The Static Single Assignment Form: Construction and Application to Program Optimizations

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Compiler Design
A new intermediate representation
Incorporates \textit{def-use} information
Every variable has exactly one definition in the program text
  - This does not mean that there are no loops
  - This is a \textit{static} single assignment form, and not a \textit{dynamic} single assignment form
Some compiler optimizations perform better on SSA forms
  - Conditional constant propagation and global value numbering are faster and more effective on SSA forms
\textbf{A \textit{sparse} intermediate representation}
  - If a variable has $N$ uses and $M$ definitions, then \textit{def-use} \textit{chains} need space and time proportional to $N \cdot M$
  - But, the corresponding instructions of uses and definitions are only $N + M$ in number
  - SSA form, for most realistic programs, is linear in the size of the original program
read A, B, C
if (A > B)
    if (A > C) max = A
    else max = C
else if (B > C) max = B
    else max = C
printf (max)
A program is in SSA form, if each use of a variable is reached by exactly one definition.

Flow of control remains the same as in the non-SSA form.

A special merge operator, $\phi$, is used for selection of values in join nodes.

Not every join node needs a $\phi$ operator for every variable.

No need for a $\phi$ operator, if the same definition of the variable reaches the join node along all incoming edges.

Often, an SSA form is augmented with $u-d$ and $d-u$ chains to facilitate design of faster algorithms.

Translation from SSA to machine code introduces copy operations, which may introduce some inefficiency.
Program 2 in non-SSA Text Form

{ Read A; LSR = 1; RSR = A;
  SR = (LSR+RSR)/2;
  Repeat {
    T = SR*SR;
    if (T>A) RSR = SR;
    else if (T<A) LSR = SR;
    else { LSR = SR; RSR = SR}
    SR = (LSR+RSR)/2;
  Until (LSR ≠ RSR);
  Print SR;
}
Program 2 in non-SSA and SSA Form

Y.N. Srikant
Program Optimizations and the SSA Form
Program 3 in non-SSA and SSA Form

Start

i = 0
read n

n <> 1

even(n)
print i
n = n/2
n = 3*n+1
Stop

i = i+1

B1
B2
B3
B5
B6
B7

Start

i_1 = 0
read n_1

i_2 = \Phi(i_3, i_1)
n_2 = \Phi(n_5, n_1)
n_2 <> 1

even(n_2)
print i_2
n_3 = n_2/2
n_4 = 3*n_2+1
Stop

n_5 = \Phi(n_3, n_4)
i_3 = i_2+1

B1
B2
B3
B4
B5
B6
B7
After translation, the SSA form should satisfy the following conditions for every variable \( v \) in the original program.

1. If two non-null paths from nodes \( X \) and \( Y \) each having a definition of \( v \) converge at a node \( p \), then \( p \) contains a trivial \( \phi \)-function of the form \( v = \phi(v, v, ..., v) \), with the number of arguments equal to the in-degree of \( p \).

2. Each appearance of \( v \) in the original program or a \( \phi \)-function in the new program has been replaced by a new variable \( v_i \), leaving the new program in SSA form.

3. Any use of a variable \( v \) along any control path in the original program and the corresponding use of \( v_i \) in the new program yield the same value for both \( v \) and \( v_i \).
Condition 1 in the previous slide is recursive.

- It implies that $\phi$-assignments introduced by the translation procedure will also qualify as assignments to $v$
- This in turn may lead to introduction of more $\phi$-assignments at other nodes

It would be wasteful to place $\phi$-functions in all join nodes.

- It is possible to locate the nodes where $\phi$-functions are essential.
- This is captured by the dominance frontier.
The Join Sets and $\phi$ Nodes

Given $S$: set of flow graph nodes, the set $JOIN(S)$ is

- the set of all nodes $n$, such that there are at least two non-null paths in the flow graph that start at two distinct nodes in $S$ and converge at $n$
  - The paths considered should not have any other common nodes apart from $n$
- The *iterated join set*, $JOIN^+(S)$ is

\[
JOIN^{(1)}(S) = JOIN(S) \\
JOIN^{(i+1)}(S) = JOIN(S \cup JOIN^{(i)}(S))
\]

- If $S$ is the set of assignment nodes for a variable $v$, then $JOIN^+(S)$ is precisely the set of flow graph nodes, where $\phi$-functions are needed (for $v$)
- $JOIN^+(S)$ is termed the *dominance frontier*, $DF(S)$, and can be computed efficiently
JOIN Example -1

- variable $i$: $JOIN^+ (\{B1, B7\}) = \{B2\}$
- variable $n$: $JOIN^+ (\{B1, B5, B6\}) = \{B2, B7\}$
JOIN Example - 2

Y.N. Srikant
Program Optimizations and the SSA Form
Given two nodes $x$ and $y$ in a flow graph, $x$ dominates $y$ ($x \in \text{dom}(y)$), if $x$ appears in all paths from the Start node to $y$.

The node $x$ strictly dominates $y$, if $x$ dominates $y$ and $x \neq y$.

$x$ is the immediate dominator of $y$ (denoted $\text{idom}(y)$), if $x$ is the closest strict dominator of $y$.

A dominator tree shows all the immediate dominator relationships.

The dominance frontier of a node $x$, $\text{DF}(x)$, is the set of all nodes $y$ such that

- $x$ dominates a predecessor of $y$ ($p \in \text{preds}(y)$ and $x \in \text{dom}(p)$)
- but $x$ does not strictly dominate $y$ ($x \notin \text{dom}(y) - \{y\}$)
A definition in node \( n \) forces a \( \phi \)-function in join nodes that lie just outside the region of the flow graph that \( n \) dominates; hence the name *dominance frontier*.

Informally, \( DF(x) \) contains the *first* nodes reachable from \( x \) that \( x \) does not dominate, on *each* path leaving \( x \).

