) = \pi_n(i2),\ \pi_c(i1) = \pi_c(i2) \quad \text{and} \quad \pi_r(i1) = \pi_r(i2).$$

Example 3.3.2

The itemsets

$$i3 = \{(R \rightarrow +RC, 2), (X \rightarrow +GC, 0), (R \rightarrow G, 3), (G \rightarrow R, 2),$$
$$(R \rightarrow +GR, 3), (R \rightarrow +RG, 3)\}$$

and

$$i5 = \{R \rightarrow CR, 2), (X \rightarrow CG, 0), (R \rightarrow +RC, 2),$$
$$(X \rightarrow +GC, 0), (R \rightarrow G, 3), (G \rightarrow R, 2),$$
$$(R \rightarrow GR, 3), (R \rightarrow +RG, 3)\}$$

are α-equivalent,

since $\pi_n(i3) = \pi_n(i5) = [R, X, G]$,

$\pi_c(i3) = \pi_c(i4) = [2, 0, 2]$ and

$\pi_r(i3) = \pi_r(i4) = [R \rightarrow +RC, X \rightarrow +GC, G \rightarrow R]$.

α-equivalence implies β-equivalence. Hence there may not be much reduction in the tables. But the space required to store information about itemsets can be reduced. The tables are modified such that they now return an equivalent representer set $I_\alpha(t)$ rather than $I(t)$. The modified table driven algorithm is as follows.

Algorithm 3.3.4 Table driven algorithm for computing itemset

$I_\alpha(t)$ = If t is a terminal A then T_A

else $T_A(M_{A1}(\beta(I_\alpha(t_1))), \ldots, M_{Aq}(\beta(I_\alpha(t_q))))$

where t is a term $A(t_1, \ldots, t_q)$.

The significance of α equivalence classes is due to following consideration. Hoffmann and O'Donnell [13] have shown that "independent" patterns give rise to exponential growth of table size. Hatcher and Christopher [8] have observed that most of the independent patterns occur in machine specifications due to duplication of patterns for commutative operator and they discard one of the pattern in favour of the other if both of them match. In our case, such independent patterns are factored by nonterminals and α-equivalence class is used for discarding patterns. Unlike the Hatcher and Christopher technique [8], our technique discards patterns only if it does not affect the selection of minimum cost pattern. In Example 3.2.2, the nonterminal X factors the

patterns "$+GC$" and "$+CG$". The nonterminal R factors the patterns "$+RC$" and "$+CR$". One of the pattern in each case is discarded by α-equivalence.

Further the size of the tables depends upon the number of operators and the rank of the operators. An n-ary operator can be converted into binary operator using a pairing function operator. A set of operators having identical patterns and cost can be replaced by a generic operator.

3.4 Code generation algorithm

The code generator makes three passes over the tree. In the first pass IR tree is traversed bottom-up and itemsets matching at each node are found. The itemset has the information about matching nonterminals and the minimum cost rule corresponding to each nonterminal. At the end of first pass the start symbol should match at the root (otherwise it is an error condition). In the second pass the tree is traversed top-down to get the minimum cost parse tree. In the third pass the parse tree is traversed and the instructions are emitted by executing actions.

4. IMPLEMENTATION

The actual implementation of the itemsets computation algorithm involves much more book keeping than described here. The target machine specifications are compiled into coded form. The set of rules is represented as a bit vector and the set union and intersection operations are implemented using bit vector "AND" and "OR" operations. Two hash tables are maintained to check whether a new Match_rule set or a new itemset is formed. Formation of a new Match_rule set implies formation of a new itemset. If the Match_rule set is already present, Δ is computed for that set and checked in the hash table for the formation of a new itemset. The hash tables store an unique index for each Match_rule set and itemset. The functions First, Father and Closure, and the itemsets corresponding to terminals are computed first. The set of itemsets is represented as a list of indices. All iterations over sets and product of sets are done over the indices corresponding to the sets and products. An itemset is formed during each iteration and if it is a new one (checked using hash search), then it is added to the list corresponding to X_{i+1}. Further it is not necessary to iterate over all the members

$$\prod_{j=1}^{j=q} S_{Aj_i}$$

but only over the members in

$$\prod_{j=1}^{j=q} S_{Aj_i} - \prod_{j=1}^{j=q} S_{Aj_{(i-1)}}$$

4.1 Experimental results

We have developed a prototype code generator system (TCGG) based on the technique described in this paper and have got some encouraging results during the investigation on Motorola MC68000 and VAX 11 code generators on a M68020 processor based machine. The relevant statistics for the code generators are tabulated in Fig. 6.

The quality of the code generated is compared for different IRs with that of Graham–Glanville scheme and it was found to be better, since TCGG utilised the addressing mode instructions optimally. The code generator produces 200 assembly instructions per sec in Motorola 68020 based system. The size of the code generator is about 70K for VAX-11 and 100K for MC68000.

