A KNOWLEDGE-BASED CODE GENERATOR GENERATOR*

Christopher W. Fraser†
Department of Computer Science
Yale University
New Haven, CT 06520

XGEN is a program that accepts a machine description and produces a good local code generator for an ALGOL-like language. It is organized as a production system of rules codifying previously acquired human skills for dealing with computer architecture and programming languages.

1. INTRODUCTION

The proliferation of machines and programming languages motivates automatic generation of compilers. Automatic generation of some compiler modules is already available; for example, parsers may be generated from syntax specifications. Automatic generation of some others is moot; for example, much global optimization is largely independent of both language [8] and machine [12]. However, the machine-dependent modules, particularly the local code generator, have resisted automation. The immediate goal of this research is a working program that derives a good local code generator for an ALGOL-like language (e.g., ALGOL, FORTRAN, PASCAL) from a simple, general machine description. The program is called XGEN.

Writing a program requires a thorough understanding of the problem at hand. So writing a program to write code generators requires a detailed understanding of the impact of computer architecture on languages and compilers. The larger goal of Yale's "automatic generation of ..." series (see also Wick's Automatic Generation of Assemblers [14]) is this understanding of common programming tasks as a step toward more general automatic programming. Since it will influence our choice of techniques, a discussion of our view of automatic programming is germane.

Automatic programming is getting programs to do what programmers do. Programmers define, write, optimize, verify, and document programs, drawing from a large base of programming skills or canned solutions. They also use more general problem-solving abilities, but just as skill sets programmers apart from other human problem-solvers, so skill sets automatic programming apart from more general artificial intelligence. Automatic programming research codifies this skill.

We proceed from programming tasks that are well-understood and worth automating. Assemblers and simple local code generators are obvious candidates. We'll understand them no better in five years; now is as good a time as any to seek automation. We define them, circumscribing the class of programs that assemble or generate code. Then we study, for these examples, machines, isolating the parameters that define a member of the class. Then we write programs to deduce these parameters from machine descriptions.

2. ORGANIZATION

Much past automatic programming research has emphasized abstract problem-solving over world knowledge about programming. For example, theorem-proving systems extract programs from proofs of output assertions [9]. Means-end analysis recursively chooses the operator (defined by its state changes) that most closely achieves the specified goal state [10, 13]. Such approaches, though elegant and general, do not meet our needs for an automatic programming system restricted to code generators. Their elegance, plain when dealing with abstract LISP additions and integers, is lost on the ad hoc domain of real machines with finite data representations and add instructions with side-effects on the program counter and status bits. Their generality forces them to resort to unacceptably slow exponential searches and to assume so little that the description of a code generator may be quite verbose. Finally, they tell us little about programming: human programmers do not write programs by proving theorems.

Instead of abstract problem solving, XGEN emphasizes specific assembly language programming skills that humans apply daily. XGEN is a production system, successively applying rules that encode bits of this knowledge. These rules specify, for example, that a conditional jump may be implemented with a skip/jump, that a loop may be massaged to use a subtract-index-and-test instruction, and that a memory structure 8 bits wide is probably for characters. A rule may not apply to all machines, but it usually applies to several.

This organization has its strengths and limitations. It is easy to understand what XGEN does with its input. It is easy to enhance XGEN to recognize new architectures, to compile new language constructs and to interface with new automatic programming systems. It is hard, though, to anti-

* This work was supported by an IBM Graduate Fellowship.

† Present address: Department of Computer Science, The University of Arizona, Tucson, AZ 85721.

cipate enough about coming needs to finalize XGEN; like the human programmer, XGEN may sometimes need new rules (entered by hand) for new architectures or languages. It is easy to generate good code by case analysis. It is hard, though, to guarantee optimal code, since XGEN proceeds from a base of programming "tricks" rather than an abstract notion of the problem and its complexity. This organization was chosen for its utility; the theoretical limitations cause few problems in practice. It is useful for higher-level programming, too; the PSI automatic programming system uses similar techniques to code symbol manipulation programs [5].

XGEN addresses only machine-dependent problems. It ignores code generation phases that require no machine analysis. Automatic generation of these is redundant. For example, XGEN ignores most global optimizations (e.g., constant propagation, strength reduction), which depend little on the source language [8] or target machine [12]. XGEN also ignores global register allocation, which requires few more machine-dependent parameters than the XGEN-supplied number and types of registers available [2]. XGEN is responsible for local register allocation, smoothing instruction set asymmetries.

