Optimal Code-Selection using MBURG

K John Gough
Faculty of Information Technology
Queensland University of Technology
Brisbane, Australia

j.gough@qut.edu.au

Jeff Ledermann
Faculty of Information Technology
Queensland University of Technology
Brisbane, Australia

lederman@dstc.qut.edu.au

Abstract

Bottom-up tree rewriting has become the technique of choice for code selection in compilers. The technique is based on dynamic programming, and can select optimal code on trees. Several tools exist which are able to automatically produce bottom up tree rewriters from grammar specifications. We describe one such tool, MBURG, and introduce two innovations: incremental labelling, and forced reductions.

Early experience with MBURG shows that it can produce code superior to that from carefully hand-tuned stack automata, by selecting optimal address modes for its emitted instructions. It is shown that the technique can be generalised to recognise tree idioms and perform some surprising code improvements. **Keywords:** *code-generation, compilers, tree rewriting.*

1 Introduction

Since the 1970s, there has been a considerable amount of research devoted to seeking ways to automate the production of programming language compilers, based on declarative specifications. The production of high quality scanners and parsers has been commonplace for many years, with results rivalling the very best hand crafted programs[15]. Other phases of compilation have proved much more resistant to automation, and it is only recently that static semantic analysers based on attribute grammars have become competitive with hand-written analysers[14].

Code selection has had a somewhat chequered history. Code selection is the phase of code generation in which a sequence of instructions is chosen to compute the value of each expression in a program. The task is problematic, because for typical machine instruction sets there are an extremely large number of different instruction sequences which compute the same value, even for quite simple expression trees. The situation is particularly difficult with the so-called complex instruction set computers, in which quite large subexpressions can sometimes be "folded" into the address modes of the machine architecture.

Submitted to the 20th Australasian Computer Science Conference, Sydney, Australia,

Perhaps the first method of automatic code-selector generation to show promise as an alternative for production compilers, was the Graham-Glanville method[6] of the early-1980s. This method is based on treating the postfix representation of the underlying trees as sentences in a phrase-structured language. Patterns are recognised using a bottom-up shift-reduce parser derived from a disambiguation of an LR(0) grammar. The LR(0) grammar is derived automatically from a declarative specification of the machine instruction set. The main effort in constructing such a code selector is in ensuring that the disambiguation rules cannot lead to "blocking" of the parse at some later stage, which would necessitate a backtracking algorithm.

Hewlett-Packard was one company which used a Graham-Glanville code selector for one of their compilers in the late 1980s. However, they later abandoned this technology, in favour of the UCode-based code selectors of their other compilers.

Another method which found some favour, and which is at least semi-automatic, was devised by Davidson and Fraser[2]. In this method, trees are initially macro expanded into naive and inefficient code, which pays no regard to the context of various expression components. Following this the code is repeatedly improved by a kind of peephole optimiser, which performs pattern matching based on a set of declaratively specified equivalences. This method essentially underlies the code selection of the widely used `gcc` compilers.

Code selection by bottom-up tree rewriting has become the method of choice only within the last few years. In essence the method uses dynamic programming to select optimal code sequences to cover expression trees. The dynamic programming optimises with respect to specified costs for each tree-to-tree reduction rule in a tree grammar. Curiously, the theory which justifies the method was known as long ago as 1976[1], but was treated as being of theoretical interest only. The revival of the method has come about as a result of a long sequence of advances, which have shown that it is possible to perform the dynamic programming at tool construction time, rather than at the runtime of the compiler into which it is incorporated. As it turns out, it is possible to reduce the costs of the dynamic programming to the point where it is feasible to perform it at compiler-runtime anyway.

2 Bottom-up tree rewriting

2.1 A motivating example

Consider the task of generating code for the language *C* expression $(f+i)$, where f is of type `float`, and i is of type `int`. We shall assume that the two operands are in memory, and that the target is the *Intel* iap$X$86 instruction set. All floating point results in the *Intel* architecture are constructed in the coprocessor in an internal format which we shall call *TReg*. Figure 1 shows the abstract syntax tree

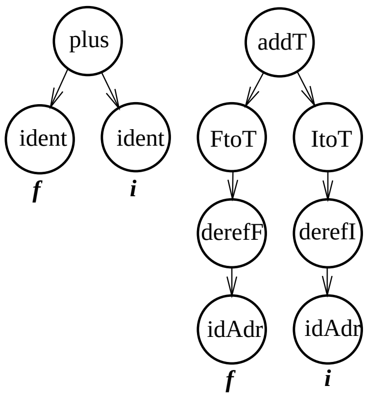

Figure 1: *abstract syntax tree (left) and code tree (right) for the expression* $(f+i)$

and code tree associated with this expression. Notice that in the code tree, on the right of the figure, the dereference and format conversion operations which are implicit in the language *C* syntax, are explicit.

