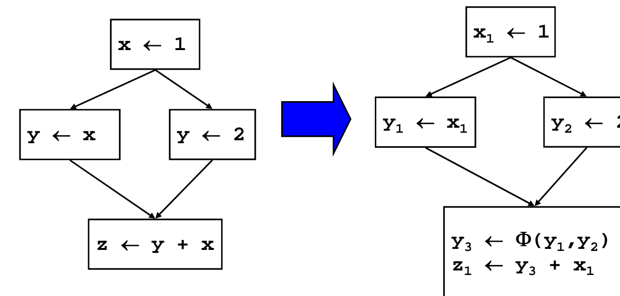
Lecture 14

SSA-Style Optimizations

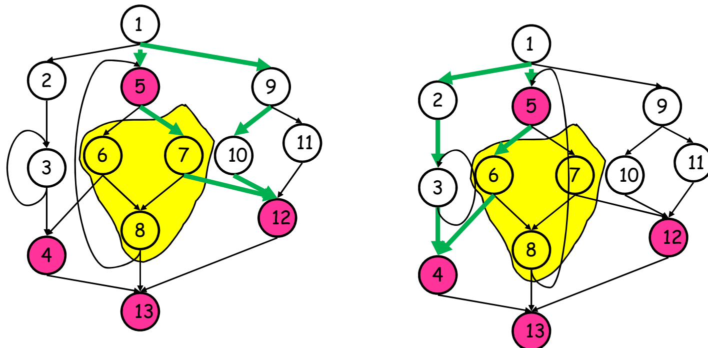
(Slides courtesy of Seth Goldstein.)

Review: Minimal SSA

- Each assignment generates a fresh variable.
- At each join point insert Φ functions for all variables with multiple outstanding defs.



Review: Dominance Frontier and Path Convergence



Constant Propagation

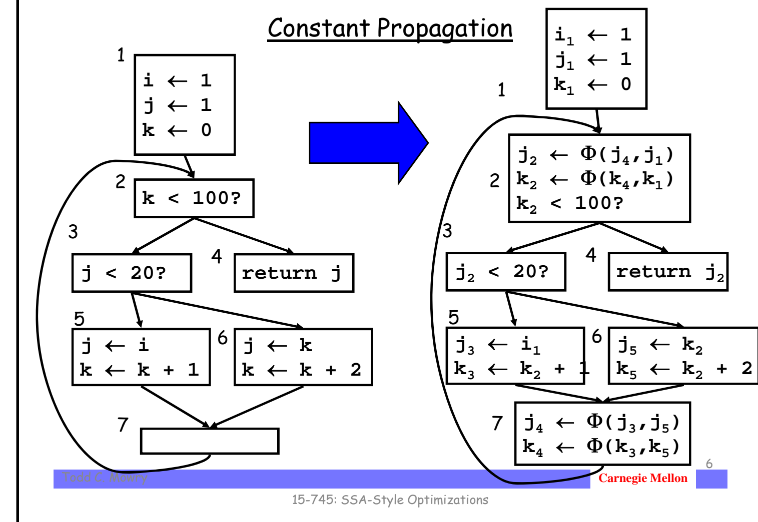
- If "v ← c", replace all uses of v with c
- If "v ← Φ(c,c,c)", replace all uses of v with c

```
W <- list of all defs
while !W.isEmpty {
        Stmt S <- W.removeOne
        if S has form "v <- Φ(c,…,c)"
          replace S with V <- c
        if S has form "v <- c" then
          delete S
          foreach stmt U that uses v,
            replace v with c in U
            W.add(U)
}
```

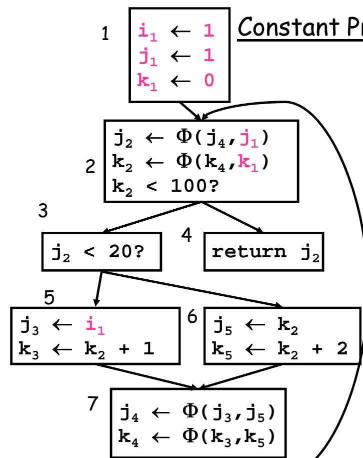
Other Optimizations with SSA

- Copy propogation
  - delete "$x \leftarrow \Phi(y,y,y)$" and replace all x with y
  - delete "$x \leftarrow y$" and replace all x with y
- Constant Folding
  - (Also, constant conditions too!)
- Unreachable Code
  - Remember to delete all edges from unreachable block

