Another Form of Data-Flow Analysis

Propagation of values

- for a variable reference, where is the value produced?
- for a variable definition, where is the value consumed?

Possible answers

- reaching definitions, live variables
- def-use, use-def chains
- static single assignment (SSA)

Def-Use and Use-Def Chains

- directly connect producers and consumers
- extend reaching definitions to add edges
- extend live variables to track uses, add edges
- advantage: bypass intervening flow graph
- disadvantage: requires more space

Static Single Assignment Form

What is SSA?

- each assignment to a variable is given a unique name
- all of the uses reached by that assignment are renamed

Example

$$\begin{array}{ll} \mathrm{V} \leftarrow 4 & \mathrm{V}_0 \leftarrow 4 \\ \quad \leftarrow \mathrm{V} + 5 & \quad \leftarrow \mathrm{V}_0 + 5 \\ \mathrm{V} \leftarrow 6 & \mathrm{V}_1 \leftarrow 6 \\ \quad \leftarrow \mathrm{V} + 7 & \quad \leftarrow \mathrm{V}_1 + 7 \end{array}$$

Why is this useful?

- representation explicitly connects definitions to their uses (and vice versa)
- more compact representation than def-use and use-def chains
- definitions kept in a separate data structure from CFG
- merging of values is explicit

Handling Multiple Reaching Definitions

ϕ-functions (aka, ϕ-nodes)

- For some CFG node n, a function of the form
  $V_n \leftarrow \phi(V_{p1}, V_{p2}, , ...)$,
  where each subscripted V corresponds to the definition reaching this point from n's predecessors $p1$ and $p2$

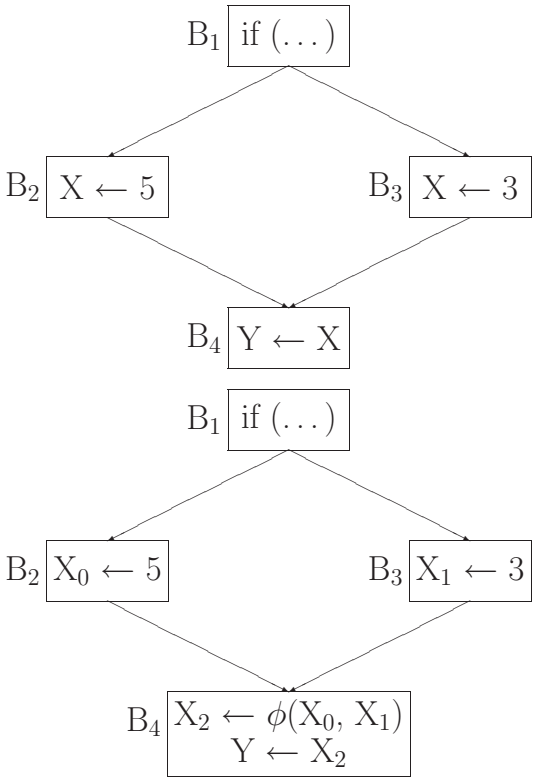
Where do we place ϕ-functions?

Intuitively. At the first point where paths in the CFG merge that have distinct definitions for the same variable

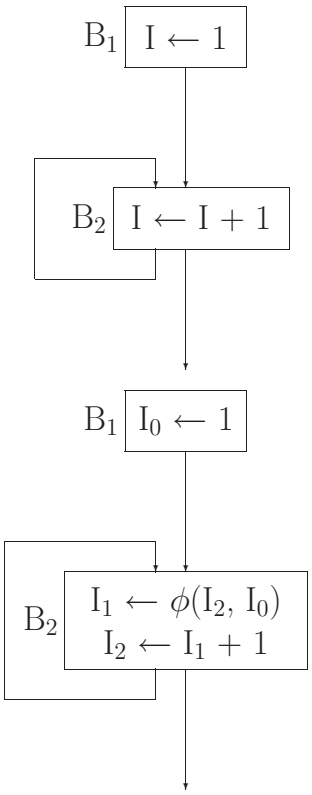
Formally. If there exist two non-null paths
X $\rightarrow^+$ Z and Y $\rightarrow^+$ Z that converge for the first time at node Z, and nodes X and Y contain assignments to V (in the original program), then a ϕ-function for V must be inserted at Z (in the new program)

