High-Quality Code Generation Via Bottom-Up Tree Pattern Matching

Philip J. Hatcher
Thomas W. Christopher
Department of Computer Science
Illinois Institute of Technology
Chicago, IL 60616

Abstract

High-quality local code generation is one of the most difficult tasks the compiler-writer faces. Even if register allocation decisions are postponed and common subexpressions are ignored, instruction selection on machines with complex addressing can be quite difficult. Efficient and general algorithms have been developed to do instruction selection, but these algorithms fail to always find optimal solutions. Instruction selection algorithms based on dynamic programming or complete enumeration always find optimal solutions, but seem to be too costly to be practical. This paper describes a new instruction selection algorithm, and its prototype implementation, based on bottom-up tree pattern-matching. This algorithm is both time and space efficient, and is capable of doing optimal instruction selection for the DEC VAX-11 with its rich set of addressing modes.

Introduction

Many tools have been developed to aid the compiler-writer in the difficult task of creating and porting code generators. The basic idea in most of these tools is to read expression trees (provided by earlier phases of the compiler) and to search these trees for patterns stored in previously built tables. In the tables, associated with each pattern, is code to be generated when that pattern is found in an expression tree. Also, when a pattern is applied to the tree, the tree is simplified by replacing the portion of the tree matching the pattern with a rename symbol associated with the pattern. In this manner code is generated as the tree is reduced to a single node. A generic code generator (one denuded of its tables of patterns) is a specification for a search (often heuristic in nature) of the expression tree. The development of a specific code generator, therefore, involves providing patterns and associated low-level code which, when encoded in tables, will allow the generic code generator to output correct, and hopefully efficient, low-level language programs for a particular machine. These existing code generation techniques differ widely in how the patterns are specified and in the nature of the search done by the generic code generator.

The PQCC system [CAT78] [CAT80] [LEV80] and the system of Schmidt and Völler [SCH84] automatically derive the patterns from descriptions of the intermediate code used to express the expression trees and the target machine. The portable C compiler [JOH78] and the Krumme-Ackley [KRU82] code generator require the patterns to be specified at a very low level and in a format very similar to the internal form the patterns will have when stored in the tables. The Amsterdam Compiler Kit [TAN83] requires the compiler-writer to provide patterns which express the translation of stack configurations, of a simple stack machine computing the expression to be compiled, to target machine instructions. The Graham-Glanville technique [GLA77] [GRA82] [AIG84] [HEN84] uses the convenient notation of augmented production rules of a formal grammar to express the patterns. This notation is also basically used in the systems of Ganapathi and Aho [GAN80] [AHO85], our earlier work [HAT83] [CHR84] [HAT85a], and the work of Horspool and Scheunemann [HOR85]. The PQCC system uses a top-down tree pattern matching algorithm which always applies the biggest matching pattern. If the later evaluation of a subtree results in an operand mismatch, the operand is coerced into a matching class by inserting extra code if necessary. Both the system of Schmidt and Völler and the portable C compiler directly match patterns and, in the absence of a direct match, apply user supplied heuristics to decompose the more complex expression subtrees. The Krumme-Ackley system and the Amsterdam Compiler Kit require the compiler-writer to associate cost data with the patterns and then use exhaustive search to find the best sequence of patterns to apply to the expression tree (within varying constraints and limitations). The Graham-Glanville systems linearize the expression trees (and the patterns) to use LR parsing to apply patterns to the expression. In ambiguous situations the code generator favors the biggest pattern (and the systems must be constructed to recover from blocks resulting from a poor choice in the strict bottom-up, left-to-right LR processing). Ganapathi's orginal extension of the Graham-Glanville technique used attributed parsing to apply the patterns. This allowed the compiler-writer to specify predicates to be consulted in ambiguous situations, thus freeing

© 1986 ACM-0-89791-175-X-1/86-0119 $00.75

the system from always choosing the biggest pattern. Ganapathi and Aho's system associates costs with the patterns and uses a top-down tree pattern matching algorithm coupled with a dynamic programming algorithm to select the best sequence of patterns. Our earlier work and the work of Horspool and Scheunemann also use associated costs and dynamic programming algorithms to locate the best sequence of patterns.

These systems either make no guarantee of an optimal application of the patterns (even in the context of ignoring register allocation and common subexpressions) or they use algorithms based upon exhaustive search or dynamic programming. Our earlier experience led us to conjecture that, with the appropriate preprocessing of the patterns, the patterns could be applied optimally to the expression tree without resorting to exhaustive search or dynamic programming. This paper provides evidence that the conjecture is true. We describe a new instruction selection algorithm, and its prototype implementation, based on bottom-up tree pattern-matching. This algorithm is both time and space efficient, and is capable of doing optimal instruction selection for the DEC VAX-11 with its rich set of addressing modes.