3. THE MACHINE DESCRIPTIONS

An automatic programming system requires a rigorous machine description. Use of Bell and Newell's ISP machine description language [3] guarantees wide application and builds on previous work [14]. An ISP description is just a program defining the machine's instruction interpreter. For example, an IBM 360 ISP contains declarations for the machine's registers and memory (viewed as 8, 16 and 32 bits wide, with alignment constraints):

```
R [0:15]<0:31>

M1[0:262143]  <0:7>
M2[0:262142:2]<0:15> := M1[0:262143]<0:7>
M4[0:262140:4]<0:31> := M1[0:262143]<0:7>.
```

Short programs define the machine's operation. For example, the IBM 360 indexes when the X2 instruction field is nonzero; otherwise, the indexed address is just the base-displacement address:

```
RX := (X2 eq 0 => BD; X2 ne 0 => R[X2] + BD).
```

Programs also define such often-used ISP "macros" as condition-code setting:

```
CCA(X) := (
  X eq 0 => 0 → CC;
  X lt 0 => 1 → CC;
  X gt 0 => 2 → CC).
```

Finally, programs define the instructions. The IBM 360 Load instruction (opcode 88) loads the fullword indicated by the indexed address calculation into the register indicated by the instruction field R1:

```
'L'  (:= OP eq 88) => M4[RX] →  R[R1].
```

At the level of detail required for code generation, ISPs are typically a page or two and may be produced in a few hours by someone experienced with the machine.

XGEN ignores those exotic machine features rarely understood by code generators (e.g., virtual memory, timers, I/O, interrupts). XGEN also assumes a pleasant, idiomatic ISP: adds must be expressed with the add operator, not with primitive bit manipulation and not by counting one operand up while counting the other down to zero. The special case of recognizing equivalent additions is, at best, unpleasant; the general case of recognizing equivalent ISP programs is undecidable. Human programmers aren't asked to recognize such equivalences; the Principles of Operation manual [7] gives them a pleasant machine description. ISP should give their automatic counterparts the same.

4. THE LANGUAGE COMPILED

XGEN produces good local code generators. They compile XL, a language of tuples suited to describing local behavior. Parse trees and Polish strings must be broken down, one operation per tuple. XL resembles other languages for code generation (see Gries' triples and quadruples [6]).

To guarantee flexibility and to focus on just the machine-dependent issues, XL is as low-level as is possible without interfering with the goal of portable, efficient programs. For example, XL has only static variables; dynamic allocation must be built from such XL primitives as stack operations. XL resembles, and may be used as, a high-level machine-independent assembly language.

XL operations include moves, combinatorics (monadic and dyadic), jumps (conditional and unconditional), loops, calls and returns. XL datatypes are integer, character and pointer variables, registers, constants, vectors, strings and stacks. XL references may be by address, by value, indirect, subscripted or one of several more complex pointer references.

The flavor of XL is given by the program below that finds the maximal element of a vector:

```
mem int m  0   ;max found so far, initially 0.
vec int a 10   ;vector to be scanned, 10 long.
...
               ;a[0] = n = actual vector length.
a[a[0]] → m    ;max found so far is last element.

reg ptr k                        ;declare loop index.
for k ← a[0]-1 to 1 by -1        ;k ← n-1, n-2, ... 1.
  a[k] le m => jump t1             ;le => cycle.
  a[k] → m                         ;gt => update max.
  t1:  end                         ; end of loop.
```

Some XL constructs may seem rather high-level. For example, XL expresses loops as primitive operations, collecting the direction and the start, step and stop values rather than scattering them over many tuples, as some code generation languages require. Such "large" primitives are used only when this additional context simplifies the generation of "large" instructions, in this case, subtract-index-and-test instructions.

5. USING XGEN

To generate a code generator, we first invoke Wick's assembler generator [14], producing a basic, conventional assembler and syntactically simpler ISP. This is presented to XGEN, which does some analysis immediately. Registers are classified as accumulators and index registers. Memories (e.g., M1, M2 and M4 above) are analyzed to determine width and alignment for the various datatypes. Instructions are classified according to type (e.g., move, jump). ISP macros representing common idioms are so classified (e.g., setting the

condition code, stacking operands).

