

# 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  
 $\Rightarrow O(NM)$  space and time

One solution: 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;
    }
    ...
}

```

One solution: 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

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

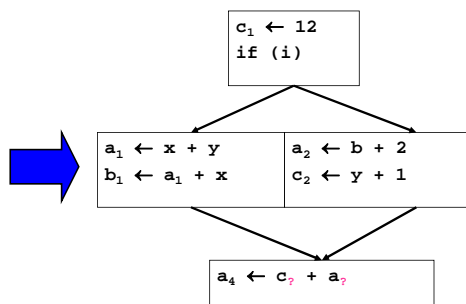
## SSA

- **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

```

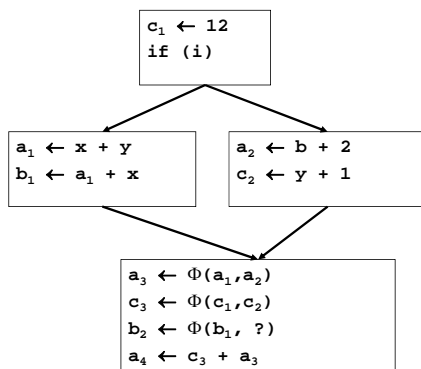
c ← 12
if (i) {
  a ← x + y
  b ← a + x
} else {
  a ← b + 2
  c ← y + 1
}
a ← c + a
    
```



### 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  $\Phi$  function

### Merging at Joins



### The $\Phi$ function

- $\Phi$  merges multiple definitions along multiple control paths into a single definition.
- At a basic block with  $p$  predecessors, there are  $p$  arguments to the  $\Phi$  function.
 
$$x_{new} \leftarrow \Phi(x_1, x_1, x_1, \dots, x_p)$$
- How do we choose which  $x_i$  to use?
  - We don't really care!
  - If we care, use moves on each incoming edge

### "Implementing" $\Phi$

```

c1 ← 12
if (i)
  a1 ← x + y
  b1 ← a1 + x
  a3 ← a1
  c3 ← c1
  a2 ← b + 2
  c2 ← y + 1
  a3 ← a2
  c3 ← c2
  a3 ← Φ(a1, a2)
  c3 ← Φ(c1, c2)
  a4 ← c3 + a3

```

15-745: Intro to SSA 13 Carnegie Mellon Todd C. Mowry

### Trivial SSA

- Each assignment generates a fresh variable.
- At each join point insert  $\Phi$  functions for *all live variables*.

```

x ← 1
y ← x
y ← 2
z ← y + x

x1 ← 1
y1 ← x1
y2 ← 2
x2 ← Φ(x1, x1)
y3 ← Φ(y1, y2)
z1 ← y3 + x2

```

Way too many  $\Phi$  functions inserted.

15-745: Intro to SSA 14 Carnegie Mellon Todd C. Mowry

### Minimal SSA

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

```

x ← 1
y ← x
y ← 2
z ← y + x

x1 ← 1
y1 ← x1
y2 ← 2
y3 ← Φ(y1, y2)
z1 ← y3 + x1

```

15-745: Intro to SSA 15 Carnegie Mellon Todd C. Mowry

### Another Example

