



NORTH-HOLLAND

Overview and Industrial Application of Code Generator Generators

Niclas Andersson and Peter Fritzson

Department of Computer and Information Science, Linköping University, S-581 83 Linköping, Sweden

During the past 10 to 15 years, there has been active research in the area of automatically generating the code generator part of compilers from formal specifications. However, little has been reported on the application of these systems in an industrial setting. This paper attempts to fill this gap, in addition to providing a tutorial overview of the most well-known methods. Four systems for automatic generation of code generators are described in this paper. CGSS, BEG, TWIG and BURG. CGSS is an older Graham-Glanville style system based on pattern matching through parsing, whereas BEG, TWIG, and BURG are more recent systems based on tree pattern matching combined with dynamic programming. An industrial-strength code generator previously implemented for a special-purpose language using the CGSS system is described and compared in some detail to our new implementation based on the BEG system. Several problems of integrating local and global register allocations within automatically generated code generators are described, and some solutions are proposed. In addition, the specification of a full code generator for SUN SPARC with register windows using the BEG system is described. We finally conclude that current technology of automatically generating code generators is viable in an industrial setting. However, further research needs to be done on the problem of properly integrating register allocation and instruction scheduling with instruction selection, when both are generated from declarative specifications.

1. OVERVIEW

First we would like to give some advice how to read this paper. The reader who is already well aware of the developments in code generator generator technology but primarily is interested in applications, should skip the first three sections and start reading in Section 4. On the other hand, the reader who does not know the subject area very well is recommended to read the tutorial introduction in Sections 2 and 3. Finally, the expert reader who would like to study the specifications in detail, should take a look at the appendices.

As already hinted, the first part—Sections 2 and 3—contains a tutorial overview of the code generation problem in general, as well as a comparison of four code generator generators, namely CGSS, TWIG, BEG and BURG. The second part, Sections 4–8 and Section 9, describes two applications of code generator technology: an implementation of a code generator for a special purpose processor in an industrial application environment and an implementation of a code generator for the well-known SPARC RISC architecture.

Section 2 provides a general tutorial introduction to code generation and how a code generator can be designed. Section 3 describes how it can be automatically generated. Here we also make an overall comparison and some evaluation of the four tools CGSS, TWIG, BEG, and BURG.

Sections 4 and 5 briefly describe an application environment: the AXE phone exchange system, the special purpose processor APZ for which code is generated, and the special purpose language PLEX, which is compiled into target code for the APZ processor.

Sections 6, 7, and 8 describe our new implementation of a code generator for a subset of PLEX, using the BEG system. We compare the register allocation approach and the code quality with the existing production-quality code generator produced by the CGSS system and discuss various strategies of porting.

Section 9 describes an application of code generator technology to the implementation of a code generator for the SPARC RISC architecture and how register windows were handled.

In Section 10, we finally present conclusions.

2. CODE GENERATION

During the last phase of compilation, final code generation, also called instruction selection, is performed, i.e., some intermediate form of the program

is converted to a semantically equivalent instruction sequence for some target machine.

High performance is usually required by such code generators. They should produce correct code of production quality that will utilize the resources of the target machine efficiently. Of course it is also desirable that the code generator itself is fast.

From a mathematical point of view, optimal code generation, or more precisely: optimal selection of target instructions is an undecidable problem. Since input data will influence the way to generated target code will be executed, no polynomial-time solutions are known for the task of selecting at compile time a code sequence that is optimal under all circumstances. In practice, we will have to resort to heuristic algorithms that do not always generate optimal code.

In the following, we primarily deal with the instruction selection part of final code generation. Thus, the meaning of the term code generation usually is instruction selection possibly in combination with register allocation.

2.1 Input Data

The form of input data to a code generator is often defined by the modules of the compiler which represent earlier phases in the compilation process. Usually the intermediate form is some kind of tree or directed acyclic graph (DAG). Other common forms are triples, quadruples, or reverse polish notation (RPN). An intermediate form tree consists of nodes which represent binary or unary operators, together with leaf nodes representing identifiers, constants, etc. and is more primitive than an abstract syntax tree. In order to simplify the work of the instruction selector and produce better code, optimizations are usually performed both before and after instruction selection. It is often useful to represent the program as acyclic graphs or trees in order to perform program flow analysis and estimate the register usage of the program. From now on, we are using trees as intermediate form.

2.2 Code for a target machine

There are several forms of output data from a code generator. Common forms are binary machine code, relocatable machine code, or assembly text. Which form is selected may depend on how the code will be linked or loaded. It is usually easier to use some assembly format because instructions can then be represented in symbolic form. The disadvantage is that an assembly phase has to be added, which increases total compilation time.

2.3 Instruction selection

How should an instruction selector be implemented? An easy way is to perform a postorder traversal of the intermediate code represented as a tree, and emit naive code using a standard template for each type of node that is visited. In this way, it is easy to generate correct code, but the code is definitely not optimal (see Figure 1). It is necessary to take the architecture of the target machine and its instruction set more into account, in order to generate good code.

2.3.1 Machine idioms. An instruction selector should ideally be able to make use of all instructions and addressing modes of the target machine. The instructions of the target machine have different execution and memory usage characteristics. It is important to choose the cheapest instructions, i.e., instructions that have the shortest execution time and which can be stored compactly in memory. For example, regard the *INC* (increment) instruction which is available on many machines. It is both faster and more compact than the more general *ADD* instruction. The instruction *INC reg* can replace the *ADD reg, #const* instruction if *#const = 1* (see Figure 2).

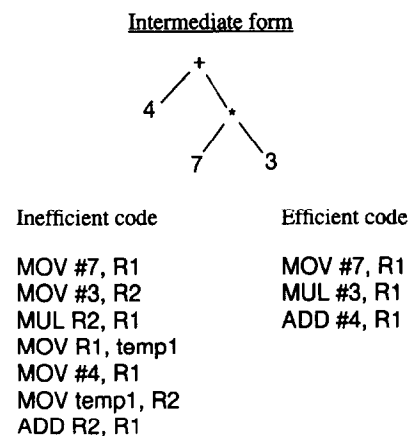


Figure 1. Examples of efficient and inefficient code.

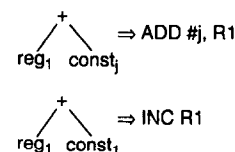


Figure 2. *ADD* and *INC* have the same representation in the IL, subject to the condition that the constant of *INC* should be 1.

2.3.2 Register allocation. Machine instructions whose operands are stored in registers execute faster than instructions which use operands stored in memory. Ideally, one would then like to keep most potential operands in registers. Unfortunately, this is usually not possible since most machines only have a small number of registers. Thus, it is important to use these registers effectively. In order to simplify the problem of register allocation, it is usually divided into two parts:

1. Global register allocation, where commonly used variables are assigned to registers. This kind of register allocation is usually performed early during the code generation phase and will not be dealt with further in this paper.
2. Local register allocation, where registers are allocated for temporary results during the computation of expressions. Since some of these temporaries are created during the instruction selection process, it may be useful to integrate local register allocation with instruction selection.

Finding the optimal register allocation is an NP-complete problem, i.e., it cannot be solved in polynomial time. Here, as in instruction selection, one has to resort to methods that produce good, but not necessarily optimal code. The problem becomes even more complicated when peculiarities of the target machine architecture have to be taken into account, e.g., the existence of special registers such as index registers or floating point registers. It is also common that machines can regard two standard registers as one large register in order to handle large operands. Other instructions may require that operands reside in special registers. For example, the integer multiplication instruction on the IBM370 requires that one of the operands reside in an even-odd register pair.

2.3.3 Order of evaluation. The efficiency of the generated code is influenced by the order in which computations are done. Certain orderings require a smaller number of registers than others. This problem is also NP-complete in its general form. One approach to selecting an acceptable ordering is to make an extra traversal of the intermediate form tree being instruction selection in order to compute the register needs of each node. This information is then used to select the ordering.

2.4 A code generation algorithm

A common approach to the generation of high quality code is to assign all results to registers, i.e., let

both intermediate results and final results be in registers. Such results are stored into memory only when there is a lack of available free registers or when there is some other reason for storing into memory, e.g., before a procedure call or return from a procedure. One code generation algorithm which produces an acceptable computational ordering and uses registers efficiently is based on dynamic programming (Aho et al., 1986). This algorithm recursively splits the code generation problem into smaller subproblems, where the register need of each subproblem controls the order in which code generation occurs. This algorithm is optimal if all target machine instructions can use all registers, i.e., there are only general registers.

Unfortunately, it is seldom the case that all registers of target machines are general registers. Instead, as mentioned in section 2.3.2, special registers must be used for certain addressing modes and for certain instructions. This complicates the use of the dynamic programming algorithm since intermediate results either must be available in certain pre-allocated registers or have to be moved in the middle of a computation.

Some other disadvantages of this algorithm:

1. Large and complex machine instructions whose semantics are equivalent to a combination of several nodes in the intermediate form cannot easily be generated using this algorithm, since it generates code one node at a time, e.g., indexed instructions which have address calculation's and are expressed explicitly in the IL-tree.
2. Intermediate results which could have been computed automatically by instructions with advanced addressing modes may instead be computed explicitly by several more primitive instructions, which is less efficient.
3. Many MOV-instructions will be generated at the end of sub-computations, to store results into memory. This can be improved by using data-flow analysis to determine which variables are still "live" at the end of a block.

The complete code generator will consist of a number of special procedures, one for each instruction. This is because the register usage, the need for special registers, and the cost depending on where operands reside, are different for almost all instructions. Additional procedures are needed to cope with those cases for which the algorithm does not succeed (it blocks). Then the values of one or more registers have to be dumped into memory (spilling) in order to make more registers available, whereafter the instruction selector tries again.

It is hard to maintain such code generation software since it is usually not easy to understand an algorithm which consists of many small special cases. The solution to this problem could be to describe the capabilities of the target machine using some declarative notation, i.e., to use some formal language to express and describe the instrument set of the target machine.

2.5 Pattern matching

One declarative way of describing the target machine instruction set is through small intermediate-code patterns. In this way, the instruction selection process can be regarded as a transformation or reduction of intermediate language (IL) trees. These trees are essentially intermediate quadruples or triples in tree form and are more primitive than abstract syntax trees. An IL tree consists of nodes which represent binary or unary operators, together with leaf nodes representing identifiers, constants, etc. Each target machine instruction can be described by an IL-pattern where the operands are represented by terminals or nonterminals in production rules of a transformation grammar for trees (see Figure 3). Associated to each pattern is a result which is also represented by a non-terminal symbol.

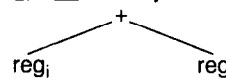
Rule 1: MOV #c, Ri

const_c ⇒ reg_i Cost: 2

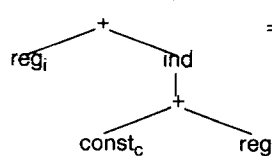
Rule 2: MOV a, Ri

mema ⇒ reg_i Cost: 2

Rule 3: ADD Rj, Ri


 ⇒ reg_i Cost: 1

Rule 4: ADD c(Rj), Ri


 ⇒ reg_i Cost: 2

Rule 5: MOV (Ri), b

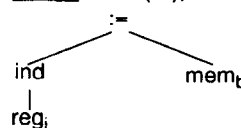

 ⇒ reg_i Cost: 2

Figure 3. Rules to transform the tree in Figure 4.

The pattern-matcher can use these non-terminals to tie different rules together. The non-terminal representing the result must be the same as the non-terminal operand symbol of patterns in which this result is used as an operand.

What kind of algorithm should be used for this pattern-matching process? There are currently three main alternatives:

1. *Parsing*.
2. *Attribute grammar techniques*. This is not covered in this paper.
3. *Pattern-matching with dynamic programming*. Special pattern-matching algorithms for trees have been developed.

2.5.1 Parsing. The intermediate language (IL) patterns can be interpreted as grammar rules, which can be used as a basis for constructing an LALR(1)-parser (or some other kind of parser). Input data to the parser is obtained by converting the intermediate-tree to a string of tokens through a preorder traversal of the tree. Code for the target machine is generated when the parser is able to perform a reduction step according to one of the grammar rules.

However, applying parsing in this way is not powerful enough for code generation since the grammar specified by the IL-patterns is ambiguous and there are many ways of reducing the input. One approach to solving this problem is through the use of additional heuristic¹ rules (Aho et al., 1986):

- *shift/reduce-conflicts* are resolved by choosing shift.
- *reduce/reduce-conflicts* between rules with different right-hand parts are resolved by choosing the rule with the longer right-hand part. The rationale is that the more complicated right-hand part usually corresponds to more powerful target-machine instructions, which would result in faster and more compact target code.
- *reduce/reduce-conflicts* between rules with equal size of the right-hand part are resolved by comparing the cost which is associated with each rule. The rule with the cheapest cost is selected. This cost is usually related to the speed and size of the code associated with this rule pattern. Smaller and faster code gives lower cost.

According to these rules it is possible to design a nonambiguous grammar. However, the result is not

¹There are several ways of mapping an IL-tree to target code.

always optimal. Overlapping rules can cause the parser² to not find the cheapest alternative. If chain rules² are used in the grammar, it is also necessary to consistently check the grammar and manually rewrite the grammar to prevent it from looping. A critical part of parsing based instruction selectors is to avoid chain loops brought about by incorrectly choosing in a reduce/reduce conflict.

2.5.2 Pattern-matching on trees. Pattern-matching on trees is a simple and attractive algorithm. For example, by using one of pattern-matching algorithms described in Aho et al. (1989) and Hoffman and O'Donnell (1982) it is only necessary to make a postorder traversal of the tree once in order to obtain the best possible match. It is also possible to prove that these algorithms will find the cheapest match if one exists. The difference in time complexity between parsing and this type of general pattern-matching is rather small. However, more space is required since the whole tree must be kept in memory while the algorithm operates.

An application of the rules described in Figure 3 on the tree in Figure 4 will generate the following code:

MOV #a, R0	(Rule 1)
ADD SP, R0	(Rule 3)
ADD i(SP), R0	(Rule 4)
MOV (R0), b	(Rule 5)

In order to fully utilize pattern-matching, all addressing modes must be explicitly expressed in the intermediate language. From a theoretical point of view, instruction selection using pattern matching will work best if the intermediate language is more primitive than the target machine language. Then a nonambiguous relation between the IL and the generated code can be obtained (see Figure 5, N:1). In practice, such an IL will be too large which will cause bad performance of the code generator. Therefore, a somewhat higher level of abstraction is usually chosen for the IL.

