Lab 5
Code generation with smart IRs

Objective

- Understand the CFG construction.
- Compute live ranges, construct the interference graph.
- Allocate registers and produce final "smart" code.
- Two sessions for this lab.
- **Lab delivery: with Lab4, all together, on Chamilo. Instructions and deadline on Chamilo**. Do not forget massive tests (on each step of the smart allocation.)

5.1 Preliminaries

(save your lab4 somewhere before doing anything else). First, get the latest version of the skeleton with:

```
git pull
```

If you get painful conflicts, you may re-clone the whole repository (see lab 1).

Be careful to not crash your last Lab (especially the file : `TP04/MiniCCodeGen3AVisitor.py`). Some files should be updated in this directory.

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 RISCV code.

You will extend your previous code (in the second part of the lab, see Section 5.4, in the same `MiniC` project, but in the `TP05/` subdirectory

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:

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

You may have to install the following PYTHON packages:

```
python3 -m pip install --user networkx
python3 -m pip install --user graphviz
python3 -m pip install --user pygraphviz
```

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

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

and then relaunch the command `python3 -m pip install ...` On the Esisar machines, you might have to update existing already installed packages:

```
python3 -m pip install --user --upgrade networkx graphviz pygraphviz
```

5.2 CFG construction

The CFG construction is given, you only have to understand how it is implemented. (The code to observe is in the `MiniC/TP04/` directory).

During class we presented Control Flow Graphs with maximal basic blocks. In this section we will read and understand how this CFG generation is implemented.

During this phase, the linear code produced in the 3-adress code generation produces a graph.

EXERCISE #1 ► **CFG By hand**
What are the expected result of the CFG construction for the each of these programs?

Listing 5.1: `df01.c`

```
int n,u,v;
n=6;
u=12;
v=n+u;
print_int(v);
```

Listing 5.2: `df04.c`

```
int x;
x=2;
if (x < 4)
   x=4;
else
   x=5;
print_int(x)
```

Listing 5.3: `df05.c`

```
int x;
x=0;
while (x < 4){
   x=x+1;
}
```

5.2.1 Understand provided code

In `TP04/BuildCFG.py` we give you the entire code for the CFG construction (whose datastructure is defined in the Lib).

EXERCISE #2 ► **Finding the leaders**
In the course on intermediate representations, we have defined the notion of *basic blocks* and *leaders*, which designate the indices of the instructions starting a block. We define the `_find_leaders` procedure as taking the list of instructions and returning a list of leaders The list of indices `leaders` should have the following properties:

- `leaders[i]` is the starting instruction of the i^{th} block.
- Each interval `leaders[i]` to `leaders[i+1]-1` delimits the instructions of a block.
- We have `leaders[0]=0` and `leaders[-1]=len(instructions)`.
- There are no duplicated indices in the list.

Compute the leaders by hand on the following example.

```
0 subi temp2, temp2, 4
1 beq temp2, zero, lelse1
2 li temp4, 7
3 mv temp1, temp4
4 jump lendif1
5 lelse1:
6 addi temp3, temp2, 1
7 mv temp1, temp3
8 lendif1:
```

EXERCISE #3 ► **Understand CFG Construction (file `TP04/BuildCFG.py`)**
The `CFG` file contains all the utilities related to Control Flow Graphs:

- the `Block` class, representing a basic block,
- the `CFG` class, representing a complete function in CFG form.

`Blocks` have a list of predecessors (`self._in`) and successors (`self._out`) and a `CFG` contains the initial control point (`self._start`) from which we can traverse the graph. This feature allows us to construct the CFG of a program.

The procedure to build the CFG is split into several pieces. The `__init__` function builds the class and sets all utility counters. The `_find_leaders` function returns a list of all the leaders. The `_add_blocks` function populates the control flow graph with the blocks extracted using the list of leaders.

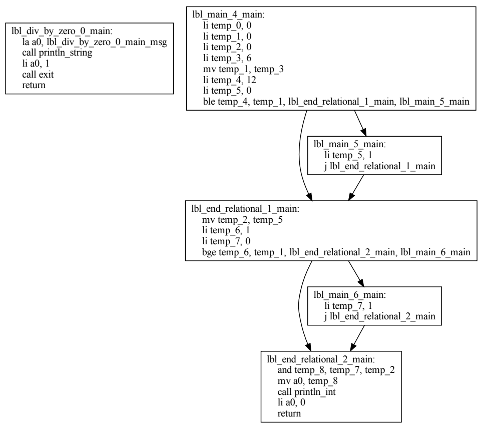
- Is the `_find_leaders` implementation conform to the one of the course?
- How is the chaining realized in the case of forward reference ? backward reference ?

EXERCISE #4 ► **Check and test your CFGs**
When run with `--graphs`, `MiniCC.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.

