15-411: Compiler Design — Spring 2025

Recitation 2: SSA — 31 January

Constructing the Dominator Tree

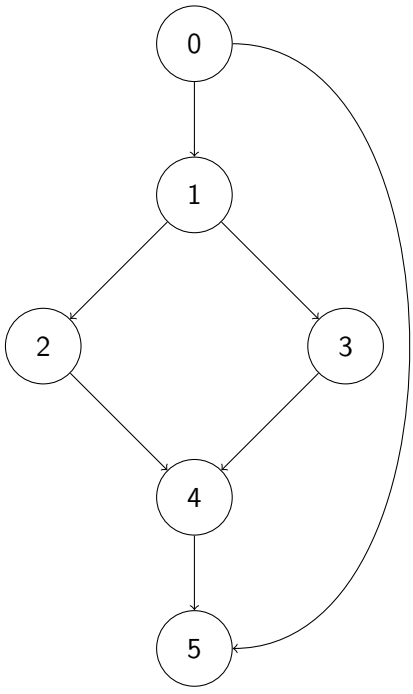
In order to enter SSA, we first require a dominator tree in order to effectively determine where Φ-functions should be inserted. We introduce some important definitions here:

- Node x **dominates** node y if every possible execution path from the CFG entry point to y includes x.
- Node x **strictly dominates** node y if x dominates y and $x \neq y$.
- The **immediate dominator** of a node x is the unique node that strictly dominates x but doesn't strictly dominate any other node that strictly dominates x.
- The **dominance frontier** of a node x is the set of nodes S such that for each node $y \in S$, x dominates a predecessor of y but x doesn't strictly dominate y. A more intuitive definition is the border of the CFG region that is dominated by x.

We construct the dominator tree by drawing an directed edge from node x to node y if x immediately dominates y.

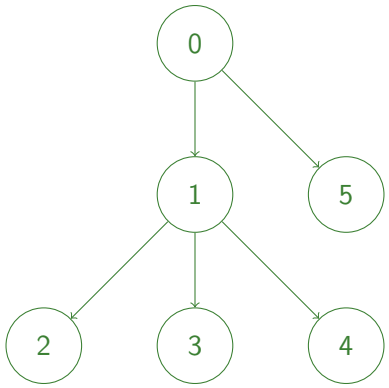
Checkpoint 0

Given the following CFG, draw its corresponding dominator tree. Then, list out the dominance frontiers for each node.



Solution:

Dominator tree:



dominance frontiers:

DF(0): {}
DF(1): {5}
DF(2): {4}
DF(3): {4}
DF(4): {5}
DF(5): {}

SSA Construction

We go about a more systematic way to construct SSA. We require the Dominance-Frontier Criterion: Whenever node x contains a definition of some variable a, then any node $z \in DF(x)$, z needs some Φ-function for a.

Applying this criterion, we insert Φ-functions where necessary:

- Gather all the defsites (basic block where variable has been defined) of every variable
- For each variable:
  - For each defsite:
    - For each node in DF(defsite):
      - If we don't have $\Phi()$ in node, put one in
      - If the node didn't previously define the variable, add this node to defsites

Note that the last step is necessary since the added Φ-function is considered a new definition for the variable.

Then, we need to rename variables so that each variable is defined once. To do this:

- Maintain a stack for each variable
- For each basic block BB in depth-first search preorder traversal through the dominator tree:
  - For each instruction in BB:
    - For each used variable in non-Φ-function, remove from variable's stack until topmost definition dominates current instruction
    - Replace each variable use by non-Φ-function with topmost stack definition

    - Add each variable definition to their corresponding stack
  - For each Φ-function in direct successors of BB:
    - For each used variable, remove from variable's stack until topmost definition dominates BB's exit point
    - Replace each variable use with topmost stack definition
  - Restore original states of variable stacks (before the current basic block was entered)

Checkpoint 1

Given the following abstract assembly, first add Φ-instructions to the abstract assembly above as necessary. Then, rename variables so that the program is in SSA form.

```
1  main:
2    x <- 1
3    if (x >= 2) then goto L1 else goto L2
4  L1:
5    x <- x + 2
6    goto L3
7  L2:
8    x <- 3
9    goto L3
10 L3:
11   %eax <- x
12   return
```

