Efficient Tree Pattern Matching: an Aid to Code Generation

(Extended Abstract)

Alfred V. Aho
AT&T Bell Laboratories,
Murray Hill, New Jersey

Mahadevan Ganapathi
Stanford University,
Stanford, California

Abstract

We show that tree pattern matching has significant advantages in the specification and implementation of efficient code generators. We present a top-down tree-matching algorithm that is particularly well suited to code generation applications. Finally, we present a new back-end language that incorporates tree pattern matching with dynamic programming into a uniform framework for the specification and implementation of efficient code generators.

1. Introduction

In the last decade research in code generation has yielded fundamental theoretical insights and promising practical approaches [Ganapathi, Fischer and Hennessy, 1982, Lunell, 1983]. On the theoretical front, efficient algorithms for generating provably optimal code on a broad class of uniform register machines have been developed for expressions with no common subexpressions [Sethi and Ullman, 1970; Aho and Johnson, 1976]. Once common subexpressions are encountered, or optimal code needs to be generated for machines with irregular architectures, then the problem of optimal code generation has been proven to be combinatorially difficult [Bruno and Sethi, 1976; Aho, Johnson and Ullman, 1977a], and heuristic techniques for generating good code have been theoretically analyzed [Aho, Johnson and Ullman, 1977b].

On the experimental front, several innovative approaches to retargetable code generation have been pursued. These approaches have focussed on the use of table driven techniques to separate the machine description from the code generation algorithm [Cattell, 1978; Fraser, 1977; Ganapathi and Fischer, 1982, 1984b; Glanville and Graham, 1978, Graham, 80; Henry, 1984; Johnson, 1978; Wulf, 1980]. Compilers based on some of these techniques have been shown to be fairly easily retargeted when compared to their monolithic counterparts [Johnson, 1978].

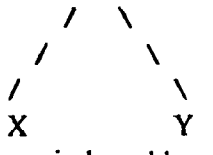
In this paper we present a language that encapsulates some of these theoretical and experimental advances into a single framework for describing and implementing code generators. The language builds on the experience of grammar and attribute-grammar based descriptions of code generators. It also incorporates recent advances in tree pattern matching technology along with the dynamic programming algorithm of [Aho and Johnson, 1976] for generating locally optimal code for expression trees in linear time for a broad class of register machines. As we shall see, it significantly facilitates the description of Ganapathi-Fischer style code generators.

2. Pattern-Directed Code Generation

The major technique adopted in the retargetable approaches is to replace virtual machine interpretation by pattern matching. Wasilew [1972] and Weingart [1973] were among the first to propose this idea. The target machine instructions are represented as individual tree patterns. The input to the code generator is a tree representation of the source program. The code generator matches the input tree and on each match, outputs target code.

For example, consider the following machine-patterns for instruction-selection:

PI: Address → index



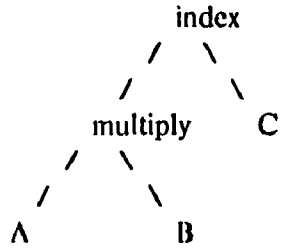
use index addressing mode X[Y]

PM: Word → multiply

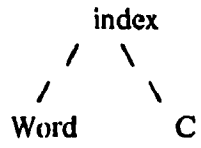
emit IMUL X, Y, Temp

Now, consider the following input to the code generator. The code generator is being asked to first select addressing modes for operands, then select an op-code and then once again select an addressing mode.

IR:



The code generator matches the multiplication pattern (PM) first, the instruction IMUL A, B, Temp is emitted and the input tree is reduced to:



Next, the code generator matches the index pattern (PI), an index addressing mode is used and the IR tree is consumed.

The early pattern matching techniques have employed direct tree-pattern matching techniques. Fraser [1977] and Cattell [1978] have emphasized the use of heuristic search. Fraser relies on knowledge-based rules that direct pattern matching whereas Cattell suggests a goal-directed heuristic search. In Cattell's approach, subgoals are created as the search continues. Heuristics are used, both to order subgoal selection and also to order patterns when trying to match.

