ML-Burg — Documentation

Florent Guillaume

École Normale Supérieure
45, rue d'Ulm
75005 Paris, France
Florent.Guillaume@ens.fr

Lal George

Room 2A-426
AT&T Bell Laboratories
Murray Hill, NJ 07922
george@research.att.com

June 23, 1993

1 Introduction

ML-Burg is a Standard ML version of the `iburg` tool developed by Fraser, Hanson and Proebsting [3]. ML-Burg generates a Standard ML program to perform bottom-up rewriting of an input tree. Cost information associated with each rewrite rule is used to derive the minimum rewrite cost for the entire tree. A successful reduction corresponds to rewriting the input tree to a special non-terminal symbol called the *start non-terminal*. Upon successful reduction, facilities are provided to walk the tree emitting semantic actions corresponding to the rules that matched.

Like `iburg`, ML-Burg generates a program that consists of a large case statement. Indeed, the `i` in `iburg` was meant to indicate *interpreted*-burg to distinguish it from table driven implementations of similar tools [1, 4]. We arbitrarily decided to drop the `i` (no pun intended).

Given a system of rewrite rules augmented with costs, called the *ML-Burg specification*, ML-Burg generates the following:

- signature BURM_INPUT_SPEC
- signature BURM
- structure BurmOps
- functor BurmGen(In : BURM_INPUT_SPEC) : BURM

The signature BURM_INPUT_SPEC specifies utilities over the user supplied input tree. The required matcher is obtained by applying the functor BurmGen to a structure matching BURM_INPUT_SPEC.

2 ML-Burg specifications

Figure 1 shows the extended BNF grammar for ML-Burg specifications. Grammar symbols are *italicized*, terminals are in typewriter font, $\{X\}$ represents zero or more occurrences of X, $[X]$ means X is optional, *cost* is a non-negative integer, *trailer* and *header* are arbitrary pieces of text, and everything else is an identifier. An identifier is a leading alphabet followed by zero or more alphanumeric characters and underscores. Comments are delimited by (*, and *).

A specification consists of three parts: *declaration*, *rule*, and *trailer*, separated by %%.

A %term declaration enumerates the operators or function symbols used to construct nodes of the tree. There must be at least one %term declaration for a valid specification. The %start declaration, which defaults to the left hand side *nonterminal* of the first rule, declares the *start non-terminal*. The *header* part is text that is included verbatim at the beginning of the matcher. The names of the modules generated by ML-Burg may be changed by using a %sig declaration (case of *signame* is not significant). For example if we had a line "%sig glop" in the declarations, the generated names would be GlopOps, GLOP_INPUT_SPEC, GLOP and GlopGen. This allows for multiple matchers in the same program.

The rule-part of the specification (following the first %%) describes the tree grammar or the rewrite rule system to use. The *nonterminal* : *tree* specification can be viewed a rewrite rule of the form, *nonterminal* ← *tree*. Each

```
spec         →  declaration %% {rule} %% trailer

declaration  →  %term op {| op}
             |  %start nonterminal
             |  %termprefix termprefix
             |  %ruleprefix ruleprefix
             |  %sig signame
             |  %{ header %}

op           →  operator [= opname]

rule         →  nonterminal : tree = rulename [( cost )] ;

tree         →  nonterminal
             |  operator [( tree{,tree} )]
```

Figure 1: EBNF ML-Burg specifications

operator used in a tree must be mentioned in a `%term` declaration. The special case of *nonterminal* ← *nonterminal*, specifies a chain rule. Associated with each rule is an optional cost that defaults to zero. The *rulename*, which is not necessarily unique, is used to identify the rule during the emission of semantic actions. It is important to note that the same *rulename* may be associated with multiple rules.

The *trailer* is an arbitrary piece of text that is inserted at the end of the generated matcher. This is typically segments of program that will perform the semantic actions.

