Automatic Derivation of Code Generators from Machine Descriptions

R. G. G. CATTELL
Carnegie-Mellon University

Work with compiler compilers has dealt principally with automatic generation of parsers and lexical analyzers. Until recently, little work has been done on formalizing and generating the back end of a compiler, particularly an optimizing compiler. This paper describes formalizations of machines and code generators and describes a scheme for the automatic derivation of code generators from machine descriptions. It was possible to separate all machine dependence from the code generation algorithms for a wide range of typical architectures (IBM-360, PDP-11, PDP-10, Intel 8080) while retaining good code quality. Heuristic search methods from work in artificial intelligence were found to be both fast and general enough for use in generation of code generators with the machine representation proposed. A scheme is proposed to perform as much analysis as possible at code generator generation time, resulting in a fast pattern-matching code generator. The algorithms and representations were implemented to test their practicality in use.

Key Words and Phrases: code generator generator, compiler compiler, machine description, optimizing compiler, code generation, compiler generator, translator writing system, computer description
CR Categories: 4.12

1. INTRODUCTION

In the past decade, there has been increasing interest in reducing the effort to construct compilers. The problem has become more important as good-quality compilers are required for the increasingly numerous machine architectures made possible through microprogramming and LSI technology. Progress has been made in automatic generation of the parsers that translate source language into internal notation. However, it has proved much more difficult to do the same for the second part of the compilation process: the translation of internal notation into machine code. The work presented here suggests that more general formalizations of machines and code generators are needed to allow the automatic derivation of code generators.

This work necessarily involves relatively disparate areas of computer science: computer architecture, compilers, automatic programming. It is only one of many

This research was supported in part by the Advanced Research Projects Agency of the Office of the Secretary of Defense under Contract F44620-73-C-0074 and is monitored by the Air Force Office of Scientific Research.

Author's present address: Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304.

possible applications of machine descriptions that include emulation of machines, automatic generation of assemblers [32] and diagnostics [23], automated hardware design [3], and this work, which is part of the Production-Quality Compiler Compiler (PQCC) project at Carnegie-Mellon University [18].

The PQCC group is interested in simplifying and/or automating the construction of a high-quality compiler generating optimized code. The work is concentrating on the machine-dependent aspects of optimizing compilers, a difficult problem that has received little attention. PQCC is using the multiple-phase structure of the Bliss-11 compiler [34] as a starting point for the research.

Some work has been done in the area of code generation in general [28, 30, 33]. There have been two classes of approach to simplifying production of code generators. The first is the development of specialized languages for building code generators, with built-in machinery for dealing with the common details; this might be called the *procedural* language approach. Early work in this area was done in compiler writing systems [10, 11, 20, 31]. Also, Elson and Rake [9] and Young [35] have concentrated specifically on code generator specification languages and have been relatively successful. The other extreme is the *descriptive* language approach: automatically building a code generator from a purely structural and behavioral machine description. Miller [21], Donegan [8], Weingart [30], Snyder [29], and Newcomer [22] fit the descriptive language category, to varying degrees. A survey of the above work, particularly as it relates to the goal of automating the production of code generators, can be found in Cattell [5].

More recently, Fraser [12], Glanville [13], Ripken [26], and Johnson [16] have done related work. All of these are concerned with formalizing the code generation process in the sense of separating the code generation algorithms from machine-dependent tables with which they operate, but they differ in the generality of the machine representation and the assistance provided in constructing the tables. Ripken and Johnson propose code generation schemes based on templates mapping program trees onto instructions. Glanville also uses these templates as a machine description, but automatically derives a transition table from them; the resulting table-driven code generator is thereby faster. In this work and in Fraser's, the machine description is more complex and an analysis of the machine is needed to derive the templates. There are therefore two parts to this work, the template-driven code generator and the template-deriving code generator generator. Fraser takes a human-knowledge-based approach to the problem, as opposed to the formal approach taken in this work; these approaches are complementary, with different strengths, providing an interesting contrast of the use of methods from artificial intelligence.

A common objection to general work in the code generation area has been that it has not been practically applicable. In order to demonstrate the feasibility of the ideas, a prototype system of the algorithms and representation proposed here has been implemented.

Figure 1[1] gives an overview of the problem viewed by this work. Three

[1] In the horizontal direction, the construction of a code generator from the machine description is shown. MD is the formal representation of the machine, which could be extracted from a machine description such as ISP. MT is the formal representation of the code generation process; the code generator is table driven by this machine table. The code generator derivation process constructs the

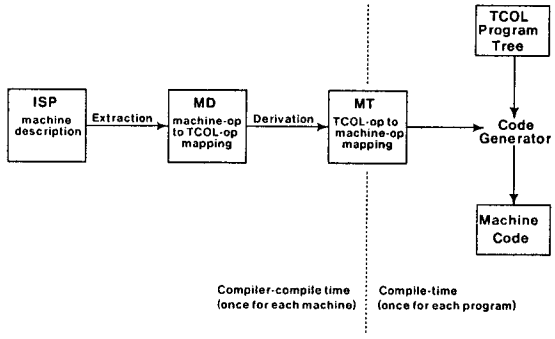

Fig. 1. Relationship of programs and representations proposed.

algorithms and representations are involved:

(1) the formal representation of the machine, labeled MD (machine description) in the figure, and its extraction from a procedural machine description language such as ISP (Bell and Newell [4]);
(2) the tabular representation of the parse-tree to machine-code translation, labeled MT (machine tables) in the figure, and the code generation procedures that use these tables;
(3) the procedures which derive (2) from (1) by heuristic search for optimal code sequences.

These three problems are discussed in Sections 2 through 4 of this paper, respectively.

2. FORMALIZATION OF INSTRUCTION SET PROCESSORS

Before we can deal with code generators or their automatic generation, we must define the class of machines with which we are dealing. This is a crucial step. We want a machine formalization that is sufficiently restrictive to make it possible to generate code with a simple fast algorithm but general enough to include a wide range of typical computer architectures.

We will assume such a machine consists of an *instruction set processor* that iteratively retrieves instructions from a *primary memory* and (conditionally) changes the contents of a set of locations termed the *processor state* as specified by the instruction.