Though the number of rules are less for MC68000 code generator than for VAX-11, the table size is more for MC68000 due to the following reason. There are two types of general purpose registers, viz. data registers and address registers. In some cases it is beneficial to use an address register instead of a data register. This irregularity in instruction set gives rise to large Δ-components. Hence the number of itemsets and preprocessing is considerably high. α-equivalence is more effective for this type of machines. In MC68000 code generator, around 25K of itemset storage (storing 2430 itemsets) is reduced to 3K (storing 342 itemsets) by α-equivalence. In VAX-11 code generator, around 3.6K of itemset storage (storing 526 itemsets) is reduced to 1.8K (storing 263 itemsets). β-equivalence reduces space for index maps and unary operator tables by almost half in both cases, taking into account the reduction of itemsets by α-equivalence.

	VAX 11	Motorola MC68000
No of rules	144	114
No of nonterminals	20	18
No of terminals	15	12
No of unary operators	8	9
No of binary operators	18	16
No of itemsets	526	2430
No of α-equivalence Classes	263	342
No of β-equivalence Classes	94	149
Table size		
Index maps	7.2K	9.6K
unary op tables	1.6K	2.7K
Interior of binary op tables	1.4K	19.2K
Itemset storage	1.8K	3.0K
Total table size	12.0K	34.5K
Preprocessing time	108 sec	720 sec

Fig. 6. Statistics of VAX 11 and MC68000 code generator systems

Note that only 9 bits are required to represent each element in the tables for MC68000 code generator. In Fig. 6, it is assumed that each element is represented by a word (two bytes). The table size can be reduced to approximately 20K by bit-compression techniques.

5. COMPARISON WITH OTHER WORK

Locally optimal instruction selection and specification ease are the main advantages of the tree pattern matching techniques [15] over LR-parser methods. Our method inherits these advantages. It differs from Aho *et al.*'s tree specification scheme in its ability to perform code generation without cost analysis during code generation and from Hatcher and Christopher's scheme in not requiring modification of specification for optimality. Since we do not have any implementation based on other tree pattern matching methods, our comparison is qualitative rather than quantitative.

The main advantage over Aho *et al.*'s [14] method is faster code generation due to following reasons:

(i) An efficient bottom-up tree pattern matching algorithm which is linear in input tree size is used compared to the top-down tree matching algorithm used by them which is quadratic in pattern size and input tree size.
(ii) Cost analysis is not required at code generation time.
(iii) The matching chain rules are precomputed.

Though the preprocessing time and space requirement are more for the bottom-up tree pattern matching techniques, Chase [10] has shown that it is practicable. Pelegri-Llopart and Graham's experiments, and our own confirm this.

Our technique is similar to Hatcher and Christopher's [8] technique in the sense that we use same type of machine descriptions and make three passes over IR tree to generate code. The main differences are the following:

(i) Their technique requires modifying some part of machine description to retain optimality. If the pattern set is large, it is difficult to do such modification by hand. In our system, the context information required to determine the minimum cost derivation is automatically generated.
(ii) In our method, a formal description based on regular tree grammar as in Giegerich and Schmal's paper [11] is adopted for specifying machine descriptions. This enables a more thorough treatment of the blocking problem. Giegerich and Schmal perform cost analysis (as in [7]) at code generation time. We define itemsets to encode cost information into tables.
(iii) An extension of Chase's [10] technique is used which improves the preprocessing time and table compression. Two schemes of table compression based on cost information are proposed. Unlike in Hatcher and Christopher's technique, the necessary information for table compression is automatically generated.

In Pelegri-Llopart and Graham's code generator system, cost information is encoded into the states so that cost analysis need not be done at code generation time. In this aspect, our technique is similar to theirs. Their system handles a larger rewrite system in which commutativity property of the operators and "constant folding" type rules can be directly specified. However we are able to use better compression schemes and detect the blocking problem more easily. Patterns duplicated for commutative operators are factored by nonterminals and α-equivalence is used to minimise states by exploiting cost information. Unlike their method, the minimised set of states is directly generated by the table generator rather than first determining the set of all possible states and then compressing them using α-equivalence relation. Further it is simpler to understand and implement. As against the notions of rewrite rules and reachability, grammar and derivation are well understood and the grammar can be easily enhanced with attributes and predicates to aid the other supporting routines in code generation as in [17].

6. CONCLUSION

Locally optimal instruction selection and specification ease are the main advantages of the tree specification scheme. Previous techniques using tree specification scheme perform expensive cost analysis during code generation or require modification of specification for optimal instruction selection. We have developed a practical technique by which it is possible to perform optimal instruction selection without requiring cost analysis during code generation. Two table compression schemes based on cost information have been proposed which reduce the table size considerably.

7. SUMMARY

This paper proposes a system based on tree pattern matching for performing optimal instruction selection for expression trees without requiring cost analysis at code generation time. The target machine instructions are specified as attributed production rules in a regular tree grammar augmented with cost information in Graham–Glanville style. The intermediate representation (IR) is a sequence of expression trees. Instruction selection is modelled as a process of determining minimum cost derivation for a given IR tree.