Later techniques have linearized the input tree to the code generator. The target machine is described in the form of grammar productions. Thus, pattern matching in trees is implicitly provided with the help of bottom-up parsers [Glanville, 1977; Ganapathi, 1980]. With each reduction step of the parser, the input linearized-tree gets smaller. In [Glanville, 1977], the target instructions are represented by context-free grammars. Every possible instruction variant is described by a grammar rule. Pattern matching is then provided by simple SLR parsing. It is a purely syntactic approach to the instruction-selection problem. The *tree-pattern-matching* is provided in a completely **left-operand biased** fashion. That is, when generating code for an entire sub-tree, the code for the left operand is selected without considering the right operand. Needless to mention, this yields suboptimal code in many cases.

For example, consider the string *op A B*. The addressing mode for A is selected without seeing B. Thus, A could be a *register-indirect* addressing mode on the iAPX-286[1]. Next, B happens to be a memory datum that gets one of the memory addressing modes. Now, comes the time to select a machine op-code. The code generator realizes that memory-to-memory operations cannot be performed in one instruction. Thus, it is forced to move A to a register.

Furthermore, the syntactic approach to instruction-set specification is inadequate on most real machines. Neither is there a mechanism to specify architectural restrictions on the programming model such as, register restrictions on addressing-mode use nor is there a mechanism to track multiple results such as, results in multiple locations and condition-code setting. Furthermore, syntactic treatment of semantics yields a very large number of grammar

[1] iAPX is a trademark of Intel Corporation

productions, unacceptably large for of-the-shelf parser generators. Time-consuming and tedious engineering decisions are required on the grammar to make the scheme partially workable [Henry, 1984].

In [Ganapathi, 1980], the target instruction-set is specified by attribute grammars instead of context-free grammars. Semantic attributes and predicates provide automated semantic handling. Predicates are used to specify architectural restrictions on the programming model. Attributes are used to track multiple instruction results. Instruction selection is done by attributed parsing [Ganapathi and Fischer, 1982, 1984a]. Addressing modes are described by separate individual productions and so are op-codes. Addressing-mode selection is left-biased in the true tree-pattern matching sense, but selection of op-codes in not biased toward any operand. Opcode productions have symmetric operand patterns. This symmetry enables the code generator to delay decisions regarding destination requirements. In effect, this decision is made on seeing the entire sub-tree for the operator. Thus, efficient code is produced in cases when either of the operands can be used to store the result of evaluation. Only in cases where their original results need be preserved is a call made to the register/temporary allocator.

For example, consider the iΛPX-286 instruction-set architecture. Only five of the eight general-purpose registers are available for offset address calculations, i.e., indexing. So, the indexing pattern PI is qualified by a predicate expression that specifies these architectural restrictions on the programming model. The symbol ↑ attaches attributes to grammar symbols.

PI-iΛPX286:

Address↑a → index

Datum↑X Datum↑Y

predicates:
  X = 16-bit constant and
  Y = SI, DI, BP, BX or SP register
action:
  use index addressing mode X[Y]
  the attribute 'a' specifies this mode

Consider three-address, two-address and single-address multiplication on the iΛPX-286 that may be selected during instruction-selection.

PM-iΛPX286:

Word↑Temp → multiply

Datum↑X Datum↑Y

predicate:
  X = 16-bit constant
action:
  emit IMUL Temp, Y, X

Word↑Y → multiply

Datum↑X Datum↑Y

predicates:
  X = 8-bit constant
  Y = 16-bit register ∧ Y = ¬ busy
action:
  emit IMUL Y, X

Word↑ΛX,DX → multiply

Datum↑X Datum↑Y

predicates:
  X = 16-bit operand
  Y = ΛX register ∧ Y = ¬ busy
action:
  emit IMUL X

The use of attributes and predicates allows the incremental development of a code generator. Initially, the most general form of production use is listed. Later, special-case productions are added to improve the performance of the target code. In the above example, we could incorporate two-address and three-address multiplies at a later stage. These

special-case productions may be added even though the grammar becomes ambiguous. Ambiguous grammars are useful in the specification of code generators because subsequent modifications can be performed with reduced effort. Parsers can be constructed from ambiguous specifications [Aho, Johnson and Ullman, 1975]. YACC [Johnson, 1975] is a parser generator that allows ambiguous specifications.

Furthermore, implicit factoring of addressing modes by use of addressing-mode productions and semantic attributes, replaces Glanville's cross-product of op-code and addressing modes by their additive sum. Consequently, the number of grammar productions is considerably smaller and practical (several hundred productions instead of several thousand). Extensive grammar engineering is not needed to implement a code generator based on semantic attributes and predicates.