2.6 Automatically generated code generators

One possibility to simplify maintenance and updating of the code generators is to generate it automatically from some specification. A necessary condition for automatic generation is to express the specification of the code generator in a formal machine-

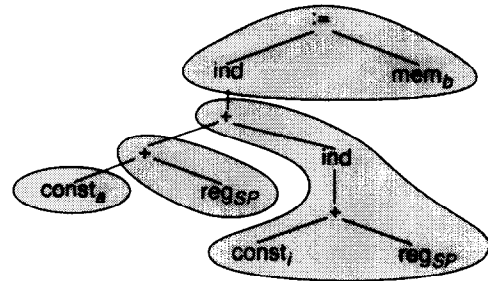


Figure 4. An IR-tree for $a[i] := b$. a is here a local variable and b is a global one.

processable notation. This can be done, for example, if one of the pattern matching algorithms is used. Given that the IL-patterns are specified in a declarative way, then the code generator generator can provide a suitable matching algorithm. Research done on this topic has already resulted in several such tools, e.g., CGSS (Jansohn and Landwehr, 1980; Landwehr et al., 1982), BEG (Emmelman et al., 1989), TWIG (Tjiang, 1986; Aho et al., 1989) and BURG (Fraser et al., 1992).

3. CODE GENERATOR GENERATORS

The input data to a code generator generator (really a code or instruction selector generator), is usually a text file which contains a declarative description of the target machine instruction set. The IL-patterns in the description correspond to target machine instructions. The generator also needs a description of the form of input data to the code generator, i.e., the form of the intermediate language. As a result of performing such a generation, a code generator is obtained in some language, e.g., Pascal or C. This code generator can subsequently be linked together with other modules of the compiler (see Figure 6).

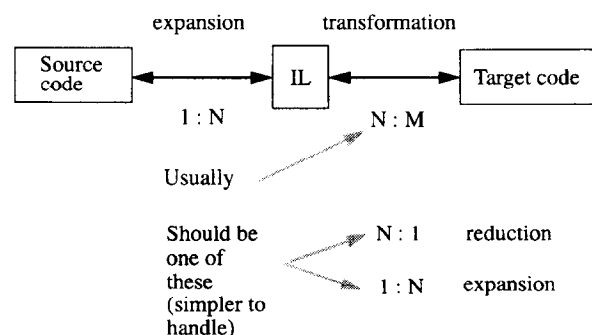


Figure 5. Relation between source code, intermediate language (IL), and target code.

²A chain has a right-hand side consisting of only a non-terminal symbol.

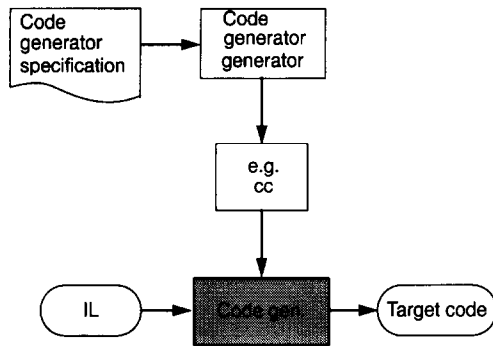


Figure 6. Flow of data for a code generator and its generator.

The following four code generator generators have been studied in this paper

- *CGSS* (Landwehr et al., 1982), a Graham-Glanville (Graham and Granville, 1978) style code generator generator based on pattern matching through LR-parsing. The obtained code generator is in Pascal.
- *BEG* (Emmelman, 1989) which is based on pattern-matching on trees (also using dynamic programming). The produced code generator is in C. This generator can also generate a local register allocator (i.e., no global allocation based on flow analysis) which is integrated with the instruction selector part of the code generator.
- *TWIG* (Aho, et al., 1989) which is a more general generator of tree pattern matchers and tree rewriting programs, also producing programs in C-code. This generator can also handle rules which specify rewriting of trees, not just pattern-matching. However, it does not include any register allocation support. Since we had no running implementation of *TWIG* available, its functionality has been studied based on available literature.
- *BURG* (Fraser et al., 1992) a generator which uses the new BURS-technique (Bottom-up rewriting system), which also is a variant of tree pattern matching combined with dynamic programming. This system has a more primitive user interface and a less flexible specification language but produces faster code generators since the dynamic programming optimizations are done once at compiler-generation time.

3.1 CGSS (Code Generating Source System)

CGSS was developed by H. S. Jahnsohn and Rudolf Landwehr at the University of Karlsruhe during the beginning of the 1980s. Unfortunately, no user's manual was available during this investigation.

CGSS uses parsing as a method for pattern-matching which causes certain problems at generation time:

- The grammar becomes ambiguous. Heuristic methods must be used to resolve related problems, see 2.5.1.
- Chain rules may cause the parser to loop. In a code generator generator, the grammar must be rewritten automatically if there is a potential risk of looping.
- Because of the conditions on rules that determine when these rules may be used, situations may occur where the parser has no allowable state to transfer to. An unconditional last alternative must then be generated, in order to prevent the parser from blocking.
- The parsing algorithm scans tokens from left to right. When it has succeeded in matching a subtree, this subtree is deleted and replaced with the non-terminal specified on at the left of the matching rule. If there is some other rule that "overlaps" the matching rule, the CGSS parsing algorithm cannot, under any circumstance, match this other rule (see Figure 7)

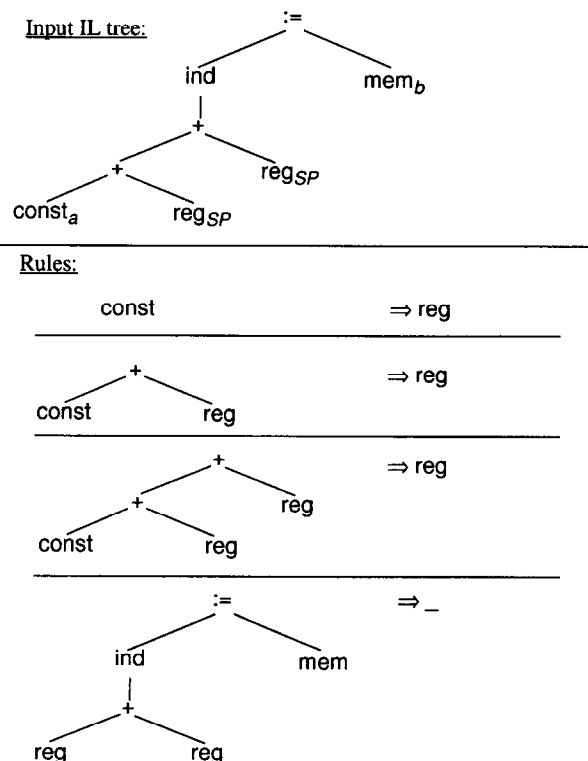


Figure 7. Example of when parsing is not sufficient. The fourth rule will never be used regardless of the cost since the parser always will reduce the third rule first.

The first three problems are solved automatically by CGSS (Landwehr et al., 1982). However, there is one caveat—it cannot handle the last problem. One must have this in mind when writing a specification for a code generator. If one has this “overlap” problem in mind, it is possible to write specifications that can be used to produce quite efficient code generators, e.g., the code generator for the PLEX compiler (PLEX). The reasons why certain rules have been chosen should be clearly documented in order not to cause future problems when the code generator specification is updated.

Example Update. A previously well-functioning specification consists of rules 1, 2, and 4 in Figure 7. This specification is extended by adding rule 3 (using a new machine instruction). This extension causes the code generator to fail matching input trees which it earlier succeeded in matching, e.g., the Input IL tree at the top of Figure 7.

3.1.1 Input data to CGSS. The structure of the specification language is simple. The rules have the following structure:

```
RULE<lhs>::= <rhs>; (*prefix expression*)
BY-DEFAULT;|COST {PASCAL-integer};
[CONDITION {PASCAL-expression};]
ACTION {PASCAL-statements};
```

The integer specified after the COST keyword should correspond to the execution or memory cost of the code that is generated when this rule is matched. The expression after CONDITION is a constraint which, if not satisfied, will prevent the rule from matching.

Example Increment. Here we assume that register allocation already has been done.

```
RULE register.a::= plus register.b const;
COST2;
CONDITION a.reg = b.reg AND const.val = 1;
ACTION emit_inc(a.reg);
```

If the condition after keyword CONDITION is not satisfied, the rule will not be used.

3.2 BEG (Back End Generator)

BEG, like CGSS, originates from GMD at the University of Karlsruhe. Its development was initiated based on experience from the earlier CGSS system, and a desire to eliminate some of its shortcomings. It was designed and implemented by Helmut Emmelmann (Emmelmann, 1990; Emmelmann et al., 1989). The version of BEG initially used by us was the first beta-version producing C-code. A few

small errors were discovered by us and reported back to Helmut Emmelmann, who quickly corrected the problems.

BEG can be regarded as a further development of CGSS in several aspects:

- Parsing as a method for pattern-matching has been replaced by the more general dynamic programming algorithm for tree pattern matching (Aho et al., 1986).
- The specification language has improved. See the BEG user manual and some examples in this paper.
- The target language in which the generated code generator is implemented can now be either C or Modula-2 (user-selectable).
- A local register-allocator can be automatically generated from the part of the specification which describes the register set and register usage. The generated register allocator is integrated with the instruction selector.

The usage of tree pattern matching algorithms has the nice consequence that the user who writes code generator specifications (input to the code generator generator) does not have to worry about the actual pattern matching mechanism of the generator, which was needed in the case of pattern matching through parsing. Instead, the designer can concentrate on expressing the semantics of the target machine instructions correctly when writing the specification. If there exists a cheapest cover for some IL-subtree using the pattern rules, the BEG-generator code generator will find it (Emmelmann et al., 1989). It is possible to prove this.

There is, however, still one caveat regarding the optimality of BEG. If you introduce constraints on the rules in an uncontrolled fashion and at the same time use the EVAL feature (which the other systems do not have), you may lose the optimality property. Another point worth noting is that the optimality property is only regarding the cheapest selection of instructions given some allocation of registers. Since the local register allocator generated by BEG is far from optimal, the total resulting code will not be optimal.

3.2.1 Register allocation. BEG can generate a local register allocator that is included and integrated in the code generator. There are two variants of register allocators to choose from:

1. On-the-fly allocation. This produces a fast but low quality allocator. Unnecessary spill code may be generated from a code generator with such an allocator.

2. General register allocation. This allocator is slower but results in better code in many cases.

In this evaluation, we have chosen the general register allocation alternative, since this method will best utilize the register set as the limited resource it usually is. There are also more possibilities to control the register allocation through the specification using this method.

The pattern matching algorithm of the BEG systems behaved as expected. However, the register allocator is not equally powerful. Several problems associated with the register allocator makes it less useful in certain cases.

- While it is possible to declare to the pattern matcher that operators such as + and * are commutative, this information is not used by the register allocator. Therefore code is never generated in reverse order, even if that would have been more efficient
- The allocator tries to allocate registers by always traversing the tree postorder and backwards. This simple algorithm is not always enough to generate efficient code.
- If a specific register is needed, and this is already occupied, the allocator will always spill this register to memory. It never tries to copy the register contents to another unused register.
- If a rule contains a large code sequence which needs a register for some intermediate result, one must always choose a specific register for this result.

There are also many useful constructs in the BEG specification language that can be used to describe the structure of the target machine instruction set. For example, it is possible to describe that registers can be combined in pairs to form double registers. It is also possible to prevent registers from being allocated, if you only want local allocation.

As part of a BEG code generation specification, there should always be three procedures which are needed by the register allocator to generate code in special situations:

1. *Spill*, a procedure which stores intermediate results in memory. This storage occurs according to the LIFO rule (Last In First Out).
2. *Restore*, which restores intermediate results which were previously stored in memory.
3. *LR*, which performs a register copy when the operands are not available in the right register.

Note, as mentioned previously, the procedure *LR* is sometimes used to move an intermediate result

which prevents an efficient allocation. However, *Spill* and *Restore* are used more often.

3.2.2 Input data to BEG. The language in BEG is to some extent more developed than CGSS's. The example in section 3.1.1 has the following look in BEG:

```
RULE Plus Register.a Const → Register.b;
CONDITION (const.val == 1);
COST 2;
TARGET a;
EMIT {print("INC%d\n", a.reg);};
```

TARGET and CHANGE are directives which can be used to control the register allocator. Using the TARGET directive, it is possible to specify into which registers the result should be stored relative to the operands, while CHANGE describes those registers which are changed when the rule is applied. It is also possible to control the register allocator through a new enhancements to BEG which make it possible to use it as a local register allocator, i.e., only include a specified set of registers when allocating a new register. Already allocated global registers are regarded as read-only. This functionality is described in the latest update to the BEG user's manual.

In the last part of a code generation specification, after the INSERTS keyword, it is possible to include support procedures for the code generator. These procedures will be included in the generated source code files which comprise the resulting code generator.

3.2.3 The algorithm of the code generator. Even if the user does not need to know about the algorithm which is used by the code generator, it is useful to know about the phases of the code generator in order to be aware of where and when certain attributes are available and when operations on these attributes will be executed.

The code generation goes through three phases:

1. The cover phase

This is where the actual pattern matching occurs. The conditions associated with rules can only use and set those attributes which are specified through COND_ATTRIBUTES in the description of the nonterminals. So far, no register allocation has occurred, so there is no possibility to use names of registers in the conditions. These constraints apply also to EVAL which is executed directly after a rule has been selected by the matching process.

2. *The register allocation phase*

This phase exists only when the general register allocation facility of BEG is used. Here, the intermediate tree will be traversed postorder and backwards. Registers will be allocated whenever they are needed. Because of this predetermined traversal order, the register allocator cannot cope with all requirements that one would like it to handle.

3. *The generation phase*

In this phase, the intermediate tree is traversed postorder and the EMIT-part of each triggered rule is executed. During this phase, all attributes are available.

Note that the intermediate tree is traversed postorder in both the first and the last phase. Thus, there are two possibilities to synthesize attributes, while there are no provisions for inheriting attributes.

3.3 TWIG

TWIG (by S. W. K. Tjiang) is a tool which is not restricted to just generating code generators. It is more general than that. In order to describe what this tool can do, it is more appropriate to say that it generates programs which can transform and reduce trees according to rules which have been given. An evaluation of TWIG has been included in this report not only because it can generate a code generator, but also because it provides certain functionality and certain constructs which are missing from the other two generators.

A TWIG-generated code generator contains the same pattern matching algorithm as a BEG-generated one. Thus, it functions the same way. Since it lacks a register allocator, only the cover phase and the generation phase will be executed.

TWIG provides no interface for intermediate form trees. It is assumed that such an interface already exists. Instead, the user must specify types and functions in order to describe the structure of the existing tree nodes so that the pattern matcher is able to navigate in the tree and access the operators of the tree (Tjiang, 1986). The cost of the rules is computed by TWIG using functions and not only constant integers as in BEG and CGSS.

The main difference between TWIG and the other two systems, is that the pattern matcher can be controlled. Within the cost function, three types of derivatives can be added. These derivatives act as a kind of control commands to the pattern matcher.

• ABORT

The pattern in this rule will not be used. The counterpart of this directive in BEG and CGSS is that the predicate specified after **CONDITION** evaluates to False.

