Lecture 12

Introduction to

Static Single Assignment (SSA)

(Portions of slides courtesy of Seth Goldstein.)

Values ≠ Locations

```
…
for (i=0; i++; i<10) {
   … = … i …;
   …
}
for (i=j; i++; i<20) {
   …  = i …
}
```

Def-use chains help solve the problem.

Def-Use Chains are Expensive

```
foo(int i, int j) {
  …
  switch (i) {
  case 0: x=3;break;
  case 1: x=1; break;
  case 2: x=6; break;
  case 3: x=7; break;
  default: x = 11;
  }
  switch (j) {
  case 0: y=x+7; break;
  case 1: y=x+4; break;
  case 2: y=x-2; break;
  case 3: y=x+1; break;
  default: y=x+9;
  }
  …
```

In general,
- N defs
- M uses
- ⇒ O(NM) space and time

<u>One solution</u>: limit each variable to ONE definition site

Def-Use Chains are Expensive

```
foo(int i, int j) {
  …
  switch (i) {
  case 0: x=3; break;
  case 1: x=1; break;
  case 2: x=6;
  case 3: x=7;
  default: x = 11;
  }
  x1 is one of the above x's
  switch (j) {
  case 0: y=x1+7;
  case 1: y=x1+4;
  case 2: y=x1-2;
  case 3: y=x1+1;
  default: y=x1+9;
  }
  …
```

<u>One solution</u>: limit each variable to ONE definition site

Advantages of SSA

- Makes du-chains explicit
- Makes dataflow analysis easier
- Improves register allocation
  - Automatically builds Webs
  - Makes building interference graphs easier
- For most programs reduces space/time requirements

SSA

- Static single assignment is an IR where every variable is assigned a value at most once in the program text
- Easy for a basic block:
  - assign to a fresh variable at each stmt.
  - each use uses the most recently defined var.
  - (Similar to Value Numbering)

Straight-line SSA

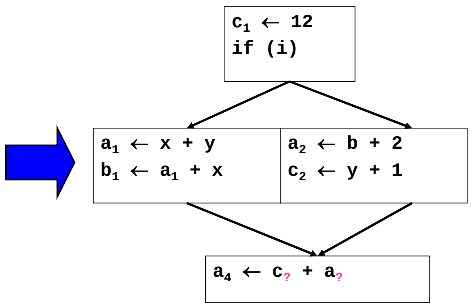
$$a \leftarrow x + y \qquad a_1 \leftarrow x + y$$
$$b \leftarrow a + x \qquad b_1 \leftarrow a_1 + x$$
$$a \leftarrow b + 2 \quad \Rightarrow \quad a_2 \leftarrow b_1 + 2$$
$$c \leftarrow y + 1 \qquad c_1 \leftarrow y + 1$$
$$a \leftarrow c + a \qquad a_3 \leftarrow c_1 + a_2$$

SSA

- Static single assignment is an IR where every variable is assigned a value at most once in the program text
- Easy for a basic block:
  - assign to a fresh variable at each stmt.
  - each use uses the most recently defined var.
  - (Similar to Value Numbering)
- *What about at joins in the CFG?*

Merging at Joins

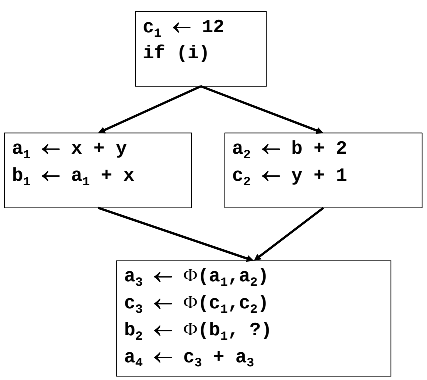
```
c ← 12
if (i) {
   a ← x + y
   b ← a + x
} else {
   a ← b + 2
   c ← y + 1
}
a ← c + a
```