In this paper, we present a tree pattern matching algorithm for use in code generation. It eliminates the deficiencies pointed out above. In addition to its ability to defer decisions and produce locally optimal code, the following advantages also accrue:

1. The underlying tree-automaton based code generator can be constructed more quickly than a bottom-up parser or attributed-grammar based parser. In code generation, only the **string pattern matching** capabilities are needed; the **bottom-up synthesis** part provided by LR parsers is not really needed.
2. Although the method used in [Ganapathi and Fischer, 1982] provided succinct and concise grammar descriptions, it was observed that many grammar productions had considerable syntactic similarity. They differed only in attributes and predicates. Our aim in replacing an attributed-parser by a **tree automaton** is to factor the repeated syntactic parts into one common pattern, so that one syntactic match can correspond to several concurrent matches of target machine instructions. As an additional benefit, the time and effort required to write description patterns for such an automaton is considerably less when compared to [Ganapathi and Fischer, 1982].
3. Since fewer patterns are needed, the description of the code generator is significantly simplified. A simple experiment showed that the description of the code generator for the iAPX-86 was cut almost in half. In addition, the original predicate-free grammar-based description has 450 shift/reduce and 5000 reduce/reduce conflicts. The tree pattern matching description eliminates the need to deal with these parsing action conflicts completely.
4. One of the restrictions in [Ganapathi and Fischer, 1982] was the use of inherited attributes to non-terminals in the grammar. In particular the left-hand side of a grammar production cannot have any inherited attribute. This restriction is due to the one-pass bottom-up code generation scheme. While for non-optimizing machine-code generators, there does not seem to be a need for inherited attributes on the left-hand side, such an allowance could be beneficial for optimization. For example, the target path of an operation can be passed as an inherited attribute to the left-hand side. At present, the target is figured out by examining the left context provided on the attribute stack of the parser. In contrast, this information could be automatically made available as an inherited attribute of the left-hand side. In the tree-automaton based code generator, the pattern matching is performed by a preorder traversal of the intermediate language tree. Consequently, inherited attributes can be easily computed in the top-down pass.
5. The linear time dynamic programming algorithm of [Aho and Johnson, 1976] has proven effective in practical code generation [Johnson, 1978; Ripken, 1978]. This scheme can be readily integrated into the tree matching process. The dynamic programming algorithm guarantees locally optimal code for expression trees, an advantage not enjoyed by the current grammar-based code generators. Furthermore, the incorporation of the dynamic programming algorithm eliminates the necessity for explicitly breaking cycles to prevent the code generator from looping.

Consider the intermediate representation for indexed multiplication given above. We compare the Graham-Glanville, the Ganapathi-Fischer and the current tree-automaton techniques in generating code for this intermediate representation.

Graham-Glanville cannot model predicate restrictions on the use of the index addressing mode and the IMUL op-code for the iAPX-286 architecture.

The Ganapathi-Fischer code generator will first match the three-address multiplication production

(PM-iAPX286). Then the temporary allocator is invoked. The attribute-stack provides information regarding the context of operation, i.e., whether the operator is a *multiply* and the operands are integer operands, more precisely, word data-types. The temporary allocator, which in this case is the register allocator, returns the AX-DX register pair. The three-address multiplication op-code is emitted and the result is the AX-DX pair that is propagated as a synthetic attribute to the left-hand side grammar symbol.

Next, the index production is selected. Because of predicate restrictions on use of this addressing mode, code is generated to move the contents of AX register to an indexable register, say SI in this case. Subsequently, the index production is selected.

If in the first step the temporary allocator had allocated an indexable register in the first place, then the redundant register move in the second step could have been avoided. In the tree pattern matching automaton, this redundant move is avoided.

3. Tree Pattern Matching Algorithms

A number of tree pattern matching algorithms have been recently proposed for different applications [Kron, 1975; Hoffman and O'Donnell, 1982; Huet and Levy, 1979; Lang, Schimmler and Schmeck, 1980]. We have found a generalization of the string matching algorithm of [Aho and Corasick, 1975] for top-down tree pattern matching well suited for our code generation application. As suggested in [Hoffman and O'Donnell, 1982], each tree pattern can be characterized as a set of root-to-leaf path strings. A string pattern matching automaton can then be constructed from these sets of characterizing path strings.

