Lab 5
Smart IRs: Control Flow Graph in SSA Form

Objective

- Construct the CFG.
- Convert to then out of SSA Form.
- Emit linear code.

During the previous lab, you wrote a dummy code generator for the MiniC language. In this lab the objective is to prepare the field for more advanced compilation techniques, which will allow us to emit more efficient RISCV code. We remind you there are slides on the course webpage to help: `https://github.com/Drup/cap-labs21`

This lab and the following lab will be graded together: at the end of lab 6, you will have to deposit your work.

You will extend your previous code, in the same `MiniC` project, but in the `TP05/` subdirectory. People with a working code generator are encouraged to keep their current code. Otherwise, you will be provided a working 3 address code generation Visitor named `TP04/MiniCCodegen3AVisitor-correct.py`.

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 ...`

5.1 CFG construction

During class we presented Control Flow Graphs with maximal basic blocks. In this lab you will transform the linear code produced during the previous lab into a CFG, using the algorithm seen during the course.

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;
}
```

EXERCISE #2 ► **Finding the leaders**
In the course, we have defined the notion of *leaders*, which designate the indices of the instructions starting a block. We defined 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.

```
subi temp2, temp2, 4
beq temp2, zero, lelse1
li temp4, 7
mv temp1, temp4
jump lendif1
lelse1:
addi temp3, temp2, 1
mv temp1, temp3
lendif1:
```

EXERCISE #3 ► **Completing the CFG Construction**
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.

`Block`s 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. The procedure `_find_leaders` is currently incomplete and always return the list `[0,len(instructions)]`. Complete the procedure. You can assume that for each label that appears in the list of instructions, there is an instruction somewhere that jumps to it, this makes the code significantly simpler and just as correct.

Reminder: At any point of this lab, you can run `make tests-pyright` to check your code for typing errors.

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.

Now you can launch:

```
python3 MiniCC.py --reg-alloc none --graphs /path/to/example.c
```

Check examples with:

1. Straight code (for instance `TP05/tests/provided/dataflow/df01.c`)
2. Boolean expressions, tests and if statements
3. While loops
4. Your own tests

If available, Use `--reg-alloc all_in_mem` to obtain executable code from the CFG.

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.

5.2 SSA Form

Most of the code related to SSA is located in the `TP05/SSA.py` file. This file contains 3 main ingredients:

- The notion of `PhiNode`, a subclass of instructions representing ϕ nodes.
- A function `enter_ssa`, which converts a control flow graph to SSA form
- A function `exit_ssa`, which removes the ϕ nodes.

Your goal is to complete the missing pieces in this class: First compute the dominators and the dominance frontier and use it to insert the ϕ nodes ; and then replace the ϕ nodes by moves.

See the end of this file for a list of utility functions on control flow graphs already implemented which might help you.

To test your SSA implementation, run

```
python3 MiniCC.py --reg-alloc none --ssa /path/to/example.c
```

You can add the `--debug` option to the `MiniCC.py` invocation to print the dominators, the domination tree and the dominance frontiers of the given program. You can also add the `--ssa-graphs` option to view the domination tree and the CFG annotated with dominance frontiers graphically. **At each step, verify your computations by running simple C programs (for instance, the three examples above) with these debugging options!**

5.2.1 Conversion to SSA

EXERCISE #5 ▶ **Computing the dominators**

As seen in the course, dominators are computed by a system of equations. We write dominators[b] the dominators of the block b. It is defined as follow:

$$\text{dominators}[b] = \{b\} \qquad \text{if } b \text{ has no predecessors}$$
$$\text{dominators}[b] = \{b\} \cup \left(\bigcap_{p \in pred(b)} \text{dominators}[p] \right)$$

This kind of system is solved by an iterative algorithm which applies the system of equations until a fix point is reached: i.e., there are no changes when the equations are applied. We obtain an algorithm of the form:

```
dominators = ...
while True:
  new_dominators = ...
  if dominators == new_dominators:
    break
```

1. Compute the dominators on the examples above using this iterative approach
2. **Complete the `computeDom` function to compute the dominators.**

For the test `TP05/tests/provided/dataflow/df03.c` the debugging output should look like

