CS 2210:
Static Single Assignment

Jonathan Misurda
jmisurda@cs.pitt.edu

SSA
Static Single Assignment (SSA) was developed by R. Cytron, J. Ferrante, et al. in the 1980s.

Every variable is statically assigned exactly one time in the source code (although that statement may execute many times at runtime).

• That is, there is only one def (definition) of a particular variable.

What about code like:

\[ x := 0 \]
\[ x := x + 1 \]

Convert original variable name to nameversion (\( x \rightarrow x_1, x_2 \)) in different places as it is assigned to:

\[ x_1 := 0 \]
\[ x_2 := x_1 + 1 \]

Multiple Paths
This version-based naming convention is sufficient for straight line code, but what about the case when multiple control flow paths may assign to the same original location?

We introduce a phi-function (\( \phi \)-function) that selects the output based upon the path that was executed.

\[ x := x + 1 \]
\[ x := 0 \]
\[ x := x + 1 \]

\[ x_0 := 0 \]
\[ y_0 := 1 \]

while(x < 100) {
  x := x + 1;
  y := y + x;
}

\[ x_1 := \phi(x_0, x_2) \]
\[ y_1 := \phi(y_0, y_2) \]
\[ x_2 := x_1 + 1 \]
\[ y_2 := y_1 + x_2 \]

if (x_2 < 100) goto loop

next: \[ x_3 := \phi(x_0, x_2) \]
\[ y_3 := \phi(y_0, y_2) \]

Multiple Paths
This version-based naming convention is sufficient for straight line code, but what about the case when multiple control flow paths may assign to the same original location?

We introduce a phi-function (\( \phi \)-function) that selects the output based upon the path that was executed.

\[ x := x + 1 \]
\[ x := 0 \]
\[ x := x + 1 \]

\[ x_0 := 0 \]
\[ y_0 := 1 \]

if (x_0 >= 100) goto next

loop: \[ x_1 := \phi(x_0, x_2) \]
\[ y_1 := \phi(y_0, y_2) \]
\[ x_2 := x_1 + 1 \]
\[ y_2 := y_1 + x_2 \]

if (x_2 < 100) goto loop

next: \[ x_3 := \phi(x_0, x_2) \]
\[ y_3 := \phi(y_0, y_2) \]

Phi Functions

<table>
<thead>
<tr>
<th>Source Code</th>
<th>SSA Form</th>
</tr>
</thead>
</table>
| \[ x := 0; \] \[ y := 1; \] \[ while(x < 100) { \] \[ x := x + 1; \] \[ y := y + x; \] \[ loop: \] \[ if (x_0 >= 100) goto next \] \[ x := \$\langle x_0, x_1 \rangle; \] \[ y := \$\langle y_0, y_1 \rangle; \] \[ x := x + 1; \] \[ y := y + x; \] \[ if (x_2 < 100) goto loop \] \[ next: \] \[ x := \$\langle x_0, x_1 \rangle; \] \[ y := \$\langle y_0, y_1 \rangle; \] |}

Phi Functions

\( \phi \)-functions are not three-address code.

• Need some alternate way to represent the variable number of arguments (one for each control-flow path to the block that assigns the variable).

• Perhaps use an extra data structure to hold the arguments

Where to insert \( \phi \)-functions?

• Insert \( \phi \)-functions for each value at the start of each basic block that has more than one predecessor in the CFG.

• Too naive, but it works
Path-Convergence Criterion

There should be a ϕ-function for variable a at node z of the flow graph exactly when all of the following are true:

1. There is a block x containing a definition of a,
2. There is a block y (with y ≠ x) containing a definition of a,
3. There is a nonempty path Px of edges from x to z,
4. There is a nonempty path Py of edges from y to z,
5. Paths Px and Py do not have any node in common other than z, and
6. The node z does not appear within both Px and Py prior to the end, though it may appear in one or the other.

Iterated Path-Convergence

The start node contains an implicit definition of every variable
- formal parameters
- a ← uninitialized

A ϕ-function also counts as a definition of a, so the path-convergence criterion must be considered as a set of equations to be solved by iteration.

while there are nodes x, y, z satisfying conditions 1–5 and z does not contain a ϕ-function for a do
insert a ← ϕ(a, a, …, a) at node z

where the ϕ-function has as many a arguments as there are predecessors of node z.

Iterated Path-Convergence

The iterated path-convergence algorithm for placing ϕ-functions is not practical
- Must examine every triple of nodes x, y, z
- Every path leading from x and y.

A much more efficient algorithm uses the dominator tree of the control flow graph.

