iburg++ : A Code Generator Generator in C++

T Sree Meghana, IIT Bombay
RnD Project
Guide : Prof. Uday Khedker

Introduction :

A code generator generator accept tree patterns, associated costs and machine model and emits a C++ code which on taking an input tree of the above grammar, emits its object code. The main objective being optimal object code generation, a few other necessary aspects are instruction selection, register allocation and optimal order of execution of instructions. BURG is a program that generates a fast tree parser using BURS (Bottom-Up Rewrite System) technology. BURG has been used to construct fast optimal instruction selectors for use in code generation. “iburg”, its implementation, is an instruction selector that accepts a cost-augmented tree grammar and emits a C program that discovers in linear time an optimal parse of trees in the language described by the grammar. The idea is to implement BURG in C++ and extend it further for register allocation and object code generation using Sethi Ullman code generation algorithm.

Abstract :

A code generator for a compiler is applied to an intermediate representation (IR) of the input program that has been computed during preceding phases of compilation. This intermediate representation of input program is passed to the code generator in the form of a tree following a grammar which is used in generating the code generator. Instruction selector generators significantly simplifies the development of code generators and “iburg” is one such simple instruction selection generators. The example below shows the necessity of an intelligent instruction selection algorithm.

Consider the simple grammar G1 given below. Here each line is a production rule where the nonterminal on the left hand side is a cover for the tree pattern on the right hand side. The cost for this matching is also given along with the production rule.

```
%%
        reg :    PLUS ( reg , reg )              cost : 2
        reg :    PLUS ( reg , CONST )            cost : 2
        reg :    CONST                           cost : 1
%%
```

The tree given for instruction matching is *PLUS (CONST, CONST)*. For this input tree, more than one covering of the tree is possible corresponding to different ways to generate code. Both the ways of generating an instruction cover are listed below.

```
        reg:    CONST                   cost -> 1
        reg:    CONST                   cost -> 1
        reg:    PLUS ( reg, reg )       cost -> 2
```

and the total cost for code generation using this cover is 4. The other one is

```
        reg:    CONST                   cost -> 1
        reg:    PLUS ( reg, CONST )     cost -> 2
```

In this case the total cost is 3.

In a complex machine model there may be many such cases where intelligent instruction selection is necessary. The instruction selection generator BURG uses bottom-up rewriting theory (BURS) for instruction selection. According to BURS, an intermediate representation tree can be translated to a sequence of instructions using the following strategy. In a bottom-up traversal all lowest-cost patterns that match each node are computed and associated with the node. This involves matching the right-hand sides of the productions to the tree node, taking into account earlier matches for its sub-trees. BURG generates a tree automaton from the input grammar for this purpose. Instructions are then selected in a top-down traversal that is driven by the goal non-terminal for the root of the tree. “iburg” is an implementation of the idea BURG in C and “iburg++” is the extended version of it including code generation in C++.

Once the instruction selection is done the main task at hand is register allocation and evaluating instructions in an order that optimizes register usage and usage of temporary memories. Though BURG does optimal instruction selection, the code generated directly from this instruction cover may not be optimal one in all ways one of which is number of registers required.

This may be a drawback especially in cases where there is a limit on the number of registers available. This is where a special code generation algorithm comes in.

Sethi Ullman algorithm for code generation is a simple algorithm that generates optimal code for a given tree by determining the evaluation order of the sub-trees which require minimum number of registers. An important aspect of Sethi Ullman algorithm is labelling the instruction tree. Given an expression tree, we can label each node by the number of registers required to evaluate that node following the labelling algorithm given by Sethi Ullman.
Once the labelling is done, generate code in the order specified by Sethi Ullman algorithm and the resultant code would the required optimal code.

Summing up “iburg++” would initially accept a tree grammar and generate a C++ file which would in turn take as input a tree of the above grammar and do instruction selection and code generation of it. The idea of instruction selection is adapted from BURG and some of the implementation aspects from “iburg”. And for code generation, a modified version of Sethi Ullman code generation algorithm is used which is described later.