Five main kinds of information are given by the formal machine description:

Storage Bases. These represent the basic storage array(s) of the processor state. Each has a length (the number of words in the array, possibly 1), a width (number of bits per word), and a type. The type essentially specifies how the storage base can be used; it may be *general purpose* (locations that may be used

MT from the MD. In the vertical direction, the use of the code generator itself is shown, translating program trees into code for the target machine. The program tree is produced by a language-dependent compiler front end that translates source code into the tree-based common language (TCOL) notation. Further processing such as peephole optimization may be performed on the machine code output. Note: In Cattell [6], for historical reasons the terms MOP and LOP are used for MD and MT, respectively.

to hold values), *temporary* (condition codes), or *reserved* (locations such as a stack pointer that may not be used to hold values). In addition, two storage bases are distinguished as special in the machine description: the *program counter* (which is type reserved), and the *primary memory* (general purpose).

Operand Addressing. The instructions have particular properties with respect to the kinds of storage base accesses they may make; they differ not only in *which* may be accessed, but in *how*, i.e., the address computation. We define an *access mode* for each distinct type of addressing on the machine, for example, indexed off a register, indirect through a memory location, or an immediate constant from an instruction field. The access mode is described by an expression that represents the access in terms of arithmetic operators and accesses to storage bases; for example, $M[C_2 + R[C_1]]$ designates accessing storage base M by indexing off register C_1 by the constant C_2. An operand of an instruction can generally belong to any of a *set* of access modes depending on opcode, mode bits, etc.; for each such set there is an *operand class.* For example, an ADD instruction might require a general-purpose register as the operand receiving the result and allow either an immediate constant or a memory location as the other operand. Formally, an operand class is a set of tuples consisting of an access mode, its time/space cost (in this context), and a specification of the corresponding instruction field values (e.g., mode bits or address field). The separation of the operand addressing function from the instructions themselves greatly reduces the number of instruction descriptions necessary for machines with a number of addressing modes, as well as simplifies the generation of good code using address mode computations.

Machine Operations. These represent the actual instructions available. For each we need to know cost (space and time), binary formatting information, and a set of *input/output assertions.* The latter describe the effects of the instruction. Each assertion specifies a destination operand class (the location to be modified), an expression over constants and operand classes (the new value of the location in terms of the previous processor state), and a Boolean function (again over the processor state) specifying when the location takes on the new value. For example, an increment-and-skip-on-zero (ISZ) instruction might have two assertions

(1) $R \leftarrow R + 1$

(2) **if** $(R + 1) = 0$ **then** $PC \leftarrow PC + 1$

represented in Algol-like form for readability. (The actual assertions are represented as trees for ease in matching to program trees; throughout this paper the details of the representations are suppressed for the sake of a clearer exposition.) The first assertion always holds; the second contains a Boolean function specifying when the location (the program counter) takes on a new value. R is an operand class that might, for example, represent any one of the machine's general-purpose registers. Note that references to such locations refer to their values before instruction execution, i.e., there is no ordering on the assertions.

Data Types. Machines are normally built around a set of data types. For each one, we need to know the abstract domain (e.g., reals, integers, characters) that it represents and the encoding/decoding function to/from binary bit strings (e.g., 16-bit twos complement). Each arithmetic or logical operator in the instruction assertions specifies the data type(s) on which it operates.

Instruction Fields and Formats. Finally, the machine description must specify the correspondence of the abstract operations and operands to the actual binary encoding. For example, an ADD instruction might have an instruction format with three fields: an opcode, address mode bits, and a displacement field used in the computation of the operand address. The values of these fields are determined by the instruction (e.g., opcode = 17) and its operands (e.g., mode = 1, displacement = 2473) as specified by the binary formatting information associated with machine operations and operand classes. The binary representation description will not be necessary for the purposes of this paper; note that we can ignore this description precisely *because* it has been separated from the rest of the abstract machine.

Note that the machine representation used differs from a procedural machine description language such as ISP due to the structure it requires of the machine and the nonsequential instruction descriptions. It is possible, however, to *derive* our representation from ISP with symbolic simulation and a little help from a human (Oakley [24]).

Note also that the machine representation does *not* say how to generate code for the machine in any way; it essentially specifies a mapping from machine operations to operations of a common semantic notation (TCOL, described in Section 3), and the code generation problem is to invert that mapping.

Space limitations do not allow a complete definition of the components of the machine model here. The interested reader is referred to Cattell [6].

3. FORMALIZATION OF CODE GENERATION

In order to automate the generation of code generators from a machine description, it is first necessary to define what a code generator *is* in a machine-independent manner and to separate good code generation algorithms from the machine-dependent tables they use.

The code generation scheme is based on a form of templates we will refer to as *tree productions*, which are collected in the machine tables (MTs). A given source program is translated into an intermediate parse-tree-like notation (TCOL) by the front end of the compiler. The code generator traverses the program tree, matching each node against *patterns* on the left-hand sides (LHSs) of the productions in the machine tables. When a pattern matches, the right-hand side (RHS) of the production specifies code to be generated, special compiler actions such as allocation, or further matches to be recursively performed. For example, a simple production might be

$$R \leftarrow R + E \Rightarrow \text{ADD R, E}$$

where R and E are operand classes, and the RHS consists of a single instruction (an add) to be emitted (the actual syntax for productions is more complex; see the examples in Section 4.3). This production might be used in generating code for the TCOL tree $X \leftarrow X + 2*Y$: the code generator recognizes that X (allocated to a register, say) can be accessed directly by operand class R, and generates a subgoal to make $2*Y$ accessible via operand class E. The subgoal is of the form $L \leftarrow 2*Y$, where L is an allocated location compatible with E. Had the expression $2*Y$ been of a form which happens to conform to E, say the $M[C_2 + R[C_1]]$ example given in Section 2 (a computation of a contiguous vector element), then

just the single ADD instruction would be generated for the entire assignment statement X ← X + A[I].

A simple example of a production dealing with a control construct is

$$\textbf{if } R = 0 \textbf{ then } S \Rightarrow \text{BNE R, L1; S; L1: } \ldots .$$

This illustrates a recursive call of the code generator (on the statement S) and a compiler-generated label (L1) and also emits an instruction (branch if not equal). More complex productions are needed to deal with constructs such as loops, for example, to take advantage of the ISZ instruction given in Section 2.

