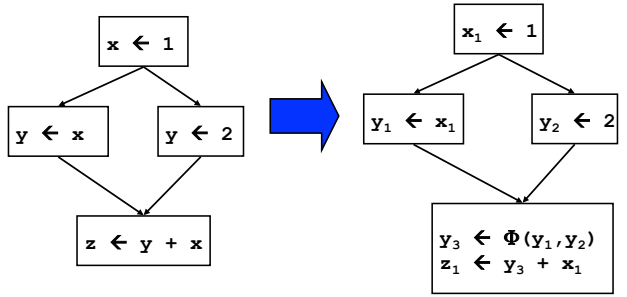
SSA-Style Optimizations

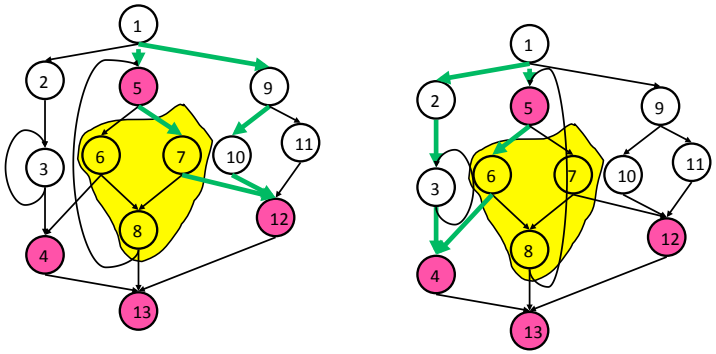
(Slide content courtesy of Seth Goldstein.)

Review: Minimal SSA

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



Review: Dominance Frontier and Path Convergence



Constant Propagation

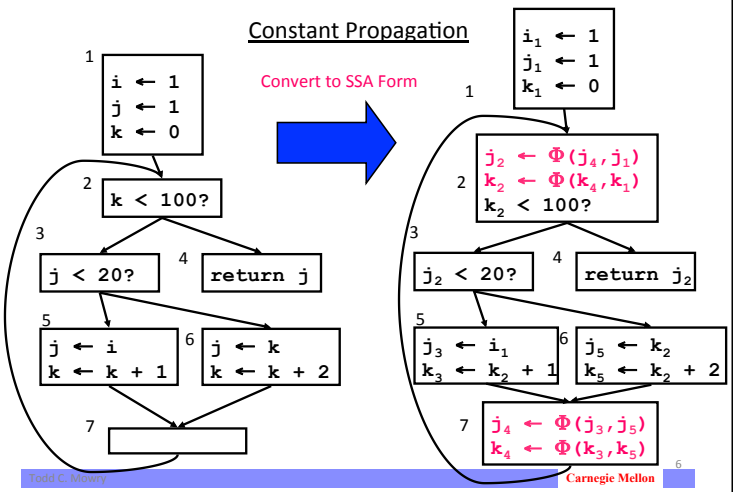
- If "v ← c", replace all uses of v with c
- If "v ← Φ(c,c,c)" (each input is the same constant), replace all uses of v with c

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

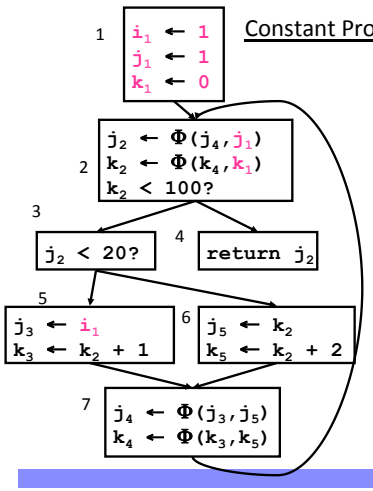
Other Optimizations with SSA

- Copy propogation
  - delete "x ← Φ(y,y,y)" and replace all x with y
  - delete "x ← y" and replace all x with y
- Constant Folding
  - (Also, constant conditions too!)

