Static Single Assignment (SSA) Form

A sparse program representation for data-flow.
Computing Static Single Assignment (SSA) Form

Overview:

- What is SSA?
- Advantages of SSA over use-def chains
- “Flavors” of SSA
- Dominance frontiers
- Inserting $\phi$-nodes
- Renaming the temporaries
- Translating out of SSA form

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

\[
\begin{align*}
v &\leftarrow 4 \\
&\leftarrow v + 5 \\
v &\leftarrow 6 \\
&\leftarrow v + 7 \\
v_0 &\leftarrow 4 \\
&\leftarrow v_0 + 5 \\
v_1 &\leftarrow 6 \\
&\leftarrow v_1 + 7 
\end{align*}
\]

- What about control flow?

⇒ \(\phi\)-nodes
What is SSA?
What is SSA?

\[ B_1: \ t \leftarrow 1 \]

\[ B_2: \ t \leftarrow t + 1 \]

\[ B_1: \ t_0 \leftarrow 1 \]

\[ B_2: \ t_1 \leftarrow \phi(t_2, t_0) \]

\[ t_2 \leftarrow t_1 + 1 \]
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 $\phi$-nodes
“Flavors” of SSA

Where do we place \( \phi \)-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 \( \phi \)-node for \( t \) must be inserted at \( z \) (in the new program)

**minimal**
As few as possible subject to condition

**semi-pruned**
As few as possible subject to condition, and \( t \) must be live across some basic block

**pruned**
As few as possible subject to condition, and no dead \( \phi \)-nodes
Dominance Frontiers

From $v$’s point of view, these are the nodes at which other control paths that don’t go through $v$ make their earliest appearance.

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 $x$ does not strictly dominate $w \in \text{DF}(v)$
  \[
  \text{DF}(v) = \{w \mid (\exists u \in \text{PRED}(w))[v \text{ DOM } u] \land v \overline{\text{DOM}}! w\}
  \]

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

- $d$ strictly dominates $v$:
  \[
  d \text{ DOM! } v \iff d \text{ DOM } v \land d \neq v
  \]

- The immediate dominator of $v$, $\text{IDOM}(v)$, is the closest strict dominator of $v$:
  \[
  d \text{ IDOM } v \iff d \text{ DOM! } v \land (\forall w \mid w \text{ DOM! } v)[w \text{ DOM } d]
  \]

$\text{IDOM}(v)$ is $v$’s parent in the dominator tree.
Dominance Frontier: Example

\[ A = \]
\[ DF(8) = \]
\[ DF(9) = \]
\[ DF(2) = \]
\[ DF({8,9}) = \]
\[ DF(10) = \]
\[ DF({2,8,9,10}) = \]
Iterated Dominance Frontier

Extend the dominance frontier mapping from nodes to sets of nodes:

\[ \text{DF}(S) = \bigcup_{n \in S} \text{DF}(n) \]

The *iterated* dominance frontier \( \text{DF}^+(S) \) is the limit of the sequence:

\[
\begin{align*}
\text{DF}_1(S) &= \text{DF}(S) \\
\text{DF}_{i+1}(S) &= \text{DF}(S \cup \text{DF}_i(S))
\end{align*}
\]

**Theorem:**

The set of nodes that need \( \phi \)-nodes for any temporary \( t \) is the iterated dominance frontier \( \text{DF}^+(S) \), where \( S \) is the set of nodes that define \( t \)
Iterated Dominance Frontier Algorithm: $\text{DF}^+(S)$

**Input:** Set of blocks $S$

**Output:** $\text{DF}^+(S)$

\[
\begin{align*}
\text{workList} & \leftarrow \{\} \\
\text{DF}^+(S) & \leftarrow \{\} \\
\textbf{foreach} \; n \in S \; \textbf{do} \\
\quad & \text{DF}^+(S) \leftarrow \text{DF}^+(S) \cup \{n\} \\
\quad & \text{workList} \leftarrow \text{workList} \cup \{n\} \\
\textbf{end} \\
\textbf{while} \; \text{workList} \neq \{\} \; \textbf{do} \\
\quad & \text{take } n \; \text{from } \text{workList} \\
\quad & \textbf{foreach} \; c \in \text{DF}(n) \; \textbf{do} \\
\quad & \quad \textbf{if} \; c \not\in \text{DF}^+(S) \; \textbf{then} \\
\quad & \quad \quad \text{DF}^+(S) \leftarrow \text{DF}^+(S) \cup \{c\} \\
\quad & \quad \quad \text{workList} \leftarrow \text{workList} \cup \{c\} \\
\quad & \quad \textbf{end} \\
\textbf{end} \\
\textbf{end}
\end{align*}
\]
Inserting \( \phi \)-nodes (minimal SSA)