Dominator Algorithm

Input:
N = set of nodes in CFG
r = root of CFG

Output:
Set of Dominator sets for each CFG node

Dominators[r] = {r}
foreach node n ∈ (N – r) 
    Dominators[n] = N 
do 
    changed = false 
    foreach node n ∈ (N – r) 
        T = N 
        foreach node p in Predecessors(n) { 
            T = T ∩ Dominators[p] 
        } 
        D = T ∪ n 
        if D ≠ Dominators[n] 
            changed = true 
            Dominators[n] = D 
    until(!changed)

Sample CFG

long evenSum=0;
int i=0;
while(i<100000) {
    if(i%2 == 0){
        evenSum+=i;
    }
    i++;
}
return;

evenSum+=i;
if(i%2 == 0)
i++;
Dominators

The root dominates itself.
Dom(1) = {1}
Dom(2) = {1, 2}
Dom(3) = {1, 2, 3}
Dom(4) = {1, 2, 3, 4}
Not all paths to 5 go through 4, so:
Dom(5) = {1, 2, 3, 5}
Dom(6) = {1, 2, 6}

Strict & Immediate Dominance

a strictly dominates b if
1. a dom b and
2. a ≠ b.

For a ≠ b, a immediately dominates b if and only if:
1. a dom b, and
2. there does not exist a node c such that:
   a. c ≠ a and c ≠ b
   b. a dom c and c dom b.

Thus, a idom b means that the closest dominator of b to the root (travelling backwards from b along the reverse control flow edges) is a.

The immediate dominator of a node is unique.

Immediate Dominator Algorithm

Input:
N = set of nodes in CFG
Dominators[n] = Dominators of n
r = root of CFG
Output:
Immediate dominator for each CFG node

temp = {}
foreach node n ∈ N
  temp[n] = Dominators[n] - {n}

foreach node n ∈ (N - {r})
  foreach node s ∈ temp[n]
    foreach node t ∈ (temp[n] - {s})
      if (t ∈ temp[s])
        temp[n] -= {t}

foreach node n ∈ (N - {r})
  idom[n] = temp[n]

Dominator Tree

The immediate dominance relation forms a tree of the nodes of a flowgraph where:
1. The root is the entry node,
2. The edges are immediate dominances, and
3. The paths display all of the dominance relationships.

Dominance Frontier

The dominance frontier of a node a is the set of all nodes s such that a dominates a predecessor of s, but does not strictly dominate s.

It is the “border” between dominated and undominated nodes.
Dominance Frontier

A definition in node \( n \) forces a \( \phi \)-function at join points that lie just outside the region of the CFG that \( n \) dominates.

A definition in node \( n \) forces a corresponding \( \phi \)-function at any join point \( m \) where:
1. \( n \) dominates a predecessor of \( m \) (i.e., \( \text{preds}(m) \) and \( n \in \text{Dom}(q) \)), and
2. \( n \) does not strictly dominate \( m \).

(Using strict dominance rather than dominance allows a \( \phi \)-function at the start of a single-block loop. In that case, \( n=m \), and \( m \not\in \text{Dom}(n) - \{n\} \).

We call the collection of nodes \( m \) that have this property with respect to \( n \) the dominance frontier of \( n \), denoted \( \text{DF}(n) \).

Dominance Frontier Criterion

Whenever node \( x \) contains a definition of some variable \( a \), then any node \( z \) in the dominance frontier of \( x \) needs a \( \phi \)-function for \( a \).

Since a \( \phi \)-function itself is a definition, we must iterate the dominance-frontier criterion until there are no nodes that need \( \phi \)-functions.

The iterated dominance frontier criterion and the iterated path convergence criterion specify exactly the same set of nodes at which to put \( \phi \)-functions.

Computing the Dominance Frontier

Alternative Algorithm:

```
foreach node \( n \) in the CFG
   \( \text{DF}(n) = \{\} \)

foreach node \( n \) in the CFG
   if \( n \) has multiple predecessors
      foreach predecessor \( p \) of \( n \)
         runner = \( p \)
         while (\( \text{runner} \neq \text{IDom}(n) \))
            \( \text{DF}(\text{runner}) = \text{DF}(\text{runner}) \cup \{n\} \)
            runner = \( \text{IDom}(\text{runner}) \)
```

Dominance Frontier

- Block \( n=1 \): Has 1 predecessor
  - \( \text{DF}(1) = \{\} \)
- Block \( n=2 \): Has multiple predecessors (1,5)
  - Runner = 1
  - \( \text{IDom}(2) = 3 \) # Done with 1
  - Runner = 5
  - \( \text{IDom}(2) = 3 \)
  - \( \text{DF}(5) = \{\} \cup \{2\} = \{2\} \)
  - Runner = \( \text{IDom}(5) \) = 3 # Done with 4
- Block \( n=3 \): Has 1 predecessor
  - \( \text{DF}(3) = \{2\} \)
- Block \( n=4 \): Has 1 predecessor
  - \( \text{DF}(4) = \{\} \)
- Block \( n=5 \): Has 2 predecessors (3,4)
  - Runner = 3
  - \( \text{IDom}(5) = 3 \) # Done with 3
  - Runner = 4
  - \( \text{IDom}(5) = 3 \)
  - \( \text{DF}(4) = \{\} \cup \{5\} \)
  - Runner = \( \text{IDom}(4) \) = 3
  - \( \text{IDom}(5) = 3 \) # Done with 4
- Block \( n=6 \): Has 1 predecessor
  - \( \text{DF}(6) = \{\} \)

Inserting \( \phi \)-Functions

Starting with a program not in SSA form, we need to insert just enough \( \phi \)-functions to satisfy the iterated dominance frontier criterion.