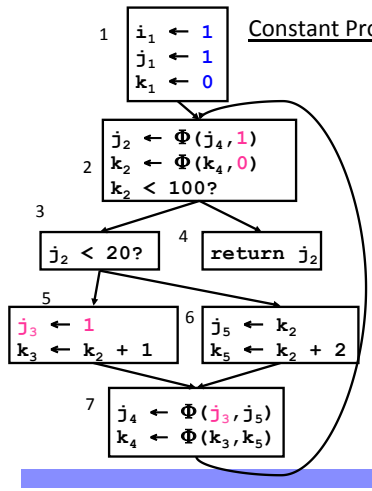
Constant Propagation



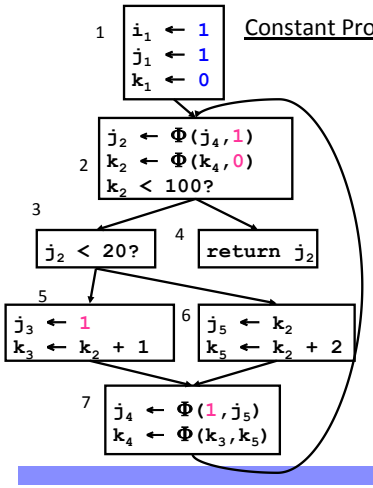
Constant Propagation



Constant Propagation

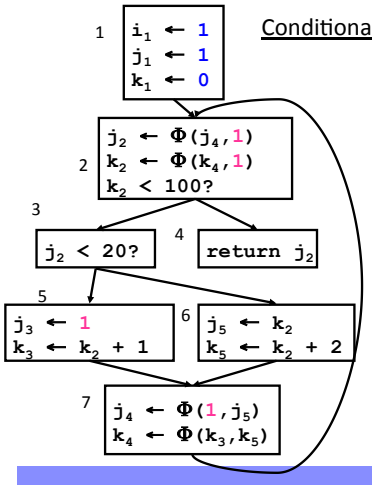


Constant Propagation



Not a very exciting result (yet)…

Conditional Constant Propagation



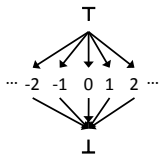
- Does block 6 ever execute?
- Simple Constant Propagation can't tell
- But "Conditional Const. Prop." *can* tell:
  - Assumes blocks don't execute until proven otherwise
  - Assumes values are constants until proven otherwise

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:

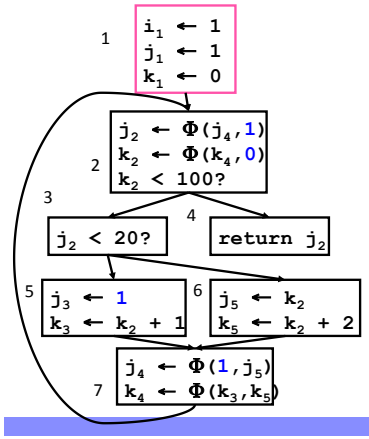


not executed

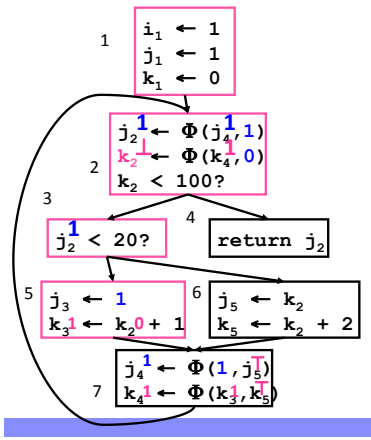
we have seen evidence that the variable has been assigned a constant with the value

we have seen evidence that the variable can hold different values at different times