We chose to use Graham-Glanville style pattern specifications. We also chose to use the purely syntactic approach of the Graham-Glanville technique. However, we feel that the algorithms described here would be applicable in a situation where the patterns are automatically derived. Also, we feel our ideas are usable in systems which allow semantic predicates to be attached to the patterns.

We will begin by describing the code generation algorithm itself. Then the preprocessing required to guarantee optimality will be discussed. Our experience using the system to generate code for the DEC VAX-11 will then be described. Finally, shortcomings and required future work will be confessed.

The Code Generation Algorithm

The code generation algorithm consists of three passes over an input expression tree and works from Graham-Glanville style pattern specifications augmented with cost information. The pattern specifications consist of the pattern tree itself, a rename symbol which can be used if the pattern matches, a cost datum (ie if the pattern is selected, what it will contribute to the cost of computing the overall expression), and actions to be taken when a pattern matches (emitting instructions or building fields to be incorporated into instructions). It is assumed that the total cost of computing an expression can be calculated simply by summing the costs of the patterns used to match the tree representing the expression.

The first pass is based upon a bottom-up tree pattern matching algorithm described by Hoffmann and O'Donnell [HOF82]. This pass labels each node of the tree with the set of patterns (and pieces of patterns) which match at that node. The current node is labelled by using the known information—the type of the node (operator name) and the labels of the children—to access tables. To keep the total table size small, two separate tables are consulted. First, the label of each child, along with the node's type and the position of the child, are looked up in the *makeindex* table to get the appropriate index value to use in the next table lookup. Then the node type, and the list of indices for the children, are looked up in the *matchtab* table to get the value with which to label the current node. See Figure 1.

The second pass is top-down over the tree and is used to select the best sequence of patterns from the collection of patterns which were found to match on pass one. At each node in the tree, the second pass uses the label placed on the node by pass one, and a choice symbol passed down by the parent, to access yet another table, *matchsettab*, to decide which pattern should be used to begin computing the subtree rooted at the current node. Each entry in *matchsettab* represents a correspondence between a rename symbol (the index) and a pattern (the value). The actions associated with these correspondences will be executed during pass three in order to emit instructions. The pattern selected then dictates the choices (the children of the root operator of the pattern) to be forced down upon the children of the current node. The choice symbol presented to the root of the tree is a symbol corresponding to a Graham-Glanville grammar start symbol.

On pass two the *matchsettab* is not consulted if the parent's choice is a pattern—this pattern is used directly to determine what choices to present to the current node's children. Also, *matchsettab* may be consulted multiple times for one node if simple renaming rules are selected—rules which rename one renaming symbol to another. Figure 2 provides a more precise description of pass two.

The third pass simply runs over the tree one final time, using the patterns selected on pass two, to output (by executing the patterns' actions) an instruction sequence computing the expression described by the tree.

This linear algorithm does make three passes over the expression tree (as opposed to the strictly one pass approach of the conventional Graham-Glanville code generators), but assuming the three table lookups of the first two passes can be efficiently implemented, the algorithm is still a streamlined one. We have found, in our prototype implementation, that the running time of this algorithm is dominated by the pass three processing. The nature of this pass three processing (and thus the cost) is the same in our implementation as in a conventional Graham-Glanville system.

Hoffmann and O'Donnell assume that no two patterns have an "independent" relationship. In an independent relationship, trees exist which both patterns match, trees exist which only the first pattern match, and trees exist that only the second pattern matches. In our experience independent patterns have most often occurred when we are forced to

```
procedure $pass1(x)$
begin
  if $x.op$ is an operator then
    for $i$ := 1 to $arity(x.op)$ do
      $c_i$ := $pass1(x.child_i)$
      if $x.child_i.op$ is an operator then
        $c_i$ := $makeindex[x.op][i][c_i]$
    end
    $x.label$ := $matchtab[x.op][c_1] \ldots [c_{arity(op)}]$
  else
    $x.label$ := $x.op$
  end
  return $x.label$
end
```

Figure 1 The first pass of the code generation algorithm.

```
procedure $pass2(choice, x)$
begin
  $x.patlist$ := empty
  while $choice$ is a rename symbol do
    $cur$ := $matchsettab[x.label][choice]$
    append $cur.patno$ to $x.patlist$
    $choice$ := $cur.rhs$
  end
  if $choice$ is a pattern = $(op, ch_1, \ldots, ch_{arity(op)})$ then
    for $i$ := 1 to $arity(op)$ do
     $pass2(ch_i, x.child_i)$
    end
/*else $choice$ is a "terminal" symbol - no action*/
  end
end
```

Figure 2 The second pass of the code generation algorithm.

duplicate a pattern because it contains a commutative operator (that is the patterns are mirror images with the same costs, renaming symbols, and actions) However, some "truly" independent patterns have existed in the families of patterns we have been studying. This fact led to our addition of the extra table, *makeindex*, to the algorithm of Hoffmann and O'Donnell. The *makeindex* table forces a choice to be made between independent patterns as the tree is being labelled on pass one. This keeps the number of possible labels small by not propagating up the tree the independent situation (Hoffmann and O'Donnell showed that independence can lead to an exponential explosion in the number of labels). In general, this forced choice must lead to suboptimal instruction selection in some cases. However, we have found that the nature of the independence, which exists in the sets of patterns we have been studying, is such that an optimal choice can be made. The analysis (described below) performed at code generator generation time in our system, either guarantees the optimality of each choice, or prints a warning that the choice cannot be guaranteed.