Input :

To generate a code generator we would want to know the machine model which would be given as a tree grammar. A tree grammar is like a context-free grammar: it has rules, terminals, non-terminals, and a special start non-terminal. The right-hand side of a rule, called the pattern, is a tree. And the left hand side is the cover for this tree. We would also want to know the instruction set and assembly format of each instruction, names and kinds of all available registers and all for code generation. So in all, the following would be required for a code generator generator: machine model, different types of available registers and their names, instruction tree grammar where for each instruction we have the production rule, cost of evaluation of this rule and the semantics of the machine instruction it corresponds to.

Since registers here are nothing but the intermediate values which cover a subtree and nonterminals do exactly the same job, we hereby reference registers by nonterminals.
So in iburg++, the different types of registers are nothing but different types of nonterminals and the names of registers are the values those nonterminals can take. The exact input given to "iburg++" is in the following manner.

```
%%
        no_instruction
        load
        imm_load
        store
        add
        sub
 %%
        no_instruction  =       op_nsy, n
        load =                  op_r_o1, lw
        imm_load =              op_r_o1, li
        store =                 op_r_o1, sw
        add =                   op_r_o1_o2, add
        sub =                   op_r_o1_o2, sub
%%
```

```
        stmt = BEGIN
        reg = {r1,r2,r3,r4,r5,r6,r7,r8,r9,r10}
%%
        ASSGN, PLUS, MINUS, CON, ADDR
%%
        stmt: ASSGN(ADDR,reg)           = (1)       {store, $2, $1}
        reg: PLUS(reg,reg)              = (1)       {add, $$,$1,$2}
        reg: MINUS(reg,reg)             = (1)       {sub, $$,$1,$2}
        reg: PLUS(reg,CON)              = (1)       {addi, $$,$1,$2}
        reg: CON                        = (1)       {imm_load, $$, $0}
        reg: ADDR                       = (1)       {load, $$, $0}
%%
```

The blocks between "%%" and "%%" would correspond to the a set of information required for code generator generation. Lets call these blocks as *information blocks*. The information blocks required by "iburg++" are

→ a list of all possible operands in machine instructions to be generated.

→ for each operand, the semantics of the instruction it can generate., i.e its assembly mnemonic and its assembly format.

→ all the non terminals that can occur in the production rules, and the values they can take. the begin non-terminal is also specified here.

→all the terminals that can occur in the production rules / intermediate representation of expression tree.

→the production rules list where each of which would have the production rule, its cost, assembly instruction it corresponds to and the order of operands in the assembly instruction the semantics of which is like this

cover : node (l-op, r-op) = (cost) {operand corresponding the machine instruction, r, o1, o2}

The order of operands are specified using the below representation

$$ - result non terminal

$1 - left operand

$2 - right operand

$0 - the root of pattern of production rule

iburg++ :

The project uses flexc++ V1.08.00 and bisonc++ V4.05.00 for scanning and parsing the input grammar. The scanner and parser scripts are in files “scanner.ll” and “parser.yy” respectively. After the parser and scanners are generated by flexc++ and bisonc++, the generated files are modified for the scanner tokens from scanner to be sent to parser. For this purpose “modified_scanner.cc” and “modified_parser.cc” are present and they are copied to “scanner.cc” and “parser.cc” at every compilation. “iburg.h” has the definitions of all the data structures required for intermediate representation of data. “main.cc” is the file which does code generator generation. The file “memorylocation.h” has the API for allocating memory location in the stack in case one is needed. A basic version is implemented here but the user can modify it as per their requirement. The instructions to run the project are in the “README” file.

Output :

The output of “iburg++” is a C++ file that should take in an IR tree as input and generate code for it. Before generating code, it should generate the optimal instruction cover, do Sethi Ullman numbering of nonterminals required for evaluation of the subtree rooted at a node for every node and then order the code generation in such a manner that the non-terminals available are used optimally. Each of these can be done in O(1) times the size of the input tree., i.e in one pass over the input tree. We can enumerate the passes in the following manner.