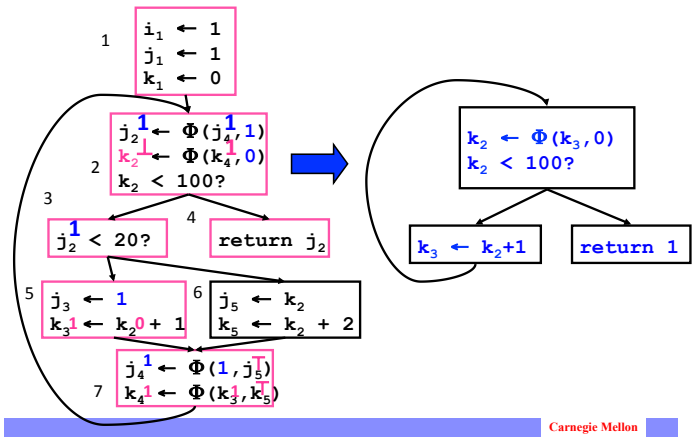
Conditional Constant Propagation



Conditional Constant Propagation



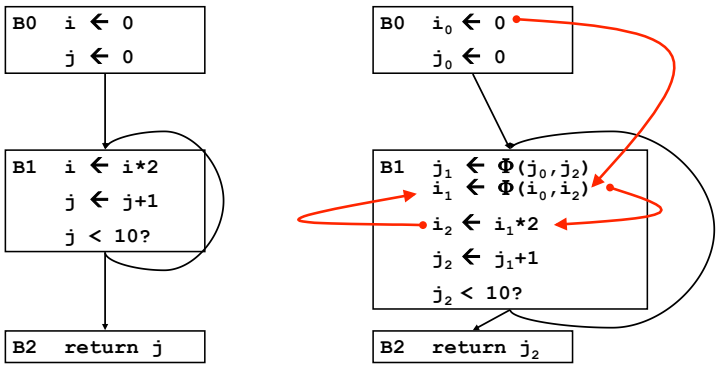
Conditional Constant Propagation



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.operands.definers {
          def.users ← def.users - {S}
          if |def.users| == 0 then
            W ← W UNION {def}
        }
        delete S
}
```


Example DCE



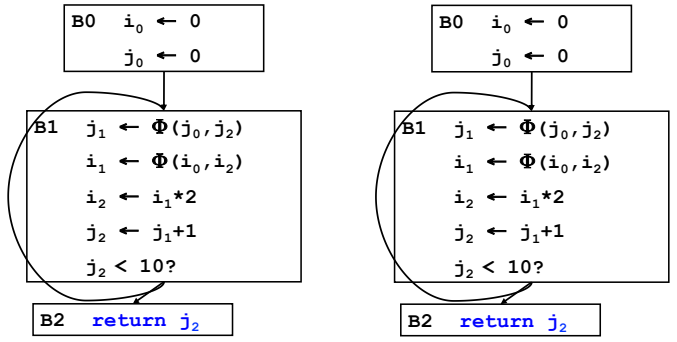
Standard DCE leaves Zombies!

Aggressive Dead Code Elimination

Assume a statement is dead until proven otherwise.

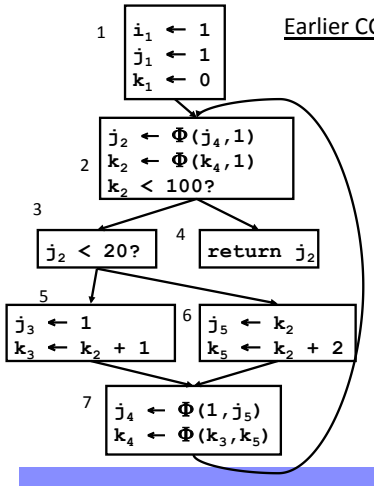
```
init:
   mark as live all stmts that have side-effects:
       - I/O
       - stores into memory
       - returns
       - calls a function that MIGHT have side-effects
   As we mark S live, insert S.operands.definers into W

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


Example DCE

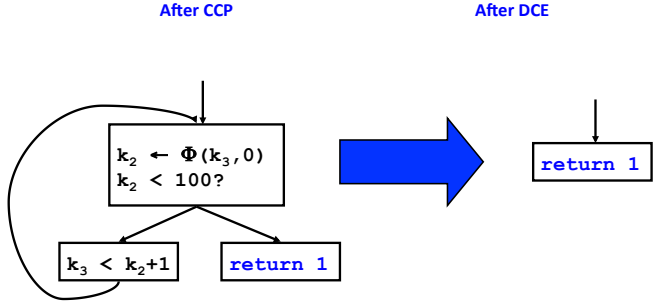


Problem!

Earlier CCP Example Revisited



Applying Dead Code Elimination to the Result of CCP

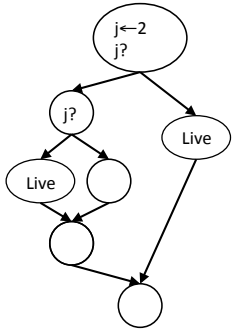


Small problem.

Fixing DCE

if S is live, then

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

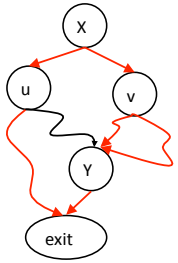


Control Dependence

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.



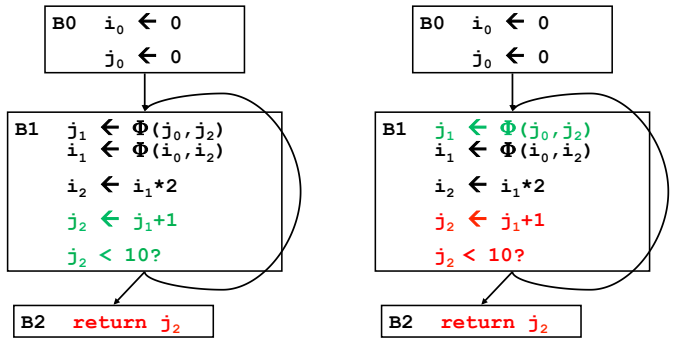
Aggressive Dead Code Elimination (Fixed Version)

Assume a statement is dead until proven otherwise.