There are at least three ways in which even this trivial expression may be encoded using the instruction set of the *Intel* coprocessor. The two expressions might each be loaded and converted into the temporary format separately, and then the floating point addition would take its two operands from registers of the coprocessor. Alternatively, just one of the operands could be loaded and converted, with the floating point addition taking its second operand directly from memory, and performing the loading and conversion as part of the addition. The tree rewritings of this example tree are shown in Figure 2. On any particular implementation of the *Intel* architecture, these three coverings of the tree will require a different number of processor cycles to complete. For example, for

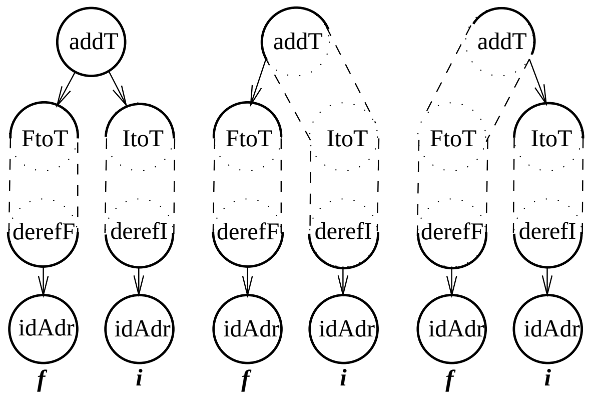

Figure 2: *three rewritings of the code tree in Figure 1, for Intel-iapX87*

the 387 coprocessor the fastest sequence is to load and convert the integer operand, but to use the version of the floating point add instruction which takes its second operand from memory, and does the conversion from single precision to the internal format as part of the operation. This is the rightmost rewriting in Figure 2.

In principle, we would like to be able to find optimal coverings for trees like that in Figure 1, based on rules such as —

- integer values in memory may be converted into the internal floating point form, and loaded into registers
- floating point values in memory may be converted into the internal format and loaded into registers
- floating point add may take both of its operands from registers
- floating point add may take one operand from a register, converting and loading a second, floating point operand from memory
- floating point add may take one operand from a register, converting and loading a second, integer operand from memory

together with an associated cost for the use of each rule. Bottom up tree rewriting provides such a rule-driven basis.

2.2 Labelling and reducing

We begin by defining a number of *TypeForms* in which data values may be realised. In the simplest examples, these forms will have names such as *Imm* (immediate, literal value), *IReg* (integer value in a register) *TReg* (floating point value in a register, in the internal temporary form) and *Adrs* (address of a value in memory). These typeforms will be the nonterminal symbols of a tree grammar. The operation-labels of the expression trees which are to be rewritten, will be the terminal symbols of the grammar. The productions of the grammar will have a non-terminal symbol on the left, and a fully-parenthesised tree expression on the right. The productions corresponding to the rules in the itemlist above would be stated thus —

$\mathit{TReg} \rightarrow$ FtoT(derefF(Adrs))
$\mathit{TReg} \rightarrow$ ItoT(derefI(Adrs))
$\mathit{TReg} \rightarrow$ addT($\mathit{TReg}, \mathit{TReg}$)
$\mathit{TReg} \rightarrow$ addT(TReg,FtoT(derefF(Adrs)))
$\mathit{TReg} \rightarrow$ addT(TReg,ItoT(derefI(Adrs)))

where non-terminals are shown in the italic font, and terminal symbols in the roman font.

It is possible to think of each of these productions as defining a tree-pattern template which is used to tile complete trees. For example, the last of these productions, corresponds to the template in Figure 3.

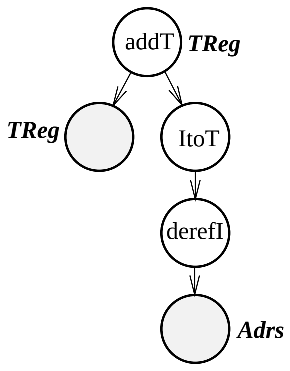

Figure 3: *tree template corresponging to production* $\mathit{TReg} \rightarrow$ addT(TReg,ItoT(derefI(Adrs)))

In this figure, the shaded leaf nodes of the template will in general be the roots of sub-trees. We make no stipulation about the form of these subtrees, except that they must be reducible to the *TReg* or *Adrs* typeform respectively. In the case of the *Intel* architecture the address typeform has a quite complex grammar not considered here. The tree-leaf terminal "idAdr" nodes in Figure 2 are just the simplest of such trees. The label on the root of the template declares that this production creates a value of the *TReg* typeform.

