Lab 5
Optimisations under SSA form

Objective

- Optimising the CFG under SSA form created by the compilation, before any allocation and before liveness computation.
- The algorithm we will implement is called *Conditional Constant Propagation*.

During this Lab, you will only modify the file `OptimSSA.py`.

We follow an optimisation algorithm called *Conditional Constant Propagation* which was seen during the course[1]. The principle is to first analyse the program using a "simplified" execution to obtain invariants regarding assignments of variables. For instance, 'is the variable x always assigned the constant value c ?' Once these invariants are known, we simplify our program by removing the variable x and replacing all of its uses by the constant c. Such invariants also allow to find blocks which can never be executed (e.g. when the condition of a branching is always the same), whence blocks that can be removed from the CFG.

As this is an optimisation, every test you wrote in one of the previous labs should still work at the end!

5.1 Check your Lab 5

To begin this lab, you need a functional Lab 5 (where the transformation of a CFG to its SSA form is done). Make sure it is working correctly with `make tests-codegen SSA=1` (to check for typing mistakes and run the test suite).

If your implementation is not working, feel free to use the corrections given in `CFG-correct.py` and `SSA-correct.py`.

5.2 Description of the algorithm

As written in the introduction, we want to find variables which stay constant through all the execution of the program. Typical cases where this happens is when we have statements of the form `if (debug) {...}` where `debug` is constant ; it would be a shame to let such statements block us from using all available registers.

The *SSA Conditional Constant Propagation* will associate a value $\mathcal{V}[v]$ to each variable (i.e., temporary) v of the program:

- $\mathcal{V}[v] = \perp$ if we have seen no evidence that v is defined.
- $\mathcal{V}[v] = c$ if we have seen evidence that v is only assigned the value c.
- $\mathcal{V}[v] = \top$ if we have seen evidence that v is assigned different or unpredictable values ($v = f(\ldots)$, read from memory, $\ldots$).

Machine integers, $\perp$ and $\top$ form a *lattice*. A *lattice* is a partially ordered ($\sqsubseteq$) set where each pair (x, y) of elements has:

- a sup: $x \sqcup y$, also named join.
- an inf: $x \sqcap y$, also named meet.

The partial order is defined as follows (with x above y and linked with an edge if $y \sqsubseteq x$):

[1] Inspired by *Modern compiler implementation in ML* by Andrew W. Appel

$$\begin{tikzcd}
 & & & \top \arrow[dll, dash] \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] \arrow[drr, dash] \arrow[drrrr, dash] & & & & & \\
\cdots & -2 & -1 & 0 & 1 & 2 & \cdots & 42 & \cdots \\
 & & & \bot \arrow[ull, dash] \arrow[ul, dash] \arrow[u, dash] \arrow[ur, dash] \arrow[urr, dash] \arrow[urrrr, dash] & & & & &
\end{tikzcd}$$

During an execution of our optimisation, which will update $\mathcal{V}$, values can only go up in this lattice (for instance we can have $\mathcal{V}[v] = \bot$, then $\mathcal{V}[v] = 4$ and then $\mathcal{V}[v] = \top$, but never $\mathcal{V}[v] = 4$ and then $\mathcal{V}[v] = -3$ or $\mathcal{V}[v] = \bot$).

We also track the executability of each edge of the CFG, to see from which branches a ϕ-node can take its values from. It is defined as follows:

- $\mathcal{E}[B,C] = False$ if we have seen no evidence that an execution can go from block B to C following this edge.
- $\mathcal{E}[B,C] = True$ if we have jumped from block B to C in our execution.

We add an initial edge in the CFG from nothing (i.e. `None`) to the starting block.

This notion gives rise to a notion of executability for blocks (resp. instructions): a block C is executable if one of its input edges is executable (resp. if it is inside an executable block).

The optimisation does not assume an edge (and so a block) can be executed until there is evidence that it can be, and does not assume a variable is non-constant until there is evidence, and so on. We thus start with $\mathcal{V}$ constant to $\bot$ and $\mathcal{E}$ to $False$.

