A REVIEW OF AUTOMATIC CODE GENERATION TECHNIQUES

by

Mahadevan Ganapathi
Charles N. Fischer

Computer Sciences Technical Report #407

January 1981

A Review of Automatic Code Generation Techniques

Mahadevan Ganapathi *

Charles N. Fischer **

Abstract

Code-generation research is classified into three categories: formal treatments, interpretive approaches and descriptive approaches. Very formal approaches have usually not considered complete machine architectures. Interpretive approaches are improvements over ad-hoc code generation techniques but retargeting requires changing the code generator for every new machine. Descriptive approaches separate the machine description from the code-generation algorithm, thus providing a higher degree of portability. A review of these approaches and a critique of automatic code-generation algorithms are presented.

---

* Now at Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051

** Research supported in part by National Science Foundation Grant MCS78-02570

1. Introduction

Code generation involves the complex task of selecting machine instructions to implement programming language constructs. Code has to be generated in the following areas:

(1) binding source-language variables to storage locations,
(2) accessing variables,
(3) evaluating arithmetic and Boolean expressions,
(4) executing control constructs and evaluating predicates (without storing an explicit Boolean result),
(5) setting up run-time display linkage during procedure calls, and
(6) procedure prologue and epilogue.

Previous research in code generation can be broadly classified into three categories: formal treatments, interpretive approaches and descriptive approaches.

2. Formal Treatments

Newcomer [Newcomer 75] uses means-end-analysis [Newell 69] to generate code templates (not machine instructions) from a parse tree and a set of operators. His scheme is very restrictive and of very little practical importance because:

(1) It only deals with arithmetic expression trees.

(2) Real machine architectures are not always readily representable in his specification scheme.

(3) His code generation algorithm can fail to produce a code template due to
   (a) a possibly inadequate set of operators, or
   (b) a limitation of the depth of search performed by means-end-analysis (a limit is needed to prevent the code generator from possibly looping).

(4) The algorithm is exhaustive and therefore far too expensive to be used in a production compiler. Sometimes, weeks of computer time are required to analyze even simple trees!

Aho and Johnson [Aho 76] consider a similar exhaustive "brute force" optimal code generation scheme. They use a three-phase dynamic programming algorithm to derive optimal code sequences for expression trees. In the first phase, trees are traversed bottom-up. For each vertex 'v', all possible machine instruction translations of the subtree rooted by 'v' are used to compute an array Cr[v] ($1 \leq r \leq$ total number of registers) of costs. Cr is the minimum number of instructions required to compute the subtree using 'r' registers. All permutations of evaluation order are considered. At the end of the first phase the following are determined:

(1) the optimal instruction sequence required to compute the subtree rooted by vertex 'v', and

(2) the optimal evaluation order for the subtree of 'v'.

The second phase uses the cost arrays and traverses top down to mark tree nodes that must be computed in memory locations (i.e. wherever 'stores' are necessary because of too few registers). The last phase walks each marked subtree and generates code to evaluate the subtree followed by appropriate 'stores' into temporary memory locations. Aho & Johnson show that this algorithm requires time linear in the number of tree nodes and exponential in the number of instruction and addressing mode choices at each point. The shortcomings of this model are:

(1) It only deals with arithmetic expression trees excluding common subexpressions.

(2) Only mathematically clean instructions are dealt with, avoiding asymmetric registers and special instructions found in real computers.

Samet implemented a verifier [Samet 75] to prove the correctness of PDP-10 assembler code produced by translating a subset of LISP. The assembler language instructions are symbolically simulated using LISP procedures resulting in a tree representation of their effect. Semantic equivalence axioms [McCarthy 63] are then used to transform trees into equivalent ones until they can be shown semantically equivalent to a tree representation of the source program. This verification technique can prove whether assembler code generated from a source program is a correct implementation independent of the translation process. It does not, however, prove the correctness of code generators (independent of the input program).

3. Interpretive Approaches

Two-level translation schemes were suggested to help design portable compilers [Ershov 58, Strong 58, Steel 61]. To implement 'p' programming languages on 'm' machine architectures only p+m translators are necessary instead of p*m. Code is produced for a virtual machine and is then expanded into real machine instructions. Such schemes use code generation languages specifically designed to describe the code generation process along with the target machine instructions (GCL [Elson 70], BCPL Ocode [Richards 71], ICL [Wilcox 71, Young 74], CGPL, CGGL [Donegan 73, Donegan 79], Pascal P-code [Ammann 77]). While such approaches are a distinct improvement over ad-hoc methods, they suffer from serious limitations:

(1) Due to the diversity in addressing modes, target machine data types and instructions, it is very hard, if not impossible, to anticipate a variety of machine organizations (e.g. whether a hardware stack exists) in one virtual machine. Interpreters tend to be very large and complex (In Elson and Rake's implementation [Elson 70], macros had to be paged in from disk).

(2) Code generation languages are closely tied to a specific language or machine. Thus they cannot be considered truly portable.

(3) The description of the target machine is mixed with the code generation algorithm; the description cannot be changed without changing the algorithm.