In the particular instance of bottom-up rewriting which the tool *MBURG* implements each production may be guarded by an optional predicate, and has an associated reduction cost which is a literal constant. This makes it possible to state such rules as "this machine instruction can take an immediate operand, but only if the literal fits into a signed 13-bit field." Other formulations have costs which are expressions evaluated at pattern matching time, and use conditional expressions to achieve the same effect as our guards.

The algorithm attaches a state vector to each node of the tree. This state consists of an array indexed by typeform, with each element containing two ordinals: the minimal cost of computing the value denoted by the subtree into that form, and the production which is used to compute that minimal cost. In the case of impossible typeforms, the production ordinal is set to some distinguished error value.

Matching requires two passes over each code tree. The first is the *labelling pass* which computes the state vectors during a bottom-up traversal of the tree. At each node, if a pattern is matched, the cost of the match is found by adding together the minimal cost of computing the subtrees into the required typeform, plus the cost of the production itself. If this cost is lower than the previously best cost of computing the value into that typeform, then the corresponding element of the state is updated.

The second pass over the tree is the *reduction pass*. The traversal proceeds top-down. At each step of the traversal, the node-visit has a typeform as argument. At the root of the tree, this argument is the goal-symbol, and the production which computes the goal symbol at minimal cost is read from the state vector. This production will *demand* that the leaves of its template be in particular typeforms. The recursion will then proceed to the node corresponding to the leaves of the template, with the demanded typeform as argument. During the reduction pass, semantic actions associated with each production are performed. These actions include the emission of code, but also typically include the evaluation of attributes which are used in the emission of code further up the same tree as the recursion unwinds.

Note that the labelling pass recurses over the *structure* of the tree. In constrast, the reduction pass is controlled by the template patterns of the selected instructions, recursing from pattern root-nodes directly to pattern leaf-nodes. The reduction traversal visits the nodes of the (virtual) rewritten tree, skipping any intermediate nodes which have been coalesed into single nodes as shown in the examples in Figure 2. Since in general it is the execution of the semantic actions which is the desired outcome, it is unusual to actually create a rewritten tree.

In the case of *chain productions*, which have the form —

$$\mathit{NonTerm} \rightarrow \mathit{NonTerm}$$

the same node is visited repeatedly. Consider the example of a compare instruction which is supposed to yield a Boolean value in a register. This might arise with the tree fragment shown in Figure 4. In this example *Stmt* is the goal symbol of the grammar.

For machines which do not have instructions which load condition code flags into registers, the integer-less-than comparison *intLT* will have to first yield a value in a condition code, and then

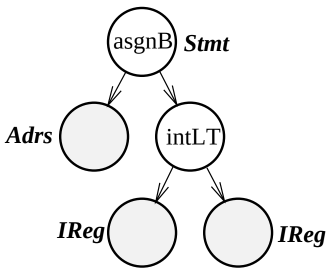

Figure 4: *integer less-than compare and assignment*

the condition code will need to be transformed expensively into a Boolean value using a test and branch instruction. The three relevant productions might be —

$$\begin{aligned}&\mathit{Stmt} \rightarrow \text{asgnB}(\mathit{Adrs},\mathit{BReg})\\&\mathit{BReg} \rightarrow \mathit{Flag}\\&\mathit{Flag} \rightarrow \text{intLT}(\mathit{IReg},\mathit{IReg})\end{aligned}$$

where *BReg* is a Boolean value in an integer register, *Flag* is a Boolean in a condition code register, and "asgnB" is the Boolean asignment terminal symbol. During reduction, the first production will demand the *BReg* typeform from the comparison node. This visit will select the chain production, which will force a second visit to the same node, this time with *Flag* as the demanded typeform.

2.3 Emitting code

The reduction pass proceeds top down, as described. Nevertheless, the semantic actions associated with each reduction are executed *after* the return of the recursive calls to the subtrees rooted at the pattern leaves. The reason for this apparent paradox, is that the semantic actions compute synthesised attributes, which the actions further up the tree require for their own semantic actions. An example will make this clear.

