Lab 5
Smart IRs, part C: 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`, in the folder `/MiniC/TPoptim/`.

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 test-codegen MODE=codegen-ssa` (to check for typing mistakes and run the test suite).

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:
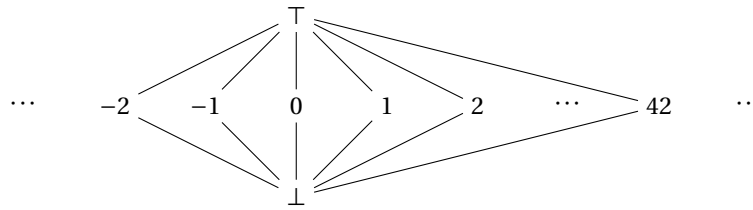
- $\mathcal{V}[v] = \bot$ if we have seen no evidence that v is defined.

- $\mathcal{V}[v] = c$ 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 ($v = f(\dots)$, read from memory, $\dots$).

Machine integers, $\bot$ 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.

$$\cdots \quad -2 \quad -1 \quad 0 \quad 1 \quad 2 \quad \cdots \quad 42 \quad \cdots$$

(with $\top$ above and $\bot$ below, connected to all)

During an execution of our optimisation, which will update $\mathcal{V}$, values can only go up in this lattice (for instance we can have $\mathcal{V}[v] = \bot$ at first, $\mathcal{V}[v] = 4$ later, and $\mathcal{V}[v] = \top$ in the end, 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 block C following this edge.

- $\mathcal{E}[B,C] = True$ if we have jumped from block B to block C in our execution.

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

This notion of executability for edges 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. an instruction is executable if it is inside an executable block).

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

Analysing the program   We now describe how to propagate information in an 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: $\mathcal{V}[v] = \mathcal{V}[x_1] \sqcup \cdots \sqcup \mathcal{V}[x_n]$.

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, with $\mathcal{V}$ and $\mathcal{E}$ computed, we use them to optimise as following:

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

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

---

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

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

E<small>XERCISE</small> #1 ► **Optimising manually**
Run by hand the analysis on the following example:
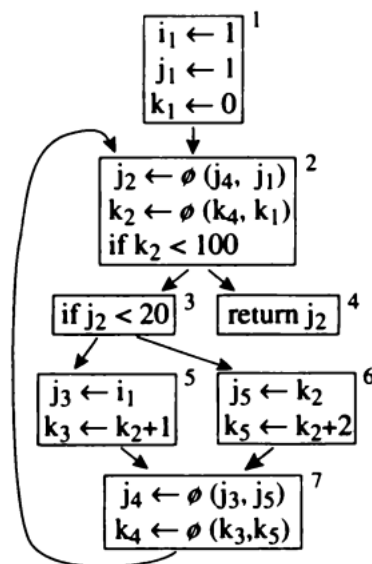
(also provided in `TPoptim\tests\provided\example_lecture.c`)

Using the results of the analysis, simplify the program.

E<small>XERCISE</small> #2 ► **Termination**
Does this analysis always terminate? If it is the case, what is its maximum number of iterations?

5.3 Implementation

A skeleton of the optimisation is given in the file `OptimSSA.py`. To launch the main file `MiniCC.py` with the optimisation, use `--mode=codegen-optim`. 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 $op(v_1, v_2)$ easily.

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

E<small>XERCISE</small> #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.

E<small>XERCISE</small> #4 ► **Step 4**
In `propagate_in()`, the main helper function for `compute`, implement analysis step 4. (The correction does so in 4 lines.)

