Computing Static Single Assignment (SSA) Form

Overview

- What is SSA?
- Advantages of SSA over use-def chains
- "Flavors" of SSA
- Dominance frontiers revisited
- Inserting ϕ-nodes
- Renaming the variables
- 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), October, 1991, pp. 451–490.

What is SSA?

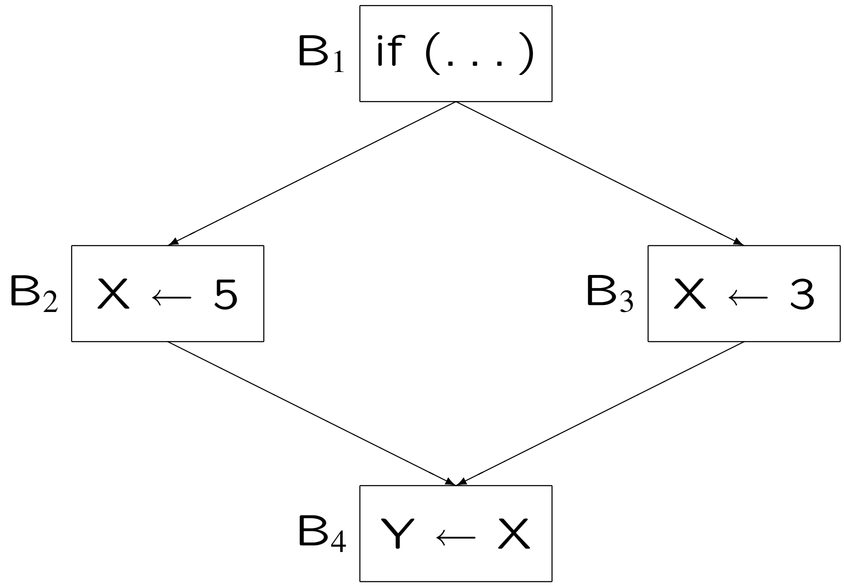
- Each assignment to a variable is given a unique name
- All of the uses reached by that assignment are renamed
- Easy for straight-line code

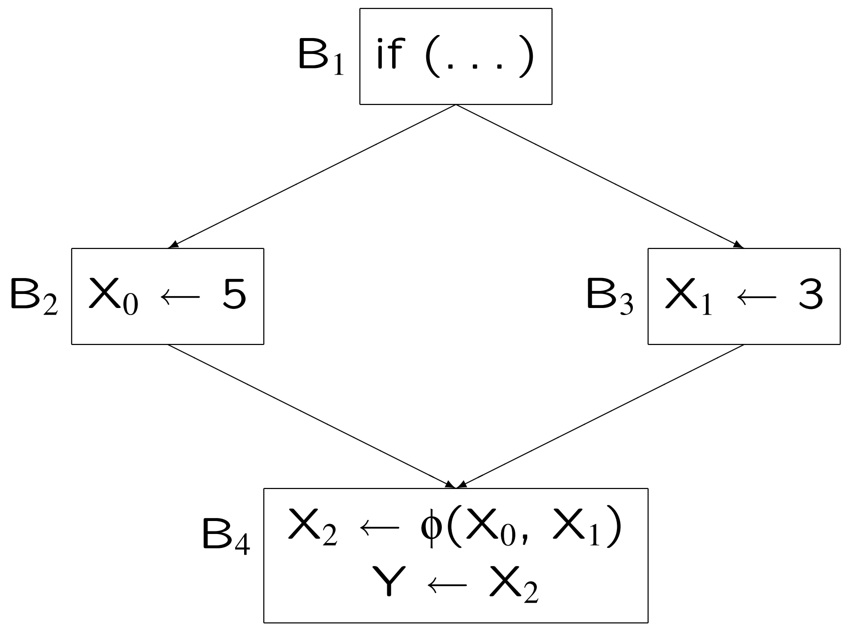
$V \leftarrow 4$	$V_0 \leftarrow 4$
$\leftarrow V + 5$	$\leftarrow V_0 + 5$
$V \leftarrow 6$	$V_1 \leftarrow 6$
$\leftarrow V + 7$	$\leftarrow V_1 + 7$

What about control flow?