• TOPDOWN

Normally a postorder bottom-up traversal of the tree is performed when the bodies of the rules, (the action parts), will be executed, which is in the generation phase. This directive instructs the pattern matcher to only execute the body of this rule, and not automatically its children.³ You will have to specify explicitly that the rules which are matched with child nodes will be executed. This facility is present to be able to select in which order the child nodes will be visited.

• REWRITE

Here, similarly as the **TOPDOWN** directive, the pattern matcher will only execute the body of this rule automatically. When the current rule has been matched, the subtree which is rooted in the current node will be traversed once more, which gives the pattern matcher one more chance to find the cheapest cover. By using **REWRITE**, it is possible to achieve recursive rewriting of the tree. When the body of a rule is executed, a new tree can be returned which replaces the tree whose root corresponds to the root of the current rule. One should be careful when using rules that include **REWRITE**. Such rules should be blocked (using **ABORT**) when they are not supposed to be invoked, in order to avoid recursively expanding trees by mistake.

3.3.1 Input specification to TWIG. At the beginning of a specification, the matcher needs to know about the structure of the input tree, i.e., how the tree is defined. There must also be type declarations in the specification to enable the pattern matcher to declare local variables and allocate memory. The matcher also requires the availability of procedures for tree traversal and access functions for node operators.

The next part contains declarations of the rules. The first section of this part consists of declarations of terminals, i.e., nodes, and non-terminals. Then come declarations of the rules, which have a very simple structure:

```
label_id : tree_pattern [cost] [= action]
```

³A child can of course also be the root in a subtree.

where *cost* and *action* are expressed as C-code. The action part corresponds to the EMIT part in a BEG specification and will be executed during the last phase.

The example from Section 3.1.1 in TWIG (the registers have already been allocated):

```
Register: Plus(Register, Const)
{($%2$.val == 1) ? 2 : ABORT;}
=
{printf("INC %d\n", $$reg);}
```

3.4 BURG

BURG is a tool which generates fast tree pattern-matchers using Bottom-Up Rewrite System techniques (Pelegrí-Llopert and Graham, 1988; Chase, 1987), in short: the BURS technique. It is somewhat less general than the other tools because the lack of conditional rules. However, it is much faster since it uses static analysis to once and for all perform the dynamic programming optimization for each instruction selection case within a specific combination of intermediate language and target instruction set. Thus, the generated code generator need not perform dynamic programming at code generation time—this was already done when the code generator was produced by BURG.

The BURG approach, however, obtains its speed at the cost of decreased flexibility: no constraints or conditions are allowed in code generator specification rules, no EVAL facility, etc. Instead, all relevant information has to be coded into the operator. This makes it harder to write specifications and makes them less readable. Nonetheless, a code generator for full ANSI C has been developed by Fraser.

To use BURG, it is essential to understand the pattern matching algorithm, since the user interface consists of a few low level procedures which are invoked by the main program to match and transverse the input-tree. The generated pattern matcher has no facilities to incorporate the action part for each rule. Instead, the user has to contribute a switch statement that takes a rule number and performs the correct action.

3.4.1 Input specification to BURG. The input specification is very similar to the input to the TWIG system. The specification file is divided into three parts, similar to Lex and Yacc. The first part is the declaration part where the access functions and operator constants are declared. Part two consists of the rules, rule numbers and costs:

```
non_terminal: tree_pattern = rule_no (cost);
```

In the third part, it is possible to insert C-functions needed by the pattern matcher. The example in Section 3.1.1 is very difficult to implement in BURG. Somehow the value of the constant must be a part of the operator. For instance, there must be different plus operators depending on the value of the constant:

```
register: OnePlus(register) = 17(2); /*const == 1*/
register: Plus(Const, register) = 18(3);
```

3.5 Comparison between CGSS, BEG, TWIG, and BURG

All four tools can be used for generating a code generator. The most important common property is that they all accept as an input text file a declarative specification of code generation rules. Most of this specification consists of source code in the language generated by the code generator generator, e.g., PASCAL for CGSS. All four tools can from such a specification produce a code generator which can be compiled directly.

However, there are also some differences. These may not be too great but are important to consider in order to choose the most appropriate tool.

3.5.1 Algorithm. CGSS uses parsing as the pattern matching algorithm while BEG, TWIG, and BURG use the more general algorithm for pattern matching on trees based on dynamic programming. Parsing is probably more efficient since it does not need any large and complicated data structures—only a simple stack. Note that the real complexity in both of these methods is in representing and unpacking the tables, for which there are numerous schemes specialized for code generation. Even though parsing might be slightly more efficient, the dynamic programming algorithm is better for pattern matching. The advantage of the dynamic programming based tree pattern matchers is that they will *always* find the cheapest cover. According to our experience, they succeed in doing this surprisingly fast. They only need to perform one traversal in order to produce a full covering of the tree. Additionally, the BURG system also analyzes the rules extensively at compile time to remove all the chain rules and reduce the number of matching rules at each node in the IL-tree. The result is that code generators produced by BURG are much faster than those produced by the other dynamic programming based tools which are fast enough compared to other compiler phases.

3.5.2 Interfaces. There are significant differences between the external interfaces to code generators produced by these tools. A CGSS type code generator accepts expressions in prefix form, whereas BEG builds intermediate trees from postfix expressions read from the input file (Figure 9). Alternatively, it is possible to construct the intermediate tree by directly calling a BEG-generated programmatic interface of routines for building the tree nodes of the intermediate tree. TWIG and BURG provide no external interface for reading the intermediate representation from a file. They expect the intermediate tree to already exist, which of course, is an advantage if that is the case. Time is saved by avoiding construction of a new tree. However, if the IL is a linear textual representation then a parser must be implemented which first builds trees that the pattern matcher can use. In BURG, a number of functions are provided for this purpose. It is up to the code generator implementor to call these in the right order to match and traverse the IL-tree.

3.5.3 Register allocation. BEG can automatically generate a register allocator. This is a capability missing from the other three tools. However, the question is how useful this feature is, i.e., the quality of the generated local register allocator. This is discussed in more detail in Section 6. It should be pointed out that the constructs used in code generator specifications to describe the target register allocation are very well integrated into the BEG specification language.

3.5.4 Tree transformations. TWIG has the advantage compared to the other tools of being able to expand and transform trees. It is possible, by using the REWRITE directive, to specify that the tree should be modified and that a new attempt to match should be performed if some rule matches and the cost is low enough. CGSS and BEG lack this feature. However, TWIG has no counterpart to the EVAL-part of a BEG rule. When a rule has matched in TWIG there is no possibility of executing something. Thus, it may be difficult to synthesize those attributes which are needed by the cost function to determine if a rule should be used or not (conditional attributes). BURG is somewhat less powerful than TWIG due to the lower level of user interface, but this makes BURG also more adaptable to different traversal and evaluation orders.

4. AXE APPLICATION

As mentioned in the introduction, we describe two application examples of using code generator gener-

ator technology in this article: the generation of a code generator in an industrial setting for a special purpose processor used in telephone exchange applications, as described in Sections 4–8, and the generation of a code generator for the well-known SPARC RISC architecture with register windows, which is described in Section 9.

One goal for this study was to investigate the feasibility of porting an existing code generator produced using the CGSS system to instead use the BEG system. The code generator is part of a compiler for a high-level language used to program a special purpose processor used in phone exchanges. Before describing the details of using CGSS and BEG to generate a code generators for this processor, it is useful to give an overview of the application environment, the processor and the language.

4.1 Application environment

AXE is the latest generation of phone exchanges from the Swedish Ericsson corporation. It contains group selectors, tone generators, etc., which are controlled by software executing in several processors. All functionality is controlled by software and can therefore easily be extended or changed. In an AXE-exchange there are several different regional processors which are controlled by a central processor. To increase reliability, there are two identical copies of each processor which execute the same program independently. By comparing the results, certain low level errors can be detected and dealt with. The software architecture is based on processes, called blocks, which communicate by sending and receiving messages called signals.

4.2 APZ central processor

The central processor in an AXE system is a special processor called APZ which has been developed by Ericsson. It has 64 registers divided as follows:

- *CR*, used for comparisons.
- *IR*, an index register.
- *PR0* and *PR1*, registers which can be used when index- or record-declared variables should be accessed.
- *DR0-DR23*, used to send signals, i.e., small messages, between processes.
- *AR0-AR3* and *WR0-WR29*, normal registers.

The memory in the AXE central processor is divided into program memory, data memory, and reference memory. This provides a simple way of memory protection between different processes

(blocks), for both program and data. The register-based instructions are much faster than instructions where memory is accessed. This is because at each access of a variable stored in memory, the size, and type of the variable is looked up in the reference memory before its value can be accessed. Besides providing some protection, this indirection scheme simplifies updating and changing programs at run-time.

The processor has no register which can be used directly as a stack pointer. This is currently not needed, since the "high-level" language used for programming these systems provides no procedures and procedure-calls in the conventional sense. However, hardware support for process programming is available. A real-time clock controls process switching. Procedure calls are emulated by sending signals to invoke code sections in other processes which can return results by signaling back.

4.3 ASA assembly language

ASA is the assembly language used to program the APZ processor. It is possible to change the instruction set by exchanging the micro code of the processor. If this possibility is used, the backward compatibility is then lost. Appendix 1 contains a short description of the assembly instructions used in the examples in this paper. A complete description can be found in the ASA manual (ASA).

5. PLEX LANGUAGE

PLEX is the high-level language used to program Ericsson's AXE phone exchanges. It is a special purpose language, developed specifically for this programming task. Certain aspects of its structure are similar to COBOL, e.g., the declare-section. The expression part of the language is similar to FORTRAN. The language was developed in the beginning of the 1970s and feels somewhat "old-fashioned" now twenty years later. There are no constructs such as procedures or functions in Pascal and C, since there is no general execution stack. However, something similar to procedure call and return can be emulated by sending messages between code blocks executing in separate processes. However, there are constraints on the number of processes, the size of code blocks, etc. This is because these programs execute in a real time environment with hard requirements on execution and waiting times.

In order to further decrease the execution time of a PLEX program, the storage of variables is strictly

determined. PLEX variables can either contain positive integers or strings. There is no need for floating point numbers in a phone exchange. However, it is possible, for reasons of efficiency, to specify the size of integer variables. It is also possible to specify that an integer variable should reside in a register. Such a variable is called a temporary variable, and will *always* be in a register.

5.1 A compiler for PLEX

The compilation of a PLEX program involves several phases. There are several analysis and optimization passes in the compiler which result in more efficient code than a single-pass approach.

The compiler consists of a front-end and a back-end. The front-end part is of no special interest for code generation and will not be further described in this report. It parses the source program, checks syntax and declarations, and produces intermediate code in a textual form of IL-trees which are used by the back-end. The back-end, will be described in more detail, as shown in Figure 8. The Code Selector, labeled CS, is one of the nine modules of the PLEX compiler back-end.

Currently, the PLEX front-end is available on both IBM370 and Sun workstations. For both machines, it is largely generated from formal specifications. However, the back-end is only available on the IBM370 and is hand-coded in PASCAL except for the instruction selector which has been generated by CGSS.

5.1.1 An overview of the modules of the PLEX back-end. The intermediate language program representation produced by the front-end and read by the back-end is called PIL, Plex International Language. The CE module transforms the PIL code into a simpler form called SPIL, Simple Plex Intermediate Language. The transformations performed are assigning values to symbols, expansion of nested statements, and simplification of expressions. After this pass, the produced SPIL code is optimized by a simple peephole optimization pass, CG.

The SPIL code is subjected to an exhaustive program flow analysis by the CB module, which splits the code into basic blocks. The temporary variables and pointers are allocated to registers by the global register allocation module, CT, using algorithms for coloring of non-planar graphs.

The CF module performs a preorder traversal of the SPIL code, calling the CGSS-generated code selector (CS) using the operators of the IL-nodes as arguments. Both the traversal module, CF, and the

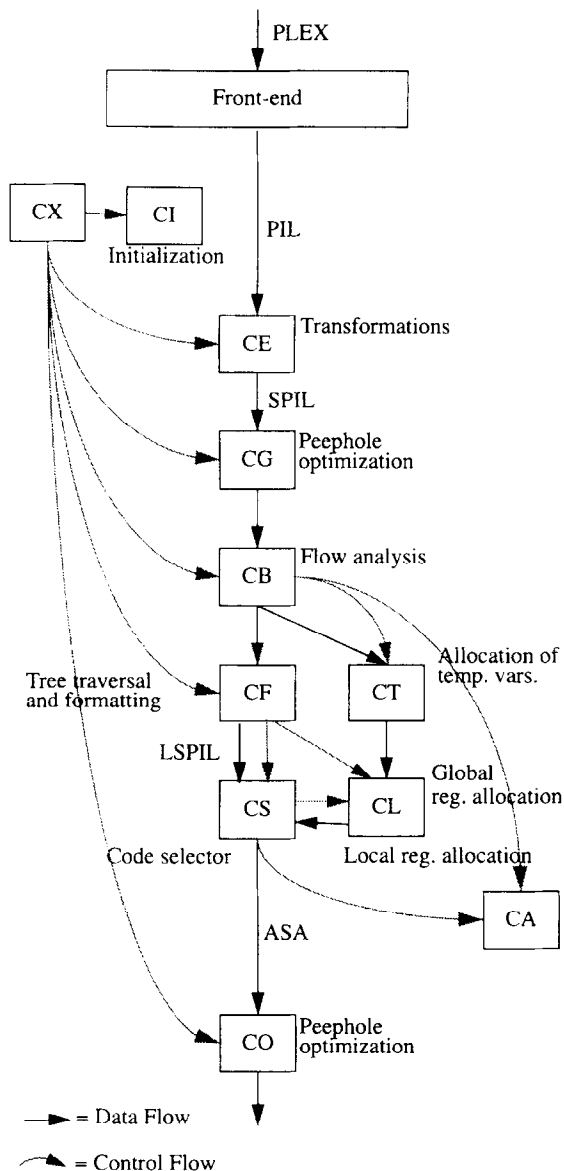


Figure 8. The module structure of the PLEX back-end.

code selector module, CS, call the local register allocator (CL) in order to reserve registers. The local register allocator (CL) uses information from the global register allocator (CT) to achieve a high quality allocation. The code selector (CS) produces an intermediate representation of the target assembly code which, like the SPIL code is subjected to a simple peephole optimization pass in order to eliminate redundant instructions. Finally, the assembly code is assembled and linked.

6. PORTING THE CODE SELECTOR

Since the PLEX front-end is available both for the IBM370 and the Sun workstation, it is desirable to

also have the back-end available on the workstation in order to be able to offer a complete programming environment including full compilation to users on both types of machines. There are at least three ways of porting the CGSS-generated code selector from the IBM370 to the Sun 3, excluding porting the CGSS system itself which was judged unrealistic:

1. To automatically transform the CGSS-generated PASCAL code into C code.
2. To translate the specification of the code selector in the CGSS specification language to the BEG specification language, perhaps in a semi-automatic way.
3. To write a new specification of a code selector for the APZ target machine in the BEG specification language, also using the fact that BEG can generate a local register allocator.

