CS6013 - Modern Compilers: Theory and Practise

SSA and optimizations

V. Krishna Nandivada

IIT Madras

SSA and optimizations

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

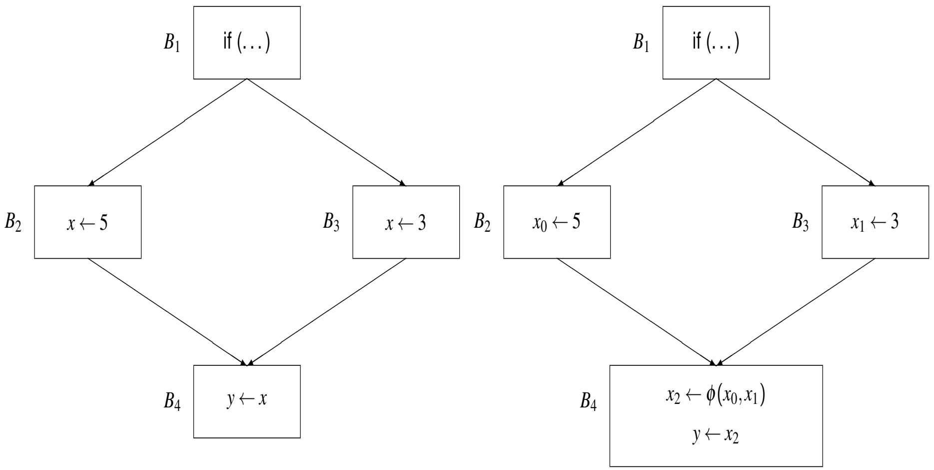
What is SSA?

- 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

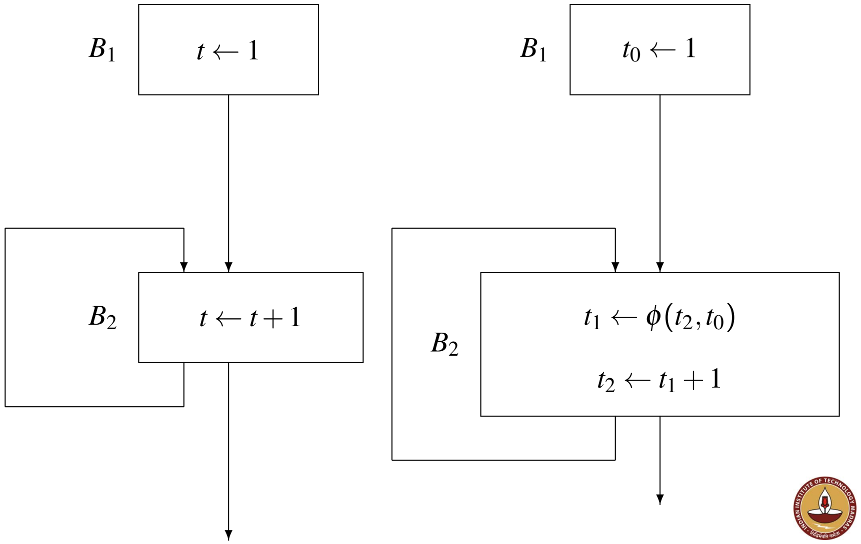
$$\begin{array}{rcl|rcl} v & \leftarrow & 4 & v_0 & \leftarrow & 4 \\ & \leftarrow & v+5 & & \leftarrow & v_0+5 \\ v & \leftarrow & 6 & v_1 & \leftarrow & 6 \\ & \leftarrow & v+7 & & \leftarrow & v_1+7 \end{array}$$

- What about control flow?
  $\Rightarrow$ ϕ-nodes

What is SSA?



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

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

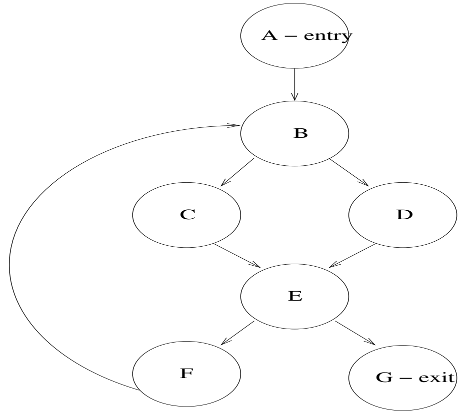
$\text{DOM}(v)$ = Dominator of v

$\text{DOM}^{-1}(v)$ = 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 $\text{DF}(v)$ such that:

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

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

- Computing DF:

