Construction of Static Single-Assignment Form

Copyright 2015, Keith D. Cooper & Linda Torczon, all rights reserved.
Students enrolled in Comp 512 at Rice University have explicit permission to make copies of these materials for their personal use.
Faculty from other educational institutions may use these materials for nonprofit educational purposes, provided this copyright notice is preserved.

Citation numbers refer to entries in the EaC2e bibliography.
DEF-USE Chains (see last lecture)

Example

- Figure shows only those DEF-USE chains that involve x
- Figure ignores other variables
- Notice that multiple DEFs can reach a given USE & each USE can reach multiple DEFs
  - Some authors call a connected set of DEFs & USEs as a “web”
  - DEF-USE webs are live ranges in global register allocation [75,74]
Constant Propagation, The Old Way

Transformation: Global Constant Folding
- Along every path to \( p \), \( v \) has same known value
- Specialize computation at \( p \) based on \( v \)'s value

Data-flow problem: Constant Propagation
Domain is the set of pairs \(<v_i,c_i>\) where \( v_i \) is a variable and \( c_i \in C \)

\[
CONSTANTS(b) = \bigwedge_{p \in \text{preds}(b)} f_p(CONSTANTS(p))
\]
- \( \bigwedge \) performs a pairwise meet on two sets of pairs
- \( f_p(x) \) is a block specific function that models the effects of block \( p \) on the \(<v_i,c_i>\) pairs in \( x \)

Constant propagation is a forward flow problem
Constant Propagation, The Old Way

Meet operation requires more explanation

- \(c_1 \land c_2 = c_1\) if \(c_1 = c_2\), else \(\bot\)  
  (bottom & top as expected)

What about \(f_p\)?

- If \(p\) has one statement then

  \[x \leftarrow y\text{ with } \text{CONSTANTS}(p) = \{\ldots<x,l_1>, \ldots<y,l_2>\ldots\}\]

  then \(f_p(\text{CONSTANTS}(p)) = \text{CONSTANTS}(p) - <x,l_1> + <x,l_2>\)

  \[x \leftarrow y \text{ op } z\text{ with } \text{CONSTANTS}(p) = \{\ldots<x,l_1>, \ldots<y,l_2>\ldots>,\ldots<z,l_3>\ldots\}\]

  then \(f_p(\text{CONSTANTS}(p)) = \text{CONSTANTS}(p) - <x,l_1> + <x,l_2 \text{ op } l_3>\)

- If \(p\) has \(n\) statements then

  \[f_p(\text{CONSTANTS}(p)) = f_n(f_{n-1}(f_{n-2}(\ldots f_2(f_1(\text{CONSTANTS}(p)))\ldots))))\]

  where \(f_i\) is the function generated by the \(i^{th}\) statement in \(p\)

\(f_p\) does not fit into the mold of the functions in our Kam-Ullman rapid frameworks.
Constant Propagation over DEF-USE Chains

**Initialization Step**

\[
\text{Worklist} \leftarrow \emptyset
\]

for \( i \leftarrow 1 \) to number of operations

if \( in_1 \) of operation \( i \) is a constant \( c_i \)

then \( \text{Value}(in_1, i) \leftarrow c_i \)

else \( \text{Value}(in_1, i) \leftarrow T \)

if \( in_2 \) of operation \( i \) is a constant \( c_j \)

then \( \text{Value}(in_2, i) \leftarrow c_j \)

else \( \text{Value}(in_2, i) \leftarrow T \)

if \( \text{Value}(in_1, i) \) is a constant &

\( \text{Value}(in_2, i) \) is a constant

then \( \text{Value}(out, i) \leftarrow \text{evaluate op } i \)

\( \text{Worklist} \leftarrow \text{Worklist} \cup \{ i \} \)

else \( \text{Value}(out, i) \leftarrow T \)

**Propagation Step**

\[\text{while ( Worklist } \neq \emptyset \) \]

remove a definition \( i \) from WorkList

for each \( j \in \text{USES}(out, i) \)

let \( x \) be operand where \( j \) occurs

\[\text{Value}(in_x, j) \leftarrow \text{Value}(in_x, j) \]

\[\land \text{Value}(out, i) \]

if \( \text{Value}(in_1, j) \) is a constant &

\( \text{Value}(in_2, j) \) is a constant

then \( \text{Value}(out, j) \leftarrow \text{evaluate op } j \)

