

CS6013 - Modern Compilers: Theory and Practise

SSA and optimizations

V. Krishna Nandivada

IIT Madras

Copyright © 2016 by Antony L. Hosking. [Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and full citation on the first page. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or fee. Request permission to publish from \[hosking@cs.purdue.edu\]\(mailto:hosking@cs.purdue.edu\).](#)



Static Single Assignment (SSA) Form

A sparse program representation for data-flow.

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?

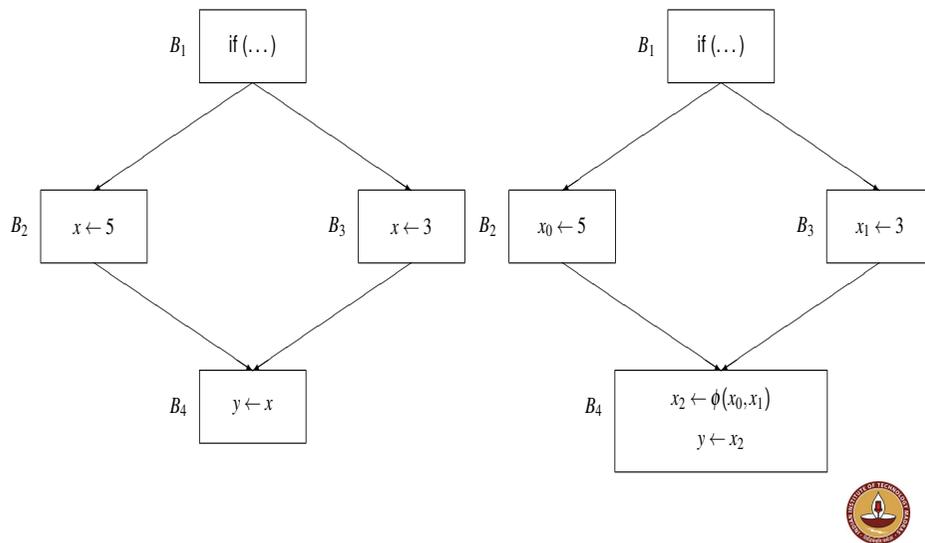
- Each assignment to a temporary is given a unique name
- All of the uses reached by that assignment are renamed
- Easy for straight-line code

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

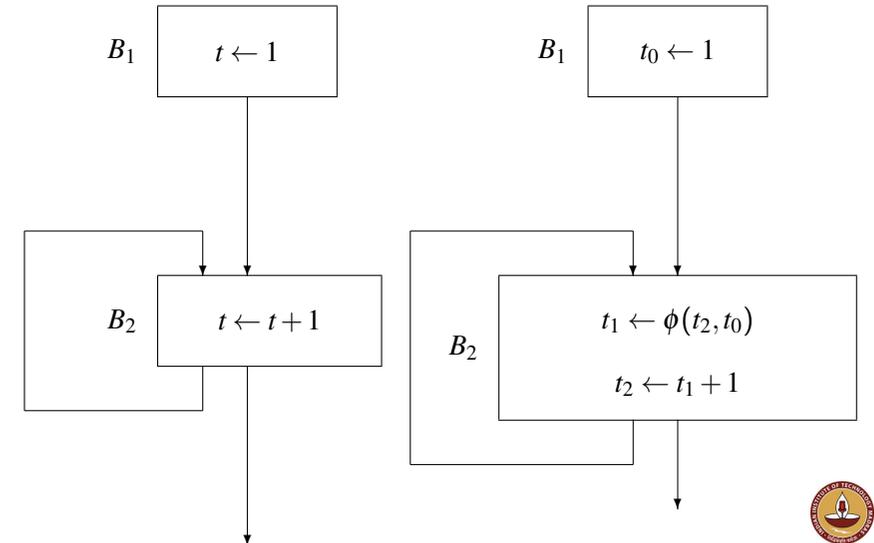
- What about control flow?
⇒ ϕ -nodes



What is SSA?



What is SSA?



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



Dominators revisited

Recall

- d dominates v , $d \text{ DOM } v$, in a CFG iff all paths from *Entry* to v include d
- d strictly dominates v

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

DOM(v) = Dominator of v

DOM⁻¹(v) = Dominated by v

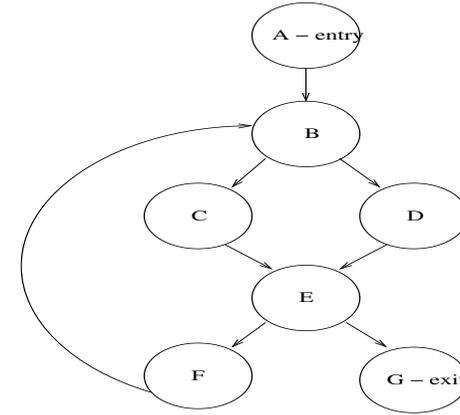


Dominator Tree

Dominator tree: a tree where each node's children are those nodes it immediately dominates.