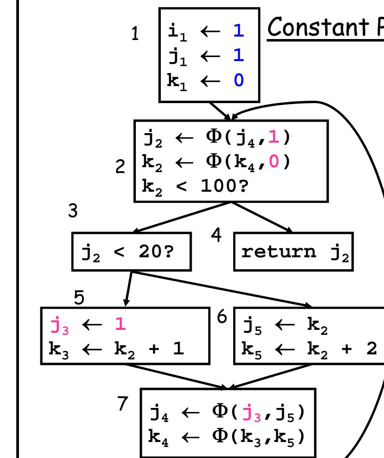
Constant Propagation



Constant Propagation



Constant Propagation



Constant Propagation

```
1  i1 ← 1
   j1 ← 1
   k1 ← 0

2  j2 ← Φ(j4,1)
   k2 ← Φ(k4,1)
   k2 < 100?

3  j2 < 20?          4  return j2

5  j3 ← 1            6  j5 ← k2
   k3 ← k2 + 1          k5 ← k2 + 2

7  j4 ← Φ(1,j5)
   k4 ← Φ(k3,k5)
```

But, so what?

Conditional Constant Propagation

```
1  i1 ← 1
   j1 ← 1
   k1 ← 0

2  j2 ← Φ(j4,1)
   k2 ← Φ(k4,1)
   k2 < 100?

3  j2 < 20?          4  return j2

5  j3 ← 1            6  j5 ← k2
   k3 ← k2 + 1          k5 ← k2 + 2

7  j4 ← Φ(1,j5)
   k4 ← Φ(k3,k5)
```

- Does block 6 ever execute?
- Simple CP can't tell
- Conditional CP can tell:
  - Assumes blocks don't execute until proven otherwise
  - Assumes values are constants until proven otherwise

Conditional Constant Propagation Algorithm

Keeps track of:

- Blocks
  - assume unexecuted until proven otherwise
- Variables
  - assume not executed (only with proof of assignments of a non-constant value do we assume not constant)

Lattice for representing variables:

Lattice	Meaning
$\top$	not executed
··· -2 -1 0 1 2 ···	we have seen evidence that the variable has been assigned a constant with the value
$\perp$	we have seen evidence that the variable can hold different values at different times

Conditional Constant Propagation

```
1  i1 ← 1
   j1 ← 1
   k1 ← 0

2  j2 ← Φ(j4,1)
   k2 ← Φ(k4,0)
   k2 < 100?

3  j2 < 20?          4  return j2

5  j3 ← 1            6  j5 ← k2
   k3 ← k2 + 1          k5 ← k2 + 2

7  j4 ← Φ(1,j5)
   k4 ← Φ(k3,k5)
```

Conditional Constant Propagation

```
1   i1 ← 1
    j1 ← 1
    k1 ← 0

2   j2 ← Φ(j4,1)
    k2 ← Φ(k4,0)
    k2 < 100?

3   j2 < 20?          4   return j2

5   j3 ← 1            6   j5 ← k2
    k3 ← k2+ 1            k5 ← k2 + 2

7   j4 ← Φ(1,j5)
    k4 ← Φ(k3,k5)
```


Conditional Constant Propagation

```
1   i1 ← 1
    j1 ← 1
    k1 ← 0

2   j2 ← Φ(j4,1)
    k2 ← Φ(k4,0)
    k2 < 100?

3   j2 < 20?          4   return j2

5   j3 ← 1            6   j5 ← k2
    k3 ← k2+ 1            k5 ← k2 + 2

7   j4 ← Φ(1,j5)
    k4 ← Φ(k3,k5)
```

```
k2 ← Φ(k3,0)
k2 < 100?

k3 ← k2+1        return 1
```


Dead Code Elimination

```
W <- list of all defs
while !W.isEmpty {
        Stmt S <- W.removeOne
        if |S.users| != 0 then continue
        if S.hasSideEffects() then continue
        foreach def in S.definers {
        def.users <- def.users - {S}
          if |def.users| == 0 then
            W <- W UNION {def}
        }
        delete S
}
```

Since we are using SSA, this is just a list of all variable assignments.

Example DCE

```
B0  i <- 0
    j <- 0

B1  i <- i*2
    j <- j+1
    j < 10?

B2  return j
```

```
B0  i0 <- 0
    j0 <- 0

B1  j1 ← Φ(j0,j2)
    i1 ← Φ(i0,i2)
    i2 <- i1*2
    j2 <- j1+1
    j2 < 10?

B2  return j2
```

Standard DCE leaves Zombies!

