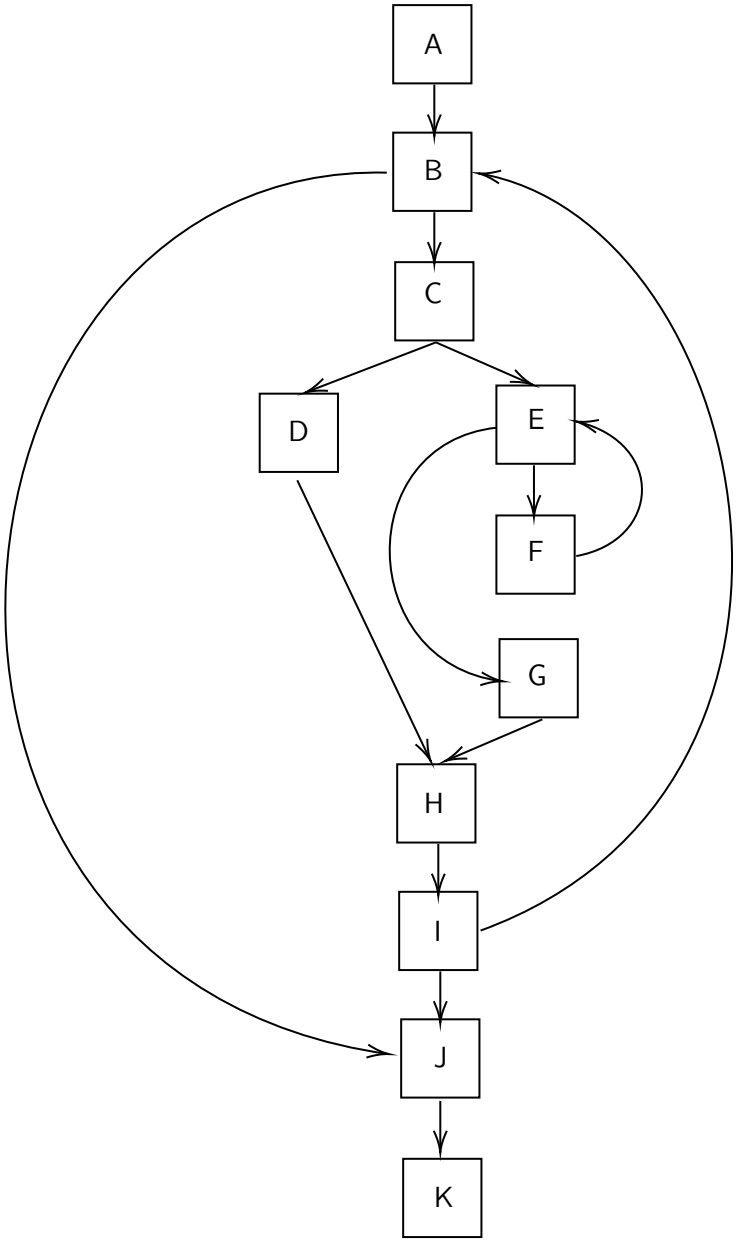
Recitation 9: Optimizations II Solutions 27 March

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).



Solution: Inner loop: E, F. Preheader: C, although a separate one can (perhaps should) be inserted between C and E.

Outer loop: B, C, D, E, F, G, H, I. Preheader: A, although a separate one can be inserted between A and B.

Loop Invariant Code Motion

A computation whose value does not change as long as control stays within the loop is called "loop-invariant." Loop invariant code motion would be the act of moving this computation to the preheader. 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 a block that dominates all exits of the loop, its destination must not be reassigned to the loop, and it must dominate all blocks in the loop that use its destination.

In SSA, these criteria can be met much more easily. Simply find a (pure) computation whose operands are defined outside of the loop, or are loop invariant themselves. Must not be assigned elsewhere - done. Must dominate all blocks that use the variable - done. The main criterion to watch out for is if dominates all exits.

Checkpoint 1

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

```
1 for (int i = 0; i < 20 + 9; i++) {
2   int a = 50;
3   int b = x * a;
4   bool c = f(b);
5   int z;
6   if (c) {
7     z = 0;
8     return 9;
9   } else {
10    z = 5 << x;
11  }
12  int d = z + 2;
13  y += d;
14  int e = 8 << x;
15  int g = a + b;
16  y += e * g;
17 }
```

Solution: 20 + 9, int a = 50, int b = x * a can all be moved out.

If f is pure, then bool c = f(b) can also be moved out.

z, d, e, and g cannot be moved out because they do not dominate all exits.

y cannot be moved out because it is not loop invariant.

