

Lab 5

Code generation with smart IRs

Objective

- Construct the CFG.
- Compute live ranges, construct the interference graph.
- Allocate registers and produce final “smart” code.

During the previous lab, you wrote a dummy code generator for the MiniC language. In this lab the objective is to generate a more efficient RISC-V code. We recall you that you have slides in the course to help you.

In the code we give you, there is a mention of for loops, and other string treatments, this is an error this code should not have been given to you (and you do not have to implement these todo). We apologize for this inconvenience.

You will extend your previous code, in the same directory. People in advance are encouraged to keep their current code, students with more difficulties will be provided a working 3 address code generation Visitor named `MiniCCodegen3AVisitor-correct.py` on Tuesday, 21, end of day. Your work is due on To-morrow at the end of the week.

Installations We are going to use `graphviz` for visualization. If it is not already installed (e.g. on your personal machine), install it, for instance with:

```
apt-get install graphviz graphviz-dev
```

You may have to install the following PYTHON packages:

```
pip3 install --user networkx
pip3 install --user graphviz
pip3 install --user pygraphviz \
  --install-option="--include-path=/usr/include/graphviz" \
  --install-option="--library-path=/usr/lib/graphviz/"
```

If the last command errors out complaining about a missing `Python.h`, run:

```
apt-get install python3-dev
```

and then relaunch the command `pip3 install ...`. On the university machines, you might have to update existing already installed packages:

```
pip3 install --user --upgrade networkx graphviz pygraphviz
```

5.1 CFG construction

In class we have presented CFGs with maximal basic blocks. In this lab we will implement CFGs with minimal basic blocks that is CFG with one node per line of code/instruction (even comments).

EXERCISE #1 ► CFG By hand

What are the expected result of the CFG construction from the 3-address code of Lab5 for each of these programs ?

<pre>int n,u,v; n=6; u=12; v=n+u; print_int(v);</pre>	<pre>int x; x=2; if (x < 4) x=4; else x=5; print_int(x)</pre>	<pre>int x; x=0; while (x < 4){ x=x+1; }</pre>
---	--	---

EXERCISE #2 ► CFG Construction

The APiRiscV is able to deal with CFGs. Instructions have a list of predecessors (`self._in`) and successors (`self._out`) and a `RiscVFunction` contains the initial control point (`self._start`) from which we can traverse the graph. This feature allows us to easily construct the CFG of a program.

We give you the construction for all idioms. Each time your Visitor creates a new RISCv instruction, the CFG updates itself automatically: when adding an instruction, it creates an edge between the last instruction (`self._end`) and the instruction to be added.

In this exercise, you only have to understand (look at the API!) and test the provided code.

When ran with `--graphs`, `Main.py` prints the CFG as a PDF file (using the tool “dot”). The file is printed as `<name>.dot.pdf` in the same directory as the source file and opened automatically.

Now you can launch:

```
python3 Main.py --reg-alloc smart --graphs /path/to/example.c
```

1. Test for lists of assignments (for instance `testdataflow/df01.c`) You should see a chain of blocks.
2. Same for boolean expressions, and tests.
3. Same for while loops
4. Propose appropriate examples and draw nice pictures!

5.2 Liveness analysis and Interference graph

For the liveness analysis, we recall the notations. A variable at the left-hand side of an assignment is *killed* by the block. A variable whose value is used in this block (before any assignment) is *generated*.

$$LV_{exit}(\ell) = \begin{cases} \emptyset & \text{if } \ell = \text{final} \\ \bigcup \{LV_{entry}(\ell') \mid (\ell, \ell') \in flow(G)\} & \end{cases}$$

$$LV_{entry}(\ell) = (LV_{exit}(\ell) \setminus kill_{LV}(\ell)) \cup gen_{LV}(\ell)$$

The sets are initialised to \emptyset and computed iteratively, until reaching a fixpoint.