The first alternative has two disadvantages. It may result in a degradation of the performance of the compiler since the automatic PASCAL to C translation is not optimal. This approach may also cause maintenance problems since all changes have to be done on the original PASCAL source code on the IBM370 mainframe including performing the generation step, and the resulting PASCAL code transported to the workstation for further translation to C. The second alternative will probably require the least amount of work, whereas the third alternative will result in an improved, shorter, and more readable specification which also can be used to generate the local register allocator. Ericsson eventually decided on the second alternative.

6.1 Transformation of CGSS specifications to BEG specifications

Since the specification language used by CGSS is a mixture between CGSS-specific directives and PASCAL, it is difficult to write an automatic translator to perform the port, according to alternative two above. Instead, a mixed approach is a suitable compromise. A text editor can be used to manually replace CGSS keywords by corresponding BEG directives. For instance, compare the example in Figure 9. Some directives also have to be moved. However, the specification of the operators and the non-terminals have to be rewritten completely by hand. It looks quite different in BEG compared to CGSS and BEG has more features that are missing in CGSS.

In this case, the specification for the BEG system will have no operators of type REGISTER, since we

```

A CGSS rule:
RULE reg.r0 ::= plus.o reg.r1 SAFE_REG.r2 ;
COST 32;
ACTION
  (*Generate : AR r1-r2 *)
  cscoAR ( r1.kind, reg_kind (r2.kind) )
  ;

  Generate_carry (o.carry.carry_goto,
    o.carry.not_carry_goto);

  r0.kind := r1.kind ;
  r0.len := 16 ;
  r0.nr_of_reg := 1 ;

  csxwregs[0] := reg_kind (r2.kind) ;
  clrafreg (csxwregs, 1) ;
END_RULE ;

```

The same rule transformed into a BEG rule:

```

RULE plus.o reg.r1 SAFE_REG.r2
-> reg.r0 ;
COST 32;
EMIT {
  /*Generate : AR r1-r2 */
  cscoAR ( r1.kind, reg_kind (r2.kind) )
  ;

  Generate_carry (o.carry.carry_goto,
    o.carry.not_carry_goto);

  r0.kind = r1.kind ;
  r0.len = 16 ;
  r0.nr_of_reg = 1 ;

  csxwregs[0] = reg_kind (r2.kind) ;
  clrafreg (csxwregs, 1) ;
};

```

Figure 9. A comparison between a CGSS-rule and a corresponding BEG rule.

are not intending to generate a local register allocator. Instead, we will use the local register allocator previously implemented.

Finally, the formatting and tree traversal module CF must be rewritten. Since CGSS and BEG provide different programmatic interfaces, this is perhaps the most complicated task in the port. The IL trees must be traversed postorder instead of preorder as before. Additionally, when using BEG, different functions have to be called for different operators, as opposed to the current situation using CGSS where one function is used for all operators.

6.2 Writing a new specification for BEG

While no part of the compiler is implemented on a Sun3, the only possible method to test a code generator is to write a new stand-alone module with small stubs to make it work. To limit the size of the code generator, only a subset of the existing code generator is implemented. The subset version can handle expressions and assignments.

6.2.1 Input data to the code selector. In order to be able to test the generated code selector, interface routines are needed to feed the code selector with correct IL trees. The only realistic possibility of doing this without having to manually write large test cases, is to use the existing compiler to generate a trace printout of the intermediate trees after the CF formatting module. Such a printout, which was generated through a preorder traversal of the intermediate tree, can then be moved over to the workstation and used as input data for the program routine to the BEG-generated code selector.

The interface program transforms the prefix textual representation to a postfix representation which can read by the BEG-generated code selector. The reader part of this conversion routine was implemented as a scanner and a parser generated by Lex and Yacc. Finally, the intermediate code is on a form which can be directly used by the BEG-generated code selector.

6.2.2 The register allocator. When PLEX compiler allocates local registers, the local allocator (CL) is guided by information from the global register allocator (CT) about which registers are available for allocation. If the new BEG-generated code selector should be able to handle expressions which contain temporary variables, a global register allocator has to be present. A very simple global allocator was implemented for this purpose. It is presented at the end of the GET-type code selector specification (see Appendix 2) and consists of two functions:

- *AllocTempVar*, which allocates a new register.
- *ContentTempVar*, which returns to register to which a certain temporary variables have been assigned. If the specified temporary variable has not been allocated, zero is returned.

Note that this simple global register allocator cannot even deallocate registers. This is not needed for testing the simple test programs described in this paper. By assigning the variable *localavail* before allocation starts, the globally allocated registers can be made inaccessible to the local register allocator, i.e., they are given a read-only status.

Before the formatting module CF calls the code selector CS, it has determined the order of evaluation of expressions and where intermediate results should be stored. In certain cases, CF cannot determine where the intermediate results should be stored and leaves this task to the CS module. The code selector (CS) in this case will choose the first available free register. In this paper, we have chosen to

study the worst case. Therefore, we do not use the information that might have been specified by CF regarding evaluation order and storage of intermediate results.

We will now describe the code selector itself.

6.2.3 CGD (Code Generator Description). CGD (Code Generator Description) is the input specification accepted by the BEG system in order to generate a code generator. Version 1.53 of BEG is used during this implementation. Such a specification contains three main parts: (1) a description of the operators of the IL, (2) a description of the non-terminals used in the code selector specification, and (3) a set of rules.

6.2.3.1 Operators. In a CGD for BEG, the first part is a description of the IL headed by the OPERATORS keyword. This is simply an enumeration of all operators which the code selector is expected to handle. The number of operands for each type of operator are specified, as are the results of operator returns (see Figure 10).

It is possible to specify for BEG that an operator is commutative by replacing the asterisk between the operands by a plus symbol. This is only allowed for operators which have two operands.

The following operators have been used for the implementation described in this paper:

```

Constant (v: int)
    → Value;
Plus
    Value + Value → Value;
Mult
    Value * Value * Value * Value * Value
    Value;
Content
    Value → Value;
Assign
    Value * Value;
Enter (signal: string);
Sub (offset: int; length: int)
    Value → Value;
Simple_var (name: string; length: int)
    → Value;
  
```

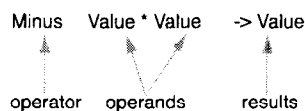


Figure 10. An example of a specification of an operator. *Value* is a declared non-terminal which is used only in the operator description.

```

Str_var (name: string; length: int)
    → Value;
TempVar (name: string; length: int)
    → Value;
Matrix
    Value * Value * Value → Value;
Reg (name: string)
    → Value;
(*Is used to get a temporary register in Mult *)
Free Reg
    → Value;
Exit;
  
```

This is just a subset of the operators available in SPIL.

Enter and **Exit** have been included just to be able to write small but complete PLEX programs which can be transformed into intermediate code trace files that can be fed into the new code selector without manual editing. **Mult**, which has many operands, will be explained later.

6.2.3.2 Non-terminals. In order for the code selector to be able to connect different rules in a BEG specification, non-terminal symbols are needed. These can be thought of as intermediate results of computations, either explicit results or implicit ones, e.g., an address computation which is made when using an indirect operation. It is important to keep this correspondence between non-terminals and computations, in order to get more readable specifications which are easier to understand in terms of the application. Therefore, introducing new non-terminals should be avoided only for the purpose of breaking rules into smaller pieces.

The following declarations were written for our PLEX code generator:

A constant where the length has been computed:

```

Const COND_ATTRIBUTES
    (v: int; length: int);
  
```

A stored integer variable.

```

DS ADRMODE
    COND_ATTRIBUTES
    (name: string; length: int);
  
```

A stored sub-variable = A field in a record.

```

DSS ADRMODE
    COND_ATTRIBUTES
    (name: string; length: int; offset:
    int);
  
```

An intermediate result which exists in a register.

Register REGISTERS

```

(dr0, dr1, dr2, dr3, dr4, dr5, dr6,
dr7, dr8, dr9, dr10, dr11, dr12,
  
```

```

dr13, dr14, dr15, dr16, dr17, dr18,
dr19, dr20, dr21, dr22, dr23, ar0,
ar1, ar2, ar3, wr0, wr1, wr2, wr3,
wr4, wr5, wr6, wr7, wr8, wr9, wr10,
wr11, wr12, wr13, wr14, wr15, wr16,
wr17, wr18, wr19, wr20, wr21, wr22,
wr23, wr24, wr25, wr26, wr27, wr28,
wr29, ir, pr0, pr1, cr, sr0, sr1)
COND_ATTRIBUTES
(length: int);

```

A temporary variable. PR0, PR1, IR, CR, SR0 and SR1 do not belong to the set of allocatable registers.

(* TempVar is used to make assignment from one TempVar to another Temp Var possible *)

TempVar REGISTERS

```

(dr0, dr1, dr2, dr3, dr4, dr5, dr6,
dr7, dr8, dr9, dr10, dr11, dr12,
dr13, dr14, dr15, dr16, dr17, dr18,
dr19, dr20, dr21, dr22, dr23, ar0,
ar1, ar2, ar3, wr0, wr1, wr2, wr3,
wr4, wr5, wr6, wr7, wr8, wr9, wr10,
wr11, wr12, wr13, wr14, wr15, wr16,
wr17, wr18, wr19, wr20, wr21, wr22,
wr23, wr24, wr25, wr26, wr27, wr28,
wr29)
COND_ATTRIBUTES
(length: int);

```

By specifying the REGISTER keyword after a non-terminal, the local register allocator will interpret this as meaning that a register should be allocated for the intermediate results represented by the non-terminal.

6.2.3.3 Rules. Using the available operators and non-terminals, we will now describe the rules of our example code selector specification.

Since it is possible to synthesize attributes already during the first phase of code generation, constant expressions can be computed already at compile time. However, in the present PLEX compiler, this has already been done in an earlier phase (CE). In order to be able to generate compact code, we need to know the size of constants. Therefore, the non-terminal **Constant** and its associated rule were introduced:

```

RULE Constant → Const;
COST 0;
EVAL {Const.v = Constant.v;
      Const.length = SizeofConst (Constant.v);};

```

where SizeOfConst is a function which returns the length of the constant.

Hereafter, we will use *Const* and not *Constant* in all cases where a constant is needed. In this way, we force a computation of the length of the constant each time which can be used later in conditions.

Now we turn to the task of defining the simple rules—those which do not contain any conditions:

```

RULE Enter;
COST 10;
EMIT {
.> RECEIVE {s Enter.signal}
};
RULE Exit;
COST 10;
EMIT {
.> EP
};
RULE Const
→ Register.a;
COST 22;
EVAL {a.length = Const.length;};
EMIT {
.> LHC {*a}/W0-{{i(Const.v & ((1 << 8)-1))}}
.> LHC {*a}/H1 = {{i(Const.v >> 8)}}
};
RULE Plus
Register.a
Register.b
→ Register.r;
COST 9;
EVAL {r.length = 16;}; (*Can be smaller! *)
TARGET b;
EMIT {
.> AR {*b}-{*a}
};

```

The costs are used here only to select between similar rules during pattern matching. It should of course be as close to the real costs as possible, for the sequence of instructions generated by each rule. The *length* attribute is assigned a value in the EVAL-part of the rule because it may be referenced later on. The lines starting with a dot are instructions to *dottool*, a preprocessor which converts the line to a sequence of somewhat longer print statements. The APZ instructions used in these examples are explained in Appendix 1.

We also need some rules to handle stored variables. First two rules which only compute and synthesize attributes:

```

RULE Simple_var.v
→ DS;
COST 0;
EVAL {DS.name = v.name;
      DS.length = v.length;};
RULE Sub
DS
→ DSS;

```



```

COST 0;
EVAL {DSS.name = DS.name;
      DSS.length = Sub.length;
      DSS.offset = Sub.offset/Sub.length};

```

Then the rest of the rules for stored variables. These rules emit code for loading or storing variable values.

RULE Content

DS

```

→ Register.r;
COST 10;
EVAL {r.length = DS.length;};
EMIT {
  .> RS {*r}-{s DS.name}
};

```

RULE Assign

DS

Register.a;

```

COST 10;
EMIT {
  .> WS {s DS.name}-{*a}
};

```

RULE Content

DSS

```

→ Register.r;
COST 10;
EVAL {r.length = DSS.length;};
EMIT {
  .> RSS {*r}-{r DSS.name}/{z
        DDS.length}{i DDS.offset}
};

```

RULE Assign

DSS

Register.a;

```

COST 11;
EMIT {
  .> WSS {s DSS.name}/{z DSS.length}{i
        DSS.offset}-{a*}
};

```

Using the rules presented so far, it is possible to generate code for additions of stored variables. Still, the target code is not efficient enough, e.g., see Figure 11 but it has a general structure: the rule that loads a constant into a register can handle *all* constants.

We continue by adding rules for more efficient code. Such rules are usually more specialized, i.e., they cannot be used in as many places as the general rules. Therefore, conditions must be introduced to express constraints where the rules apply. Remember that the conditions are only allowed to test attributes previously declared in the non-terminal section of the CGD using COND_ATTRIBUTES (see Section 3.2.2 and Section 6.2.3.2). No other attributes may be tested.

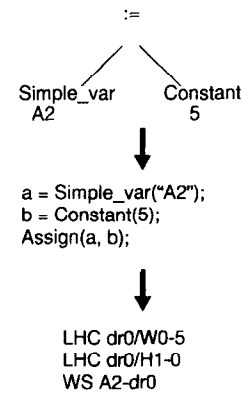


Figure 11. An example of code produced by the first version of our code selector using only general rules.

RULE Assign

DS

Const;

```

CONDITION {Const.length == 0};
COST 9;
EMIT {
  .> WZ {s DS.name}
};

```

RULE Assign

DS

Const;

```

CONDITION {Const.v == ((1 << DS.length)-1)};
COST 9;
EMIT {
  .> WO {s DS.name}
};

```

RULE Const

→ Register.a;

```

CONDITION {Const.length <= 4};
COST 10;
EVAL {a.length = Const.length;};
EMIT {
  .> LCC {*a}-{i Const.v}
}; (*Rest of a is cleared *)

```

RULE Const

→ Register.a;

```

CONDITION {Const.length <= 8};
COST 11;
EVAL {a.length = Const.length;};
EMIT {
  .> LHC {*a}/{z Const.length}0-{i Const.v}
};

```

RULE Plus

Const

Register.a

```

→ Register.r;
CONDITION {Const.length <= 4};
COST 8;
EVAL {r.length = 16;};
TARGET a;

```

```

EMIT {
.> ACC {*a}-{i Const.v}
};
RULE Assign
  DS
  Const;
CONDITION {DS.length <= 8};
COST 10;
EMIT {
.> WHC (s DC.name)-{i(Const.v &
  ((1 <= DS.length)-1))}

```

Finally a more advanced rule:

```

RULE Assign
  DS.v
  Plus
  Content
  DS.w
  Const;
CONDITION {(strcmp(v.name, w.name)) &&
  (Const.length <= 8)};
COST 9;
EMIT {
.> AHCS{s v.name}-{i Const.v}
};

```

Almost all rules check the length attribute. From this observation, we draw the conclusion that the choice of instruction in the ASA assembly language often is influenced by the length of the operand.

