**SSA Technicalities**

**Last Time**
- Introduced SSA

**Today**
- Aliasing in SSA
- Building SSA
- Backward data-flow analyses
- Transforming SSA back to code

**Next Time**
- Using SSA

---

**SSA**

**Merging Definitions**
- $\phi$-functions merge multiple reaching definitions

**Example**

```
 1  v_0 := ...  2  v_2 := \phi(v_0, v_1)  3  v_1 := ...
                  \ldots v_2 \ldots
```
Technicalities

How can we handle aliasing in SSA?

How do we generate SSA?

What about backward data-flow analysis problems?

How do we generate code from SSA?

SSA and Aliasing

Simple solution
– treat all of memory as one variable
– MayDef and MayUse semantics degrade analysis accuracy

Add more functions into SSA to represent semantics
– MayUse and MayDef can be added before the computation of SSA
– Optimizations on SSA must handle the semantics of MayUse and MayDef

Fig. 1. Example of μ, ν and φ

[Chow et al. 96]
Transformation to SSA Form

Two steps
- Insert ϕ-functions
- Rename variables

Where Do We Place ϕ-Functions?

Basic Rule
- If two distinct (non-null) paths x→z and y→z converge at node z, and
  nodes x and y contain definitions of variable v, then a
  ϕ-function for v is inserted at z

\[ v_3 := \phi(v_1, v_2) \]
\[ ...v_3... \]
Approaches to Placing $\phi$-Functions

**Minimal**
- As few as possible subject to the basic rule

**Briggs-Minimal**
- Same as minimal, except $v$ must be live across some edge of the CFG

**Pruned**
- Same as minimal, except dead $\phi$-functions are not inserted

What’s the difference between Briggs Minimal and Pruned SSA?

---

**Briggs Minimal vs. Pruned**

Briggs Minimal will add a $\phi$ function because $v$ is live across the blue edge, but Pruned SSA will not because the $\phi$ function is dead.

Neither Briggs Minimal nor Pruned SSA will place a $\phi$ function in this case because $v$ is not live across any CFG edge.

Why would we ever use Briggs Minimal instead of Pruned SSA?
Machinery for Placing φ-Functions

Recall Dominators
- \( d \text{ dom} i \) if all paths from entry to node \( i \) include \( d \)
- \( d \text{ sdom} i \) if \( d \text{ dom} i \) and \( d \neq i \)

Dominance Frontiers
- The dominance frontier of a node \( d \) is the set of nodes that are “just barely” not dominated by \( d \); i.e., the set of nodes \( n \), such that
  - \( d \) dominates a predecessor \( p \) of \( n \), and
  - \( d \) does not strictly dominate \( n \)
- \( DF(d) = \{ n \mid \exists p \in \text{pred}(n), d \text{ dom} p \text{ and } d \neq \text{sdom} n \} \)

Notational Convenience
- \( DF(S) = \bigcup_{s \in S} DF(s) \)

Nodes in \( \text{Dom}(5) \)
\{4, 5, 12, 13\}

Dominance Frontier Example

\( DF(d) = \{ n \mid \exists p \in \text{pred}(n), d \text{ dom} p \text{ and } d \neq \text{sdom} n \} \)

\( \text{Dom}(5) = \{5, 6, 7, 8\} \)

\( DF(5) = \{4, 5, 12, 13\} \)

What’s significant about the Dominance Frontier?
In SSA form, definitions must dominate uses
### SSA Exercise

```
<table>
<thead>
<tr>
<th></th>
<th></th>
<th>v_3 := . . .</th>
<th></th>
<th>v_4 := \phi(v_2, v_3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>8</td>
<td>v_1 := . . .</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>9</td>
<td>v_2 := . . .</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

DF(8) = \{10\}
DF(9) = \{10\}
DF(2) = \{6\}
DF(\{8,9\}) = \{10\}
DF(10) = \{6\}
DF(\{2,8,9,10\}) = \{6,10\}

### Variable Renaming

**Basic idea**
- When we see a variable on the LHS, create a new name for it
- When we see a variable on the RHS, use appropriate subscript

**Easy for straightline code**

```
x =
  = x
x =
  = x
```

```
x_0 =
  = x_0
x_1 =
  = x_1
```

**Use a stack when there's control flow**
- For each use of x, find the definition of x that dominates it

```
x =
  = x
```

```
x_0 =
  = x_0
```

Traverse the dominance tree
Backward Analyses vs. Forward Analyses

For forward data-flow analysis, at phi node apply meet function

For backward data-flow analysis?

\[
\begin{align*}
  v_0 & := \ldots \\
  v_1 & := \ldots \\
  v_2 & := \phi(v_0, v_1) \\
  \ldots & v_2 \ldots
\end{align*}
\]

Static Single Information Form (SSI)

Figure 5.1: A comparison of SSA (left) and SSI (right) forms.

Ananian’s Masters Thesis, 1997 MIT
Transformation from SSA Form

Proposal
- Restore original variable names (i.e., drop subscripts)
- Delete all φ-functions

Complications
- What if versions get out of order? (simultaneously live ranges)

Alternative
- Perform dead code elimination (to prune φ-functions)
- Replace φ-functions with copies in predecessors
- Rely on register allocation coalescing to remove unnecessary copies

Concepts

SSA and aliasing
- Simple involves may uses and defs to a single memory variable
- Other methods insert more functions into SSA
- For both optimization codes must handle semantics

SSA construction
- Place phi nodes
- Variable renaming

Backward data-flow analyses can use SSI modification to SSA

Transformation from SSA to executable code depends on the optimizations dead-code elimination and copy propagation
Next Time

Assignments
– Schedule for project 2 due Wednesday
– HW1 is due Friday

Lecture
– Using SSA