Check the output for all files in `TP04/tests/provided/dataflow/` with the following command line:

```
python3 MiniCC.py --mode codegen-cfg --reg-alloc naive \
--graphs TP04/tests/provided/dataflow/df02.c
```

For example, the CFG for `df02.c` should look like:



```
lbl_div_by_zero_0_main:
    la a0, lbl_div_by_zero_0_main_msg
    call println_string
    li a0, 1
    call exit
    return
```

```
lbl_main_4_main:
    li temp_0, 0
    li temp_1, 0
    li temp_2, 0
    li temp_3, 6
    mv temp_1, temp_3
    li temp_4, 12
    li temp_5, 0
    ble temp_4, temp_1, lbl_end_relational_1_main, lbl_main_5_main
```

```
lbl_main_5_main:
    li temp_5, 1
    j lbl_end_relational_1_main
```

```
lbl_end_relational_1_main:
    mv temp_2, temp_5
    li temp_6, 1
    li temp_7, 0
    bge temp_6, temp_1, lbl_end_relational_2_main, lbl_main_6_main
```

```
lbl_main_6_main:
    li temp_7, 1
    j lbl_end_relational_2_main
```

```
lbl_end_relational_2_main:
    and temp_8, temp_7, temp_2
    mv a0, temp_8
    call println_int
    li a0, 0
    return
```

Observe how straight-line code, if statements, while loops are converted.

Important remark Note that register allocation does not affect the CFG printed by `--graphs`, as it is output before register allocation. In the rest of the lab, your compiler should do the allocation on the CFG, and all the tests from the previous lab should still pass (with `-reg-alloc all-in-mem` option).

5.3 Liveness analysis and Interference graph

Be prepared! From now on, we work in `MiniC/TP05/` directory. You should ignore the file LivenessSSA.py, which will not be used this year.

In order to implement the "smart allocation", you have to modify `MiniC/TP05/SmartAllocation.py`.You already used `NaiveAllocator` and `AllInMemAllocator` in the previous lab (the mapping from temporary to physical register or memory location was provided to you, and you had to modify the 3 address code to take this mapping into account). We well now write the `SmartAllocator`, that maps temporaries to physical registers in an optimized way, and use memory (spilling) only when necessary. Read the body of `SmartAllocator.prepare()`, that gives the main steps of the allocation:

- liveness dataflow analysis,
- conflict graph,
- graph coloring
- and finally 3 address code modification to get the final executable.

Similarly to previous lab, `SmartAllocator.rewriteCode()` then applies the allocation to the generated code. To help you with debugging, a `raise NotImplementedError` statement in `prepare` stops the execution. Move it down as you progress in the lab, and remove it completely when you are done.

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*.

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

$$LV_{entry}(\ell) = \left(LV_{exit}(\ell) \backslash kill_{LV}(\ell)\right) \cup gen_{LV}(\ell)$$

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

EXERCISE #5 ▸ Liveness Analysis, Implement the Initialisation

In this exercice, you have to complete the method `set_gen_kill()` of the `CFG.Block` class. This method is called for each block by `LivenessDataFlow.set_gen_kill()` and initialises the Gen(B) and Kill(B) sets for each basic block (add, let, ...) in the program. To help you, each instruction provides a `.defined()` (respectively `.used()`) method that returns the set of temporaries assigned (respectively used) in the instruction. Be careful to properly handle the following cases:

```
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 `SmartAllocation.py` (in `SmartAllocator.prepare()`) should help you (use with `MiniCC.py --debug`, which sets `debug=True` for you):

```
    if self._debug:
        self._liveness.print_gen_kill()
```

As an example, once initialization is implemented, running the command (without –graphs!)

```
python3 MiniCC.py --mode codegen-cfg --debug  --reg-alloc smart TP05/tests/provided/dataflow/df02.c
```

should give the expected initialisation for `TP05/tests/provided/dataflow/df02.c`:

```
Dataflow Analysis, Initialisation
block lbl_div_by_zero_0_main : 0
gen: {}
kill: {}

block lbl_end_relational_2_main : 1
gen: {temp_7,temp_2}
kill: {temp_8}

block lbl_main_6_main : 2
gen: {}
kill: {temp_7}

block lbl_end_relational_1_main : 3
gen: {temp_1,temp_5}
kill: {temp_6,temp_7,temp_2}

block lbl_main_5_main : 4
gen: {}
kill: {temp_5}

block lbl_main_4_main : 5
gen: {}
kill: {temp_0,temp_2,temp_3,temp_4,temp_5,temp_1}
```

Note:

- If you ran the compiler with `--graphs`, the CFG should be displayed in a graphical window. The names of basic blocks in the text output correspond to the leading label in the blocks shown in the graphical window.
- Only temporaries are shown. Block `lbl_div_by_zero_0_main` uses physical registers but no temporary, hence `gen` and `kill` sets are empty.

EXERCISE #6 ► **Liveness Analysis, fixpoint. (Only test!)**

We implemented for you the fixpoint iteration as a method (`run_dataflow_analysis`) in `SmartAllocation.py` "while it is not finished, store the old values, do an iteration, decide if its finished". The `run_dataflow_analysis` 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).

What you have to do in this exercice is to check that the results that are obtained with with analysis are correct at least for the examples of the `TP05/tests/provided/dataflow/` directory.

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

