Static Single Assignment

Values ≠ Locations

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

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

A solution is to limit each var to ONE def site

A solution is to limit each var to ONE def site

Copyright © Seth Copen Goldstein 2001
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

\[
\begin{align*}
  a & \leftarrow x + y \\
  b & \leftarrow a + x \\
  a & \leftarrow b + 2 \\
  c & \leftarrow y + 1 \\
  a & \leftarrow c + a
\end{align*}
\]

Merging at Joins

\[
\begin{align*}
  c & \leftarrow 12 \\
  \text{if (i) } & \\
  \{ \\
  a & \leftarrow x + y \\
  b & \leftarrow a + x \\
  \} \text{ else } \\
  \{ \\
  a & \leftarrow b + 2 \\
  c & \leftarrow y + 1 \\
  \} \\
  a & \leftarrow c + a
\end{align*}
\]
**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 notional fiction: A $\Phi$ function

**Merging at Joins**

```
\begin{align*}
\text{c}_1 & \leftarrow 12 \\
\text{if (i)} \\
\text{\quad a}_1 & \leftarrow x + y \\
\text{\quad b}_1 & \leftarrow a_1 + x \\
\text{\quad c}_1 & \leftarrow c_1 \\
\text{\quad a}_2 & \leftarrow b + 2 \\
\text{\quad c}_2 & \leftarrow y + 1 \\
\end{align*}
```

```
\begin{align*}
\text{a}_1 & \leftarrow \Phi(\text{a}_1, \text{a}_2) \\
\text{c}_1 & \leftarrow \Phi(\text{c}_1, \text{c}_2) \\
\text{b}_2 & \leftarrow \Phi(\text{b}_1, \text{?}) \\
\text{a}_4 & \leftarrow c_3 + a_3
\end{align*}
```

**The $\Phi$ function**

- $\Phi$ merges multiple definitions along multiple control paths into a single definition.
- At a BB with $p$ predecessors, there are $p$ arguments to the $\Phi$ function.
  
  $x_{\text{NEW}} \leftarrow \Phi(x_1, x_2, x_3, \ldots, x_p)$

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

**“Implementing” $\Phi$**

```
\begin{align*}
\text{c}_1 & \leftarrow 12 \\
\text{if (i)} \\
\text{\quad a}_1 & \leftarrow x + y \\
\text{\quad b}_2 & \leftarrow a_1 + x \\
\text{\quad a}_2 & \leftarrow b + 2 \\
\text{\quad a}_3 & \leftarrow y + 1 \\
\text{\quad c}_3 & \leftarrow c_1 + a_3
\end{align*}
```

**Trivial SSA**

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

```
\begin{align*}
x & \leftarrow 1 \\
y & \leftarrow x \\
z & \leftarrow y + x
\end{align*}
```

**Way too many $\Phi$ functions inserted.**

**Minimal SSA**

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

```
\begin{align*}
x & \leftarrow 1 \\
y & \leftarrow x \\
z & \leftarrow y + x
\end{align*}
```
Another Example

```
a ← 0
```

```
b ← a + 1
c ← c + b
a ← b * 2
a < N
return c
```

Another Example

```
a ← 0
```

```
b ← a + 1
c ← c + b
a ← b * 2
a < N
return c
```

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 from \( y \) to \( z \), \( P_{yz} \). s.t.
    - \( P_{xz} \cap P_{yz} = \{ z \} \)
    - \( z \not\in P_{xq} \) or \( z \not\in P_{xr} \) where \( P_{xz} = P_{xq} \) or \( P_{xz} = P_{xr} \)
  - 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 \( x_i \) is used in \( x \leftarrow \Phi(\ldots, x_i, \ldots) \), then \( BB(x_i) \) dominates \( i \)th pred of \( BB(\Phi) \)
  - If \( x \) is used in \( y \leftarrow \ldots x \ldots \), then \( BB(x) \) dominates \( BB(y) \)
- We can use this for an efficient alg to convert to SSA

Dominance

\( x \) strictly dominates \( w \) (\( sdom \ w \)) iff \( x \) dom \( w \) AND \( x \neq w \)

© Seth Copen Goldstein 2001-2
The dominance Frontier of a node $x = \{w \mid x \text{ dom pred}(w) \text{ AND } (x \text{ sdom } w)\}$

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