

Compilation

TP 6: Intermediate representations – CFG and DAG

A. ISOARD & C. ALIAS

credits: G. IOOSS

While the direct code production is quick, the resulting program yields poor performances. In order to provide a common base for optimisation, code selection, scheduling, memory allocation, etc. we will work on an intermediate representation of our program. The objectives of this TP is to define, build and optimize this intermediate form.

Exercise 1. *Pseudo-code, control flow graph*

Download and unzip the file `src_tp6.tar.gz`. Compared to the previous TP, the new classes are the following:

- **Register.h/.cc** replace maintenance of temporaries. We assume that we possess an infinite number of registers and we will take care of their actual allocation (on the stack or inside one of the 4 free registers) at code generation.
- **BitVector.h/.cc** is simply a class to manipulate bit vectors.
- **Code.h** contains an abstract class. It gathers methods common to different classes that implements intermediate representations (`PseudoCode`, `BasicBlock`, ...).
- **PseudoCode.h** represents a single pseudo-instruction. Remark the implementation of the different abstract methods of the `Code` class. The first group of instructions (line 15) corresponds to instructions seen in the lecture. Next groups implements specialized ones (over reserved registers: SP, ARP). Finally, remark the pretty-constructors, alike the ones generating the Digmips code in the previous TP.
- **Cfg.h** implements a control flow graph. Every node of this graph contains a `Code` object (thus, either a `PseudoCode`, or a `BasicBlock`). `live-in[i]` contains the temporaries that are live *just before* executing the `i` node. `live-out[i]` contains the temporaries that are live *just after* the execution of the `i` node. Remark the pretty-constructors, that add one pseudo-code instruction to a global CFG (`cfg` variable, defined inside `parser.ypp`, line 53).

Manip.

- In `main.cc`, build the CFG corresponding to the following code (the temporaries are created using the `new_register()` function):

```
r0 = 1
r1 = 0
r2 = 10
loop :
  cjump r1, GE, r2, end_loop
  r1 = r1 + r0
  r3 = 1
  r4 = r3 + r0
end_loop :
  r4 = r4 + r0
```

- **Display the CFG.** For this, produce the `dotty` representation by calling `cfg->print_dot(cout)`, then compile the result¹.
- **Compute live ranges** using the `do_liveness()` methods of `Cfg`. Displays the resulting CFG.
- **Extract the basic blocks** using the `extract_basic_blocks()` method of `Cfg`. The extraction has to be done after the computation of the live ranges. Displays the resulting CFG.

¹The command is still `dot -Tps cfg.dot > cfg.ps`

Exercise 2. DAG generation

`Dag.h` implements a direct acyclic graph between pseudo-instructions. `node_reg[tmp]` is the root node of the expression computed inside the `temp` temporary. `node_def[noeud]` is the list of temporaries that contains the results computed until node `noeud`.

Manip.

- Open `Dag.cc` and **review the constructor**. It is a variation of the redundancies elimination seen in the lecture, but without hash function. For each instruction `r = r' op r''`, we examine if `r'` and `r''` are associated to existing nodes, and if those nodes have a common ancestor `n`, that executes `op`. If yes, `node_reg[r] := n`. Similar rules exist for the other kinds of operator.
- In `main.cc`, **build the DAG for node 2²**. Display it with `dag->print_dot(cout);`.

Exercise 3. Production of intermediate code

`parser.ypp` is modified to create a new CFG for every function (line 812). Remark the use of the pretty-constructors of `Cfg.h` inside the translation rules. After having opened the function, we compute live ranges (line 827), then extract the basic blocks (line 830) and finally produce a DAG for each basic block (line 834 and following).

Manip.

- In `main.cc`, comment your additions and uncomment the call to the parser. **Try it over `test/test.c`**.

²With the instruction: `Dag* dag = new Dag((BasicBlock*)cfg_bb->node[2]);`, where `cfg_bb` is the CFG with basic blocks built during exercise 1