Thus we have a simple, machine-independent code generation algorithm, and MTs that specify productions, addressing, and formatting information for the target machine. The strategy used by the basic code generation algorithm (tree traversal and subgoals) is essentially identical to the heuristic search for code sequences used in the code generator generator we discuss in Section 4.2. The difference is that the code generator need only deal with one kind of mismatch between the source tree and pattern tree: the operands must be assigned or moved to locations compatible with the pattern requirements. This register allocation (and associated operand moves) is as essential to every code generator as the actual selection of instructions.

Register allocation is actually done before the code generation in the PQCC compiler by performing a pseudocode-generation pass to determine where storage bases of various types are desirable. Peephole optimization and code output are performed after code generation because certain operations are more conveniently done on a symbolic representation of the object code (e.g., resolving absolute versus short relative addressing, dealing with base registers, and eliminating redundant instructions).

A prototype of the code generator has been built. It appears that code comparable to a good handcrafted compiler (e.g., Bliss-11) can be generated with proper care; experimental results are discussed in Section 5.1.

Further details of the code generator, register allocation, and code optimizations have been deferred to a separate paper on this subject [7], so that we may proceed to the central topic of this paper, the automatic derivation of the code generators.

4. AUTOMATIC DERIVATION OF CODE GENERATORS

In this section we consider the problem of deriving the machine tables that control the code generation process from the machine description (see Figure 1).

One of the central procedures used in the *derivation* of code generators is itself a code generator, specifically one which takes as input a machine description and (heuristically generated) TCOL tree. This machine-independent code generator could be used directly as the code generation phase of the compiler (i.e., use MD directly instead of MT in Figure 1). In practice, however, it is preferable to separate *compile*-time from *compiler compile*-time, to make the code generator as compact and efficient as possible, allowing a thorough analysis of the alternatives at (much less expensive) compiler compile-time. Thus we introduce this extra level of table for efficiency.

As a result the code generator derivation process must be broken into two parts: selection of the special cases to be put into the MT for the derived code

Boolean axioms

not not E ⇔ E
E_1 and E_2 ⇔ not ((not E_1) or (not E_2))
E_1 and E_2 ⇔ E_2 and E_1
E and E ⇔ E

Arithmetic axioms

E + 0 ⇔ E
−(−E) ⇔ E
−E ⇔ 0 − E
E_1*E_2 ⇔ E_2*E_1
E shift 1 ⇔ E*2

Relational axioms

not ($E_1 \geq E_2$) ⇔ ($E_1 < E_2$)
($E_1 < E_2$) or ($E_1 = E_2$) ⇔ $E_1 \leq E_2$

Fetch/store decomposition rules

$E_1(E_2)$ ⇔ S ← E_2; E_1(S)
$S_1 \leftarrow E$ ⇔ $S_2 \leftarrow E$; $S_1 \leftarrow S_2$

Side-effect compensation axioms

S; D ← E ⇔ S if D is temporary state
S; D ← E ⇔ Alloc(D); S if D is general purpose

Sequencing semantics axioms

S_1 ⇔ S_1; PC ← PC + n; S_2⟨space n⟩
if E **then** PC ← PC + n; S⟨space n⟩ ⇔ **if not** E **then** S
PC ← E ⇒ **goto** E (unconditional jump)
goto L_1 ⇔ **goto** L_1 − L_2 + PC; L_2: (relative jump)

Implementation rules

while E **do** S ⇒ L_1: **if** not E **then goto** L_2; S; **goto** L_1; L_2:
if E **then** S_1 **else** S_2 ⇒ **if** E **then goto** L_1; S_2; **goto** L_2; L_1: S_1; L_2:
if E **then** S ⇒ **if** (not E) **then goto** L; S; L:

Notation

L: location in instruction store
D: location in data store
E: combinatorial tree
S: statement (assignment or conditional) tree

Fig. 2. Tree equivalence rules (examples).

generator (the LHSs of the productions), and for each of these, finding the best code sequence for the target machine (the RHSs of the productions). The former is performed by a heuristic procedure we will refer to as Select, the latter by the machine-independent code generator we will refer to as Search. We first discuss the design of Search.

4.1 Tree Equivalence Axioms

The central formalism on which Search is based is a set of axiom schemata that specify semantic equivalences over computations; some examples of these are shown in Figure 2. The axioms express classical arithmetic and Boolean laws, as well as rules about programs and the model of instruction set processors. They will be used to specify the legal tree transformations (programs, instructions, and

axioms are represented as trees) in the heuristic search for optimal code sequences.

The advantage to the use of the axioms and the formal search methods we are about to discuss is that these are almost entirely machine independent and thus only the MD itself need be changed to generate different target code. (However, there are exceptions to this statement: the axioms for the machine's particular binary representation of integers must be used, and it may be necessary to add axioms to reflect new data types/operations provided by the machine.)

The arithmetic and Boolean axioms are relatively straightforward: they include laws such as commutativity of **and** and +, DeMorgan's law, the double-complement rule, and the idempotence of adding zero.

The remaining axioms (see Figure 2) were developed specifically for this work and require some explanation. Note, for example, the axiom labeled Fetch Decomposition:

$$E_1(E_2) \Leftrightarrow S \leftarrow E_2;\ E_1(S)$$

This states that an expression E_1 with a subexpression E_2 can be computed by first computing E_2 and storing the result in a location S, then replacing E_2 with S in the computation of E_1. This essentially says that *storage may be used for temporary results.* The companion axiom Store Decomposition is simply a special case in which E_1 is an assignment statement; this case is treated separately because of the way the search works.

Other axioms deal with *side effects.* If an instruction has more than one assertion, it may be possible to use it for a *subset* of its effects, and to ignore or compensate for any undesired side effects on the processor state, depending on the type of storage base involved (see Section 2): *temporaries* such as condition codes may be ignored, *general-purpose* locations such as registers and primary memory may be used if allocated, and *reserved* locations such as the stack pointer and program counter must not be destroyed.