The pattern matching algorithm uses the automaton to partition an expression tree into a set of subtree matches by performing a preorder traversal of the target tree. A dynamic programming algorithm is run concurrently with the matching to select a minimal cost partition.

Our tree matching algorithm has several advantages for code generation purposes. The construction of the pattern matcher can be done in time linear in the size of the tree pattern specifications. The size of the resulting pattern matcher is also a linear function of the size of the input specification. Neither of these claims can be made for the SLR parser construction algorithm (or for a purely bottom-up style of tree matching). Practical experience with the Aho-Corasick algorithm for string matching shows that thousands of string patterns can be matched simultaneously with a running time that is comparable to the UNIX[2] system grep command looking for a single-string pattern.

4. A Pattern-Directed Code Generation Language

In this section we sketch the design of the code generator language CGL that incorporates the tree pattern matching and dynamic programming language into a uniform framework for specifying code generators. The language is modeled after a production system, as is the parser-generator language YACC. A CGL program is basically a sequence of pattern-action statements. Each pattern is a tree-rewriting rule of the form $A \rightarrow \alpha$ where A is a node and α a tree template. Each action is a sequence of guarded commands, like Dijkstra's language but without nondeterminism. Each guard is a predicate whose variables are inherited and synthesized attributes. Each command is a program fragment, like a YACC action. Attached to each command is a cost that is used in the dynamic programming algorithm to select locally optimal code. The pattern-action statement works in the following manner. If the pattern α is found in the input tree, then the pattern is rewritten as an A, provided that some predicate in the action is satisfied. The command associated with the satisfied predicate is executed. If more than one pattern and predicate match, then the cost associated with each command is used by the dynamic programming algorithm to determine which command to execute. An omitted cost is assumed to be unity.

Here is an example showing the iAPX-286 multiplication productions written in CGL:

[2]Unix is a trademark of Bell Laboratories.

Word↑R → multiply Word↑X Word↑Y

```
predicate:
  X = 16-bit constant;
action:
  emit "IMUL Temp, Y, X";
  R := Temp;
cost[3]: 4, 24

predicate:
  X = 8-bit constant
  Y = 16-bit register ∧ Y = ¬ busy;
action:
  emit "IMUL Y, X";
  R := Y;
cost: 3, 24

predicate:
  X = 16-bit operand
  Y = ΛX register ∧ Y = ¬ busy
action:
  emit "IMUL X";
  R := ΛX,DX;
cost: 2, 16
```

An experimental version of this code generation language called TWIG has been implemented by Tjiang [1984]. Preliminary results indicate that the performance of a TWIG generated code generator is superior to that of the code generator, Pcc2, of the Portable C compiler [Johnson, 1978].

5. Conclusions

We have shown that tree pattern matching can be used to eliminate several major difficulties with grammar and attribute-grammar based approaches to code generation. We have isolated a tree pattern matching algorithm that seems well suited for use in code generation. A code generation language that encapsulates the tree matching and dynamic programming algorithms has been designed and implemented. It can be used to describe Ganapathi-Fischer style code generators with considerable ease.

[3]The cost takes into account the space required by the instruction, and the number of clock cycles that this form of instruction takes to execute: i.e., the time required to calculate an operand's effective address, the internal processing overhead in clock cycles and the time needed to read or write a memory operand.

References

1. A. V. Aho and M. J. Corasick, "Efficient string matching: an aid to bibliographic search", Comm. ACM, 18, 6, June 1975, 333-340.
2. A.V. Aho, S.C. Johnson and J.D. Ullman, "Deterministic Parsing of Ambiguous Grammars", Comm. ACM 18(8), 1975.
3. A. V. Aho and S. C. Johnson, "Optimal code generation for expression trees", J. ACM, 23, 3, 1976, 488-501.
4. A. V. Aho, S. C. Johnson and J. D. Ullman, "Code generation for expressions with common subexpressions ", J. ACM, 24, 1, 1977, 146-160.
5. A. V. Aho, S. C. Johnson and J. D. Ullman, "Code generation for machines with multiregister operations", Fourth ACM Symposium on Principles of Programming Languages, 1977, 21-28.
6. A. V. Aho and J. D. Ullman, "Principles of Compiler Design", Addison-Wesley, Reading MA, 1977,
7. J. Bruno and R. Sethi, "Code generation for a one-register machine", J. ACM, 23, 3, 1976, 502-510.
8. R.G.G. Cattell, "Formalization and Automatic Derivation of Code Generators", PhD dissertation, Carnegie-Mellon University, April 1978.
9. C.W. Fraser, "Automatic Generation of Code Generators", PhD dissertation, Computer Science Department, Yale University, New Haven, Connecticut, July 1977.
10. M. Ganapathi, "Retargetable Code Generation and Optimization using Attribute Grammars", PhD dissertation, Technical Report #406, University of Wisconsin - Madison, 1980.
11. M. Ganapathi and C.N. Fischer, "Description-Driven Code Generation Using Attribute Grammars", Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, January 25 - 27, 1982.
12. M. Ganapathi, C.N. Fischer and J.L. Hennessy "Retargetable Compiler Code Generation", ACM Computing Surveys, Vol. 14, No. 4, December 1982.
13. M. Ganapathi and C.N. Fischer, "Instruction Selection by Attributed Parsing", Technical