A Code Generation Example

The following simple collection of patterns is listed using Graham-Glanville notation. Each pattern consists of (1) a production rule, where the left-hand-side is the rename symbol and the right-hand-side is the pattern linearized in prefix form; (2) a list of actions describing the emitting of instructions and the building of instruction result fields; and (3) a cost datum. The patterns generate code for the DEC PDP-11 with costs being the size of the resultant instructions in words.

pattern1: start → reg
action:
cost: 0

pattern2: reg↑R → Add reg↑X reg↑Y
action: emit "ADD Y,X"; R := X
cost: 1

pattern3: reg↑R → Add address_mode↑X reg↑Y
action: emit "ADD X,Y"; R := Y
cost: 1

pattern4: reg↑R → Add reg↑X address_mode↑Y
action: emit "ADD Y,X"; R := X
cost: 1

pattern5: reg↑R → address_mode↑X
action: Z := alloc(reg); emit "MOV X,Z"; R := Z
cost: 1

pattern6: address_mode↑R → Deref Global↑X
action: R := X
cost: 1

pattern7: address_mode↑R → Deref reg↑X
action: R := "(" concat X concat ")"
cost: 0

pattern8: address_mode↑R → Deref Add reg↑X Const↑Y
action: R := Y concat "(" concat X concat ")"
cost: 1

pattern9: address_mode↑R → Const↑X
action: R := X
cost: 1

The algorithm tracks partial pattern matches, so it is necessary to enumerate the *pattern forest* (all patterns and subpatterns):

P1 Add,reg,reg
P2 Add,address_mode,reg
P3 Add,reg,address_mode
P4 Deref,Global
P5 Deref,reg
P6 Add,reg,Const
P7 Deref,P6

For these patterns the content of *matchtab* is the following:

[Add][address_mode][Const] = P1-P2-P3-P6
[Add][address_mode][address_mode] = P1-P2-P3
[Add][address_mode][reg] = P1-P2
[Add][reg][Const] = P1-P3-P6
[Add][reg][address_mode] = P1-P3
[Add][reg][reg] = P1
[Deref][P6] = P5-P7
[Deref][Global] = P4
[Deref][reg] = P5

The content of *makeindex* is:

[Deref][1][P1-P2] = reg
[Deref][1][P1-P3-P6] = P6
[Deref][1][P1-P3] = reg
[Deref][1][P1] = reg
[Deref][1][P4] = reg
[Deref][1][P5] = reg
[Deref][1][P5-P7] = reg
[Deref][1][P1-P2-P3-P6] = P6
[Deref][1][P1-P2-P3] = reg

[Add][1][P1-P2] = reg
[Add][1][P1-P3-P6] = reg
[Add][1][P1-P3] = reg
[Add][1][P1] = reg
[Add][1][P4] = address_mode
[Add][1][P5] = address_mode
[Add][1][P5-P7] = address_mode
[Add][1][P1-P2-P3-P6] = reg
[Add][1][P1-P2-P3] = reg

[Add][2][P1-P2] = reg
[Add][2][P1-P3-P6] = reg
[Add][2][P1-P3] = reg
[Add][2][P1] = reg
[Add][2][P4] = address_mode
[Add][2][P5] = address_mode
[Add][2][P5-P7] = address_mode
[Add][2][P1-P2-P3-P6] = reg
[Add][2][P1-P2-P3] = reg

The content of *matchsettab* is:

[P1-P2][reg] = P2 (*pattern3*)
[P1-P2][start] = reg (*pattern1*)

[P1-P3][reg] = P3 (*pattern4*)
[P1-P3][start] = reg (*pattern1*)

[P1-P3-P6][reg] = P3 (*pattern4*)
[P1-P3-P6][start] = reg (*pattern1*)

[P1][reg] = P1 (*pattern2*)
[P1][start] = reg (*pattern1*)

[P4][reg] = address_mode (*pattern5*)
[P4][start] = reg (*pattern1*)
[P4][address_mode] = P4 (*pattern6*)

[P5][reg] = address_mode (*pattern5*)
[P5][start] = reg (*pattern1*)
[P5][address_mode] = P5 (*pattern7*)

[P5-P7][reg] = address_mode (*pattern5*)
[P5-P7][start] = reg (*pattern1*)
[P5-P7][address_mode] = P7 (*pattern8*)