1. Labelling every node of the tree with optimal cost required to evaluate it into every possible nonterminal. The function which does this is *label(Tree * p)*.
2. Picking the instructions for which code has to be generated at each node such that the given input tree evaluates into the required non-terminal. The function that does this is *pick_instr(Tree * p, Nonterminal begin_nt)*.
3. Numbering each subtree with the number of nonterminals of different kinds required to evaluate it or simply numbering it following Sethi Ullman code generation algorithm. The function which does this is *su_number(Tree * p)*.
4. The final pass is to generate code using the results of each of the above passes. The function that generates code is *gencode(Tree * p)*.

Before describing what each of the above passes does in detail, another important aspect of the output generated is to be put forward. The representation of data taken as input initially from the input file. Since the output file itself acts as a code generator it should contain all the data initially given along with the input tree grammar such as list of all terminals, list of all nonterminals and the values they can take, list of all the instructions and assembly formats and assembly mnemonics and list of all the rules and their semantics. Each of these should be provided before hand to the code generator before it starts with code generation.

Terminals do not carry much information along with them, so a simple list of them would suffice. There is a preprocessor directive *terms* which equates to the number of terminals present and an array *a_terminal* and a set *terminal* which has all the terminals listed initially. In case of the nonterminals, a simple list would not suffice. Since they are heavily involved in the further process a mapping from their name to assigned number, and from the assigned number to name and a data structure to hold their values and all are required. Similar to terminals they also have preprocessor directive *nonterms* which equates to the number of non-terminals present. There are also preprocessor directives of the form nt_<nonterminal name> which equates to their number. And again an array *a_nonterminal* and a set *nonterminal* and a vector containing a stack of values it each nonterminal can take *nonterminal_values* are present.
A sample is list of preprocessor directives is shown below.

```
#define nonterms 5
#define terms 19

#define nt_stmt 1
#define nt_reg 2
#define nt_regf 3
#define nt_cndtnl_goto 4
#define nt_uncndtnl_goto 5
```

After the nonterminals and terminals other defined, the other data that need to be stored is the machine model., i.e the instructions and the rules. Each instruction is represented by the *struct Instruction* which has members *oper, af and assembly_mnemonic* which correspond the operand that represents this instruction, its assembly format which is an enum, and its assembly mnemonic respectively. All the instructions are stored in the map named *instruction_map.* Similar is the case for rules. Each rule as explained in input has a tree called *pattern*, an nonterminal *lhs*, *cost*, instruction it corresponds to etc. The tree here is further a data structure and it is explained below

```
class State
{
public:
        std::vector<int> cost;
        std::vector<int> rule;
};

class Tree
{
public:
        string node;      /* non-terminal or terminal which is the root of this tree */
        Tree* left;       /* operands */
        Tree* right;      /* operands */
        string value;     /* the value this node takes. applicable in case where tree has only terminal  */
        State state;                    /* state of automaton */
        list<int> picked_instruction;   /* the instructions picked to generate code at this node */
        vector<int> nt_reqd;            /* the no.of non-terminals of each kind required */
        vector<bool> store_reqd;        /* whether the nonterminal is to be stored after evaluation */
        string result;                  /* result of the tree evaluated here */
        bool result_computed_by_store;
};

class Rule {
public:
        string lhs;             /* left hand side non-terminal */
        Tree pattern;           /* rule pattern */
        int number;             /* rule number */
        int cost;               /* associated cost */
        string instr;           /* corresponding machine instruction operator */
        Order args_order;       /* order of arguments of the machine instruction to be generated */
};
```

All the rules are stored in a map named vector named *rules* indexed by the number the rule is assigned. And by this all the data required is present in the output file for further use in code generation.

The first pass over the input tree encodes all full and partial optimal pattern matches viable at that node. The function which does this is *label(Tree * p)*. It recursively calls the same function over all of its subtrees and then goes call function *state_label(Tree* p)* to label itself. The function *state_label(Tree * p)* has the tree automaton and this is similar to the one in “iburg”. Closure functions are defined to complete the labelling process. If *reg : ADD(reg, reg)* and *reg : mem* are two production rules then for a tree with *ADD* as node, the cover of the tree can be both reg and

mem each with its own cost. So when labelling a node, the costs of evaluating into all possible non terminals should be computed. Closure functions come into picture in this kind of cases. Before generating the state_label function all closure functions must be generated. For all production rules of kind *nonterminal1 : nonterminal2*, a closure function on *nonterminal1* is created by name *closure_nonterminal1(State * s)*. In this way the tree is labelled in a bottom up manner.