- In example 1 (next slide), \( DF(B1) = \emptyset \), since \( B1 \) dominates all nodes in the flow graph except \( \text{Start} \) and \( B1 \), and there is no path from \( B1 \) to \( \text{Start} \) or \( B1 \).
- In the same example, \( DF(B2) = \{B2\} \), since \( B2 \) dominates all nodes except \( \text{Start} \), \( B1 \), and \( B2 \), and there is a path from \( B2 \) to \( B2 \) (via the back edge).
- Continuing in the same example, \( B5 \), \( B6 \), and \( B7 \) do not dominate any node and the first reachable nodes are \( B7 \), \( B7 \), and \( B2 \) (respectively). Therefore, \( DF(B5) = DF(B6) = \{B7\} \) and \( DF(B7) = \{B2\} \).
- In example 2 (second next slide), \( B5 \) dominates \( B6 \) and \( B7 \), but not \( B8 \); \( B8 \) is the first reachable node from \( B5 \) that \( B5 \) does not dominate; therefore, \( DF(B5) = \{B8\} \).
**DF Example - 1**

DF(x) is the set of all nodes y such that x dominates a predecessor of y, but x does not strictly dominate y.

DF(x) contains the first nodes reachable from x, that x does not dominate.
DF Example - 2

Program Optimizations and the SSA Form

Y.N. Srikant
1. Identify each join node $x$ in the flow graph
2. For each predecessor, $p$ of $x$ in the flow graph, traverse the dominator tree upwards from $p$, till $idom(x)$
3. During this traversal, add $x$ to the $DF$-set of each node met

- In example 1 (second previous slide), consider the join node $B_2$; its predecessors are $B_1$ and $B_7$
  - $B_1$ is also $idom(B_2)$ and hence is not considered
  - Starting from $B_7$ in the dominator tree, in the upward traversal till $B_1$ (i.e., $idom(B_2)$) $B_2$ is added to the $DF$ sets of $B_7$, $B_3$, and $B_2$

- In example 2 (previous slide), consider the join node $B_8$; its predecessors are $B_4$, $B_6$, and $B_7$
  - Consider $B_4$: $B_8$ is added to $DF(B_4)$
  - Consider $B_6$: $B_8$ is added to $DF(B_6)$ and $DF(B_5)$
  - Consider $B_7$: $B_8$ is added to $DF(B_7)$; $B_8$ has already been added to $DF(B_5)$
  - All the above traversals stop at $B_3$, which is $idom(B_8)$
DF Algorithm

{ 
for all nodes $n$ in the flow graph do 
$DF(n) = \emptyset$;
for all nodes $n$ in the flow graph do {
/* It is enough to consider only join nodes */
/* Other nodes automatically get their DF sets */
/* computed during this process */
for each predecessor $p$ of $n$ in the flow graph do {
    $t = p$;
    while ($t \neq idom(n)$) do {
        $DF(t) = DF(t) \cup \{n\}$;
        $t = idom(t)$;
    } 
} 
} 
}
1. Compute $DF$ sets for each node of the flow graph
2. For each variable $v$, place trivial $\phi$-functions in the nodes of the flow graph using the algorithm $place-phi-function(v)$
3. Rename variables using the algorithm $Rename-variables(x,B)$

$\phi$-Placement Algorithm
- The $\phi$-placement algorithm picks the nodes $n_i$ with assignments to a variable
- It places trivial $\phi$-functions in all the nodes which are in $DF(n_i)$, for each $i$
- It uses a work list (i.e., queue) for this purpose
φ-function placement Example

Dominance frontier is written beside BB no.
function \textit{Place-phi-function}(v) \ comments \ v \ is \ a \ variable

\ comments \ This \ function \ is \ executed \ once \ for \ each \ variable \ in \ the \ flow \ graph

\begin{verbatim}
begin

// \textit{has-phi}(B, v) \ is \ \textit{true} \ if \ a \ \phi-\function \ has \ already
// \ been \ placed \ in \ B, \ for \ the \ variable \ v

// \textit{processed}(B) \ is \ \textit{true} \ if \ B \ has \ already \ been \ processed \ once
// \ for \ variable \ v

for \ all \ nodes \ B \ in \ the \ flow \ graph \ do

\hspace{0.08in} \textit{has-phi}(B, v) = false; \ \textit{processed}(B) = false;

end \ for

W = \emptyset; \ comments \ W \ is \ the \ work \ list

// \textit{Assignment-nodes}(v) \ is \ the \ set \ of \ nodes \ containing
// statements \ assigning \ to \ v

for \ all \ nodes \ B \in \textit{Assignment-nodes}(v) \ do

\hspace{0.08in} \textit{processed}(B) = true; \ Add(W, B);

end \ for
\end{verbatim}
The function \textit{place-phi-function}(v) - 2

while $W \neq \emptyset$ do
begin
\linebreak
$B = \text{Remove}(W)$
\linebreak
for all nodes $y \in DF(B)$ do
begin
if (not \textit{has-phi}(y, v)) then
begin
\linebreak
place $< v = \phi(v, v, \ldots, v) >$ in $y$
\linebreak
\textit{has-phi}(y, v) = true
\linebreak
if (not \textit{processed}(y)) then
begin
\textit{processed}(y) = true
\linebreak
\textit{Add}(W, y)
\endend
\linebreak
\linebreak
end
\linebreak
end for
\linebreak
end
\linebreak
\linebreak
Program Optimizations and the SSA Form