This instantiated version of XGEN is saved as a code generator. The code generator currently generated is interpreted, not compiled, in the sense that code generation still invokes ISP analysis. This is a matter of convenience, not necessity; all ISP queries could be made when the ISP is loaded with a simple but tedious series of changes to XGEN.

This specialized XGEN now accepts XL statements. If it finds an instruction implementing the operation, it returns the associated assembly language code. Otherwise, it tries rules until one succeeds, typically decomposing the XL into two simpler operations. XGEN iterates, compiling these. This is best understood by watching XGEN in action. Consider the compilation of IBM 360 code for this fragment of the XL program above:

```
mem int m  0
vec int a 10
reg ptr k
a[k] le m => jump t1.
```

To process the declarations, XGEN must learn the width and alignment for integers and the names of the index registers; the assembler handles encoding. Among the rules (translated from LISP to English for this presentation) applied to each ISP variable X are:

> If X is the widest memory overlay for which {? + X[?] → ?} matches some instruction then X is the integer memory. (? indicates that XGEN doesn't care what appears in that position).
>
> If {M[X+?] → ?} matches some instruction for some memory M then X is an index register.

The first rule selects M4 as the integer memory; its width and alignment define the integer storage format. Substituting this machine-dependent data into general templates gives this compilation for the declarations:

```
align 4;  m:  field <32> 0;
align 4;  a:  org .+40.
```

The second rule classifies R[1:15] as index registers. Declaring k an index register picks one of these (say, R[3]), marks it busy and equates it with k. No code is generated.

XGEN begins compiling the conditional branch by rewriting it as if in the IBM 360 ISP: R[3] must be substituted for k, multiplied by 4 (perhaps by shifting), added to the vector's base address and used to index the integer memory, M4; the jump must be replaced by an assignment to the ISP variable tagged as the program counter by the assembler generator:

```
M4[a + R[3] shift 2] le M4[m] => t1 → PC.
```

XGEN first applies rules that simplify subscript calculations:

> If compiling some operation with a subscripted operand that is not in an index register then allocate one, load it with the subscript and change the original operation to reference it.

So XGEN allocates another index register (say, R[4]) and produces:

```
R[3] shift 2 → R[4]
M4[a+R[4]] le M4[m] => t1 → PC.
```

Subscripting simplified, XGEN applies rules tailored to the input tuple's operation class. These rules describe common hardware realizations of, for example, binary combinatorials and conditional jumps. The rules that apply to our example are:

> If compiling {x shift y → z}, and z specifies an accumulator different from x then compile {x → z next z shift y → z}. (The rule actually transforms all binary combinatorial operators, not just shift. "Next" denotes sequential, as opposed to parallel, execution.)
>
> If compiling {x le y => 1 → PC}, and the machine uses a condition code (CC) then compile {SET-CC(x-y) next LE-MASK[CC] => 1 → PC}. (When the ISP was loaded, XGEN recognized the routine that sets the condition code, defining the name "SET-CC" and the mask "LE-MASK." The rule actually transforms all relationals, not just le.)

The two lines expand to four:

```
R[3] → R[4]
R[4] shift 2 → R[4]
CCA(M4[a+R[4]]-M4[m])
1100[CC] => t1 → PC.
```

Only the third isn't an IBM 360 instruction. It is expanded by one of XGEN's last rules, which loads and stores registers to conform to the requirements of the instructions chosen above. The applicable rule is:

> If compiling some operation with an operand that is not in an accumulator and if the operation otherwise matches some instruction then allocate an accumulator, load it with the operand and change the original operation to reference it.

XGEN allocates an accumulator (say, R[5]) and the program becomes, finally:

```
R[3] → R[4]
R[4] shift 2 → R[4]
M4[a+R[4]] → R[5]
CCA(R[5]-M4[m])
1100[CC] => t1 → PC.
```

This compiles into:

```
LR  4,3
SLA 4,2
L   5,a(4)
C   5,m
BC  1100,t1.
```

6. PERFORMANCE

XGEN must be evaluated along several different dimensions. First, how good is the generated code? The example above is typical. Locally, the code was optimal; XGEN did as well as possible with the original input tuple. Even global performance, not entirely XGEN's responsibility, is good. The entire program compiles into 18 instructions; with global (i.e., inter-tuple) register allocation, 14. To do better, we must go beyond the scope of traditional compilers and change the algorithm slightly to get an apparently optimal 12 instructions. Metrics other than instruction counts yield similar comparisons. Good, but not always

optimal, code is typical.