[P1-P2-P3][reg] = P2 (*pattern3*)
[P1-P2-P3][start] = reg (*pattern1*)

[P1-P2-P3-P6][reg] = P2 (*pattern3*)
[P1-P2-P3-P6][start] = reg (*pattern1*)

Consider the following expression (also expressed in prefix form):

Add Deref A Deref Add Deref B 4

where A and B are global variables. The bottom-up nature of the first pass of the new code generation algorithm causes nodes immediately above leaves to be labelled first (leaves are considered to be labelled by the leaf symbol itself). In this case this causes the subtrees expressing the dereference of a global to be examined first. The *makeindex* table is not consulted for leaves (the index is the leaf itself), so these subtrees are simply labelled with the contents of *matchtab*[Deref][Global] which is P4. Then the subtree rooted by the bottom-most Add operator is labelled. The *makeindex* table is consulted with the label of the left child: *makeindex*[Add][1][P4] contains address_mode. The right child is a leaf so it is used as the second index to *matchtab*. Therefore the subtree is labelled with the contents of *matchtab*[Add][address_mode][Const] which is P1-P2-P3-P6. Now the dereference node immediately above can be labelled. The *makeindex*[Deref][1][P1-P2-P3-P6] entry is consulted and its value, P6, is used in *matchtab*[Deref][P6] to result in the label P5-P7. Finally, the root node can be labelled. The *makeindex* table is examined for the labels of the two children: *makeindex*[Add][1][P4] is address_mode; *makeindex*[Add][2][P5-P7] is also address_mode. So the root is labelled with the contents of *matchtab*[Add][address_mode][address_mode] which is P1-P2-P3.

The second pass of the new algorithm can now proceed to traverse the labelled tree and select the best sequence of patterns from which to generate code. This pass is top-down and uses the *matchsettab* table. The root must always conform to the start symbol. This causes the *matchsettab* entry for the label of the root to be examined for the start symbol. The contents of this entry (*matchsettab*[P1-P2-P3][start]) is reg. Since this is not a pattern, the process is continued for the current node using reg as the new choice symbol. The entry *matchsettab*[P1-P2-P3][reg] is consulted and its value, P2, is used to continue the process down the tree. Since P2 is (Add,address_mode,reg), the choice symbol passed to the left subtree is address_mode and the choice symbol passed to the right subtree is reg. For the left subtree, address_mode is "connected" to P4 by consulting *matchsettab*[P4][address_mode] which contains P4. Similarly, the right subtree is handled by first consulting *matchsettab*[P5-P7][reg] which is address_mode, and then examining *matchsettab*[P5-P7][address_mode] which is P7. This leads the subtree expressing the dereference of B to be handled by looking up *matchsettab*[P4][reg]. This produces address_mode which causes *matchsettab*[P4][address_mode] to be examined resulting in P4. This then completes pass two. It results in the tree having the following labelling (the label for each node is a sequence of patterns to be used to generate code, and is expressed in parentheses after each node):

Add (start → reg → P2) Deref (address_mode → P4) A Deref (reg → address_mode → P7) Add Deref (reg → address_mode → P4) B 4

Pass three retraces this sequence of patterns in reverse order, bottom-up through the tree, executing the semantic actions. This results in the following instructions being emitted:

```
MOV B,R1
MOV 4(R1),R2
ADD A,R2
```

Pattern Preprocessing

The preprocessing required by the first pass of the code generator is derived from the work of Hoffmann and O'Donnell. First, subsumption relationships (if the first pattern matches, then the second pattern must also match) are detected between patterns by examining patterns in order of increasing height. A pattern subsumes another if the root operators of the two patterns are the same and if each of the first pattern's children subsumes the corresponding child in the other pattern. The subsumption relationships of pattern leaves can be detected by examining the relationships between renaming symbols and patterns (patterns which consist simply of a renaming symbol are allowed).

As the subsumption relationships are detected, cost information is also compiled. If one pattern subsumes another, then the first pattern can be "converted" to the second pattern by recursively applying renaming rules to its children. As these renaming rules are applied, the costs of using the rules can be summed to obtain the cost of "converting" the first pattern to the second. This cost, or distance, between patterns is saved for use in later analysis.

The known subsumption relationships are then used to generate the *matchtab* table required on pass one of the code generator. Each pattern

$$p = op\ ch_1 \ldots ch_{arity(op)}$$

is taken in turn and the subsumption information is used to determine what set of *matchtab* entries should be (partially) labelled with the current pattern. The set of entries accessed is exactly the set of all

$$[op][ch'_1]\dots[ch'_{arity(op)}]$$

where for $1 \leq i \leq arity(op)$

$$ch'_i \text{ subsume } ch_i$$

and there exists patterns

$$q_i = op\ ch''_1 \dots ch''_{arity(op)}$$

such that

$$ch'_i = ch''_j \text{ and } i = j$$

Note, that for a given operator, its indices in *matchtab* are resticted (by position) to the set of symbols which appear as children (in the particular position) of the operator in some pattern. This also serves to keep the overall table size small and is justified because, if code is to be generated for a tree, each of its subtrees must be converted to match a form specified in a pattern for the root operator of the tree.