 SSA Form Construction Example - 1

SSA form

Dominator tree with dominance frontier
SSA Form Construction Example - 2

Program Optimizations and the SSA Form

Y.N. Srikant
Renaming Algorithm

- The renaming algorithm performs a top-down traversal of the dominator tree.
- A separate pair of version stack and version counter are used for each variable.
  - The top element of the version stack $V$ is always the version to be used for a variable usage encountered (in the appropriate range, of course).
  - The counter $v$ is used to generate a new version number.
- The algorithm shown later is for a single variable only; a similar algorithm is executed for all variables with an array of version stacks and counters.
An SSA form should satisfy the *dominance property*:
- the definition of a variable dominates each use or
- when the use is in a $\phi$-function, the predecessor of the use
Therefore, it is apt that the renaming algorithm performs a top-down traversal of the dominator tree
- Renaming for non-$\phi$-statements is carried out while visiting a node $n$
- Renaming parameters of a $\phi$-statement in a node $n$ is carried out while visiting the appropriate predecessors of $n$
function *Rename-variables*(*x*, *B*) // *x* is a variable and *B* is a block begin  

\[ v_e = \text{Top}(V); \] // *V* is the version stack of *x* 

// variables are defined before use; hence no renaming can // happen on empty stack 

for all statements \( s \in B \) do 
if \( s \) is a non-\( \phi \) statement then 
    replace all uses of *x* in the \( \text{RHS}(s) \) with \( \text{Top}(V) \); 
if \( s \) defines *x* then 
    begin 
        replace *x* with \( x_v \) in its definition; push \( x_v \) onto \( V \); 
        // \( x_v \) is the renamed version of *x* in this definition 
        \[ v = v + 1; \] // *v* is the version number counter 
    end 
end for
The function \textit{Rename-variables}(x,B) \newline

for all successors \(s\) of \(B\) in the flow graph do \newline
\(j = \) predecessor index of \(B\) with respect to \(s\) \newline
for all \(\phi\)-functions \(f\) in \(s\) which define \(x\) do \newline
replace the \(j^{th}\) operand of \(f\) with \(\text{Top}(V)\); \newline
end for \newline
end for \newline
for all children \(c\) of \(B\) in the dominator tree do \newline
\textit{Rename-variables}(x, c); \newline
end for \newline
repeat \(\text{Pop}(V)\); until \(\text{Top}(V) == v_e\); \newline
end \newline
begin // calling program \newline
for all variables \(x\) in the flow graph do \newline
\(V = \emptyset; \ v = 1;\) push 0 onto \(V\); // end-of-stack marker \newline
\textit{Rename-variables}(x, \text{Start}); \newline
end for \newline
end
Renaming Variables Example 0.1

Start

$i_1 = 0$
read $n_1$

$i = \Phi(i, i_1)$
$n = \Phi(n, n_1)$
$n \leftrightarrow 1$

Renaming variables
Processing B1

Renamed (red)
while visiting
node B1

B1

B2

B3 even($n$) print i B4

B6

B5

B7

$n = n/2$
$n = 3n+1$
Stop

$n = \Phi(n, n)$
i = i+1

Order of visiting the blocks:
Start, B1, B2, B3, B5, B6, B7, B4, Stop
(depth-first order on dominator tree)

Y.N. Srikant
Program Optimizations and the SSA Form
Renaming Variables Example 0.2

Start

i₁ = 0
 read n₁

Renaming variables
Processing B2

i₂ = Φ(i, i₁)
n₂ = Φ(n, n₁)
n₂ ↔ 1

B2

Even(n)
print i
B4

n = n/2
n = 3*n + 1
Stop

B5
B6

n = Φ(n, n)
i = i + 1
B7

Order of visiting the blocks:
Start, B1, B2, B3, B5, B6, B7, B4, Stop
(depth-first order on dominator tree)
Renaming Variables Example 0.3

Start

\[ i_1 = 0 \]
read \( n_1 \)

B1

\[ i_2 = \Phi(i, i_1) \]
\[ n_2 = \Phi(n, n_1) \]
\[ n_2 \leftrightarrow 1 \]

B2

B3 even(\( n_2 \))

B4 print i

B6

B5

n = n/2

n = 3*n+1

Stop

Order of visiting the blocks:
Start, B1, B2, B3, B5, B6, B7, B4, Stop
(depth-first order on dominator tree)
Renaming Variables Example 0.4

```
Start

i_1 = 0
read n_1

B1

i_2 = \Phi(i, i_1)
n_2 = \Phi(n, n_1)
n_2 <= 1

B2

even(n_2)
print i

B3

n_3 = n_2/2

B5

n = \Phi(n_3, n)
i = i+1

B7

n = 3*n + 1

B6

Stop

Renaming variables
Processing B5

Renamed (red)
while visiting
node B5

Order of visiting the blocks:
Start, B1, B2, B3, B5, B6, B7, B4, Stop
(depth-first order on dominator tree)
```
Renaming Variables Example 0.5

- **Start**
  - $i_1 = 0$
  - Read $n_1$

- **B1**
  - $i_2 = \Phi(i, i_1)$
  - $n_2 = \Phi(n, n_1)$
  - $n_2 \not\equiv 1$

- **B2**
  - \textbf{even}(n_2)

- **B3**
  - $n_3 = n_2/2$

- **B4**
  - Print $i$

- **B5**
  - $n_4 = 3n_2 + 1$

- **B6**
  - $n = \Phi(n_3, n_4)$
  - $i = i + 1$

- **B7**
  - Stop

- **Renaming variables Processing B6**

- **Renamed (red) while visiting node B6**

- **Order of visiting the blocks:**
  - Start, B1, B2, B3, B5, B6, B7, B4, Stop
  - (depth-first order on dominator tree)
Renaming Variables Example 0.6

Order of visiting the blocks:
Start, B1, B2, B3, B5, B6, B7, B4, Stop
(depth-first order on dominator tree)
Renaming Variables Example 0.7