Placement of ϕ-functions subject to this condition yields "minimal" SSA form

Examples of ϕ-Functions



Another Example: Loops



Computing SSA Form

1. Insert ϕ-functions
   (a) Build dominator tree
   (b) Compute dominance frontiers and their closure
   (c) Rename variables
2. Translate back from SSA form

> *The SSA graph is simply the graph built by adding def-use and/or use-def chains to the program in SSA form during SSA construction*

R. Cytron *et al.*, "Efficiently computing static single assignment form and the control dependence graph", ACM Transactions on Programming Languages and Systems (TOPLAS), 13(4), October 1991

Insert ϕ-functions: Dominators

If X appears on every path from **Entry** to Y,
then X *dominates* Y ($X \gg\!\!= Y$)

If $X \gg\!\!= Y$ and $X \neq Y$,
then X *strictly dominates* Y ($X \gg Y$)

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 Frontiers

The *dominance frontier* of node X is the set of nodes Y such that

X dominates a predecessor of Y, but
X does not strictly dominate Y

$$DF(X) = \{Y \mid \exists\, P \in Pred(Y), (X \gg\!\!= P \text{ and } X \not\gg Y)\}$$

The dominance frontier can be subdivided into two components:

$$DF_{local}(X) \equiv \{Y \in Succ(X) \mid X \not\gg Y\}$$
$$DF_{up}(X) \equiv \{Y \in DF(Z) \mid Z \in Children(X) \wedge X \not\gg Y\}$$

Then,

$$DF(X) = DF_{local}(X) \cup DF_{up}(X)$$

Succ = immediate successors in the CFG

Children = descendents in the dominator tree

† *Intuitively, dominance frontier is point just beyond region a node dominates*

Dominance Frontiers Algorithm

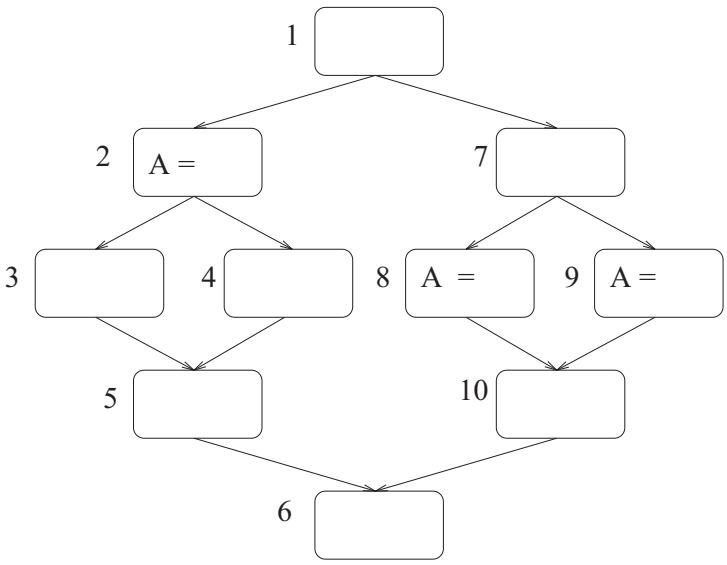
for each X in a bottom-up traversal of the dominator tree

```
DF(X) ← ∅
for each Y ∈ Succ(X)              /* local */
    if idom(Y) ≠ X then
        DF(X) ← DF(X) ∪ {Y}
for each Z ∈ Children(X)          /* up */
    for each Y ∈ DF(Z)
        if idom(Y) ≠ X then
            DF(X) ← DF(X) ∪ {Y}
```

Succ = immediate successors in the CFG

Children = descendents in the dominator tree

Dominance Frontiers Example



DF(8) =

DF(9)=

DF(7)=

DF({8,9}) =

Dominance Frontier Closure

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

$\mathrm{DF}(\mathcal{L}) = \cup_{X \in \mathcal{L}} \mathrm{DF}(X)$

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

$\mathrm{DF}_1 = \mathrm{DF}(\mathcal{L})$

$\mathrm{DF}_{i+1} = \mathrm{DF}(\mathcal{L} \cup \mathrm{DF}_i)$

(*i.e.,* a closure)