The start node is the root of the tree.

Why is it a tree?



Dominance Frontiers

The dominance frontier of v is the set of nodes DF(v) such that:

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

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

- Computing DF:

Let

$$\text{SUCC}(S) = \bigcup_{s \in S} \text{SUCC}(s)$$

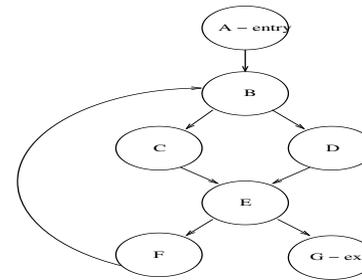
$$\text{DOM!}^{-1}(v) = \text{DOM}^{-1}(v) - \{v\}$$

Then

$$\text{DF}(v) = \text{SUCC}(\text{DOM}^{-1}(v)) - \text{DOM!}^{-1}(v)$$



Dominance Frontier: Example



$$\text{DF}(v) = \text{SUCC}(\text{DOM}^{-1}(v)) - \text{DOM!}^{-1}(v)$$

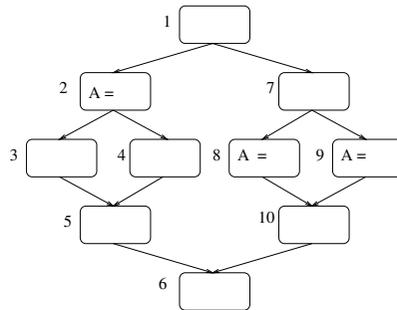
where $\text{DOM!}^{-1}(v) = \text{DOM}^{-1}(v) - \{v\}$

v	DOM ⁻¹ (v)	SUCC(DOM ⁻¹ (v))
A	{A, B, C, D, E, F, G}	
B	{B, C, D, E, F, G}	
C	{C}	
D	{D}	
E	{E, F, G}	
F	{F}	
G	{G}	

v	DOM ⁻¹ (v) - { v }	DF(v)
A		
B		
C		
D		
E		
F		
G		



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:

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

The iterated dominance frontier $DF^+(S)$ is the limit of the sequence:

$$DF_1(S) = DF(S)$$

$$DF_{i+1}(S) = DF(S \cup DF_i(S))$$

Theorem:

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



Iterated Dominance Frontier Algorithm: $DF^+(S)$

Input: Set of blocks S

Output: $DF^+(S)$

begin

$workList \leftarrow \{\}$;

$DF^+(S) \leftarrow \{\}$;

foreach $n \in S$ **do**

$DF^+(S) \leftarrow DF^+(S) \cup \{n\}$;

$workList \leftarrow workList \cup \{n\}$;

end

while $workList \neq \{\}$ **do**

take n from $workList$;

foreach $c \in DF(n)$ **do**

if $c \notin DF^+(S)$ **then**

$DF^+(S) \leftarrow DF^+(S) \cup \{c\}$;

$workList \leftarrow workList \cup \{c\}$;

end

end

end

end



Inserting ϕ -nodes (minimal SSA)

foreach $t \in \text{Temporaries}$ **do**

$S \leftarrow \{n \mid t \in \text{Def}(n)\} \cup \text{Entry}$;

Compute $DF^+(S)$;

foreach $n \in DF^+(S)$ **do**

Insert a ϕ -node for t at n ;

end

end



Inserting fewest ϕ -nodes (pruned SSA)

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

```

foreach  $t \in \text{Temporaries}$  do
  if  $t \in \text{Globals}$  then // variables live across multiple basic blocks
     $S \leftarrow \{n \mid t \in \text{Defs}(n)\} \cup \text{Entry}$ ;
    Compute  $\text{DF}+(S)$ ;
    foreach  $n \in \text{DF}+(S)$  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 the use that avoids any 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

