Static Single Assignment (SSA) Form

A sparse program representation for data-flow.

Computing Static Single Assignment (SSA) Form

Overview:

- What is SSA?
- Advantages of SSA over use-def chains
- “Flavors” of SSA
- Dominance frontiers
- Inserting ϕ-nodes
- Renaming the temporaries
- Translating out of SSA form

R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, *Efficiently Computing Static Single Assignment Form and the Control Dependence Graph*, *ACM TOPLAS* 13(4):451–490, Oct 1991

What is SSA?

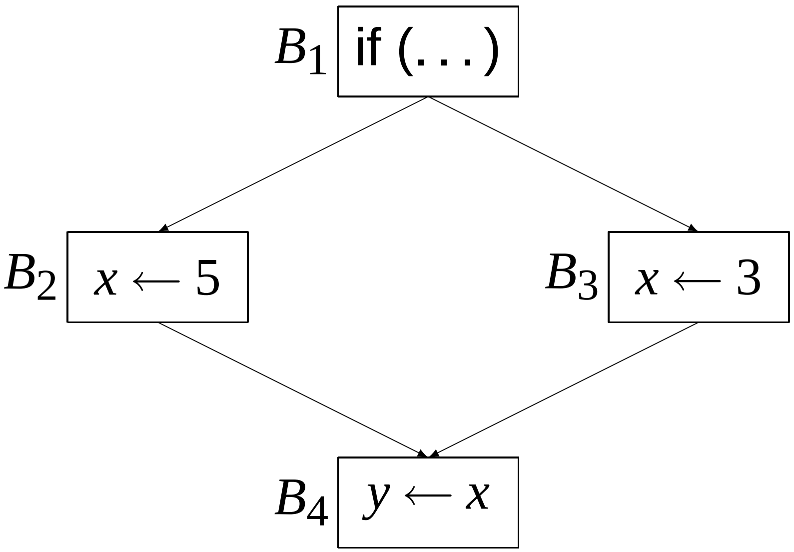
- Each assignment to a temporary is given a unique name
- All of the uses reached by that assignment are renamed
- Easy for straight-line code

$$
\begin{array}{rcl|rcl}
v & \leftarrow & 4 & v_0 & \leftarrow & 4 \\
 & \leftarrow & v+5 & & \leftarrow & v_0+5 \\
v & \leftarrow & 6 & v_1 & \leftarrow & 6 \\
 & \leftarrow & v+7 & & \leftarrow & v_1+7
\end{array}
$$

- What about control flow?

$\Rightarrow$ ϕ-nodes

What is SSA?

