An Introduction to Static Single Assignment Form

Copyright 2019, Keith D. Cooper, Linda Torczon & Zoran Budimlić, all rights reserved.

Students enrolled in Comp 506 at Rice University have explicit permission to make copies of these materials for their personal use.

Faculty from other educational institutions may use these materials for nonprofit educational purposes, provided this copyright notice is preserved.

Chapter 9 in EaC2e.
Static Single-Assignment Form

The Fundamental Idea: A Name Space In The IR That Simplifies Analysis

• Each name in the code is defined exactly once
• Each use refers to exactly one name

Why do we want these properties?

• Provide a unique name for each value
  ♦ Remember the naming issue in LVN?
• Expose both the flow of values & the ranges of values
  ♦ Can simplify implementation of analyses & transformations

Why not use SSA for everything?

• Some compilers, in effect, do use SSA as their primary IR
  ♦ LLVM/CLANG, FLANG
• Some aspects of SSA are not easily implementable in code
  ♦ $\phi$-functions have no direct analog in most ISAs

COMP 506, Spring 2019
Static Single-Assignment Form

The Fundamental Idea: A Name Space In The IR That Simplifies Analysis

- Each name in the code is defined exactly once
- Each use refers to exactly one name

What is easy?
- Straight-line code is trivial
  - New name at every definition
  - Add a subscript, bump it at definition, rewrite with current subscript
- Splits in the CFG are trivial
  - Each block inherits the name space of its (sole) predecessor

What is hard?
- Joins (or merge points) in the CFG are hard
  - Block can inherit two names for the same original program variable
  - Need a mechanism to reconcile such conflicts
At this point, consider just the values of x

- There are 4 definitions of x & 2 uses of x
  - At each definition, x takes on a new value
  - Think about those values
- Understanding those values can help the compiler optimize the code

- We will use the symbol $\land$, pronounced “meet” to represent the operation that happens when two paths converge
  - In AVAIL and DOM, $\land$ was intersection
Static Single-Assignment Form

Handling Merge Points in the CFG

At this point, consider just the values of $x$

- There are 4 definitions of $x$ & 2 uses of $x$
  - At each definition, $x$ takes on a new value
  - Think about those values
- If we rename the $x$’s for uniqueness, the situation becomes more clear
  - In “$z \leftarrow x \cdot q$” $x$ has the value $x_0 \land x_2 \land x_3$
  - In “$s \leftarrow w - x’$” $x$ has the value $x_0 \land x_1 \land x_2 \land x_3$
- These values are formed at the merge points in the CFG, as shown in blue
- What names do we write for $x’$?

COMP 506, Spring 2019

SSA is inherently a (global | intraprocedural | full-procedure) analysis
To make the SSA rules work we need new names for values formed at merge points

- SSA introduces $\varnothing$-functions to define the new values created at merge points
  - $\varnothing(x_0, x_1)$ is $x_0$ if control enters along one path and $x_1$ if control enters along the other
  - SSA inserts a $\varnothing$-function for $x$ at each merge where a new value is created
- Now, each name is defined by exactly one statement and each use refers to exactly one name

$\Rightarrow$ Static Single-Assignment Form
Static Single-Assignment Form

SSA Form
- Each name is defined by one statement
- Each use refers to exactly one name

What’s hard & what’s easy
- 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 of values
- Rename all values for uniqueness

A $\phi$-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.

No machine implements a $\phi$-function directly in hardware.
SSA Construction Algorithm (High-level sketch)

Conceptually, the algorithm is simple

- Insert $\phi$-functions where distinct values come together
- Rename values to maintain the principle of static single assignment

... that’s all ...

... of course, there is some bookkeeping to be done ...
SSA Construction Algorithm (The naïve algorithm)

1. Insert $\varnothing$-functions at every CFG join point\(^1\) for every name
2. Solve a data-flow problem to connect definitions & uses
3. Rename each use to the definition that reaches it\(^2\)

What is wrong with this approach?

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

To do better, we need a much more complex approach

---

\(^1\) Values only *merge*, *meet*, or “*come together*” at join points in the CFG.

\(^2\) The $\varnothing$-functions guarantee that only one definition reaches each use. If two values reach a use, they must have flowed through a join point in the CFG, where we inserted a $\varnothing$-function.
The naïve algorithm inserts a Ø-function for $x$ at each join point
- In this example, that is correct and necessary
- In general, that is excessive

The naïve algorithm produces
- Correct SSA form
- More Ø-functions than any other known SSA construction algorithm

The rest is optimization (!)

