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 \( \Phi \)-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, \( x \) can also be \( a+b \)
- 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

Birth Points (cont)
Consider the flow of values in this example

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

A \( \Phi \)-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 ...

Reaching Definitions

The equations

$\text{REACHES}(n_0) = \emptyset$

$\text{REACHES}(n) = \bigcup_{p \in \text{preds}(n)} \text{DEFUT}(p) \cup \text{REACHES}(p) \cap \text{SURVIVED}(p)$

- $\text{REACHES}(n)$ is the set of definitions that reach block $n$
- $\text{DEFUT}(n)$ 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
- This particular problem has a very-fast solution (Zadeck)

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

SSA Construction Algorithm (Less high-level)

1. Insert $\Phi$-functions of every join 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 (precise)
- Too many $\Phi$-functions (space)
- Too many $\Phi$-functions (new)
- Need to relate edges to $\Phi$-functions parameters (bookkeeping)

To do better, we need a more complex approach

SSA Construction Algorithm (Less high-level)

1. Insert $\Phi$-functions
   a.) calculate dominance frontiers
   b.) find global names
      i. for each name, build a list of blocks that define it
   c.) insert $\Phi$-functions
      i. block $d$ in which $n$ is defined
      ii. block $d$ in $n$’s dominance frontier

2. Rename variables in a pre-order walk over dominator tree
   (use an array of stacks, one stack per global name)
   - Use a checklist to avoid putting blocks on the worklist twice; keep another checklist to avoid inserting the same $\Phi$-function twice
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 \( \text{Dom}(x) \) 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 \( x_0 \)

Dominator tree

Dominator sets

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 \( m \).
- \( n \)'s immediate dominator is its closest dominator, \( \text{IDom}(x) \)
  \[ \text{Dom}(n) = \{ n \} \cup \bigcap_{p \in \text{preds}(n)} \text{Dom}(p) \]

Computing 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 \( O(N^2) \)

Example

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

Example

Example

Dominator Frontiers & \( \Phi \)-Function Insertion
- A definition at \( n \) forces a \( \Phi \)-function at all \( m \) s.t. \( n \text{-} \text{Dom}(m) \) but \( n \n \text{-} \text{Dom}(p) \) for some \( p \text{-} \text{pred}(m) \)
- \( \text{DF}(x) \) is fringe just beyond region \( x \)-dominates
  \[ \text{DF}(x) = \{ y | \text{DF}(y) \subseteq \text{DF}(x) \} \]

Example

Example
Computing Dominance Frontiers
- Only join points are in DF(n) for some n
- Leads to a simple, intuitive algorithm for computing dominance frontiers
  For each join point x
  For each CFG predecessor of x
    Run up to IDOM(x) in the dominator tree, adding x to DF(n) for each n between x and IDOM(x)

For some applications, we need post-dominance, the post-dominator tree, and reverse dominance frontiers, RDF(n)
> Just dominance on the reverse CFG
> Reverse the edges & add unique exit node

We will use these in dead code elimination using SSA

SSA Construction Algorithm (Reminder)
1. Insert \( \Phi \)-functions of some join points
   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 defined
      insert a \( \Phi \)-function for n in b
      add b to n's list of defining blocks

   Needs a little more detail

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, 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

Adding all the details ...
for each global name n
    counter[n] = 0
    stack[n] = Ø
for each \( \Phi \)-function in b, x \( \Phi \)(...) rename x as NewName(x)
for each operation “x \( \Phi \) 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 s of b in the CFG
    rewrite appropriate \( \Phi \) parameters for each successor s of b in dom. tree
    Rename(s)
for each operation “x \( \Phi \) y op z” in b
    pop(stack[z])
    return s

Here are a few examples ...
We're done …
• Semi-pruned SSA form

Example

After renaming
• Semi-pruned SSA form
• We're done …

Semi-pruned: all only names live in 2 or more blocks are "global names"
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 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

SSA Deconstruction

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

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