After all patterns have been processed, the value associated with each entry in *matchtab* is a list of patterns. These lists of patterns are the sets of patterns which may label a node of the expression tree at code generation time. From these labels the table *matchsettab* is built. For each label, the set of possible choices, which may be presented to it during pass two of the code generator, are evaluated; and a pattern, in the label, or a renaming symbol, subsumed by a pattern in the label, are connected to each possible choice by doing a cost analysis when multiple connections exist. The cost analysis is performed by building a tree representing the set of trees which could possibly be matched by the label. This is done by overlapping the patterns which make up the label. Then, for each choice, the cost data is consulted to see which of the possible connections is cheapest with respect to the representative tree.

When overlapping the patterns, if two patterns have a subtree and renaming symbol in the same position, the subtree is used. If two patterns have renaming symbols in the same position, then the renaming symbol which subsumes the other is chosen. If two renaming symbols mutually subsume, then the analysis must be repeated with each renaming symbol in the disputed position. Cycles of renaming symbols do exist in the pattern families we have studied, but members of cycles very rarely occupy the same position in related patterns.

The cost analysis works by verifying that the conversion from the given choice to a possible connection (pattern or renaming symbol) is part of the minimum cost conversion of the representative tree to the choice. This is done by comparing the cost of converting the tree to the choice and the following sum: the cost of converting the tree to a possible connection plus the cost of converting the connection to the choice. If they are the same, the connection is part of a minimum cost conversion.

The cost of a conversion can be computed from the cost data determined earlier. Either the cost of a conversion is stored directly, or it can be computed by repeatedly decomposing the two inputs and summing the costs of converting the respective children. If the cost of the conversion between a tree and a renaming symbol is being computed, then all patterns which subsume the renaming symbol are examined; and the minimum cost, of converting the tree to one of the patterns and converting the pattern to the required renaming symbol, is returned.

Finally, the table *makeindex* is built by looking at each label, and each possible use of it as the label of a child of each operator, and seeing if one of the members of the label can be used as an index optimally in all possible situations. A possible index is validated by looking at what label will result for the parent node if the index is used and then seeing if the *matchsettab* entry for that label is still valid. The *matchsettab* entry is validated with the same sort of cost analysis done when the *matchsettab* was originally built, but now more (lower) context is known (a bigger representative tree can be built). This analysis is complicated (and slowed) by the fact that, for operators with arity greater than one, the index being validated may appear in many different contexts (determined by the indices for the other children). To speed this analysis we currently assume that the proper choice of an index can be made independent of the choice of indices for other siblings.

If the choice of an index causes a pattern to be dropped from the parent's label (some information from lower in the tree is purposely thrown away to guide the algorithm to make an optimal choice higher in the tree), and the dropped pattern appears as the subpattern of another pattern, then similar analysis is repeated to validate the grandparent's *matchsettab* entry. If the choice causes a pattern to be dropped which is the subpattern of a subpattern (or worse), the analysis prints a warning which indicates that optimality cannot be guaranteed in this instance. Our experience to date indicates that this two level validation is sufficient.

The grandparent's entry is also validated if the resulting parent's label contains a subpattern and information about the down context will not be propagated up the tree at this point (ie some subtree must be renamed). This check is needed to detect a situation where two larger patterns in conjunction will match a tree more cheaply than a sequence of smaller patterns. Note that if there is not a subpattern in the parent's label, then there must be a renaming applied at that level—there are no smaller patterns which could be mistakenly chosen. Note also that, if there is no loss of information concerning the down context, then the *matchsettab* analysis is able to detect that the two larger patterns are preferable to a sequence of smaller patterns.

```
initialize all entries in $matchtab$ to empty
for each pattern $p = op\ ch_1 \ldots ch_{arity(op)}$ do
  for each tuple $[op][ch'_1] \ldots [ch'_{arity(op)}]$ where, for $1 \leq i \leq arity(op)$, $ch'_i$ subsumes $ch_i$ and
    there exists patterns $q_i = op\ ch''_1 \ldots ch''_{arity(op)}$ such that $ch'_i = ch''_j$ with $i = j$
      append $p$ to $matchtab[op][ch'_1] \ldots [ch'_{arity(op)}]$
  end
end
```

