15-411: Compiler Design — Spring 2025

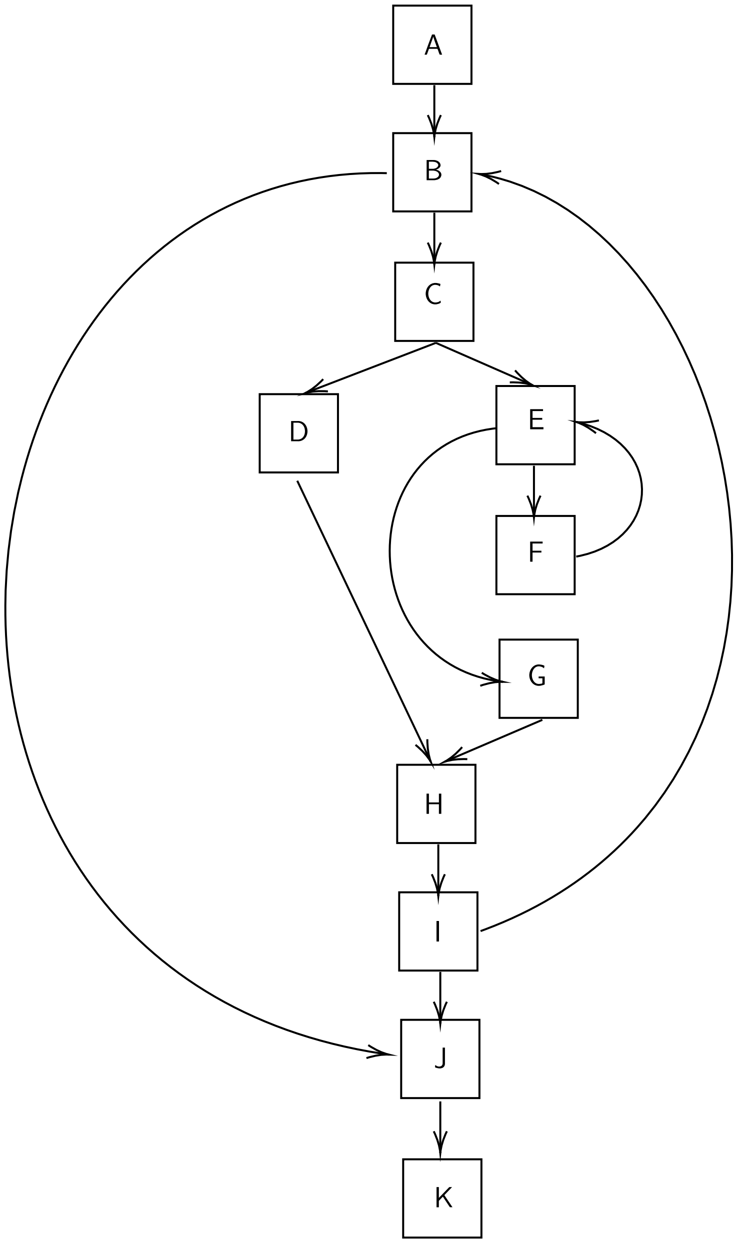
Recitation 9: Optimizations II — 28 March

Finding Loops

Recall that a loop is a strongly connected component of the CFG that has an entry edge (the source is not in the loop, but the destination is), an exit edge (the source is in the loop, but the destination is not), a loop header (the target of the entry edge), a back edge (the target is the header and the source is in the loop), and a preheader (the source of the only entry edge). A natural loop is one with only a single loop header. In C0, all of your loops should be natural loops.

Checkpoint 0

Given the following CFG, find the loop(s).



Loop Invariant Code Motion

A computation whose value does not change as long as control stays within the loop is called "loop-invariant." The code motion would be to move this computation to the preheader. Finding a loop invariant computation in SSA is very simple: all of the operands must be defined outside of the loop or must be invariant themselves. **However**, not all loop-invariant computations are able to be moved.

In order for a computation to be moved to the preheader, it must meet some conditions: it must be loop-invariant, it must be in blocks that dominate all exits of the loop, it must not be assigned elsewhere in the loop, and it must dominate all blocks in the loop that use the variable. In SSA, these are simplified. Must not be assigned elsewhere - done. Must dominate all blocks that use the variable - done. The main one to watch out for is if it dominates all exits.