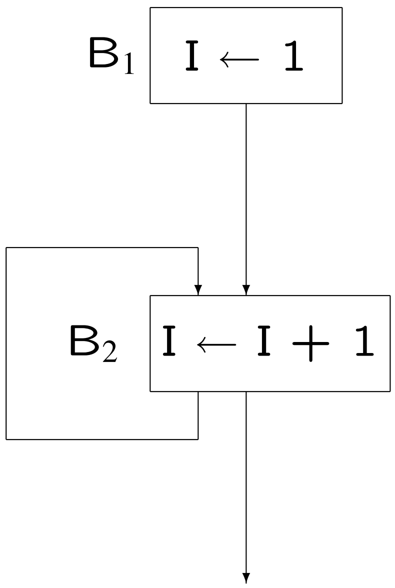
$\Longrightarrow$ ϕ-nodes

What is SSA?

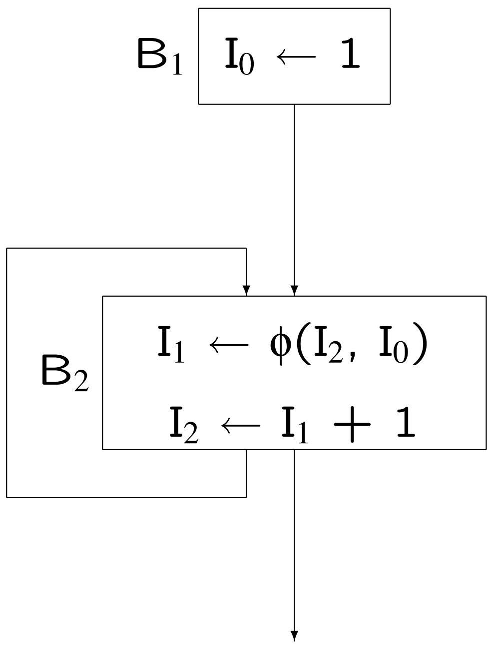




What is SSA?



B_1
$I \leftarrow 1$
B_2
$I \leftarrow I + 1$



B_1
$I_0 \leftarrow 1$
B_2
$I_1 \leftarrow \phi(I_2, I_0)$
$I_2 \leftarrow I_1 + 1$

Advantages of SSA over use-def chains

- More compact representation
- Easier to update?
- Each USE has only one definition
- Definitions are explicit merging of values
  definitions may still reach multiple ϕ-node

“Flavors” of SSA

Where do we place ϕ-nodes?

Condition:

If two non-null paths $X \xrightarrow{+} Z$ and $Y \xrightarrow{+} Z$ converge at node Z, and nodes X and Y contain assignments to V (in the original program), then a ϕ-node for V must be inserted at Z (in the new program).

minimal

As few as possible subject to condition

Briggs-minimal Invented by Preston Briggs

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

pruned

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

Dominance Frontiers Revisited

The *dominance frontier* of X is the set of nodes Y *s.t.*
X dominates a predecessor of Y, but
X does not strictly dominate Y.