```

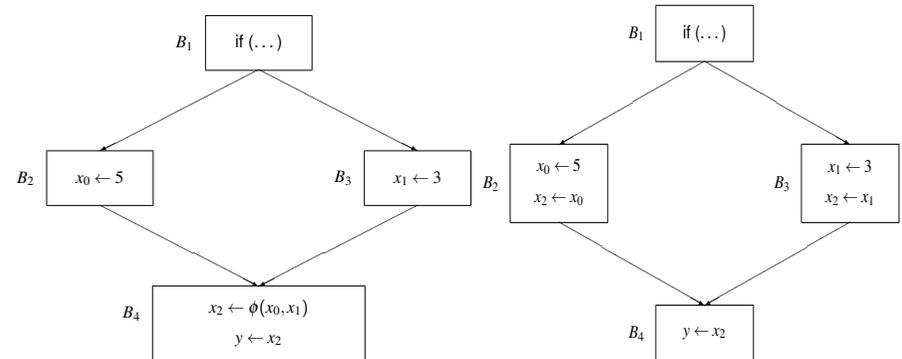
begin
  foreach  $t \in \text{Temporaries}$  do  $\text{count}[t] \leftarrow 0$ ;  $\text{stack}[t] \leftarrow \text{empty}$ ;  $\text{stack}[t].\text{push}(0)$ ;
  Call  $\text{Rename}(\text{Entry})$ ;
end
Rename}(n) begin
  foreach  $\text{statement } I \in n$  do
    if  $\text{stack} \neq \phi$  then
      foreach  $t \in \text{Uses}(I)$  do  $i \leftarrow \text{stack}[t].\text{top}$ ; replace use of  $t$  with  $t_i$  in  $I$ ;
    foreach  $t \in \text{Defs}(I)$  do
       $i \leftarrow ++\text{count}[t]$ ;  $\text{stack}[t].\text{push}(i)$ ;
      replace def of  $t$  with  $t_i$  in  $I$ ;
    foreach  $s \in \text{SUCC}(n)$  do
      given  $n$  is the  $j$ th predecessor of  $s$ ;
      foreach  $\phi \in s$  do
        given  $t$  is the  $j$ th operand of  $\phi$ ;
         $i \leftarrow \text{stack}[t].\text{top}$ ;
        replace  $j$ th operand of  $\phi$  with  $t_i$ ;
      foreach  $c \in \text{Children}(n)$  do  $\text{Rename}(c)$ ;
      foreach  $\text{statement } I \in n, t \in \text{Defs}(I)$  do  $\text{stack}[t].\text{pop}()$ ;
    end
end

```



Translating Out of SSA Form

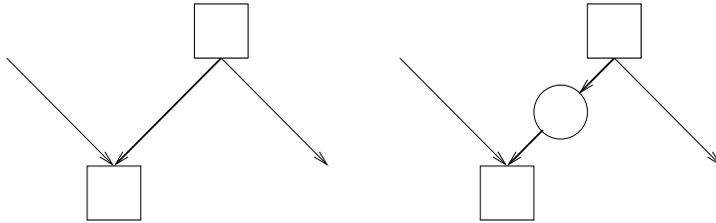
Replace ϕ -nodes with copy statements in predecessors



Issues in translation - critical edge split

Translating out ϕ nodes.

- The compiler inserts copy statements in the predecessors.
- Is it always safe?
- What if the predecessor has more than one successor?



- The lost copy problem:

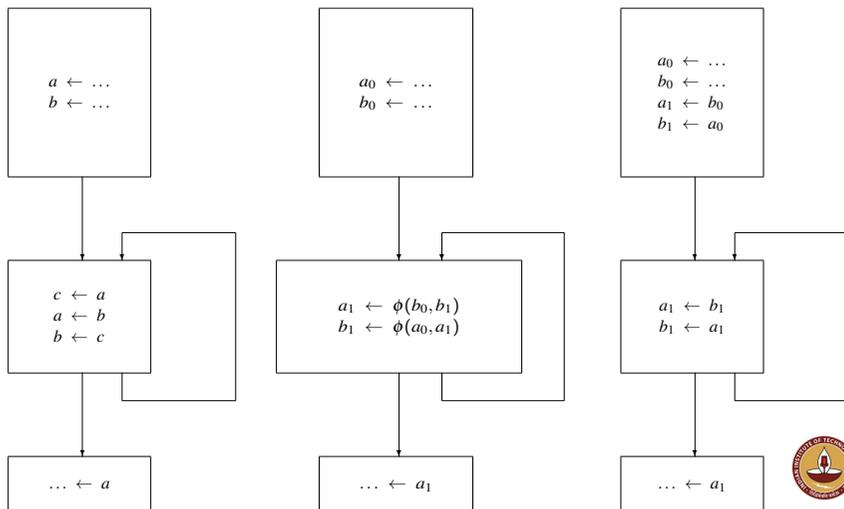
```