Analysing the program We now describe how to propagate information in a execution to update these tables $\mathcal{V}$ and $\mathcal{E}$.

1. Variables v from outside the function (parameters and function call) are of unknown value ($\top$). In our case, these are exactly registers of A.
2. The initial edge to the `start` block, the edge (`None`, `start`) is executable.
3. For any executable block B with only one successor C, the edge (B,C) is executable.
4. For any executable assignment $v \leftarrow \mathrm{op}(x,y)$, we update the value of v: $\mathcal{V}[v] = \mathrm{op}(\mathcal{V}[x], \mathcal{V}[y])$.
5. For any executable assignment $v \leftarrow \phi(x_1,\ldots,x_n)$, we update the value of v: $\mathcal{V}[v] = \mathcal{V}[x_1] \sqcap \cdots \sqcap \mathcal{V}[x_n]$.
6. For any executable branching $b\ x\ C\ y$ (with x and y variables and C a comparison operand), in a block B to blocks B_1 and B_2, set $\mathcal{E}[B,B_1] =$ `True` and/or $\mathcal{E}[B,B_2] =$ `True` depending on the evaluation of $\mathcal{V}[x]\ C\ \mathcal{V}[y]$.

Using only executable instructions allows us to ignore instructions in an inexecutable block, and to make ϕ nodes ignore any operand coming from an inexecutable edge. We run the 6 steps repeatedly until reaching a fixpoint[2].

Simplifying the program Once we have finished the analysis and $\mathcal{V}$ and $\mathcal{E}$ computed, we use them to optimise as following:

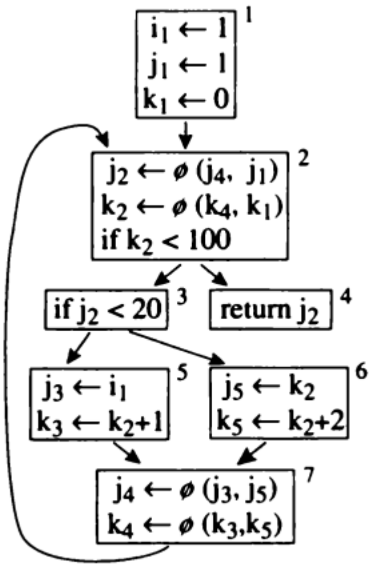
a. Whenever $\mathcal{V}[v] = c$ is a constant, we substitute v by c and delete the assignment of v. Initially, we precede each instruction i using v by an instruction `li temp c`, and replace the usage of v in i by the use of `temp`. Be mindful of the placement of the new `li` when v is used in a ϕ node!

b. Whenever $\mathcal{E}[B,C] = False$, we delete this edge (and the associated jump) from the CFG

c. Whenever a block B is not executable, we delete this block from the CFG.

[2] A *flag* is used in the code to check if we changed a value during an iteration.

EXERCISE #1 ► **Optimising manually**



Run by hand the analysis on the example above (also provided in `TP05c\tests\provided\example_lecture.c`). Using the results of the analysis, simplify the program.

EXERCISE #2 ► **Termination**
Does this analysis always terminate? Estimate its maximum number of iterations.

5.3 Implementation

A skeleton of the optimisation is given in the file `OptimSSA.py`. The lattice with integers, $\top$ and $\bot$ is already defined thanks to `LATTICE_VALUE`. Likewise, the valueness $\mathcal{V}$ and the executability $\mathcal{E}$ are defined. Furthermore, evaluation functions for arithmetic and booleans operations are provided, to compute $\mathrm{op}(v_1, v_2)$ easily.

What remains to do is to implement steps 4, 5, 6 and a of the algorithm.

EXERCISE #3 ► **Understand the code**
Before completing the implementation, read the existing code and understand how it is structured: many helper functions are defined, some of which will be useful for the exercises that follow.

EXERCISE #4 ► **Step 4**
In `propagate_in()`, implement analysis step 4. (The correction does so in 3 lines.)