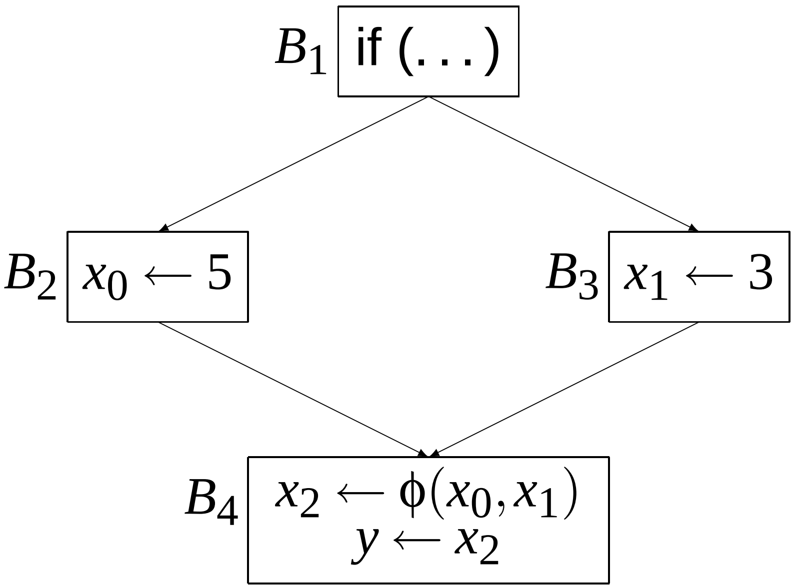


B_1 if $(\ldots)$

B_2 $x \leftarrow 5$

B_3 $x \leftarrow 3$

B_4 $y \leftarrow x$



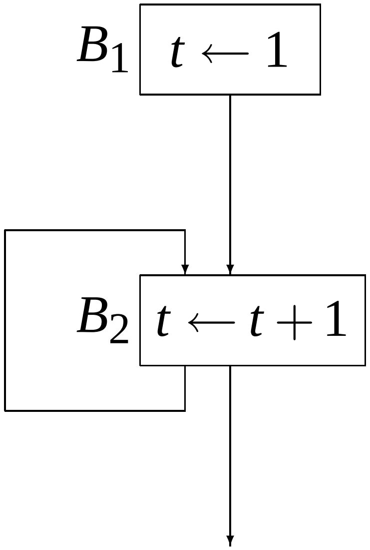
B_1 if $(\ldots)$

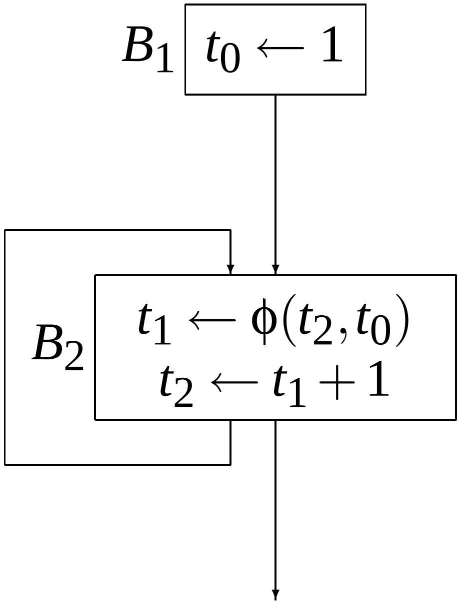
B_2 $x_0 \leftarrow 5$

B_3 $x_1 \leftarrow 3$

B_4 $x_2 \leftarrow \phi(x_0, x_1)$
$y \leftarrow x_2$

What is SSA?


B_1
$t \leftarrow 1$
B_2
$t \leftarrow t + 1$


B_1
$t_0 \leftarrow 1$
B_2
$t_1 \leftarrow \phi(t_2, t_0)$
$t_2 \leftarrow t_1 + 1$

Advantages of SSA over use-def chains

- More compact representation
- Easier to update?
- Each use has only one definition
- Definitions explicitly merge values
  May still reach multiple ϕ-nodes

“Flavors” of SSA

Where do we place ϕ-nodes?

Condition:

If two non-null paths $x \rightarrow^{+} z$ and $y \rightarrow^{+} z$ converge at node z, and nodes x and y contain assignments to t (in the original program), then a ϕ-node for t must be inserted at z (in the new program)

minimal

As few as possible subject to condition

semi-pruned by Preston Briggs

As few as possible subject to condition, and t must be live across some basic block

pruned

As few as possible subject to condition, and no dead ϕ-nodes

Dominance Frontiers

From v's point of view, these are the nodes at which other control paths that don't go through v make their earliest appearance.

The *dominance frontier* of v is the set of nodes $\textsf{DF}(v)$ such that:

- v dominates a predecessor of $w \in \textsf{DF}(v)$, but
  x does not strictly dominate $w \in \textsf{DF}(v)$

$$\textsf{DF}(v) = \{w \mid \big(\exists u \in PRED(w)\big)[v \text{ DOM } u] \wedge v\ \overline{\text{DOM!}}\ w\}$$

- d dominates v, d DOM v, in a CFG *iff* *all* paths from $Entry$ to v include d

- d *strictly* dominates v:

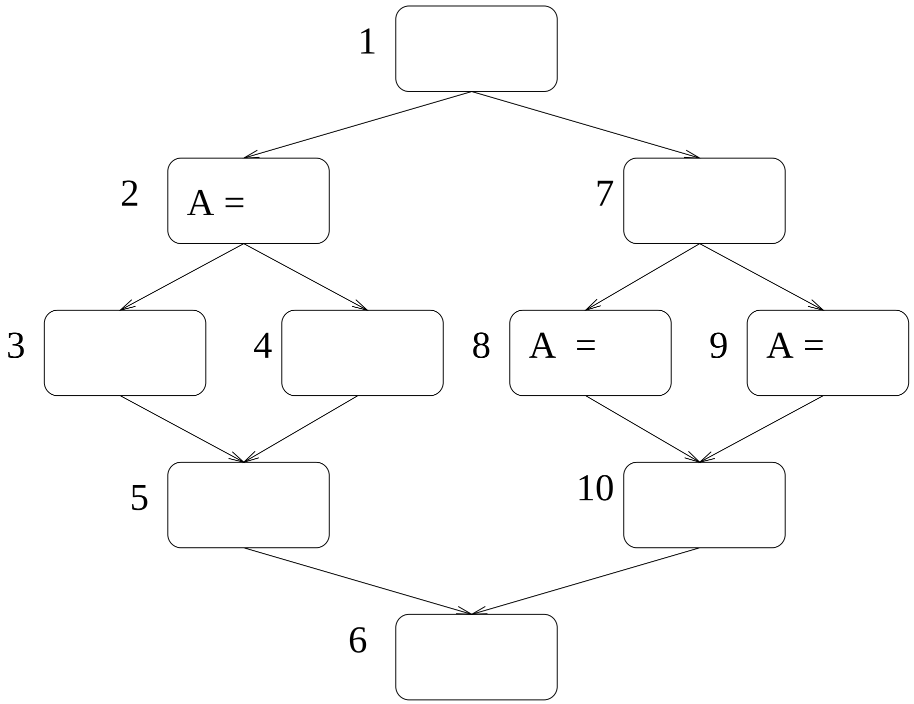
$$d \text{ DOM! } v \iff d \text{ DOM } v \text{ and } d \neq v$$