DF(X) = $\{$Y $|$ $\exists$ P $\in$ pred(Y),
(X DOM P and X $\neg$DOM! Y)$\}$

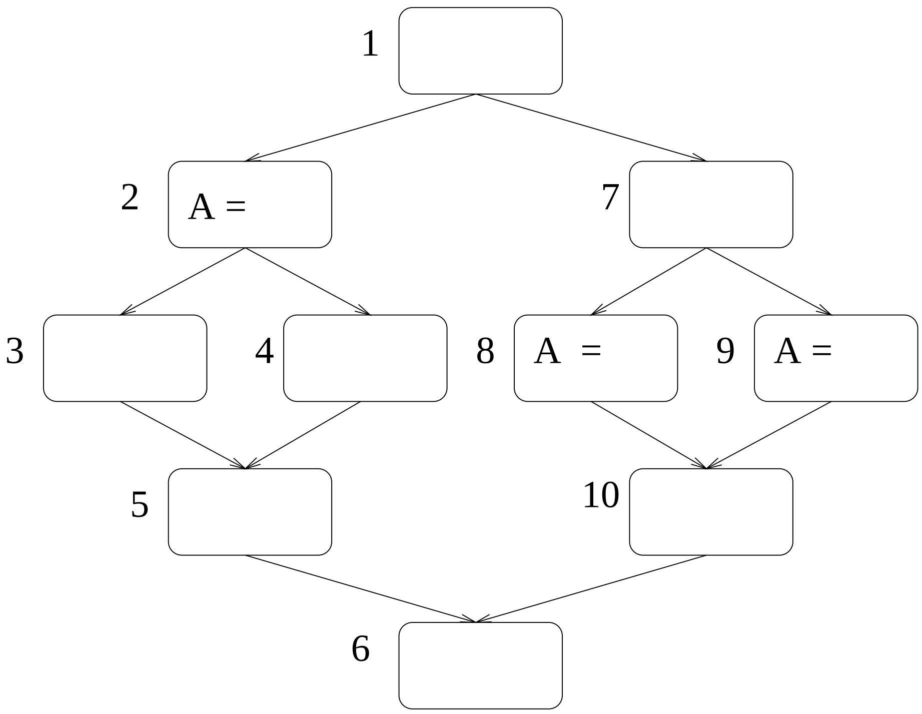
If X appears on every path from *entry* to Y,
then X *dominates* Y (X DOM Y).

If X DOM Y and X $\neq$ Y,
then X *strictly dominates* Y (X DOM! Y).

The *immediate dominator* of Y (IDOM(Y))
is the closest strict dominator of Y.

IDOM(Y) is Y'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:

$$\text{DF}(\mathcal{L}) = \bigcup_{X \in \mathcal{L}} DF(X)$$

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

$$\text{DF}_1 = \text{DF}(\mathcal{L})$$
$$\text{DF}_{i+1} = \text{DF}(\mathcal{L} \bigcup \text{DF}_i)$$

Theorem 1

The set of nodes that need ϕ-nodes for any variable V is the iterated dominance frontier $\text{DF}^{+}(\mathcal{L})$, where $\mathcal{L}$ is the set of nodes with assignments to V.

Inserting ϕ-nodes

```
for each variable V
    HasAlready ← ∅
    EverOnWorkList ← ∅
    WorkList ← ∅
    for each node X containing an assignment to V
        EverOnWorkList ← EverOnWorkList ⋃{X}
        WorkList ← WorkList ⋃{X}
    end for

    while WorkList ≠ ∅
        remove X from WorkList
        for each Y ∈ DF(X)
            if Y ∉ HasAlready
                insert a ϕ-node for V at Y
                HasAlready ←HasAlready ⋃{Y}
                if Y ∉ EverOnWorkList
                    EverOnWorkList ← EverOnWorkList ⋃{Y}
                    WorkList ← WorkList ⋃{Y}
        end for
    end while
endfor
```

Renaming the variables

Data Structures

Stacks array of stacks, one for each original variable V
The subscript of the most recent definition of V
Initially, Stacks[V] = EmptyStack, $\forall$ V

Counters an array of counters, one for each original variable
The number of assignments to V processed
Initially, Counters[V] = 0, $\forall$ V

procedure **GenName**(Variable V)
  i $\leftarrow$ Counters[V]
  replace V by V_i
  Push i onto Stacks[V]
  Counters[V] $\leftarrow$ i + 1

Rename - a recursive procedure

- Walks the dominator tree in preorder
- Initially, call Rename(*entry*)

Renaming the variables

procedure Rename(Block X)

//first process ϕ-nodes
    for each ϕ-node P in X
        GenName(LHS(P))