From now on, you have to modify `Allocations.py`

EXERCISE #3 ► Liveness Analysis, Initialisation

Initialise the `Gen(B)` and `Kill(B)` for each kind of instruction (add, let, . . .). This is done by the `set_gen_kill` method of `SmartAllocator`. Be careful to properly handle the following cases:

```
1 addi temp1 temp1 12
```

```
and
```

```
bge temp_1, temp_3, lbl_foo # temp_1 is read from
```

To test/debug this initialisation, the following statements in `Allocations.py` (function `SmartAllocator.run()`) should help you (use with `Main.py --debug`, which sets `debug=True` for you):

```

if debug:
    self.print_gen_kill()

```

As an example, here is the expected initialisation for `testdataflow/df04.c`, obtained by:

```
python3 Main.py --debug --reg-alloc smart testdataflow/df04.c
```

instr 0: comment	kill: {}
gen: {}	
kill: {}	
instr 1: li temp_2, 2	instr 11: beq temp_4, zero, lbl_end_cond_1
gen: {}	gen: {temp_4}
kill: {temp_2}	kill: {}
instr 2: mv temp_1, temp_2	instr 12: li temp_5, 4
gen: {temp_2}	gen: {}
kill: {temp_1}	kill: {temp_5}
instr 3: comment	instr 13: mv temp_1, temp_5
gen: {}	gen: {temp_5}
kill: {}	kill: {temp_1}
instr 6: li temp_3, 4	instr 14: j lbl_end_if_0
gen: {}	gen: {}
kill: {temp_3}	kill: {}
instr 8: li temp_4, 0	instr 5: lbl_end_cond_1
gen: {}	gen: {}
kill: {temp_4}	kill: {}
instr 9: bge temp_1, temp_3,	instr 15: li temp_6, 5
lbl_end_relational_2	gen: {}
gen: {temp_1,temp_3}	kill: {temp_6}
kill: {}	
instr 10: li temp_4, 1	instr 16: mv temp_1, temp_6
gen: {}	gen: {temp_6}
kill: {temp_4}	kill: {temp_1}
instr 7: lbl_end_relational_2	instr 4: lbl_end_if_0
gen: {}	gen: {}
	kill: {}

EXERCISE #4 ► **Liveness Analysis, fixpoint. (Only test!)**

We implemented for you the fixpoint iteration as a method (`run_dataflow_analysis`) in `Allocations.py` “while it is not finished, store the old values, do an iteration, decide if its finished”. The `run_dataflow_analysis` program method makes calls to `dataflow_one_step` instruction methods. The result (live in, live out sets of variables, are stored in `_mapin` and `_mapout` member sets of the `SmartAllocator` class).

All you have to do in this exercise is to check that the results that are obtained with with analysis are correct at least for the examples of the `testdataflow/` directory.

To do so, the following lines should help you (again, using `--debug`) in the same file:

```
mapin, mapout = self.run_dataflow_analysis()
if debug:
    self.print_map_in_out()
```

As an example, here is the expected output for `testdataflow/df04.c`:

```
In: {0: {}, 1: {}, 2: {temp_2}, 3: {temp_1}, 4: {}, 5: {},
    6: {temp_1}, 7: {temp_4}, 8: {temp_1,temp_3},
    9: {temp_1,temp_4,temp_3}, 10: {}, 11: {temp_4},
    12: {}, 13: {temp_5}, 14: {}, 15: {}, 16: {temp_6}, 17: {}, 18: {}}
Out: {0: {}, 1: {temp_2}, 2: {temp_1}, 3: {temp_1}, 4: {}, 5: {},
     6: {temp_1,temp_3}, 7: {temp_4}, 8: {temp_1,temp_4,temp_3},
     9: {temp_4}, 10: {temp_4}, 11: {},
    12: {temp_5}, 13: {}, 14: {}, 15: {temp_6}, 16: {}, 17: {}, 18: {}}
```

EXERCISE #5 ► Interference graph

We recall that two temporaries x, y 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:

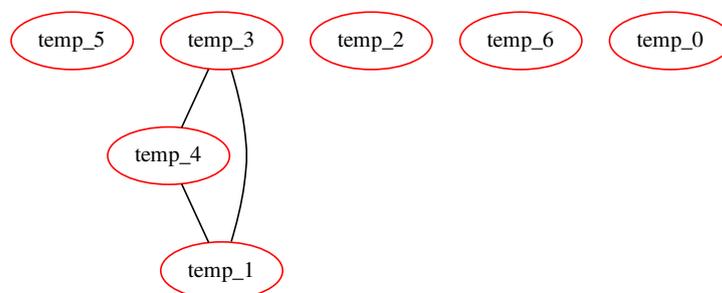
```
y=2
x=1
z=y+1
```

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 alive¹.