SSA

- Static single assignment is an IR where every variable is assigned a value at most once in the program text
- Easy for a basic block:
  - assign to a fresh variable at each stmt.
  - Each use uses the most recently defined var.
  - (Similar to Value Numbering)
- What about at joins in the CFG?
  - Use a notational fiction: a Φ function

Merging at Joins



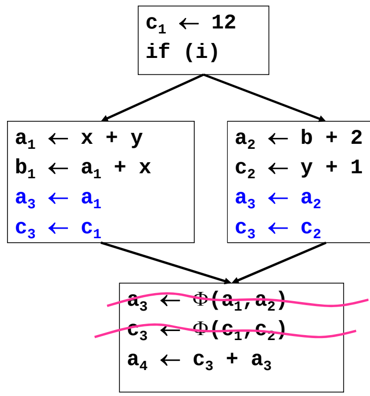
The Φ function

- Φ merges multiple definitions along multiple control paths into a single definition.
- At a basic block with *p* predecessors, there are *p* arguments to the Φ function.

$$x_{new} \leftarrow \Phi(x_1, x_1, x_1, \ldots , x_p)$$

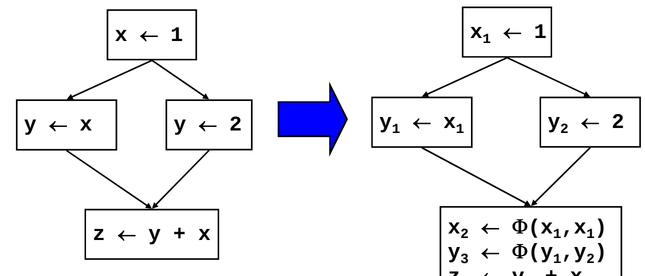
- How do we choose which x_i to use?
  - We don't really care!
  - If we care, use moves on each incoming edge

"Implementing" Φ



Trivial SSA

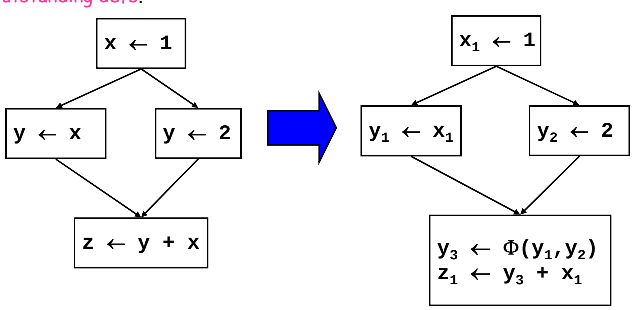
- Each assignment generates a fresh variable.
- At each join point insert Φ functions for all live variables.



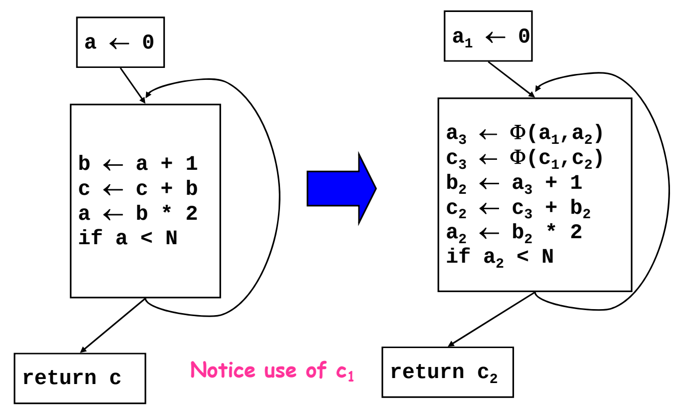
Way too many Φ functions inserted.

Minimal SSA

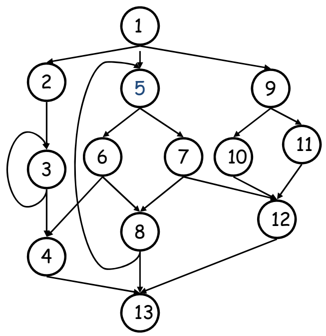
- Each assignment generates a fresh variable.
- At each join point insert Φ functions for all live variables with multiple outstanding defs.