Start

\[ i_1 = 0 \]
read \( n_1 \)

\[ i_2 = \Phi(i_3, i_1) \]
\[ n_2 = \Phi(n_5, n_1) \]
\[ n_2 \equiv 1 \]

\[ \text{even}(n_2) \]
\[ \text{print } i_2 \]

\[ n_3 = n_2 / 2 \]
\[ n_4 = 3 \times n_2 + 1 \]

\[ n_5 = \Phi(n_3, n_4) \]
\[ i_3 = i_2 + 1 \]

Renaming variables Processing B4

Order of visiting the blocks:
Start, B1, B2, B3, B5, B6, B7, B4, Stop
(depth-first order on dominator tree)
Renaming Variables Example 0.8

Order of visiting the blocks:
Start, B1, B2, B3, B5, B6, B7, B4, Stop
(depth-first order on dominator tree)
Translation to Machine Code - 1

Y.N. Srikant

Program Optimizations and the SSA Form
Translation to Machine Code - 2

Original program:
- $x_1 = 1$
- $x_2 = \Phi(x_1, x_3)$
- $x_3 = x_2 + 1$
- if $p$ then
- return $x_2$

Wrong translation:
- $x_1 = 1$
- $x_2 = x_1$
- $x_3 = x_2 + 1$
- if $p$ then
- return $x_2$

Correct translation:
- $x_1 = 1$
- $t = x_1$
- $x_2 = t$
- $x_3 = x_2 + 1$
- if $p$ then
- return $x_2$

Y.N. Srikant  Program Optimizations and the SSA Form
The parameters of all $\phi$-functions in a basic block are supposed to be read concurrently before any other evaluation begins.
Dead-code elimination

- Very simple, since there is exactly one definition reaching each use
- Examine the $du$-chain of each variable to see if its use list is empty
- Remove such variables and their definition statements
- If a statement such as $x = y + z$ (or $x = \phi(y_1, y_2)$) is deleted, care must be taken to remove the deleted statement from the $du$-chains of $y$ and $z$ (or $y_1$ and $y_2$)

Simple constant propagation
Copy propagation
Conditional constant propagation and constant folding
Global value numbering
Simple Constant Propagation

{ Stmtpile = {S|S is a statement in the program} 
  while Stmtpile is not empty { 
    S = remove(Stmtpile);
    if S is of the form \( x = \phi(c, c, ..., c) \) for some constant c 
      replace S by \( x = c \)
    if S is of the form \( x = c \) for some constant c 
      delete S from the program  
    for all statements T in the du-chain of x do 
      substitute c for x in T; simplify T 
    Stmtpile = Stmtpile ∪ {T}
  } 
}

Copy propagation is similar to constant propagation

• A single-argument \( \phi \)-function, \( x = \phi(y) \), or a copy statement, \( x = y \) can be deleted and \( y \) substituted for every use of \( x \)
The Constant Propagation Framework - An Overview

<table>
<thead>
<tr>
<th>$m(y)$</th>
<th>$m(z)$</th>
<th>$m'(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNDEF</td>
<td>UNDEF</td>
<td>UNDEF</td>
</tr>
<tr>
<td>$c_2$</td>
<td>UNDEF</td>
<td>UNDEF</td>
</tr>
<tr>
<td>NAC</td>
<td>NAC</td>
<td>NAC</td>
</tr>
<tr>
<td>$c_1$</td>
<td>UNDEF</td>
<td>NAC</td>
</tr>
<tr>
<td>$c_2$</td>
<td>$c_1 + c_2$</td>
<td>NAC</td>
</tr>
<tr>
<td>NAC</td>
<td>NAC</td>
<td>NAC</td>
</tr>
</tbody>
</table>

\[
\text{any} \sqcap \text{UNDEF} = \text{any}
\]

\[
\text{any} \sqcap \text{NAC} = \text{NAC}
\]

\[
c_1 \sqcap c_2 = \text{NAC, if } c_1 \neq c_2
\]

\[
c_1 \sqcap c_2 = c_1, \text{ if } c_1 = c_2
\]

Y.N. Srikant

Program Optimizations and the SSA Form
SSA forms along with extra edges corresponding to $d-u$ information are used here
- Edge from every definition to each of its uses in the SSA form (called henceforth as *SSA edges*)

Uses both flow graph and SSA edges and maintains two different work-lists, one for each (*Flowpile* and *SSApile*, resp.)

Flow graph edges are used to keep track of reachable code and SSA edges help in propagation of values

Flow graph edges are added to *Flowpile*, whenever a branch node is symbolically executed or whenever an assignment node has a single successor
SSA edges coming out of a node are added to the SSA work-list whenever there is a change in the value of the assigned variable at the node.

This ensures that all uses of a definition are processed whenever a definition changes its lattice value.

This algorithm needs only one lattice cell per variable (globally, not on a per node basis) and two lattice cells per node to store expression values.

Conditional expressions at branch nodes are evaluated and depending on the value, either one of outgoing edges (corresponding to \texttt{true} or \texttt{false}) or both edges (corresponding to \(\bot\)) are added to the worklist.