The naïve algorithm inserts a Ø-function for $x$ at each join point, regardless of whether or not different values meet at that point, or even whether a definition of $x$ reaches that point.
How Can We Execute The Code in SSA Form?

Processors don’t have $\phi$-functions

- To make the code executable, we need to translate out of SSA form
- Naïve idea would be to drop all the subscripts & delete the $\phi$-functions
  - That would work, if all the compiler did was translate into & out of SSA
  - Even after simple compiler optimizations, such as LVN, or any code motion, the naïve scheme can produce incorrect code.

In practice, the translation is a bit more complex than that.

See § 9.3.5 in EaC2e

COMP 506, Spring 2019
Static Single-Assignment Form

Why go to all this trouble? Why build SSA?

• In SVN and DVNT, we needed the “right” name space
  ♦ SSA provides that name space

• In many cases, SSA leads to better analysis
  ♦ Either faster analysis, once SSA is built, or more precise analysis
    → We can interpret the unique names as creating a graph that connects each use to the definition of the value. That graph is sparse relative to classic data-flow analysis
  ♦ Constant propagation can be reformulated from definition to use
  ♦ Even LVN was improved by the SSA name space

• In many cases, SSA leads to cleaner, simpler, faster algorithms
  ♦ Examples include constant propagation, dead code elimination & operator strength reduction

Some modern compilers, such as LLVM/CLANG, use SSA as their primary IR
Static Single-Assignment Form

Is there some more general point here?

• The SSA work popularized the notion that the name space for translation and optimization is critically important

• Compilers had always changed and moved operations, but had not changed the name space in systematic ways

The name space matters

(a lot)

• Programmers don’t find single-assignment languages all that compelling
  ♦ They have been proposed and built. They have not taken over the world.

• Compilers can convert code into SSA form automatically.
  ♦ Reap the benefits of a single-assignment name space for analysis
  ♦ Leave the base language free from the mathematical artifice of the $\phi$-function.

Building SSA lets the compiler have the advantages of a single-assignment name space without burdening the programmer.
Conceptually, the algorithm is simple

1. Insert $\varnothing$-functions where distinct values come together
2. Rename values to maintain the principle of static single assignment

... that’s all ... 

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

1. Insert $\phi$-functions
   a. Calculate dominance frontier
   b. Find global names
      For each name, build a list of blocks that define it
   c. Using dominance frontiers, insert $\phi$-functions for each global name

- $\forall$ global name $n$
- $\forall$ block $b$ in which $n$ is assigned
  - $\forall$ block $d$ in $b$'s dominance frontier
    - insert a $\phi$-function for $n$ in $d$
    - add $d$ to $n$'s list of defining blocks

Use a checklist to avoid putting blocks on the worklist twice;
Use another checklist to avoid inserting the same $\phi$-function twice.

Algorithm, except for 1.b, is from CFRWZ [110]. See § 9.3.3 in EaC2e.
2. Rename variables in a pre-order walk over the dominator tree
   (use an array of stacks, one stack per global name)

   Starting with the root block, \( b \)
   a. Generate unique names for the result of each \( \phi \)-function
   b. Rewrite each operation in the block
      i. Rewrite uses of global names with the current version number (from its stack)
      ii. Rewrite definition by incrementing the version number & pushing it on the stack
   c. Fill in \( \phi \)-function parameters for successor blocks of \( b \)
   d. Recurse on \( b \)’s children in the dominator tree
   e. \(<\text{on exit from block } b >\) pop the names generated in \( b \) from the stacks

Algorithm, except for 1.b, is from CFRWZ [110]. See § 9.3.4 in EaC2e.
SSA Construction *(semi-pruned SSA)*

1. Insert $\varnothing$-functions
   
   a. Calculate dominance frontier ⇐ focus on this step
   
   b. Find global names
      
      For each name, build a list of blocks that define it
   
   c. Using dominance frontiers, insert $\varnothing$-functions for global names

\[
\forall \text{ global name } n
\]
\[
\forall \text{ block } b \text{ in which } n \text{ is assigned}
\]
\[
\forall \text{ block } d \text{ in } b\text{’s dominance frontier}
\]

   insert a $\varnothing$-function for $n$ in $d$

   add $d$ to $n$’s list of defining blocks
Dominance Frontiers & $\phi$-Function Insertion

Where does an assignment in block $n$ induce a $\phi$-function?