Another Example



When Do We Insert Φ?



CFG

If there is a def of **a** in block 5, which nodes need a Φ()?

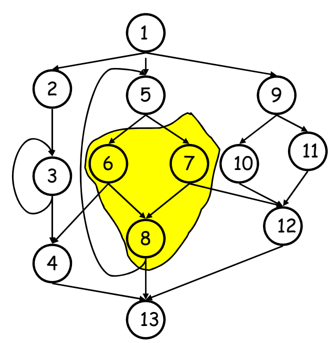
When do we insert Φ?

- We insert a Φ function for variable **A** in block Z iff:
  - **A** was defined more than once before
    - (i.e., **A** defined in X and Y AND X ≠ Y)
  - There exists a non-empty path from x to z, P_{xz}, and a non-empty from y to z, P_{yz}, s.t.
    - $P_{xz} \cap P_{yz} = \{ z \}$
    - $z \notin P_{xq}$ or $z \notin P_{yr}$ where $P_{xz} = P_{xq} \rightarrow z$ and $P_{yz} = P_{yr} \rightarrow z$
- Entry block contains an implicit def of all vars
- Note: *A* = Φ(...) is a def of *A*

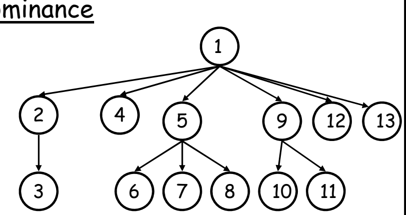
Dominance Property of SSA

- In SSA, definitions dominate uses.
  - If x_i is used in x ← Φ(..., x_i, ...), then BB(x_i) dominates i^{th} predecessor of BB(PHI)
  - If x is used in y ← ... x ..., then BB(x) dominates BB(y)
- We can use this for an efficient algorithm to convert to SSA

Dominance



CFG

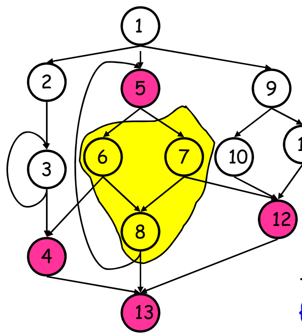


D-Tree

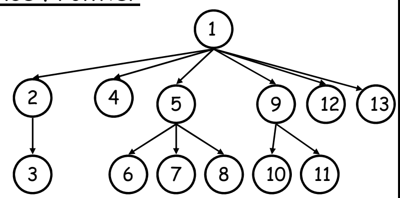
If there is a def of a in block 5, which nodes need a Φ()?

x strictly dominates w (x sdom w) iff x dom w AND x ≠ w

Dominance Frontier



CFG

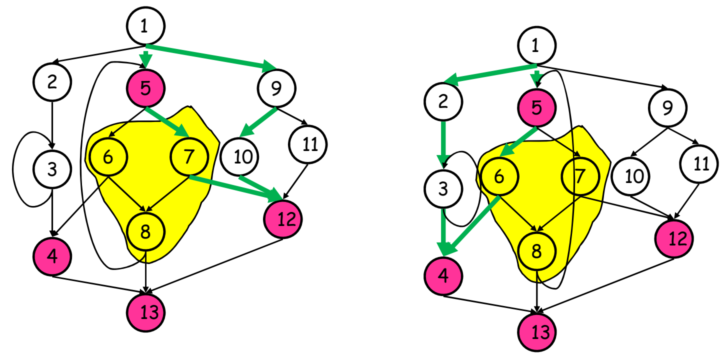


D-Tree

The Dominance Frontier of a node x = { w | x dom pred(w) AND !(x sdom w)}

x strictly dominates w (x sdom w) iff x dom w AND x ≠ w

Dominance Frontier and Path Convergence



Using Dominance Frontier to Compute SSA

- place all Φ()
- Rename all variables

Using Dominance Frontier to Place Φ()