```
SSA -dominators: {lbl_main_5_main: {lbl_main_4_main, lbl_begin_while_1_main,
    lbl_main_5_main}, lbl_end_relational_3_main: {lbl_end_relational_3_main,
    lbl_main_4_main, lbl_begin_while_1_main}, lbl_div_by_zero_0_main_msg: {
    lbl_div_by_zero_0_main_msg, lbl_div_by_zero_0_main}, lbl_main_6_main: {
    lbl_end_relational_3_main, lbl_main_6_main, lbl_begin_while_1_main,
    lbl_main_4_main}, lbl_end_while_2_main: {lbl_end_relational_3_main,
    lbl_main_4_main, lbl_end_while_2_main, lbl_begin_while_1_main}, lbl_main_4_main: {
    lbl_main_4_main}, lbl_begin_while_1_main: {lbl_main_4_main, lbl_begin_while_1_main
    }, lbl_div_by_zero_0_main: {lbl_div_by_zero_0_main}}
```

and the resulting domination tree should be as depicted in figure 5.1.

EXERCISE #6 ▶ **The Dominance Frontier**

The dominance frontier of a block b is the set of blocks which are "almost dominated" by b. More formally, B belongs to A's dominance frontier if:

- A does not strictly dominate B

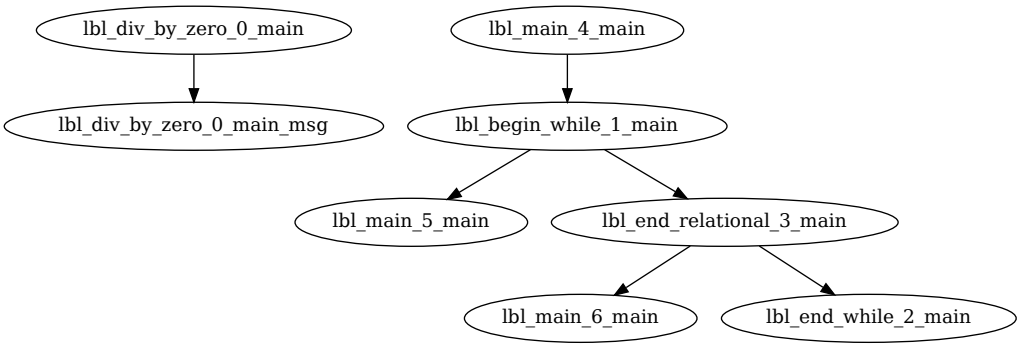

Figure 5.1: The domination tree of df03.c

- A dominates a direct predecessor of B

We recall the following algorithm, seen during the course, to compute the frontier:

```
computeDF(n) ::=
  S ← { y | y successor of n in G but not in DT}
  for c in children(DT,n):
    computeDF(c)
    for w in DF[c]:
      if n does not strictly dominate w:
        S ← S ∪ {w}
  DF[n] ← S

DF ::= computeDF(entry)
```

This algorithm requires the dominance tree, whose implementation is provided in the `computeDT` function.

Complete the procedure `computeDF_at_node` which, alongside `computeDF`, computes `DF`, the dictionary associating a node to its frontier.

For the test `TP05/tests/provided/dataflow/df03.c` the annotated CFG should look like figure 5.2.

EXERCISE #7 ▶ **Insertion of ϕ nodes**

We are now ready to insert the ϕ nodes. We recall the following algorithm from the course