```
self._liveness.run() #update mapin, mapout
if self._debug:
    self.print_map_in_out()
```

As an example, here is the expected output for `TP05/tests/provided/dataflow/df02.c`:

```
Dataflow Analysis
finished in 2 iterations
In: {lbl_div_by_zero_0_main: {},
lbl_end_relational_2_main: {temp_7,temp_2},
lbl_main_6_main: {temp_2},
lbl_end_relational_1_main: {temp_1,temp_5},
lbl_main_5_main: {temp_1}, lbl_main_4_main: {}}

Out: {lbl_div_by_zero_0_main: {}, lbl_end_relational_2_main: {},
 lbl_main_6_main: {temp_7,temp_2},
lbl_end_relational_1_main: {temp_7,temp_2},
lbl_main_5_main: {temp_1,temp_5},
lbl_main_4_main: {temp_1,temp_5}}
```

EXERCISE #7 ► **Interference graph: Implement interfere**

The interference graph contains an edge (x, y) if temporaries x and y are in conflict. It is built by `SmartAllocator.build_interference_graph`, that calls the `interfere` method.

We recall that two temporaries x, y are in conflict if they are simultaneously alive after a given **instruction**[1], which means:

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

To understand why the last two cases are needed, consider the following list of instructions:

```
// Can x and y be mapped to the same place? Obviously not.
y=2
x=1 // dead assignment
println_int(y)
x=666 // x not live before this
```

The live ranges of x and y are disjoint, however x is in conflict with y since we generate the code for `x=1` while y is alive[2].

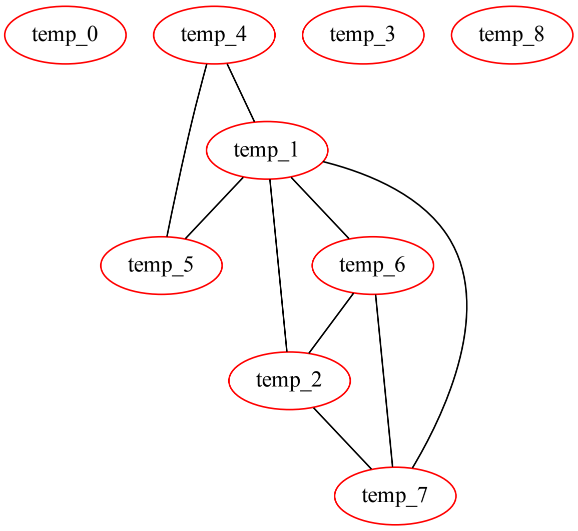
From the result of the previous exercise, the code in `LivenessDataFlow` builds a `self._liveness._liveout` field of type `Dict[Instruction, Set[Operand]]` that gives the set of temporaries that are live after a given **instruction**[3]. Use this data to construct the interference graph (the job is done by function `build_interference_graph`) of your program. We give you a undirected graph API

[1] yes, an instruction, not a block
[2] Another solution consists in eliminating dead code before generating the interference graph.
[3] The dataflow analysis is indeed performed at the block level, and a postanalysis is run afterwards to get the information at the instruction level.

(`Lib/Graphes.py`) for that. Use the `print_dot` (it should be called with the `-graph` option) 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 `df02.c` (command line as usual):



5.4 “Smart’ register allocation and code production

We will implement the following algorithm for register allocation:

- Color the interference graph with an infinite number of colors, using the first ones as much as possible.
- The first `len(GP_REGS)` 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 3 address code modification:

- For non-spilled variable: replace the temporary with its associated color/register, as we did for the naive allocator.
- For spilled variables: add `ld` / `sd` statements as needed and replace the temporary with one of `s1`, `s2`, `s3` as we did for the “all in mem” allocator.

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]]`, where `coloringreg` is graph coloring returned by the `.color()` function, and for offsets you have the method `self._function_code.new_offset()` that returns a fresh one (all in `Operands.py`).
- The easiest way to build `alloc_dict` is probably to iterate over all the temporaries of the program (available in `self._function_code._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 #8 ► **Smart Register Allocation: implement!**
In this exercice, you have first to complete method `SmartAllocation.smart_alloc()` to perform an allocation based on a graph coloring. The purpose of this method is to allocate a physical register or a memory

location for each temporary in the program. Next, you will have to complete the function `replace_smart()` that replaces the temporary operands of a given instruction according to the allocation computed by `smart_alloc()`.

Use the algorithm and the coloration method of the `LibGraphes` class to allocate registers (or a memory location) in `smart_alloc()`. Comments will help you design this (non trivial) function. The allocation is followed by statement rewriting, like in previous lab. You need to implement it in `SmartAllocation.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.

Each color+shape pair indicates a different location. Temp numbering and coloring may be different in your output.

EXERCISE #9 ► **Massive tests**
Debug, . . . and test on all test files you have (test cases in `TP05/tests/students`, please). The testsuite must not open any PDF file (this is normally OK since it launches the compiler without the `--debug` option).

5.5 Final delivery

Instructions will be given on Chamilo. Lab 4 and Lab 5 will be delivered at the same time, and the two commands `make test-lab4` and `make test-codegen` should perform correctly.