- Gather all the defsites of every variable
- Then, for every variable
  - foreach defsite
    - foreach node in DF(defsite)
      - if we haven't put Φ() in node, then put one in
      - if this node didn't define the variable before, then add this node to the defsites
- This essentially computes the Iterated Dominance Frontier on the fly, inserting the minimal number of Φ() neccesary

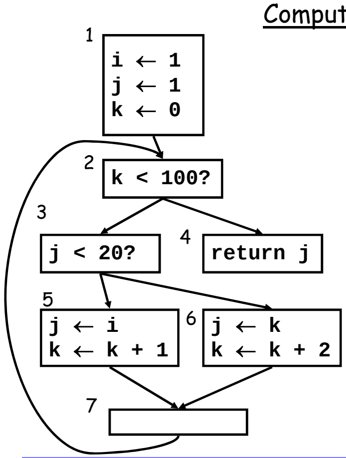
Using Dominance Frontier to Place Φ()

```
foreach node n {
   foreach variable v defined in n {
     orig[n] ∪= {v}
     defsites[v] ∪= {n}
   }
}
foreach variable v {
   W = defsites[v]
   while W not empty {
     n = remove node from W
     foreach y in DF[n]
     if y ∉ PHI[v] {
       insert "v ← Φ(v,v,…)" at top of y
       PHI[v] = PHI[v] ∪ {y}
       if v ∉ orig[y]: W = W ∪ {y}
     }
   }
}
```

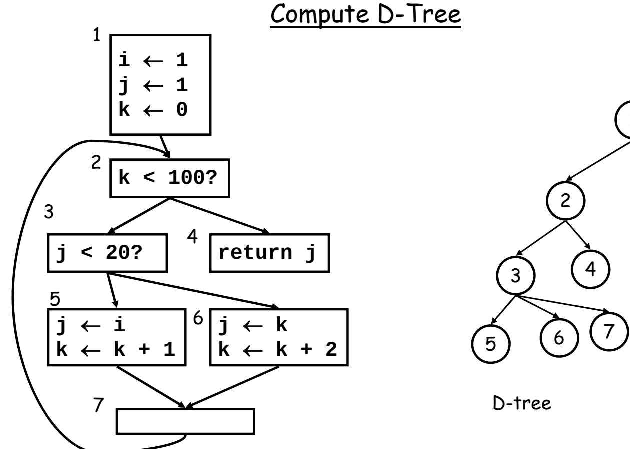

Renaming Variables

- Algorithm:
  - Walk the D-tree, renaming variables as you go
  - Replace uses with more recent renamed def
- For straight-line code this is easy
- What if there are branches and joins?
  - use the closest def such that the def is above the use in the D-tree
- Easy implementation:
  - for each var: rename (v)
  - rename(v): replace uses with top of stack
    at def: push onto stack
    call rename(v) on all children in D-tree
    for each def in this block pop from stack

Compute D-Tree

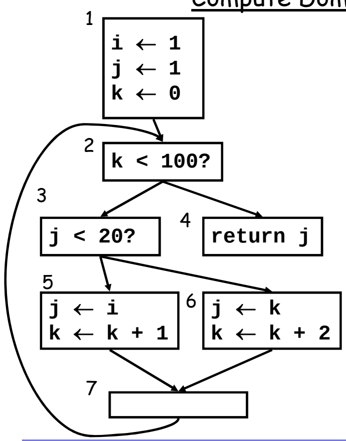


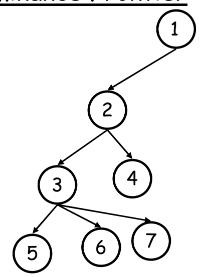
Compute D-Tree



D-tree

Compute Dominance Frontier

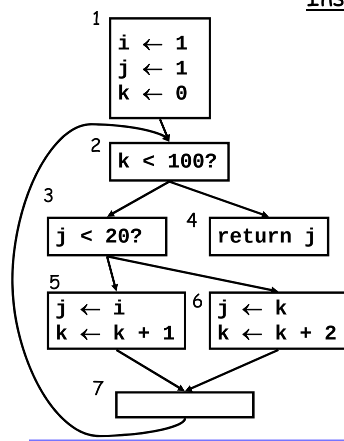




1	{}
2	{2}
3	{2}
4	{}
5	{7}
6	{7}
7	{2}

DFs

Insert Φ()