The remaining axioms are concerned with flow of control. Some define higher level constructs in terms of low-level ones. For example,

$$\textbf{if E then S} \Leftrightarrow \textbf{if not E then goto L};\ S;\ L:$$

describes how to implement an **if** with a conditional jump. The other flow axioms define the semantics of the program counter and low-level control:

$$\textbf{goto E} \Leftrightarrow PC \leftarrow E$$
$$PC \leftarrow PC + n;\ S\langle \text{space } n\rangle \Leftrightarrow \langle \text{nil}\rangle$$
$$PC \leftarrow L_1 \Leftrightarrow PC \leftarrow PC + L_1 - L_2;\ L_2:$$

The first of these simply defines the program counter (PC); the second rule says that incrementing the PC by n causes the next n units of code to have no effect; and the last allows the use of relative and absolute jumps interchangeably.

4.2 Search

Now consider the machine-independent code generation problem. We are given a machine M with instructions $m_1, m_2, \ldots, m_n$, and a goal tree G (from the Select procedure) for which we would like to generate code (in the machine language of M). That is, we would like to find a sequence of instruction tree instantiations

that is semantically equivalent to the goal tree G. The axioms presented in Section 4.1 define "equivalent": if a subtree matches one side of an axiom schema, the subtree may be replaced by the instantiation of the other side. In this way, the goal G can be successively transformed into other trees, until eventually we may arrive at a tree that is a sequence of instruction trees:

$$G \Rightarrow G' \Rightarrow G'' \Rightarrow \cdots \Rightarrow m_{i_1}; m_{i_2}; \ldots; m_{i_k}.$$

Because more than one axiom may be applicable to a tree at any point, and we can test for the termination condition of a sequence of instructions, we have a classical search problem. That is, starting with G, we may use all applicable axioms to obtain a set of equivalent trees, recursively apply all applicable axioms to *those* trees, and so on, until we have one or more instruction sequences for the goal tree.

Applying this search procedure literally is undesirable, as the search space is combinatorially large. Note that axioms may be applicable at more than one point in a goal tree, and more than one axiom may be applicable at each one of these points.

To reduce the size of the search space, we use some established methodology from the field of artificial intelligence. In fact, we use not one method, but several, allowing the strongest applicable method to be used for each kind of information. For this purpose, the axioms have been divided into three classes.

(1) *Transformations.* These are the axioms concerned with arithmetic and Boolean equivalence. Transformations will be used in conjunction with means–ends analysis in the search.
(2) *Decompositions.* These axioms are normally concerned with control constructs; they decompose constructs into sequences of other constructs, allowing the search to proceed recursively on subgoals. Decompositions will be used in conjunction with a heuristic search.
(3) *Compensations.* These are the axioms concerned with side effects. No search at all will be associated with these axioms; it will be possible to use them in a prepass on the MD.

Briefly, the basic Search procedure, which is applied to each pattern (goal) tree determined by Select, is as follows:

S1. If the goal tree matches an instruction directly (or matches a pseudoinstruction with side effects, as described in Section 4.3), we return that instruction as the code sequence for the goal tree.
S2. If there are any Decomposition axioms applicable to the goal tree, this search procedure is applied recursively to try each new goal tree resulting from their application. If any of these recursive instantiations succeed in finding code sequences, the alternatives will be returned. Decomposition is used in the second example in Section 4.3.
S3. Means–ends analysis is applied to the goal tree as follows. A set of instructions whose assertion trees are *semantically close* to the goal tree is selected (see below). For each of these selected instructions, an attempt is made to transform it recursively so that it may be used for the goal tree, by applying the transformation axiom(s) that reduces the difference between the two. The first example in Section 4.3 illustrates the use of this strategy.

All possible code sequences found for a given tree are returned by the Search routines, and the best of these is chosen by Select to be entered into the MT. The "best" cost is determined by a user-supplied function of time and space; the time and space costs for instructions are known from the MD.

A relatively simple heuristic measure of the semantic closeness used in S3 was found to work quite well. The measure is based on comparing the *primary operator* of the goal tree and a potential instruction. The primary operator of a tree T, po(T), is defined in terms of the top operator of T, op(T), as follows:

If op(T) is	po(T) is
a conditional	op(lhs(T)) (i.e., the conditional expression)
an assignment	op(rhs(T)) (i.e., the expression to compute)
anything else	op(T)

We now define a tree S to be semantically close to a tree T iff po(S) = po(T) or there exists an axiom $P_1 \Rightarrow P_2$ such that $po(P_1) = po(T)$ and $po(P_2) = po(S)$. The net effect of this heuristic measure is to select an instruction tree S if it performs an operation that is identical or arithmetically related to the goal tree. Some further performance improvements can be obtained by some minor refinements of this closeness measure; the reader is referred to Cattell [6] for further details. Note that instructions/axioms can be *indexed* by their primary operator/operators. As a result the selection of semantically close instructions in S2, the selection of potentially applicable axioms in S3, and the selection of axioms that reduce differences (defined as the difference in po's) in S2 can be performed in essentially constant time.

It is important to the Search procedure that the three classes of axioms deal with orthogonal types of TCOL constructs. Note in particular steps S2 and S3. The decomposition axioms used in S2 deal primarily with control constructs, while the transformation axioms used in S3 deal with arithmetic and Boolean computations. The means–ends analysis used in S3 efficiently handles the large search space defined by the arithmetic/Boolean axioms by selecting potential instructions to determine the choice of axioms to apply, while the much smaller search space defined by axioms on control constructs can be handled by the (almost) brute-force search used in S2. The particular order here of S2 and S3 was not essential to success. The reverse order in fact has some advantages, as the set of axioms used in the brute-force step can be expanded to include transformation axioms in the event that the means–ends analysis fails (though empirically this failure did not occur).

The Search procedure could potentially run forever: it is necessary to restrict the search both in the depth of recursion and in the breadth (the number of semantically close instructions tried in S3). The Select procedure described in Section 4.4 was designed to increase the depth and breadth if a searched failed, although a fixed search actually worked adequately.