\( \text{Worklist} \leftarrow \text{Worklist} \cup \{ j \} \)

else if \( \text{Value}(in_1, j) \) is \( \bot \) or

\( \text{Value}(in_2, j) \) is \( \bot \)

then \( \text{Value}(out, j) \leftarrow \bot \)

\( \text{Worklist} \leftarrow \text{Worklist} \cup \{ j \} \)

Any \( T \) left after fixed point derives from an uninitialized value. What should we do?
DEF-USE Chains

Example

Applying the algorithm involves:
- Initialization step at each operation
  → Two DEFs go on the worklist
  → Others are not constant valued
- A multi-way meet at each use of x
Constant Propagation over DEF-USE Chains

Back to the Example

• At each **USE** that can refer to multiple definitions, the analysis takes the meet of the incoming definitions.

• No work in blocks where info just “passes” through

```
\begin{align*}
x & \leftarrow 17 - 4 \\
x & \leftarrow a + b \\
x & \leftarrow y - z \\
x & \leftarrow 13 \\
z & \leftarrow x \ast q \\
s & \leftarrow w - x
\end{align*}
```
Constant Propagation over DEF-USE Chains

**Complexity**

- Initial step takes $O(1)$ time per operation
- Propagation takes
  - $|USES(v,i)|$ for each $i$ pulled from Worklist
  - Summing over all ops, becomes $|\text{edges in DEF-USE graph}|$
  - A definition can be on the worklist twice
  - $O(|\text{operations}| + |\text{edges in DEF-USE graph}|)$

This sparse-graph\(^1\) approach is faster than the straightforward iterative approach in the Kildall style — both in asymptotic complexity and in practical implementation.

Still, the number of meets is $O(|\text{definitions}|^2)$ in the worst case.
We can do better.

\(^1\) We think of the DEF-USE graph as sparse because it connects the DEF directly to the USE without touching blocks in between them.
DEF-USE Chains and Birth Points

Birth Points Of Values

Value is born here
17 - 4 ∧ y - z

Value is born here
17 - 4 ∧ y - z ∧ 13

Value is born
17 - 4 ∧ y - z ∧ 13 ∧ a+b
DEF-USE Chains and Birth Points

Birth Points Of Values

We should be able to compute the values that we need with fewer meet operations, if only we can find these birth points.

- Need to identify birth points
- Need to insert some artifact to force the evaluation to follow the birth points
- Enter Static Single Assignment form, or SSA

Essentially, we want a DEF-USE graph that has fewer edges.
DEF-USE Chains and Birth Points

Making Birth Points Explicit

- \( x_0 \leftarrow 17 - 4 \)
- \( x_5 \leftarrow a + b \)
- \( x_1 \leftarrow y - z \)
- \( x_3 \leftarrow 13 \)
- \( z \leftarrow x_4 \times q \)
- \( s \leftarrow w - x_6 \)

There are three birth points for \( x \)
Each birth point needs a definition to reconcile the values of $x$

- Insert a $\phi$-function at each birth point
- Rename values so each name is defined once
- Now, each use refers to one definition

⇒ Static Single-Assignment Form
Building Static Single-Assignment Form

**SSA Form**

- Each name is defined exactly once
- Each use refers to exactly one name

**What’s hard**

- Straight-line code is trivial
- Splits in the CFG are trivial
- Joins in the CFG are hard

**Building SSA Form**

- Insert $\phi$-functions at birth points of values
- Rename all values for uniqueness

A $\phi$-function is a special kind of copy that selects one of its parameters.

The choice of parameter is governed by the CFG edge along which control reached the current block.

\[
\begin{align*}
y_1 & \leftarrow \ldots \\
y_2 & \leftarrow \ldots \\
y_3 & \leftarrow \phi(y_1, y_2)
\end{align*}
\]

I know of no machine that implements a $\phi$-function directly in hardware.
SSA Construction Algorithm (High-level sketch)

1. Insert $\phi$-functions
2. Rename values

... *that's all* ...