Consider the production —

$$\mathit{IReg} \rightarrow \text{addI}(\mathit{IReg},\mathit{IReg})$$

which adds together two values in integer registers. The semantic action will be to emit a register-to-register addition instruction. However, before that can be done, the code which computes the two subexpressions must be emitted.

The reduction traversal will visit the root node which selects this production, with a demanded typeform of *IReg*. The production will cause the two child nodes to be visited, also with a demanded typeform of *IReg*. The semantic actions at the two child nodes will emit the instructions which load the two subexpressions into two registers, and will annotate the state of the two child nodes with the selected register identifier ordinals. Thus, when the recursion returns and the addition instruction is emitted, the identity of the two operand registers will be already known.

This separation of the pattern matching and the semantic actions into the two separate passes has an unfortunate consequence. A plausible constant folding production such as —

$$\mathit{Imm} \rightarrow \text{addI}(\mathit{Imm},\mathit{Imm})$$

with a semantic action which adds together the two immediate values, will simply not work as expected. If a production further up the tree requires a leaf node of typeform *Imm* and has a guard which tests the value-range of the immediate, then the rewriter will fail. The problem is that the semantic action which computes the folded immediate value does not get executed until the reduction pass, but the folded value is required for pattern matching during the labelling pass.

An exception to this general observation occurs in the case of leaf productions, which are of the form —

$$\mathit{NonTerm} \rightarrow \text{terminal}$$

where "terminal" is a terminal symbol of the grammar with arity equal to zero. In this case, there is no need to perform any pattern matching, since this pattern will always be matched. Therefore, it makes sense to perform both the (trivial) pattern matching, and the semantic action during the labelling traversal. This is a helpful choice, since it allows the semantic action to immediately evaluate attributes of the leaf nodes, which may then be used in predicates which guard productions used further up the tree.

2.4 Code selector generators

There are several tools available which produce tree rewriters from tree-grammar specifications. *BURG*[5] uses a clever method to reduce the compiler runtime computation by encoding the unbounded number of tree configurations during labelling into a finite set of equivalence classes. The pattern matchers generated by this method may use as little as 50 machine cycles per match. The price which is paid is the loss of flexibility, as the grammars may have neither guarded productions, nor non-constant cost functions. Proebsting's thesis[12] gives a comprehensive bibliography of all of the theory which lies behind this tool.

IBURG[4], and the closely related tool `lburg`, used by `lcc`[3], both perform their dynamic programming at compiler runtime. They accept conditional expressions as production costs, and produce their matchers in ANSI C. The output consists of the source code of the labelling pass, and a skeleton of the reduction pass into which the semantic actions must be editted.

We created *MBURG*[9] for several reasons. We wanted to experiment with the algorithm itself, and to produce ISO Modula-2, rather than C as output.

We also wished to implement more comprehensive checks on the grammar, than is the case for the existing tools. Finally, we wanted to try to create a tool in which the complete rewriter was created from the specification, without any need to include semantic actions by hand. The tool has evolved over several versions in the last year, and now incorporates the innovations discussed below.

MBURG is freely available, and both the tool and the reference manual are available by ftp[7].

We have used *MBURG* to create a number of experimental code selectors for gardens point compilers[13], and have incorporated such a selector into our production compilers for *SUN SPARC/Solaris*. Previous versions of these compilers used shadow stack automata, which match patterns in the abstract stack machine-based form which all of our compilers use.[8]

For our most complex grammars, there are as many as 20 typeforms, so the amount of memory used by the state vectors is significant. However, we only construct one code-tree at a time, incoporating the state inline in the tree nodes. The total space of each tree is reclaimed after reduction has completed on that tree. In the case that the code-tree is to be retained, *MBURG* allows the state vector space to be dynamically allocated, and reclaimed separately.

3 Creating the code tree

One of the advantages of using tools which are based on specifications, is that specifications can be composed, to produce new tools. In particular, it is desirable to use the same code selection grammar for the implementation of multiple languages for the same target machine architecture. The goal is therefore to create a language-independent code selector, which rewrites trees which are produced by target-independent language processing front-ends.

This factorisation of tasks is the argument which has been used in the past to justify the use of language and machine independent intermediate representations (*IRs*). From this perspective, it is clear that the code-tree itself is the machine and language independent representation. This observation has some rather non-obvious consequences.

Firstly, it might be thought that the annotated abstract syntax trees produced by the language processing front-end might be a suitable subject for tree rewriting. Indeed this is one view of the way in which `lcc`'s tree-rewriting code selectors work. However, this is not a general solution, since in that case `lcc`'s code selectors must understand semantic details of the ANSI-C language. If a language independent code selector is to be used, then the semantic fine-print of the language must be already explicit in the code tree.