4.3 Example

As an example of the use of transformations, consider the problem of loading the accumulator on a simple PDP-8-like machine (there is no load instruction to do this directly). The process can most easily be understood by following the steps

```
Search: (← %ACC %MP)                                  *Search is passed goal tree*
Attempting M-op-match                                 *no instructions match goal*
Attempting Decompositions
Attempting Transformations                            *...attempt transforming twos complement*
 Feasible[1]: (← %ACC (+ %ACC $1:Z))                  *add (TAD) instruction to use for the goal*
  Transform: (← %ACC %MP) => (← %ACC (+ %ACC $1:Z))
   Transform: %ACC => %ACC                            *LHS of the "←" matches*
   Transform: %MP => (+ %ACC $1:Z)                    *but RHS mismatches*
    Applying $1 :: (+ 0 $1) to: %MP                   *try this axiom to reduce difference*
     Transform: (+ 0 %MP) => (+ %ACC $1:Z)            *now "+" node matches*
      Transform: 0 => %ACC                            *but 0 still mismatches %ACC*
       Applying Fetch Decomposition to: 0             *ACC←0 will fix this mismatch*
        Search: (← %ACC 0)                            *and there is a CLRA machine-op*
        Attempting M-op-match                         *(M-op match explained later)*
        M-op Match: (; (ALLOC $$2:Z) (EMIT[DCA 1 1 1] 3 $$2:Z))   ▲ see text
        M-op Match: (EMIT[CLRA 3 1 1] 7 0 20)         *there are 2 ways to clear ACC*
      Transform: %MP => $1:Z                          *Z is an operand class, and matches %MP*
 Feasible[2]: (← %ACC (+ %ACC 1))                     *try other feasible M-ops...*
  Transform: (← %ACC %MP) => (← %ACC (+ %ACC 1))
                                                      *but no other solutions found*
...
Best Sequence is:
 [Alloc $$1:%ACC]
 CLRA
 TAD %MP
-------------------------
```

Fig. 3. Machine description has been input, and the top-level search routine is given the goal tree (← %ACC %MP), the TCOL representation of the problem of interest.

of the actual search procedures; a trace output from the implementation is shown in Figure 3.[2]

The first feasible instruction found is the twos complement add (TAD) instruction, whose tree representation is (← %ACC(+ %ACC $1:Z)); no other instruction matches the primary operator and also has the appropriate destination (%ACC). The system therefore attempts to transform

$$(\leftarrow \%ACC\ \%MP) \Rightarrow (\leftarrow \%ACC(+\ \%ACC\ \$1{:}Z)).$$

The %ACC part matches, but the RHSs mismatch. The program finds the transformation, $1 ⇒ (+ 0 $1), whose root operators match the mismatching subtrees, and it is applied to create the subproblem of transforming

$$(+\ 0\ \%MP) \Rightarrow (+\ \%ACC\ \$1{:}Z).$$

The +'s now match, but the 0 and %ACC mismatch. Fetch Decomposition is applied to make these match by storing 0 into %ACC. Two instructions are found to do this (see next paragraph), the better one being CLRA (clear accumulator). The %MP matches the operand class Z, because Z is defined to allow either a direct or indirect memory reference. We have then completed the match. The search proceeds to try other feasible instructions, but no further code sequences are found. The best code sequence to load %ACC is therefore to clear %ACC and add %MP.

As an example of the use of compensations, note the line in the figure marked

[2] In Figures 3 and 4, the comments within asterisks have been inserted to annotate the output; also, parts have been truncated with "..." for readability. A parenthesized LISP-like form is used for the TCOL trees. For example, (← %ACC (+ %ACC %MP)) means add a memory location (%MP) to the accumulator (%ACC). Parameters, e.g., "$1", are associated with nodes for later reference. Global parameters, e.g., "$$1", are parameters whose scope is over an entire search, as opposed to a single axiom or instruction; they are used to refer to temporaries needed in the code sequence. Access modes are preceded with "%" by convention; operand classes (e.g., Z in the example) are not. For complete and more extensive examples, see Cattell [6].

with a ▲ on the right. The compensation rules tell the search that the accumulator can be cleared by using the deposit and clear instruction if a memory location is allocated into which the accumulator may be stored, resulting in the sequence

```
[Alloc $$2:%MP]
DCA $$2:%MP
TAD %MP
```

to load the accumulator. This is of higher cost than the best sequence, however, so it is rejected (the reader may be curious as to the case where we already know ACC = 0; this optimization is handled in a separate compiler phase (FINAL)).

As an example illustrating the use of *decompositions*, Figure 4[2] shows the generation of code for **if** ACC = 0 **then** ACC ← 1. Both the definition of **if** and skip-decomposition get applied in this derivation, and two alternative code sequences are found depending on which is tried. The better sequence is to do a SKPNE (skip if accumulator nonzero) followed by SETIA (set accumulator to 1).

4.4 Select

As explained earlier, the machine-independent code generator (Search) is not used directly in the compiler for performance reasons. Rather, a level of indirection is introduced by running Search only on a preselected set of tree patterns to be inserted into the MT used by the actual code generator. The preselection is done by the Select procedure, basically as follows:

S1. Include all instructions in the MT as is. Thus, if a program tree segment matches directly, the instruction will be selected.

S2. For each instruction with *multiple assertions* (i.e., for which two or more locations could be modified), for each of its assertions for which the instruction may be used according to the compensation axioms, add a new pseudoinstruction for that assertion alone. The RHS of this new pseudoinstruction production may include not only the instruction itself, but some compensation for the other side effects. For example, an increment-and-skip-if-zero instruction could be used for the increment action alone by appending a no-op; or an instruction that stores into both memory and a register can be used for the latter action alone by preceding it with an allocation of a dummy memory location.

S3. Insure there is a production for A ← B for every pair of distinct access modes A and B such that A and B are "simple" references to locations of the same size. A "simple" reference is one in which the index into the storage base is a cardinal (as opposed to, say, indirect or relative addressing). If there is already such an entry from S1 and S2, no action is taken. Otherwise, Search is called to find the best code sequence for A ← B, and a tree production is added whose pattern (LHS) is the A ← B tree and whose RHS is the code sequence.