After this, if you lauch your optimisation on `example_ex1.c` with the `--debug` option (with the command `python3 MiniCC.py --reg-alloc none --ssa --ssa-optim --debug TP05c/tests/provided/example_ex1.c` for instance) you should find that j is constant. Therefore, you last output should be (up to renaming temporaries):

```
Valueness:
temp_4: 0
temp_5: 0
temp_6: 1
temp_7: 1
temp_8: 1
temp_9: 0
a0: Lattice.Top
a1: Lattice.Top
a2: Lattice.Top
```

```
a3: Lattice.Top
a4: Lattice.Top
a5: Lattice.Top
a6: Lattice.Top
a7: Lattice.Top
```

EXERCISE #5 ► **Step a**

In `replaceInstruction`, finish the implementation of rewriting step a: whenever $\mathcal{V}[v] = c$ is a constant, substitute v by c by preceding each instruction (not phi node) i using v by an instruction `li temp c`, and replace the usage of v in i by the use of `temp`. (The correction does it in 8 lines.)

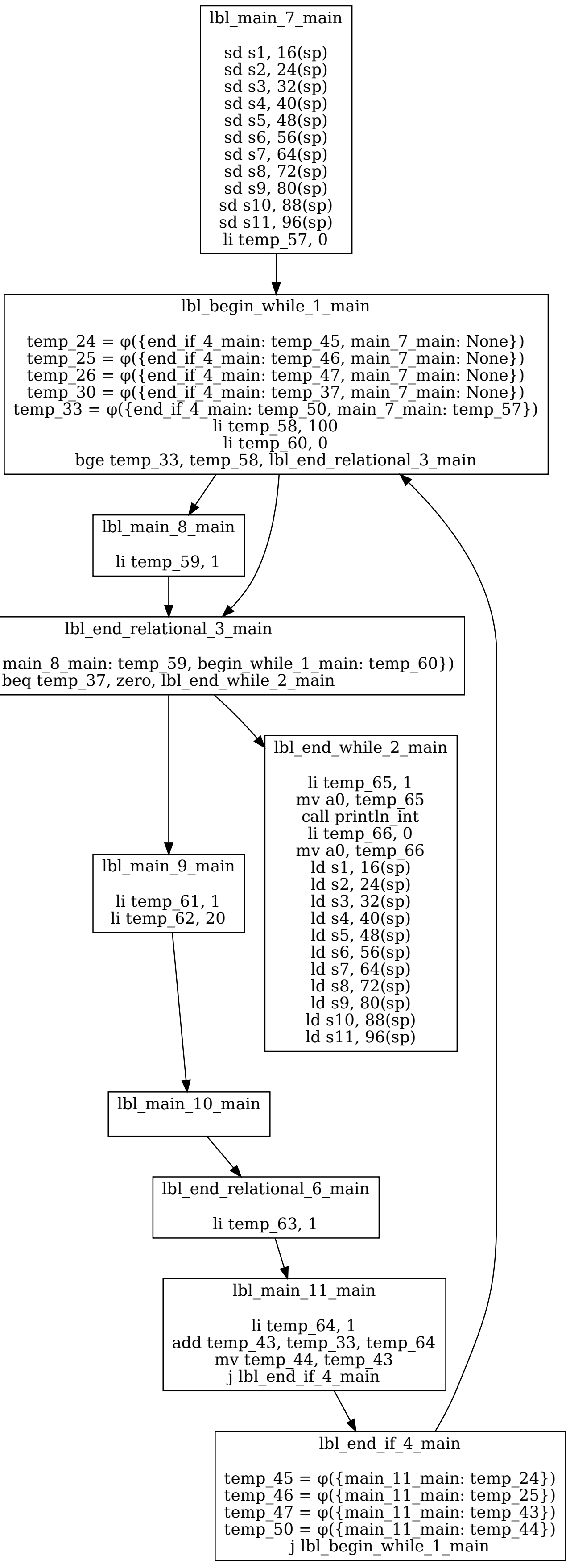
After this, if you lauch your optimisation on `example_ex1.c` with the `--ssa-graphs` option, you should obtain in the only block of `example_ex1.main.optimssa.dot.pdf` (up to renaming temporaries):

```
...
li temp_10, 1
mv a0, temp_10
call println_int
li temp_11, 0
mv a0, temp_11
...
```

EXERCISE #6 ► **Steps 5 & 6**

In `propagate_in()`, implement analysis steps 5 and 6. (The correction implements step 5 in 3 lines and step 6 in 20.)