... *of course, there is some bookkeeping to be done* ...

COMP 512, Rice University
**SSA Construction Algorithm** *(The naïve algorithm)*

1. Insert $\phi$-functions at every join\(^1\) for every name
2. Solve *reaching definitions*
3. Rename each use to the def that reaches it *(will be unique)*

What’s wrong with this approach

- Too many $\phi$-functions *(precision)*
- Too many $\phi$-functions *(space)*
- Too many $\phi$-functions *(time)*
- Need to relate edges to $\phi$-functions parameters *(bookkeeping)*

To do better, we need a more complex approach

---

\(^1\) Every birth point occurs at a definition or a join point in the **CFG**.
Back to the Example and Birth Points

The naïve algorithm inserts too many $\phi$ functions
- Our goal was a $\phi$-function at each birth point
- Naïve algorithm inserts a $\phi$ for each name at each merge in the CFG

The naïve algorithm produces
- Correct SSA form
- More $\phi$’s than any other known algorithm for SSA construction

The rest is optimization (!)

Key Point: number of meet operations that constant propagation performs is now a property of both placement of definitions & CFG structure. In practice, we expect to perform many fewer meets & to see that the number of meets grows more slowly.
SSA Construction Algorithm (Detailed sketch for pruned SSA)

1. Insert $\phi$-functions
   a. calculate dominance frontiers
   b. find global names
      for each name, build a list of blocks that define it
   c. insert $\phi$-functions
      $\forall$ global name $n$
      $\forall$ block $b$ in which $n$ is assigned
      $\forall$ block $d$ in $b$’s dominance frontier
      insert a $\phi$-function for $n$ in $d$
      add $d$ to $n$’s list of defining blocks

Use a checklist to avoid putting blocks on the worklist twice; keep another checklist to avoid inserting the same $\phi$-function twice.
SSA Construction Algorithm

(Detailed sketch)

2. Rename variables in a **pre-order walk over dominator tree**
   (use an array of stacks, one stack per global name)

Staring with the root block, \( b \)

a. generate unique names for result of each \( \phi \)-function and push them on the appropriate stacks

b. rewrite each operation in the block
   i. Rewrite uses of global names with the current version (from the stack)
   ii. Rewrite definition by inventing & pushing new name

c. fill in \( \phi \)-function parameters of successor blocks

d. recurse on \( b \)'s children in the dominator tree

e. \(<\text{on exit from block } b >\) pop names generated in \( b \) from stacks
Dominance Frontiers & Inserting $\phi$-functions

Where does an assignment in block $n$ induce a $\phi$–function?

• $n \text{ DOM } m \Rightarrow$ no need for a $\phi$–function in $m$
  ♦ Definition in $n$ blocks any previous definition from reaching $m$

• If $m$ has multiple predecessors, and $n$ dominates one of them, but not all of them, then $m$ needs a $\phi$–function for each definition in $n$

More formally, $m$ is in the dominance frontier of $n$ if and only if

1. $\exists p \in \text{preds}(m)$ such that $n \in \text{DOM}(p)$, and
2. $n$ does not strictly dominate $m$ \hspace{1cm} (n \notin \text{DOM}(m) - \{ m \})

This dominance frontier is precisely what we need to insert $\phi$–functions:

\[
\text{A def in block } n \text{ induces a } \phi\text{–function in each block in DF}(n).
\]

“Strict” dominance allows a $\phi$–function at the head of a single-block loop.
DOM Example

Flow Graph

Results of iterative solution for DOM

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOM</td>
<td>0</td>
<td>0,1</td>
<td>0,1,2</td>
<td>0,1,3</td>
<td>0,1,3,4</td>
<td>0,1,3,5</td>
<td>0,1,3,6</td>
<td>0,1,7</td>
</tr>
<tr>
<td>IDOM</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>
Example

Dominance Tree

Results of iterative solution for DOM

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOM</td>
<td>0,1</td>
<td>0,1,2</td>
<td>0,1,3</td>
<td>0,1,3,4</td>
<td>0,1,3,5</td>
<td>0,1,3,6</td>
<td>0,1,7</td>
<td></td>
</tr>
<tr>
<td>IDOM</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>
Example

