Bottom-up Tree Rewriting Tool MBURG

K John Gough *
j.gough@qut.edu.au

July 18, 1995

Abstract

mburg is a tool for producing bottom up tree rewriters. It has been used for code selection in compilers. It produces hard coded tree pattern matchers from tree grammars, with dynamic programming at runtime. It is comparable in its capabilities with *iburg*[1], but has a rather different implementation, and produces its output in *ISO Modula-2*. The source code for the tool is available by ftp.

1 Background

Bottom up tree rewriting is the method of choice for code selection in modern compilers. Given a tree grammar, and a cost expression for each production, it allows for optimal code selection from trees.

The *iburg*[1] tool has been widely used, and forms the basis for code selection in the widely available retargetable *C* compiler **lcc**[2]. The **gardens point** compilers written at Queensland University of Technology use a stack-based intermediate form[3] to communicate between target independent frontends and language independent backends. Our current backends target *Intel 486, Mips, Sparc* and *Alpha* architectures, and produce very high quality code using shadow stack automata for code selection. We have noticed however, that as each of our backends has evolved, the code selection automata have become increasingly complex, as we recognize increasing numbers of special case patterns.

We decided to experiment with bottom up tree rewriting code selection as an alternative technology. In order to do this we created the **mburg** tool, created some alternative code selectors, and evaluated their performance. We are making the tool generally available in the hope that it will be of interest to two separate groups. Firstly, developers working with Modula-2 rather than the *C* language will find integration of the resulting code selectors simpler than is the case with *iburg*. Secondly, the implementation is sufficiently different from *iburg* to be of some intrinsic interest.

*Faculty of Information Technology, Queensland University of Technology. Brisbane, Australia.

2 The MBURG Tool

Code selectors generated by **mburg** perform dynamic programming at compile-time, and make two passes over the subject trees. In the first pass, the nodes of the tree are labelled during a recursive bottom-up traversal. The labelling for each node encodes the minimal cost of placing the denoted value of that subtree in any of the possible "typeforms." The labelling also records the ordinal of the production rule which produces each of the minimal costs. This is the *labelling* pass.

The second pass uses the state information to guide a top-down traversal, which performs a virtual rewriting of the tree. In this second pass the nodes are not visited in the order determined by the structure of the tree, but rather are visited in an order determined by the matched patterns of the production rules which are selected by the labelling. At each node a specified semantic action may be performed. This is the *reduction* pass.

mburg currently produces the labelling pass and a single reduction pass completely automatically, requiring no hand-written code at all. This is sufficient for the production of virtual assembly language. We may at some future stage define declarative syntax for specifying semantic actions for a second reduction pass.

3 Rationale

We decided to create a separate tool to produce our rewriters, rather than just use *iburg*, for a number of reasons. Firstly, we wished to experiment with some alternative implementation techniques, including the possiblility of entirely specifying the reduction semantic actions declaratively. Secondly, if we had to create a new tool, we wished to do it in Modula-2, which is our implementation language of choice.

The tool accepts a declarative specification, which specifies the terminal symbol grammar, and the names of the data types and record fields which the matcher needs to access. Of course in a *C* implementation all of this is handled by preprocessor macros. A prolog declares all of the externally defined imports which the pattern matcher or the reducer requires.

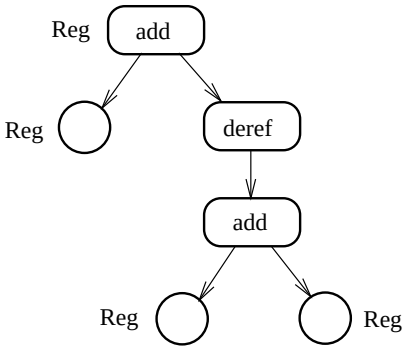

Figure 1: an example nested tree-pattern

Productions are specified as fully parenthesized prefix expressions. For example, the tree pattern shown in figure 1 is given by the production —

```
Reg = add(Reg,deref(add(Reg,Reg)))
```

In this example, *Reg, Imm* are *typeforms* corresponding to register and immediate values, while *add, deref* are terminal symbols of the grammar.