```

a ← 0
b ← a + 1
c ← c + b
a ← b * 2
if a < N
  return c
  a1 ← 0
  a3 ← Φ(a1, a2)
  c3 ← Φ(c1, c2)
  b2 ← a3 + 1
  c2 ← c3 + b2
  a2 ← b2 * 2
  if a2 < N
    return c2

```

Notice use of  $c_1$

15-745: Intro to SSA 16 Carnegie Mellon Todd C. Mowry

### When Do We Insert $\Phi$ ?

CFG

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

Carnegie Mellon

15-745: Intro to SSA 17 Todd C. Mowry

### When do we insert $\Phi$ ?

- We insert a  $\Phi$  function for variable  $A$  in block  $Z$  iff:
  - $A$  was defined more than once before
    - (i.e.,  $A$  defined in  $X$  and  $Y$  AND  $X \neq Y$ )
  - There exists a non-empty path from  $x$  to  $z$ ,  $P_{xz}$ , and a non-empty path 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:  $A = \Phi(\dots)$  is a def of  $A$

Carnegie Mellon

15-745: Intro to SSA 18 Todd C. Mowry

### Dominance Property of SSA

- In SSA, **definitions dominate uses**.
  - If  $x_i$  is used in  $x \leftarrow \Phi(\dots, x_i, \dots)$ , then  $BB(x_i)$  dominates  $i^{\text{th}}$  predecessor of  $BB(x)$
  - If  $x$  is used in  $y \leftarrow \dots x \dots$ , then  $BB(x)$  dominates  $BB(y)$
- We can use this for an **efficient algorithm to convert to SSA**

Carnegie Mellon

15-745: Intro to SSA 19 Todd C. Mowry

### Dominance

CFG D-Tree

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

$x$  strictly dominates  $w$  ( $x \text{ sdom } w$ ) iff  $x \text{ dom } w$  AND  $x \neq w$

Carnegie Mellon

15-745: Intro to SSA 20 Todd C. Mowry

### Dominance Frontier

The **Dominance Frontier** of a node  $x = \{ w \mid x \text{ dom pred}(w) \text{ AND } !(x \text{ sdom } w) \}$

CFG D-Tree

$x$  strictly dominates  $w$  ( $x \text{ sdom } w$ ) iff  $x \text{ dom } w$  AND  $x \neq w$

15-745: Intro to SSA 21 Carnegie Mellon  
Todd C. Mowry

### Dominance Frontier and Path Convergence

15-745: Intro to SSA 22 Carnegie Mellon  
Todd C. Mowry

### Using Dominance Frontier to Compute SSA

- place all  $\Phi()$
- Rename all variables

15-745: Intro to SSA 23 Carnegie Mellon  
Todd C. Mowry

### Using Dominance Frontier to Place $\Phi()$

- Gather all the defsites of every variable
- Then, for **every variable**
  - foreach **defsite**
    - foreach **node in DF(defsite)**
      - if we haven't put  $\Phi()$  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  $\Phi()$  necessary**

15-745: Intro to SSA 24 Carnegie Mellon  
Todd C. Mowry

## Using Dominance Frontier to Place $\Phi()$

```

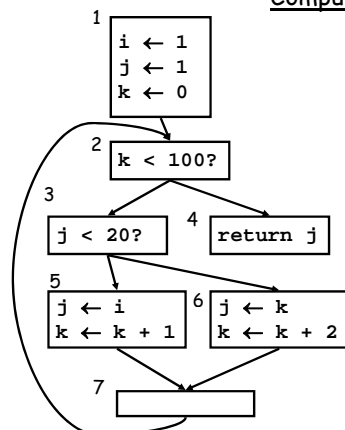
foreach node n {
  foreach variable v defined in n {
    orig[n]  $\cup$ = {v}
    defsites[v]  $\cup$ = {n}
  }
}
foreach variable v {
  W = defsites[v]
  while W not empty {
    n = remove node from W
    foreach y in DF[n]
      if y  $\notin$  PHI[v] {
        insert "v  $\leftarrow \Phi(v,v,\dots)$ " at top of y
        PHI[v] = PHI[v]  $\cup$  {y}
        if v  $\notin$  orig[y]: W = W  $\cup$  {y}
      }
  }
}

```

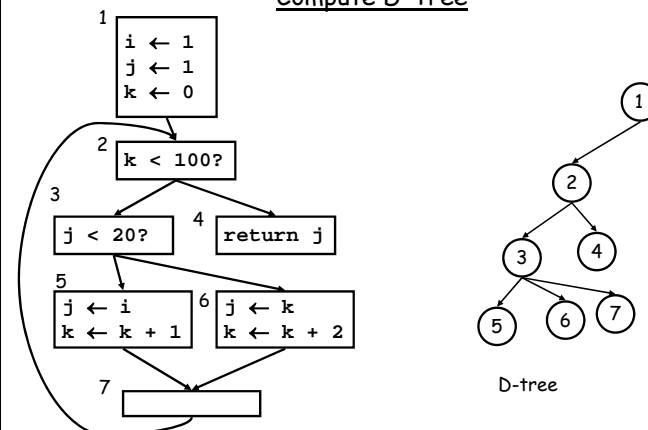
## Renaming Variables

- **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: i ← 1
   j ← 1
   k ← 0
2: k < 100?
3: j < 20?
4: return j
5: j ← i
   k ← k + 1
6: j ← k
   k ← k + 2
7:

```

Control flow graph structure:

- Node 1 (Root) branches to Node 2.
- Node 2 branches to Node 3 and Node 4.
- Node 3 branches to Node 5 and Node 6.
- Node 5 and Node 6 both branch to Node 7.
- Node 7 branches back to Node 2.

DFs

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

Todd C. Mowry 15-745: Intro to SSA Carnegie Mellon 29

### Insert $\Phi()$

```

1: i ← 1
   j ← 1
   k ← 0
2: k < 100?
3: j < 20?
4: return j
5: j ← i
   k ← k + 1
6: j ← k
   k ← k + 2
7:

```

Control flow graph structure:

- Node 1 (Root) branches to Node 2.
- Node 2 branches to Node 3 and Node 4.
- Node 3 branches to Node 5 and Node 6.
- Node 5 and Node 6 both branch to Node 7.
- Node 7 branches back to Node 2.