Dominance Frontiers & $\phi$-Function Insertion

- A definition at $n$ forces a $\phi$-function at $m$ iff $n \notin \text{DOM}(m)$ but $n \in \text{DOM}(p)$ for some $p \in \text{preds}(m)$
- DF($n$) is fringe just beyond region $n$ dominates

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOM</td>
<td>0</td>
<td>0,1</td>
<td>0,1,2</td>
<td>0,1,3</td>
<td>0,1,3,4</td>
<td>0,1,3,5</td>
<td>0,1,3,6</td>
<td>0,1,7</td>
</tr>
<tr>
<td>Strict DF</td>
<td>1</td>
<td>1</td>
<td>7</td>
<td>7</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>1</td>
</tr>
</tbody>
</table>

- DF(4) is {6}, so \leftarrow in 4 forces $\phi$-function in 6
- \leftarrow in 6 forces $\phi$-function in DF(6) = {7}
- \leftarrow in 7 forces $\phi$-function in DF(7) = {1}
- \leftarrow in 1 forces $\phi$-function in DF(1) = {1}  
  (halt – the $\phi$ is already there)

For each assignment, we insert the $\phi$-functions
Example

Computing Dominance Frontiers

- Only join points are in $\text{DF}(n)$ for some $n$
- Leads to a simple, intuitive algorithm for computing dominance frontiers
  - For each join point $x$ \( (i.e., |\text{preds}(x)| > 1) \)
  - For each CFG predecessor $p$ of $x$
    - Run from $p$ to $\text{IDOM}(x)$ in the dominator tree, & add $x$ to $\text{DF}(n)$ for each $n$ from $p$ up to but not $\text{IDOM}(x)$

- For some applications (other than building SSA), we need post-dominance, the post-dominator tree, and reverse dominance frontiers, $\text{RDF}(n)$
  - Just dominance on the reverse CFG
  - Reverse the edges & add unique exit node
- We will use these ideas in dead code elimination
SSA Construction Algorithm

1. Insert $\phi$-functions at every join for every name
   a. calculate dominance frontiers
   b. find global names
      for each name, build a list of blocks that define it
   c. insert $\phi$-functions
      \[ \forall \text{ global name } n \]
      \[ \forall \text{ block } b \text{ in which } n \text{ is assigned} \]
      \[ \forall \text{ block } d \text{ in } b\text{'s dominance frontier} \]
      insert a $\phi$-function for $n$ in $d$
      add $d$ to $n$'s list of defining blocks

Step 1.b is not in the CFRWZ [110] algorithms
It produces an SSA form with fewer $\phi$-functions [50]
SSA Construction Algorithm

Finding global names

• Difference between different forms of SSA
• Minimal SSA uses all names [CFRWZ, 110]
• Semi-pruned uses names that are *live* on entry to some block [50]
  ♦ Shrinks name space & number of $\phi$-functions
  ♦ Pays for itself in compile-time speed
• For each “global name”, need a list of blocks where it is defined
  ♦ Drives $\phi$-function insertion
  ♦ $b$ defines $x$ implies a $\phi$-function for $x$ in every $c \in DF(b)$

Pruned SSA adds a test to see if $x$ is live at insertion point

Occasionally, building pruned is faster than building semi-pruned.

Any algorithm that has non-linear behavior in the number of $\phi$-functions will have a size where pruned is the SSA flavor of choice.
Example

Example CFG

• Lots of assignments
• Expression details ellided

Assume \( a, b, c, \) & \( d \) defined before \( B_0 \)
**Example**

**After Ø insertion**
- Lots of new ops
- Renaming is next

Excluding local names avoids Ø’s for y & z

Assume a, b, c, & d defined before $B_0$
SSA Construction Algorithm

One Final Point About $\Phi$-function Insertion

• $\Phi$-functions have an unusual semantics
  ◦ When execution enters a block, all $\Phi$-functions evaluate their arguments, \textit{in parallel}, and then perform their assignments, \textit{in parallel}
  ◦ This behavior allows the compiler to manipulate $\Phi$-functions without worrying about the order in which they appear at the head of a block

• The parallel semantics of $\Phi$-functions will introduce complications when the compiler tries to translate code in SSA form back into executable code
SSA Construction Algorithm (Detailed sketch)