Solution: We observe that the abstract assembly has four basic blocks (main, L1, L2, L3). L1 and L2 both have a non-empty dominance frontier, both being {L3}. Adding phi functions, we get:

```
1  main:
2    x <- 1
3    if (x >= 2) then goto L1 else goto L2
4  L1:
5    x <- x + 2
6    goto L3
7  L2:
8    x <- 3
9    goto L3
10 L3:
11   x <- phi()
12   %eax <- x
13   return
```

Then, for renaming variables, we observe that we first visit the main block, followed by any ordering of the other three blocks. This gives us:

```
1  main:
2    x_0 <- 1
3    if (x_0 >= 2) then goto L1 else goto L2
4  L1:
5    x_1 <- x_0 + 2
```

```
  goto L3
L2:
  x_2 <- 3
  goto L3
L3:
  x_3 <- phi(x_1, x_2)
  %eax <- x_3
  return
```

SSA Deconstruction

Once we are finished working in SSA, we need to deconstruct it before converting the abstract assembly to x86, since Φ-functions are not executable machine instructions. To do this, we want to make conventional SSA (where the variables in Φ-functions never interfere). This is a form that we naturally receive from SSA construction, but certain optimizations may make the SSA non-conventional.

To fix this, we introduce the idea of splitting critical edges. A critical edge is an edge $x \to y$ such that x has more than one successor and y has more than one predecessor. To split such a critical edge, we insert a new node z, then replace the previous edge with $x \to z$ and $z \to y$.

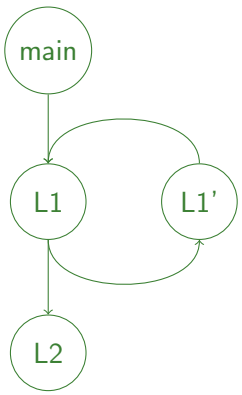
Once we do have conventional SSA, SSA deconstruction is easy – for every Φ-function assignment $x_0 \leftarrow \Phi(B_1 : x_1, B_2 : x_2, \ldots, B_n : x_n)$ (where $B_i : x_i$ refers to the definition of x_i arriving from a direct predecessor B_i), we insert the move $x_0 \leftarrow x_i$ to the parallel copy PC_i in B_i.

Checkpoint 2

Given the following abstract assembly, draw the CFG. Then, redraw the CFG, splitting any critical edges.

Solution:

Observe that the only critical edge in the CFG is L1's self-edge. Thus, after splitting, the CFG would look like so:




```
main:
  x_0 <- 1
  goto L1
L1:
  x_2 <- phi(x_0, x_1)
  x_1 <- x_2
  if (true) then goto L1 else goto L2
```

```
8  L2:
9    %eax <- x_1
10   return
```

Checkpoint 3

As x86 doesn't support parallel copies, we need to replace them with sequential copies. Given a parallel copy, so long as there exists a move $b \leftarrow a$ such that there aren't any $c \leftarrow b$, we can add $b \leftarrow a$ to the list of sequential copies, removing the move from the PC. Otherwise, the parallel copy only consists of cycles, so we need to break one by introducing a fresh temp. We do this by picking a move $b \leftarrow a$ in the PC, adding $a' \leftarrow a$ to the sequential moves, and replacing $b \leftarrow a$ in the PC with $b \leftarrow a'$ (such that a' is fresh).

Given the parallel copy PC$(b \leftarrow a, c \leftarrow b, a \leftarrow c, d \leftarrow c)$, convert it to sequential moves, breaking cycles as necessary.

Solution:

One possible solution is as follows (segmenting $b \leftarrow a$):

```
1  d <- c
2  a' <- a
3  a <- c
4  c <- b
5  b <- a'
```

SSA Resources

As Lab 2 onwards require that you implement SSA, these references may be helpful to read about SSA implementation intricacies in more detail:

(a) `https://link.springer.com/book/10.1007/978-3-030-80515-9`

(b) `https://homepages.dcc.ufmg.br/~fernando/publications/papers/CC09.pdf`

Lab 2 Hints

- For the expression `if (a < 0) if (b < 0) x = 4 else x = 5`, x is not assigned if $a \geq 0$ (`else` binds to the most recent `if`)
- You have to add support for Boolean variables now, and you will have to add support for pointers in lab 4. Plan ahead when making design decisions to support different types in type checking and instruction selection.
- We suggest adding support for a `-O0` flag that disables register allocation and places all temps on the stack. Interference bugs fail in subtle, hard to understand ways.
- You can step through your programs with gdb. Place a breakpoint with `break _c0_main`, then use `step` to advance the program.