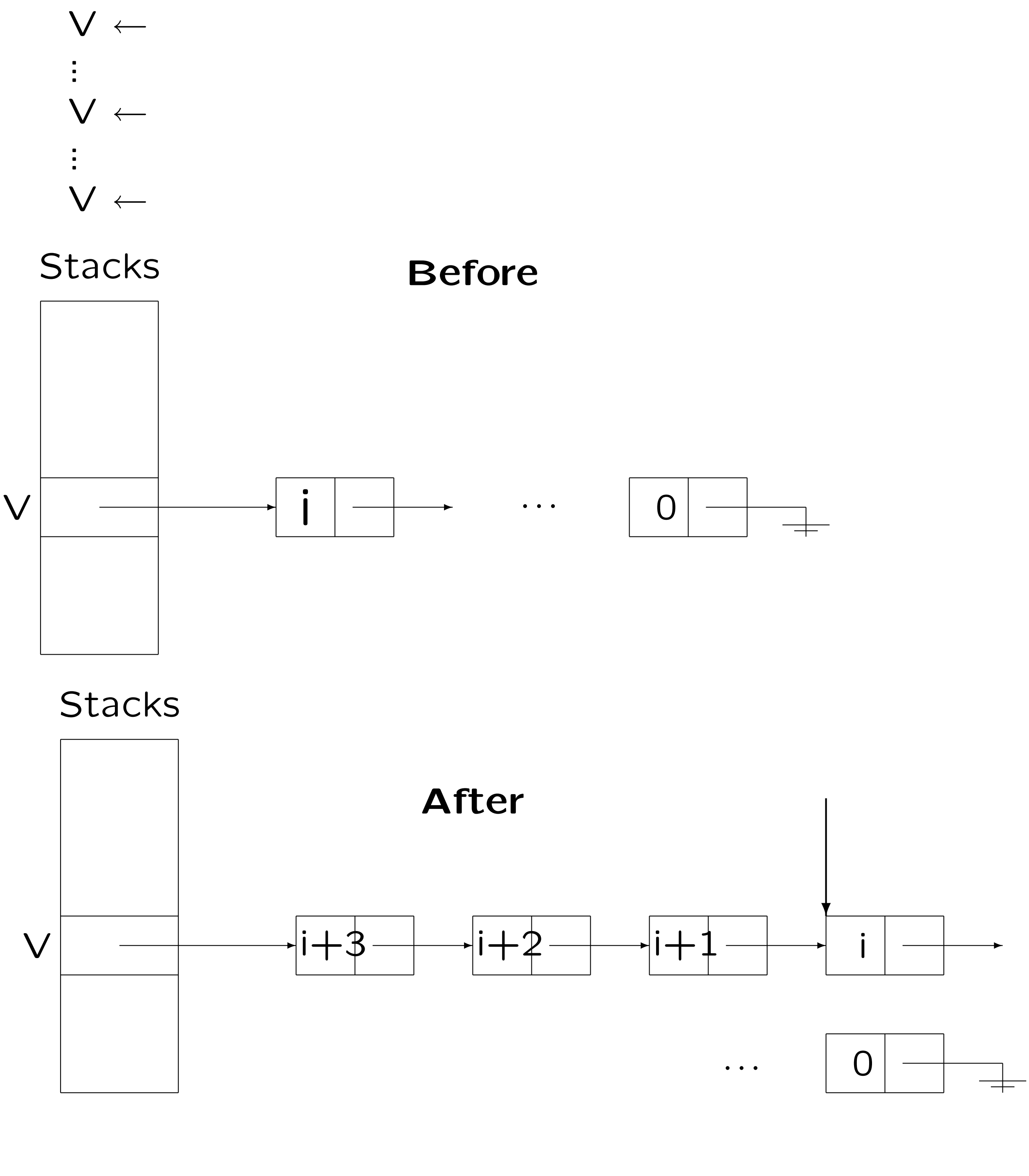
//then process statements in block X
    for each statement A in X
        for each variable V $\in$ RHS(A)
            replace V by V_i, where i = Top(Stacks[V])
        for each variable V $\in$ LHS(A)
            GenName(V)

//then update any ϕ-functions in CFG successors of X
    for each Y $\in$ SUCC(X)
        $j \leftarrow$ position in Y's ϕ-nodes corresponding to X
        for each ϕ-node P in Y
            replace the jth operand of RHS(P) by V_i
                where i = Top(Stacks[V])

//recursively visit children of X in dominator tree
    for each Y $\in$ SUCC(X)
        Rename(Y)

//when backing out of X, pop variables defined in X
    for each ϕ-node or statement A in X
        for each V_i $\in$ LHS(A)
            Pop (Stacks[V])

What happens to Stacks during Renaming?



