Building SSA Form

Why have SSA?

SSA-form
- Each name is defined exactly once, thus
- 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 ϕ-functions at birth points
- Rename all values for uniqueness

Birth Points (a notion due to Tarjan)

Consider the flow of values in this example

The value $x$ appears everywhere
It takes on several values.
- Here, $x$ can be 13, $y-z$, or $17-4$
- Here, it can also be $a+b$
If each value has its own name ...
- Need a way to merge these distinct values
- Values are "born" at merge points

Birth Points (cont)

Consider the flow of values in this example

New value for $x$ here
17 - 4 or $y - z$

New value for $x$ here
13 or (17 - 4 or $y - z$)

New value for $x$ here
$a+b$ or (13 or (17-4 or $y$-$z$))
Birth Points (cont)
Consider the flow of values in this example

\[ x \leftarrow 17 - 4 \]
\[ x \leftarrow a + b \]
\[ x \leftarrow y - z \]
\[ x \leftarrow 13 \]
\[ z \leftarrow x \times q \]

- All birth points are join points
- Not all join points are birth points
- Birth points are value-specific ...

These are all birth points for values

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
- Rename all values for uniqueness

\( \phi \)-function is a special kind of a move instruction that selects one of its parameters.
The choice of parameter is governed by the CFG edge along which control reached the current block.

However, real machines do not implement a \( \phi \)-function 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 ...

SSA Construction Algorithm (Less high-level)
1. Insert \( \Phi \)-functions at every join for every name
2. Solve reaching definitions
3. Rename each use to the def that reaches it (will be unique)
Reaching Definitions

Domain is \[\text{DEFINITIONS}\], same as number of operations

The equations

\[
\text{REACHES}(n_0) = \emptyset
\]

\[
\text{REACHES}(n) = \bigcup_{p \in \text{preds}(n)} \text{DEFOUT}(p) \cup \left( \text{REACHES}(p) \cap \text{SURVIVED}(p) \right)
\]

- \(\text{REACHES}(n)\) is the set of definitions that reach block \(n\)
- \(\text{DEFOUT}(n)\) is the set of definitions in \(n\) that reach the end of \(n\)
- \(\text{SURVIVED}(n)\) is the set of defs not obscured by a new def in \(n\)

Computing \(\text{REACHES}(n)\)

- Use any data-flow method \((i.e., \text{the iterative method})\)
- This particular problem has a very-fast solution \((\text{Zadeck})\)


SSA Construction Algorithm (Less high-level)

1. Insert \(\Phi\)-functions at every join for every name
2. Solve reaching definitions
3. Rename each use to the def that reaches it \((\text{will be unique})\)

What's wrong with this approach

- Too many \(\Phi\)-functions \((\text{precision})\)
- Too many \(\Phi\)-functions \((\text{space})\)
- Too many \(\Phi\)-functions \((\text{time})\)
- Need to relate edges to \(\Phi\)-functions parameters \((\text{bookkeeping})\)

To do better, we need a more complex approach

SSA Construction Algorithm (Less high-level)

1. Insert \(\Phi\)-functions
   a.) calculate dominance frontiers \(\text{Moderately complex}\)
   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 defined
      \(\forall\) block \(d\) in \(b\)'s dominance frontier
      insert a \(\Phi\)-function for \(n\) in \(d\)
      add \(d\) to \(n\)'s list of defining blocks

   Compute list of blocks where each name is assigned. Use this list as the worklist.

   Creates the iterated dominance frontier

   This adds to the worklist!

   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 (Less high-level)

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

   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

   Reset the state

   Need the end-of-block name for this path

*9

*10

*11

*12
Aside on Terminology: Dominators

Definitions
- $x$ dominates $y$ if and only if every path from the entry of the control-flow graph to the node for $y$ includes $x$
- By definition, $x$ dominates $x$
- We associate a Dom set with each node
- $|\text{Dom}(x)| \geq 1$

Immediate dominators
- For any node $x$, there must be a $y$ in $\text{Dom}(x)$, $y \neq x$ such that $y$ is closest to $x$
- We call this $y$ the immediate dominator of $x$
- As a matter of notation, we write this as $\text{IDom}(x)$
- By convention, $\text{IDom}(x_0)$ is not defined for the entry node $x_0$

SSA Construction Algorithm (Low-level detail)

Computing Dominance
- First step in $\Phi$-function insertion computes dominance.
  - A node $n$ dominates $m$ iff $n$ is on every path from $n_0$ to $m$.
    - Every node dominates itself
    - $n$’s immediate dominator is its closest dominator, $\text{IDom}(n)$

\[
\text{Dom}(n_0) = \{ n_0 \}
\]
\[
\text{Dom}(n) = \{ n \} \cup (\cap p \in \text{pred}(n) \text{ Dom}(p))
\]

Computing $\text{DOM}$
- These equations form a rapid data-flow framework.
- Iterative algorithm will solve them in $d(G) + 3$ passes.
  - Each pass does $N$ unions & $E$ intersections,
  - $E$ is $O(N^2) \Rightarrow O(N^2)$ work