(4) The implementor must perform a tedious case analysis of code sequences and make all low-level decisions as to what kind of code is to be generated. The quality of the code produced depends on the implementor's ability to design and debug code generation routines.

(5) The implementor has a very local view of the code to be generated. It is hard to incorporate context-dependent optimizations such as:
   (a) use of indexing instead of explicit addition in an addressing context,
   (b) differentiation between Boolean values to be stored (i.e., expressions) and Boolean values that need only be tested (i.e., predicates),
   (c) branch chaining and other flow-dependent optimizations [Wulf 75].

4. Descriptive Approaches

For reasons of portability, code generation research has concentrated on separating machine descriptions from the code generation algorithm itself. The advantage of this approach is the potential ability to use one code generation algorithm for all machines.

4.1 Hand-written Code Generation Algorithms

Miller made the first attempt to isolate machine-dependent issues from the code generation algorithm [Miller 71]. The source language is mapped to two-address code sequences, which are macros written in MIML (Machine Independent Macro Language). These macros specify the actual code generation algorithm, e.g.

```
macro Add x, y
    If type of x = integer and type of y = integer
        then Iadd x, y
    else if type of x = float and type of y = float
        then Fadd x, y
    else error
```

The specifications of, e.g., macros Iadd and Fadd in OMML (Object Machine Macro Language) form the description of addition on the target machine. Thus, for the IBM-360:

```
macro Iadd a, b
    from a in R1, b in R2 emit (AR a, b) result in R1
    from a in R, b in M emit (A a, r) result in R
    from a in M, b in R emit (A b, a) result in R
```

States are defined as configurations of operand locations. A state is 'permitted' if from that state code can be emitted with operands unmoved. Every macro is associated with a set of permitted states only. The designer is therefore required to specify transitions between memory and registers so that the code generator automatically moves to a permitted state if needed (e.g. movement of an operand in storage to a register to implement storage-to-storage addition). To retarget a compiler to a new machine, Iadd and Fadd must be changed. The algorithm represented in Add is expected to remain unchanged. Miller's model, however, is too restrictive because it deals with expression evaluation and very simple addressing schemes only; it does not allow indexing, auto-increment, or indirect addressing.

Weingart introduced pattern matching to avoid interpretation [Weingart 73]. Target machine characteristics are encoded into a single pattern tree that is expected to be a compact and efficient means of representing most machine-dependent information. The code generator is a tree traverser that accepts tokens from a parse tree of the source language and stores them until a suitable match can be found in the pattern tree. To transport this code generation scheme to a different target machine, the user creates a new pattern tree for the new machine. In practice, Weingart's ideas are not easy to use because:

(1) Creating a single tree structure to encode all potential instruction patterns and code sequences is often hard. For example, Weingart had difficulties creating the pattern tree for the PDP-11. He tried to generate the tree from a machine description automatically, but did not succeed very well.

(2) There exists a possibility that on some machines no instructions at all will match parse trees. Pattern mismatches are handled by a set of conversion patterns. However, there is no way to determine if a sufficient set of conversion patterns has been supplied. The code generator might therefore fail to produce any code for some legal subtree of the source language.

(3) Some machines provide a choice of instructions to implement a source language construct. Code quality depends critically on the selection of the most appropriate instructions (e.g. using 'increment' instead of 'add one'). Special care must therefore be taken by the tree traverser to make the best possible instruction choice (Weingart's technique cannot make such a choice).

Snyder [Snyder 75] attempted to write a portable compiler for the language C [Ritchie 78] (but he did not succeed very well). His compiler uses a two-phase translation scheme very similar to Miller's. In a first phase, the code generator walks an expression tree and generates three-address instructions. The classification of registers and the register requirements of these instructions are defined by the programmer. A second phase then translates three-address instructions into assembler code for the target machine. Macros and C routines are used to perform tedious case analysis of code sequences. Snyder to a large extent ignored object code optimization.

A number of Snyder's ideas are used by Johnson in his successful implementation of the portable C compiler [Johnson 77, Johnson 78]. Templates and a template-matching algorithm form the central idea around which code generation is designed. Templates specify:

(1) the operator of the subtree (e.g. an assign-op),

(2) the desired result location on the target machine (e.g. a register location or a condition-code setting),

(3) the machine addressing mode and the language data type of the operands of the expression, if any (e.g. register mode, pointer type),

(4) the resource requirements: the number of temporaries and scratch registers needed for implementing the subtree,

(5) a rewrite rule specifying how to replace a subtree by another, and

(6) the machine instruction(s) to be emitted on a successful match; the opcodes and operands are, in general, macros that are expanded into assembler mnemonics of the target machine (e.g. emit Integer-Opcode, Address-form-of-Left-Operand, Address-form-of-Right-Operand).

The template-matching algorithm tries to match a subtree against suitable templates in an attempt to transform the subtree. Such transformations must consider the result location specified in the template. For an efficient implementation of the algorithm, it is essential to restrict the search for an acceptable template. A template matches a subtree when all the template specifications (1) through (5) match. Condition (4) includes a call to a resource allocator; the match fails if it is unable to allocate the required resources. On a failure, an attempt is made to transform the subtree using default or machine-dependent rewrite rules, for example,

```
a += b  becomes a = a + b
x++     becomes ((x += 1) -1)
```

The shortcomings of Johnson's approach are:

(1) Templates are not the only places where code selection is specified. Other phases of the compiler must emit code for storage allocation and subroutine prologue.

