Building SSA Form

Slides mostly based on Keith Cooper’s set of slides (COMP 512 class at Rice University, Fall 2002).
Used with kind permission.

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 $\theta$-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, $y_{12}$, or 17-4
- Here, it can also be $y_{13}$
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
13 or (17-4 or $y_{12}$)

New value for $x$ here
17-4 or $y_{12}$

New value for $x$ here
13 or (17-4 or $y_{12}$)

Birth Points (cont)

Consider the flow of values in this example:

- 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 $\theta$-functions at birth points
  - Rename all values for uniqueness

A $\theta$-function is a special kind of copy that selects one of its parameters.
The choice of parameter is governed by the CFG edge along which control reached the current block.

$y_{12} = \theta_{13}$
However, real machines do not implement a $\theta$-function in hardware.
SSA Construction Algorithm (High-level sketch)

1. Insert 0-functions
2. Rename values

... that's all ...

... of course, there is some bookkeeping to be done ...

SSA Construction Algorithm (Less high-level)

1. Insert 0-functions of every join for every name
2. Solve reaching definitions
3. Rename each use to the def that reaches it (will be unique)

Reaching Definitions

The equations

\[ \text{REACHES}(n) = \emptyset \]
\[ \text{REACHES}(n) = \{ \cup_{p \in \text{pre}(n)} \text{DEPOUT}(p) \cup (\text{REACHES}(\rho) \cap \text{SURVIVED}(\rho)) \} \]

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

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


SSA Construction Algorithm (Less high-level)

1. Insert 0-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 0-functions (imprecise)
- Too many 0-functions (space)
- Too many 0-functions (time)
- Need to relate edges to 0-functions parameters (bookkeeping)

To do better, we need a more complex approach

SSA Construction Algorithm (Less high-level)

1. Insert 0-functions
   a) calculate dominance frontiers
   b) find global names
   for each name, build a list of blocks that define it
   c) insert 0-functions
      \[ \forall \text{global name } n \]
      \[ \forall \text{block } b \text{ in which } n \text{ is defined} \]
      \[ \text{create the (named) dominance frontier} \]
      \[ \text{insert a } \theta \text{-function for } n \text{ in } b \]
      \[ \text{add to its list of defining blocks} \]
      \[ \text{use a checklist to avoid putting blocks on the worklist twice; keep another checklist to avoid inserting the same } \theta \text{-function twice.} \]

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 \( \theta \)-function and push them on the appropriate stacks
   b) rewrite each operation in the block
      i. Extract uses of global names with the current version (from the stack)
      ii. Rewrite definition by inserting \& pushing new name
   c) fill in \( \theta \)-function parameters of successor blocks
   d) recurse on \( \theta \)-children in the dominator tree
   e) on exit from block \( \theta \)-pop names generated in \( \theta \) from stacks

KT 2006
Aside on Terminology: Dominators

Definitions
- \text{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, \(x \ominus \dominates \ y\)
- We associate a \(\text{Dom}\) \set \ with \ each \ node
- \(|\text{Dom}(x)| \leq 1\)

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 \text{immediate dominator} of \(x\)
- As a matter of notation, we write this as \(\text{IDom}(x)\)
- By convention, \(\text{IDom}(x)\) is not defined for the entry node \(x\).

SSA Construction Algorithm (Low-level detail)

Computing Dominance
- First step in \(\Phi\)-function insertion computes dominance.
- A node \(n\) \ominus \text{dominates} \(m\) if \(n\) is on every path from \(n\) to \(m\)
  - Every node dominates itself
  - \(n\)'s immediate dominator is its closest dominator, \(\text{IDom}(n)\)
  - \(|\text{Dom}(n)| = (\ n \ominus \text{dominates} \ n\)

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

Example

Progress of iterative solution for \(\text{Dom}\)

<table>
<thead>
<tr>
<th>Iteration</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Results of iterative solution for \(\text{Dom}\)

<table>
<thead>
<tr>
<th>Iteration</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example

Dominators (cont)

- Dominators have many uses in program analysis & transformation
  - Finding loops
  - Building SSA form
  - Making code motion decisions

<table>
<thead>
<tr>
<th>Dominator sets</th>
<th>Block</th>
<th>Dom</th>
<th>IDom</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>A,B</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>A,C</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>A,C,D</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>A,C,E</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>A,C,F</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>A,G</td>
<td>A</td>
<td></td>
</tr>
</tbody>
</table>

Let’s look at how to compute dominators...

Example

Dominance Frontiers & \(\Phi\)-Function Insertion

- \(A\) definition at \(n\) \ominus \text{forces a} \(\Phi\)-function in \(m\)
- \(n \ominus\text{Dom}(m)\)
- \(n \ominus \text{Dom}(m)\) for some \(p \ominus \text{preds}(m)\)
- \(\Phi\)-function at \(m\)

There are asymptotically faster algorithms. With the right data structures, the iterative algorithm can be made faster.

See Cooper, Harvey, and Kennedy.
**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
- Reverse the edges & add unique exit node
- We will use these in dead code elimination using SSA

**SSA Construction Algorithm (Reminder)**

1. Insert θ-functions at some join points
   a) calculate dominance frontiers
   b) find global names for each name, build a list of blocks that define it
   c) insert θ-functions
     - for global name n in block b in which n is defined
     - for each operation "x = y op z" in b
     - for each successor of b in the CFG
     - for each successor s of b in the dominator tree
     - for each operation "x = y op z" in b
   Needs a little more detail

2. Rename variables in a pre-order walk over dominator tree
   - Storing with the root block b
     - 1 counter per name for subscript
   a) generate unique names for each θ-function
   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 θ-function parameters of successor blocks
   d) recurse on s’s children in the dominator tree
   e) an exit from block b & pop names generated in b from stacks
   - Need the end-of-block name for this path

**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 θ-function
   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 θ-function parameters of successor blocks
   d) recurse on s’s children in the dominator tree
   e) an exit from block b & pop names generated in b from stacks
   - Need the end-of-block name for this path

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

Adding all the details...

for each θ-function in b, x = Φ(...)
rename x as NewName(x)
for each operation "x = y op z" in b
  rewrite x as top(stack(x))
  rewrite y as top(stack(y))
  rewrite z as NewName(z)
for each successor s of b in the CFG
  rewrite appropriate θ-parameters for each successor
  of b in dom. tree
  Rename(s)
  for each operation "x = y op z" in b
  push onto stack(s)
Example

End of $B_i$

Example

End of $B_j$

Example

Before $B_j$

Example

End of $B_j$

Example

After renaming

- Semi-pruned SSA form
- We're done...

Semi-pruned ⇒ 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 0-functions
- Inserts some 0-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 0-functions
    - Needs only local liveness information
      (cheap to compute)
  - Pruned SSA: only insert 0-functions where their value is live
    - Inserts even fewer 0-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
- Few machines implement 0-operations
- Need to fix up the flow of values

Basic idea:
- Insert copies 0-function pred's
- Simple algorithm
  - Works in most cases
- Adds lots of copies
  - Most of them coalesce away