Figure 3 Building *matchtab*

```
for each label $\ell$ do
  $closure := \emptyset$
  for each pattern $p$ in $\ell$ do
    $closure := closure \bigcup \{p\}$
    for each pattern or renaming symbol $q$ such that $p$ subsumes $q$ do
      $closure := closure \bigcup \{q\}$
    end
  end
  for each renaming symbol $x \in closure$ do
    $xlist$ := empty
    for each pattern specification $z : x \rightarrow y$ such that $y \in closure$ do
      append $z$ to $xlist$
    end
    if size($xlist$)> 1 then
      $T := buildRtree(\ell)$ /* build tree representing the set of trees matched by $\ell$ */
      choose $z'$ from $xlist$ such that $computecost(T \rightarrow^* x) = computecost(T \rightarrow^* z'.rhs) + z'.cost$
        /* cost is computed by decomposing trees until base costs,
           determined when discovering subsumption realtionships,
           can be applied and then costs of subtrees are summed */
    else
      $z' := xlist[1]$
    end
    $matchsettab[\ell][x] = z'$
  end
end
```

Figure 4 Building *matchsettab*

```
for each operator $op$ do
  for $pos := 1$ to $arity(op)$ do
    for each label $\ell$ do
      $closure := \emptyset$
      for each pattern $p$ in $\ell$ do
        $closure := closure \bigcup \{p\}$
        for each $q$ such that $p$ subsumes $q$ do
          $closure := closure \bigcup \{q\}$
        end
      end
      let $x := \{ ch \mid \exists$ pattern $q = op\ ch'_1 \ldots ch'_{arity(op)}$ where $ch = ch'_{pos} \}$
      $y := x \bigcap closure$
      choose a $y' \in y$ such that $validate(y', op, pos, closure) =$ TRUE
      /* check, if $y'$ is used as index, will the resultant matchsettab entry
        (or entries, if the grandparent could be affected) be valid */
      if no such $y'$ exists then
        print error message: optimality cannot be guaranteed
      else
        $makeindex[op][pos][\ell] = y'$
      end
    end
  end
end
```

Figure 5 Building *makeindex*

If no choice of index can be validated, then the system also prints a warning message that states that the pattern family, in its current state, cannot be guaranteed to always be used optimally by the generic code generator. Our experience to date indicates that by adding bigger patterns to the family, the code generator can be coerced to make optimal choices. Our experience will be discussed more fully in a later section.

The argument that the system either guarantees that the patterns will be used optimally, or prints warning messages, essentially mirrors the above discussion. If the code generator outputs suboptimal code, an incorrect choice must be made on pass two. There are three ways an incorrect choice could be made. First, the optimal choice is not even available on pass two—it was a "dropped" pattern, or the ancestor of a "dropped" pattern, during pass one. Second, a *matchsettab* entry is incorrect. Third, a loss of information concerning the down context causes a suboptimal sequence of patterns to be chosen. It is exactly these three possibilities that the analysis addresses. So it simply must be shown that the cost analysis is relevant and correct. The key idea is whether the tree representing a label (built by overlapping the patterns in the label) is an accurate approximation of all possible input trees which could be matched by the label. Again the argument is that, to generate code for a tree, all the subtrees must first be converted to forms matching subpatterns contained within a pattern headed by the root operator—it is these forms that the "overlapped" tree represents. The details of this argument appear in [HAT85b].

What is this analysis *really* doing? It is choosing between independent patterns (the underlying algorithm does not allow them) and it is finding instances where a shorter pattern should be used in place of a bigger pattern (in general the algorithm has a bias towards bigger patterns).

It is important to note that, assuming there is a limited amount of "real" pattern independence, the failure of the preprocessing analysis could not cause the code generator to fail (block or generate incorrect code), rather it causes the code generator to generate suboptimal code. Also, when developing a set of patterns, the analysis can be turned off and a default of always choosing the biggest pattern can be applied. Like in a conventional Graham-Glanville code generator, fairly good code will result, but this system has the extra benefit of never having to worry about the code generator blocking because of a bad choice—when we choose one pattern over another, we know that they both match.

This preprocessing of the patterns is expensive, but it is bounded by a polynomial in the size of the pattern family. The size of the three tables is most greatly dependent on the number of labels found during pass one of the preprocessing. The size of *matchsettab* is bounded by the number of labels times the number of renaming symbols in the pattern family. The size of *makeindex* is bounded by the number of operators times the number of labels raised to the power equal to the maximum arity of the operators. The size of *matchtab* is bounded by the number of operators times the size of the pattern family raised to the power equal to the maximum arity of the operators. Of course, the number of labels is bounded by the size of *matchtab*; but, in practice (see below), it is equal to a small multiple of the size of the pattern family.

Experience With DEC VAX-11

We have prototyped our system using the Icon programming language [GRI83]. The prototype has been used to analyze a family of patterns describing thirty-two bit integer arithmetic on the DEC VAX-11 with cost fields based upon the lengths of instructions. As Henry described [HEN84], this type of "kernel grammar" contains the interesting cases because of the many different ways to do integer arithmetic via both instructions and addressing modes. In general, the bulk of the patterns in a complete family are uninteresting because they represent one-to-one translations (if there is only one choice, it is hard to choose incorrectly!). We felt the VAX would be an interesting machine to study because of its rich set of addressing modes.