(2) The intermediate representation is specifically designed for the C language. Language dependent data types are embedded in the templates. Feldman uses C's code generator in his implementation of the portable Fortran 77 compiler [Feldman 79]. Most register and temporary allocation is taken care of by the code generator. However, mapping the different flavors of Fortran integer variables to C's types and generating the necessary type conversions are not easy tasks. Furthermore, Fortran's power operator (**) must be treated as a special case, and MIN and MAX functions are implemented as nested conditional expressions.

(3) Macro interpretation is used for selecting the assembler instruction from a set of possible instructions matching the subtree. Multiple matches between templates and subtrees are thus avoided, but these macros must be changed when the compiler is transported to a new machine.

(4) On a mismatch, machine-dependent rewrite rules call the code generator for possible tree alterations. Such rules potentially can produce infinite loops.

(5) Not much thought is given to machine specific optimizations. Condition codes are not saved between expressions.

4.2 Table-driven Code Generation

To suit a variety of target architectures, a lot of flexibility and tuning of the code generation algorithm is usually necessary. Lately, research has concentrated on providing this flexibility by an automatic analysis of a formal description of the target machine. A critique and survey of such research is presented under the following titles:

(1) Intermediate representation (IR) and code generation, and

(2) Machine analysis and code generator-generators (algorithms that produce code generators).

4.2.1 Intermediate Representation and Code Generation

Fraser [Fraser 77] uses ad-hoc rules (coded as MLISP [Smith 70] subroutines) to minimize machine dependency in code generation. He uses XL, a machine-independent IR that may need to be adapted to accommodate new source languages or target machines. Rules are used to perform storage allocation (more on this topic in the next section) and in this process XL is rewritten into ISP' [Wick 75] (a modified version of ISP [Bell 71]). Code generation then consists of matching this ISP' form with machine instruction patterns that are also in ISP'. Pattern mismatches invoke rules (subroutines written in MLISP) that try to rewrite the ISP' form of the IR. Examples of such rules include: invert relational tests and alter control flow, replace indirect references with indexed ones, load non-accumulator operands into accumulators. The rules, of course, do not guarantee that a code sequence will eventually be found. Fraser's knowledge-based approach has several shortcomings:

(1) Rules compromise generality for efficiency. They are based on the observation that computer architectures are similar in design (as Wick postulated in assemblers). The same rules are not usable for diverse architectures; often completely new rules are necessary. Some rules such as "load non-accumulator operands into accumulators" could potentially contradict other parts of a compiler (such as the register allocator). In Fraser's scheme, redundant loads and stores are unavoidable.

(2) It is hard, if not impossible, to utilize special instructions and addressing modes of a target machine. An XL primitive such as 'a = a+1' may match multiple machine instructions ('add #1,a' and 'inc a' on the PDP-11). It is not clear when (if ever) his code generator would resolve such multiple matches and choose the best alternative.

(3) The code generator is very slow: the implementation in Lisp on a PDP-10 KA10 generates one line of assembler code each second.

Glanville [Glanville 77, Glanville 78, Graham 80] chose a very low level IR in the form of Polish prefix expressions. Storage allocation and binding are assumed to be already done by other phases of a compiler. The code generation algorithm is derived from context-free parsing theory [Aho 73]. Instructions in the target machine are also expressed in prefix form and they form grammar productions with the left hand side (LHS) specifying the result of an operation and the right hand side (RHS) the operation. The assembler instruction computing the RHS is supplied with each production. Thus, r.1 -> + r.1 k=1 "inc r1" specifies that an addition of 1 to register1 (with the sum going to the same register) can be obtained by an "inc r1" instruction. A one-to-one mapping is assumed between productions (serving as machine templates to the

code generator) and target machine instructions. Since the addressing modes of operands are explicitly described as grammar terminals, this one-to-one restriction is essential. The IR string is parsed according to the context-free grammar and the appropriate assembler instructions are emitted. Since the grammar is usually ambiguous, a modified LR(1) parsing algorithm is used. The table driven code generator is automatically derived from instruction patterns (more on this technique in the next section). In practice, reasonably compact tables are obtained and also, because standard context-free parsing techniques (which forbid backup) are used, a linear time algorithm is obtained. Multiple matches produce shift-reduce or reduce-reduce conflicts and are resolved heuristicly. Shift-reduce conflicts are resolved in favor of a shift so that more powerful single machine-instructions are preferred to equivalent sequences of instructions. Similarly, reduce-reduce conflicts are resolved in favor of the production with the longer RHS. In case of conflicts between identical length productions, a "best instruction first" ordering is used to select the first production.