2. Rename variables in a pre-order walk over dominator tree
   (use an array of stacks, one stack per global name)
   Starting with the root block, \( b \)
   a. generate unique names for each \( \phi \)-function and push them on the appropriate stacks
   b. rewrite each operation in the block
      i. Rewrite uses of global names with the current version (from the stack)
      ii. Rewrite definition by inventing \& pushing new name
   c. fill in \( \phi \)-function parameters of successor blocks
   d. recurse on \( b \)'s children in the dominator tree
   e. <on exit from block \( b \)> pop names generated in \( b \) from stacks

1 counter per name for subscripts

Reset the state

Need the end-of-block name for this path

1 counter per name for subscripts

Need the end-of-block name for this path

 Reset the state

Need the end-of-block name for this path
SSA Construction Algorithm

Adding the details ...

for each global name i
  counter[i] ← 0
  stack[i] ← ∅
call Rename(n₀)

NewName(n)
  i ← counter[n]
  counter[n] ← counter[n] + 1
  push nᵢ onto stack[n]
  return nᵢ

Rename(b)
  for each φ-function in b, x ← φ (...)
    rename x as NewName(x)
  for each operation “x ← y op z” in b
    rewrite y as top(stack[y])
    rewrite z as top(stack[z])
    rewrite x as NewName(x)
  for each successor of b in the CFG
    rewrite appropriate φ parameters
  for each successor s of b in dom. tree
    Rename(s)
  for each operation “x ← y op z” in b
    or each phi-function
    pop(stack[x])

Minor engineering nit: assume, up front, that we convert all names into unique small integers
Example

Before processing $B_0$

Assume $a$, $b$, $c$, & $d$ defined before $B_0$

Counts

Stacks

1 1 1 1 0

$i$ has not been defined

COMP 512, Fall 2013
Example

End of $B_0$

Counters

Stacks

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_0$</td>
<td>$b_0$</td>
<td>$c_0$</td>
<td>$d_0$</td>
<td>$i_0$</td>
</tr>
</tbody>
</table>

COMP 512, Fall 2013
Example

End of $B_1$

Counters

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Stacks

<table>
<thead>
<tr>
<th></th>
<th>a₀</th>
<th>b₀</th>
<th>c₀</th>
<th>d₀</th>
<th>i₀</th>
</tr>
</thead>
<tbody>
<tr>
<td>a₁</td>
<td>b₁</td>
<td>c₁</td>
<td>d₁</td>
<td>i₁</td>
<td></td>
</tr>
<tr>
<td>a₂</td>
<td>c₂</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example

End of $B_2$

Counters

Stacks

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>$a_0$</td>
<td>$b_0$</td>
<td>$c_0$</td>
<td>$d_0$</td>
<td>$i_0$</td>
</tr>
<tr>
<td>$a_1$</td>
<td>$b_1$</td>
<td>$c_1$</td>
<td>$d_1$</td>
<td>$i_1$</td>
</tr>
<tr>
<td>$a_2$</td>
<td>$b_2$</td>
<td>$c_2$</td>
<td>$d_2$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$c_3$</td>
<td></td>
</tr>
</tbody>
</table>
Example

Before starting $B_3$

Counters

Stacks

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

COMP 512, Fall 2013
Example

End of $B_3$

Counters

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

Stacks

<table>
<thead>
<tr>
<th>$a_0$</th>
<th>$b_0$</th>
<th>$c_0$</th>
<th>$d_0$</th>
<th>$i_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_1$</td>
<td>$b_1$</td>
<td>$c_1$</td>
<td>$d_1$</td>
<td>$i_1$</td>
</tr>
<tr>
<td>$a_2$</td>
<td></td>
<td>$c_2$</td>
<td></td>
<td>$d_3$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$a_3$</td>
</tr>
</tbody>
</table>