report #256, Computer Systems Laboratory, Stanford Electronics Laboratories, Departments of Electrical Engineering and Computer Science, Stanford University, February 1984. (Also, *to appear in ACM Transactions on Programming Languages and Systems*).

14. M. Ganapathi and C.N. Fischer, "Attributed Linear Intermediate Representations for Retargetable Code Generators", Software - Practice and Experience, Vol. 14, April 1984.
15. R.S. Glanville, "A Machine Independent Algorithm for Code Generation and its Use in Retargetable Compilers", PhD dissertation, University of California, Berkeley, December 1977.
16. R.S. Glanville and S.L. Graham, "A New Method for Compiler Code Generation", Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, pp. 231-240, January 1978.
17. S.L. Graham, "Table-Driven Code Generation", IEEE Computer, Vol. 13 No. 8, pp. 25-34, August 1980.
18. R.R. Henry, "Graham Glanville Code Generators", PhD Dissertation, Computer Science Division, EECS, University of California, Berkeley, 1984.
19. C.M. Hoffman and M.J. O'Donnell, "Pattern Matching in Trees", J. ACM 29, 1, 1982, 68-95.
20. G.Huet and J.-J. Levy, "Call by need computations in non-ambiguous linear term rewriting systems", TR 359, IRIA Laboria, LeChesnay, France, 1979.
21. S.C. Johnson, "YACC - Yet Another Compiler Compiler", Computer Sciences Technical Report #32, Bell Telephone Laboratories, Murray Hill, New Jersey, 1975.
22. S.C. Johnson, "A Portable Compiler: Theory and Practice", Proc. 5th ACM Symp. Principles of Programming Languages, pp. 97-104, January 1978.
23. H. Kron, "Tree Templates and Subtree Transformational Grammars", PhD Dissertation, University of California, Santa Cruz, 1975.
24. H.-W. Lang, M. Schimmler and H. Schmeck, "Matching Tree Patterns sublinear on the average", Technical Report, Department of Informatik, University of Kiel, Kiel, West Germany, 1980.
25. H. Lunell, "Code Generator Writing Systems", Software Systems Research Center, S-58183, Linkoping, Sweden, 1983.
26. K. Ripken, "Formale Beschreibun von Maschinen, Implementierungen und Optimierender Maschinen-codeerzeugung aus Attributierten Programmgraphe", TUM-INFO-7731, Institut fur Informatik, Technische Universitat Munchen, Munich, West Germany, July 1977.
27. R. Sethi and J. D. Ullman, "The generation of optimal code for arithmetic expressions", J. ACM, 17, 4, 1970, 715-728.
28. S. Tjiang, Private communication, October 1984.
29. S.G. Wasilew, "A Compiler Writing System with Optimization capabilities for Complex Order Structures", PhD thesis, Northwestern University, 1972.
30. S.W. Weingart, "An Efficient and Systematic Method of Compiler Code Generation", PhD dissertation, Computer Sciences Department, Yale University, 1973.
31. W. Wulf, B. Leverett, R. Cattell, S. Hobbs, J. Newcomer, A. Reiner and B. Schatz, "An Overview of the Production Quality Compiler-Compiler Project", IEEE Computer Vol. 13 No. 8, pp. 38-49, August 1980.
32. W.A. Wulf, "PQCC: A Machine-Relative Compiler Technology", IEEE 4th International COMPSAC Conference, pp. 24 - 36, Chicago, October 1980.