While Glanville's scheme is very efficient (easily the fastest among comparable code generation schemes) and provably correct, it is not truly portable because:

(1) The IR is very low level; it contains assumptions about the addressing structure of the target machine. The mapping between operators in the IR and target machine opcodes is required to be one-to-one. Thus, in transporting a compiler from one machine to another, changes have to be made to the IR. Such changes are reflected in Glanville's IRs for the PDP-11

and the IBM-360 (16 bit address computations as opposed to 24 bits). Since storage allocation and binding issues are avoided, any change in the implementation of (e.g.) the run-time display will result in changes to the IR code to access variables. Some interfacing problems, such as the allocation of registers that are used for display purposes, might arise between the register allocator and the display mechanism.

(2) Very good code cannot be generated by purely context-free expansion (e.g. 'a & b' in 'if (a & b)' and 'c := a & b' may need to yield different code because of the context in which it is used). Because this method uses limited context, the quality of generated code is strongly dependent on the exact IR form generated by the front end (e.g. in an addressing context, explicit addition is performed instead of using indexing).

(3) Heuristic resolution of multiple matches fails in certain cases (e.g., in the choice of two-address or three-address instructions on the VAX-11/780 [DEC 79]). Such cases can be resolved by using semantics to control the parser [Milton 77] (the interested reader is referred to [Ganapathi 80] for more details).

(4) The code generator does not worry about information retention (e.g. values left in registers from previous computations). Thus, redundant load and store elimination, recognition of equivalent locations, subsumption of addition or subtraction via auto-increment and decrement are not done.

As an extension and natural successor to Glanville's work, attribute grammars are used to specify translations from a linear representation of parse trees to a target code representation of programs [Ganapathi 80]. The intermediate representation is at a higher level than that proposed by Glanville. Semantics and context in the form of attributes are used to control parsing of the intermediate representation. Machine-dependent optimizations such as choosing between two-address and three-address instructions, using auto-increment and auto-decrement addressing modes are "cleanly" organized within the attributed parsing framework of code generation. Implementations of a code generator based on this model exist for the VAX-11/780 and the PDP-11/70. The results reveal better code quality than that produced by C compilers with their additional pass of peephole optimization.

Ripken [Ripken 77] uses an extended version of Aho & Johnson's algorithm to generate locally optimal code. His IR consists of attributed expression trees linked together as a graph according to the flow of control of the source program. The instruction set of the machine is described as attributed tree patterns with a set of attribute transformation (AT) rules (more details in the next section). A pre-pass to code generation maps simple (i.e. non-aggregate) source language types to characteristic value-ranges of machine storage locations.

Code generation then consists of a two-phase transformation of the IR. In the first phase, AT rules (derived from an analysis of the machine's AT rules) are used to generate code for expression trees. A machine operation is assumed to exist for every IR operator and its attribute values. A three-pass tree traversal scheme (very similar to Aho & Johnson's) determines the order of AT-rule applications and which AT rule is to be applied at each node. The difference between Aho & Johnson's algorithm and Ripken's is that Ripken considers real instruction sets with several register classes and addressing modes. Transfer operations (not only 'stores') between machine storage locations are also considered (sufficient transfer operations are assumed to allow operand transfer between all storage classes) together with register, temporary allocation and assignment. Like Aho & Johnson's, Ripken's first phase emphasizes locally optimal code. The second phase linearly arranges such locally optimal code blocks and generates the necessary branch instructions among them.

Ripken did not implement his proposal. A straightforward implementation would require a great deal of computation of different permutations with combinatorially explosive choices. A code generator based on this model would be very slow. Also, in spite of emphasis on optimal code generation, certain inefficiences are likely to occur at the border between code blocks (which represent individual statements rather than the basic blocks of [Aho 77]) of different expression trees. These inefficiencies include redundant loads and stores, and failure to take advantage of auto-increment/decrement possibilities.

Cattell [Cattell 78, Cattell 79, Cattell 80, Wulf 79, Wulf 80] uses TCOL (a tree-based intermediate representation) as the IR and a recursive tree traversing algorithm to generate code. Templates of the form 'tree pattern -> result sequence' are used to specify the translation from a TCOL program tree to machine code. The result sequence specifies code to be generated, calls to a register allocator or label generator, further matches to be recursively performed (e.g. a statement within a control construct). Templates are grouped into schemata representing the context (e.g. flow result, value result) in which code is to be generated. The code generator starts from the root of the IR tree and attempts to match templates with the largest possible subtree at the current tree node. On a match, the corresponding result sequence is processed. Templates must therefore be composed recursively to match an entire program tree. Operand mismatches are forcefully resolved by a subtargeting operation that consists of allocating a location of the desired data type and emitting a 'store' into that location. If an IR operator does not match any template operator, an attempt is made to transform the operator using tree equivalence axioms and heuristic search (details in the next section). Multiple matches are handled by sorting the alternatives with decreasing preference and choosing the first (e.g. x <- x+1 occurs before x <- x+constant). While Cattell's model is more general than Newcomer's (which only deals with arithmetic expressions), it has the following drawbacks:

(1) The code generator avoids machine-dependent issues such as binding variables to storage formats, space allocation and addressing of variables. The model fits only the 'Code' part of Bliss' [Wulf 75] Delay-Tnbind-Code-Final model. The allocation commands emitted by the code generator may conflict with the requirements of Tnbind [Johnsson 75]. Interfacing the code generator within Bliss' framework might therefore be hard.