```
Insert-phi ::=
  for x in Vars:
    for d in Defs(x):
      for b in DF(d):
        if there are no φ-node associated to x in b:
          add one such φ-node
          add b to Defs(x)
```

Complete the procedure `insertPhi` which inserts the ϕ nodes. Pay attention on where to insert the ϕ instructions in the blocks.

At this point, you should be able to call the `enter_ssa` procedure on any control flow graph to convert it to SSA. Use the `--graph` option to visualize the resulting CFG and verify its correctness.

5.2.2 Conversion out of SSA

The ϕ instructions we have added are convenient for manipulating the control flow graph, but are not implemented by processors. We need to remove them before emitting machine code.

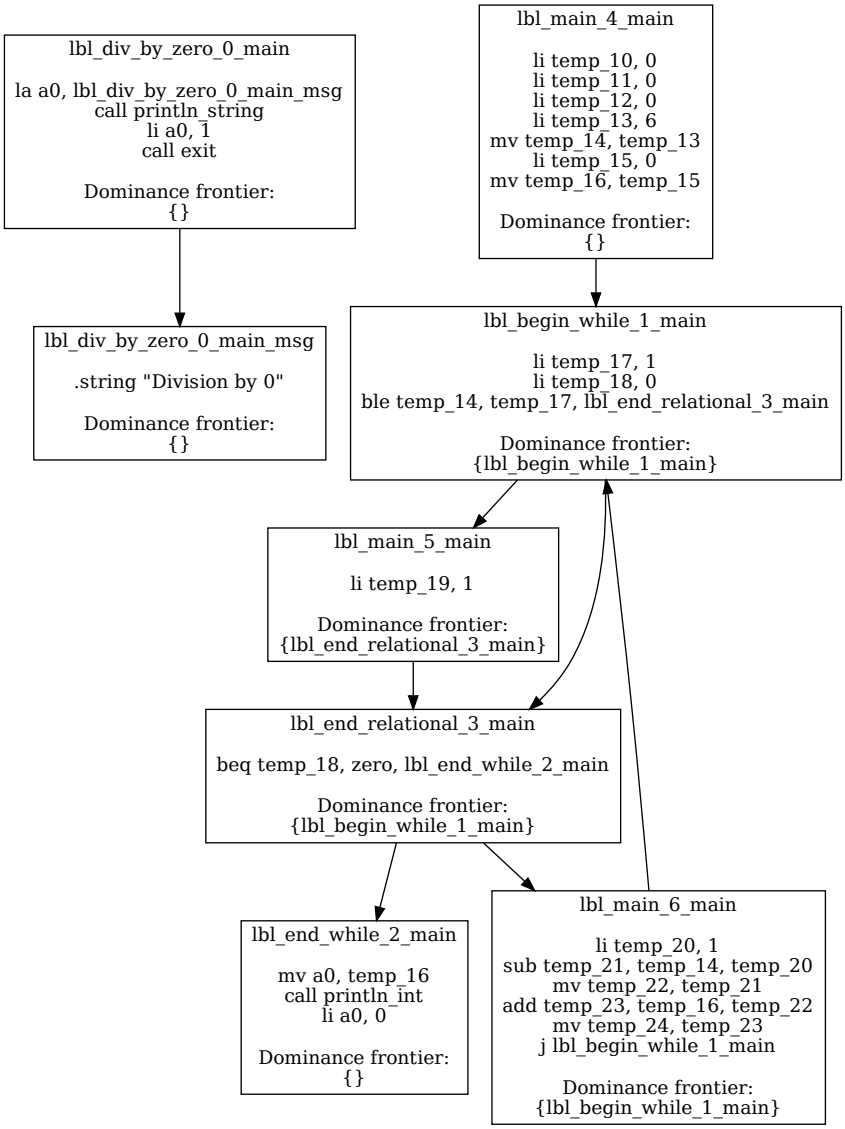

Figure 5.2: The annotated CFG of df03.c

EXERCISE #8 ▶ **Replacement of ϕ nodes by moves**

ϕ nodes can be eliminated by creating *new nodes* containing moves.
For instance, if we consider the block b_2 (with two parents b_0 and b_1) containing the following:

$$x_2 = \phi(b_0 : x_0, b1 : x_1)$$
$$y_2 = \phi(b0 : y_0, b1 : y_1)$$

We will insert two new blocks:
- Between b_0 and b_2 containing $x_2 \leftarrow x_0; y_2 \leftarrow y_0$
- Between b_1 and b_2 containing $x_2 \leftarrow x_1; y_2 \leftarrow y_1$

Complete the procedure `generate_moves_from_phis` **which creates a list of** `mv` **instructions equivalent to the ϕ nodes to replace, and the procedure** `exit_ssa` **which removes the ϕ nodes.** Do not forget to remove all edges and to modify instructions appropriately.

At this point, you should be able to call the `exit_ssa` procedure on any control flow graph to convert it out of SSA. Use the `--graph` option to visualize the resulting CFG and verify its correctness. You should also be able to run all tests. **At this point, all the tests should pass after going in and out of SSA!**

5.3 Bonus: Optimized CFG linearization

EXERCISE #9 ▶ **CFG Linearization**

Before emitting instructions, a control flow graph needs to be *linearized*, i.e., turned back into a linear sequence of instructions. This is currently in the `CFG` class by the `linearize` method.

Currently, this method is not very efficient: it emits jump instructions at the end of each block to link to the next block, even if it is immediately next. This results in the assembly instructions on the left, even though the instructions on the right is sufficient (and more efficient). Furthermore, the current implementation doesn't try to re-order the block to avoid jumps. This is particularly problematic after exiting SSA, which adds new blocks.

Listing 5.4: Code with extra jump

```
lbl_end_relational_3_main:
        ld s1, -144(fp)
        beq s1, zero, lbl_end_while_2_main
        j lbl_main_6_main
lbl_main_6_main:
        li s2, 1
        sd s2, -152(fp)
        ...
```

Listing 5.5: Code without extra jump

```
lbl_end_relational_3_main:
        ld s1, -144(fp)
        beq s1, zero, lbl_end_while_2_main
lbl_main_6_main:
        li s2, 1
        sd s2, -152(fp)
        ...
```

1. Inspect the assembly of simple programs after exiting SSA. Try to linearize them by hand to minimize the number of jumps.
2. Improve `cfg.linearize` to avoid jumps to blocks that are immediately following.
3. How would you choose a better order to linearize the blocks? Try to implement it.

Hint: Functions on Control Flow Graphs

Here are some functions implemented in `CFG` which might help you:
- `block.get_instructions()` return all the instructions of a block.
- `block.add_instruction(i,instr)` add some instruction at position i.
- `block.get_label()` return the label of a block.
- `block.get_jump()` return the final jump of a block, if there is one.
- `cfg.get_blocks()` return all the blocks in `cfg`.

- `cfg.get_entries()` return all the nodes with no predecessors in `cfg`.
- `cfg.add|remove_edge(src,dest)` adds or remove edges in the control flow graph.
- `cfg.gather_defs()` returns a dictionnary associating variables to all the blocks containing one of their definition.
- `jump.set_label(label)` replaces the label of a jump instruction.

We also encourage you to consult the documentation on set operations:

- The tutorial `https://docs.python.org/3/tutorial/datastructures.html#sets`
- The API `https://docs.python.org/3/library/stdtypes.html#set`