After this, if you lauch your optimisation on `example_lecture.c` with the `--ssa-graphs` option your optimisation should yield:



```
lbl_main_7_main

sd s1, 16(sp)
sd s2, 24(sp)
sd s3, 32(sp)
sd s4, 40(sp)
sd s5, 48(sp)
sd s6, 56(sp)
sd s7, 64(sp)
sd s8, 72(sp)
sd s9, 80(sp)
sd s10, 88(sp)
sd s11, 96(sp)
li temp_57, 0
```

```
lbl_begin_while_1_main

temp_24 = φ({end_if_4_main: temp_45, main_7_main: None})
temp_25 = φ({end_if_4_main: temp_46, main_7_main: None})
temp_26 = φ({end_if_4_main: temp_47, main_7_main: None})
temp_30 = φ({end_if_4_main: temp_37, main_7_main: None})
temp_33 = φ({end_if_4_main: temp_50, main_7_main: temp_57})
li temp_58, 100
li temp_60, 0
bge temp_33, temp_58, lbl_end_relational_3_main
```

```
lbl_main_8_main

li temp_59, 1
```

```
lbl_end_relational_3_main

temp_37 = φ({main_8_main: temp_59, begin_while_1_main: temp_60})
beq temp_37, zero, lbl_end_while_2_main
```

```
lbl_end_while_2_main

li temp_65, 1
mv a0, temp_65
call println_int
li temp_66, 0
mv a0, temp_66
ld s1, 16(sp)
ld s2, 24(sp)
ld s3, 32(sp)
ld s4, 40(sp)
ld s5, 48(sp)
ld s6, 56(sp)
ld s7, 64(sp)
ld s8, 72(sp)
ld s9, 80(sp)
ld s10, 88(sp)
ld s11, 96(sp)
```

```
lbl_main_9_main

li temp_61, 1
li temp_62, 20
```

```
lbl_main_10_main
```

```
lbl_end_relational_6_main

li temp_63, 1
```

```
lbl_main_11_main

li temp_64, 1
add temp_43, temp_33, temp_64
mv temp_44, temp_43
j lbl_end_if_4_main
```

```
lbl_end_if_4_main

temp_45 = φ({main_11_main: temp_24})
temp_46 = φ({main_11_main: temp_25})
temp_47 = φ({main_11_main: temp_43})
temp_50 = φ({main_11_main: temp_44})
j lbl_begin_while_1_main
```

EXERCISE #7 ► **Massive tests**

Once your optimisation completed, check that every test still pass. For that purpose, `make tests-codegen SSA_OPTIM=1` runs your compiler on the whole test suite with the optimisation active. If you did implement the all-in-mem allocator but not the smart allocator (the correction of SSA we gave you does not support the smart allocator), you can use `make tests-notsmart SSA_OPTIM=1` instead. If you implemented neither of them, you can use `make tests-naive SSA_OPTIM=1`.

Make sure no debug output (`dump`...) is printed when options `--debug`, `--graphs` and `--ssa-graphs` are not given.

Do not forget to check that what you wrote in the file `OptimSSA.py` is covered. To see detailed information on coverage, open `htmlcov/TP05c_OptimSSA_py.html` in your web browser after a run of the test suite.

EXERCISE #8 ► **Optimise the use of constants (Extension)**

With our current implementation, we may have instructions of the form:

```
li t0 5
add t2 t1 t0
```

This code could be further optimised into:

```
addi t2 t1 5
```

You can make such optimisations, when one of the arguments of an `add` is constant, and this constant is not too big (as the value of an immediate given to an `addi` is limited). You can also do the same with `andi` and `ori` (and this time you cannot have too big values, for booleans are either 0 or 1 in MiniC programs).