(2) Templates are part of the code generation algorithm because some result sequences specify further matches to be performed. It is therefore hard to alter the templates without changing the algorithm.

(3) For subtargeting to be successful, there must be 'store' instructions in the target machine between all possible location types. Otherwise, the code generator may block generation of code for a valid program tree.

(4) Multiple matches are 'statically' resolved by ordering alternatives. This strategy does not result in optimal code sequences in certain cases. For example, on machines such as the VAX-11/780 that have both two-address and three-address operations for a single IR operator, the optimal choice depends on whether the operator is commutative and the operands are destroyable.

(5) Operator mismatches invoke a heuristic search that is recursive and combinatorially explosive. The search must be cut off at some point so that the code generator will not loop or use excessive amounts of time. Consequently, no machine code may be generated in cases where the search is cut off.

(6) Optimal code sequences are usually not produced. Special case subsumption operations such as auto-increment are hard to describe as templates. Also, equivalent locations are not recognized. Thus, if a register contains an operand that is also in a memory location, the code generator fails to identify this equivalence and use the register. Even if a value is already in a register, it is invariably reloaded. This reload happens because the code preceding the reload could possibly be generated from an arbitrarily distant section of the program tree and this optimization is therefore hard to recognize in any local tree context analysis. A separate peephole optimizer package [Mckeeman 70, Fraser 79, Fraser 80] may solve some of these problems but there may be conflicts with other parts of the compiler (such as the register allocator [Rudmik 79]).

4.2.2 Machine Analysis and Code-Generator Generators

In the previous section a number of code-generation techniques were analyzed with respect to their generality and target-code quality. However, another important issue is the variety of ways machine descriptions are utilized to perform code synthesis. It will be seen that these same techniques differ widely in the generality and depth of analyses they perform (e.g. in the formal correctness of code generation and range of machine-dependent aspects that are included).

Fraser performes syntactic analysis of ISP [Bell 71] descriptions at code generation time to recognize stack operations, macros that set condition codes, index registers and accumulators. Rules (subroutines in MLISP) are used to allocate storage for variables and classify registers as index registers and accumulators. Examples of such rules are "store integers in the widest possible memory that can participate in an add instruction" and "if a single instruction can add a register to some offset and use the result to index some memory then the register is an index register". On machines such as the IBM-360 or the PDP-11 where small integers can be stored in a half-word or a byte, the allocation rule for integers is inefficient. On architectures with no index registers (e.g. Intel 8080) the index register rule is useless. In general, Fraser's rules are ad-hoc and machine specific. Machine descriptions could be used as a substitute for some of these rules (it is not hard for the user to specify index registers and accumulators as part of the machine description).

Glanville's machine description (Polish prefix expressions) is not very formal. Different data types of the target machine (e.g. bytes, words, floating point) and special addressing modes (e.g. auto-increment, auto-decrement) are not used. The code generating IR parser is automatically constructed from the instruction-set description using an LR(1)-like table constructor [Aho 76]. Correctness of the code generator is emphasized. Possible looping configurations (where V $\Rightarrow^*$ V) are detected by analysis of grammar tables using a transitive closure algorithm on a relation characterizing parser moves. Instruction grammars are analyzed for uniformity (all operands being uniformly valid to operators independent of the context in which they appear). States are inspected to check that for all first symbols of left or right operands, either a shift or a reduce is signaled (i.e. no error actions are encountered). Although some of the semantics (e.g. register number, source-destination relationship) that are necessary to emitting instructions are used in productions, they are not used to control parsing. Sometimes semantic restrictions (e.g. constant required to have value 1 or required register usage) may not be satisfied for any production in the set of possible reductions in a particular state. In such cases, the action of the code generator is simulated with the semantically restricted instruction pattern using only those reduction rules with patterns shorter than the one under consideration. Default instruction lists for reduce states are thus automatically constructed. Action tables are changed to consider default reductions instead of signaling an error. This consideration ensures that a necessary set of conversion patterns has been supplied and thus that the code generator cannot block for a valid IR input.

In [Ganapathi 80], a more complete machine description is used by adding attributes to instruction-set productions (including machine data types and addressing mode productions). Storage allocation is viewed as part of the issue of portable code generation. This approach essentially retains all formal properties established by Glanville including:

(1) correctness of the code generation algorithm,
(2) detection of syntactic errors in the intermediate representation, and
(3) detection of incomplete instruction-set specification by blocking (instead of looping or generating incorrect code).

In Ripken's scheme, storage locations are described as pairs containing an operand class and address (e.g. (bytes, 15), (words, 16), (register, 2)). An operand is described by its address descriptor and value-range (e.g. 'n' bit, -2**(n-1) .. 2**(n-1), 2's complement). Operations are described by tree patterns (at least one pattern per IR operator) with predicates on attribute values and evaluation rules to describe the semantics.