S4. Insure there is at least one production in the MT for every TCOL operator. Productions are obtained in a way similar to the previous step, namely, by calling Search for every tree of the form A ← B *op* C, A ← *op* B, and **if** A *op* B **then goto** C. It is unimportant what locations are represented by A, B, and C, since the code generator will make any moves needed to put the data

```
Search: (IF (EQL %ACC 0) (← %ACC 1))
Attempting M-op-match
Attempting Decompositions                    *first Search tries applying definition of if*
Applying (IF $1 $2) :: (; (-> (NOT $1) $3:%MP) $2 (LABEL $3:%MP))
*note: (-> A B) means "if A then goto B", LABEL means emit a label, and ";" means perform*
*its arguments in sequence (its three arguments shortly become three subgoals...)*
  Simplifying (NOT (EQL %ACC 0)) to (NEQ %ACC 0) *note logical simplifications must be done*
  Search: (; (-> (NEQ %ACC 0) $$1:%MP) (← %ACC 1) (LABEL $$1:%MP))
  Attempting M-op-match
  Attempting Decompositions                  *Search decomposes ";" node from if*
  Applying Sequence-Decomposition            *and treats each subnode as a subgoal*
    Search: (-> (NEQ %ACC 0) $$1:%MP)        *1st subgoal (from if definition)*
    Attempting M-op-match
    Attempting Decompositions
    Applying Skip-Decomposition              *decompose into skip and goto*
      Search: (GOTO $$1:%MP)
      Attempting M-op-match
      Attempting Decompositions
      Applying (GOTO $1) :: (← %PC $1)       *the goto is recognized as a store into PC*
        Search: (← %PC $$1:%MP)
        Attempting M-op-match
        M-op Match: (EMIT[JMP 2 1 1] 5 $$1:%MP)  *and a JMP is used to satisfy it*
      Attempting Transformations
      Simplifying (NOT (NEQ %ACC 0)) to (EQL %ACC 0)
      Search: (-> (EQL %ACC 0) (+ %PC 1))    *the skip is recognized as SKPE*
      Attempting M-op-match
      M-op Match: (EMIT[SKPE 3 1 1] 7 1 5)
        ...
    Breadth Limit Reached(6)
    Search: (← %ACC 1)                       *now do 2nd subgoal (then part of if)*
    Attempting M-op-match
    M-op Match: (EMIT[SET1A 3 1 1] 7 0 30)
    Search: (LABEL $$1:%MP)                  *3rd (and last) subgoal: generate label
*now we have one possible code sequence: "SKPE; JMP $$1; SET1A; $$1:".*
*Search then tries applying Skip-decomposition to original goal instead of if definition, i.e.,*
*if X then Y => if not X then PC←PC+1; Y (since Y is Acc←Acc+1, code takes 1 word)*
Applying Skip-Decomposition
  Search: (← %ACC 1)                         *first try then-part*
  Attempting M-op-match
  M-op Match: (EMIT[SET1A 3 1 1] 7 0 30)     *OK, it takes 1 word*
  Simplifying (NOT (EQL %ACC 0)) to (NEQ %ACC 0)
  Search: (-> (NEQ %ACC 0) (+ %PC 1))        *so try conditional skip of 1*
  Attempting M-op-match
  M-op Match: (EMIT[SKPNE 3 1 1] 7 1 2)      *success. now have 2 seqs.*
Attempting Transformations                   *keeps trying, but no more solutions..*
    ...

----------
Nodes Examined:  15
Est. Seconds:    .062
Result Sequence(s):
<<<<<<<<<<<<<<<<<<<<
    SKPE
    JMP    $$1:%MP
    SET1A
$$1:
----------or----------
    SKPNE
    SET1A
>>>>>>>>>>>>>>>>>>>>

Best Sequence is:
    SKPNE
    SET1A
----------
```

Fig. 4. Generation of code for if ACC = 0 then ACC ← 1.

in the required locations. For example, if logical **and** did not exist as the primary operator of an instruction directly on the machine, a code sequence for it would be derived and the resulting production (LHS is the **and** tree, RHS is the derived code sequence) added to the MT. All derived productions are also indexed as if they were machine instructions for use in further searches.

S5. Finally, add to the MT the productions for control operators. These correspond to the axioms in Figure 2 which define **while-do, if-then-else,** etc., in terms of conditional and unconditional jumps.

This procedure insures that the minimal code generator using the MT will be able to generate code for all TCOL operators, and that if there exists a one-instruction code sequence for a subtree, the code generator will find it (by S1). It does *not* guarantee that if the Search procedure discussed in Section 4.2 generates optimal code then the code generator using the MT generated therefrom will do so, because the necessary special case combination of TCOL operators may not have been included in the MT. Of course, special cases could be suggested by a human, but interestingly, such help was not found to be necessary: special cases more complex than single TCOL operators (in all contexts) were not needed for the machines tested, except for cases in which instructions match directly (Select handles these in step S1) and cases which are already handled by the peephole optimization phase. (The second example in Section 4.3 is one of the few in the latter case; peephole optimization is not even needed here if the skip-decomposition axiom is instead used at compile time to recognize the one-instruction **then**-parts of conditionals.) Nevertheless, a procedure to guarantee an optimal set of special cases would be desirable (say, by some exhaustive analysis of possible pattern trees). This is an area for future research.

Note also that the search procedure presented in Section 4.2 does not guarantee optimal code, or any code at all for that matter, because the search may not be deep enough to discover the equivalence. Furthermore, even if we searched to an arbitrary depth, a code sequence might still not be found, because a necessary axiom to determine the sequence's equivalence to the goal tree may not be in Search's repertoire. The search failure implies that the axiom set is not complete; however, *no* set of axioms could form a basis for all equivalences true over all programs [19]. This suggests that the goal of this work, i.e., to take an arbitrary machine description and generate code, is unachievable! Fortunately, this result does not have great practical impact: the set of about 50 axioms was adequate for the machines tested.

5. SUMMARY

This work has dealt with (1) a model of instruction set processors, (2) a code generation algorithm in which machine-dependent information is separated into tabular form, and (3) a scheme for heuristic search for optimal code sequences, based on an axiomatization of tree equivalence.

5.1 Results

The results have been encouraging. The machine representation was general enough to deal with a variety of actual machine architectures (the IBM-360,

PDP-10, PDP-11, Intel 8080, Motorola 6800, and PDP-8 are discussed in Cattell [6]). The code generation algorithm satisfies the goals of tabularizing machine dependence and at the same time remaining flexible and fast enough for use in a production compiler. The last and perhaps most interesting result is that the formal approach of heuristic search for code sequences did not fail to find the optimal code sequences (for the machines tested within the scope of the data types and operations covered by the axioms).