Consider a simple example. The modulus operation *MOD* is subtly different in the closely related languages Modula-2 and Modula-3. In the former language, the value is only defined for positive values of the right operand. Thus the same abstract syntax tree in the two languages must produce code which in the case of one language requires a range-test and trap. Either the code selector must be parameterised for subject language — a most undesirable design feature — or the code tree must encode the range test explicitly.

Thus in general it is necessary to perform a tree-to-tree transformation, starting with the annotated abstract syntax tree, and producing the language independent code tree. For some languages it may be possible to incrementally transform one tree into the other, by adding extra nodes as required. However, it is probably more systematic (and hence more amenable to automation) to build an entirely new tree, as a synthesised attribute of the original abstract syntax tree.

In the case of the gardens point compilers, our front-ends produce the code for an abstract stack machine, which we call the "D-Machine"[8]. This textual form is parsed, and a code tree is recreated from the stack code. Going through the textual form is certainly inefficient, but we believe that a procedural interface between the (abstract syntax) tree-walker and the (abstract code) tree builder would be as efficient as any other algorithm for tree-to-tree transformation. Furthermore, as shown in the next section, rebuilding the tree bottom-up can completely eliminate the labelling traversal of the code tree.

4 Incremental labelling

When a code tree is built bottom up, first the subtrees are created, then parent nodes are created and the child nodes linked to them. The order of node creation is thus exactly the order in which the labelling traversal will visit the nodes of the tree.

This suggests a clever variation of the labelling algorithm. Usually labelling starts from the root of each tree, and performs a recursive traversal to evaluate the synthesised attributes of the state vectors. The alternative is to label each node as it is created. The labelling becomes a *non-recursive* computation, because when each node is created, it is known that the state vectors of its children have already been computed.

Thus, in the case where the code tree is built strictly bottom-up, the nodes may be labelled incrementally, and one complete recursive traversal of the tree avoided.

It should be admitted that this saving somewhat smacks of "making a virtue of necessity", since our IR forces us to create the code tree bottom-up. However, since we hypothesise that the bottom-up creation of a code tree from an abstract syntax tree is as efficient as any other method, we believe that the gain is a real one.

MBURG can produce either recursive or non-recursive labellers, for the same grammars. A declaration in the grammar file selects either incremental or recursive labelling. The grammars are otherwise unchanged by the choice.

5 Forced reductions

Section 2.3 pointed out a limitation on the use of bottom-up rewriting to effect constant folding. This was initially of little concern, given that it is a simple matter for language processing front-ends to fold such constants during static semantic attribute evaluation. Later however, the limitation posed by not being able to use the result of semantic actions during pattern matching became an irritant.

Our code generators, aggressively try to move scalar local variables to registers, and create a table of the frame offsets of all of those variables which have been so promoted. At every assignment and dereference node which involves a frame-pointer address, the table is consulted to see if the variable is actually in memory or instead in a register. We assumed that front-ends would never perform computations which would compute the frame offset of a register variable at a non-leaf node.

Typical grammars had productions such as —

$$FpAdr \rightarrow \text{addI}(FpAdr, Imm)$$

where *FpAdr* are frame-pointer relative addresses in the stack frame. This production would be exercised whenever a field of a local structure is accessed, effectively folding the base-address plus literal offset expression at compile time. We did not think that any such computation would compute the addresses of register variables. Unfortunately, we had not specified this as a limitation of the intermediate form, and we found that other front-end authors did not recognise this limitation.

This is a serious problem, since we need to know the folded offset attribute to determine whether or not an *FpAdr* actually refers to the stack frame or to a register variable. It is also a variant of the constant folding problem discussed above, so we sought a uniform solution.

The solution which we have implemented allows the grammar to declare that certain patterns should be preemtively reduced as soon as they are recognised. We have applied this facility to fold constants, and global and local invariant address expressions. Forced, preemptive reductions are a potentially dangerous facility, since it would be possible to force the rewriter down a dead-end path. However, used for the kind of attribute folding that we have suggested, we think that it is a useful extension to praxis.

6 How MBURG works

MBURG produces two modules as output. One, *FormDefs*, defines the datatypes which are required, and the other, *Match* has the code of the selector. The process of incorporating an *MBURG*-created code-selector into a larger program is shown diagramatically in Figure 5.

