Compilation

TP 6: Intermediate Representation – SSA form

S. Filip & C. Alias

credits: G. Iooss & A. Isoard

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.

Part I - Production of the 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** replaces 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) during the code generation phase.

- **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 implement 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. The next groups implement specialized ones (over reserved registers: SP, ARP). Finally, remark the pretty-constructors, like 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 alive *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
    jump loop
end_loop :
    r4 = r4 + r0
```

- **Display the CFG**. For this, produce the `dotty` representation by calling `cfg->print_dot(cout)`, then compile the result[1].
- **Compute live ranges** using the `do_liveness()` methods of `Cfg`. Display 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. Display the resulting CFG.

Exercise 2. DAGs

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 operators.
- In `main.cc`, **build the DAG for node 2**[2]. **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`.

Part II - SSA Form

Exercise 1. Warm-up

Consider the following program:

```
a = 3;
b = 2;
while (a < 15) {
  c = 0;
  if (b<6) {
    a = a * 2 ;
    b = b + 1 ;
  } else {
    a = a + 1 ;
    b = b * 2 ;
  }
  c = c + a
}
return a + (b + c);
```

Questions:

- Build by hand the CFG of this program.

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

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

- Translate this program into its SSA form. What is the meaning of each ϕ functions that appear in your program?

Exercise 2. Inter-block optimizations

Consider the following program:

```
a = 1;
b = 2;
e = 3;
while (e<11) {
  if (a<10) {
    c = b*a;
    e = e+1;
  } else {
    d = b;
    t = d;
    c = t*a;
    e = e-3;
  }
  e = e+1;
}
```

Questions:

- Compute the CFG for this program. Put it into SSA form.
- How can you reduce the number of instructions? Is the method of redundant expressions (seen in the lecture) enough?