One might expect the code generator in the compiler to be relatively slow, since it involves a table-driven pattern-matching scheme. However, the prototype implementation on a PDP-10/KL10 was basically input–output bound, generating about 2000 instructions per second from the intermediate TCOL representation. It is written in Bliss [34]. The code generator (and the entire PQCC compiler) cross-compiles rather than compiles. With some care in making the compiler code portable, of course, the compiler could be used to compile itself by the usual bootstrapping procedure to obtain a compiler running on an arbitrary machine. The code itself is quite compact, requiring only 1K 36-bit words, because all the machine-dependent information is in the tables. The tables require considerably more space (the amount being target-machine dependent, but with an order of 10K words). The prototype system described here is under redesign and integration into the PQCC compiler; the complete compiler will be necessary for an objective evaluation of the code quality, although small examples (see Cattell [6]) led to code comparable to a hand-coded optimizing compiler.

The code generator generator is also surprisingly fast in comparison to previous results using formal methods (e.g., Newcomer [22]). The derivations of code sequences for templates typically took about 0.1 second (KL10). The generation of the MT itself took about 10 seconds for a typical machine (the PDP-11). The code generator generator uses 40K words plus 10 to 20K words for data; it is implemented in SAIL [25].

The speed of the code generator generator is not greatly affected by either the number of axioms or the number of instructions on the target machine, because the indexing scheme allows the search routines to go almost directly to the applicable axiom (for a mismatch) or instruction (for a goal tree). Note that the axioms are machine independent, so that it should only be necessary to add new axioms when a new domain is added, e.g., when TCOL is extended to include a new data type such as character strings.

The machine descriptions used in this work (MD in Figure 1) are in a relatively compact parenthesized text form. The PDP-11/20 description (a fairly basic machine with no floating point or unusual operations), for example, is about 200 lines. An understanding of the components of the machine model (Section 2) is of course necessary to construct such a description, and a man-week or so is required to write and debug a typical one (the PDP-11). The machine model and description format are under further development in PQCC.

The success with formal methods is probably due to the choice of *representation*. In general, efficient procedures were straightforward when the problems were expressed in the right way. This principle can be seen to apply in several areas of the work, including the machine representation, code generator representation, and the use of axioms and trees in the search for code sequences. In

particular, some important representational issues were

(1) the use of a common notation, TCOL, to represent procedural semantics; also important is the extensibility of TCOL with respect to new data types and operators (these then require additional axioms);
(2) the restricted form of the instruction interpreter, reducing the selection of primitives to sequences of actions represented by input/output assertions;
(3) abstraction of orthogonal properties such as addressing and binary representation from the representation of the abstract operations themselves (the instructions).

Some of the techniques used in this work may be useful to other applications of machine descriptions. For example, automated hardware generation is conceptually analogous to code generation, as it involves decomposing a given algorithm into a set of given primitives [17].

The techniques used here may also be useful in the generation of microcode, although the latter calls for somewhat different methods. For example, it is typical rather than exceptional for microinstructions to have more than one action (see S2 of Section 4.4), so the code generation algorithm might routinely perform a lookahead in an attempt to use an instruction for several of its actions.

5.2 Limitations and Future Research

The model of machines used in this work is more general but more complex than that used in previous work in an attempt to allow a wide range of architectures but enable good code generation. Considerably more work in machine formalization is needed, however; success with the current model suggests this research would be profitable. The model summarized in Section 2 did not deal adequately with description of machine data types (e.g., character strings), input/output, and special architectural features such as instruction lookahead, pipelines, and caches (which must be considered for good code). Extensions to the tree notation (TCOL) are needed in conjunction with additional axioms and the specification of data types in order to deal with more complex machine instructions and optimizations, specifically bit field extraction/modification, byte string manipulation, and machine operations tailored to high-level language operations. Except for these machine features, there are no significant difficulties with either the machine representation or code generation for the six machines mentioned in Section 5.1. In the thesis work, however, time limited closer study to just the PDP-11/20, DEC KA10, and the simple PDP-8-like machine used in the examples: the code generator generator ran successfully on descriptions of these machines. Descriptions of the other machines have since been written in the PQCC work, although not yet used for compiler generation at the time of this writing.

A better template selection scheme than the one outlined in Section 4.4 almost surely exists. Alternatively, the performance obtained in the code generator generator suggests that at least some of the axioms could instead be applied at compile-time without unreasonable speed degradation.

A final but important area for future research, particularly for optimizing compilers, is the integration and generation of the other compiler phases: register allocation, the compiler-writer's virtual machine (translation of high-level oper-

ations like parameter passing and compound data structure access into primitive TCOL operations), and peephole optimization. Continued research building on the work described here is still in progress in the PQCC project; a summary of the PQCC work can be found in Leverett et al. [18]. The interaction of the phases of the optimizing compiler are complex compared to any one phase and centrally important to the generation of good code.

ACKNOWLEDGMENTS

I would like to thank Bill Wulf, Mario Barbacci, Allen Newell, John Oakley, Steve Saunders, Bruce Leverett, Joe Newcomer, Alice Parker, Bruce Schatz, Steve Hobbs, and Doug Clark for their invaluable comments and conversations concerning my thesis work and this paper.