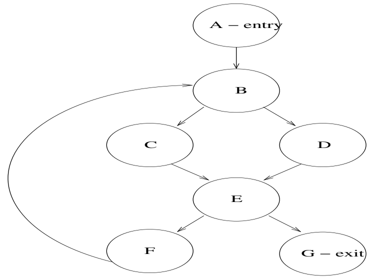
Let

$$\underline{\text{SUCC}}(S) = \bigcup_{s \in S} \underline{\text{SUCC}}(s)$$
$$\text{DOM!}^{-1}(v) = \text{DOM}^{-1}(v) - \{v\}$$

Then

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

Dominance Frontier: Example

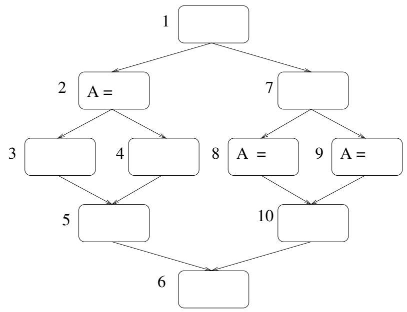


$\text{DF}(v) = \underline{\text{SUCC}}(\text{DOM}^{-1}(v)) - \text{DOM!}^{-1}(v)$
where $\text{DOM!}^{-1}(v) = \text{DOM}^{-1}(v) - \{v\}$

v	$\text{DOM}^{-1}(v)$	$\underline{\text{SUCC}}(\text{DOM}^{-1}(v))$
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	$\text{DOM}^{-1}(v) - \{v\}$	$\text{DF}(v)$
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:

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

The <u>iterated</u> dominance frontier $\mathtt{DF}+(S)$ is the limit of the sequence:

$$\mathtt{DF}_1(S) = \mathtt{DF}(S)$$
$$\mathtt{DF}_{i+1}(S) = \mathtt{DF}(S \cup \mathtt{DF}_i(S))$$

Theorem:

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

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

Input: Set of blocks S
Output: $\mathtt{DF}+(S)$
begin
  $workList \leftarrow \{\}$;
  $\mathtt{DF}+(S) \leftarrow \{\}$;
  foreach $n \in S$ **do**
    $\mathtt{DF}+(S) \leftarrow \mathtt{DF}+(S) \cup \{n\}$;
    $workList \leftarrow workList \cup \{n\}$;
  end
  while $workList \neq \{\}$ **do**
    take n from $workList$;
    foreach $c \in \mathtt{DF}(n)$ **do**
      if $c \notin \mathtt{DF}+(S)$ **then**
        $\mathtt{DF}+(S) \leftarrow \mathtt{DF}+(S) \cup \{c\}$;
        $workList \leftarrow workList \cup \{c\}$;
      end
    end
  end
end

Inserting ϕ-nodes (minimal SSA)

foreach $t \in Temporaries$ **do**
  $S \leftarrow \{n \mid t \in Def(n)\} \cup Entry$;
  Compute $\mathtt{DF}+(S)$;
  foreach $n \in \mathtt{DF}+(S)$ **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 $t \in Temporaries$ **do**
  if $t \in Globals$ **then** // variables live across multiple basic blocks
    $S \leftarrow \{n \mid t \in Defs(n)\} \cup Entry$;
    Compute $\mathrm{DF}+(S)$;
    foreach $n \in \mathrm{DF}+(S)$ **do**
      if t live-in at n **then**
        Insert a ϕ-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.

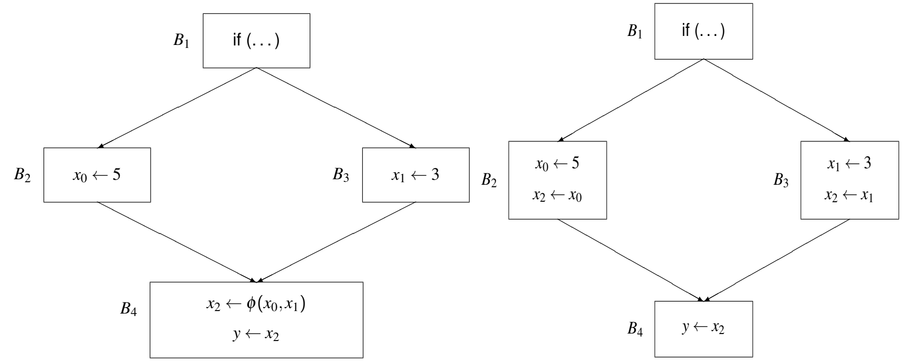
Propagate names from each block to corresponding ϕ-node operands of its successors.