i = 1;
loop
  y = i
  i = i + 1
endloop
z = y
    
```



% with copy folding done, y gets replaced with the phi variable. And if we
 % are not careful and insert the copy statement in the predecessor, we
 % lose the copy of 'y' (the old value of 'i')

(Swap problem) Normal Form, Optimized SSA, Incorrect Translation

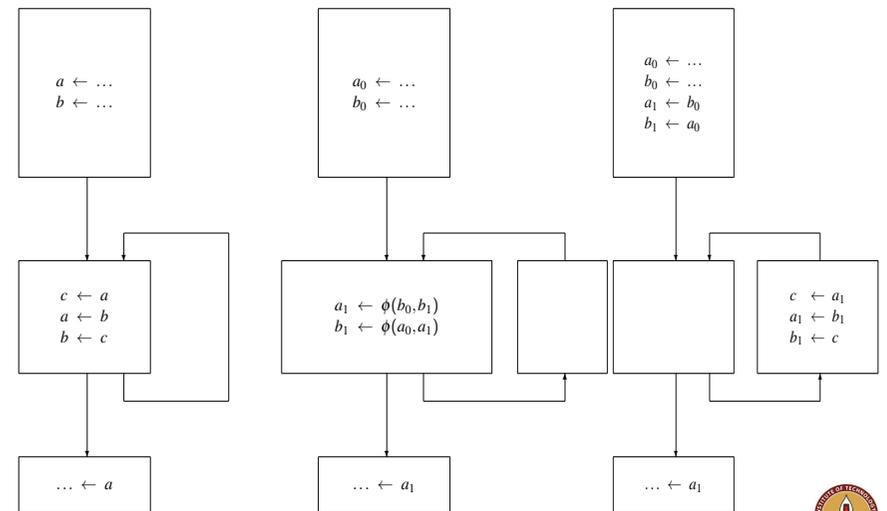


Translation - the swap problem

- The definition of ϕ function:
 - When a block executes all of its ϕ functions execute concurrently before any other statement in the block.
 - All the ϕ -functions simultaneously read their appropriate input parameters and simultaneously redefine their targets.



Normal Form, Edge-Split Opt SSA, Correct Translation



- Simply splitting a critical edge does not help.
- One simple way:
 - Step 1: Copy each of the ϕ function arguments to its own temporary name.
 - Step 2: Copy the temps to the appropriate ϕ -function targets.
- Disadvantage: Doubles the number of copy operations.
- Way out - Introduce copy only when required.
 - Detect cases in which ϕ -functions reference the targets of other ϕ functions in the same block.
 - For each cycle of references - introduce copy instructions.



Self reading: Wegman & Zadeck, Constant Propagation with Conditional Branches, TOPLAS 13(2):181–210, Apr 1991



Sparse Simple Constants

- Start with a worklist of all SSA edges.
- Process one edge at a time.
- If the lhs value of an assignment node changes, add all the SSA edges starting from that node to the worklist.

Details: Self reading.



Sparse Conditional Constants

- SSA edge: Data flow (def-use) edges in a program in SSA form.
- Basic idea: Instead of passing all the constants from all the control flow edges, pass constants from SSA edges.
- Resulting analysis - faster.



Sparse Conditional constants

- Works on two worklists:
 - FlowWorkList (contains program flow edges) and
 - SSAWorkList (contains SSA edges).
- Each flow edge has an executable flag – tells if the ϕ function at the destination is to be evaluated because of this flow edge – initialized to false.

Initialization and termination

- Initialize the FlowWorkList to contain the edges exiting the start node of the program.
- The SSAWorkList is initially empty.
- Halt execution when both worklists become empty.
- Execution may proceed by processing items from either worklist.



Processing flow edges

- if e is a flow edge from FlowWorkList then
 - if ExecutableFlag(e)=false then
 - ExecutableFlag(e) = true
 - Say $e = a \rightarrow b$
 - Perform Visit- ϕ for all ϕ -nodes at destination node.
 - on the destination node, if only one incoming flow-edges is executable then this is the first visit to the node
 - If first visit then Perform $v = \text{VisitExpr}(\text{Expr}(b))$ destination node
 - if the dest node contains one outgoing CFG-edge then add the edge to FlowWorkList
 - If the dest node contains two outgoing edges then add one / two of them depending on constant value of v .



Processing SSA edges

- If e is an SSA edge from SSAWorkList then
 - SSAWorkList -= e
 - Say $e = a \rightarrow b$
 - If b is a phi node, then Visit- ϕ (b)
 - Else If b is an expression and if $\exists c : \text{ExecutableFlag}(c \rightarrow \text{Node}(b)) = \text{true}$ then VisitExpr(b); // Note: $c \rightarrow \text{Node}(b)$ is a control edge.



Helper function: Visit- ϕ

- Updates the operands of the ϕ node.
- For each operand x_i of the ϕ node:
 - set the operand value to \top if the corresponding program flow edge has ExecutableFlag set to false.
 - Otherwise, replace x_i with the value of the operand at the definition point.



- Evaluate e
 - Normal expressions: using the values of the variables from the definition.
 - ϕ node: take a meet of values of all the operands.
- If the value of the e changes:
 - Say, n is the statement node containing e .
 - If n is an assignment add to SSAWorkList all the SSA edges starting from n .
 - If n is a conditional branch: add the newly enabled (because of change in the value of e) flow graph edges to the FlowWorkList.



- Static Single Assignment form.
- Sparse Conditional Constant propagation.