The rewriters which *MBURG* creates work in a rather different way to those created by *IBURG*, even in the case of non-incremental labelling. *MBURG* creates two arrays of procedure variables (pointers to functions in *C*). The first is an array of labelling procedures, *match*, which is indexed on the node terminal symbol type.

```
VAR match:ARRAY Tag OF PROCEDURE(Tree);
```

where *Tag* is the terminal symbol type.

Thus when a "derefF" node is encountered during a labelling traversal, the *match* array element `match[derefF]` will be dispatched, with the current node as argument. The dispatched procedure `match[derefF]` in this case will be the procedure with declaration

```
PROCEDURE derefFMatch(self :  Tree)
```

and will be hard-wired to recognise all of the patterns which are rooted at "derefF" nodes. The match procedures will be recursive or non-recursive according to whether the incremental declaration is in force. In the recursive case, the procedure will dispatch another element of the same *match* array on each child node, by calling —

```
match[child^.tag](child);
```

where *tag* is the terminal symbol tag value of the code-tree nodes.

The second array of procedures is called *rHelp*, and is indexed on production ordinal. As each node is visited during the reduction traversal, the reduction helper appropriate to that production is called. In the case of the production —

$$IReg \rightarrow \text{addI}(IReg, IReg)$$

the helper would be as follows —

```
PROCEDURE ReduceNN(self :  Tree);
  VAR leaf1,leaf2 : Tree;
BEGIN (* IReg → addI(IReg,Ireg) *)
  leaf1 := self^.lOp;
  rHelp[leaf1^.state.rule[IReg]](leaf1);
  leaf2 := self^.rOp;
  rHelp[leaf2^.state.rule[IReg]](leaf2);
  rHelp[prod2](leaf1);
  -- now do semantic actions for this production
END ReduceNN;
```

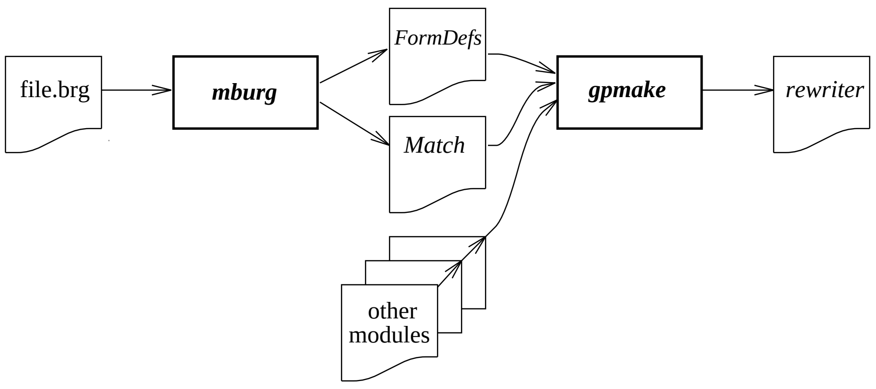

Figure 5: *incorporating an MBURG-generated code selector into a compiler*

In this case, it has been assumed that the selected rule may be accessed from the *rule* array in the node state vector, and that the right and left children of the node are accessed by fields *rOp,lOp*.

Initial measurements showed that this particular implementation of the algorithm, using procedure dispatch, was faster than an alternative which used large case statements selected on node-tag or production ordinal. However, the code size is somewhat larger for the dispatched version than for the case-statement version, probably as a result of code repetition in the bodies of the large number of procedures.

7 Why rewriting works

Bottom-up tree rewriting produces optimal code from trees, based on the given costs of each production. The equivalent problem on directed graphs, is known to be NP-complete.

The power of the rewriting method stems directly from the use of dynamic programming. The problem with other technologies is that they force the code selector to make a committment too early. With bottom-up rewriting, a choice of reduction is not made until every single possible use of the value higher up the tree has been explored.

Two examples should suffice. Many machines provide compare instructions which can either set condition code flags, or can place a Boolean value in a register. As discussed earlier, the conversion of a condition code into a Boolean in a register will typically take several instructions. With bottom-up rewriting, both possibilites are considered. When labelling reaches the root of the tree the created value will have been used in one or other way, automatically selecting the production which produces the typeform which is actually used, for the least possible cost.

The second example arises in the case of machines with complex addressing modes, particularly those with *based, indexed, scaled* address modes. In such machines scaling of indexes by some small powers of two (scaling by 2 or 4, in the *Intel* case) can be folded into an address mode. The problem is that when a multiplication by two or four is encountered, it is unknown whether the result is required as part of an address expression, or will actually need to be realised in an integer register.