Results of iterative solution for $\text{DOM}$

<table>
<thead>
<tr>
<th>Iteration</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></td>
<td>0</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>1</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>0.1,3,4,5</td>
<td>0.1,3,4,5,6</td>
<td>0.1,3,4,5,6,7</td>
</tr>
<tr>
<td>2</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>0.1,3,4,5</td>
<td>0.1,3,4,5,6</td>
<td>0.1,3,4,5,6,7</td>
</tr>
</tbody>
</table>

Flow Graph
For some applications, we need post-dominance to DF(\text{Dominance Frontiers & \Phi-function}) function for in which ID is defined to DF\left|_{\text{B}}\right) and ID to DF(\text{B}).

There are asymptotically faster algorithms. With the right data structures, the iterative algorithm can be made faster. See Cooper, Harvey, and Kennedy.

### Example

**SSA Construction Algorithm (Reminder)**

1. Insert \Phi-functions at some join points
   a.) calculate dominance frontiers
   Needs a little more detail for each name, build a list of blocks that define it
   b.) find global names
   c.) insert \Phi-functions

\[ \forall \text { global name } n \]
\[ \forall \text { block } b \text { in which } n \text { is defined } \]
\[ \forall \text { block } d \text { in } B's \text { dominance frontier } \]

\[ \text { insert a } \Phi \text {-function for } n \text { in } d \]
\[ \text { add } d \text { to its list of defining blocks } \]
**SSA Construction Algorithm**

Finding global names

- Different between two forms of SSA
- Maximal uses all names
  - Semi-pruned SSA uses names that are live on entry to some block
    - Shrinks name space & number of Φ-functions
    - Pays for itself in compile-time speed
- For each "global name", need a list of blocks where it is defined
- Drives Φ-function insertion

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

---

**SSA Construction Algorithm (Less high-level)**

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, 1 counter per name for subscripts
   a.) generate unique names for each Φ-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 Φ-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

---

**Example**

With all the Φ-functions
- Lots of new ops
- Renaming is next

---

**SSA Construction Algorithm (Less high-level)**

Adding all the details ...

for each global name i
   counter[i] ← 0
   stack[i] ← Ø
   call Rename(n_i)

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
   pop(stack(x))
Example

Before processing $B_0$

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

i has not been defined

Example

End of $B_0$

Example

End of $B_1$

Example

End of $B_2$

Example

End of $B_3$
Example

Before starting $B_3$

$\phi = a \leftarrow \cdots$

$i > 100 \downarrow$

Counters

$\begin{array}{cccc}
3 & 3 & 4 & 3 \\
\end{array}$

Stacks

$\begin{array}{cccc}
a_0 & b_0 & c_0 & d_0 \ \\
\end{array}$

$\begin{array}{cccc}
a & b & c & d & i \\
a, b, c, d, i \ \\
a & c, i \\
\end{array}$

Example

End of $B_3$

$\phi = a \leftarrow \cdots$

$i > 100 \downarrow$

Counters

$\begin{array}{cccc}
4 & 3 & 4 & 4 \\
\end{array}$

Stacks

$\begin{array}{cccc}
a_0 & b_0 & c_0 & d_0 \ \\
\end{array}$

$\begin{array}{cccc}
a, b, c, d, i, a, c, d \ \\
a, a \\
\end{array}$

Example

End of $B_4$

$\phi = a \leftarrow \cdots$

$i > 100 \downarrow$

Counters

$\begin{array}{cccc}
4 & 3 & 4 & 3 \\
\end{array}$

Stacks

$\begin{array}{cccc}
a_0 & b_0 & c_0 & d_0 \ \\
\end{array}$

$\begin{array}{cccc}
a, b, c, d, a, c, d \ \\
a, c, d \\
\end{array}$

Example

End of $B_5$

$\phi = a \leftarrow \cdots$

$i > 100 \downarrow$

Counters

$\begin{array}{cccc}
4 & 3 & 5 & 5 \\
\end{array}$

Stacks

$\begin{array}{cccc}
a_0 & b_0 & c_0 & d_0 \ \\
\end{array}$

$\begin{array}{cccc}
a, b, c, d, a, c, d \ \\
a, a, c, d \\
\end{array}$
**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
  - Significant reduction in total number of $\Phi$-functions
  - Needs only local liveness information (cheap to compute)
- *Pruned SSA*: only insert $\Phi$-functions where their value is live
  - Inserts even fewer $\Phi$-functions, but costs more to do
  - Requires global live variable analysis (more expensive)

In practice, both are simple modifications to step 1.

---

**SSA Deconstruction**

At some point, we need executable code
- Machines do not implement $\Phi$ instructions
- Need to fix up the flow of values

Basic idea
- Insert copies $\Phi$-function pred's
- Simple algorithm
- Adds lots of copies
  - Most of them coalesce away

---

**SSA Construction Algorithm**

We can improve the stack management
- Push at most one name per stack per block (save push & pop)
- 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