However, at any join node, the \texttt{meet} operation considers only those predecessors which are marked \texttt{executable}.
// $G = (N, E_f, E_s)$ is the SSA graph,
// with flow edges and SSA edges, and
// $V$ is the set of variables used in the SSA graph
begin
Flowpile = \{(Start → n) | (Start → n) ∈ E_f \};
SSApile = \emptyset;
for all $e ∈ E_f$ do $e$.executable = false; end for
// $v.cell$ is the lattice cell associated with the variable $v$
for all $v ∈ V$ do $v.cell = \top$; end for
// $y.\text{oldval}$ and $y.\text{newval}$ store the lattice values
// of expressions at node $y$
for all $y ∈ N$ do
  $y.\text{oldval} = \top$; $y.\text{newval} = \top$;
end for
while \((\text{Flowpile} \neq \emptyset)\) or \((\text{SSApile} \neq \emptyset)\) do
begin
  if \((\text{Flowpile} \neq \emptyset)\) then
  begin
    \((x, y) = \text{remove} (\text{Flowpile});\)
    if (not \((x, y)\).\text{executable}) then
    begin
      \((x, y)\).\text{executable} = \text{true};
      if (\(\phi\)-\text{present}(y)) then \text{visit-} \phi(y)
        else if (\text{first-time-visit}(y)) then \text{visit-expr}(y);
    // \text{visit-expr} is called on \(y\) only on the first visit
    // to \(y\) through a flow edge; subsequently, it is called
    // on \(y\) on visits through SSA edges only
    if (\text{flow-outdegree}(y) == 1) then
      // Only one successor flow edge for \(y\)
      \text{Flowpile} = \text{Flowpile} \cup \{(y, z) \mid (y, z) \in \mathcal{E}_f\};
    end
  end
end
// if the edge is already marked, then do nothing
end
if ($SSA pile \neq \emptyset$) then
begin
    $(x, y) = \text{remove}(SSA pile);$  
    if ($\phi$-present($y$)) then visit-$\phi$(y)
        else if ($\text{already-visited}(y)$) then visit-expr($y$); 
            // A false returned by \text{already-visited} implies
            // that $y$ is not yet reachable through flow edges 
        end 
    end
end // Both piles are empty
end
function $\phi$-present($y$) // $y \in \mathcal{N}$
begin
    if $y$ is a $\phi$-node then return true
    else return false
end
function visit-$\phi(y)$ // $y \in \mathcal{N}$
begin
  $y$.newval = $\top$; // $\| y$.instruction.inputs $\|$ is the number of
  // parameters of the $\phi$-instruction at node $y$
  for $i = 1$ to $\| y$.instruction.inputs $\|$ do
    Let $p_i$ be the $i^{th}$ predecessor of $y$;
    if (($p_i$, $y$).executable) then
      begin
        Let $a_i = y$.instruction.inputs[$i$];
        // $a_i$ is the $i^{th}$ input and $a_i$.cell is the lattice cell
        // associated with that variable
        $y$.newval = $y$.newval $\sqcap$ $a_i$.cell;
      end
  end for
if $(y.\text{newval} < y.\text{instruction.output.cell})$ then begin
    $y.\text{instruction.output.cell} = y.\text{newval}$;
    $\text{SSApile} = \text{SSApile} \cup \{(y, z) \mid (y, z) \in \mathcal{E}_s\}$;
end
end

function $\text{already-visited}(y) \ // y \in \mathcal{N}$
// This function is called when processing an SSA edge
begin // Check in-coming flow graph edges of $y$
    for all $e \in \{(x, y) \mid (x, y) \in \mathcal{E}_f\}$
        if $e.\text{executable}$ is true for at least one edge $e$
            then return $true$ else return $false$
    end for
end

function \textit{first-time-visit}(y) \ // y \in \mathcal{N} \\
\text{// This function is called when processing a flow graph edge} \\
\text{begin} \ // \text{Check in-coming flow graph edges of } y \text{ for all } e \in \{(x, y) \mid (x, y) \in \mathcal{E}_t\} \\
\text{if } e.\text{executable} \text{ is true for more than one edge } e \text{ then return } \text{false} \text{ else return } \text{true} \text{ end for} \\
\text{// At least one in-coming edge will have executable true} \\
\text{// because the edge through which node } y \text{ is entered is} \text{ marked as executable before calling this function} \\
\text{end}
function visit-expr(y) // y ∈ ℕ
begin
    Let input₁ = y.instruction.inputs[1];
    Let input₂ = y.instruction.inputs[2];
    if (input₁.cell == ⊥ or input₂.cell == ⊥) then
        y.newval = ⊥
    else if (input₁.cell == ⊤ or input₂.cell == ⊤) then
        y.newval = ⊤
    else // evaluate expression at y as per lattice evaluation rules
        y.newval = evaluate(y);
    // It is easy to handle instructions with one operand
    if y is an assignment node then
        if (y.newval < y.instruction.output.cell) then
            begin
                y.instruction.output.cell = y.newval;
                SSApile = SSApile ∪ {(y, z) | (y, z) ∈ E_s };
            end
        end
Y.N. Srikant       Program Optimizations and the SSA Form
else if $y$ is a branch node then
 begin
  if ($y$.newval $<$ $y$.oldval) then
  begin
   $y$.oldval = $y$.newval;
  end
 switch($y$.newval)
   case ⊥: // Both true and false branches are equally likely
      Flowpile = Flowpile $\cup \{(y, z) \mid (y, z) \in \mathcal{E}_f \}$;
   case true: Flowpile = Flowpile $\cup \{(y, z) \mid (y, z) \in \mathcal{E}_f \text{ and } (y, z) \text{ is the true branch edge at } y \}$;
   case false: Flowpile = Flowpile $\cup \{(y, z) \mid (y, z) \in \mathcal{E}_f \text{ and } (y, z) \text{ is the false branch edge at } y \}$;
 end switch
 end
end
CCP Algorithm - Example 1 - Trace 3

```
C0
  start
  ↓
C1
  a₁ = 10
  ↓
C2
  b = 20
  ↓
C3
  true
  ↓
C4
  yes
  ↓
C5
  a₂ = 30
  ↓
C6
  a₃ = 30
  ↓
C7
  stop
```