- The *immediate* dominator of v, $\textsf{IDOM}(v)$, is the closest strict dominator of v:

$$d \text{ IDOM } v \iff d \text{ DOM! } v \wedge (\forall w \mid w \text{ DOM! } v)[w \text{ DOM } d]$$

  $\textsf{IDOM}(v)$ is v's parent in the *dominator tree*

Dominance Frontier: Example



DF(8) =

DF(9)=

DF(2)=

DF({8,9}) =

DF(10) =

DF({2,8,9,10}) =

Iterated Dominance Frontier

Extend the dominance frontier mapping from nodes to sets of nodes:

$$\mathrm{DF}(S) = \bigcup_{n \in S} \mathrm{DF}(n)$$

The *iterated* dominance frontier $\mathrm{DF}^{+}(S)$ is the limit of the sequence:

$$\begin{aligned}\mathrm{DF}_1(S) &= \mathrm{DF}(S)\\ \mathrm{DF}_{i+1}(S) &= \mathrm{DF}(S \cup \mathrm{DF}_i(S))\end{aligned}$$

Theorem:

The set of nodes that need ϕ-nodes for any temporary t is the iterated dominance frontier $\mathrm{DF}^{+}(S)$, where S is the set of nodes that define t

Iterated Dominance Frontier Algorithm: $\text{DF}^{+}(S)$

Input: Set of blocks S

Output: $\text{DF}^{+}(S)$

```
workList ← $\{\}$
$\text{DF}^{+}(S) \leftarrow \{\}$
foreach $n \in S$ do
    $\text{DF}^{+}(S) \leftarrow \text{DF}^{+}(S) \cup \{n\}$
    workList ← workList $\cup \{n\}$
end
while workList $\neq \{\}$ do
        take $n$ from workList
        foreach $c \in \text{DF}(n)$ do
            if $c \notin \text{DF}^{+}(S)$ then
                $\text{DF}^{+}(S) \leftarrow \text{DF}^{+}(S) \cup \{c\}$
                workList ← workList $\cup \{c\}$
            end
        end
end
```

Inserting ϕ-nodes (minimal SSA)

```
foreach $t \in Temporaries$ do
    $S \leftarrow \{n \mid t \in Def(n)\} \cup Entry$
    Compute $\mathrm{DF}^{+}(S)$
    foreach $n \in \mathrm{DF}^{+}(S)$ do
        Insert a $\phi$-node for $t$ at $n$
    end
end
```

Inserting ϕ-nodes for globals (semi-pruned SSA)

Compute *local* liveness: globals are those live across block boundaries (*ie*, used before definition in *any* basic block)

```
foreach $t \in Temporaries$ do
    if $t \in Globals$ then
        $S \leftarrow \{n \mid t \in Def(n)\} \cup Entry$
        Compute $\mathrm{DF}^{+}(S)$
        foreach $n \in \mathrm{DF}^{+}(S)$ do
            Insert a $\phi$-node for $t$ at $n$
        end
    end
end
```

Inserting fewest ϕ-nodes (pruned SSA)

Compute *global* liveness: nodes where each temporary is live-in

```
foreach $t \in Temporaries$ do
  if $t \in Globals$ then
    $S \leftarrow \{n \mid t \in Defs(n)\} \cup Entry$
    Compute $\mathrm{DF}^{+}(S)$
    foreach $n \in \mathrm{DF}^{+}(S)$ do
      if $t$ live-in at $n$ then
        Insert a $\phi$-node for $t$ at $n$
      end
    end
  end
end
```

Renaming the temporaries

After ϕ-node insertion, uses of t are either:

original: dominated by the definition that computes t.

If not, then $\exists$ path to use avoiding definition, which means separate paths from definitions converge between definition and use, thus inserting another definition.
ie, each use dominated by an evaluation of t or a ϕ-node for t

ϕ: has a corresponding predecessor p, dominated by the definition of t (as before)

Thus, walk dominator tree, replacing each definition and its dominated uses with a new temporary.