After this, if you launch your optimisation on `no_loop_constant.c` with the `--debug` option (with the command

```
python3 MiniCC.py --mode=codegen-optim --reg-alloc=none --debug \
TPoptim/tests/provided/no_loop_constant.c
```

for instance) you should find that j is constant. Therefore, your last output should be (up to renaming of temporaries):

```
Valueness:
temp_3: 0
temp_4: 0
```

```
temp_5: 1
temp_6: 1
temp_7: 1
a0: Lattice.Top
a1: Lattice.Top
a2: Lattice.Top
a3: Lattice.Top
a4: Lattice.Top
a5: Lattice.Top
a6: Lattice.Top
a7: Lattice.Top
Executability:
 -> lbl_main_2_main: True
```

EXERCISE #5 ▶ **Step b**

Finish the implementation of "Simplifying step b": whenever $\mathcal{V}[v] = c$ is a constant, substitute v by c by preceding each instruction i using v by an instruction `li temp, c`, and replace the usage of v in i by the use of `temp`.

The solution for `replacePhi` is given. You have to implement substitutions for instructions that are in the body of a block in `replaceInstruction`, and for terminators in `replaceTerminator` (the correction does it in 7 lines per function).

After this, if you launch your optimisation on `no_loop_constant.c` with the `--ssa-graphs` option, you should obtain a CFG (in a file called `no_loop_constant.main.optimssa.dot.pdf`) which is the one on Figure 5.1 (up to renaming temporaries).

```
lbl_main_2_main:
    li temp_8, 1
    mv a0, temp_8
    call println_int
    li a0, 0
    return
```

Figure 5.1: CFG of `no_loop_constant.c` after optimisation

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 16.)