Y.N. Srikant
Program Optimizations and the SSA Form
CCP Algorithm - Example 2

Start

B1

a1 = 1
b1 = 1
c1 = 0

B2

b2 = \Phi(b4, b1)
c2 = \Phi(c4, c1)
if c2 < 100

B3

if b2 < 20

B4

Stop

false

B5

b3 = a1
c3 = c2 + 1

true

B6

b5 = c2
c5 = c2 + 1

false

B7

b4 = \Phi(b3, b5)
c4 = \Phi(c3, c5)
CCP Algorithm - Example 2 - Trace 1

Start

B1
a1 = 1
b1 = 1
c1 = 0

B2
b2 = Φ(b4, b1)
c2 = Φ(c4, c1)
if c2 < 100

B3
if b2 < 20

B4
Stop

B5
b3 = a1
c3 = c2 + 1

B6
b5 = c2
c5 = c2 + 1

B7
b4 = Φ(b3, b5)
c4 = Φ(c3, c5)
CCP Algorithm - Example 2 - Trace 2

Start

B1:
- \( a_1 = 1 \)
- \( b_1 = 1 \)
- \( c_1 = 0 \)

B2:
- \( b_2 = \Phi(b_4, b_1) \)
- \( c_2 = \Phi(c_4, c_1) \)
- \( \text{if } c_2 < 100 \)

B3:
- \( \text{if } b_2 < 20 \)

B5:
- \( b_3 = a_1 \)
- \( c_3 = c_2 + 1 \)

B6:
- \( b_5 = c_2 \)
- \( c_5 = c_2 + 1 \)

B7:
- \( b_4 = \Phi(b_3, b_5) \)
- \( c_4 = \Phi(c_3, c_5) \)

B4:
- Stop
CCP Algorithm - Example 2 - Trace 3

Start

B1

a1 = 1
b1 = 1
c1 = 0

B2

b2 = \Phi(b1) = 1
c2 = \Phi(c1) = 0
if c2 < 100: true

B3

if b2 < 20

true

B5

b3 = a1
c3 = c2 + 1

false

B6

b5 = c2
c5 = c2 + 1

B7

b4 = \Phi(b3, b5)
c4 = \Phi(c3, c5)

Stop

B4
**CCP Algorithm - Example 2 - Trace 5**

```
B1
a1 = 1
b1 = 1
c1 = 0

B2
b2 = \Phi(b1) = 1
c2 = \Phi(c1) = 0
if c2 < 100: true

B3
if b2 < 20: true

B5
b3 = a1 = 1
c3 = c2 + 1 = 1

B6
b5 = c2
c5 = c2 + 1

B7
b4 = \Phi(b3, b5)
c4 = \Phi(c3, c5)
```

B4
Stop

Y.N. Srikant
Program Optimizations and the SSA Form
CCP Algorithm - Example 2 - Trace 6

Y. N. Srikant

Program Optimizations and the SSA Form
CCP Algorithm - Example 2 - Trace 7

Start

B1

a1 = 1
b1 = 1
c1 = 0

B2

b2 = \Phi(b4, b1) = 1
b2 = \Phi(c4, c1) = 1
if c2 < 100: unknown

second visit, change in value of c2; no change in value of b2

true

B3

if b2 < 20: true

false

true

B5

b3 = a1 = 1
c3 = c2+1=1

false

B6

b5 = c2
c5 = c2 + 1

false

B7

b4 = \Phi(b3) = 1
c4 = \Phi(c3) = 1

Stop

Y.N. Srikant
Program Optimizations and the SSA Form
CCP Algorithm - Example 2 - Trace 8

Start

B1

a1 = 1
b1 = 1
c1 = 0

B2

b2 = \Phi(b4, b1) = 1
c2 = \Phi(c4, c1) = \bot
if c2 < 100: unknown

true

B3

if b2 < 20: true

false

B4

Stop

B5

b3 = a1 = 1
c3 = c2 + 1 = \bot

true

false

B6

b5 = c2
c5 = c2 + 1

B7

b4 = \Phi(b3) = 1
c4 = \Phi(c3) = 1
CCP Algorithm - Example 2 - Trace 9

Y.N. Srikant

Program Optimizations and the SSA Form
After first round of simplification

Start

B1

a1 = 1
b1 = 1
c1 = 0

B2

b2 = 1
c2 = Φ(c4,c1)
if c2 < 100

B4

false

true

B5

b3 = 1
c3 = c2+1

B7

b4 = 1
c4 = Φ(c3) = c3

Stop
After second round of simplification – elimination of dead code, elimination of trivial $\Phi$-functions, copy propagation etc.
Global value numbering scheme

- Similar to the scheme with extended basic blocks
- Scope of the tables is over the dominator tree
- Therefore more redundancies can be caught
  - For example, an assignment $a_{10} = u_1 + v_1$ in block $B9$ (if present) can use the value of the expression $u_1 + v_1$ of block $B1$, since $B1$ is a dominator of $B9$

No $d-u$ or $u-d$ edges needed

- Uses *reverse post order* on the DFS tree of the SSA graph to process the dominator tree
  - This ensures that definitions are processed before use

- Back edges make the algorithm find *fewer* equivalences (more on this later)
Value Numbering with SSA Forms

- Variable names are not reused in SSA forms
  - Hence, no need to restore old entries in the scoped HashTable when the processing of a block is completed
  - Just deleting new entries will be sufficient
- Any copies generated because of common subexpressions can be deleted immediately
- Copy propagation is carried out during value-numbering
- Ex: Copy statements generated due to value numbering in blocks B2, B4, B5, B6, B7, and B8 can be deleted
- The ValnumTable stores the SSA name and its value number and is global; it is not scoped over the dominator tree (reasons in the next slide)
- Value numbering transformation retains the dominance property of the SSA form
  - Every definition dominates all its uses or predecessors of uses (in case of phi-functions)
Example: An SSA Form

Processing order: B1, B3, B7, B6, B2, B5, B4, B8, B9

Start

B1

a1 = u1+v1
b1 = u2+v2

B2

a2 = u2+v2
b2 = u3+v3
c1 = u2+v2

B3

a3 = u3+v3

B4

a4 = u2+v2

B5

b3 = u2+v2

B6

a6 = u2+v2
c2 = u2+v2

B7

b5 = u3+v3
c3 = u3+v3

B8

a5 = \Phi(a4, a2)
b6 = \Phi(b2, b3)