Use a stack to hold current name (subscript) for each set of dominated nodes.

Propagate names from each block to corresponding ϕ-node operands of its successors.

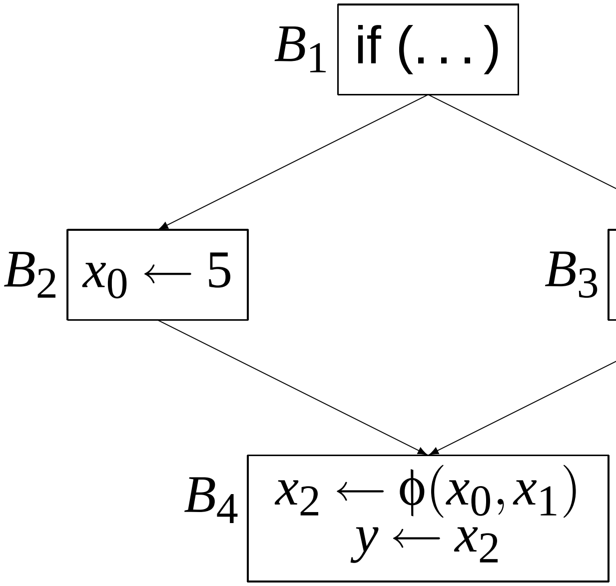
Renaming the temporaries

```
foreach $t \in$ Temporaries do count$[t] \leftarrow 0$; stack$[t] \leftarrow$ empty; stack$[t]$.push$(0)$
Rename$(Entry)$

proc Rename$(n)$ $\equiv$
  foreach statement $I \in n$ do
    if $s \not\equiv \phi$ then foreach $t \in Uses(I)$ do
                     $i \leftarrow$ stack$[t]$.top
                     replace use of $t$ with $t_i$ in $I$
    foreach $t \in Defs(I)$ do
      $i \leftarrow$ ++count$[t]$; stack$[t]$.push$(i)$
      replace def of $t$ with $t_i$ in $I$
  foreach $s \in SUCC(n)$ do
    given $n$ is the $j$th predecessor of $s$
    foreach $\phi \in s$ do
      given t is the $j$th operand of $\phi$
      $i \leftarrow$ stack$[t]$.top
      replace $j$th operand of $\phi$ with $t_i$
  foreach $c \in Children(n)$ do Rename$(c)$
  foreach statement $I \in n, t \in Defs(I)$ do stack$[t].pop()$
```

Translating Out of SSA Form

Replace ϕ-nodes with copy statements in predecessors

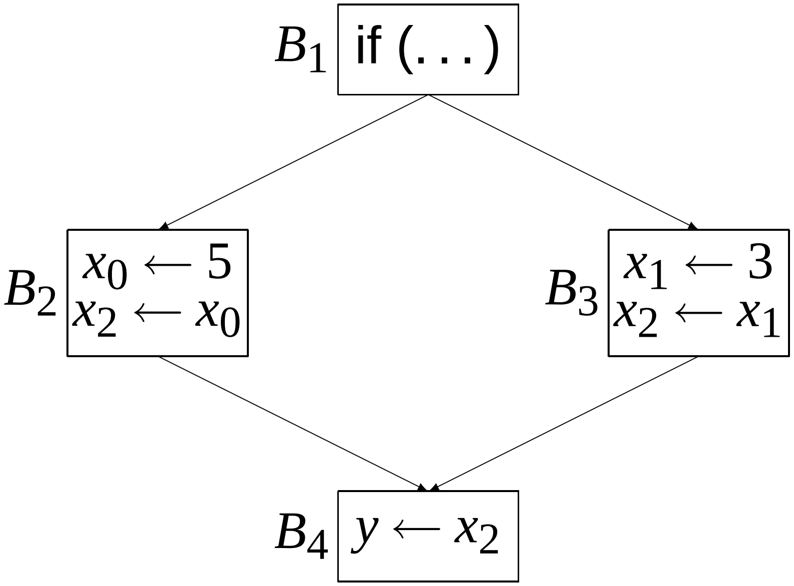


B_1 if (...)

B_2 $x_0 \leftarrow 5$

B_3 $x_1 \leftarrow 3$

B_4 $x_2 \leftarrow \phi(x_0, x_1)$ $y \leftarrow x_2$



B_1 if (...)

B_2 $x_0 \leftarrow 5$ $x_2 \leftarrow x_0$

B_3 $x_1 \leftarrow 3$ $x_2 \leftarrow x_1$

B_4 $y \leftarrow x_2$

Next Time

Static Single Assignment

- Induction variables (standard vs. SSA)
- Loop Invariant Code Motion with SSA