Renaming the temporaries

begin
  foreach $t \in Temporaries$ **do** $count[t] \leftarrow 0$; $stack[t] \leftarrow empty$; $stack[t]$.push(0);
  Call Rename($Entry$);
end
Rename(n) begin
  foreach statement $I \in n$ **do**
    if $stack \not\equiv \phi$ **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 jth predecessor of s;
    foreach $\phi \in s$ **do**
      given t is the jth operand of ϕ;
      $i \leftarrow stack[t]$.top;
      replace jth operand of ϕ with t_i;
  foreach $c \in Children(n)$ **do** Rename(c);
  foreach statement $I \in n$, $t \in Defs(I)$ **do** stack[t].pop();
end

Translating Out of SSA Form

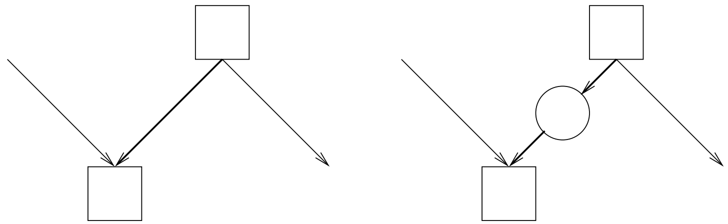
Replace ϕ-nodes with copy statements in predecessors



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?



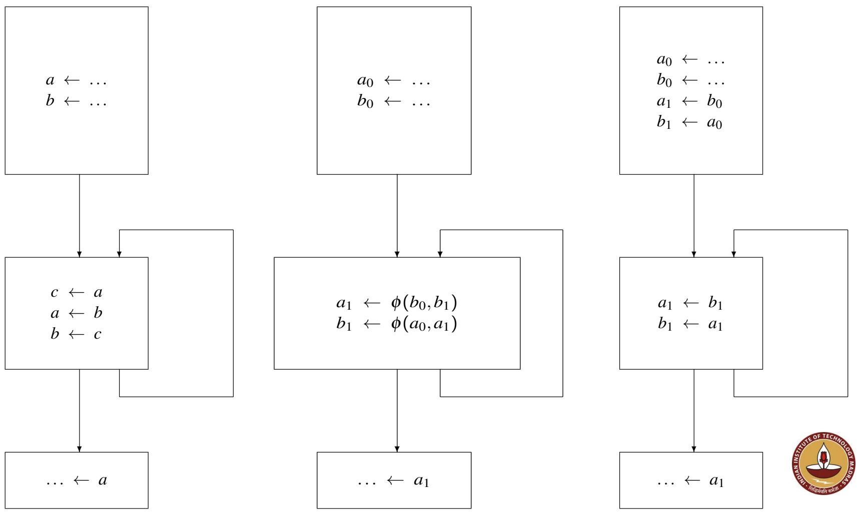
- The lost copy problem:

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

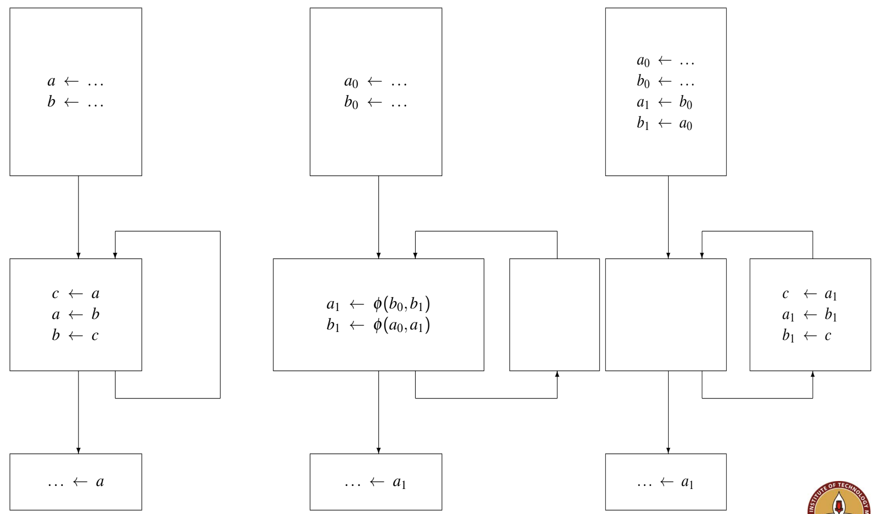

Translation - the swap problem

- 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.

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



Normal Form, Edge-Split Opt SSA, Correct Translation



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 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 this is the first visit to the node
    - If first visit then Perform v = VisitInst(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-$\phi(b)$
  - Elseif if $\exists c$: ExecutableFlag($c \rightarrow b$) = true then VisitInst (b);