E.g. addition on the Intel 8080; template: + O1 O2 -> O3

```
AT-rules:
    choice (1)
        predicates:
           cell_class(O1)          = accumulator
           cell_class(O2)          = immediate mode
           value(O2)               = 1
        evaluations:
           cell_class(O3)          := accumulator
           address(O3)             := address(O2)
           code                    :=                    inr A
           Z, S, P, AC affected

    choice (2)
        predicates:
           cell_class(O1)          = H and L register pair
           cell_class(O2)          = H and L register pair
           value_range(O1)         = F±15
           value_range(O2)         = F±15
        evaluations:
           cell_class(O3)          := H and L register pair
           value_range(O3)         := F±16
           code                    :=                    dad HL
           CY affected

    choices (3), (4)        similar to above.
```

From these AT-rules, IR operand specific application rules are selected for code generation. Ripken also requires templates for operand transfer between two storage classes even if the machine architecture does not have a 'move' instruction between them. This specification is needed so that transfer paths exist from any storage class to any other.

```
e.g.    :=  Register_pair  HL_pair
                mov 2*i, H      (i=address of register pair)
                mov 2*i+1, L
```

Addressing modes are also represented as tree templates. They are inserted in applicable places for operands in the IR tree before code generation. An attempt is made to subsume address computations by machine addressing modes.

Cattell proposes a formal model of instruction set processors (Mop: genealogically related to ISP) containing descriptions of storage locations, addressing modes and instructions. A set of assertions (in a parenthesized Lisp-like notation) are written for addressing modes and instructions. Such descriptions are significantly more useful for automating software than ISPS procedural descriptions. An attempt is made to derive code sequences for IR operators that do not have equivalent machine opcodes (e.g. subtraction on the PDP-8) by using tree equivalence axioms (e.g. DeMorgan's laws, relations between addition and subtraction) and heuristic search. The Mop assertion templates are then augmented with such derived sequences and pseudo-operations (utilizing side-effects of instructions to implement IR operators).

For example, consider 'c <- a & b' on the PDP-11 (which does not have a Boolean 'and'). Heuristic search obtains the closest machine instruction 'bic'. Means-end-analysis is then used to try matching 'c <- a & b' with 'c <- c & ~d' (assertion for 'bic d,c').

			code emitted
IR:	c <- a & b		
goal:	c <- c & ~d 'bic d,c'		
mismatch:	a with c, decomposition	'c <- a'	mov a,c
	b with ~d, transformation	'b <- ~~b'	
IR:	c <- c & ~~b		
goal:	c <- c & ~d 'bic d,c'		
mismatch:	~b with d, decomposition	'd <- ~b'	
heuristic search obtains 'com'			
IR:	d <- ~b		
goal:	d <- ~d 'com d'		
mismatch:	b with d, decomposition	'd <- b'	mov b,d
match:			com d
match:			bic d,c

Attempts are also made to match the IR with other potentially useful instructions ('c ← a & b' with 'c ← ~c' ('com')) but the search is too deep and subsequently cut-off before a code sequence can be found.

Such a heuristic search is too time consuming to be applied during code generation (on machines with condition codes, a conditional jump requires several transformations), so Cattell suggests doing such searches before code generation and tabulating the results for the code generator. In practice, it is very hard and time consuming (if not impossible) for such an axiomatic approach to automatically derive code sequences for floating point operations or doing a '2n'-bit arithmetic on an 'n'-bit machine (e.g. 16-bit arithmetic on the Intel 8080 or 32-bit arithmetic on the PDP-11). The Intel 8080 has no explicit 'branch on greater' or 'branch on equal' instructions. It has 'jz' (branch if zero flag is set) and 'jp' (branch if sign flag is clear). Code sequences for IR control-statements are very long,

e.g. 'Beq x y La' if x = y jump to La

```
assumptions:    x and y are 16 bit integers
                x is in register pair BC
                y is in memory addressed by the HL pair

mov A, M        accumulator ← y's low order byte
cmp C           compare x's low order byte with accumulator
jnz Lb          jump to Lb if the zero flag is not set
inx H           increment address register
mov A, M        accumulator ← y's higher order byte
cmp B           compare x's higher order byte with accumulator
jz  La          jump to La if zero flag is set
Lb:
```

The code for 'Bge x y La' (if $x \geq y$ jump to La) is twice as long!

5. Summary

There has been much theoretical research done in code generation. Very formal research has usually not considered real machine architectures. Interpretive approaches are improvements over ad-hoc code generation because only 'p+m' translators are needed to implement 'p' languages on 'm' architectures. But for such schemes machine descriptions are intermixed with the code generation algorithm. Retargeting thus requires changing the code generator for every new machine. Descriptive approaches separate the machine description from the code generation algorithm, thus providing a higher degree of portability. In such schemes, pattern matching is used to replace interpretation. Fraser, Glanville, Ripken and Cattell have tried to automatically derive code generators from a machine description though their methods are very much different. Fraser's rule based system is inefficient and its portability is questionable. Ripken has considered in detail the interaction between different phases in a compiler. However, an implementation of his dynamic programming algorithm can be expected to be prohibitively slow. Cattell uses a heuristic search algorithm to derive code sequences in cases of operator mismatch between the IR and target machine templates. Such automatic derivation using axioms is not practical for a variety of machine instructions. Glanville's scheme is best from the point of view of practicality. However, using his scheme in a production compiler requires a lot of machine-dependent work to be done by other phases of the compiler.