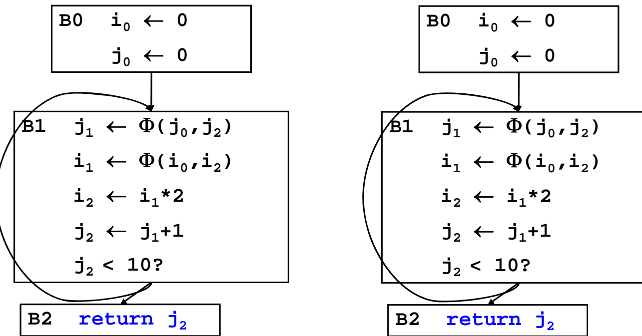
Aggressive Dead Code Elimination

Assume a statement is dead until proven otherwise.

```
init:
   mark as live all stmts that have side-effects:
       - I/O
       - stores into memory
       - returns
       - calls a function that MIGHT have side-effects
   As we mark S live, insert S.defs into W

while (|W| > 0) {
   S <- W.removeOne()
   if (S is live) continue;
   mark S live, insert S.defs into W
}
```


Example DCE

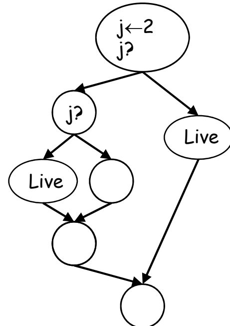


Problem!

Fixing DCE

if S is live, then

if T determines if S can execute, T should be live

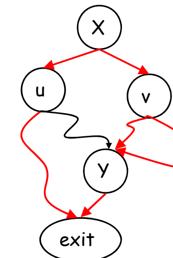


Control Dependence

Y is control-dependent on X if

- X branches to u and v
- ∃ a path u→exit which does not go through Y
- ∀ paths v→exit go through Y

i.e. X can determine whether or not Y is executed.

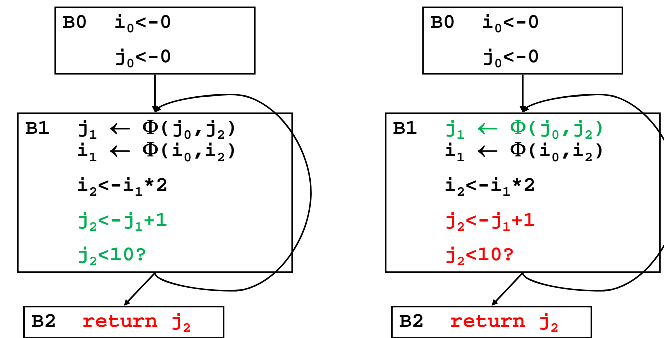


Aggressive Dead Code Elimination

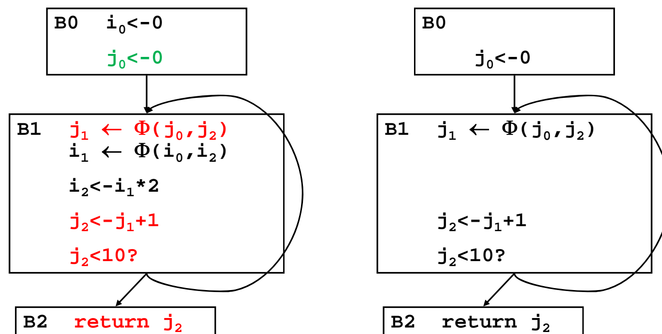
Assume a statement is dead until proven otherwise.

```
while (|W| > 0) {
    S <- W.removeOne()
    if (S is live) continue;
    mark S live, insert:
        - forall operands, S.operand.definers into W
        - S.CD-1 into W
}
```

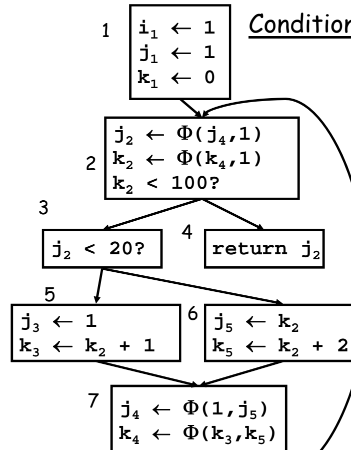

Example DCE



Example DCE



Conditional Constant Propagation

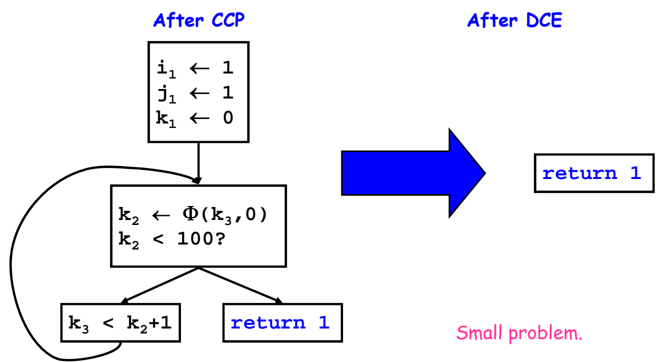


(Recall from earlier.)