In the case that productions are only applicable when certain semantic conditions are met, productions may have an optional predicate expression, as well as a constant cost. This seems to be conceptually cleaner than the use of a conditional cost function with an "infinite" cost to eliminate impossible cases. Thus the possibility of folding a multiplication into a left shift might be specified by the production —

```
Reg = mul(Reg,Imm) & IsPwr2($2.litVal)
```

mburg simply adds the optional predicate as an extra conjunct in the normal pattern matching expression —

```
IF (this^.l^.s.rules[Reg] <> noRule) AND
   (this^.r^.s.rules[Imm] <> noRule) AND
    IsPwr2(this^.r^.s.litVal) THEN
  update the state vector of this;
END;
```

In this example it has been assumed that the left and right subtrees of the node type have been declared as being selected by fields "l" and "r", and that "s" is the state vector field.

It is possible to specify semantic actions which are performed during the reduction traversal of the tree. These semantic actions are specified as code which may refer to the nodes and state vector components of the pattern symbolically. Thus $0 is the state vector of the pattern root, while $1,2... are the state vectors of the first, second, ... leaf-nodes of the pattern respectively. For example, a typical semantic action for the conditional production seen above might be specified by —

```
Reg = mul(Reg,Imm) & IsPwr2($2.litVal)  1
     (. NewVReg($0.dst);
        EmitRRI(shll, $0.dst, $1.dst,
                Log2($2.litVal)); .)
```

where "(." and ".)" are meta-symbols delimiting the semantic action. The reducer will include the following code fragment —

```
  do reduction recursion(s);
  NewVReg(self^.s.dst);
  EmitRRI(shll, self^.s.dst,
          self^.l^.s.dst,
          Log2(self^.r^.s.litVal));
END;
```

The sematic actions in the automatically produced reduction are placed *after* the return of the recursions which carry the *required typeform* down the tree. This placement ensures that synthesized attributes such as the destination registers of the subtrees will be available when the instruction is to be emitted.

4 Results on Sparc

We decided to investigate the merits of changing our code generators to use bottom up rewriting as a means of code selection, in a controlled environment. We took our *Sparc-Solaris* backend, and replaced the code selector with a tree-rebuilder which reconstructed a forest of code trees from the incoming stack instructions. The automatically produced matcher and reducer traversed this forest emitting *virtual assembly language* into a code buffer in the completely unchanged register allocator and file writer. Of course it would been much more convenient to connect the matcher to the abstract syntax tree of the frontend, but we wished to leave *everything* completely unchanged except for the code selection module, in order to isolate those issues which related to code selection.

We chose to experiment on the *Sparc* version, since selecting optimal code for this architecture is of intermediate complexity, being slightly more complex than *mips R3000* but much simpler than *Intel 486*.

After some experimentation we found that we could achieve the same code quality as our existing code selection, using a total of approximately 160 productions. It should be noted that this figure is for an operator set which is sufficiently rich to implement all current procedural languages. For example, it has operators for two different flavors of integer division and remainder, and six different flavors of floating point to whole number conversion. It also implements both unchecked and overflow-checked arithmetic for both signed and unsigned values. The new version actually does produce marginally better code than the original, since is was convenient to recognize some patterns the contexts of which were simply too complex for our handwritten automata.

The produced code selectors ran somewhat slower than our original code generators on sparc platforms, but the code ran either at the same speed or even slightly faster on other architectures. We speculate that the deep recursions involved in the traversals were adding overhead due to the spilling of register windows when hosted on sparc.

5 Implementation details

mburg parses the input declarations, and builds an internal representation of the set of productions. Unlike *iburg*, **mburg** performs comprehensive consistency checks on the tree grammar. It checks that each non-terminal symbol is reachable from the start symbol, and that each has a terminating derivation. The tool checks that every terminal symbol is declared, and that every use of each terminal symbol agrees on the arity.