1	{}
2	{2}
3	{2}
4	{}
5	{7}
6	{7}
7	{2}

DFs

orig[n]

1	{ i,j,k}
2	{}
3	{}
4	{}
5	{j,k}
6	{j,k}
7	{}

defsites[v]

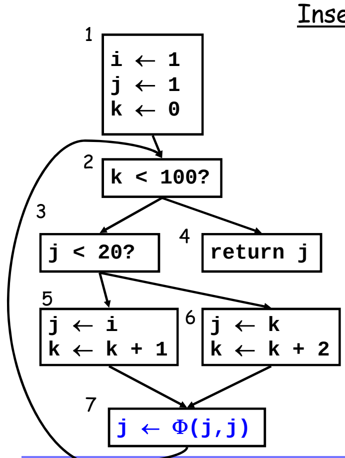
i	{1}
j	{1,5,6}
k	{1,5,6}

var i: W={1}

var j: W={1,5,6}

DF{1}, DF{5}

Insert Φ()



1	{}
2	{2}
3	{2}
4	{}
5	{7}
6	{7}
7	{2}

DFs

orig[n]

1	{ i,j,k}
2	{}
3	{}
4	{}
5	{j,k}
6	{j,k}
7	{}

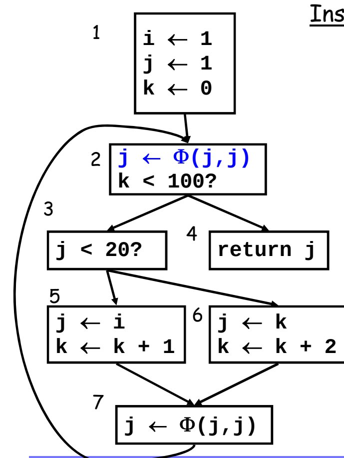
defsites[v]

i	{1}
j	{1,5,6}
k	{1,5,6}

var j: W={1,5,6}

DF{1}, DF{5}

Insert Φ()



1	{}
2	{2}
3	{2}
4	{}
5	{7}
6	{7}
7	{2}

DFs

orig[n]

1	{ i,j,k}
2	{}
3	{}
4	{}
5	{j,k}
6	{j,k}
7	{}

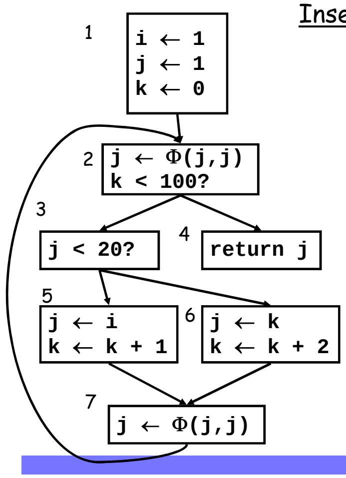
defsites[v]

i	{1}
j	{1,5,6}
k	{1,5,6}

var j: W={1,5,6}

DF{1}, DF{5}

Insert Φ()



			orig[n]
1	{}	1	{ i,j,k}
2	{2}	2	{}
3	{2}	3	{}
4	{}	4	{}
5	{7}	5	{j,k}
6	{7}	6	{j,k}
7	{2}	7	{}

defsites[v]