Already, the code selector starts to generate efficient code. For example, only the instruction **WHC A2-5** is generated from the IL-tree in Figure 12. So far we have no rules to handle temporary variables. Now is the time to also implement such rules:

```

RULE Assign
  Temp_var.t
  Register.v ({ContentTempVar(t.name)});
CONDITION {t.length >= v.length};
COST 0;
RULE Assign
  Temp_var.t
  Register.v ({ContentTempVar(t.name)});
COST 10;
EMIT {
.> NHC {*v}-{i(1 <= (t.length))-1}
};
RULE Content
  Temp_var.t
  → Register({ContentTempVar(t.name)});
COST 0;
EVAL {Register.length = 16};
  (*Can be smaller!
  *)

```

The second rule truncates the value if the destination variable is shorter than the result. After that, these three rules have been appended to the CGD, the code selector can handle temporary variables.

The rules which have been presented so far can only generate a single instruction each. How about rules for generating several instructions? The structure of such rules is of course the same. However, as usual, we have to be careful with register allocation. As an example, consider multiplication. Since the ASA assembly language has no instruction for multiplication, the PLEX-compiler generates a small code sequence which needs five registers in order to perform the multiplication. One of these is the IR register. First, we declare the operator in the most simple-minded way:

Mult

Value * Value → Value;

After that, we declare the rule. For the sake of simplicity and readability, the labels in this example here have been inserted directly into the rule itself. In the real code generator, these labels must be generated by some procedure in order to keep all labels unique.

RULE Mult

```

  Register.fak1 (dr0..wr29, pr0..sr1)
  Register.fak2 (dr0..wr29, pr0..sr1)
  → Register.prod (dr0..wr29, pr0..sr1)
COST 100;
EVAL {prod.length = 16};
CHANGE (ir);
EMIT {
.> LCC {*prod}-0
.> LCC ir-15
.> Label11)
.> BLO {*fak2},0,Label12
.> JLN label13
.> Label12)
.> MFR {*temp}-{*fak1}
.> SHL {*temp},0
.> AR {*prod}-{*temp}
.> JLN label11
.> Label13)
};

```

How then, should the allocation of the local register *temp* take place? Note that after the **CHANGE** keyword, a specific register has to be mentioned. This is overly constraining and may lead to inefficient code if there are few free registers available. The register that was specified may already have been allocated.

It is possible to avoid this problem by specifying an additional operand, a dummy operand (**FreeReg**), and to modify the interface program to produce **Mult** nodes that use this dummy. Then the allocator can choose a register that is free. Since **Mult** now has three operands, it has lost the property of commutativity. However, commutativity is not necessary in this case since both factors (*fak1* and *fak2*) must be in registers.

Mult

Value * Value * Value → Value;

RULE FreeReg

→ Register;

COST 0;

RULE Mult

Register.fak1 (dr0..wr29, pr0..sr1)

Register.fak2 (dr0..wr29, pr0..sr1)

Register.temp (dr0..wr29, pr0..wr1)

→ Register.prod (dr0..wr29, pr0..sr1)

COST 100;

EVAL {prod.length = 16};

CHANGE (ir);

EMIT {

.> LCC {*prod}-0

.> LCC ir-15

.> Label11)

.> BLO {*fak2},0,Label12

.> JLN label13

.> Label12)

.> MFR {*temp}-{*fak1}

.> SHL {*temp},0

.> AR {*prod}-{*temp}

.> JLN label11

.> Label13)

};

This is not enough to produce correct code. Simultaneously as the rule is being executed, the registers used by the operands are freed. When a register is allocated to hold the result, it may happen to be one of the freed operand registers. However, this is not allowed by this multiplication algorithm due to the used machine instructions. The result may not be put in the same register as one of the operands. One solution is to add additional dummy operands which allocate registers for the result. The CHANGE directive is also removed, since it does not work well with assignment of *localavail* (Section 6.2.2), i.e., the set of locally available registers. Instead, an additional dummy operand is used to allocate this register, together with the constraint that it must be the IR register.

Mult

Value * Value * Value * Value * Value

→ Value;

RULE FreeReg

→ Register;

COST 0;

RULE Mult

Register.fak1

Register.fak2

Register.temp

Register.IR (ir)

Register.prod

→ Register.a;

COST 100;

EVAL {a.length = 16};

TARGET prod;

EMIT {

.> LCC {*prod}-0

.> LCC ir-15

.> Label11)

.> BLO {*fak2},0,Label12

.> JLN Label13

.> Label12)

.> MFR {*temp}-{*fak1}

.> SHL {*temp},0

.> AR {*prod}-{*temp}

.> JLN label11

.> Label13)

};

This constraint mechanism can also be used in the rule for the matrix operator.

RULE Matrix

DS.a

Register(pr0)

Register(ir)

→ DS.b;

COST 0;

EVAL {b.name = a.name;

b.length = a.length;};

This rule forces the indices to be put in *PR0* and *IR* when accessing a user-declared matrix variable. A complete listing of the code selector discussed in this section is available in Appendix 2.

6.2.4 An incremental methodology of developing CGD. According to our experience, an incremental strategy of developing CGDs works very well. The developer should first concentrate on specifying a code selector which is partially correct, i.e., that it can generate code for all types of input data it can handle. In our case, this means at least one rule for each input tree node type. The next step is to make the specification complete enough to handle all possible combinations of input data (Emmelmann, 1992). Then the specification is correct. Now we can add rules for the generation of more efficient code. It is impossible to destroy the completeness of the specification by adding more rules. This is the beauty of the code generation scheme based on pattern matching in combination with dynamic programming.

7. TESTING A CODE SELECTOR

What then, is the quality of the generated code generator? Is the code optimal? Is the code good enough that the code selector can be used in the back-end of the production PLEX compiler? In order to obtain at least approximate information about the answer to these questions, several PLEX pro-

grams were written for testing. The intermediate code trace files produced by the CF module of the compiler were fed into the new code selector. The assembly code produced by the code selector was then compared with the code produced by the original PLEX compiler (see Figure 12).

The trees produced by the CF module and fed into the code selector are already simplified and improved. Variables have been moved and constant expressions are already computed. CF has determined the best evaluation order for expressions. For these reasons, the new code generator generates high quality code, sometimes better than the original PLEX compiler (See Figure 13), if the complexity of the compiled programs is not too great. This is primarily because the way the new rules have been written and not because we use BEG instead of CGSS.

The fact that the code produced by the two code generators is not the same is usually caused by differences in the register allocation. How good then, is the quality of code from the new code generator for complex intermediate trees? In order to avoid writing advanced PLEX programs which, when compiled, would give rise to the desired intermediate language trees (the turn-around time was rather long since two different computer systems had to be used), the necessary restructuring and modifications

of the IL trees were done by directly editing the IL trace file.

Now the resulting code sequences can no longer be directly compared because the intermediate code is modified manually. However, this would still be hard to do because of the different algorithms used for the global register allocation. As mentioned before (Section 6.2.2), the BEG-generated code selector has a very simple global register allocator compared to the existing one.

In order to show how hard it is to control the BEG-generated register allocator we present an example based on the assignment statement $T2 = (T2 + 80) + (T2 + 90)$, where $T2$ is a temporary variable. Different results are shown in Figure 14, depending on the order in which the operands occur in the tree.

One may notice that the code generator sometimes moves intermediate results around. This happens only if the register allocator cannot exploit commutativity.

The current production PLEX-compiler does not allow spilling intermediate results to memory, due to the time constraints. The generated code sequences in this example meet this demand, but the efficiency is still low (Figure 14). The A, C, and D alternative move around the intermediate results, which leaves B as an acceptable alternative. Summarizing, we may conclude that the ordering within the input IL trees to a large extent determines the code, since the code selector never emits code in the reverse order. Unfortunately, it cannot be guaranteed that the BEG-generated register allocator never spills.

8. PORTING FROM CGSS TO BEG

To change code generator generation tool from a parser-based system, such as CGSS, to a dynamic

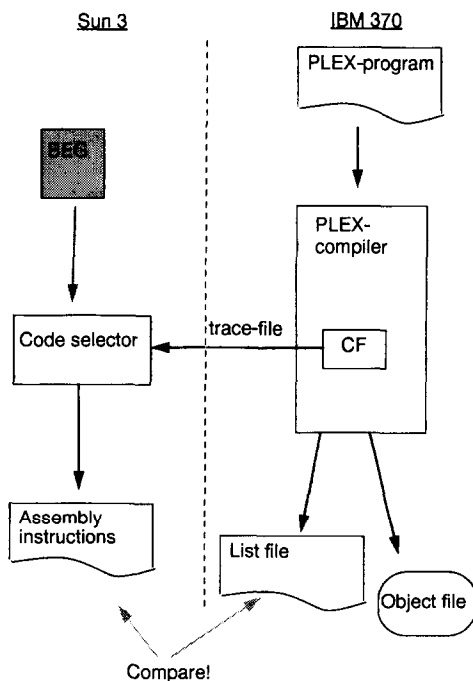


Figure 12. Transportation of data when testing a BEG-generated code selector.

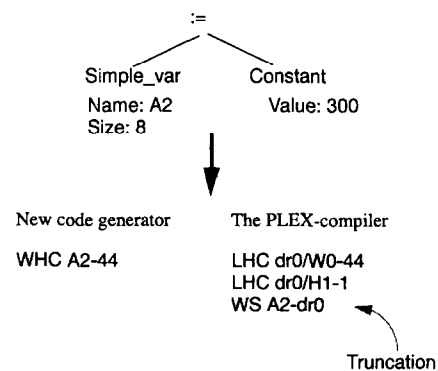


Figure 13. Comparison between the new code generator and the current PLEX compiler.

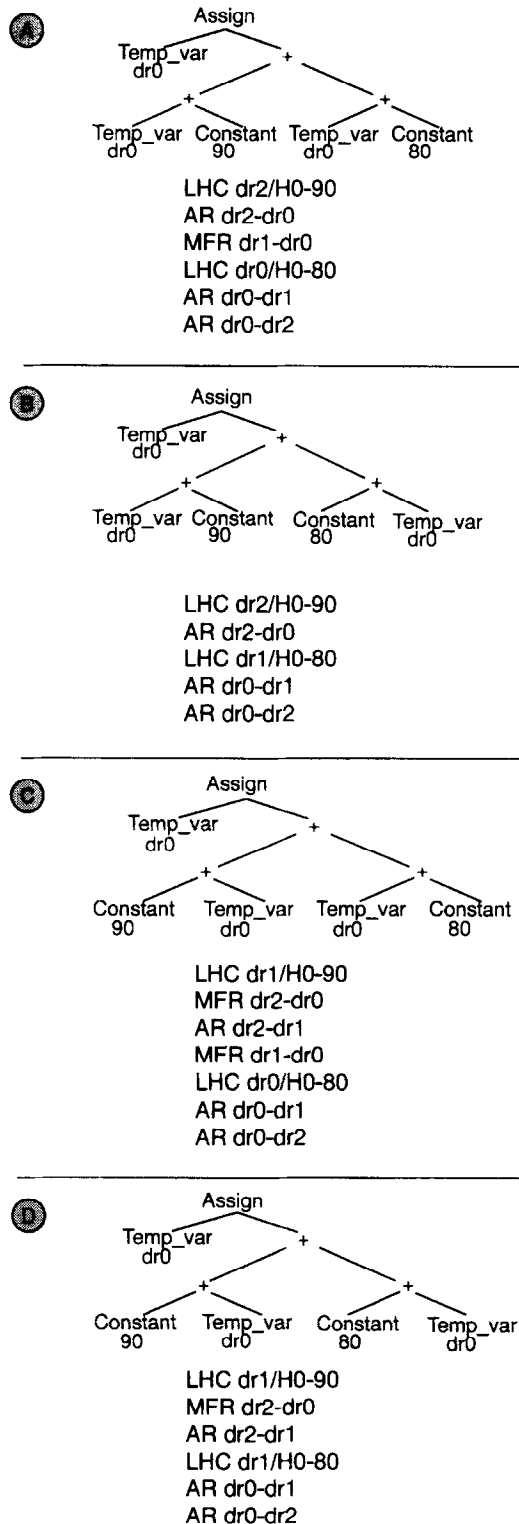


Figure 14. Different code sequences are produced depending on the ordering within the expression.

programming based system, such as BEG, causes no serious problems. Since BEG uses a better algorithm for pattern matching, no serious problems occur even if the existing CGSS specification is moved over to BEG edited, of course, to conform to the BEG specification language. Size and execution time differences between CGSS and BEG have been impossible to measure since the complete stand-alone code generator for PLEX currently cannot execute on the Sun3 workstation. Many additional modules would have to be translated and ported in order to have a complete code generator up and running on the Sun3.

If we choose to write a new specification in BEG, instead of adapting the existing CGD specification, it is best to do this development in an incremental fashion. Experience has shown that such an incremental development style works well using these kinds of tools. It is also possible to improve an existing code generator, partly by adding rules for more efficient code and partly by removing redundant rules.

8.1 Quality of generated register allocators

The register allocator used by BEG is not of high enough quality to be used in a production PLEX compiler. It is not possible to prevent spilling. Since spilling is not allowed in PLEX, it cannot be used. Perhaps a future version of BEG will contain a register allocator which can understand that certain operators are commutative and allocate registers, possibly generate code, in the reverse order. Perhaps such an allocator would generate good enough code to be used in a PLEX compiler.

9. SPARC APPLICATION

From our group, Mikael Petterson used BEG for the implementation of this code generator for the SPARC architecture as part of an effort to port the incremental PASCAL compiler of the DICE system (Distributed Incremental Compiling Environment) (Fritzson 1983) to the SPARC, and was later slightly adapted and modified by our group. The code generator specification for this SPARC code generator (excluding code emission and utility functions) can be found in Appendix 3 of this article.

The main difficulty using BEG to develop this code generator involved how to handle register windows correctly, while being able to generate reasonable efficient code, handling nested function calls, and being compatible with the SUN PASCAL and C calling conventions. In fact, we needed three design iterations of the code generator to get it right. It is

notable that the Mocca Modula-2 compiler developed by the Karlsruhe compiler group using BEG does not make use of register windows. They chose the easier option of building the Modula-2 track directly in memory, which, however, makes cross-callability to C and Pascal more troublesome.

