Experience with an Automatic Code Generator Generator

Rudolf Landwehr, Hans-Stephan Jansohn, Gerhard Goos
Universität Karlsruhe, Institut für Informatik II

Abstract :

We implemented an automatic code generator generator based on the approach of Glanville and Graham. We describe our experience with this system and compare the generated code with that of a conventional Pascal compiler.

1 Introduction

Recent research in code generation focused on systems that automatically generate code generators from machine descriptions ([Cattell 78, Ganapathi 80, Glanville 78, Kozlak 81], more references in [Ganapathi/Fischer 81]). The aim of this research is to formalize and automate the construction of compiler back ends similar to the use of parser generators, attribute grammars and other tools in front ends.

A code generator has to perform several tasks that are more or less dependent on the target machine. The most important tasks are type mapping, memory allocation and code selection. In this paper, we will concentrate on the issue of code selection, the final mapping of the source program into a sequence of target instructions.

Present automatic code generator generators reduce code selection to a pattern matching problem. The program to be compiled is given in some intermediate representation IR. The effect of the instructions is described by patterns in terms of the IR. Instructions can be selected if their pattern is matched in the program. The various code generator generators use different algorithms to implement this pattern matching.

We have implemented an automatic code generator generator CGSS [Jansohn/Landwehr 80] on the basis of the approach taken by Glanville and Graham [Glanville 78]. With this system code generators for Pascal have been produced for the processor types Siemens 7.000 and Motorola 68000. In the following, we discuss the main aspects of this system, with special emphasis on the differences between our system and that of Glanville as described in his thesis. Chapter three reports our experience with using CGSS in a Pascal compiler.

2 The Code Generator Synthesis System CGSS

2.1 The Code Selection Algorithm

As mentioned above, code selection can be regarded as a special kind of pattern matching. In our system the pattern matching process is divided into two steps: first, a syntactic parse of the intermediate representation to recognize a set of patterns and, second, the selection of one of these patterns according to a cost function and the evaluation of attributes.

The intermediate representation of the program to be compiled is a sequence of operator trees, where each tree represents an elementary statement of the input program, e.g. an assignment. For the syntactic parse, the trees are linearized in prefix order.

The syntactic parse is done with a modified LALR(1)-parser. For this purpose a context-free production is associated with each instruction of the target machine. The right hand side of this production represents the subtree that is matched by the instruction. The left hand side is the replacement for this subtree representing the result of the instruction. As an example, consider an add immediate instruction ADDI that adds a constant to a register. The associated production is

register ::= plus register constant

Several instructions can be associated with the same production. An increment instruction INCR that adds the constant one to a register would have the same production as the ADDI instruction.

The context-free grammar constructed from the collection of these productions is usually ambiguous, a consequence of the fact that there are normally several possible code sequences for the same source statement. Because of this ambiguity, and since several instructions can be associated with the same production, the syntactic parse can only determine a set of instructions. The selection of a single instruction from this set has to be controlled by other means.

The instructions in the set may differ in their cost, for example code size or execution time, or in restrictions they impose on their operands. The INCR instruction above, for example, can only be used if the constant to be added has the value one.

After the parse has determined a subtree that can be matched by one or more patterns, the code generator will test the conditions of the corresponding instructions, for instance whether a constant belongs to a certain range. Only instructions with valid conditions are applicable. If there remains more than one applicable instruction, a cost function is applied to determine the cheapest instruction. To check the restrictions the code generator uses attributes that are either given in the intermediate representation or computed during code selection.

After the code generator has selected a single instruction, additional actions specific for this instruction are performed, for example, calls to the register allocator or computation of attributes.

In order to generate the code generator, the CGSS needs informations about the instruction set of the target machine, especially about the productions, restrictions and additional actions associated with each instruction. This information is given in a "target machine description". The following chapter gives a summary of the form of a target machine description used in the CGSS.