The second pass over the tree is to pick the instructions to be generated at each node. A node may have multiple instructions due to closure functions and hence a list of instructions to be generated is maintained in order. Picking of instructions starts from the root, with the instruction corresponding to that of begin nonterminal. And the chain of instructions to be generated at each node are picked using the state labels and production rule patterns. The procedure followed is same as the one implemented in “iburg”.

Since after instruction selection, the main task at hand is code generation, and Sethi Ullman code generation algorithm suggests numbering each subtree with the number of registers required in evaluating it, the next pass of the input tree is to number it with number of non terminals required for its evaluation. A function *su_number(Tree * p)* is called recursively to number the tree in a bottom up manner. Along with numbering, it is also marked at every node if a store is further required in the code generation process(when number of nonterminals required exceed available number). The detailed explanation of the modification to Sethi Ullman algorithm is mentioned later in the next section.

The last and final pass over the subject tree is for code generation. The function *codegen(Tree * p)* initiates this process. This process of code generation requires two passes over the subject tree, but since both of them does code generation they are explained together. The first pass is to check if a store is required at any node and to generate code for that subtree. The function *check_and_generate_stores(Tree * p)* initiates this process. It checks at every node, if a store is required at it and generates code for it first by calling the function *gencode_store(Tree * p)* which generates code for the subtree first and then stores the result in a memory location. After code for all the nodes that require a store are generated, a final call is made to generate code for the remaining part of the subject tree. Some design decisions made in implementing this part are explained later.

Design Decisions Taken :

The Sethi Ullman algorithm for code generation is for a single non-terminal case(only register). But since we have multiple non-terminals to consider, totally following Sethi Ullman algorithm would not work. A simple modification is done to it for it to be applied for multiple non-terminal scenario. Though this may not be the optimal case, this would be better than applying Sethi Ullman algorithm at a node for each non terminal.

At a node p in the IR if r is the rule selected by instruction selector and t is its pattern then, the modified version for Sethi Ullman number is mentioned below

```
if(t.left==NULL && t.right==NULL){
        for (int i = 1; i <= nonterms; ++i)
        {
                if ( a_nonterminal[i] == r.lhs )
                        p->nt_reqd[i] = 1 ;
```

```
                else if ( t.node == a_nonterminal[i] )
                        p->nt_reqd[i] = 1 ;
        }
}

else if(t.right==NULL)
{
        su_number(p->left);
        for (int i = 1; i <= nonterms; ++i)
        {
                p->nt_reqd[i] = p->left->nt_reqd[i];
                if(rules[rno].lhs == a_nonterminal[i] && p->nt_reqd[i] == 0 )
                        p->nt_reqd[i]++;
        }
}
else
{
        su_number(p->left);
        su_number(p->right);
        for (int i = 1; i <= nonterms; ++i)
        {
                p->nt_reqd[i] = max(p->left->nt_reqd[i], p->right->nt_reqd[i]);
                if(p->left->nt_reqd[i] == p->right->nt_reqd[i] && p->left->nt_reqd[i] != 0)
                {
                        p->nt_reqd[i]++;
                        if(a_nt_values_count[i] < p->nt_reqd[i])
                        {
                                p->left->store_reqd[i] = true;
                                p->nt_reqd[i]--;
                                p->left->nt_reqd[i]=1;
                        }
                }
                if(rules[rno].lhs == a_nonterminal[i] && p->nt_reqd[i]==0)
                        p->nt_reqd[i]++;
        }
}
```

If the selected pattern for a node t has no children, then nonterminal allocation can be simply done by allotting 1 nonterminal each of its kind for the nonterminal at the lhs of the rule and node of pattern t if it is a nonterminal. So we simply iterate over all the available non terminals and allot them the number 1 if it satisfies any of the above condition.

