Lab 5

Code generation with smart IRs

Objective

- Construct a CFG, and the interference graph.
- Allocate registers and produce final code

During the previous lab, you have written a dummy code generator for the Mu language. In this lab the objective is to generate a more efficient LC3 code. Your code is due on TOMUSS on **December, 9th** (code, readme, relevant testfiles, makefile and scripts if any).

First download the archive from the course website.

Todo in this lab:
- Paperwork in Ex1.
- Pertinent tests in Ex2.
- Gen and Kill set initialisations in Ex3.
- Interference graph construction in Ex4.
- Graph coloration call + code generation (printing method) in Ex5.
- Modify register coloring in Ex6 + code generation.

Graphviz install - Nautibus On the Nautibus's machines, `graphviz` is already installed but not `graphviz-dev`, however, we installed it for you, you only have to install the PYTHON binding:

```
pip install --user networkx
pip install --user graphviz
pip install --user pygraphviz
--install-option="--include-path=/home/pers/laure.gonnord/include"
--install-option="--library-path=/home/pers/laure.gonnord/lib/graphviz"
```

Graphviz install: on your machines First install Graphviz, and then the python bindings:

```
apt-get install graphviz graphviz-dev
pip install --user networkx
pip install --user graphviz
pip install --user pygraphviz
--install-option="--include-path=/usr/include/graphviz"
--install-option="--library-path=/usr/lib/graphviz/"
```

5.1 CFG Construction and liveness analysis

EXERCISE #1 ► **By hand**

For the three following examples, draw by hand the 3-address code CFG (where block are individual 3-address code statements with temporaries):

```
x=2;
y=2+x;
z=x+y;
x=7;
```

```
x=2;
if (x < 4)
   x=4;
else
   x=5;
y=x+1;
```

```
x=0;
while (x < 4){
   x=x+1;
}
y=x+3;
z=y+x;
```

EXERCISE #2 ► Play with the CFG construction

We give you the visitor code for the CFG construction. This is an adaptation of the 3-address code generation you made in the previous lab, API calls that generate code, for instance:

```
self._prog.addInstructionNOT(dr,reg)
```

have been replaced by the same code generation inside a new block:

```
self._cfg.append(BlockNOT(dr,reg))
```

All you have to do in this exercise is to test this CFG construction: `Main.py` already contains a call to the function that prints a dot file from the CFG (A `dot` file and its corresponding `pdf` file must be generated next to the `mu` input file).

1. Write simple programs without branches and loops and observe the obtained CFGs.
2. Observe the CFG construction for tests and loops:

Listing 5.1: MyMuVisitor

```
# We have a branch!
blockBRn = self._cfg.append(BlockBR("n",labelfalse))

# We create the true and false branches
blockTrue = blockBRn.append(BlockLabel(labeltrue)) # TRUE case
    comes first
blockFalse = blockBRn.append(BlockLabel(labelfalse))

# TRUE case:
self._cfg.setEnd(blockTrue) # The end of the CFG now points to
    blockTrue
self._cfg.append(...)
endTrue = self._cfg.append(BlockGOTO(labelend)) # When done, we
    jump to labelend

# FALSE case:
self._cfg.setEnd(blockFalse)
endFalse = self._cfg.append(...)

# Finally, we merge the branches
blockLabelend = BlockLABEL(labelend) # Must be the last block
    created
self._cfg.setEnd(endTrue).append(blockLabelend)
self._cfg.setEnd(endFalse).append(blockLabelend)
```

3. Write programs to test the CFG construction for tests, imbricated tests, and while loops.

EXERCISE #3 ► Liveness Analysis

For the liveness analysis, in the `CFG.py` file we give you a method that performs the liveness dataflow analysis on the CFG [1]. However, it doesn't work out-of-the-shell since the $Gen(B)$ and $Kill(B)$ are not initialised.

1. Look at the code and observe the two-level implementation (block, graph).
2. Initialise the $Gen(B)$ and $Kill(B)$ for each block (statement or comment). Be careful to properly handle the following cases:
   `ADD temp1 temp1 12`
   and
   `AND temp1 temp1 0`
   Be careful not to consider constants (imm5) as generated variables.
3. Uncomment the method call in the main PYTHON file.
4. Write pertinent tests to test this dataflow analysis.