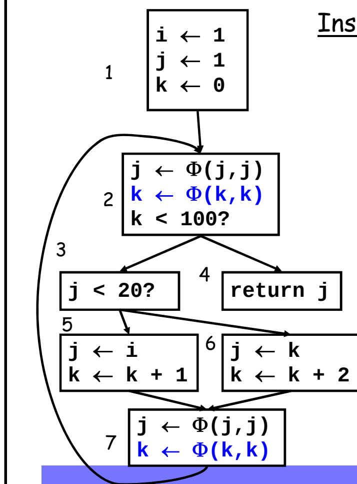
i	{1}
j	{1,5,6}
k	{1,5,6}

DFs

var j: W={1,5,6}

DF{1}, DF{5}, DF{6}

Insert Φ()



			orig[n]
1	{}	1	{ i,j,k}
2	{2}	2	{}
3	{2}	3	{}
4	{}	4	{}
5	{7}	5	{j,k}
6	{7}	6	{j,k}
7	{2}	7	{}

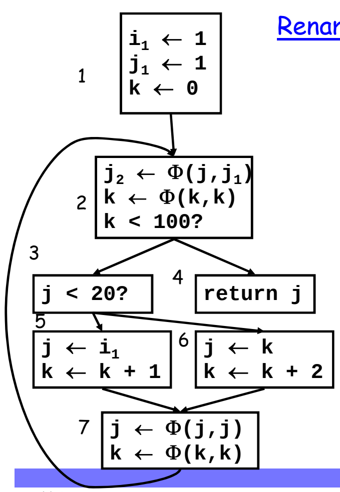
defsites[v]

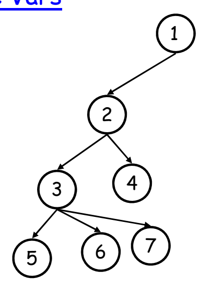
i	{1}
j	{1,5,6}
k	{1,5,6}

DFs

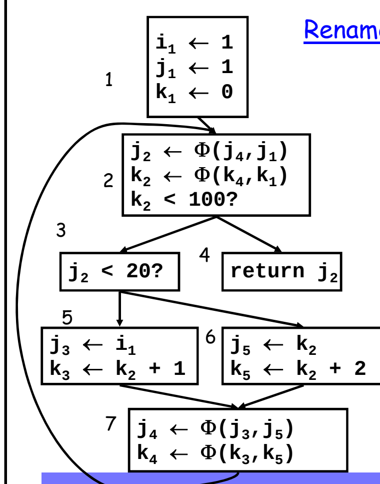
var k: W={1,5,6}

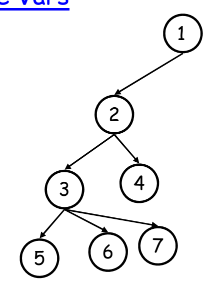
Rename Vars



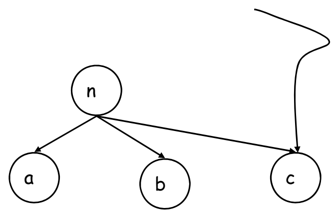


Rename Vars



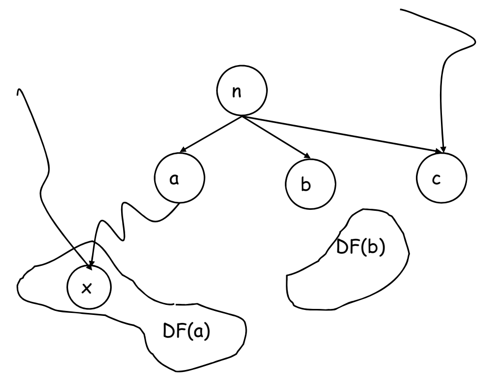


Computing DF(n)



n dom a
n dom b
!n dom c

Computing DF(n)



n dom a
n dom b
!n dom c

Computing the Dominance Frontier

The Dominance Frontier of a node x =
{ w | x dom pred(w) AND !(x sdom w)}

```
compute-DF(n)
    S = {}
    foreach node y in succ[n]
        if idom(y) ≠ n
            S = S ∪ { y }
    foreach child of n, c, in D-tree
        compute-DF(c)
        foreach w in DF[c]
            if !n dom w
                S = S ∪ { w }
    DF[n] = S
```


SSA Properties

- Only 1 assignment per variable
- Definitions dominate uses