The SPARC has 32 integer registers (i0–i7, o0–o7, g0–g7) and 32 single precision floating point registers (f0–f31). The latter are usually used in pairs corresponding to 16 double precision registers. The 32-bit integer registers are divided into 8 local registers (i0–i7) which are allocated new for each function activation; input registers (i0–i7) which hold some (formal) parameter values which were transferred when the current function was called; output registers (o0–o7) which are used to transfer some actual arguments at function calls, and the global registers (g0–g7) which are always the same, i.e., no new register instances are created by the machine. At function call, the output registers containing the actual parameter values automatically become the input registers of the called functions, containing formal parameter values, without any physical data transfer taking place which prove to be very efficient. The first six registers of the output and input registers (o0–o5, i0–i5) are used for parameter transfers. If there are more than six parameters to be transferred, an overflow area on the stack is used for additional parameters. Function return values are put into i0 which becomes o0 in the caller; floating point return values are input into f0 and f1.

Some registers are reserved for special purposes. For example, g0 always contains the constant 0; g1–g5 are scratch registers sometimes used by routines in the run-time system; g7 points at the global object table; o7 holds the outgoing return address which becomes the incoming return address in i7; o6 contains the stack pointer (sp) at function call which becomes the frame pointer i6, also denoted fp. Floating point registers are only used in pairs denoted (f0, f2, ..., f28) since we are only doing double precision floating point arithmetic.

Specifying the integer arithmetic and logical operations and floating point arithmetic is rather straightforward and similar to what is one for many other processors and to some extent has been discussed in the introduction. The code generation rules for these operations can be found in Appendix 3. Thus, we focus mainly on function calls and register windows in the rest of this section. Register windows implement a very efficient parameter passing mechanism as shown in Figure 15, since out-registers are automatically converted to in-registers at

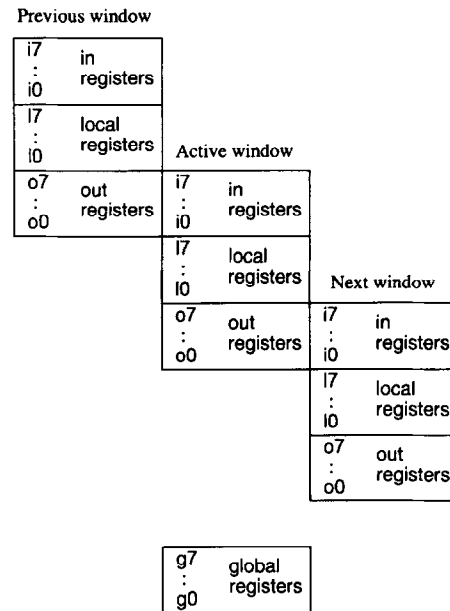


Figure 15. Parameter passing using SPARC register windows.

function calls, and new local and out registers are allocated from the register stack by the hardware.

One difficulty involves dealing with nested function calls (i.e., actual argument expressions which contain function calls), since parameter values are passed in the registers o0–o6. For example, the code generator might have emitted code for the first two arguments of a function call, storing those argument values in registers o0 and o1 which then are marked as unavailable. Now the third argument happens to contain a function call, which, in order to be evaluated needs to transfer its arguments into o0 and o1. However, since o0 and o1 are already reserved, they must be spilled to some spilling area before this nested function call can be evaluated. We had difficulty expressing in the BEG code generator specification language that is register may be both allocatable and reserve for some purpose, like parameter passing. If we, on the other hand, always preserve all output registers for parameter passing and first store argument values in other registers, poor code containing unnecessary register transfers and spilling will be emitted.

We tried several different ways of expressing these constraints in the BEG code generator description language, but none of them were satisfactory. The problem is caused by limitations in the BEG specification language when expressing sequencing in evaluation. Finally, we solved the problem by insert-

ing an extra transformation of the intermediate tree representation of the program before code generation. The tree was traversed to extract all function arguments containing nested function calls for which code would be emitted for evaluation into temporaries, before code for the actual call could be emitted. Since the register window mechanism can pass at most six arguments in registers, overflow arguments were handled the same way: i.e., they were evaluated before arguments passed in registers, and stored into pre-allocated slots in the stackframe.

10. CONCLUSIONS

10.1 The usage story

In this paper, we have described the development of code generators and our experience regarding two applications—the AXE processor and the SPARC processor. These developments and investigations took place during 1991–1992.

What happened later? Ericsson had to make a decision whether to port their code generator from CGSS to BEG technology, or to keep CGSS but port it from being Pascal-based to C and from the IBM main-frame to a workstation.

Ericsson chose the latter alternative—to keep the CGSS-based code generator but port it to C and a workstation environment. The reasons were two-fold: easy validation and quick availability. By keeping the same code generation method and code generation rules as before, together with the already developed global register allocator, one could avoid extensive retesting and validation of the new code generator and re-compiled application software. The instruction sequences generated by the new code generator would be identical to those from the old compiler. This simplified testing and preserved the timing characteristics of the generated code, which are especially important for Ericsson's real-time applications. Easier validation also contributed to quicker availability of the new code generator. However, an improved BEG is still being considered for possible future projects where the target might be a different processor. The flexibility and power of dynamic programming based code generator descriptions will be advantageous for such new development.

Concerning the developed SPARC code generator, it is part of the incremental Pascal compiler in the DICE programming environment (Fritzon 1983) which is not the regular use but is run occasionally for demo or test purposes. The generated code generator is fast; the code generation time is only 4.5%

of the total compilation time of 0.8 seconds for a typical 400-line example program on a Sparc-2 workstation. The emitted code is of reasonable quality but not optimal, due to the fact that no post-processing peep-hole optimization is currently performed to remove unnecessary load/stores at statement boundaries. This is because the DICE compiler is incremental at the statement level, and thus, code for a single statement should be insertable or removable.

10.2 Recent developments

As noted several times, thus far, global register allocation must be integrated with code generators produced from systems such as BEG to obtain code of acceptable quality. This has, in fact, been achieved for the BEG system during the past two years. The new BEG, however, is not released for public use since it is part of a commercial development.

Another issue which has become increasingly important to obtain efficient code for modern superscalar RISC processors is *instruction scheduling*. In fact, this is becoming one of the most important optimizations for current RISC processors which often have three to six functional units. Also, instruction scheduling needs to be integrated with instruction selection. Thus, a competitive code generator should be able to produce a good instruction scheduler integrated with the instruction selector. Some work on the integration of instruction scheduling into the BEG system is described in (Müller 1993). PAGODE, described in (Can et al., 1993), is another recently developed code generator which also addresses these issues. However, treating these developments in greater detail is outside the scope of this paper.

10.3 Final remarks

It is both possible and desirable to generate the code generation part of a compiler using the kind of tools described in this paper. A declarative and readable specification of the code generator is obtained, which can easily be modified to meet new requirements. In fact, a gradual consensus is being reached (Zadeck, 1994) that really good machine independent optimizers are very hard to achieve for modern superscalar RISC processors, and that generator technology is the best way to produce optimizers and code generators if both portability and very high code quality are the main goals. Results presented in (Aho et al., 1989) also indicates that an

automatically generated code generator often executes faster than a hand-coded one. Since instruction selection is simpler for modern RISC processors and thus less important than for older processors, and good register allocation and instruction scheduling are becoming increasingly important for good code quality, the latter capabilities must also be included in code generator generators to keep this technology competitive. Current research in the area is addressing these issues.

ACKNOWLEDGMENTS

Mikael Pettersson implemented the BEG-generated code generator for the SPARC. Personnel at the Ericsson Telecom programming support department (ETX/TX/DV) provided information about the existing PLEX-compiler and AXE. Helmut Emmelmann and Karlsruhe quickly answered all questions about BEG. NUTEK, the Swedish Board for Technical Development, provided support for part of this work.

REFERENCES

- Aho, A. V., and Corasick, M. J., Efficient String Matching: An Aid to Bibliographic Search, *Communications of ACM*, 18, 6 333-340 (1975).
- Aho, A. V., Sethi, R., and Ullman, J. D., *Compilers: Principles, Techniques and Tools*, Addison-Wesley, Reading, Massachusetts, 1986.
- Aho, A. V., Ganapathi, M., and Tjiang, S. W. K., Code Generation Using Tree Matching and Dynamic Programming, *ACM Transactions on Programming Language and Systems* 11,4 491-516 (1989).
- ASA211c, Listing of assembly statements, Ericsson Telecom AB, Stockholm, Sweden, 3/1551-ANZ 211 01.
- Canalda, P., Cognard, L., Mzard, M., and Despland, A., PAGODE: a back-end generator for RISC machines. INRIA Technical Report NO. 152, May 1993.
- Chase, D. R., An Improvement to Bottom-up Tree Pattern Matching, *Fourteenth Annual ACM Symposium on Principles of Programming Languages*, Munich, Germany, 1987.
- Emmelmann, H., Schröer, F. W., and Landwehr, R., BEG—a Generator for Efficient Back Ends, *ACM Sigplan Notices*, Vol 24, No. 7, 227-237, (1989).
- Emmelmann, H., BEG—a Back End Generator, User Manual, GMD Forschungsstelle an der Universität Karlsruhe, 1990.
- Emmelmann, H., Testing Completeness of Code Selector Specifications, International workshop on Compiler Construction, 1992.
- Fraser, C. W., Henry, R. R., Proebsting, T. A., BURG—Fast Optimal Instruction Selection and Tree Parsing, *ACM Sigplan Notice*, Vol. 27, No. 4 (1992).
- Fritzson, P., Symbolic Debugging Through Incremental Compilation in an Integrated Environment, *Journal of Systems and Software* 3, 285-294 (1983).
- Graham, S. L., and Glanville, R. S., A New Method for Compiler Code Generation, *ACM 5th Symposium on Principles of Programming Language*, 1978, pp. 231-240.
- Hoffmann, C. M., and O'Donnell, M. J., Pattern Matching in Trees, *Journal of ACM* 29, 1 68-95 (1982).
- Jansohn, H. S., and Landwehr, R., CGSS: A System for Automatic Generation of Code Generators, University of Karlsruhe, Faculty for Information, 1980.
- Landwehr, R., Jansohn, H. S., and Goos, G., Experience with an Automatic Code Generator Generator, *ACM Sigplan Notices*, Vol. 17, No. 6, 56-66 (1982).
- Lunell, H., Code Generator Writing Systems, Ph.D. Thesis, IDA, Linköping, Sverige, 1983.
- Müller, T. Finite Automata for Resource Scheduling, in *Proc. of MICRO-26—26th Annual Int. Symp. on Microarchitecture*, IEEE in SIGMICRO Newsletter, Vol. 24, Dec. 1993.
- PLEX-C Language description, Ericsson Telecom AB, Stockholm, Sweden, LZB 101 1903 R1A.
- Pelegri-Llopart, E., and Graham, S. L., Optimal Code Generation for Expression Trees: An Application of BURS Theory, *Fifteenth Annual ACM Symposium on Principles of Programming Languages*, San Diego, California, 1988.
- Tjiang, S. W. K., *Twig Reference Manual*, AT & T Bell Laboratories, 1986.
- Zadeck, K., Personal Communications, IBM Labs, New Jersey, USA.

Appendix 1. ASA Instructions

RS	r-var	Read from store
RSS	r-var/vln	Read subvariable from store
WS	var-r	Write to store
WSS	var/vln-r	Write subvariable to store
WHC	var/vln-h	Write halfword constant
WZ	var	Write zeroes
WO	var	Write ones
LCC	r-c	Load character constant
LHC	r/vlp-h	Load halfword constant
NHC	r/vlp-h	Mask halfword constant (AND)
MFR	r1-r2	Move from register
AR	r1-r2	Add register to register
ACC	r-c	Add character to register
AHCS	var-h	Add halfword constant to store
c	character constant, 4 bits.	
h	halfword constant, 8 bits.	
r	processor register	
	ir, pr0, pr1, cr, sr0, sr1, dr0-dr23, ar0-ar3, wr0-wr29	
vl	variable length	
	B = 1 bit	
	T = 2 bit	
	C = 4 bit	
	H = 8 bit	
	W = 16 bit	
	D = 32 bit	
vln	Part of composite variable. vl = size, n = position.	
vlp	Part of processor register. vl = size, p = position (0 or 1).	

Appendix 2. Code generator specification (CGD) for the PLEX subset code generator produced by BEG.

```

(*)
PLEX
Costs are not realistic. They are only guiding BEG.
*)

%noonthefly (*No on the fly register allocation *)
%test (* Option for BEG to generate test output routines *)
%RegNameTable

CODE_GENERATOR_DESCRIPTION Plex;
INTERMEDIATE_REPRESENTATION
NONTERMINALS Value;
OPERATORS
  Constant ( v : int)          -> Value;
  Plus      Value + Value      -> Value;
  Mult Value * Value * Value * Value * Value -> Value;
  Div      Value / Value      -> Value;
  Content  Value               -> Value;
  Assign   Value = Value       -> Value;
  Enter ( signal : string ) ;
  Sub (offset: int;
      length: int) Value      -> Value;
  Simple_var (name: string;
             length: int)     -> Value;
  Str_var (name: string;
          length: int)        -> Value;
  TempVar (name: string;
          length: int)        -> Value;
  Matrix   Value * Value * Value -> Value;
  Reg (name: string)          -> Value;
  (* Is used to get a temporary register in Mult *)
  FreeReg -> Value;

  Exit ;

REGISTERS
(*****)
dr0, dr1, dr2, dr3, dr4, dr5, dr6, dr7,
dr8, dr9, dr10, dr11, dr12, dr13, dr14, dr15,
dr16, dr17, dr18, dr19, dr20, dr21, dr22, dr23,
ar0, ar1, ar2, ar3,
wr0, wr1, wr2, wr3, wr4, wr5, wr6, wr7,
wr8, wr9, wr10, wr11, wr12, wr13, wr14, wr15,
wr16, wr17, wr18, wr19, wr20, wr21, wr22, wr23,
wr24, wr25, wr26, wr27, wr28, wr29,
ir, pr0, pr1, cr, sr0, srl;

NONTERMINALS
(*****)
Const  COND_ATTRIBUTES (v: int;
                       length: int);
DS     ADRMODE          (name: string;
                       length: int);
DSS    ADRMODE          (name: string;
                       length: int;
                       offset: int);

Register REGISTERS
(dr0, dr1, dr2, dr3, dr4, dr5, dr6, dr7,
dr8, dr9, dr10, dr11, dr12, dr13, dr14, dr15,
dr16, dr17, dr18, dr19, dr20, dr21, dr22, dr23,
ar0, ar1, ar2, ar3,
wr0, wr1, wr2, wr3, wr4, wr5, wr6, wr7,
wr8, wr9, wr10, wr11, wr12, wr13, wr14, wr15,
wr16, wr17, wr18, wr19, wr20, wr21, wr22, wr23,
wr24, wr25, wr26, wr27, wr28, wr29,
ir, pr0, pr1, cr, sr0, srl)
COND_ATTRIBUTES (length: int);

(* ----- *)
(* Constant folding is not needed. (Is taken care of by CE.) *)

(* To calculate the length of the constant *)
RULE Constant -> Const;
COST 0;
EVAL {Const.v = Constant.v;
      Const.length = SizeofConst(Constant.v);};

(* Statements ----- *)

RULE Enter;
COST 10;
EMIT {
.> RECEIVE [s Enter.signal]
};

RULE Exit;
COST 10;
EMIT {
.> EP
};