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

In this exercise, we care about correctness more than complexity. It is OK to write an $O(n^3)$ algorithm (for each t_1 , for each t_2 , for each control point c , check whether t_1 and t_2 have a conflict).

As an example, here is the conflict graph that should be obtained for `df04.c` (command line as usual):



¹Another solution consists in eliminating dead code before generating the interference graph.

5.3 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 interference graph with an infinite number of colors, using the first ones as much as possible.
- The first $k - 3$ colors will be mapped to registers.
- All the other variables will be allocated on the stack. For each color, we use a memory location according to their coloring number.

Then the memory allocation:

- For non-spilled variable: replace the temporary with its associated color/register.
- For spilled variables: allocate in memory.

Some help:

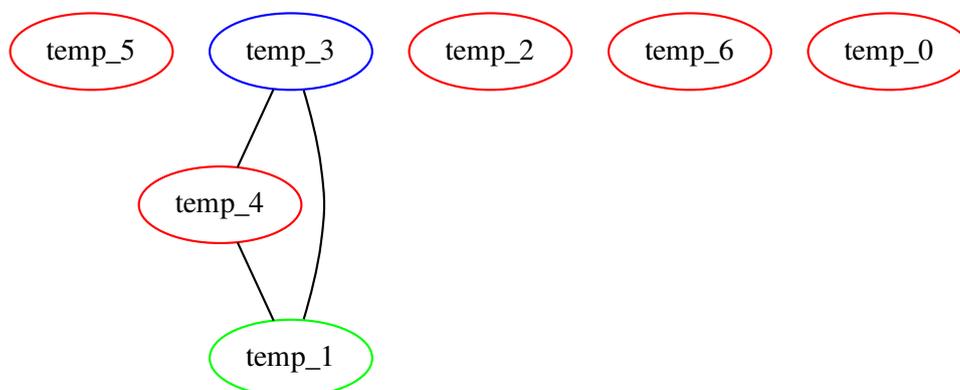
- GP_REGS is an array of registers available for the register allocator.
- An element of type Register can be obtained from a given register color with the helper function GP_REGS[coloringreg[xxx]], and for offsets you have a constructor Offset(SP, xxx) (all in Operands.py).
- Be careful with types when dealing with the graph. As the comment in Allocations.py states, self._igraph contains only elements of type string, while the alloc_dict map given to self._pool.set_temp_allocation() must have Temporary objects as keys. There is no easy way to retrieve a Temporary object from its name, but it is easy to get the name as a string from a Temporary: just use str(). The easiest way to build alloc_dict is probably to iterate over all the temporaries of the program (available in self._pool._all_temps), and for each temporary check the corresponding color to associate it to the right register or memory location in alloc_dict.

EXERCISE #6 ► Smart Register Allocation: implement!

Use the algorithm and the coloration method of the LibGraphes class to allocate registers (or a place in memory). For that, you have to complete the program method SmartAllocator.run() (in file Allocations.py). Comments will help you design this (non trivial) function.² The allocation is followed by statement rewriting, like in previous lab. You need to implement it in Allocations.py (replace_smart): it is very similar to the previous lab's version, but you have to deal with both memory locations and registers in the same function.

Validate your allocation on tiny well chosen test files (especially tests that augment the register pressure) and all the benchmarks of the previous lab. We adapted the previous script for that.

On the df04.c example, the graph coloring succeeds with:



²it seems that we also gave you the interfere function this year, this was not supposed to happen, ...

EXERCISE #7 ▶ Massive tests

Comment out all the `print_dot` instructions, `debug`, ... and test on all test files you have. **In particular, we do not want any pdf file to be opened when we will use make tests on your delivered code.**

EXERCISE #8 ▶ Lab delivery

Make a clean archive with `README.md`, your test files, ... (same instructions as before) and put it on TOMUSS before Sunday January, 26th, 23:59.

`http://tomuss.univ-lyon1.fr`