How much effort goes into generating a code generator? XGEN has completed code generators for Perlis' pedagogical AJP21A [11], the PDP10 and the IBM 360. XGEN's rule base grew as follows:

AJP21A	30 rules	4 man-months
PDP10	+10 rules	+4 man-weeks
IBM360	+ 5 rules	+10 man-days.

This convergence was predicted by the arguments for knowledge-based code generator generation, and borne out by the example above: only the "condition code" rule applies to the IBM 360 and not the PDP10 and the AJP21A. The convergence is by no means absolute. The present rule base suffices for simple, conventional machines. Machines with a few characteristics new to XGEN (e.g., microprocessors, the PDP11, the CDC 6000 series) might require five new rules apiece. Exotic architectures (e.g., the CDC STAR) would defeat many existing rules and require substantial effort; indeed, XL and ISP, unenhanced, are unsuitable for programming and describing such machines.

How much does it cost to run XGEN? This experimental XGEN was developed in LISP, runs in 55K words on a PDP10 KA10 and generates a line of assembly code each second. A production version eliminating the current multi-level interpretation would require substantially less time and space. XGEN can be made practical.

7. UNDERSTANDING CODE GENERATION

The knowledge gained when automating a process is usually more detailed than profound. XGEN's contributions to our understanding of code generation are no exception. Ignoring them, however, is ignoring the experience that "the quality of the local code has a greater impact on both the size and speed of the final program than any other optimization" [15]. They take the form of many little hints about languages and machines: strings should be distinguished from character vectors (sequential versus random access) to suggest the storage format on word-addressed machines (packed versus unpacked); integers should be declared with ranges to permit small ones to be stored as halfwords on the IBM 360 and as 8-bit bytes on microprocessors. Most are too complex to present here [4]. Taken one at a time, they each seem obvious, trivial observations. Taken together, they define what code generators must know about programs and machines to produce good code.

8. SUMMARY

XGEN is a useful tool. It contributes to our understanding of code generation, demonstrates the viability of knowledge-based code generator generation and encourages knowledge-based automatic programming. Work in progress includes enhancements for new architectures and languages, as well as the incorporation of XGEN in a larger compiler generator.

ACKNOWLEDGEMENTS

Alan Perlis guided this dissertation research. David Hanson suggested several improvements to this paper.

REFERENCES

1] F. E. Allen and J. Cocke.
A catalogue of optimizing transformations.
In R. Rustin, editor,
Design and Optimization of Compilers, 1-30.
Prentice-Hall, 1972.

2] J. Beatty.
A register assignment algorithm for the generation of highly optimized object code.
IBM Journal of Research and Development 18(1): 20-39, January 1974.

3] C. G. Bell and A. Newell.
Computer Structures: Readings and Examples.
McGraw-Hill, 1971.

4] C. W. Fraser.
Automatic Generation of Code Generators.
Ph.D. dissertation, Yale University, in preparation, 1977.

5] C. Green.
The design of the PSI program synthesis system.
In *Proceedings of the 2nd International Conference on Software Engineering*, October 1976.

6] D. Gries.
Compiler Construction for Digital Computers.
Wiley, 1971.

7] IBM.
IBM System/360 Principles of Operation.
IBM, 1968.

8] E. S. Lowry and C. W. Medlock.
Object code optimization.
CACM 12(1):13-22, January 1969.

9] Z. Manna and R. J. Waldinger.
Toward automatic program synthesis.
CACM 14(3):151-165, March 1971.

10] J. M. Newcomer.
Machine-Independent Generation of Optimal Local Code.
Ph.D. thesis, Carnegie-Mellon University, May 1975.

11] A. J. Perlis.
Introduction to Computer Science.
Preliminary edition, Harper and Row, 1972.

12] P. B. Schneck and E. Angel.
A FORTRAN to FORTRAN optimising compiler.
Computer Journal 16(4):322-330, November 73.

13] H. A. Simon.
Experiments with a heuristic compiler.
JACM 10(4):493-506, October 1963.

14] J. D. Wick.
Automatic Generation of Assemblers.
Ph.D. dissertation, Yale University, December 1975.

15] W. Wulf, R. K. Johnsson, C. B. Weinstock, S. O. Hobbs and C. M. Geschke.
The Design of the Optimizing Compiler.
American Elsevier, 1975.