```

```

RULE Assign
  DS
  Register.a;
  COST 10;
  EMIT {
.> WS [s DS.name]-[*a]
};

RULE Assign
  DSS
  Register.a;
  COST 11;
  EMIT {
.> WSS [s DSS.name]/[z DSS.length][i DSS.offset] [*a]
};

RULE Assign
  DS
  Const;
  CONDITION (DS.length <= 8);
  COST 10;
  EMIT {
.> WRC [s DS.name]-[i (Const.v & ((1<<DS.length)-1))]
};

RULE Assign
  DS
  Const;
  CONDITION (Const.length == 0);
  COST 9;
  EMIT {
.> WZ [s DS.name]
};

RULE Assign
  DS
  Const;
  CONDITION (Const.v == ((1<<DS.length)-1));
  COST 9;
  EMIT {
.> WO [s DS.name]
};

RULE Assign
  Str_var
  Content
  Str_var;
  COST 100;
  EMIT {
.Code for copying a string.
};

RULE Assign
  DS.v
  Plus
  Content
  DS.w
  Const;
  CONDITION ({!strcmp(v.name, w.name)) &&
             (Const.length <= 8)};
  COST 9;
  EMIT {
.> AHCS [s v.name]-[i Const.v]
};

RULE Assign
  Temp_var.t
  Register.v ({ContentTempVar(t.name)});
  CONDITION {t.length >= v.length};
  COST 0;

RULE Assign
  Temp_var.t
  Register.v ({ContentTempVar(t.name)});
  COST 10;
  EMIT {
.> NHC [*v]-[i (1<<(t.length))-1]
};

(* Expressions ----- *)

RULE Register.r
-> TempVar.v;
COST 0;
EVAL {v.length = r.length};
TARGET r;

RULE Simple_var.v
-> DS;
COST 0;
EVAL {DS.name = v.name;
      DS.length = v.length;};

RULE Matrix
  DS.a
  Register(pr0)
  Register(ir)
-> DS.b;
COST 0;
EVAL {b.name = a.name;
      b.length = a.length;};

```

```

RULE Sub
  DS
  -> DSS;
  COST 0;
  EVAL {DSS.name = DS.name;
        DSS.length = Sub.length;
        DSS.offset = Sub.offset/Sub.length;};

(* ----- *)

RULE FreeReg
  -> Register;
  COST 0;

RULE Const
  -> Register.a;
  COST 22;
  EVAL {a.length = Const.length;};
  EMIT {
    LHC [*a]/W0-{i (Const.v & {(1<<8)-1})}
    LHC [*a]/B1-{i (Const.v >> 8)}
  };

RULE Const
  -> Register.a;
  CONDITION {Const.length <= 8};
  COST 11;
  EVAL {a.length = Const.length;};
  EMIT {
    LHC [*a]/[z Const.length]0-{i Const.v}
  };

(* Rest of a is cleared *)

RULE Const
  -> Register.a;
  CONDITION {Const.length <= 4};
  COST 10;
  EVAL {a.length = Const.length;};
  EMIT {
    LCC [*a]-[i Const.v]
  }; (* Rest of a is cleared *)

RULE Content
  DS
  -> Register.r;
  COST 10;
  EVAL {r.length = DS.length;};
  EMIT {
    RS [*r]-[s DS.name]
  };

RULE Content
  DSS
  -> Register.r;
  COST 10;
  EVAL {r.length = DSS.length;};
  EMIT {
    RSS [*r]-[s DSS.name]/[z DSS.length]{i DSS.offset}
  };

RULE Content
  Temp_var.t
  -> Register {(ContentTempVar(t.name))};
  COST 0;
  EVAL {Register.length = 16;}; (* Can be smaller! *)

RULE Plus
  Register.a
  Register.b
  -> Register.r;
  COST 9;
  EVAL {r.length = 16;}; (* Can be smaller! *)
  TARGET b;
  EMIT {
    AR [*b]-[*a]
  };

RULE Plus
  Const
  Register.a
  -> Register.r;
  CONDITION {Const.length <= 4};
  COST 8;
  EVAL {r.length = 16;};
  TARGET a;
  EMIT {
    ACC [*a]-[i Const.v]
  };

(* Expensive! Uses five registers. It is easy to write a
routine that
uses three register (Not IR). *)

(* The rule does not work. It does not allocate registers
correctly
CHANGE does not work when LOCALAVAIL is used. *)
(*
RULE Mult
  Register.fak1 (dr0..wr29, pr0..sr1)
  Register.fak2 (dr0..wr29, pr0..sr1)
  Register.temp (dr0..wr29, pr0..sr1)
  -> Register.prod (dr0..wr29, pr0..sr1);
  COST 100;
  EVAL {prod.length = 16;};
  CHANGE {ir};

```

```

  EMIT {
    LCC [*prod]-0
    LCC ir-15
    Label11
    BLO [*fak2],0,Label12
    JLN label13
    Label12
    MFR [*temp]-[*fak1]
    SHL [*temp],0
    AR [*prod]-[*temp]
    JLN label11
    Label13
  };
  *);

(* Five operands are necessary to make the
register allocation work properly. *)

RULE Mult
  Register.fak1 (dr1..wr29)
  Register.fak2 (dr1..wr29)
  Register.temp (dr1..wr29)
  Register.IR (ir)
  Register.prod (dr1..wr29)
  -> Register.a;
  COST 100;
  EVAL {a.length = 16;};
  TARGET prod;
  CHANGE (dr0);
  EMIT {
    LCC [*prod]-0
    LCC ir-15
    Label11
    BLO [*fak2],0,Label12
    JLN label13
    Label12
    MFR [*temp]-[*fak1]
    SHL [*temp],0
    AR [*prod]-[*temp]
    JLN label11
    Label13
  };

(* ----- *)
INSERTS
IpGcgTypes {
  typedef char Operand[256];

  typedef char *string;
}

IpPlex_c {
  RegisterSet globalavail;

  int SizeofConst(int a)
  {
    int i;

    for (i=0; a != 0; i++)
      a = a>>1;
    return(i);
  }

  IpEmit_c {
    #include <string.h>
    #include "GcgTab.h"
    #include "GenOut.h"

    RegisterSet globalavail;

    char TempVarTable[1+NumRegister][20];

    char VarSize[] = "BTCCHHHHHWWWWWWWW";
  }

  IpEmitInit {
    {
      int i;

      for (i=0; i<=NumRegister; i++)
        strcpy(TempVarTable[i],"");
      for (i=0; i<=(NumRegister>>5); i++)
        globalavail[i] = -1;
    }
  }

  IpTestImport {
    void gcg_PrintOperand(Operand o)
    {
      printf("%256s",o);
    }

    void gcg_Printstring(string a)
    {
      printf("%s",a);
    }
  }

  IpEmit {
    /* Control lines for the dottool: */
    ..* GenRegister(%)
    ..i GenInt (%)
    ..= GetLine(%)
    ..s GenString(%)
    ..$ GenLn()
  }

```

```

..% GenString("%")
..^ gcg_PrintRegisterSet(%)
..z GenVarSize(%)
void copop(char *o1,char *o2)
{
    while (*o1++ == *o2++);
}

void GenRegister (Register r)
{
    GenString(gcg_RegNameTable[r]);
}

void GenVarSize (int i)
{
    GenChar(VarSize[i-1]);
}

Register ContentTempVar(char *var)
{
    Register r;

    for (r=0; r<=NumRegister; r++) {
        if (!strcmp(var, TempVarTable[r])) {
            if (gcg_OptFlags.OptEmitMatch)
                printf("*****C<%s>%s\n",var,gcg_RegNameTable[r]);
            return(r);
        }
    }
    if (gcg_OptFlags.OptEmitMatch)
        printf("*****C<%s>%s\n",var,gcg_RegNameTable[0]);
    return(RegNil);
}

Register AllocTempVar(char *var)
{
    int i;
    Register r;

    for (r=1; r<=NumRegister; r++) {
        if (!strcmp("", TempVarTable[r])) {
            strcpy(TempVarTable[r], var);
            for (i=0; i<=(NumRegister>>5); i++)
                globalavail[i] = (globalavail[i] &
                    ~ gcg_RegDestroy[r][i]);
            return(r);
        }
    }
    printf("No registers are unused!");
    exit(0);
}

/*****
/* Routines needed by the Register Allocator */
*****/

void LR (Register to, Register from)
{
    /* Copy Register from into Register to */
    MFR {to}-{from}
}

void Spill (Register reg, Spilloccation loc)
{
    >SPILLING REG: {reg}<
    /*
    exit(0);
    */
}

void Restore (Register reg, Spilloccation loc)
{
    >RESTORE REG: {reg}<
}

IpEmitIl {
    int i;

    for (i=0; i<=(NumRegister>>5); i++)
        localavail[i] = globalavail[i];
}

ipInOut {
    /* This ip is used because there is no IpRegAlloc_c */
    #include <string.h>
}

Register translate_reg(char *str)
{
    char *b,*c;
    Register i;

    b = strdup(str);
    c = strtok(b,"_");
    c = strtok(NULL,".");
    free(b);

    for (i=NumRegister;
        ((i>0) && strcmp(c,gcg_RegNameTable[i]));
        i--)
        return(i);
}

END CODE_GENERATOR_DESCRIPTION Plex.

```

Appendix 3. Code generator specification for the BEG-generated code generator to SPARC machine code using register windows

```

%test
%noIRConsCheck
%noonthefly
%RegNameTable (* ineffective if 'test' is on *)

CODE_GENERATOR_DESCRIPTION CgDice:
(*****
(* SPARC Code Generator Specification in the BEG code
generator specification language, for a Pascal compiler
Initial version was written by Mikael Pettersson,1991.
Most of this is straight-forward, except for function
calls. The problems there is due to the calling
conventions (compatible with those of Sun's compilers),
and the inherent limitations of BEG. Another problem is
that 'double' isn't always double-word aligned, so two
32-bit moves are usually emitted.*/)
(*****

INTERMEDIATE REPRESENTATION
(*****

NONTERMINALS
    IExp, FExp, AExp, VExp;

OPERATORS
(* Constants ----- *)
IntConst (val : int) -> IExp;
RegOff (base : int; off : int) -> IExp;

(* Integer Calculations ----- *)
IntAdd IExp * IExp -> IExp;
IntSub IExp * IExp -> IExp;
IntMul IExp * IExp -> IExp;
IntDiv IExp * IExp -> IExp;
IntMod IExp * IExp -> IExp;
IntCmp (rel : relop) IExp * IExp -> IExp;
IntBitOp (op : intop) IExp * IExp -> IExp;

(* Floating-point Calculations ----- *)
FloatBinOp (op : fpbinop) FExp * FExp -> FExp;
FloatNeg FExp -> FExp;
FloatCmp (rel : relop) FExp * FExp -> IExp;

(* Miscellaneous ----- *)
IntToFloat IExp -> FExp;
FloatToInt FExp -> IExp;
Load (siz : unsigned) IExp -> IExp;
LoadFloat IExp -> FExp;
Store (siz : unsigned) IExp * IExp -> VExp;
StoreFloat IExp * FExp -> VExp;
StoreStruct (siz : unsigned) IExp * IExp -> VExp;
CopyStruct (siz : unsigned) IExp -> IExp;
AsgInt (siz : unsigned; op : intop) IExp * IExp;
AsgFloat (op : fpbinop) IExp * FExp;

(* Function Calls (major kludges here) --- *)
Arg0 IExp -> AExp;
Arg1 IExp -> AExp;
Arg2 IExp -> AExp;
Arg3 IExp -> AExp;
Arg4 IExp -> AExp;
Arg5 IExp -> AExp;
ArgFloat01 FExp -> AExp;
ArgFloat12 FExp -> AExp;
ArgFloat23 FExp -> AExp;
ArgFloat34 FExp -> AExp;
ArgFloat45 FExp -> AExp;
ArgFloat5x FExp -> AExp;
NoArg -> AExp;
VSeq VExp * VExp -> VExp;
VNil -> VExp;
Call (slots: unsigned)
VExp*AExp*AExp*AExp*AExp*AExp*AExp -> IExp;
CallFloat (slots: unsigned)
VExp*AExp*AExp*AExp*AExp*AExp*AExp -> FExp;
CallVoid (slots: unsigned)
VExp*AExp*AExp*AExp*AExp*AExp*AExp;

(* Miscellaneous top-level operators --- *)
ForEff VExp;
JFalse (lab : int) IExp;
Force00 IExp;

REGISTERS
(*****
(* Integer Registers *)
L0, L1, L2, L3, L4, L5, L6, L7, (* Local *)
I0, I1, I2, I3, I4, I5, I6, I7, (* In *)
O0, O1, O2, O3, O4, O5, O6, O7, (* Out *)
G0, G1, G2, G3, G4, G5, G6, G7, (* Global *)
(* Floating-point Registers *)
F0, F1, F2, F3, F4, F5, F6, F7,
F8, F9, F10, F11, F12, F13, F14, F15,
F16, F17, F18, F19, F20, F21, F22, F23,
F24, F25, F26, F27, F28, F29, F30, F31,
(* %o-reg pairs for f.p. parameters *)
O01(O0,O1), O12(O1,O2), O23(O2,O3), O34(O3,O4), O45(O4,O5);

(* %g0-%g5, %g7, %o6-%o7, %i6-%i7 and odd f.p. regs are
unavailable *)