COMP 512, Fall 2013
Example

End of $B_4$

Counters

Stacks

<table>
<thead>
<tr>
<th>$a$</th>
<th>$b$</th>
<th>$c$</th>
<th>$d$</th>
<th>$i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$a_0$</th>
<th>$b_0$</th>
<th>$c_0$</th>
<th>$d_0$</th>
<th>$i_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_1$</td>
<td>$b_1$</td>
<td>$c_1$</td>
<td>$d_1$</td>
<td>$i_1$</td>
</tr>
<tr>
<td>$a_2$</td>
<td>$c_2$</td>
<td>$d_3$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$a_3$</td>
<td>$d_4$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

COMP 512, Fall 2013
Example

End of $B_5$

Counters

Stacks

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_0$</td>
<td>$b_0$</td>
<td>$c_0$</td>
<td>$d_0$</td>
<td>$i_0$</td>
</tr>
<tr>
<td>$a_1$</td>
<td>$b_1$</td>
<td>$c_1$</td>
<td>$d_1$</td>
<td>$i_1$</td>
</tr>
<tr>
<td>$a_2$</td>
<td>$c_2$</td>
<td>$d_3$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$a_3$</td>
<td>$c_4$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example

End of $B_6$

Counters

Stacks

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>4</td>
<td>6</td>
<td>6</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_0$</td>
<td>$b_0$</td>
<td>$c_0$</td>
<td>$d_0$</td>
<td>$i_0$</td>
</tr>
<tr>
<td>$a_1$</td>
<td>$b_1$</td>
<td>$c_1$</td>
<td>$d_1$</td>
<td>$i_1$</td>
</tr>
<tr>
<td>$a_2$</td>
<td>$b_3$</td>
<td>$c_2$</td>
<td>$d_3$</td>
<td></td>
</tr>
<tr>
<td>$a_3$</td>
<td>$c_5$</td>
<td>$d_5$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example

Before \( B_7 \)

Counts

Stacks

\[
\begin{array}{cccccc}
\text{a} & \text{b} & \text{c} & \text{d} & \text{i} \\
4 & 4 & 6 & 6 & 2 \\
\end{array}
\]

\[
\begin{array}{cccccc}
a_0 & b_0 & c_0 & d_0 & i_0 \\
a_1 & b_1 & c_1 & d_1 & i_1 \\
a_2 & c_2 \\
\end{array}
\]
Example

End of $B_7$

Counters

<table>
<thead>
<tr>
<th>$a$</th>
<th>$b$</th>
<th>$c$</th>
<th>$d$</th>
<th>$i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>5</td>
<td>7</td>
<td>7</td>
<td>3</td>
</tr>
</tbody>
</table>

Stacks

<table>
<thead>
<tr>
<th>$a_0$</th>
<th>$b_0$</th>
<th>$c_0$</th>
<th>$d_0$</th>
<th>$i_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_1$</td>
<td>$b_1$</td>
<td>$c_1$</td>
<td>$d_1$</td>
<td>$i_1$</td>
</tr>
<tr>
<td>$a_2$</td>
<td>$b_4$</td>
<td>$c_2$</td>
<td>$d_6$</td>
<td>$i_2$</td>
</tr>
<tr>
<td>$a_4$</td>
<td>$c_6$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example

After renaming
- Semi-pruned SSA form
- We’re done ...

Semi-pruned $\Rightarrow$ only names live in 2 or more blocks are “global names”.

COMP 512, Fall 2013
SSA Construction Algorithm (Pruned SSA)

What’s this “pruned SSA” stuff?
• Minimal SSA still contains extraneous $\phi$-functions
• Inserts some $\phi$-functions where they are dead
• Would like to avoid inserting them

Two ideas
• Semi-pruned SSA: discard names used in only one block [50]
  ♦ Significant reduction in total number of $\phi$-functions
  ♦ Needs only local Live information (cheap to compute)
• Pruned SSA: only insert $\phi$-functions where their value is live $^1$
  ♦ Inserts even fewer $\phi$-functions, but costs more to do
  ♦ Requires computation of $LIVE$ sets (more expensive)

In practice, both are simple modifications to step 1.

SSA Construction Algorithm

We can improve the stack management
• Push at most one name per stack per block
• Thread names together by block
• To pop names for block $b$, use $b$’s thread

This is another good use for a scoped hash table
• Significant reductions in pops and pushes
• Makes a minor difference in SSA construction time
• Scoped table is a clean, clear way to handle the problem