We have been able to obtain a set of patterns which our system will guarantee will be used optimally by the code generator. We did need to add patterns to the family to more fully describe the VAX index addressing mode (interestingly, Henry reported that all his difficulties with suboptimal code revolved around the VAX index addressing mode [HEN84]). Our system could not originally guarantee optimality in situations where index mode could be used via loading constants or variables into registers (not situations where a subtree would "normally" be computed into a register). In particular, sometimes it is cheaper to scale the index with an explicit multiply rather than to allow the index addressing mode to perform the scaling implicitly. Our system, with the initial patterns, could not build the tables so that this situation would always be handled correctly. This problem was overcome by adding patterns to separate the two cases and to instruct the code generator exactly what to do in the two cases. This increased the size of the pattern family by thirty-three percent, but this percentage is made more dramatic by the fact that we are working with a "kernel" pattern family.

Even when the patterns are in optimal form, the system still prints some warnings. The system worries about the addition and multiplication of the constant zero (because of the CLRx instruction), but we assume that we will not receive input for such a case.

A interesting phenomenon, which occurred while working with these patterns, was that often, when studying warning messages generated by the system, we found the real problem was incorrect or missing patterns. The system would complain when we made a mistake in specifying the patterns! We do not fully understand this phenomenon, but feel that it is worth further study.

In a pattern family with no pattern independence, the number of labels equals the size of the pattern family. In our VAX description we found that the number of labels to be roughly twice the size of the pattern family. This gives a rough measure of the amount of independence present. Also, as described above, this figure is the key to the sizes of the three code generation tables.

Shortcomings and Required Future Work

This work is obviously just a starting point, not the final word. First, at this point we only have a prototype implementation. To experimentally judge the practicality of this method we need to have an efficient implementation. Also, we need to see if it is practical when given a full pattern family rather than just a kernel description. Both of these will require considerable engineering efforts. For instance, it may be possible to only do the extensive analysis on the kernel patterns, and then merge in the bulk of the patterns. Also, as Henry noted [HEN84], the patterns can in fact be generated from a condensed description. This description makes note of pattern commonalities: the commutativity of operators, common cost derivations, similarities by datatypes, etc. It may be possible to do the analysis directly from this type of condensed description. As it is now, the prototype repeats similar analysis over and over.

Second, will it always be possible to coerce pattern families into forms which will be used optimally by the generic code generator? If so, will the coercion always involve simply adding more patterns? Can the coercion process be automated? Clearly, a study of many different machines and their corresponding pattern families is needed.

Third, it would be reassuring to have an algorithm to statically categorize the pattern independence present in a pattern family. In particular, the need exists to be able to tell if the extensive analysis can be bypassed without fear of the code generator blocking. Right now we work under the assumption that this is true, that we can safely use the system (perhaps suboptimally) without the extensive analysis. But it would be better to have an algorithm that would verify this for a given pattern family, and do it more cheaply than running the full analysis.

Finally, we have chosen to use the fully syntactic approach of the conventional Graham-Glanville technique. We have also ignored the issues of register allocation and common subexpressions. Much study is needed to decide how to best use this algorithm in a real compiler which must address all these issues. We feel that our algorithm will give us the freedom to approach these problems from many different angles and does not tie us down to one specific approach.

As an aside: we have run experiments with our system analyzing patterns describing Intel's 8086 architecture. This architecture does not appear amenable to postponing register decisions until after instruction selection; but if you do, the pattern family is considerably simpler than the corresponding VAX pattern family. In fact, we did not need to add any patterns to the family in order for our system to guarantee that the code generator would use the patterns optimally. Also the number of labels was again twice the size of the pattern family. But, more to the point, we observed that choosing instructions based simply on instruction length resulted in "optimal" sequences which used many more registers than corresponding "suboptimal" sequences. Perhaps, on machines like this, instructions should be chosen based on a combination of program size and register usage—maybe then it will be adequate to postpone register decisions. In any event, much more study is required before our system is feasible in the setting of target machines like Intel's 8086.

Conclusion

We have designed, prototyped, and tested a code generator generator system, which works from Graham-Glanville-like patterns, but uses a bottom-up tree pattern matching algorithm to apply the patterns to an input tree. Our system can not build its tables in linear time, but, unlike Aho and Ganapthi's system, it can generate optimal code without resorting to dynamic programming. We feel that, if a trade-off can be made between code generator generation time and code generation time, we are willing to slow the code generator generator to speed up the code generator. Moreover, our system can work (perhaps suboptimally) without the expensive preprocessing of the patterns.