- Does block 6 ever execute?
- Simple CP can't tell
- Conditional CP can tell:
  - Assumes blocks don't execute until proven otherwise
  - Assumes values are constants until proven otherwise

Applying Dead Code Elimination to the Result of CCP

After CCP **After DCE**

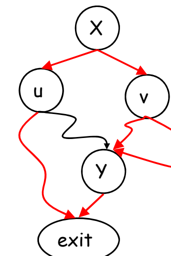


Small problem.

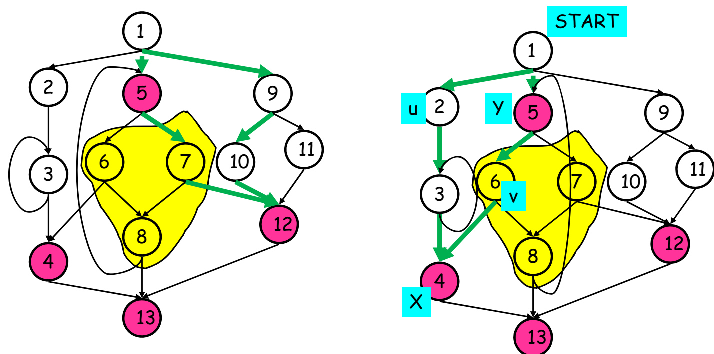
Finding the Control Dependence Graph

Y is control-dependent on X if
- X branches to u and v
- ∃ a path u→exit which does not go through Y
- ∀ paths v→exit go through Y

i.e. X can determine whether or not Y is executed.



Dominance Frontier and Path Convergence

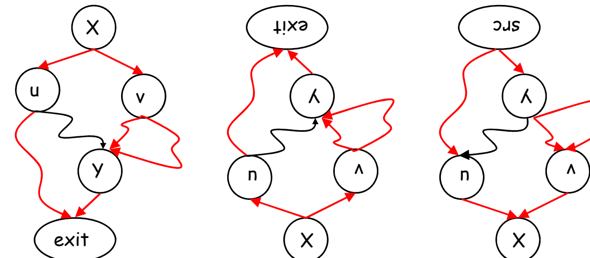


Any ideas?

Finding the Control Dependence Graph

Y is control-dependent on X if
- X branches to u and v
- ∃ a path u→exit which does not go through Y
- ∀ paths v→exit go through Y

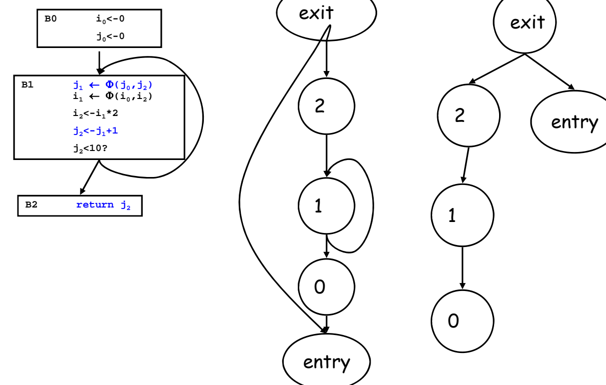
i.e. X can determine whether or not Y is executed.



Finding the CDG

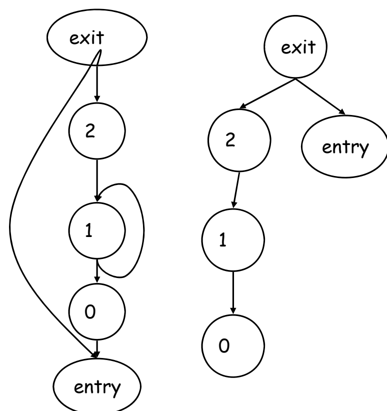
- Construct CFG
- Add entry node and exit node
- Add (entry,exit)
- Create G', the reverse CFG
- Compute D-tree in G' (post-dominators of G)
- Compute $DF_{G'}(y)$ for all $y \in G'$ (post-DF of G)
- Add $(x,y) \in G$ to CDG if $x \in DF_{G'}(y)$

CDG of example



exit
2
entry
1
0

CDG of example



exit: {}
2: {entry}
1: {1,entry}
0: {entry}
entry: {}