Since we have criticized other approaches to automatic code generation, it is but fair to evaluate our own technique [Ganapathi 80]. The shortcomings of our implementations are:

(1) Register allocation is not driven by machine description. (This shortcoming is questionable in view of recent architectures such as the Intel iAPX-432 that do not contain general purpose registers. In such cases, register allocation is an irrelevant issue.)

(2) Many optimizations are not extended beyond basic blocks.

(3) The code generator is not intelligent enough to automatically derive code sequences in cases of non-orthogonal instruction sets (Cattell's scheme could possibly derive code sequences). However, in our scheme it is fairly easy for the programmer to specify such code sequences.

(4) Certain compiler generated temporaries need not be allocated memory space since they can reside in registers for the entire duration of a procedure activation. In our implementation, although memory space is allocated for such temporaries, this storage space is never used because redundant stores are never emitted. Thus, code quality is not affected.

The following contributions of our technique are noteworthy:

(1) Conventionally, it has been an extremely difficult task to organize the different phases of a compiler. Our design of a flexible (attributed) Polish-prefix intermediate representation and attributed parsing framework seems to have solved this problem. Almost all machine-dependent aspects of compiler code generation have been isolated to a single software package.

(2) Attributes in the intermediate representation have provided a convenient interface between the machine-independent and the machine-dependent parts of a compiler. They have helped in solving the difficult problem of operand binding (an issue not addressed by other researchers in the area of automatic code generation).

(3) Machine-dependent and peephole optimizations have been incorporated in a routine, cheap and reliable manner within the attributed parsing framework of code generation. Our attempt seems to be the first to organize optimization within any framework.

The important point to note is that an amazingly wide variety of code generation optimizations can be realized in a highly modular manner. In particular, a simple (but unoptimized) code generator can be implemented for a machine easily and rapidly. Then, as time permits, and the need arises, improvements can be added by simply including new rules to the machine description (and automatically regenerating the code generator). In effect, the chief difference between an optimized and an unoptimized code generator is how carefully and thoroughly the attributed production rules used reflect the details and complexities of the target machine.

Acknowledgements

We gratefully appreciate the help provided by Johannes Heigert, Raphael Finkel and William Leland in improving the readability of this paper.

Bibliography

[Aho 73] A.V. Aho and J.D. Ullman, "The Theory of Parsing, Translation and Compiling", Vols. 1 and 2, Prentice-Hall, Inc., 1973.

[Aho 76] A.V. Aho and S.C. Johnson, "Optimal Code Generation for Expression Trees", JACM Vol. 23 No. 3 pp. 488-501, 1976.

[Aho 77] A.V. Aho and J.D. Ullman, "Principles of Compiler Design", Addison-Wesley publishing Co., 1977.

[Ammann 77] U. Ammann, "On Code Generation in a Pascal Compiler", Software-Practice and Experience, Vol. 7 No. 3 pp. 391-423, June/July 1977.

[Bell 71] C.G. Bell and A. Newell, "Computer Structures: Readings and Examples", McGraw Hill, 1971.

[Cattell 78] R.G.G. Cattell, "Formalization and Automatic Derivation of Code Generators", PhD thesis, Carnegie Mellon University, 1978.

[Cattell 79] R.G.G. Cattell, J.M. Newcomer and B.W. Leverett, "Code Generation in a Machine-Independent Compiler", ACM Sigplan Symp. Compiler Construction, Boulder, Colo., Aug. 1979.

[Cattell 80] R.G.G. Cattell, "Automatic Derivation of Code Generators from Machine Descriptions", ACM Trans. Programming Languages and Systems, Vol. 2, No. 2 pp. 173-190, April 1980.

[DEC 79] Digital Equipment Corporation, VAX 11/780 Architecture Handbook.

[Donegan 73] M.K. Donegan, "An Approach to the Automatic Generation of Code Generators", PhD thesis, Rice University, Houston, Texas, 1973.

[Donegan 79] M.K. Donegan et.al., "A Code Generator Language", ACM Sigplan Symp. Compiler Construction, Boulder, Colo., Aug. 1979.

[Elson 70] M. Elson and S.T. Rake, "Code Generation Technique for Large Language Compilers", I.B.M. Systems Journal Vol. 9 No. 3 pp. 166-188, 1970.

[Ershov 58] A.P. Ershov, "On Programming of Arithmetic Operations", CACM Vol.1 No. 8 pp. 3-6, 1958.

[Feldman 79] S.I. Feldman, "Implementation of a Portable Fortran 77 Compiler using Modern Tools", ACM Sigplan Symp. Compiler Construction, Boulder, Colo., Aug. 1979.

[Fraser 77] C.W. Fraser, "Automatic Generation of Code Generators", PhD thesis, Computer Science Department, Yale University, New Haven, Conn., 1977.