Induction Variable Analysis

Induction variables (IVs) are the key to understanding loop behavior at compile-time. The computation of an IV v on iteration i of a loop can be neatly summarized by a recurrence relation of the following

form:

$$v_0 = \langle \text{base} \rangle$$
$$v_{i+1} = v_i \oplus \langle \text{step} \rangle$$

Here $\langle \text{base} \rangle$ and $\langle \text{step} \rangle$ must be loop-invariant. Operator $\oplus$ can represent a variety of operators. Typically $+$, $-$, and $\times$ will suffice, but you could also support more interesting ones like smin (signed min). Altogether, the IV can be characterized by the following triple:

$$(\langle \text{base} \rangle, \oplus, \langle \text{step} \rangle)$$

Without SSA, a variable x 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 y in a loop is a **derived IV** if its one and only definition is the affine transformation of another IV x:

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

We restrict c_1 and c_2 to be loop-invariant.

However, you can't find IVs by directly searching for definitions of this form if you are working in an SSA 3-address abstract assembly. The first definition violates SSA, and the second definition is not 3-address. This SSA book gives such an algorithm to find **basic IVs**. For every ϕ in the loop header, traverse its use-def chain in DFS fashion until you find two things:

(a) A constant or a def outside (and thus before) the loop. This becomes $\langle \text{base} \rangle$.

(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 $\langle \text{base} \rangle$ and $\langle \text{step} \rangle$ by traversing the circuit (reject IV if you cannot).

To find **derived IVs**, traverse the def-use chains of the basic IVs. For IV x, a def d along def-use chain of x is a derived IV if it is applies an affine transformation to x, e.g. it could be $d \leftarrow x + c$ or $d \leftarrow x * c$. Here c is loop invariant as usual. In general, the recurrence for this new derived IV d can be derived from the recurrence for the original IV x. For example, suppose that IV x's recurrence is characterized by the triple $(b_x, +, s_x)$. Then we can determine d's recurrence depending on how it is defined:

- $d \leftarrow x + c$: $(b_x + c, +, s_x)$
- $d \leftarrow x * c$: $(c * b_x, +, c * s_x)$

Checkpoint 2

```
1 H:
2   c <- phi(a, f)
3   d <- phi(b, g)
4   if (d >= n) goto E else goto B
5
6 B:
7   e <- d + 7
8   f <- e + c
9   g <- d + 5
10  goto H
11
12 E:
```

Are `c` and `d` basic IVs of the loop? If so, find their ϕ-circuits and state the triple characterizing their recurrences.

Solution:

```
d is a basic IV. c is not.

d's <base> is b; d's phi-circuit is:
  d <- phi(b, g) => g <- d + 5 => d <- phi(b, g)

5 is constant, so it becomes <step>. The final recurrence of d is {b, +, 5}.

c's <base> is a; c's phi-circuit is (backtrack at e):
  c <- phi(a, f) => f <- e + c => c <- phi(a, f)

However, e is NOT loop-invariant, so you can't derive the recurrence {a, +, e}.
```

Induction Variable Elimination

Once you have performed IV analysis, you can eliminate all the derived IVs stemming from each basic IV. Consider an IV x derived from basic IV y with a recurrence characterized by:

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

IV x can be eliminated through the following steps:

(a) Create a proxy variable $x_0 \leftarrow \langle \text{base} \rangle$ in the loop preheader. As it can involve several affine operations, $\langle \text{base} \rangle$ could generate more than one instruction.

(b) Create $x_1 \leftarrow \phi(x_0, x_2)$ in loop header.

(c) Remove x's def, and replace its uses with x_1.

(d) Put $x_2 \leftarrow x_1 + \langle \text{step} \rangle$ right before the back edge. Again $\langle \text{step} \rangle$ could generate more than one instruction.

Checkpoint 3

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

```
1  for (int i = 0; i < n; ++i) {
2    A[i] = i;
3  }
```

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

```
1 H:
2  i1 <- phi(i0, i2)
3  if (i1 >= n) goto E else goto B
4
5 B:
6  t1 <- i1 * 4
7  t2 <- A + t1
8  Mem[t2] <- i1
9  i2 <- i1 + 1
10 goto H
11
12 E:
```

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

Solution:

```
i1 is the basic IV. Its <base> is i0. Its phi-circuit is
  i1 <- phi(i0, i2) => i2 <- i1 + 1 => i1 <- phi(i0, i2)
so its recurrence is {i0, +, 1}.

t1 and t2 are IVs derived from i1 (discovered by following def-use chain i1->t1->t2).
t1's recurrence is {4*i0, +, 4}, and t2's recurrence is {A+4*i0, +, 4}.

P: # preheader
  t1_a <- i0 * 4
  t2_a <- A + t1_a

H: # header
  i1 <- phi(i0, i2)
  t1_b <- phi(t1_a, t1_c)
  t2_b <- phi(t2_a, t2_c)
  if (i1 >= n) goto E else goto B

B: # body
  t1 <- t1_b
  t2 <- t2_b
  Mem[t2] <- i1
  i2 <- i1 + 1
  t1_c <- t1_b + 4
  t2_c <- t2_b + 4
  goto H

E:
```