Recitation 2: SSA — 17 September

We talk a lot about SSA in this class. Today, we're going to talk all about SSA: getting into it, some pitfalls, and getting out.

Getting into SSA

Recall the Fibonacci sequence:

$$
\begin{aligned}
F_0 &= 0 \\
F_1 &= 1 \\
F_n &= F_{n-1} + F_{n-2} && n > 1
\end{aligned}
$$

Check out this lil program that computes the nth Fibonacci number:

```
int fib(int n) {
  if (n == 0) return 0;
  int a = 0;
  int b = 1;
  int i = 1;
  while (i < n) {
    int c = b;
    b = a + b;
    a = c;
    i++;
  }
  return b;
}
```

Checkpoint 0

Translate this program into abstract assembly, organized as basic blocks within a control flow graph.

Checkpoint 1

First, some definitions (Cooper et al.):

- $\textsc{Dom}(b)$: A node n in the CFG dominates b if n lies on every path from the entry node of the CFG to b. $\textsc{Dom}(b)$ contains every node n that dominates b. For $x, y \in \textsc{Dom}(b)$, either $x \in \textsc{Dom}(y)$ or $y \in \textsc{Dom}(x)$. By definition, for any node b, $b \in \textsc{Dom}(b)$.
- $\textsc{IDom}(b)$: For a node b, the set $\textsc{IDom}(b)$ contains exactly one node, the *immediate dominator*. Intuitively, b's immediate dominator is the node $n \in \textsc{Dom}(b)$ which is closest to b. If n is b's immediate dominator, then every node in $\{\textsc{Dom}(b) - b\}$ is also in $\textsc{Dom}(n)$.

We suggest that you use an algorithm from the paper, *A Simple, Fast Dominance Algorithm*:

```
for all nodes, b /* initialize the dominators array */
    doms[b] ← Undefined
doms[start_node] ← start_node
Changed ← true
while (Changed)
    Changed ← false
    for all nodes, b, in reverse postorder (except start_node)
        new_idom ← first (processed) predecessor of b /* (pick one) */
        for all other predecessors, p, of b
            if doms[p] ≠ Undefined /* i.e., if doms[p] already calculated */
                new_idom ← intersect(p, new_idom)
        if doms[b] ≠ new_idom
            doms[b] ← new_idom
            Changed ← true

function intersect(b1, b2) returns node
    finger1 ← b1
    finger2 ← b2
    while (finger1 ≠ finger2)
        while (finger1 < finger2)
            finger1 = doms[finger1]
        while (finger2 < finger1)
            finger2 = doms[finger2]
    return finger1
```

Calculate (immediate) dominance relations between basic blocks and draw the dominator tree.

Checkpoint 2

Find the dominance frontier of each block, b, by looking at the successors of each block that is dominated by b. Again, we use an algorithm from the same paper:

```
for all nodes, b
    if the number of predecessors of b ≥ 2
        for all predecessors, p, of b
            runner ← p
            while runner ≠ doms[b]
                add b to runner's dominance frontier set
                runner = doms[runner]
```

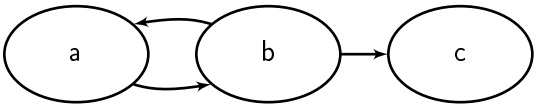
Checkpoint 3

Insert Φ functions at the beginning of basic blocks and rename variables to convert the program into SSA form. Whenever a node x has a (live) definition of a variable a, then any node z in the dominance frontier of x needs a phi-function for a. There are a few algorithms for this, but for this exercise, just do it by inspection.

Getting out of SSA

We can get out of SSA by inserting appropriate copies. There are some problems that can occur, including the "Lost copy problem" and the "Swap problem". The lost copy problem can be resolved by splitting critical edges, making sure that there is never an edge from a node with multiple children to a node with multiple parents. If you're not doing register allocation on SSA, you can also just convert to CSSA and this will also fix the problem. The swap problem can be resolved by properly implementing the parallel semantics of phi-functions.

One way to think about parallel moves is through location transfer graphs, where locations are nodes and directed edges represent the flow of values. For example, the parallel move $a \leftarrow b, b \leftarrow a, c \leftarrow b$ can be represented by the following graph:

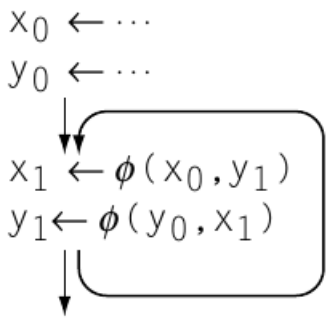


Checkpoint 4

Translate the previous fib exercise back out of SSA.

Checkpoint 5

Translate the following out of SSA:



Hint: this is a minimal example of the swap problem.

Checkpoint 6

Conventional SSA (CSSA) is a form of SSA where phi-related variables do not interfere.

Since you're not doing any optimizations yet, don't worry about CSSA. If you're using the above algorithm to get into SSA, then you're already in CSSA!

But when you do add some optimizations like copy prop, you might need to get back into CSSA.

- If you want to do register allocation on SSA, CSSA will guarantee that you end up with no memory to memory operations, provided that your register allocator always puts phi-related variables in the same memory location.
- Going into CSSA also solves the lost copy problem without requiring that critical edges be split.

To convert to CSSA: For phi-functions $a_0 \leftarrow \phi(a_1, \ldots, a_n), b_0 \leftarrow \phi(b_1, \ldots, b_n), \ldots$

- Insert parallel copies $a'_1 \leftarrow a_1$, $b'_1 \leftarrow b_1, \ldots$ at the end of the block corresponding to column 1, and so on for each column.
- Replace $a_0 \leftarrow \phi(a_1, \ldots, a_n), b_0 \leftarrow \phi(b_1, \ldots, b_n), \ldots$ with $a'_0 \leftarrow \phi(a'_1, \ldots, a'_n), b'_0 \leftarrow \phi(b'_1, \ldots, b'_n), \ldots$
- Insert parallel copies $a_0 \leftarrow a'_0, b_0 \leftarrow b'_0, \ldots$ after the ϕ-functions.

For elimination, we can then safely rename out of CSSA, dropping the subscripts in the primed names. So $a'_0 \leftarrow \phi(a'_1, \ldots, a'_n)$ becomes $a' \leftarrow \phi(a', \ldots, a')$. Implement these using parallel copies, as we did above, and finally delete the phi functions and coalesce. If you want to do elimination after register allocation, see the Pereira paper, *SSA-Elimination after Register Allocation*.

(Optional) Convert the previous example to CSSA and eliminate the phi-functions by the method given above.