2.2 The Target Machine Description

The target machine description (TMD) is the input to the CGSS. It consists of a definition of the intermediate representation (IR) that serves as input to the code generator and of a collection of instruction descriptions called "rules". Each rule must contain

- the context-free production for the instruction
- the semantic conditions that must be fulfilled if the instruction shall be selected
- the cost of the instruction, e.g. its size or execution time
- the actions that must be performed when the instruction is selected, e.g. emitting the instruction to the code file or calling the register allocator.

We were faced with severe difficulties when we tried to formalize the conditions and actions of a rule because of the diversity of the instruction sets of different machines. The target machine description language used by Glanville, for example, does not allow the correct description of the IBM 360 integer multiplication instructions M and MR.

We circumvent this problem by allowing in the TMD Pascal expressions and statements. (Pascal is the implementation language of the generated code generator as well as that of the CGSS.) Access to the attributes of a symbol in the context-free production is allowed via identifiers attached to these symbols.

As an example, the description of an increment instruction INCR and an add immediate instruction ADDI as described in section 2.1 will be as follows:

```
RULE register.r0
        ::= plus register.r1 constant.c
         |  plus constant.c  register.r1;
  COST      2; (* size 2 bytes *)
  CONDITION c.value = 1;
  ACTION
    r0.number := r1.number;
    writeln (symcode, 'INCR ', r0.number);
END_RULE;

RULE register.r0
        ::= plus register.r1 constant.c
         |  plus constant.c  register.r1;
  COST      4; (* size 4 bytes *)
  (* no condition,all constants allowed *)
  ACTION
    r0.number := r1.number;
    writeln (symcode, 'ADDI ',r0.number
                              ,c.value );
END_RULE;
```

As already mentioned, the input to the code generator is a sequence of trees, each tree representing a source statement. We call a node in such a tree an operator. Each operator has a fixed number of operands (plus has two, constant has zero operands) and may have arbitrary attributes. All operators are defined in the TMD, and each definition of an operator op contains the following information :

- the name of the operator
- a code for this operator, a natural number for its identification
- the attributes of the operator
- the number of operands and
- for each operand i the set FIRST(op, i) of operators which may occur as the first operator of this operand in the input to the code generator.

The last requirement yields an implicit definition of the possible input to the code generator and is used for uniformity checking (see 2.3). The definitions of the operators in the example above are

```
register [CODE = 1; number : 0 .. 15];
constant [CODE = 2; value  : integer];
plus     [CODE = 3]
         (plus, register, constant)
         (plus, register, constant);
```

Some instructions do not return a result, e.g. branch or store instructions. These instructions normally correspond to operators in the IR that also have no result, such as an assign operator. We call such an operator a STMT-operator, and as the left hand side of a production describing an instruction with no result we use the symbol STMT, as in

STMT ::= assign address.a register.r

for STORE r,a. The set of operators that are STMT-operators is also defined in the TMD.

2.3 Reliability of the Generated Code Generator

In view of the many special cases which have to be considered in a code generator it is very difficult for the compiler writer to check that every combination is correctly handled, especially if the code generator is used for some years and has been changed by several people. The essential advantage of using the pattern matching approach is due to the splitting of the code selection task into manageable pieces: pattern matching, attribute evaluation, decision within a small set of possible instructions for a pattern. The proper cooperation of these pieces is guaranteed by the generator.

If the description of each instruction is correct (this is quite easily checked since the length of a rule is 10 lines on the average), one of the following errors may occur during code generation :

(1) The code could be emitted in wrong order.

(2) The parser may loop (due to the ambiguity of the grammar).

(3) The parser may enter an error state (syntactic blocking).

(4) All instructions corresponding to a given syntactic pattern have conditions that evaluate to false (semantic blocking).

As is shown below, the CGSS will either avoid these errors or give an error message at code generator generation time, so the code generator will produce correct code for every legal input. For a non-legal input the parser will enter an error state and the compilation is aborted after issuing an error message. Note that this can only occur if another part of the compiler is erroneous.