We have demonstrated our system's capability to generate optimal code (ignoring register allocation and common subexpressions) for the DEC VAX-11 with its rich set of addressing modes. In the process of this experiment our system demonstrated the capability to detect its own weaknesses—allowing the compiler-writer to massage the pattern family into an optimal form. This ability to detect the sources of suboptimal code is not present in conventional Graham-Glanville systems. The original system of Ganapathi allowed predicates which could be used to guide the system to making the proper choices, but the compiler-writer was still faced with the difficult task of forseeing where suboptimal code might arise.

Bibliography

[AHO85] Aho, A. and Ganapathi, M. Efficient Tree Pattern Matching: an Aid to Code Generation. *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, January 1985, pp. 334-340.

[AIG84] Aigrain, P., Graham, S., Henry, R., McKusick, M. and Pelegrí-Llopart, E. Experience with a Graham-Glanville Style Code Generator. *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, Montreal, Canada June 17-22, 1984, *SIGPLAN Notices* **19**, June 1984, pp. 13-24.

[CAT78] Cattell, R. *Formalization and Automatic Derivation of Code Generators*. PhD Dissertation, Carnegie-Mellon University, 1978.

[CAT80] Cattell, R. Automatic Derivation of Code Generators from Machine Descriptions. *TOPLAS* **2**, April 1980, pp. 173-190.

[CHR84] Christopher, T., Hatcher, P. and Kukuk, R. Using Dynamic Programming To Generate Optimized Code in a Graham-Glanville Style Code Generator. *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, Montreal, Canada June 17-22, 1984, *SIGPLAN Notices* **19**, June 1984, pp. 25-36.

[GAN80] Ganapathi, M. *Retargetable Code Generation and Optimization Using Attribute Grammars*. PhD Dissertation, Univ. of Wisconsin - Madison, 1980.

[GAN81] Ganapathi, M. and Fischer, C. A Review of Automatic Code Generation Techniques. Technical Report #407, University of Wisconsin - Madison, 1981.

[GAN82] Ganapathi, M., Fischer, C. and Hennessy, J. Retargetable Compiler Code Generation. *Computing Surveys* **14**, December 1982, pp. 573-592.

[GLA77] Glanville, R. *A Machine Independent Algorithm for Code Generation and Its Use in Retargetable Compilers*. PhD Dissertation, Univ. of California, Berkeley, 1977.

[GRA82] Graham, S., Henry, R. and Schulman, R. An Experiment in Table Driven Code Generation. *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, Boston, Mass. June 23-25, 1982, *SIGPLAN Notices* **17**, June 1982, pp. 32-43.

[GRI83] Griswold, R. and Griswold, M. *The Icon Programming Language*. Prentice-Hall, Englewood Cliffs, New Jersey, 1983.

[HAT83] Hatcher, P. and Christopher, T. Using Aho and Johnson's Linear Dynamic Programming Code Generation Algorithm in a Table-Driven Code Generator. Technical Report #49-83, Department of Computer Science, Illinois Institute of Technology, 1983.

[HAT85a] Hatcher, P. CGG User's Guide (Versions 1.1 & 2.0). Technical Report #49-85, Department of Computer Science, Illinois Institute of Technology, 1985.

[HAT85b] Hatcher, P. *A Tool For High-Quality Code Generation*. PhD Dissertation, Illinois Institute of Technology, 1985.

[HEN84] Henry, R. *Graham-Glanville Code Generators*. PhD Dissertation, Univ. of California, Berkeley, 1984.

[HOF82] Hoffmann, C. and O'Donnell, M. Pattern Matching In Trees. *JACM* **29**, January 1982, pp. 68-95.

[HOR85] Horspool, R. and Scheunemann, A. Automating the Selection of Code Templates. *Software-Practice And Experience* **15**, May 1985, pp. 503-514.

[JOH78] Johnson, S. A Portable Compiler: Theory and Practice. *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, January 1978, pp. 97-104.

[KRU82] Krumme, D. and Ackley, D. A Practical Method for Code Generation Based on Exhaustive Search. *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, Boston, Mass. June 23-25, 1982, *SIGPLAN Notices* **17**, June 1982, pp. 185-196.

[LEV80] Leverett, B., Cattell, R., Hobbs, S., Newcomer, J., Reiner, A., Schatz, B. and Wulf, W. An Overview of the Production-Quality Compiler-Compiler Project. *Computer* **13**, August 1980, pp. 38-49.

[SCH84] Schmidt, U. and Völler, R. A Multi-Language Compiler System with Automatically Generated Code Generators. *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, Montreal, Canada June 17-22, 1984, *SIGPLAN Notices* **19**, June 1984, pp. 202-212.

[TAN83] Tanenbaum, A., van Staveren, H., Keizer, E. and Stevenson, J. A Practical Tool Kit For Making Portable Compilers. *CACM* **26**, September 1983, pp. 654-660.