Figure 2 show a sample specification taken from [3]. The `%termprefix`, and `%ruleprefix` are explained in subsequent sections.

3 Interface between the matcher and the program

3.1 structure BurmOps

The `structure BurmOps` declares a type `ops` that enumerates the operators or functions symbols specified in `%term` declarations of the specification. The matcher cannot extract the operator from the user supplied tree, or establish a correspondence between nodes in the tree and operators in the specification.

```
%term ASGNI | ADDI | CVCI | INDIRC | IOI | ADDRLP | CNSTI
%termprefix T_
%start stmt
%%
stmt:   ASGNI(disp,reg)         = stmt_ASGNI_disp_reg   (1);
stmt:   reg                     = stmt_reg;
reg:    ADDI(reg,rc)            = reg_ADDI_reg_rc       (1);
reg:    CVCI(INDIRC(disp))      = reg_CVCI_INDIRC_disp  (1);
reg:    IOI                     = reg_IOI;
reg:    disp                    = reg_disp              (1);
disp:   ADDI(reg,con)           = disp_ADDI_reg_con;
disp:   ADDRLP                  = disp_ADDRLP;
rc:     con                     = rc_con;
rc:     reg                     = rc_reg;
con:    CNSTI                   = con_CNSTI;
con:    IOI                     = con_IOI;
%%
```

Figure 2: Example of ML-Burg specification.

In the example above (Figure 2), the user may have defined the tree to be:

```
datatype tree = ...
              | CNSTI of int
              ...
```

The data constructor CNSTI is of arity 1, whereas, in the specification it is used with arity 0.

For the example, the generated structure would be:

```
structure BurmOps = struct
  datatype ops =
       T_ASGNI
     | T_ADDI
     | T_CVCI
     | T_INDIRC
     | T_IOI
     | T_ADDRLP
     | T_CNSTI
end
```

The `%termprefix`, if specified, is used to prepend the *termprefix* to each operator. If the optional `=`*opname* is specified with the operator, then *opname* is used in the datatype `ops` instead of the operator.

3.2 signature BURM_INPUT_SPEC

The `signature BURM_INPUT_SPEC`, shown below, specifies the interface to the user supplied input tree.

```
signature BURM_INPUT_SPEC = sig
  type tree
  val opchildren : tree -> BurmOps.ops * (tree list)
end
```

It contains:

- The type `tree` of trees on which the program operates.
- A function `opchildren` which takes a `tree` and returns the operator (of type `BurmOps.ops`) at the root of this tree, and a list of children of this root (a `tree list`).

The function `opchildren` must return the children in the order in which they appear in the rules (which is the only order the matcher knows of). For example, if the root of the tree corresponds to the operator `ASGNI` (Figure 2, the first element of the list must be the tree corresponding to `disp`, and the second to `reg`.

3.3 signature BURM

The structure generated by the functor `BurmGen` matches the signature `BURM`. Specified in `BURM` are:

- An exception `NoMatch`, which is raised if `reduce` is called on a tree which cannot be rewritten to the start non-terminal.
- The type `tree` (the one passed to the functor).

- A datatype `rule` enumerating the rules of the tree grammar. This datatype is defined by prepending to each *rulename*, the *ruleprefix* from a `ruleprefix` declaration (if any). The arity of each constructor is equal to the number of non-terminal symbols in the pattern. Each `(rule,tree)` pair specifies the `rule` and the `tree` that matched each non-terminal symbol. These pairs describe the remaining steps in the reduction.

- A function `reduce` which takes a `tree` and returns a pair `(rule * tree)`. As described above, the `rule` describes the best match that generated the start non-terminal, and the `tree` is the original input tree.