With hand-written code-selection, it is necessary to program each such deferral of code emission separately, creating a new data type to hold each new category of symbolic value. With bottom up rewriting, *every* decision is deferred, until the root of the tree has been reached. In this case no symbolic data types are needed, since the tree itself is retained until all decisions have been made. Instead, the housekeeping information which is required to create the intermediate data values from the tree is held in the state vectors of the nodes.

8 Recognising language idioms

Most uses of tools of these kinds have stressed the optimal way in which the technique matches the instruction sets of the target architecture. We have discovered however, that there is some benefit in considering longer-range patterns which occur as a result of idioms in particular programming languages. It is not a question of building in some language dependence, but rather of recognising patterns which are frequently occurring in practice. In this context, the recognition of longer-range patterns may be thought of as a more powerful alternative to the usual kind of peephole optimiser.

We give two illustrative examples of patterns which we have built into our *SPARC* backend. One is an idiom from Modula-2, the other from ANSI-C.

8.1 Set membership in Modula

One of the primitive predicates of Modula is the set membership test —

Element `IN` *Set*

where *Set* is an expression of some word-sized set type, and *Element* is an expression of an appropriate ordinal type. In the case, that the element expression evaluates to a value outside of the range of the base type of the set type, the predicate evaluates to false.

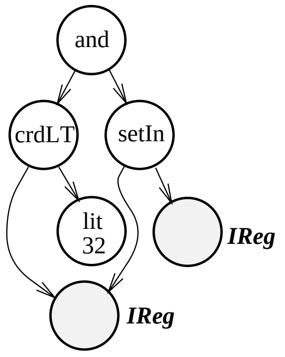

Figure 6: *guarded set membership test*

In our code-trees, the unguarded set membership test is a primitive. The canonical form of Modula's *IN* statement, is as shown in Figure 6.[1] The straightforward encoding of this tree fragment would first evaluate the Boolean cardinal less-than (crdLT) condition, and if the condition evaluated to true perform the set membership test. For *SPARC* as for most *RISC* machines, the set membership test would load the element value into a register, and perform a logical right shift of the set by the element value. This single instruction places the bit of interest in the zero-th position, where it is extracted into a register by logically *AND*ing with the immediate value 1.

The necessity of this final masking operation gives the clue to the optimisation which we seek. If the Boolean result of the range test is evaluated into a register, then that value can be used as the extraction mask, avoiding a branch, and obtaining the logical *AND* operation for free!

We may generalise this pattern by recognising that we may perform the same trick whenever a Boolean value in a register is logically *AND*ed with a bitset membership test. The required production is simplicity itself —

$$\mathit{BReg} \rightarrow \mathrm{and}(\mathit{BReg},\mathrm{setIn}(\mathit{IReg},\mathit{IReg}))$$

where it will be recollected that *BReg* is Boolean value in a register.

If the register names at the three leaves of the pattern are *r1,r2,r3* respectively, then the emitted *SPARC* code should be —

```
srl  r2,r3,rTmp
and  rTmp,r1,rDst
```

where *r1* is the Boolean result of the range test, *r2* is the set, and *r3* is the element. The final Boolean result is developed into the destination *rDst*. A similar production makes an equivalent saving in the case that the final result is required as a condition code flag.

[1]This diagram is for a 32-bit machine, and is a "tree" with a shared leaf node.

8.2 Bitfield extraction in C

An idiom of language *C* yields to the same kind of pattern recognition. For almost all machines, in order to extract a signed bitfield from the middle of a word it is necessary to shift the word to the left until the sign bit of the field is in the sign bit of the register. The register value is then arithmetically shifted until the least significant bit of the field is in the zero-th position of the word. This is always necessary if the resulting extracted bitfield is required in a register.

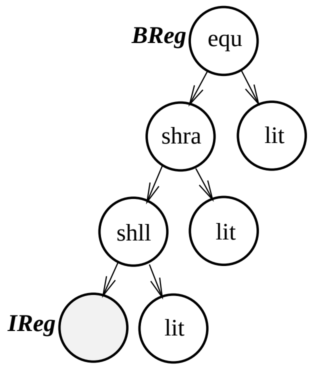

Figure 7: *bitfield extraction idiom*

The special idiom arises when the extracted value is simply compared with a literal for equality or inequality. In that case the *unshifted* bitfield can be extracted with a single masking instruction, and compared with a preshifted version of the original comparand literal. This again saves a whole instruction. The production template for this, shown in Figure 7, is strikingly non-local. In the figure *shll,shra* are the logical left shift and arithmetic right shift operators respectively. Analogous productions apply to inequality tests, unsigned bitfields, and when the resulting value must be developed into a condition code register rather than the *BReg* shown in Figure 7.