\[
\textbf{foreach} \; t \in \text{Temporaries} \; \textbf{do} \\
S \leftarrow \{ n \mid t \in \text{Def}(n) \} \cup \text{Entry} \\
\text{Compute } \text{DF}^+(S) \\
\textbf{foreach} \; n \in \text{DF}^+(S) \; \textbf{do} \\
\quad \text{Insert a } \phi \text{-node for } t \text{ at } n \\
\textbf{end} \\
\textbf{end}
\]
Inserting $\phi$-nodes for globals (semi-pruned SSA)

Compute *local* liveness: globals are those live across block boundaries (*ie*, used before definition in *any* basic block)

\[
\textbf{foreach } t \in \text{Temporaries do} \\
\textbf{if } t \in \text{Globals then} \\
\quad S \leftarrow \{ n \mid t \in \text{Def}(n) \} \cup \text{Entry} \\
\quad \text{Compute } DF^+(S) \\
\quad \textbf{foreach } n \in DF^+(S) \textbf{ do} \\
\quad \quad \text{Insert a } \phi \text{-node for } t \text{ at } n \\
\textbf{end} \\
\textbf{end} \\
\textbf{end}
\]
Inserting fewest $\phi$-nodes (pruned SSA)

Compute \textit{global} liveness: nodes where each temporary is live-in

\texttt{foreach } $t \in$ \texttt{Temporaries do}

\hspace{1em} \textbf{if } $t \in$ \texttt{Globals then}

\hspace{2em} $S \leftarrow \{ n \mid t \in \text{Defs}(n) \} \cup \text{Entry}$

\hspace{2em} Compute $\text{DF}^+(S)$

\hspace{2em} \texttt{foreach } $n \in \text{DF}^+(S)$ \texttt{do}

\hspace{3em} \textbf{if } $t$ live-in at $n$ \texttt{then}

\hspace{4em} Insert a $\phi$-node for $t$ at $n$

\hspace{3em} \texttt{end}

\hspace{2em} \texttt{end}

\hspace{1em} \texttt{end}

end
Renaming the temporaries

After $\phi$-node insertion, uses of $t$ are either:

**original:** dominated by the definition that computes $t$.

If not, then $\exists$ path to use avoiding definition, which means separate paths from definitions converge between definition and use, thus inserting another definition.

*i.e.*, each use dominated by an evaluation of $t$ or a $\phi$-node for $t$

$\phi$: 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 $\phi$-node operands of its successors.
Renaming the temporaries

foreach \( t \in \text{Temporaries} \) do \( \text{count}[t] \leftarrow 0; \text{stack}[t] \leftarrow \text{empty}; \text{stack}[t].\text{push}(0) \)

\( \text{Rename(Entry)} \)

**proc**\( \text{Rename}(n) \) \( \equiv \)

foreach statement \( I \in n \) do

if \( s \neq \phi \) 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 statement \( I \in n, t \in \text{Defs}(I) \) do \( \text{stack}[t].\text{pop()} \)
Translating Out of SSA Form

Replace \( \phi \)-nodes with copy statements in predecessors

\[
\begin{align*}
B_1 \quad & \text{if (...) } \\
B_2 \quad & x_0 \leftarrow 5 \\
B_3 \quad & x_1 \leftarrow 3 \\
B_4 \quad & x_2 \leftarrow \phi(x_0, x_1) \\
& y \leftarrow x_2
\end{align*}
\]

\[
\begin{align*}
B_1 \quad & \text{if (...) } \\
B_2 \quad & x_0 \leftarrow 5 \\
B_3 \quad & x_1 \leftarrow 3 \\
B_4 \quad & x_2 \leftarrow x_0 \\
& y \leftarrow x_2
\end{align*}
\]
Next Time

Static Single Assignment

- Induction variables (standard vs. SSA)
- Loop Invariant Code Motion with SSA