In the example above, the signature `BURM` generated is:

```
signature BURM = sig
  exception NoMatch
  type tree

  datatype rule =
       stmt_ASGNI_disp_reg  of (rule*tree) * (rule*tree)
     | stmt_reg             of (rule*tree)
     | reg_ADDI_reg_rc      of (rule*tree) * (rule*tree)
     | reg_CVCI_INDIRC_disp of (rule*tree)
     | reg_I0I
     | reg_disp             of (rule*tree)
     | disp_ADDI_reg_con    of (rule*tree) * (rule*tree)
     | disp_ADDRLP
     | rc_con               of (rule*tree)
     | rc_reg               of (rule*tree)
     | con_CNSTI
     | con_I0I

  val reduce : tree -> (rule*tree)
end
```

The functions `reduce` is used as follows: given a tree t_0, `reduce` returns an initial pair (r_0, t_0) (it returns t_0 to make the user program simpler - Section 4). r_0 describes the first rule to apply to t_0 to perform the optimal reduction. This rule, except in trivial cases, will have in its pattern several *nonterminal*s which represent other trees to be reduced. To that end, the constructor r_0 carries the pairs $(r_1, t_1), \ldots, (r_n, t_n)$ of its children. (r_i, t_i) corresponds to the ith *nonterminal* in the tree pattern for the rule r_0, when read from left to right. These pairs can be used to find the rule to use to reduce each child. In turn, the rules $r_1, \ldots, r_n$ carry information about their children.

Why return the tree in addition to each rule? Often, additional information is stored in the tree, and it may be necessary to access this information when a semantic action is executed. This information may include constants like *integers*, *reals* or *string*, or more complex objects like symbol table information.

4 Example

Using the ML-Burg specification of Figure 2, a sample input to provide to the functor BurmGen is shown below:

```
structure In : BURM_INPUT_SPEC = struct
  structure BO = BurmOps
  datatype tree =
        ASGNI  of tree * tree
      | ADDI   of tree * tree
      | CVCI   of tree
      | INDIRC of tree
      | IOI
      | ADDRLP of string
      | CNSTI  of int
  fun opchildren t =
    case t of
      ASGNI(t1,t2)  => (BO.T_ASGNI, [t1,t2])
    | ADDI(t1,t2)   => (BO.T_ADDI,  [t1,t2])
    | CVCI(t1)      => (BO.T_CVCI,  [t1])
    | INDIRC(t1)    => (BO.T_INDIRC,[t1])
    | IOI           => (BO.T_IOI,   [])
    | ADDRLP _      => (BO.T_ADDRLP,[])
    | CNSTI _       => (BO.T_CNSTI, [])
end
```

In Figure 3 we show a sample function called walk that performs semantic actions. The semantic actions merely prints out the rules that applied assuming the children of each node are traversed from left to right. Note that in the action corresponding to the reg_CVCI_INDIRC_disp rule, the recursive call to walk, specifically walk disp, steps over the CVCI and INDIRC nodes — yet, information associated with the CVCI or INDIRC is available to this rule.

A graphical representation of sampleTree and the result of executing:

```
- open Example; doit sampleTree;
```

is shown in Figure 4.

5 Using mlburg and debugging

The executable for ML-Burg is usually called mlburg. When mlburg is presented with a file name *filename*.burg, a file *filename*.sml is created - assuming no errors were encountered. This generated file will contain all the modules described above, and can be directly loaded into an interactive session. The error messages displayed during the execution of mlburg are self-explanatory.

During execution, a NoMatch is raised when the tree cannot be reduced to the start non-terminal. For example, suppose that the function reduce was called on the tree CNSTI. Obviously, CNSTI can only be reduced to a con and an rc. The matcher would then print the message :