8.3 Plugging in new patterns

The examples shown above are just two of the kinds of patterns which it is simple to recognise using the *MBURG* infrastructure. Once such a pattern is recognised an extra production is added to the specification, *MBURG* is invoked, and the compiler rebuilt. It is entirely possible to recognise such an idiom, construct a matching production, and incorporate it into the compiler in a matter of minutes. As usual, prudence demands that some hours be spent in regression testing to ensure the safety of these five-minute enhancements.

9 Future work

Our production *MBURG*-generated *SPARC* code selector now produces significantly better code than the previous hand-written code selectors[10], even though the previous selectors had been carefully hand-tuned over several years. However, the new code selector is both larger than the previous version, and is also somewhat slower at compile time. We happen to think that the advantages in maintainability of the automatically produced selectors is compensation for these factors. Nevertheless, we are actively pursuing methods to reduce the memory and time overheads of the new technology.

In particular, the multiple deep recursions of the method lead to wildly fluctuating call depth traces. This is a very undesirable characteristic for machine architectures which are based on register windows, since windows are continuously spilled and restored. Proebsting[11] has recently shown that for the *BURG* automata, this effect can be reduced by preemtive reductions. In fact, he was able to show that for any given grammar there is some characteristic small maximum number of unreduced nodes which need to be retained. Unfortunately, this theory does not directly apply to rewriters which perform their dynamic programming at pattern matching time. We believe that an intuitive grasp of the circumstances under which the equivalent of a preemtive reduction can be invoked is at the heart of the "art" of construction of fast hand-written code selectors. We would like to understand and automate these intuitions, without giving up the flexibility which is inherent in compile time dynamic programming.

References

[1] A V Aho and S C Johnson. Optimal code generation for expression trees. *Journal of the ACM*, Volume 23, Number 3, pages 488–501, 1976.

[2] Jack W Davidson and Christopher W Fraser. Code selection through object code optimization. *Transactions on Programming Languages and Systems*, Volume 6, Number 4, pages 506–526, 1984.

[3] C W Fraser and D R Hanson. *A Retargetable C Compiler: Design and Implementation.* Benjamin Cummings, 1995.

[4] C W Fraser, D R Hanson and T A Proebsting. Engineering a simple efficient code-generator generator. *Letters on Programming Languages and Systems*, Volume 1, Number 3, pages 213–226, 1992.

[5] C W Fraser, R R Henry and T A Proebsting. Burg — fast, optimal instruction selection and tree-parsing. *SIGPLAN Notices*, Volume 27, Number 4, pages 68–76, 1992.

[6] R S Glanville and S L Graham. A new method for compiler code generation. *5th POPL conference record*, pages 231–240, 1978.

[7] K John Gough. Bottom up tree rewriting with mburg: the mburg reference manual. `ftp://ftp.fit.qut.edu.au/` in directory `/pub/coco`. Complete source code and reference manual is in `mburg08.tar.gz`.

[8] K John Gough. The dcode intermediate program representation: Reference manual and report. `ftp://ftp.fit.qut.edu.au/ /pub/papers/dcode300.ps.Z`. Online document specifying the intermediate form used by the gardens point compilers.

[9] K John Gough. Bottom-up tree rewriting tool mburg. *SIGPLAN Notices*, Volume 31, Number 1, 1996.

[10] K.John Gough and Jeffrey Ledermann. Register allocation in the gardens point compilers. In *Proceedings ACSC18, Adelaide, Australia.* Australian Computer Science Society, 1994.

[11] T Proebsting and B R Whaley. One-pass, optimal tree parsing – with or without trees. In *Proceedings of the International Conference on Compiler Construction*, Lecture Notes in Computer Science 1060, pages 294–308, Linköping, Sweden, 24-26 April 1996. Springer Verlag.

[12] Todd A Proebsting. *Code Generation Techniques.* Ph.D. thesis, University of Wisconsin – Madison, 1992.

[13] QUT. Gardens point modula home page. `http://www.fit.qut.edu.au/CompSci/PLAS/GPM`. Information on gardens point compilers, their availability, and documentation.

[14] Anthony M Sloane. An evaluation of an automatically generated compiler. *Transactions on Programming Languages and Systems*, Volume 17, Number 5, pages 691–703, 1985.

[15] William M Waite. The cost of lexical analysis. *Software Practice and Experience*, Volume 16, Number 5, pages 473–488, 1986.