[Fraser 79] C.W. Fraser, "A Compact Machine Independent Peephole Optimizer", Principles Of Programming Languages, 1979.

[Fraser 80] C.W. Fraser and J.W. Davidson, "The Design and Application of a Retargetable Peephole Optimizer", ACM Transactions on Programming Languages and Systems, Vol. 2 No. 2, 1980.

[Ganapathi 80] M. Ganapathi, "Retargetable Code Generation and Optimization using Attribute Grammars", PhD dissertation, University of Wisconsin - Madison, 1980.

[Glanville 77] R.S. Glanville, "A Machine Independent Algorithm for Code Generation and its Use in Retargetable Compilers", PhD thesis, University of California, Berkeley, Dec. 1977.

[Glanville 78] R.S. Glanville and S.L. Graham, "A New Method for Compiler Code Generation", Conf. Record Fifth ACM Symp. Principles of Programming Languages, Jan. 1978.

[Graham 80] S.L. Graham, "Table-Driven Code Generation", IEEE Computer, Vol. 13 No. 8 pp. 25-34, August 1980.

[Johnson 77] S.C. Johnson, "A Tour through the Portable C Compiler", Bell Telephone Laboratories, 1977.

[Johnson 78] S.C. Johnson, "A Portable Compiler: Theory and Practice", Proc. 5th ACM Symp. Principles of Programming Languages, pp. 97-104, Jan 1978.

[Johnsson 75] R.K. Johnsson, "An Approach to Global Register Allocation", PhD dissertation, Carnegie-Mellon University, 1975.

[McCarthy 63] J. McCarthy, "A Basis for a Mathematical Theory of Computation", Computer Programming and Formal Systems (Eds. Braffort and Hirshberg), North Holland, Amsterdam, 1963.

[Mckeeman 70] W.M. Mckeeman, "Peephole Optimization", CACM Vol. 8 No. 7 pp. 443-444, 1970.

[Miller 71] P.L. Miller, "Automatic Creation of a Code Generator from a Machine Description", M.I.T. Tech report MAC TR-85, 1971.

[Milton 77] D.R. Milton, "Syntactic Specification and Analysis with Attribute Grammars", PhD thesis, University of Wisconsin-Madison, 1977.

[Newcomer 75] J.M. Newcomer, "Machine Independent Generation of Optimized Local Code", PhD thesis, Computer Science Department, Carnegie Mellon University, 1975.

[Newell 69] A. Newell and G.W. Ernst, "GPS: A Case Study in Generality and Problem Solving", Academic Press, 1969.

[Richards 71] M. Richards, "The Portability of the BCPL Compiler", Software Practice and Experience, 1, pp. 135-146, 1971.

[Ripken 77] K. Ripken, "Formale Beschreibun von Maschinen, Implementierungen und Optimierender Maschinen-codeerzeugung aus Attributierten Programmgraphe", Technische Univer. Munchen, Munich, Germany, July 1977.

[Ritchie 78] D.M. Ritchie and B.W. Kernighan, "The C Programming Language", Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[Rudmik 79] A. Rudmik and E.S. Lee, "Compiler Design for Efficient Code Generation and Program Optimization", ACM Sigplan Symp. Compiler Construction, Boulder, Colo., Aug. 1979.

[Smith 70] D.C. Smith, MLISP, Stanford Artificial Intelligence Project Memo AIM-135, Stanford University, 1970.

[Snyder 75] A. Snyder, "A Portable Compiler for the Language C", master's thesis, MIT, Cambridge, Mass., May 1975.

[Steel 61] T.B. Steel, Jr., "A First Version of UNCOL", Proceedings WJCC, 19, pp. 371-378, 1961.

[Strong 58] J. Strong et. al., "The Problem of Programming Communication with Changing Machines: A Proposed Solution", CACM Vol.1 No. 8 pp. 12-18, 1958.

[Weingart 73] S.W. Weingart, "An Efficient and Systematic Method of Compiler Code Generation", PhD thesis, Computer Sciences Department, Yale University, 1973.

[Wick 75] J.D. Wick, "Automatic Generation of Assemblers", PhD Dissertation, Yale University, 1975.

[Wilcox 71] T.R. Wilcox, "Generating Machine Code for High Level Programming Languages", Tech. report 71-103, PhD thesis, Department of Computer Sciences, Cornell University, 1971.

[Wulf 75] W. Wulf et. al. "The Design of an Optimizing Compiler", American Elsevier Publishing Co., 1975.

[Wulf 79] W. Wulf et. al., "An Overview of the Production Quality Compiler-Compiler Project", Tech. Report CMU-CS-79-105, Carnegie Mellon University, Feb. 1979.

[Wulf 80] W. Wulf et. al., "An Overview of the Production-Quality Compiler-Compiler Project", IEEE Computer Vol. 13 No. 8 pp. 38-49, August 1980.

[Young 74] R. Young, "The Coder: A Program Module for Code Generation in High Level Language Compilers", M.S. thesis, Computer Sciences Department, University of Illinois, 1974.