(1) Correct code in correct order:

The code is generated by the code generator in a single bottom up, left to right tree traversal, parallel to the LR-parsing of the intermediate representation. Therefore the code is generated in a correct order provided a postfix tree traversal of the IR specifies a legal execution order. If any reordering is required for optimization purposes, it must be performed before the code is generated.

(2) Looping:

Loops can only occur when the parser reduces according to a sequence of chain productions such as X -> Y -> Z -> X. Each loop can be broken up by changing the resolution of a reduce/reduce-conflict. The CGSS does this automatically. The algorithm is similar to that of Glanville. However, it uses and modifies

the generated parser tables instead of the item set.

(3) Syntactic blocking:

The input language for the code generator is a sequence of IR-"statements", where each statement is a prefix expression starting with a STMT-operator and each operand i of an operator op starts with an operator belonging to the set FIRST(op,i) (see 2.2). The code generator will not syntactically block if the language accepted by the parser includes the input language. We call the parser "uniform" if this condition holds. The special form of the input language and the productions in the machine description allow to check the uniformity of the generated parser in the CGSS. To this end the system computes for each state q in the parser the pairs (op,i) where i is the operand of the operator op expected when the parser is in state q. The parser is uniform if for all op1 in FIRST(op,i) the parser action for state q and lookahead op1 is admissable.

Non-uniform situations seem to be rare if the full instruction sets of real machines are described (but they do occur!). They are more often the consequence of an incomplete TMD, and the uniformity test exhibits that something is still missing.

(4) Semantic blocking:

If the conditions for all instructions associated with a given syntactic pattern evaluate to false, the code generator cannot select an instruction and will block.

The situation will only occur if all instructions associated with the pattern have a condition attached. To avoid this problem, the CGSS searches for patterns with this property and then adds a new "abstract instruction" to the set of attached instructions which can be generated unconditionally. Very often this abstract instruction can be implemented as a sequence of real instructions. This sequence is called the "default list" for the pattern. If no default list is found by the CGSS, an appropriate message is given, and the compiler writer has to check whether he forgot something or whether the combination of all conditions covers all cases occurring in a legal input.

The quality of the generated code can depend on the quality of the default lists. As shown in the next chapter, the CGSS may not find the best possible solution, and it is reasonable to check the generated defaults after the machine description is complete in all other aspects. Default instructions can be defined explicitly by adding an extra rule to the TMD.

3 Using the CGSS for a Pascal Compiler

For judging the performance of the CGSS and the quality of the generated code, we used our system, amongst other projects, for implementing Pascal compilers for a Siemens 7.000 processor (same instruction set as the IBM 360/370 series), and the Motorola M68000 processor [Armingeon 81]. A Pascal front end provided by W.M.Waite was used in these compilers. Chapter 3.1 contains information about the TMD's and the performance of the CGSS, while chapter 3.2 is concerned with the quality of the generated code and the performance of the code generators.

3.1 Generating the Code Generator

To generate a code generator for some target machine, a description of this machine must be written and processed by the CGSS. This section is concerned with the target machine descriptions that we used for our Pascal compiler and with the processing of these descriptions with the CGSS. Table 3-1 gives some figures about the TMD's and the performance of the CGSS. The TMD for the M68000 does not include real arithmetic, whereas the Siemens description does.

As mentioned in section 2.2, we use Pascal statements in the rules describing the machine instructions. This fact substantially contributes to the size of the TMD's. Usually our descriptions for real machines are about 2000 lines long. The Pascal statements from the TMD, supplemented with the access pathes to the attributes, are incorporated into the code generator frame.

The generated tables comprise the parser table and tables to control the evaluation of the conditions and actions associated with the pattern selected by the parser. The parser tables are constructed and optimized using modules from our parser generating system PGS [Dencker 77].