Checkpoint 1

Find the loop invariant computations in the following code, and state which ones can be moved to the preheader.

```
for (int i = 0; i < 20 + 9; i++) {
  int a = 50;
  int b = x * a;
  bool c = f(b);
  int z;
  if (c) {
    z = 0;
    return 9;
  } else {
    z = 5 << x;
  }
  int d = z + 2;
  y += d;
  int e = 8 << x;
  int g = a + b;
  y += e * g;
}
```

Induction Variable Analysis

Induction variable is the key to understand loop behavior at compile-time. The computation of a IV inside a loop can be neatly summarized by a recurrence relation, which can in turn uniquely represent this IV:

$$\{\langle \text{base} \rangle, +, \langle \text{step} \rangle\}$$

Here $\langle \text{base} \rangle$ and $\langle \text{step} \rangle$ must be either a constant, a def that is loop-invariant, or another IV.

Note: The third case represents a high-order recurrence, which you can disable for simplicity, as the previous two represents most IVs in the program and can already provide powerful analysis.

Note: Also note that the operator can be more than just $+$: $\times$, $-$, umin (unsigned min), smin (signed min), $\cdots$. For most IVs, $+$, $-$, and $\times$ should be more than enough.

You have already seen Induction Variable Analysis in lecture without SSA. A variable in a loop is an **basic IV** if its one and only definition **in the loop** has the form:

$$x \leftarrow x + \langle \text{step} \rangle$$

A variable is an IV from a basic IV if its one and only definition **in the loop** is the affine transformation of basic IV:

$$y \leftarrow c_1 * x + c_2$$

here `c1` and `c2` are either constants or loop-invariant.

However, you can't find IVs in your IR using this method as is: `y <- c1 * x + c2` is not 3-address and `x <- x + <step>` is not SSA. SSA book gives such an algorithm to find `basic IVs`. For every ϕ in the loop header of loop `L`, traverse its use-def chain in DFS fashion until you find two things:

(a) A constant or a def outside `L`. This becomes $\langle$base$\rangle$.

(b) A circuit that starts and ends at the same ϕ, where all intermediate instructions are (1) non-phi and (2) strictly contained inside this loop.

You can discover $\langle$base$\rangle$ and $\langle$step$\rangle$ by traversing the circuit (reject IV if you cannot).

To find `derived IV`, simply traverse the def-use chain of your basic IV. For basic IV `x`, a def `d` along def-use chain of `x` is derived IV if:

- `d <- x + c`
- `d <- x - c`
- `d <- x * c`

Here `c` is loop invariant as usual.

Checkpoint 2

```
  a <- 3
  b <- 1
H:
  if (a != n) then H else E
  c <- phi(a, f)
  d <- phi(b, g)
  e <- d + 7
  f <- e + c
  g <- d + 5
  goto H
E:
```

Are `c` and `d` basic IV of the loop? If so, find their $\langle$base$\rangle$ and their ϕ-circuits. Discover their recurrence at the end.

Induction Variable Elimination

Once you find the basic IV, you can proceed to eliminate its derivations. For a derived IV `A` from basic IV `B`, relative to loop `L`, with `A`'s recurrence:

$$A = \{\langle \text{base} \rangle, +, \langle \text{step} \rangle\}$$

(a) Create a proxy variable $A_0 \leftarrow \langle$base$\rangle$ in `L`'s preheader. $\langle$base$\rangle$ could generate more than one instrutions.

(b) Create $A_1 \leftarrow \phi(A_0, A_2)$ in loop header.

(c) Replace `A`'s def, e.g. $A \leftarrow B + c$ with $A \leftarrow A_1$.

(d) Put $A_2 \leftarrow A_1 + \langle\text{step}\rangle$ right before the back edge. Again $\langle\text{step}\rangle$ could generate more than one instrutions.

Checkpoint 3

IV can optimize array-based loops significantly. Here assume `int A[]` is declared and malloc'ed:

```
for (int i = 0; i < n; ++i) {
  A[i] = i;
}
```

The 3-address SSA IR looks something like (unsafe mode, no need to store length or do bound check):

```
  i0 <— 0
H:
  if (i0 < n) goto E
  i1 <— phi(i0, i2)
  t1 <— i1 * 4
  t2 <— A + t1
  Mem[t2] <— i1
  i2 <— i1 + 1
  goto H
E:
```

Discover all basic and derived IVs and perform IV elimination. What optimizations can you perform after IVE?