• If $n \text{ DOM } m$, there is no need for a $\phi$-function in $m$
  ♦ The definition in $n$ blocks any previous definition from reach $m$  
  \text{ (Def'n of DOM)}

• If $m$ has multiple predecessors and $n$ dominates one of them, but not all of them, then $m$ needs a $\phi$-function for each name defined in $n$

More formally, $m$ is in the dominance frontier of $n$, written $m \in \text{DF}(n)$, iff

1. $\exists p \in \text{predecessors}(m)$ such that $n \in \text{DOM}(p)$, and
2. $n$ does not \textit{strictly dominate} $m$ \hspace{1cm} (n \notin \text{DOM}(m) — \{ m \})

The dominance frontier precisely describes where to insert $\phi$-functions:

\textit{A definition in block $n$ requires a $\phi$-function in each block in DF($n$)}

“Strict” dominance allow a $\phi$-function at the head of a single-block loop.

We say $x \in \text{STRICT IDOM}(y)$ if $x \in \text{IDOM}(y)$ and $x \neq y$. 

Dominance Frontiers

If \( n \notin \text{DOM}(m) \), \( n \notin \text{DOM}(p) \), & \( n \notin \text{DOM}(q) \), then \( m \) does not matter to \( \text{DF}(n) \)

All of \( m \)'s CFG predecessors are dominated by \( n \)

\[ \Rightarrow n \in \text{DOM}(m) \]
\[ \Rightarrow m \notin \text{DF}(n) \]
Dominance & Dominance Frontiers

Example

Control Flow Graph

Results of iterative solution for DOM

<table>
<thead>
<tr>
<th></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>DOM</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>0,1,3,5</td>
<td>0,1,3,6</td>
<td>0,1,7</td>
</tr>
<tr>
<td>IDOM</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>
Dominance & Dominance Frontiers

Example

Results of iterative solution for DOM

<table>
<thead>
<tr>
<th></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><strong>DOM</strong></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>0,1,3,5</td>
<td>0,1,3,6</td>
<td>0,1,7</td>
</tr>
<tr>
<td><strong>IDOM</strong></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>
Dominance & Dominance Frontiers

Example

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

- A definition at \(n\) forces a \(\phi\)-function at \(m\) iff \(n \notin \text{DOM}(m)\) but \(n \in \text{DOM}(p)\) for some \(p \in \text{preds}(m)\)
- \(\text{DF}(n)\) is fringe just beyond region \(n\) dominates

\[
\begin{array}{c|cccccccc}
\text{DOM} & 0 & 0,1 & 0,1,2 & 0,1,3 & 0,1,3,4 & 0,1,3,5 & 0,1,3,6 & 0,1,3,7 \\
\text{DF (strict)} & \emptyset & 1 & 7 & 7 & 6 & 6 & 7 & 1 \\
\end{array}
\]

- \(\text{DF}(4)\) is \(\{6\}\), so \(\leftarrow\) in \(4\) forces \(\phi\)-function in \(6\)
- \(\leftarrow\) in \(6\) forces \(\phi\)-function in \(\text{DF}(6) = \{7\}\)
- \(\leftarrow\) in \(7\) forces \(\phi\)-function in \(\text{DF}(7) = \{1\}\)
- \(\leftarrow\) in \(1\) forces \(\phi\)-function in \(\text{DF}(1) = \{1\}\)

\((\text{halt} – \text{the } \phi\text{-function is already there })\)

For each assignment, we insert the \(\phi\)-functions
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 (i.e., |preds(x)| > 1)

For each CFG predecessor p of x
Run from p to IDOM(x) in the dominator tree, & add x to DF(n) for each n from p up to but not STRICT IDOM(x)

Consider B₇, which is a join point

- B₂ is a CFG predecessor & not IDOM(B₇)  \( \Rightarrow B₇ \in DF(B₂) \)
- B₁ is B₂’s dom. tree ancestor, but is IDOM(B₇) \( \Rightarrow stop \)
- B₆ is a CFG predecessor & not IDOM(B₇) \( \Rightarrow B₇ \in DF(B₆) \)
- B₃ is B₆’s dom. tree ancestor & not IDOM(B₇) \( \Rightarrow B₇ \in DF(B₃) \)
- B₁ is B₃’s dom. tree ancestor, but is IDOM(B₇) \( \Rightarrow stop \)
Dominance & Dominance Frontiers

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 $x$ ($i.e., |preds(x)| > 1$)

For each CFG predecessor $p$ of $x$

Run from $p$ to $IDOM(x)$ in the dominator tree, & add $x$ to $DF(n)$ for each $n$ from $p$ up to but not $IDOM(x)$

- For some applications (other than building SSA), 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 ideas in dead code elimination