After this, if you launch your optimisation on `example_lecture.c` with the `--ssa-graphs` option your optimisation should yield a CFG (in a file called `example_lecture.main.optimssa.dot.pdf`) looking like the one on Figure 5.2.
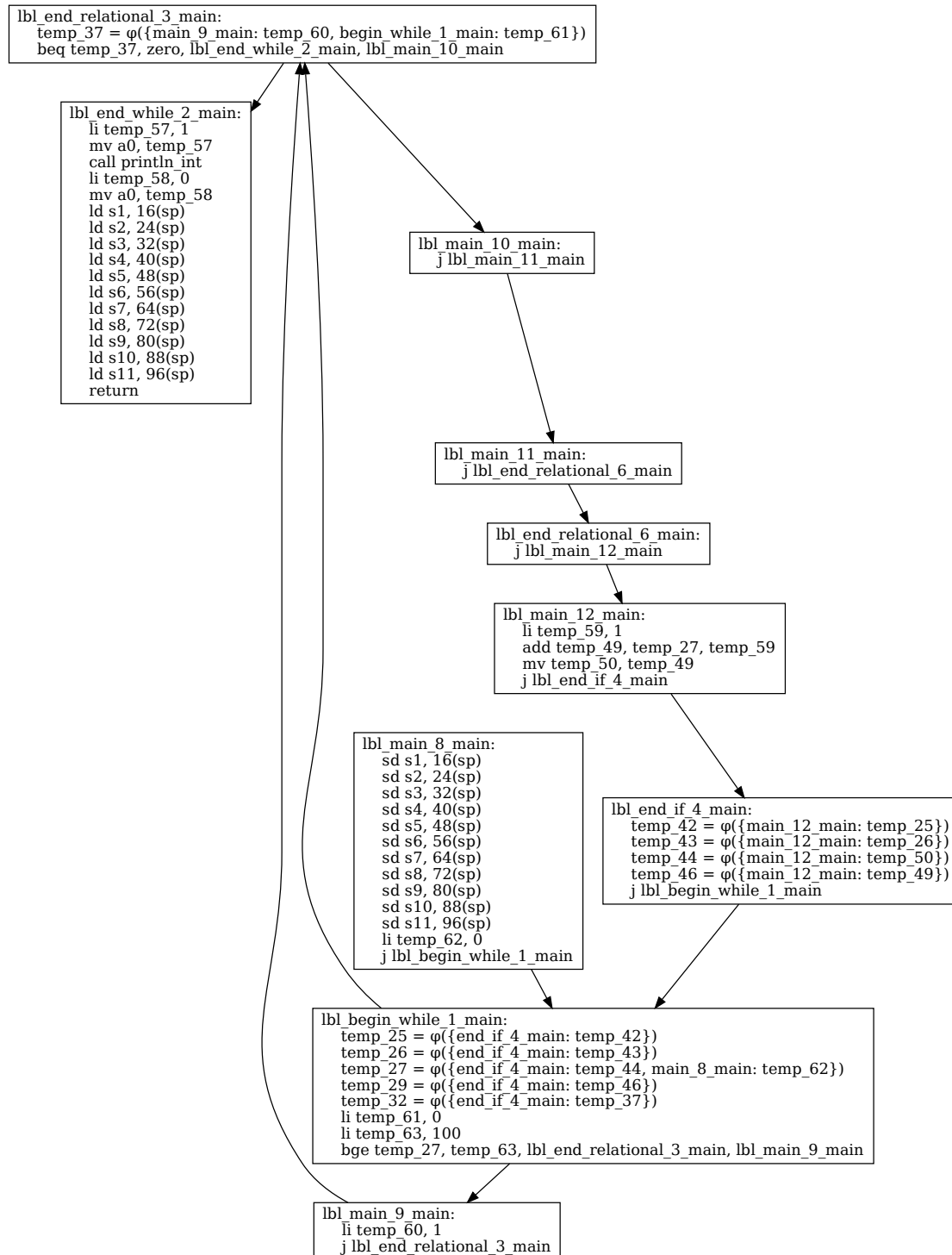
```
lbl_end_relational_3_main:
    temp_37 = φ({main_9_main: temp_60, begin_while_1_main: temp_61})
    beq temp_37, zero, lbl_end_while_2_main, lbl_main_10_main
```

```
lbl_end_while_2_main:
    li temp_57, 1
    mv a0, temp_57
    call println_int
    li temp_58, 0
    mv a0, temp_58
    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)
    return
```

```
lbl_main_10_main:
    j lbl_main_11_main
```

```
lbl_main_11_main:
    j lbl_end_relational_6_main
```

```
lbl_end_relational_6_main:
    j lbl_main_12_main
```

```
lbl_main_12_main:
    li temp_59, 1
    add temp_49, temp_27, temp_59
    mv temp_50, temp_49
    j lbl_end_if_4_main
```

```
lbl_main_8_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_62, 0
    j lbl_begin_while_1_main
```

```
lbl_end_if_4_main:
    temp_42 = φ({main_12_main: temp_25})
    temp_43 = φ({main_12_main: temp_26})
    temp_44 = φ({main_12_main: temp_50})
    temp_46 = φ({main_12_main: temp_49})
    j lbl_begin_while_1_main
```

```
lbl_begin_while_1_main:
    temp_25 = φ({end_if_4_main: temp_42})
    temp_26 = φ({end_if_4_main: temp_43})
    temp_27 = φ({end_if_4_main: temp_44, main_8_main: temp_62})
    temp_29 = φ({end_if_4_main: temp_46})
    temp_32 = φ({end_if_4_main: temp_37})
    li temp_61, 0
    li temp_63, 100
    bge temp_27, temp_63, lbl_end_relational_3_main, lbl_main_9_main
```

```
lbl_main_9_main:
    li temp_60, 1
    j lbl_end_relational_3_main
```

Figure 5.2: CFG of `example_lecture.c` after optimisation

EXERCISE #7 ► **Massive tests**

Once your optimisation is completed, check that every test you have still pass. For that purpose, the command `make test-codegen MODE=codegen-optim` runs your compiler on the whole test suite with the op-

timisation active. If you did implement the all-in-mem allocator but not the smart allocator, you can use `make test-lab4 MODE=codegen-optim` instead. If you implemented neither of them, you can use `make test-naive MODE=codegen-optim`.

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/index.html` in your web browser after a run of the test suite.

5.4 Extensions

EXERCISE #8 ► **Optimise the use of constants**
With our current implementation, we may have instructions of the following 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).

You can also simplify the code produced for instructions of the form `li t0, 0` by making use of the ZERO register, replacing all uses of `t0` with ZERO.

Finally, you can remove any empty block whose terminator is an absolute jump.