DFs

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

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}

Todd C. Mowry 15-745: Intro to SSA Carnegie Mellon 30

### Insert $\Phi()$

```

1: i ← 1
   j ← 1
   k ← 0
2: k < 100?
3: j < 20?
4: return j
5: j ← i
   k ← k + 1
6: j ← k
   k ← k + 2
7: j ← Φ(j,j)

```

Control flow graph structure:

- Node 1 (Root) branches to Node 2.
- Node 2 branches to Node 3 and Node 4.
- Node 3 branches to Node 5 and Node 6.
- Node 5 and Node 6 both branch to Node 7.
- Node 7 branches back to Node 2.

DFs

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

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}

Todd C. Mowry 15-745: Intro to SSA Carnegie Mellon 31

### Insert $\Phi()$

```

1: i ← 1
   j ← 1
   k ← 0
2: j ← Φ(j,j)
   k < 100?
3: j < 20?
4: return j
5: j ← i
   k ← k + 1
6: j ← k
   k ← k + 2
7: j ← Φ(j,j)

```

Control flow graph structure:

- Node 1 (Root) branches to Node 2.
- Node 2 branches to Node 3 and Node 4.
- Node 3 branches to Node 5 and Node 6.
- Node 5 and Node 6 both branch to Node 7.
- Node 7 branches back to Node 2.

DFs

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

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}

Todd C. Mowry 15-745: Intro to SSA Carnegie Mellon 32



**Insert  $\Phi()$**

```

1  i ← 1
   j ← 1
   k ← 0
2  j ←  $\Phi(j, j)$ 
   k < 100?
3  j < 20?
4  return j
5  j ← i
   k ← k + 1
6  j ← k
   k ← k + 2
7  j ←  $\Phi(j, j)$ 

```

orig[n]		defsites[v]	
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	{}

**DFs**

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

Todd C. Mowry      15-745: Intro to SSA      Carnegie Mellon      33

**Insert  $\Phi()$**

```

1  i ← 1
   j ← 1
   k ← 0
2  j ←  $\Phi(j, j)$ 
   k ←  $\Phi(k, k)$ 
   k < 100?
3  j < 20?
4  return j
5  j ← i
   k ← k + 1
6  j ← k
   k ← k + 2
7  j ←  $\Phi(j, j)$ 
   k ←  $\Phi(k, k)$ 

```

orig[n]		defsites[v]	
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	{}

**DFs**

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

Todd C. Mowry      15-745: Intro to SSA      Carnegie Mellon      34

**Rename Vars**

```

1  i1 ← 1
   j1 ← 1
   k ← 0
2  j2 ←  $\Phi(j, j_1)$ 
   k ←  $\Phi(k, k)$ 
   k < 100?
3  j < 20?
4  return j
5  j ← i1
   k ← k + 1
6  j ← k
   k ← k + 2
7  j ←  $\Phi(j, j)$ 
   k ←  $\Phi(k, k)$ 

```

Todd C. Mowry      15-745: Intro to SSA      Carnegie Mellon      35

**Rename Vars**

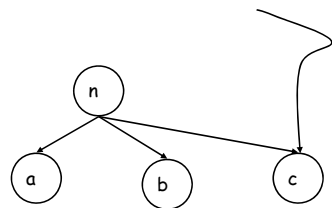
```

1  i1 ← 1
   j1 ← 1
   k1 ← 0
2  j2 ←  $\Phi(j_4, j_1)$ 
   k2 ←  $\Phi(k_4, k_1)$ 
   k2 < 100?
3  j2 < 20?
4  return j2
5  j3 ← i1
   k3 ← k2 + 1
6  j5 ← k2
   k5 ← k2 + 2
7  j4 ←  $\Phi(j_3, j_5)$ 
   k4 ←  $\Phi(k_3, k_5)$ 

```

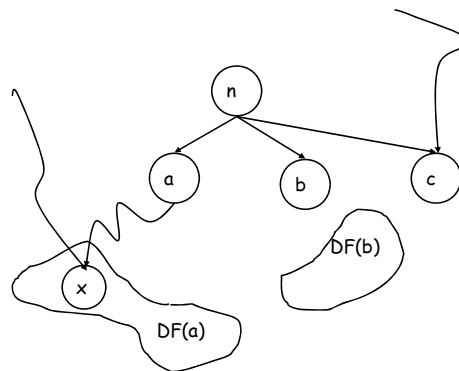
Todd C. Mowry      15-745: Intro to SSA      Carnegie Mellon      36

### 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

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

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

### SSA Properties

- Only 1 assignment per variable
- Definitions dominate uses