```

```

AVAIL
(L0..L7,I0..I5,O0..O5,G6,O01..O45,F0,F2,F4,F6,F8,F10,F12,F14,
F16,F18,
F20,F22,F24,F26,F28);

NONTERMINALS
(*****)
Ireg   REGISTERS (L0..L7, I0..I5, O0..O5, G6, G0);
(* %g0: always 0 *)
(* %g1..%g5: scratch registers *)
(* %g7: global obj. table (offset +4096)*)
(* %o6: stack pointer %sp *)
(* %o7: outgoing return address *)
(* %i6: frame pointer %fp (caller's %sp)*)
(* %i7: incoming return address *)
Freg   REGISTERS (F0,F2,F4,F6,F8,F10,F12,F14,
F16,F18,F20,F22,F24,F26,F28);
(* N.B. only even pairs are allocatable *)
(* %f30,%f31: scratch registers *)
Oreg   REGISTERS (O01..O45, O5);
Displ  ADRMODE (d : displ);
(* d.okind == ISCON until code generation *)
RegIm  ADRMODE (isreg : int; val : int);
Constant COND_ATTRIBUTES (val : int);
Arg     ADRMODE (* no attributes *) ;
Void    ADRMODE (* no attributes *) ;

(* Chain Rules ----- *)

RULE Ireg -> Displ.di;
COST 0;
{
  EMIT {
    di.d.base = beg_reg[Ireg]; di.d.okind = ISCON; di.d.off = 0;
    AsmComment("ChainRule: Ireg -> Displ");
  };
}

RULE Ireg -> RegIm;
COST 0;
EMIT { RegIm.isreg = TRUE; RegIm.val = Ireg; };

RULE Constant.c -> RegIm;
CONDITION { -4096 <= c.val && c.val <= 4095 };
COST 0;
EMIT { RegIm.isreg = FALSE; RegIm.val = c.val; };

RULE Constant.c -> Displ.di;
COST 0;
EMIT { di.d.base = RegG0; di.d.okind = ISCON; di.d.off = c.val; };

RULE Constant.c -> Ireg(RegG0);
CONDITION { c.val == 0 };

COST 0;

RULE Constant.c -> Ireg;
COST 4;
EMIT {
  AsmComment("Constant -> Ireg");
  doSet(c.val, Ireg);
};

RULE Displ.di -> Ireg;
COST 4;
CHANGE (G1);
EMIT {
  AsmComment("Displ -> Ireg");
  normalize(sdi.d);
  doArith(opADD, di.d.base, di.d.okind, di.d.off, Ireg);
};

(* Constants ----- *)

RULE IntConst -> Constant;
COST 0;
EVAL { Constant.val = IntConst.val; };

RULE RegOff.op -> Displ.di;
COST 0;
EMIT {
  di.d.base = beg_reg[op.base]; di.d.okind = ISCON; di.d.off = op.off;
};

(* Integer Calculations ----- *)

RULE IntAdd Displ.a Constant.b -> Displ.c;
COST 0;
EMIT { c.d.base = a.d.base; c.d.okind = ISCON; c.d.off = a.d.off + b.val; };

RULE IntAdd Ireg.rsl RegIm.ri -> Ireg.rd;
COST 4;
EMIT {
  AsmComment("IntAdd Ireg RegIm -> Ireg");
  doArith(opADD, rsl, ri.isreg, ri.val, rd);
};

RULE IntSub Displ.a Constant.b -> Displ.c;
COST 0;
EMIT { c.d.base = a.d.base; c.d.okind = ISCON; c.d.off = a.d.off - b.val; };

```

```

RULE IntSub Ireg.rsl RegIm.ri -> Ireg.rd;
COST 4;
EMIT {
  AsmComment("IntSub Ireg RegIm -> Ireg");
  doArith(opSUB, rsl, ri.isreg, ri.val, rd);
};

RULE IntMul Ireg.rsl Constant.c -> Ireg.rd;
CONDITION { IsPowerOfTwo(c.val) };
COST 4;
EMIT {
  AsmComment("IntMul Ireg Constant -> Ireg");
  doArith(opSLL, rsl, ISCON, log2(c.val), rd);
};

RULE IntMul Ireg.rl(O0) Ireg.r2(O1) -> Ireg(O0);
COST 10;
TARGET rl;
CHANGE (O4,O5);
EMIT {
  AsmComment("IntMul Ireg Ireg -> Ireg");
  doJmpl(RegG7, ISCON, MULOFF, RegO7);
  doNop();
};

RULE IntDiv Ireg.rsl Constant.c -> Ireg.rd;
CONDITION { IsPowerOfTwo(c.val) };
COST 8;
CHANGE (G1);
EMIT {
  AsmComment("IntDiv Ireg Constant -> Ireg");
  /* For some reason, the sign bit must be added to the low
  shcnt bits. Both Sun's CC and GNU's CC do this. */
  int shcnt = log2(c.val);
  doArith(opSRA, rsl, ISCON, shcnt-1, RegG1);
  doArith(opSRL, RegG1, ISCON, 32-shcnt, RegG1);
  doArith(opADD, RegG1, ISREG, rsl, RegG1);
  doArith(opSRA, RegG1, ISCON, shcnt, rd);
};

RULE IntDiv Ireg.rl(O0) Ireg.r2(O1) -> Ireg(O0);
COST 10;
TARGET rl;
CHANGE (G1..G3,O2..O5);
EMIT {
  AsmComment("IntDiv Ireg Ireg -> Ireg");
  doJmpl(RegG7, ISCON, DIVOFF, RegO7);
  doNop();
};

RULE IntMod Ireg.rl(O0) Ireg.r2(O1) -> Ireg(O0);
COST 10;
TARGET rl;
CHANGE (G1..G3,O2..O5);
EMIT {
  AsmComment("IntMod Ireg Ireg -> Ireg");
  doJmpl(RegG7, ISCON, REMOFF, RegO7);
  doNop();
};

RULE IntCmp.op Ireg.rsl RegIm.ri -> Ireg.rd;
COST 4;
EMIT {
  AsmComment("IntCmp Ireg RegIm -> Ireg");
  doArith(opSUBCC, rsl, ri.isreg, ri.val, RegG0);
  label = getlab();
  doBranch(Bicc(invrel(op.rel)), ANNUL, label);
  doArith(opOR, RegG0, ISCON, 0, rd);
  doArith(opOR, RegG0, ISCON, 1, rd);
  deflab(label);
};

RULE IntBitOp Ireg.rsl RegIm.ri -> Ireg.rd;
COST 4;
EMIT {
  AsmComment("IntBitOp Ireg RegIm -> Ireg");
  doArith(IntBitOp.op, rsl, ri.isreg, ri.val, rd);
};

(* Floating-point Calculations ----- *)

RULE FloatBinOp Freg.a Freg.b -> Freg.c;
COST 6;
EMIT {
  AsmComment("FloatBinOp Freg Freg -> Freg");
  doFpOp3(FloatBinOp.op, a, b, c);
};

RULE FloatNeg Freg.src -> Freg.dst;
COST 12;
EMIT {
  AsmComment("FloatNeg Freg -> Freg");
  doFpOp2(opFNEGS, src, dst);
  if (src != dst)
    doFpOp2(opFMOVs, src+1, dst+1);
};

RULE FloatCmp.op Freg.a Freg.b -> Ireg.rd;
COST 10;
EMIT {
  AsmComment("FloatCmp Freg Freg -> Ireg");
  doFcmpd(a, b);
  doBranch(FBfcc(invrel(op.rel)), ANNUL, 3);
  doArith(opOR, RegG0, ISCON, 0, rd);
  doArith(opOR, RegG0, ISCON, 1, rd);
};

```

```

};

(* Miscellaneous ----- *)

(* By recognizing (IntToFloat (Load <displ>)) [i.e. (double)i]
we can avoid one pair of load/store operations. *)
RULE IntToFloat Load Displ di -> Freg;
CONDITION { Load.siz == 4 };
COST 6;
CHANGE (G1);
EMIT {
  AsmComment("IntToFloat Load Displ -> Freg");
  normalize(&di.d);
  doLD(opLDF, di.d.base, di.d.okind, di.d.off, Freg);
  doFpOp2(opFITOD, Freg, Freg);
};

RULE IntToFloat Ireg -> Freg;
COST 8;
EMIT {
  AsmComment("IntToFloat Ireg -> Freg");
  doST(opST, Ireg, RegSP, ISCON, 64);
  doLD(opLDF, RegSP, ISCON, 64, Freg);
  doFpOp2(opFITOD, Freg, Freg);
};

RULE FloatToInt Freg -> Ireg;
COST 8;
CHANGE (F31);
EMIT {
  AsmComment("FloatToInt Freg -> Ireg");
  doFpOp2(opFDTOI, Freg, RegF31);
  doST(opSTF, RegF31, RegSP, ISCON, 64);
  doLD(opLD, RegSP, ISCON, 64, Ireg);
};

RULE Load Displ -> Ireg;
COST 4;
CHANGE (G1);
EMIT {
  AsmComment("Load Displ -> Ireg");
  doLoad(Load.siz, &Displ.d, Ireg);
};

RULE LoadFloat Displ -> Freg;
COST 6;
CHANGE (G1);
EMIT {
  AsmComment("LoadFloat Displ -> Freg");
  doLoadFloat(&Displ.d, Freg);
};

(* By recognizing (Store <displ> (FloatToInt <freg>)) [i.e.
i=(int)d]
we can avoid one pair of load/store operations. *)
RULE Store Displ di FloatToInt Freg -> Void;
CONDITION { Store.siz == 4 };
COST 6;
CHANGE (F31,G1);
EMIT {
  AsmComment("Store Displ FloatToInt Freg -> Void");
  doFpOp2(opFDTOI, Freg, RegF31);
  normalize(&di.d);
  doST(opSTF, RegF31, di.d.base, di.d.okind, di.d.off);
};

RULE Store Displ Ireg -> Void;
COST 4;
CHANGE (G1);
EMIT {
  AsmComment("Store Displ Ireg");
  doStore(Store.siz, Ireg, &Displ.d);
};

RULE StoreFloat Displ Freg -> Void;
COST 6;
CHANGE (G1);
EMIT {
  AsmComment("Store Displ Freg");
  doStoreFloat(Freg, &Displ.d);
};

RULE StoreStruct.op Ireg.dst Ireg.src -> Void;
COST 10;
CHANGE (G1..G4);
EMIT {
  AsmComment("StoreStruct Ireg Ireg");
  doCopy(op.siz, src, dst);
};

RULE CopyStruct.op Ireg.src -> Ireg.dst;
COST 10;
CHANGE (G1..G5);
EMIT {
  AsmComment("CopyStruct Ireg -> Ireg");
  {
    int off; Register src1;

    if( src == dst ) {
      src1 = RegG5;
      doArith(opOR, RegG0, ISREG, src, src1);
    } else
      src1 = src;
    if( (off = doAlloca(op.siz, 7)) >= -4096 )
      doArith(opSUB, RegFP, ISCON, off, dst);
    else {
      doSet(off, RegG1);
      doArith(opSUB, RegFP, ISREG, RegG1, dst);
    }
    doCopy(op.siz, src1, dst);
  };
};

RULE AsgInt Displ RegIm;
COST 0;
CHANGE (G1,G2);
EMIT {
  AsmComment("AsgInt Displ RegIm");
  doLoad(AsgInt.siz, &Displ.d, RegG2); /* implicit
  normalize(&Displ.d) */
  doArith(AsgInt.op, RegG2, RegIm.isreg, RegIm.val, RegG2);
  doStore(AsgInt.siz, RegG2, &Displ.d);
};

RULE AsgFloat Displ Freg;
COST 0;
CHANGE (G1,F30);
EMIT {
  AsmComment("AsgFloat Displ Freg");
  doLoadFloat(&Displ.d, RegF30); /* implicit
  normalize(&Displ.d) */
  doFpOp3(AsgFloat.op, RegF30, Freg, RegF30);
  doStoreFloat(RegF30, &Displ.d);
};

(* Function Calls ----- *)

RULE Arg0 Ireg(O0) -> Arg;
COST 0;

RULE Arg1 Ireg(O1) -> Arg;
COST 0;

RULE Arg2 Ireg(O2) -> Arg;
COST 0;

RULE Arg3 Ireg(O3) -> Arg;
COST 0;

RULE Arg4 Ireg(O4) -> Arg;
COST 0;

RULE Arg5 Ireg(O5) -> Arg;
COST 0;

RULE ArgFloat01 Oreg(O01) -> Arg;
COST 0;

RULE ArgFloat12 Oreg(O12) -> Arg;
COST 0;

RULE ArgFloat23 Oreg(O23) -> Arg;
COST 0;

RULE ArgFloat34 Oreg(O34) -> Arg;
COST 0;

RULE ArgFloat45 Oreg(O45) -> Arg;
COST 0;

RULE ArgFloat5x Oreg(O5) -> Arg;
COST 0;

RULE LoadFloat Displ di -> Oreg;
COST 8;
CHANGE (G1);
EMIT {
  AsmComment("LoadFloat Displ -> Oreg");
  normalize(&di.d);
  doLD(opLD, di.d.base, ISCON, di.d.off, Oreg);
  if( Oreg == RegO5 ) {
    doLD(opLD, di.d.base, ISCON, di.d.off+4, RegG1);
    doST(opST, RegG1, RegSP, ISCON, 68+6*4);
  } else
    doLD(opLD, di.d.base, ISCON, di.d.off+4, Oreg+1);
};

RULE Freg -> Oreg;
COST 12;
EMIT {
  AsmComment("Freg -> Oreg");
  if( Oreg == RegO5 ) {
    doST(opSTDF, Freg, RegSP, ISCON, 68+5*4);
    doLD(opLD, RegSP, ISCON, 68+5*4, RegO5);
  } else {
    doST(opSTDF, Freg, RegSP, ISCON, 64);
    if( Oreg == RegO12 || Oreg == RegO34 ) {
      doLD(opLD, RegSP, ISCON, 64, Oreg);
      doLD(opLD, RegSP, ISCON, 68, Oreg+1);
    } else
      doLD(opLDD, RegSP, ISCON, 64, Oreg);
  };
};

RULE NoArg -> Arg;
COST 0;

RULE VSeq Void Void -> Void;
COST 0;

```

```

RULE VNil -> Void;
COST 0;

RULE Call Void Arg Arg Arg Arg Arg Arg Displ -> Ireg(00);
COST 10;
CHANGE (G1..G6,F0..F31);
EMIT {
  AsmComment("Call ... -> Ireg");
  doCall(&Displ.d, Call.slots);
};

RULE CallFloat Void Arg Arg Arg Arg Arg Arg Displ -> Freg(F0);
COST 10;
CHANGE (G1..G6,F1..F31);
EMIT {
  AsmComment("CallFloat ... -> Freg");
  doCall(&Displ.d, CallFloat.slots);
};

RULE CallVoid Void Arg Arg Arg Arg Arg Arg Displ;
COST 10;
CHANGE (G1..G6,F0..F31);
EMIT {
  AsmComment("CallVoid ...");
  doCall(&Displ.d, CallVoid.slots);
};

(* Miscellaneous Top-Level Operators -- *)

RULE ForEff Void; (* for top-level assignments *)
COST 0;

RULE JFalse IntCmp Ireg.rsl RegIm.ri;
COST 4;
EMIT {
  AsmComment("JFalse IntCmp Ireg RegIm");
  doArith(opSUBCC, rsl, ri.isreg, ri.val, RegG0);

```

```

  doBranch(Bicc(invrel(IntCmp.rel)), NOANNUL, JFalse.lab);
  doNop();
};

RULE JFalse FloatCmp Freg.a Freg.b;
COST 4;
EMIT {
  AsmComment("JFalse FloatCmp Freg Freg");
  doFcmpd(a, b);
  doBranch(FBfcc(invrel(FloatCmp.rel)), NOANNUL, JFalse.lab);
  doNop();
};

RULE JFalse Ireg;
COST 6;
EMIT {
  AsmComment("JFalse Ireg");
  doArith(opSUBCC, Ireg, ISREG, RegG0, RegG0);
  doBranch(Bicc(releQ), NOANNUL, JFalse.lab);
  doNop();
};

RULE Force00 Ireg(I0);
COST 0;

INSERTS
.....
/* 12 pages of low-level binary code emitting routines
 * called by the code
 * generator.
 */
.....

END CODE_GENERATOR_DESCRIPTION CgDice.

```