This is not the place to discuss all aspects of the two machine descriptions. But two points are especially important and should be mentioned. One of them is the handling of the different address modes available on real machines, the other is the use of special instructions by the automatically generated code generator and the necessity of default lists (see section 2.3) as a consequence of this.

	S7000	M68000	
Size of TMD	2500	2100	lines
Number of rules	150	130	
Average size of rules	10	10	lines
Number of operators	87	64	
Number of parser states	149	154	
Size of generated tables			
not optimized	30	36	KByte
optimized	5	7	KByte
Size of generated code			
to be included in the	2000	1100	lines
code generator frame			Pascal
Total processing time of			
the CGSS on a S7.760	148	132	sec.

Table 3-1

An RX-address on the Siemens 7.000, consisting of two registers and a constant offset, can be represented by ten different patterns using the operators +, r and c for address-addition, register and constant resp. : c, r, +cr, +rc, +c+rr, +r+cr, +r+rc, ++crr, ++rcr and ++rrc. It is, in practice, impossible to describe all these addressing modes together with the instruction. For an operation using an RX-address (almost all register-storage operations on the S7.000), the machine description would contain not one, but ten patterns. The complete description of a MOV instruction on the Motorola would require more than fifty different patterns. On the other hand, it is necessary to use the possibilities offered by the hardware to produce good code. Fortunately, the same addressing modes are used for most instructions, and we decided to use a special nonterminal "address" from which all the possible address expressions can be derived, and the address computation is decribed by rules like

```
RULE address.a ::= address_plus
                   constant.c register.r;
  COST      0;
  CONDITION (c.value  IN [0 .. 4095]) AND
            (r.number IN [1..15]     );
  ACTION a.offset := c.value;
         a.base   := r.number;
         a.index  := 0;
END_RULE;
```

It is characteristic for such rules that they do not correspond to an instruction and do not cause any costs since no code is emitted. In the description of an instruction using an address, only this nonterminal appears in the pattern, as in

```
register ::= int_plus register
                        int_cont address.
```

As the cost of an instruction can only be given by a constant, this solution allows the exact description of an instruction only if the cost of the instruction does not depend on the specific addressing mode chosen. For machines such as the M68000, where e.g. an instruction occupies an extra word if the address mode contains an offset, the addressing modes can be grouped into classes with equal cost, and a different nonterminal can be used for each class.

As an example for the use of special instructions and default lists let us consider the ADD instruction on the M68000 that adds the contents of a register to a memory cell. The description of this instruction is

```
RULE STMT ::= int_assign address.a1
                  int_addition data_reg.dr
                      int_cont address.a2;
  COST 2;
  CONDITION is_same_address (a1, a2);
  ACTION    (* emit ADD dr,a1 *)
END_RULE;
```

This instruction can be used for Pascal assignments of the form "i := i + expr" where the value of expr is in the data register dr. The syntactic pattern given above does not express that the two addresses a1 and a2 must denote the same memory cell. As a consequence, this pattern will also be found for a statement like "i := k + expr", and the condition will then evaluate to false, so that this instruction cannot be used. There is no other instruction with the same pattern, so the code generator could not generate any code in this case unless it can split the pattern into smaller parts for which it can generate code. This is exactly what the default list construction is doing. The possible patterns for our example and the corresponding instructions are

```
P1: data_reg.dr  ::= int_cont address.a
                                "MOV a,dr"
P2: data_reg.dr0 ::= int_addition
                data_reg.dr1 data_reg.dr2
                          "ADD dr2, dr0"
P3: data_reg.dr0 ::= int_addition
                data_reg.dr1
                int_cont address.a
                               "ADD a,dr0"
P4: STMT ::= int_assign address.a
                        data_reg.dr
                               "MOV dr, a"
```

Obviously, we can split the pattern

```
STMT ::= int_assign address.a1
             int_addition data_reg.dr
                int_cont address.a2
```

in two different ways : using the patterns P3 and P4 (resulting in the code sequence ADD a2,dr; MOV dr,a1) or P1, P2 and P4 (resulting in the code MOV a2,dr1; ADD dr1,dr; MOV dr,a1). Both possible default lists are correct, but the first one is better because it results in smaller and faster code.