[1] "while it is not finished, store the old values for liveness sets, do an iteration (propage information from sons to fathers), decide if its finished"

EXERCISE #4 ► **Interference graph**

We recall that two temporaries are in conflict if they are simultaneously alive after a given instruction, which means:

- There exists a block (an instruction) b and $x, y \in LV_{out}(b)$
- OR There exist a block b such that $x \in LV_{out}(b)$ and y is defined in the block
- OR the converse.

For the two last cases, consider the following list of instructions:

```
y=2
x=1
z=y+1
```

where x is not alive after the `x=1` statement, however x is in conflict with y since we generate the code for `x=1` while y is alive[2].

From the result of the previous exercise, construct the interference graph of your program (each time a pair of temporaries [3] are in conflict, add an edge between them). We give you a non-oriented graph API (`LibGraphes.py`, and an example of use in `ExGraphes.py`) for that. Use the `print_dot` method and relevant tests to validate your code.

5.2 Register allocation and code production

Instead of the iterative algorithm of the course, we will implement the following algorithm for k register allocation:

- Color the graph with $k-3$ colors ($R0$ to $R4$).
- All the other variables will be allocated on the stack. To compute the offset from the stack pointer ($R6$), recolor the subgraph of remaining variables with an infinite number of colors.

Then the code generation:

- For non-spilled variable: replace the temporary with its associated color/register.
- For a spilled variable (say, $temp5$ here):
  `ADD temp6 temp1 temp5`
  becomes (we use $R5$ and $R7$ to make load and stores for spilled variables):
  `LDR R5 R6 #-dec`
  `ADD alloc(temp6) R5 alloc(temp5)`
  (this is why we need to color with $k-3$ registers).

EXERCISE #5 ► **Register Allocation and code production without spilled variables**

Use the algorithm (with k=8) and the coloration method of the `LibGraphes` class to allocate registers (or a place in memory). For this particular exercise, only consider the favorable case when the graph is 5-colorable.

- Call the graph coloring implementation on your interference graph.
- Test on tiny test files that do not need more than 5 physical registers...
- Compare the number of registers that are necessary for your programs and the number of temporaries that where generated (print!).
- Modify the `CFG` print method to be able to replace temporaries with their “color”, ie allocated register.
- Test the generated code on Pennsim !

EXERCISE #6 ► **Allocation of spilled variables**

Now for spilled variables, you have to:

1. Change a bit the coloring method to output the subgraph of variables to spill.
2. Recall this coloring method with an infinite[4] number of colors.
3. Generate code with register-allocated variables as well as spilled variables.
4. Test on Pennsim.

[2] Another solution consists in eliminating dead code before generating the interference graph.

[3] Here is a bit of PYTHON code for enumerating pairs of elements of t: `tpairs = [(t[p1], t[p2]) for p1 in range(len(t)) for p2 in range(p1+1,len(t))]`

[4] 100 is a good approximation of infinite here!

5.3 Bonus: to go further

If you have time, you can choose among the following improvements for your compiler.

EXERCISE #7 ► **Optimise the test process!**
Use the LC3 command line generator and scripts to perform your tests:

`https://highered.mheducation.com/sites/0072467509/student_view0/lc-3_simulator.html`

You can get inspiration from this webpage:

`https://www.cs.colostate.edu/~fsieker/misc/lc3.html`

EXERCISE #8 ► **Big constants**
Find a way to handle numerical constants that are two big to be stored in 5 bits.

EXERCISE #9 ► **Chains**
Find a way to handle log instructions:

- First, constant chains that will be stored in memory.
  ```
  LEA R0, mychain ; in R0 only
  PUTS  #print
  ...
  mychain: .STRINGZ "Hello"
  ```
  prints "Hello.".
- Then, numerical values computed in a given register (you may have to store it somewhere).
- And finally all log instructions.
- If you want to print a char, you must store its (ASCII) value in the $R0$ register and use the `OUT` system call to print it.

EXERCISE #10 ► **Constant propagation**
Design and implement a "constant propagation" dataflow algorithm. Design new examples to test your optimisation.

EXERCISE #11 ► **Multiplication**
Implement a multiplication routine, and produce the code for the multiplication that calls this routine.