Now if the pattern has a left child, first number the left child and then for each nonterminal copy the number assigned to its child subtree as the number of itself. Now if the number assigned to the nonterminal corresponding to that of the rhs of production rule is 0, then increment it as a nonterminal of that kind would be required in generating code for the selected instruction.

And finally if the pattern has both left and right children, firstly number both of them. For each nonterminal using the numbers of its left and right subtrees number the current node using Sethi Ullman numbering. The changes in doing so are: if the numbers for this nonterminal are equal in left and right subtree, Sethi Ullman algorithm asks to mark the current node with number left.number + 1, whereas in our case this would be applicable only if the number assigned

to the child subtrees is not 0. When doing this if the number of nonterminals required for this subtree exceeds the available number of nonterminals of this kind then mark the left subtree for store, make its assigned number to be 1 and also decrement the number of nonterminals required at our current node. And finally if the number assigned to the nonterminal corresponding to that of the rhs of production rule is 0, then simply increment it.

This is not completely it. Apart from the instructions covering the tree at a node, there will also be those instructions whose code is to be generated which are basically closure instructions for the above one. They are in the format a : b where a and b are no terminals. In case of numbering for such instructions, the list is covered in a reverse manner, assigning a value of 1 to the non terminal on left hand side if the number assigned to it is 0, otherwise continue.

A similar problem occurs at the time of code generation. The order of evaluation suggested by Sethi Ullman algorithm is for single nonterminal scenario. For multiple nonterminals we need to twerk it a little. Here, I assumed that the order of evaluation of subtrees would matter only if both the right and left subtrees would result in same kind of nonterminal, which is also same as when the pattern of instruction matched at a node of a subtree has same right and left children. In this case the order suggested by Sethi Ullman is followed, else it is assumed that any order would not matter.

Another design decision had to be made to handle cycles while checking for closure over nonterminals. If the instructions which lead to closures for two non terminals are of the kind a : b and b : a each of them with cost zero then we would go in an infinite loop finding the optimal closure over them. To overcome this, a simple hack of reassigning weights of zero weighted edges is done. For all the rules with non zero cost, their costs would be multiplied by 10 and maintained, while for those with zero cost, their cost would be set to 0. Now that we don't have any zero cost rules, we won't be stuck in a cycle.

Work to be done :

Though the basic requirement is fulfilled there are a lot of aspects that can be improved on. A few of them are listed below.
→ Handling various error cases(caused by error in input grammar or error in the input IR tree) both in the code generator generator and the code generator should be done.
→ Input Output of all kinds is currently taking place through standard I/O and this could be done using files and command arguments.
→ The API created for memory locations could be made more rigorous and expanded.
→ When some data is to be stored in a memory location, currently for all cases, the default "mv" instruction is being used, but some flexibility is to be given to the user to modify it as per their requirement.
→ Sethi Ullman algorithm suggests in maintaining an order while allocating registers. This is not implemented yet and can be done further.
→ A detailed study on Sethi Ullman numbering and extending it for multiple non-terminal case to generate optimal code would be very useful.
→Arity and depth of instructions in the initial input machine model are now restricted to 2 and 1 respectively. A more general implementation could be done so that this restriction is relaxed.
→ The output C++ file has some redundant data structures that could be removed.

References :

1. Fraser, C. W., Henry, R. R., and Proebsting, T. A. BURG—Fast optimal instruction selection and tree parsing. SIGPLAN Notices 27, 4 (Apr. 1992), 68–76.
2. C. W. Fraser, D. R. Hanson, and T. A. Proebsting, Engineering a Simple, Efficient Code Generator Generator, ACM Letters on Programming Languages and Systems
3. Ferdinand, C., Seidl, H., and Wilhelm, R. Tree automata for code selection. In Code Generation — Concepts, Tools, Techniques, Proceedings of the International Workshop on Code Generation, Dagstuhl, Germany (May 1991), R. Giegerich and S. L. Graham, Eds., Springer-Verlag, pp. 30–50.
4. http://www.cse.iitb.ac.in/~as/lpcourse/impact/code/code.ps.gz