REFERENCES

1. ALLEN, F., CARTER, J., HARRISON, W., LOEWNER, P., TAPSCOTT, R., TREVILLYAN, L., AND WEGMAN, M. The experimental compiling systems project. IBM Res. Rep. RC 6718 (#28922), IBM, Yorktown Heights, N. Y., 1977.
2. BARBACCI, M., BARNES, G., CATTELL, R., AND SIEWIOREK, D. ISPS reference manual. Tech. Rep. TR 79-137 (latest revision), Comput. Sci., Carnegie-Mellon Univ., Pittsburgh, Pa., 1977.
3. BARBACCI, M., AND SIEWIOREK, D. The CMU RT-CAD system: An innovative approach to computer aided design. *Carnegie-Mellon Univ. Comput. Sci. Rev.*, 1974–1975.
4. BELL, C.G., AND NEWELL, A. *Computer Structures: Readings and Examples.* McGraw-Hill, New York, 1971.
5. CATTELL, R.G. A survey and critique of some models of code generation. Unnumbered Tech. Rep., Comput. Sci., Carnegie-Mellon Univ., Pittsburgh, Pa., 1977.
6. CATTELL, R.G. Formalization and automatic derivation of code generators. Ph.D. dissertation, Tech. Rep. TR 78-115, Comput. Sci., Carnegie-Mellon Univ., Pittsburgh, Pa., 1978.
7. CATTELL, R.G., NEWCOMER, J.M., AND LEVERETT, B.W. Code generation in a machine-independent compiler. Proc. SIGPLAN Symp. Compiler Construction, Denver, Colo., Aug. 1979.
8. DONEGAN, M.K. An approach to the automatic generation of code generators. Ph.D. dissertation, Comput. Sci. Eng., Rice Univ., Houston, Tex., 1973.
9. ELSON, M., AND RAKE, S.T. Code-generation technique for large-language compilers. *IBM Syst. J. 9*, 3 (1970), 166–188.
10. FELDMAN, J. A formal semantics for computer-oriented languages. Ph.D. dissertation, Comput. Sci., Carnegie-Mellon Univ., Pittsburgh, Pa., 1964.
11. FELDMAN, J., AND GRIES, D. Translator writing systems. *Commun. ACM 11*, 2 (Feb. 1968), 77–113.
12. FRASER, C.W. Automatic generation of code generators. Ph.D. dissertation, Comput. Sci., Yale Univ., New Haven, Conn., 1977.
13. GLANVILLE, R.S. A machine independent algorithm for code generation and its use in retargetable compilers. Ph.D. dissertation, Tech. Rep. TR UCB-CS-78-01, Comput. Sci., Univ. California at Berkeley, Dec. 1977.
14. GLANVILLE, R.S., AND GRAHAM, S.L. A new method for compiler code generation. Proc. 5th Conf. Principles of Programming Languages, Jan. 1978.
15. HOBBS, S. Object code optimization. Unpublished thesis proposal, Comput. Sci., Carnegie-Mellon Univ., Pittsburgh, Pa., 1976.
16. JOHNSON, S.C. A portable compiler: Theory and practice. Proc. 5th Conf. Principles of Programming Languages, Jan. 1978, pp. 97–104.
17. LEIVE, G. The binding of modules to abstract digital hardware descriptions. Unpublished thesis proposal, Elec. Eng., Carnegie-Mellon Univ., Pittsburgh, Pa., 1977.
18. LEVERETT, B., CATTELL, R., HOBBS, S., NEWCOMER, J., REINER, A., SCHATZ, B., AND WULF, W. An overview of the production quality compiler-compiler project. Tech. Rep. TR 79-105, Comput. Sci., Carnegie-Mellon Univ., Pittsburgh, Pa., Feb. 1979 (condensation to appear in *IEEE Comput.*).

19. LUCKHAM, D.C., PARK, D.M.R., AND PATERSON, M.S. On formalized computer programs. *J. Comput. Syst. Sci. 4*, 3 (1970), 220–249.
20. MCKEEMAN, W.M., HORNING, J.J., AND WORTMAN, D.B. *A Compiler Generator*. Prentice-Hall, Englewood Cliffs, N.J., 1970.
21. MILLER, P.L. Automatic creation of a code generator from a machine description. Tech. Rep. TR 85, Project MAC, M.I.T., Cambridge, Mass., 1971.
22. NEWCOMER, J.M. Machine independent generation of optimal local code. Ph.D. dissertation, Comput. Sci., Carnegie-Mellon Univ., Pittsburgh, Pa., 1975.
23. OAKLEY, J.D. Automatic generation of diagnostics from ISP. Unpublished thesis proposal, Comput. Sci., Carnegie-Mellon Univ., Pittsburgh, Pa., Dec. 1976.
24. OAKLEY, J.D. Symbolic execution of formal machine descriptions. Ph.D. dissertation, Tech. Rep. TR 79-117, Comput. Sci., Carnegie-Mellon Univ., Pittsburgh, Pa., April 1979.
25. REISER, J., ET AL. SAIL Stanford Artificial Intelligence Lab. Memo. AIM-289, Comput. Sci., Stanford Univ., Stanford, Calif., 1976.
26. RIPKEN, K. Formale Beschreibung von Maschinen, Implementierungen und optimierender Maschinencodeerzeugung aus attributierten Programmgraphe. Dissertation, Technische Univ. Munchen, Munich, Germany, 1977.
27. SAMET, H. Automatically proving the correctness of translations involving optimized code. Ph.D. dissertation, Comput. Sci., Stanford Univ., Stanford, Calif., 1975.
28. SIMONEAUX, D.C. High-level language compiling for user-definable architectures. Ph.D. dissertation, Elec. Eng., Naval Postgraduate School, 1975.
29. SNYDER, A. A portable compiler for the language C. Tech. Rep. TR 149, Project MAC, M.I.T., Cambridge, Mass., 1975.
30. WEINGART, S.W. An efficient and systematic method of compiler code generation. Ph.D. dissertation, Comput. Sci., Yale Univ., New Haven, Conn., 1973.
31. WHITE, J.R. JOSSLE: A language for specifying and structuring the semantic phase of translators. Ph.D. dissertation, Univ. California at Santa Barbara, 1973.
32. WICK, J.D. Automatic generation of assemblers. Ph.D. dissertation, Comput. Sci., Yale Univ., New Haven, Conn., 1975.
33. WILCOX, T.R. Generating machine code for high-level programming languages. Ph.D. dissertation, Comput. Sci., Cornell Univ., Ithaca, N.Y., 1971.
34. WULF, W., JOHNSSON, R., WEINSTOCK, C., HOBBS, S., AND GESCHKE, C. *The Design of an Optimizing Compiler*. American Elsevier, New York, 1975.
35. YOUNG, R. The coder: A program module for code generation in high-level language compilers. M.S. thesis, Comput. Sci., Univ. Illinois, Urbana, 1974.

Received February 1979; revised June and October 1979; accepted December 1979