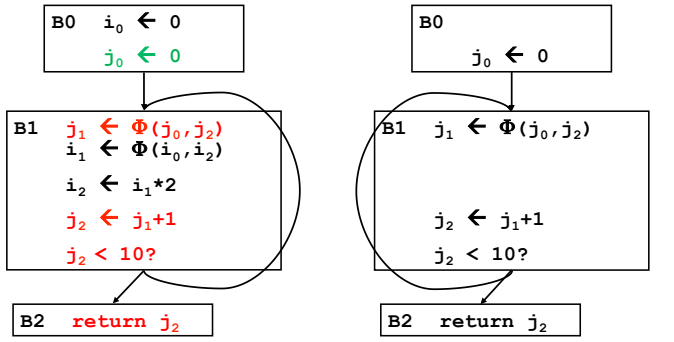
```
init:
   mark as live all stmts that have side-effects:
       - I/O
       - stores into memory
       - returns
       - calls a function that MIGHT have side-effects
   As we mark S live, insert:
       - S.operands.definers into W
       - S.CD^-1 into W

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

Conditional tests for blocks upon which S are control-dependent are also inserted into work-list.

Example DCE



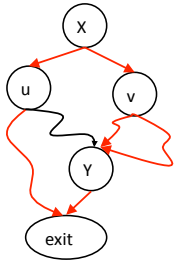
Example DCE



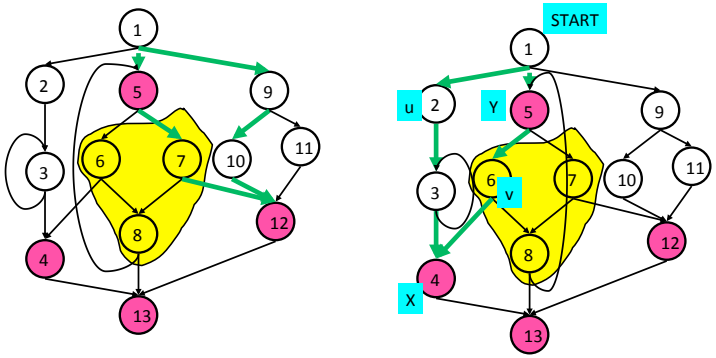
Finding the Control Dependence Graph

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

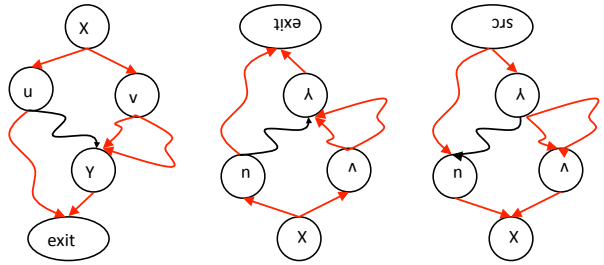


Any ideas?

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

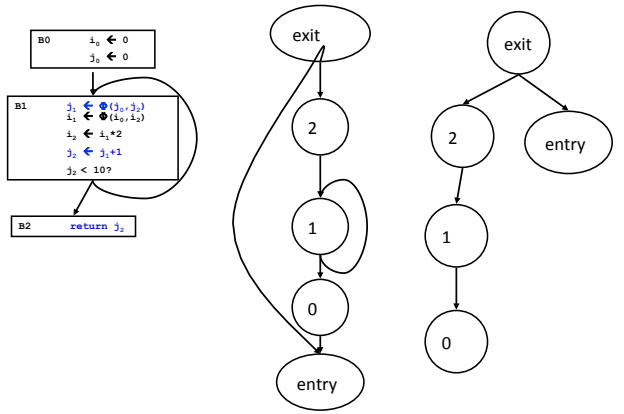
i.e. X can determine whether or not Y is executed.



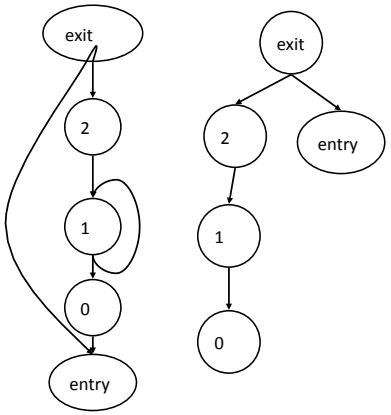
Finding the CDG

- 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



CDG of example



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