The CGSS automatically generates default lists at code generator generation

time if they are necessary. If a default list can be found, the CGSS will find one. If, however, more than one default list is possible, the CGSS will select one at random even if another one might be better. The problem to determine a "good" default list at code generator generation time has not yet been satisfactorily solved. As soon as the code generator is complete in that it can generate code for every legal input, it is, therefore, worthwhile to have a look at the generated default lists. If one is too bad, it is always possible to insert an extra rule in the TMD with the pattern in question and provide the default list explicitly in the body. This rule will have no condition and a high cost so it will only be used if no other rule with the same pattern is applicable.

3.2 The Code Generator and the Generated Code

The generated code generator for the Siemens 7.000 is about 3800 lines of Pascal (1800 lines for the frame and 2000 lines generated by the CGSS). The equivalent program for the M68000 has 3500 lines (2400 for the frame and 1100 generated lines). These programs perform the code selection and register allocation phase for our compiler, type mapping and variable allocation are done by a separate module.

Because our purpose was to test the power of the code selection algorithm, all other parts of the compiler have been implemented straightforward with as little effort as possible. This is also true for register allocation, which is done "on the fly". Calls to the register allocator are imbedded in the actions of the rules in the TMD's. The register allocation algorithm assumes that a sufficient number of registers exist, means for register spilling are not provided. If the register allocator runs out of registers, the compilation is aborted after issuing an appropriate error message (this case did not occur except for specially written test programs). Because no context is considered when allocating a register, it can happen that a value is loaded into the wrong kind of register (e.g. into an odd instead of an even one) and must then be moved into another register.

The generated code generator processes about 200 source lines per second, including reading the intermediate representation from a file and writing the generated instructions. For our front end (see table 3-2), the code generator needed 33 seconds. Experiments where the io has been separated from the code selection indicate that the performance of the code generator can at least be tripled.

The code generated for the S7.760 was compared with the code generated by the standard compiler on this system, a product developed on the basis of the Pascal-P compiler. Table 3-2 gives the code size for the modules of our compiler when compiled with our and with the standard compiler. Overall, the automatically generated code is about 10 % shorter than that of the standard compiler. A more detailed analysis shows that the automatically generated code made better use of special instructions and addressing modes, whereas the register usage was better in the standard compiler. Especially, the standard compiler eliminates simple common subexpressions inside basic blocks and uses more register-register operations.

We have also compared the run time of the code generated by the two compilers. Table 3-3 shows the CPU-time used on a

S7.760 processor by some sort programs for sorting an array of 1000 real numbers with several methods : heap-sort (HEAP), shell-sort (SHELL), a hybrid bubble sort (DOBOS) and quick-sort (QUICK). As can be seen, the automatically generated code is approximately 15 % slower, a figure that was confirmed by other experiences. The main reason for the advantage of the standard compiler is the better register usage and the fact that we did not put a great effort into the run-time system for our generated Pascal system.

The code generated for the Motorola M68000 was compared with code generated for this processor by compilers from Hewlett-Packard and from Motorola. Table 3-4 shows the code size for the sort program already mentioned as generated by our compiler and by the Motorola compiler. A comparison with the HP compiler for the same programs was not possible. A first analysis shows that the automatically generated code is shorter than the one generated by the other two compilers. The main reason for this seems to be the better usage of storage-storage operations for simple statements.