The current version of **mburg** produces a pattern matcher which performs its control flow by procedure dispatch. There are two arrays of procedure variables. The first is an array of hard-wired labelling procedures, one for each terminal symbol value. Each of these procedure knows the arity of its own symbol, and uses the node tag of its child nodes to dispatch the appropriate recursive calls to each child. When the recursions return the procedure performs matching for all those patterns which start with the correct terminal symbol. Figure 2 is the skeleton of the label procedure for a unary symbol *neg*.

```
PROCEDURE negLabel(this : Tree);
BEGIN
 (* neg node: first do recursion *)
  match[this^.l^.tag](this^.l);
 (* now match patterns *)
  IF this^.l^.s.rules[Reg] <> noRule THEN
    ...
END negLabel;
```

Figure 2: Unary labelling procedure

The second array selects reduction helper procedures, one for each production rule. Each of these procedures knows how to visit the leaf nodes of the pattern which it recognizes. Each reduction helper checks the state labels of its leaf-nodes to determine the correct production rule, and hence the proper helper procedure to dispatch on each such leaf. Figure 3[1] is the skeleton of the reduction procedure for the pattern shown in figure 1. Note here that although the *add* node is structurally binary. The selected pattern has three leaves. Thus the recursion

[1] Of course, this is not a sparc production!

```
PROCEDURE Reduce27(self : Tree);
  VAR pix : ProdIndex;
BEGIN
 (* Reg = add(Reg,deref(add(Reg,Reg))) *)
  pix := self^.l^.s.rules[Reg];
  rHelp[pix](self^.l);
  pix := self^.r^.l^.l^.s.rules[Reg];
  rHelp[pix](self^.r^.l^.l);
  pix := self^.r^.l^.r^.s.rules[Reg];
  rHelp[pix](self^.r^.l^.r);
 (* do semantic action *)
  ...
END Reduce27;
```

Figure 3: Reduction helper procedure

is three fold, and skips directly to the (structural) great-grandchildren in the right subtree.

We compared other organisations for the code, such as having the labelling pass controlled by a table of node arities, and using a switch statement selected on node tag value. However, we found that the described organisation was at least as fast on all architectures.

6 Limitations, and future development

The simple, predetermined recursion pattern followed during reduction turns out to be adequate for those cases where a single reduction pass is sufficient. However, those cases which require a second reduction pass currently require some handwritten code. For example, exact Sethi numbers may be determined as a semantic action during reduction. To use these attributes to reduce register pressure would require code to be emitted during a second reduction pass. In this second reduction, the node visit order would be controlled by the attributes generated by the first reduction pass.

It is a relatively simple matter to manually edit a second reduction skeleton, but it is possible that future versions of **mburg** may introduce syntax to allow semantic actions to be declaratively specified for additional reductions.

We are interested in finding ways of automatically determining when subtrees may be pre-emptively reduced during labelling. This should improve the runtime performance of the generated code selectors on machine architectures with register windows, by avoiding multiple deep recursions.

7 How to get MBURG

The source code of **mburg** is available by anonymous ftp, together with a postscript format reference manual.

host:	`ftp.fit.qut.edu.au` (131.181.2.16)
directory:	`/pub/coco`
files:	`mburg06.tar.Z`
	`mburg06.ps.Z`
	`sigplan.ps.Z` (this file)

Binary distributions of the Linux and OS2/EMX versions of **gardens point modula2** are available from the same site.

References

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

[2] C.W. Fraser and D.R. Hanson. A code-generation interface for ANSI-C, *Software Practice and Experience*, Vol 21, Sep. 1991. (Also fully described in the same author's *A Retargetable Compiler for ANSI C*, Benjamin Cummings, 1995).

[3] K.J. Gough. The DCode Intermediate Program Representation, Reference Manual and Report, QUT 1992–1994.This report is maintained in electronic form on the ftp server `ftp.fit.qut.edu.au` (Internet 131.181.2.16). The current (January 1995) version is 2.2