Theorem

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

Algorithm For Inserting ϕ-nodes

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

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

Computing SSA Form

Complete algorithm

- compute the dominance frontiers
- insert ϕ-nodes
- rename the variables

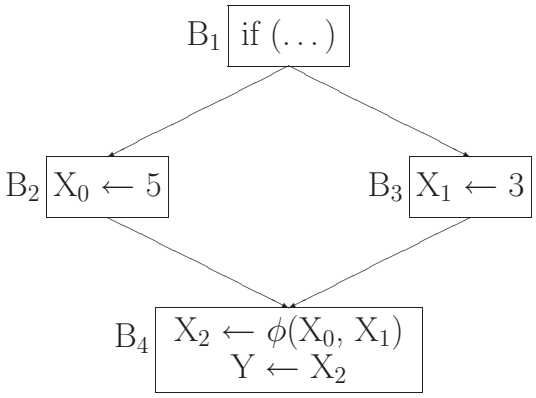
Theorem

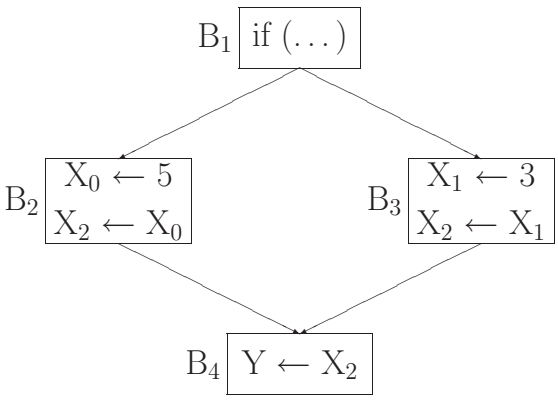
> Any program can be put into "minimal" SSA form using this algorithm.

Optimizations: [fewer ϕ-nodes than "minimal"]

- pruned – eliminate any dead ϕ-nodes
- semi-pruned – eliminate ϕ-nodes for variables dead on exit from all basic blocks

Example: Translating From SSA Form





Static Single Assignment

Size of SSA graph

- potentially large (pathological cases)
- in practice appears linear in size of program

Applications for SSA

- loop invariant code motion
  (hoist invariant live ranges)
- induction variable detection
  (find cycles in SSA graph)
- constant propagation
  (sparse conditional constant)
- many more...

R. Cytron *et al.*, "Efficiently computing static single assignment form and the control dependence graph," ACM Transactions on Programming Languages and Systems (TOPLAS), 13(4), Oct 1991

Constant Propagation Revisited

Sparse simple constant

- optimistic assumptions
- propagation along sparse graph (SSA)

Algorithm

1. Initialization
   (a) mark values for LHS of unknown expressions as $\perp$ (e.g., READ stmts)
   (b) mark values for LHS of constant assignment as having its constant value
   (c) everything else is marked with $\top$
2. Add to Worklist all SSA edges where definition is not $\top$
3. Iterate until worklist is empty
   (a) meet values at definition and use points
   (b) if resulting value at use is different, replace value at use
   (c) recompute the value of expression at use
   (d) if new value, update and add outgoing SSA edges from this def to Worklist

Sparse Conditional Constant

Enhanced constant propagation

- ignore edges in CFG that are not executable

Conditional definition

- when expression in conditional branch is constant, determine direction of branch
- only propagate definitions for executable code
- at join points, ignore edges not marked as executable

```
i ← 1
if (i = 1)
     then j ← 1
     else j ← 2
k ← j
```

Wegman and Zadeck, "Constant propagation with conditional branches," ACM Transactions on Programming Languages and Systems (TOPLAS), 13(2), Apr 1991

Sparse Conditional Constant

Complexity

1. edges in SSA graph examined at most twice
2. edges in flow graph examined once

$\Rightarrow$ linear in edges of SSA graph + flow graph

Advantages of SSA

- ideal for analyses where uses and definitions are examined
- particularly profitable if program contains definitions and uses for lots of variables, and different variables are referenced in different parts of program
- may also be more suitable for algorithms obtaining more precise analysis results

Savings

- time during propagation
- space requirements for analysis
  (better match to how information is used)