Matching of a rule and matching of a nonterminal are defined and a derivation is expressed in terms of matching rules and nonterminals. A variation of bottom-up tree pattern matching automaton is proposed to find matching rules and nonterminals. To determine the minimum cost derivation, each state of the automaton is defined to be a set of pairs (rule, delta-cost) called itemset. An algorithm is described to build the automaton. Table folding technique of Chase is extended to this automaton which reduces the preprocessing time and table size. Two compression techniques (called α-equivalence and β-equivalence techniques) based on cost information are proposed.

A prototype code generator system is developed and a report on the investigations of Motorola M68000 and VAX-11 code generators is included.

REFERENCES

1. Ganapathi, M., Fischer, C. and Hennessy, J. Retargetable compiler code generation. *Comput. Surv.* **14**: 573–592; December 1982.
2. Graham, S. Table driven code generators. *IEEE Comput.* **13**: 25–34; August 1980.
3. Leverett, B., Cattell, R. *et al.* An overview of the production-quality compiler–compiler project. *IEEE Comput.* **13**: 38–49; August 1980.
4. Bell, C. G. and Newell, A. *Computer Structures: Reading and Examples.* New York: McGraw–Hill; 1971.
5. Henry, R. R. Graham–Glanville code-generators. Ph.D. dissertation, University of California, Berkeley; 1984.
6. Aho, A. V., Ganapathi, M. and Tjiang, S. W. K. Code generation using tree matching and dynamic programming. *ACM TOPLAS.* To appear.
7. Aho, A. V. and Johnson, S. C. Optimal code generation for expression trees. *J. ACM* **23**: 488–452; 1976.
8. Hatcher, P. J. and Christopher, T. W. High quality code generation via bottom-up tree pattern matching. *Proc. 13th ACM Symposium on Principles of Programming Languages*, pp. 119–129; 1986.
9. Pelegri-Llopart, E. and Graham, S. Optimal code generation for expression trees: An application of BURS theory. *Proc. 15th Annual ACM Symposium on Principles of Programming Languages*, pp. 294–308; January 1988.
10. Chase, D. R. An improvement to bottom-up tree pattern matching. *Proc. 14th ACM Symposium on Principles of Programming Languages*, pp. 168–177; 1987.

11. Giegerich, R. and Schmal, K. Code selection techniques: Pattern matching, tree parsing, and inversion of derivors. *European Symposium on Programming Languages. Lecture Notes in Computer Science* **300**: 247–268; 1988.
12. Ganapathi, M. and Fischer, C. N. Affix grammar driven code generation. *ACM TOPLAS* **7**: 560–599; October 1985.
13. Hoffmann, C. and O'Donnell, M. Pattern matching in trees. *J. ACM* **29**: 68–95; 1982.
14. Aho, A. V. and Ganapathi, M. Efficient tree pattern matching: An aid to code generation. *Proc. 12th ACM Symposium on Principles of Programming Languages*, pp. 334–340; 1985.
15. Appel, A. W. Concise specifications of locally optimal code generators. Technical Report, University of Princeton; February 1987.
16. Ganapathi, M. Retargetable code generation and optimisation using attribute grammars. Ph.D. dissertation, University of Wisconsin–Madison; 1980.
17. Glanville, R. S. and Graham, S. L. A new method for compiler code generation. *Proc. 5th Annual ACM Symposium on Principles of Programming Languages*, pp. 231–240; January 1978.

About the Author—ARUNACHALAM BALACHANDRAN received his B.Sc in physics from Madras University in 1979 and his B.E. in electrical technology and electronics from the Indian Institute of Science, Bangalore in 1982. He worked in ORG Systems, Baroda as a Hardware Engineer from 1982 to 1985. Presently he is working for his Ph.D. in the Department of Computer Science and Engineering at the Indian Institute of Technology, Bombay. His current areas of interest are automatic code generators, tree transformation systems and parallel compilers.

About the Author—DHANANJAY MADHAV DHAMDHERE was born in Pune, India in 1949. He received his B.Tech., M. Tech. degrees in electrical engineering and Ph.D. in computer science from the Indian Institute of Technology, Bombay. Dr Dhamdhere joined the faculty of the Indian Institute of Technology in 1972, becoming Assistant Professor of computer science in 1974. He became Associate Professor in 1983 and Professor of computer science in 1985. During 1974–81, while in charge of the System Software Group, he led the design and implementation of the fast turn around Fortran system *IITFORT* for IBM mainframe computers. As a member of the Computer Society of India, he served as the editor of its journal *Computer Science and Informatics* for the period 1983–86. During 1986–88 he was a Visiting Professor at the University of Connecticut, Storrs. His research and consultancy interests are in the area of optimising compilers, programming languages and operating systems. He is the author of several research papers and two books entitled *Compiler Construction* and *Introduction to System Software*.

About the Author—SUPRATIM BISWAS received his Ph.D. degree in computer science from the Indian Institute of Technology, Kharagpur in 1982. He joined the Department of Computer Science and Engineering, Indian Institute of Technology, Bombay in 1980 and is presently an Assistant Professor. His research interests are in the areas of compiler optimisation, compilers for multiprocessors, algorithms and complexity, and combinatorial optimisation.