Lecture 13
Introduction to Static Single Assignment (SSA)

(Slides courtesy of Seth Goldstein.)

Todd C. Mowry 15-745: Intro to SSA 1

Values ≠ Locations

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

Def-use chains help solve the problem.

Todd C. Mowry 15-745: Intro to SSA 2

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
```

One solution: limit each variable to ONE definition site

Todd C. Mowry 15-745: Intro to SSA 3

Def-Use Chains are Expensive

```
foo(int i, int j) {
  switch (i) {case 0: x1=x; break;
  case 1: x1=x1; break;
  case 2: x1=x2; break;
  case 3: x1=x3; break;
  default: x1 = x11;
  }
  switch (j) {case 0: y=x1+7; break;
  case 1: y=x1+4; break;
  case 2: y=x1-2; break;
  case 3: y=x1+3; break;
  default: y=x1+9;
  }
}
```

x1 is one of the above x’s

One solution: limit each variable to ONE definition site

Todd C. Mowry 15-745: Intro to SSA 4
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

\[
\begin{align*}
a & \leftarrow x + y \\
b & \leftarrow a + x \\
a & \leftarrow b + 2 \\
c & \leftarrow y + 1 \\
a & \leftarrow c + a \\
a_1 & \leftarrow x + y \\
b_1 & \leftarrow a_1 + x \\
a_2 & \leftarrow b_1 + 2 \\
c_1 & \leftarrow y + 1 \\
a_3 & \leftarrow c_1 + a_2
\end{align*}
\]
Merging at Joins

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

a ← a1 + a2
```

### 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

\( \Phi \) 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.
- How do we choose which \( x_i \) to use?
  - We don’t really care!
  - If we care, use moves on each incoming edge
"Implementing" $\Phi$

- $c_1 \leftarrow 12$
- \text{if (i)}
  - $a_2 \leftarrow b + 2$
  - $c_2 \leftarrow y + 1$
  - $a_3 \leftarrow a_2$
  - $c_3 \leftarrow c_2$

- $a_1 \leftarrow x + y$
- $b_1 \leftarrow a_1 + x$
- $a_3 \leftarrow a_1$
- $c_3 \leftarrow c_1$

Trivial SSA

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

Way too many $\Phi$ functions inserted.

Minimal SSA

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

Another Example

- $a \leftarrow 0$
- $\Phi(a_1, a_2)$
- $c_3 \leftarrow \Phi(c_1, c_2)$
- $b_2 \leftarrow a_3 + 1$
- $c_2 \leftarrow c_1 + b_2$
- $a_3 \leftarrow b_2 * 2$
- \text{if $a_2 < N$}

Notice use of $c_1$
When Do We Insert $\phi$?

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

$\text{CFG}$

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.
    - $P_{xz} \cap P_{yz} = \{ z \}$
    - $z \notin P_{xy}$ or $z \notin P_{yx}$ where $P_{xy} = P_{xy} \rightarrow z$ and $P_{yx} = P_{yx} \rightarrow z$
  - Entry block contains an implicit def of all vars
- Note: $A = \phi(\ldots)$ is a def of $A$

Dominance Property of SSA

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

Dominance

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

$\text{CFG}$

$\text{D-Tree}$

$x$ strictly dominates $w$ ($x \text{ sdom } w$) iff $x$ dom $w$ AND $x \neq w$
Using Dominance Frontier to Compute SSA

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

Using Dominance Frontier to Place $\phi()$

- Gather all the defsites of every variable
- Then, for every variable
  - foreach defsite
    - foreach node in $\text{Dominance Frontier(defsite)}$
      - 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
Using Dominance Frontier to Place $\phi()$

foreach node $n$ {
    foreach variable $v$ defined in $n$ {
        $\text{orig}[n] \cup= \{v\}$
        $\text{defsites}[v] \cup= \{n\}$
    }
}

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

Renaming Variables

- **Algorithm:**
  - Walk the D-tree, renaming variables as you go
  - Replace uses with most 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 Dominance Tree

Compute Dominance Frontiers
Let's insert \( \Phi() \) and compute \( \Phi() \):

1. \( i \leftarrow 1 \)
2. \( j \leftarrow 1 \)
3. \( k \leftarrow 0 \)
4. \( j \leftarrow \Phi(j,j) \)
5. \( k \leftarrow \Phi(k,k) \)
6. \( k < 100? \)
7. \( j < 20? \)
8. \( \text{return } j \)
9. \( j \leftarrow 1 \)
10. \( k \leftarrow k + 1 \)
11. \( j \leftarrow j \)
12. \( k \leftarrow k + 2 \)
13. \( j \leftarrow \Phi(j,j) \)
14. \( k \leftarrow \Phi(k,k) \)

**DFs**

1. \( i \leftarrow 1 \) \( \{i,j,k\} \)
2. \( j \leftarrow \Phi(j,j) \) \( \{j\} \)
3. \( k \leftarrow \Phi(k,k) \) \( \{k\} \)
4. \( k < 100? \) \( \text{defsites[v]} \)
5. \( j < 20? \) \( \text{return } j \)

**Rename Vars**

1. \( i_1 \leftarrow 1 \)
2. \( j_1 \leftarrow 1 \)
3. \( k_1 \leftarrow 0 \)
4. \( j_2 \leftarrow \Phi(j_1,j_1) \) \( \{j_1\} \)
5. \( k_2 \leftarrow \Phi(k_1,k_1) \) \( \{k_1\} \)
6. \( k_2 < 100? \)
7. \( j_2 < 20? \) \( \text{return } j_2 \)
8. \( j_3 \leftarrow 1 \)
9. \( k_3 \leftarrow k_1 + 1 \)
10. \( j_3 \leftarrow j_1 \)
11. \( k_3 \leftarrow k_1 + 2 \)
12. \( j_4 \leftarrow \Phi(j_3,j_3) \) \( \{j_3\} \)
13. \( k_4 \leftarrow \Phi(k_3,k_3) \) \( \{k_3\} \)

**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 w in DF[n]
         S = S ∪ { w }
DF[n] = S

SSA Properties

- Only 1 assignment per variable
- Definitions dominate uses