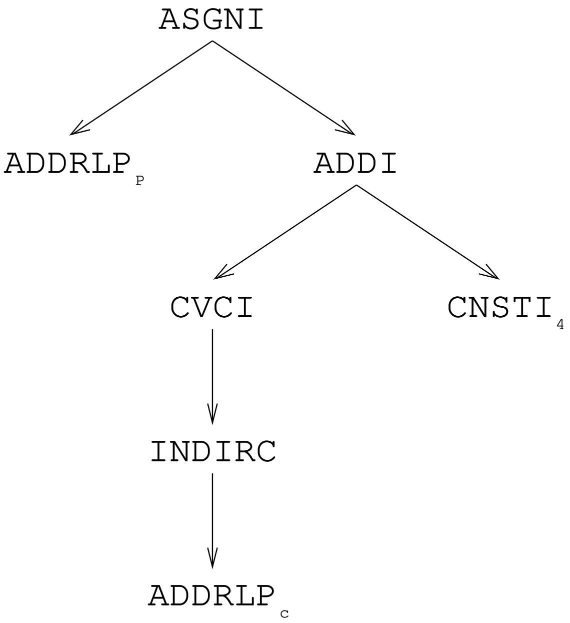
```
No Match on nonterminal 0
Possibilities were :
rule 9 with cost 0
rule 11 with cost 0
```

The *nonterminal*s and rules are printed using integers, but a correspondence between these integers and the identifiers used in the specification can be found at the beginning of the generated SML file.

Note, however, that such debugging information will only be useful if the incorrect match occurs at the first level of the reduction to the start nonterminal. If reduce is called with ASGNI(IOI,x), the problem occurs deeper, because it is ultimately IOI that cannot be reduced to a disp, and the fact

```
structure Example = struct
  structure Burm = BurmGen (In)
  open In

  local val num = ref 1 in
    fun new s = (s^(makestring (!num)) before inc num)
  end

  fun walk (Burm.stmt_ASGNI_disp_reg (disp,reg), _) =
        let
          val (disp’,reg’) = (walk disp, walk reg)
          val stmt = new "stmt"
        in
          say (stmt^" <- ASGNI ("^disp’^" + "^reg’^")\n"); stmt
        end
    ...
    | walk (Burm.reg_CVCI_INDIRC_disp disp, _) =
        let
          val disp’ = walk disp
          val reg = new "reg"
        in
          say (reg^" <- CVCI (INDIRC ("^disp’^"))\n"); reg
        end
    ...
    | walk (Burm.con_CNSTI, CNSTI i) =
        let
          val con = new "con"
        in
          say (con^" <- CNSTI "^(makestring i)^"\n"); con
        end
    ...
    | walk _ = (print "Error, bad match in walk\n"; raise Match)


  fun doit t = walk (Burm.reduce t)

  val sampleTree = ASGNI (ADDRLP "p",
                          ADDI (CVCI (INDIRC (ADDRLP "c")),
                                CNSTI 4))
end
```

Figure 3: Example program.



```
disp1 <- ADDRLP p
disp2 <- ADDRLP c
reg3 <- CVCI (INDIRC (disp2))
con4 <- CNSTI 4
disp5 <- ADDI (reg3,con4)
reg6 <- disp5
stmt7 <- ASGNI (disp1 + reg6)
```

Figure 4: `sampleTree` and the produced output.

that the whole tree cannot be reduced to the start non-terminal is only a consequence of it. In such cases, the matcher will only give the message :

```
No Match on nonterminal 0
Possibilities were :
```

At this stage it would be necessary to check the completeness of the rewrite system. Automated tools to do this, may be expected in the future[2].

References

[1] Balachandran, A., Dhamdhere, D. M., and Biswas, S. Efficient retargetable code generation using bottom-up tree pattern matching. *Computer Languages 15(3)* (1990), 127–140.

[2] Emmelmann, H. *Testing completeness of code selector specifications.* Springer-Verlag, 1992, pp. 163–175.

[3] Fraser, C. W., Hanson, D. R., and Proebsting, T. A. Engineering a simple, efficient code generator generator. In *Letters on Programming Languages and Systems* (1992), ACM.

[4] Proebsting, T. A. Simple and efficient burs table generation. In *SIGPLAN '92 Conf. on Programming Language Design and Implementation* (June 1992), ACM, pp. 331–340.