B9

a7 = \Phi(a5, a6, a3)
b7 = \Phi(b6, b1, b5)
c4 = \Phi(c1, c2, c3)

Stop

Y.N. Srikant

Program Optimizations and the SSA Form
Dominator Tree and Reverse Post order

Postorder on the DFS tree:
Stop, B9, B8, B4, B5, B2, B6, B7, B3, B1, Start

Reverse postorder on the SSA graph that is used with the domimator tree above:
Start, B1, B3, B7, B6, B2, B5, B4, B8, B9, Stop
Global Unscoped \textit{ValnumTable}

- Needed for $\phi$-instructions
- A $\phi$-instruction receives inputs from several variables along different predecessors of a block
- These inputs are defined in the immediate predecessors or dominators of the predecessors of the current block
- For example, while processing block $B_9$, we need definitions of $a_5$, $a_6$, and $a_3$
  - $a_5$, $a_6$: defined in the predecessor blocks, $B_8$, and $B_6$ (resp.)
  - $a_3$: defined in $B_3$, the dominator of the predecessor of $B_9$
- If the \textit{ValnumTable} were to be scoped, only names in $B_1$ would be available while processing $B_9$
- SSA names being unique, unscoped \textit{ValnumTable} does not cause problems
- Making \textit{HashTable} also unscoped is not possible since expressions are not unique
HashTable entry  
(indexed by expression hash value)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value number</th>
<th>Parameters for $\phi$-function</th>
<th>Defining variable</th>
</tr>
</thead>
</table>

ValnumTable  
(indexed by name hash value)

<table>
<thead>
<tr>
<th>Variable name</th>
<th>Value number</th>
<th>Constant value</th>
<th>Replacing variable</th>
</tr>
</thead>
</table>
SSA Value-numbering Example - 1.0

Y.N. Srikant
Program Optimizations and the SSA Form
SSA Value-numbering Example - 1.1

Processing order: B1, B3, B7, B6, B2, B5, B4, B8, B9

Y.N. Srikant  Program Optimizations and the SSA Form
 SSA Value-numbering Example - 1.2

Processing order: B1, B3, B7, B6, B2, B5, B4, B8, B9

Start

B1

a1 = u1 + v1
b1 = u2 + v2

B2

a2 = u2 + v2
b2 = u3 + v3
c1 = u2 + v2

B3

a3 = u3 + v3

B4

a4 = u2 + v2

B5

b3 = u2 + v2

B6

a6 = u2 + v2
c2 = u2 + v2

B7

b5 = u3 + v3
c3 = u3 + v3

B8

a5 = \Phi(a4, a2)
b6 = \Phi(b2, b3)

B9

a7 = \Phi(a5, a6, a3)
b7 = \Phi(b6, b1, b5)
c4 = \Phi(c1, c2, c3)

Stop

Y.N. Srikant
Program Optimizations and the SSA Form
SSA Value-numbering Example - 1.3

Processing B7

Start

B1

\[ a1 = u1+v1 \]
\[ b1 = u2+v2 \]

B2

\[ a2 = u2+v2 \]
\[ b2 = u3+v3 \]
\[ c1 = u2+v2 \]

B3

\[ a3 = u3+v3 \]
\[ b5 \equiv a3 \]
\[ c3 \equiv a3 \]

B4

\[ a4 = u2+v2 \]

B5

\[ b3 = u2+v2 \]

B6

\[ a6 = u2+v2 \]
\[ c2 = u2+v2 \]

B7

\[ b5 = u3+v3 \]
\[ c3 = u3+v3 \]

B8

\[ a5 = \Phi(a4, a2) \]
\[ b6 = \Phi(b2, b3) \]

B9

\[ a7 = \Phi(a5, a6, a3) \]
\[ b7 = \Phi(b6, b1, b5) \]
\[ c4 = \Phi(c1, c2, c3) \]

Processing order:
B1, B3, B7, B6, B2, B5, B4, B8, B9

Stop
SSA Value-numbering Example - 1.4

Processing order: B1, B3, B7, B6, B2, B5, B4, B8, B9
SSA Value-numbering Example - 1.5

Processing B2

B1: a1 = u1+v1
b1 = u2+v2

B2: a2 = u2+v2
b2 = u3+v3
c1 = b1
e4 = u2+v2

B3: a3 = u3+v3

B4: a4 = u2+v2

B5: b3 = u2+v2

B6: a6 = u2+v2
c2 = u2+v2

B7: b5 = u3+v3
c3 = u3+v3

B8: a5 = \Phi(a4, a2)
b6 = \Phi(b2, b3)

B9: a7 = \Phi(a5, a6, a3)
b7 = \Phi(b6, b1, b5)
c4 = \Phi(c1, c2, c3)

Processing order: B1, B3, B7, B6, B2, B5, B4, B8, B9

Stop

Y.N. Srikant | Program Optimizations and the SSA Form
SSA Value-numbering Example - 1.6

Processing order:
B1, B3, B7, B6, B2, B5, B4, B8, B9

Y.N. Srikant  Program Optimizations and the SSA Form
SSA Value-numbering Example - 1.8

Processing B8

Start

B1

a1 = u1 + v1
b1 = u2 + v2

B2

a2 = u2 + v2
b2 = u3 + v3
c1 = b1
e4 = u2 + v2

B3

a3 = u3 + v3

B4

a4 = u2 + v2

B5

b3 = u2 + v2

B6

a6 = b1
c2 = b1

B7

b5 = u3 + v3
c3 = u3 + v3

B8

a5 = \Phi(a4, a2)
b6 = \Phi(b2, b3)

a5 is meaningless

B9

a7 = \Phi(a5, a6, a3)
b7 = \Phi(b6, b1, b5)
c4 = \Phi(c1, c2, c3)