4 Conclusions

The CGSS is now in use for more than a year and our overall experience is positive. With an effort of six man month we have been able to produce a code generator for Pascal whose quality is comparable with an industrial product into which, to our knowledge, at least six man years have been invested. In addition, it was possible to retarget this code generator from the Siemens 7.000 to the M68000 in approximately three man month.

module	Pascal source lines	size of code compiler our	stand.	change
front end	6940	78240	83636	+7 %
type mapping, IR generator	6060	48682	61260	+25 %
handling of sets	2400	21104	23644	+12 %
constant folding	1500	11234	10738	-4 %
code generator	3770	57272	56652	-1 %
code editing	2430	26314	26664	+1 %
listing generator	1400	12136	11804	-2 %
sum	24500	254982	274398	+8 %

Table 3-2 : code sizes of compiler modules on the S7.000 (in byte)

program	our compiler size	time	standard comp. size	time
SHELL	2424	219	2484	184
HEAP	2762	244	2736	206
DOBOS	2596	264	2628	225
QUICK	3360	124	3200	111

Table 3-3 : comparison of sort programs on a S7.760 processor (code size in byte, time in milliseconds)

program	code size [byte] our comp.	Motorola	change
SHELL	1976	2048	+ 4 %
HEAP	2250	2394	+ 6 %
DOBOS	2140	2256	+ 5 %
QUICK	2746	3036	+10 %

Table 3-3 : comparism of sort programs on a M68000 processor

We are just starting to study how our code selection algorithm fits into the framework of an optimizing compiler. A lot of optimizations interfere with code selection, especially the decision whether a possible optimization is profitable. Common subexpression elimination at the addressing level, for example, may deteriorate the quality of the generated code if an address expression is loaded into a register while it could have been evaluated within the addressing part of an instruction. We believe that codegenerators generated by the CGSS can be used to support such optimizations, e.g. by computing machine dependent attributes for controlling optimization modules or by actually performing optimizations which have been found possible by previous, machine independent phases of the compiler.

The most important improvement on the way to an automatically generated "production quality" code generator is the integration of a more sophisticated register allocation algorithm into our system. This will reduce the size of the generated code and above all shorten the execution time.

One problem already mentioned is the formalization of the conditions and actions in the TMD, which would considerably reduce the size of machine descriptions. We are not working in this direction in the moment because the present form allows more experiments with the code generator.

The code selection algorithm proposed by Glanville and implemented by the CGSS uses a special kind of pattern matching in trees, the parsing with an LR parser, for the purpose of code selection. Our results prove that it is possible to produce good code with this method and that it is efficient enough to be used in compilers. Nevertheless, other realizations of the pattern matching are possible, and more experiments in this field would be interesting.

In the present form, the CGSS allows to build, with a minimum of effort, code generators that are reliable, need only reasonable resources at run time and produce good code that can stand a comparison with a conventional code generator.

References

[Armingeon 81] W. Armingeon :
Ein Pascal-Übersetzer für den MC 68000
Diplomarbeit, Universität Karlsruhe,
Fakultät für Informatik 1981

[Cattell 78] R.G.G. Cattell:
Formalization and Automatic Derivation of Code Generators
PhD Thesis, Carnegie-Mellon Univ. 1978

[Dencker 77] P. Dencker :
Ein neues LALR(1)-System
Diplomarbeit, Universität Karlsruhe,
Fakultät für Informatik 1977

[Ganapathi 80] M. Ganapathi:
Retargetable Code Generation and Optimization using Attribute Grammars
PhD Thesis,
University of Wisconsion-Madison 1980

[Ganapathi/Fischer 81]
M. Ganapathi, C. Fischer:
Bibliography on Automated Retargetable Code Generation
Sigplan Notices, Vol. 10, No. 10,
October 1981

[Glanville 78] R.S. Glanville:
A Machine Independent Algorithm for Code Generation and its Use in Retargetable Compilers
PhD Thesis, University of Carlifornia, Berkeley 1978

[Jansohn/Landwehr 80]
H.-St. Jansohn, R. Landwehr:
CGSS: Ein System zur automatischen
Erzeugung von Codegeneratoren
Diplomarbeit, Universität Karlsruhe,
Fakultät für Informatik 1980

[Kozlak 81] R.H. Kozlak:
Machine Independent Code Generation
Technical Report CSRG-125,
University of Toronto 1981