Static Single Assignment (SSA) Form

A sparse program representation for data-flow.

Overview:

- What is SSA?
- Advantages of SSA over use-def chains
- “Flavors” of SSA
- Dominance frontiers revisited
- 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
  
  $v \leftarrow 4$
  $v \leftarrow v + 5$
  $v \leftarrow 6$
  $v \leftarrow v + 7$
  $v_0 \leftarrow 4$
  $v_0 \leftarrow v_0 + 5$
  $v_1 \leftarrow 6$
  $v_1 \leftarrow v_1 + 7$

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

What is SSA?

B1 \( t \leftarrow 1 \)  
\[ \begin{array}{c}
B2 \ t \leftarrow t + 1 \\
\end{array} \]

B1 \( t_0 \leftarrow 1 \)  
\[ \begin{array}{c}
B2 \ t_1 \leftarrow \phi(t_2, t_0) \\
\end{array} \\
\begin{array}{c}
t_2 \leftarrow t_1 + 1 \\
\end{array} \]

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 by Preston Briggs

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 Revisited

The dominance frontier of \( v \) is the set of nodes \( DF(v) \) such that:

\( v \) dominates a predecessor of \( w \in DF(v) \), but \( x \) does not strictly dominate \( w \in DF(v) \)

\[ DF(v) = \{ w \mid (\exists u \in \text{PRED}(w))[(v \text{ DOM } u) \land v \text{ DOM! } w] \} \]

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

\( d \) strictly dominates \( v \):

\( d \text{ DOM! } v \iff d \text{ DOM } v \text{ and } d \neq v \)

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

Iterated Dominance Frontier

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

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

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

\[ DF_1(S) = DF(S) \]
\[ DF_{i+1}(S) = DF(S \cup DF_i(S)) \]

Theorem:

The set of nodes that need \( \phi \)-nodes for any temporary \( t \) is the iterated dominance frontier \( DF^+(S) \), where \( S \) is the set of nodes that define \( t \)

Iterated Dominance Frontier Algorithm: \( DF^+(S) \)

Input: Set of blocks \( S \)

Output: \( DF^+(S) \)

```
workList ← {}
DF^+(S) ← {}
foreach \( n \in S \) do
    DF^+(S) ← DF^+(S) ∪ \{n\}
    workList ← workList ∪ \{n\}
end
while workList ≠ {} do
    take \( n \) from workList
    foreach \( c \in DF(n) \) do
        if \( c \notin DF^+(S) \) then
            DF^+(S) ← DF^+(S) ∪ \{c\}
            workList ← workList ∪ \{c\}
        end
    end
end
```

Inserting \( \phi \)-nodes (minimal SSA)

```
foreach \( t \in Temporaries \) do
    S ← \{n | t ∈ Def(n)\} ∪ Entry
    Compute DF^+(S)
    foreach \( n ∈ DF^+(S) \) do
        Insert a \( \phi \)-node for \( t \) at \( n \)
    end
end
```

Inserting \( \phi \)-nodes for globals (semi-pruned SSA)

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

```plaintext
foreach \( t \in \text{Temporaries} \) do
  if \( t \in \text{Globals} \) then
    \( S \leftarrow \{ n \mid t \in \text{Def}(n) \} \cup \text{Entry} \)
    Compute \( \text{DF}^+(S) \)
    foreach \( n \in \text{DF}^+(S) \) do
      Insert a \( \phi \)-node for \( t \) at \( n \)
    end
  end
end
```

Inserting fewest \( \phi \)-nodes (pruned SSA)

Compute global liveness: nodes where each temporary is live-in

```plaintext
foreach \( t \in \text{Temporaries} \) do
  if \( t \in \text{Globals} \) then
    \( S \leftarrow \{ n \mid t \in \text{Def}(n) \} \cup \text{Entry} \)
  Compute \( \text{DF}^+(S) \)
  foreach \( n \in \text{DF}^+(S) \) do
    if \( t \) live-in at \( n \) then
      Insert a \( \phi \)-node for \( t \) at \( n \)
  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 a 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.

```plaintext
proc Rename(n) ≡
foreach statement \( l \in n \) do
  if \( s \neq \phi \) then foreach \( t \in \text{Uses}(l) \) do
    \( i \leftarrow \text{stack}[r].\text{top} \)
    replace use of \( t \) with \( t_i \) in \( l \)
  foreach \( t \in \text{Def}(l) \) do
    \( i \leftarrow ++\text{count}[r]. \text{stack}[r].\text{push}(i) \)
    replace \( \text{def of } t \) with \( t_i \) in \( l \)
  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 
      \( i \leftarrow \text{stack}[r].\text{top} \)
      replace \( j \)th operand of \( \phi \) with \( t_i \)
  foreach \( c \in \text{Children}(n) \) do Rename(c)
  foreach statement \( l \in n, t \in \text{Def}(l) \) do stack[r].pop()
```
Translating Out of SSA Form

Replace $\phi$-nodes with copy statements in predecessors

Normal Form, Optimized SSA, Incorrect Translation

Solution: critical edge splitting

Critical Edge:
source has multiple out-edges and target has multiple in-edges

Good for other transformations too (cf landing pads)
Next Time

Static Single Assignment

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

Wegman & Zadeck, *Constant Propagation with Conditional Branches*,