Processing order:
B1, B2, B3, B4, B5, B6, B7, B8, B9

Stop
SSA Value-numbering Example - 1.9

Processing order: B1, B3, B7, B6, B2, B5, B4, B8, B9

Start

B1

a1 = u1+v1
b1 = u2+v2

B2

a2 = u2+v2
b2 = u3+v3
c1 = b1
ea2 ≡ b1
c4 = u2+v2

B3

a3 = u3+v3
b5 ≡ a3
c3 ≡ a3

B4

a4 = b1
ea4 = u2+v2

B5

b3 = u2+v2
b3 ≡ b1

B6

a6 = b1
c2 = b1
ea6 = u2+v2
c2 = u2+v2

B7

b5 = u3+v3
c3 = u3+v3

B8

a5 = \Phi(a4, a2)
b6 = \Phi(b2, b3)
a5 = \Phi(b1, b1) \equiv b1
b6 = \Phi(b2, b1)
a5 is meaningless

B9

a7 = \Phi(a5, a6, a3)
b7 = \Phi(b6, b1, b5)
c4 = \Phi(c4, c2, c3)
c4 is redundant
SSA Value-numbering Example - 1.10

**Final SSA graph**

- **Start**
  - $a_1 = u_1 + v_1$
  - $b_1 = u_2 + v_2$

- **B1**
- **B2**
  - $b_2 = u_3 + v_3$

- **B3**
  - $a_3 = u_3 + v_3$

- **B4**
- **B5**
- **B6**
- **B7**
- **B8**
  - $b_6 = \Phi(b_2, b_1)$

- **B9**
  - $a_7 = \Phi(b_1, b_1, a_3)$
  - $b_7 = \Phi(b_6, b_1, a_3)$

**Stop**
function SSA-Value-Numbering (Block \(B\)) {
    Mark the beginning of a new scope;
    For each \(\phi\)-function \(f\) of the form \(x = \phi(y_1, ..., y_n)\) in \(B\) do {
        search for \(f\) in HashTable;
        //This involves getting the value numbers of the parameters also
        //Dominance property ensures that parameters are assigned
        //either in predecessor or dominator of predecessor of \(B\)
        if \(f\) is meaningless //all \(y_i\) are equivalent to some \(w\)
            replace value number of \(x\) by that of \(w\) in ValnumTable;
            delete \(f\);
        else if \(f\) is redundant and is equivalent to \(z = \phi(u_1, ..., u_n)\)
            replace value number of \(x\) by that of \(z\) in ValnumTable;
            delete \(f\);
        else insert simplified \(f\) into HashTable and ValnumTable;
    }
}
For each assignment $a$ of the form $x = y + z$ in $B$ do {
    search for $y + z$ in $HashTable$;
    //This involves getting value numbers of $y$ and $z$ also
    if present with value number $n$
        replace value number of $x$ by $n$ in $ValnumTable$;
        delete $a$;
    else add simplified $y + z$ to $HashTable$ and $x$ to $ValnumTable$;
}
For each child $c$ of $B$ in the dominator tree do
    //in reverse postorder of DFS over the SSA graph
    SSA-Value-Numbering($c$);
    clean up $HashTable$ after leaving this scope;
}

//Calling program
SSA-Value-Numbering($Start$);
Some times, one or more of the inputs of a \( \phi \)-instruction may not be defined yet

- They may reach through the back edge of a loop
- Such entries will not be found in the \textit{ValnumTable}
- For example, see \textit{a7} and \textit{c4} in the \( \phi \)-functions in block B3 (next slide); their equivalence would not have been decided by the time B3 is processed
- Simply assign a new value number to the \( \phi \)-instruction and record it in the \textit{ValnumTable} and the \textit{HashTable} along with the new value number and the defining variable

If all the inputs are found in the \textit{ValnumTable} (subject to dominance property being satisfied)

- Replace the inputs by the respective entries in the \textit{ValnumTable}
- Now, check whether the \( \phi \)-instruction is either \textit{meaningless} or \textit{redundant}
- If neither, simplify expression and enter into the tables
Example: Effect of Back Edge on Value Numbering

Processing order: B1, B3, B7, B6, B2, B5, B4, B8, B9

Effect of back edge

- u6 and u7 in block B3 are not detected as equivalent, even though a7 and c4 in block B9 are equivalent.

Y.N. Srikant
Program Optimizations and the SSA Form
Processing $\phi$-instructions

Meaningless $\phi$-instruction

- All inputs are identical. For example, see block B8
- It can be deleted and all occurrences of the defining variable can be replaced by the input parameter. \textit{ValnumTable} is updated

Redundant $\phi$-instruction

- There is another $\phi$-instruction in the \textit{same basic block} with exactly the same parameters
- Block B9 has a redundant $\phi$-instruction
- Another $\phi$-instruction from a dominating block cannot be used because the control conditions may be different for the two blocks and hence the two $\phi$-instructions may yield different values at runtime
- \textit{HashTable} can be used to check redundancy
- A redundant $\phi$-instruction can be deleted and all occurrences of the defining variable in the redundant instruction can be replaced by the earlier non-redundant one. Tables are updated
For each variable $v$, walk backwards from each use of $v$, stopping when the walk reaches the definition of $v$.

Collect the block numbers on the way, and the variable $v$ is *live* at the entry/exit (one or both, as the case may be) of each of these blocks.

In the example (next slide), consider uses of the variable $i_2$ in B7 and B4. Traversing upwards till B2, we get: B7, B5, B6, B3, B4(IN and OUT points), and OUT[B2], as blocks where $i_2$ is live.

This procedure works because the SSA forms and the transformations we have discussed satisfy (preserve) the *dominance property*:

- The definition of a variable dominates each use or the predecessor of the use (when the use is in a $\phi$-function).
- Otherwise, the whole SSA graph may have to be searched for the corresponding definition.