Computing SSA Form

Compute dominance frontiers

Insert ϕ-nodes

Rename variables

Theorem 2

Any program can be put into minimal SSA form using this algorithm.

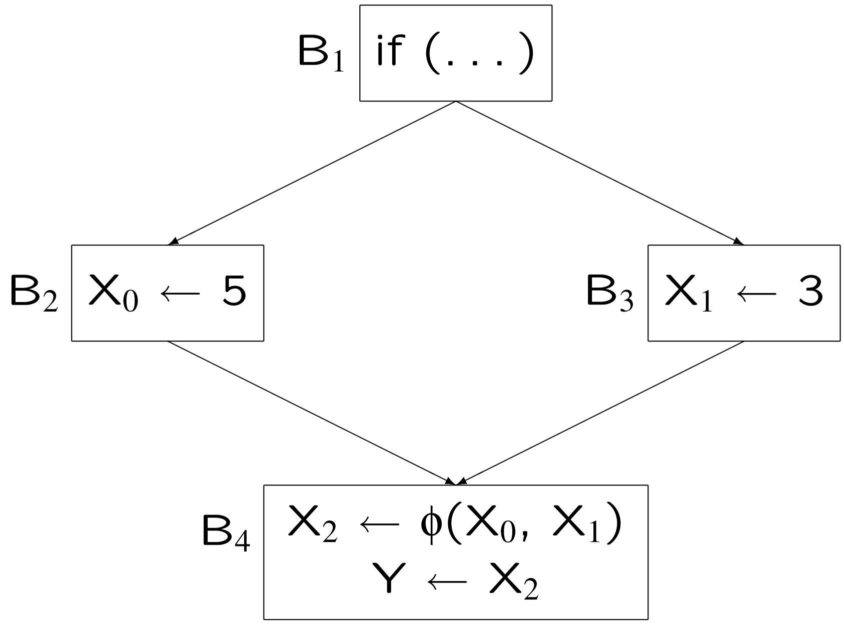
Translating Out of SSA Form

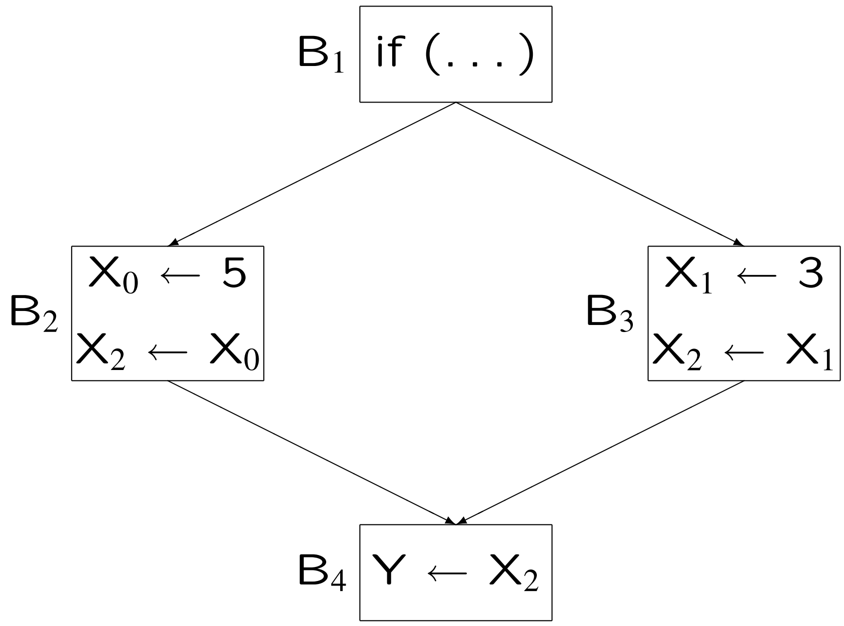
Restore original names to variables

Delete all ϕ-nodes

Replace ϕ-nodes with copies in predecessors

Translating Out of SSA Form





Next Time

Static Single Assignment

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