Start with a set \( V \) of variables, a graph \( G \) of control-flow nodes, and for each node \( n \) a set \( \text{A}_n[a] \) of variables defined in node \( n \).

Compute \( \text{A}_n[a] \), the set of nodes that must have \( \phi \)-functions for variable \( a \).

Use a work list \( W \) of nodes that might violate the dominance-frontier criterion.
Placing Phi Functions

\[ \text{Place-\(\phi\)-Functions = } \]
\[ \text{foreach node } n \]
\[ \text{foreach variable } a \text{ in } \text{defsites}(n) \]
\[ \text{defsites}(a) = \text{defsites}(a) \cup \{n\} \]
\[ \text{foreach variable } a \]
\[ \text{W = \text{defsites}(a)} \]
\[ \text{while W not empty} \]
\[ \text{remove some node } n \text{ from } W \]
\[ \text{foreach } y \text{ in } \text{DF}(n) \]
\[ \text{if( } a \notin \text{A}(\phi[y]) \text{)} \]
\[ \text{insert the statement } a \leftarrow \phi(a,a,\ldots,a) \text{ at the top of block } y, \text{ where the } \phi-\text{function has as many arguments as } y \text{ has predecessors} \]
\[ \text{A}(\phi[y]) = \text{A}(\phi[\bar{y}]) \cup \{a\} \]
\[ \text{W = W \cup \{y\}} \]

Inserting \(\phi\)-Functions

\[ V = \text{(evenSum, i)} \]
\[ \text{Def(evenSum) = (1, 4)} \]
\[ \text{Def(i) = (1, 5)} \]

For evenSum:
\[ W = \{1, 4\} \]
\[ n = 1 \]
\[ \text{DF}(1) = \{4\} \text{ #Done} \]
\[ n = 4 \]
\[ \text{DF}(4) = \{5\} \]
\[ \text{Insert } \phi \text{ into block 5 for evenSum} \]

For i:
\[ W = \{1, 5\} \]
\[ n = 1 \]
\[ \text{DF}(1) = \{} \text{ #Done} \]
\[ n = 5 \]
\[ \text{DF}(5) = \{2\} \]
\[ \text{Insert } \phi \text{ into block 2 for i} \]

Renaming the Variables

After the \(\phi\)-functions are placed, we can walk the dominator tree, renaming the different definitions (including \(\phi\)-functions) of variable \(a\) to \(a_1, a_2, a_3, \ldots\), and so on.

In a straight-line program, we would rename all the definitions of \(a\), and then each use of \(a\) is renamed to use the most recent definition of \(a\).

For a program with control-flow branches and joins whose graph satisfies the dominance-frontier criterion, we rename each use of \(a\) to use the closest definition \(d\) of \(a\) that is above \(a\) in the dominator tree.

Renaming Variables (I)

\[ \text{Initialization:} \]
\[ \text{foreach variable } a \]
\[ \text{Count}[a] = 0 \]
\[ \text{Stack}[a] = \text{empty} \]
\[ \text{push 0 onto Stack}[a] \]

\[ \text{Rename}(n) = \]
\[ \text{foreach statement } S \text{ in block } n \]
\[ \text{if } S \text{ is not a } \phi-\text{function} \]
\[ \text{foreach use of some variable } x \text{ in } S \]
\[ i \leftarrow \text{top(} \text{Stack}[a] \text{)} \]
\[ \text{replace the use of } x \text{ with } a_i \text{ in } S \]
\[ \text{foreach definition of some variable } a \text{ in } S \]
\[ \text{Count}[a] = \text{Count}[a] + 1 \]
\[ i \leftarrow \text{Count}[a] \]
\[ \text{push } i \text{ onto Stack}[a] \]
\[ \text{replace definition of } a \text{ with definition of } a_i \text{ in } S \]

Renaming Variables (II)

\[ \text{foreach successor } Y \text{ of block } n \]
\[ \text{Suppose } n \text{ is the } j^{th} \text{ predecessor of } Y \]
\[ \text{foreach } \phi-\text{function in } Y \]
\[ \text{suppose the } j^{th} \text{ operand of the } \phi-\text{function is } a \]
\[ i \leftarrow \text{top(} \text{Stack}[a] \text{)} \]
\[ \text{replace the } j^{th} \text{ operand with } a_i \]
\[ \text{foreach child } X \text{ of } n \text{ in the dominator tree} \]
\[ \text{Rename}(X) \]
\[ \text{foreach statement } S \text{ in block } n \]
\[ \text{foreach definition of some variable } a \text{ in } S \]
\[ \text{pop Stack}[a] \]
Numbering

Just done for variable i in this example

<table>
<thead>
<tr>
<th>Numbering</th>
<th>Speed of SSA Conversion</th>
</tr>
</thead>
<tbody>
<tr>
<td>evenSum+=i;</td>
<td>Speed of SSA Conversion</td>
</tr>
<tr>
<td>if(i%2 == 0)</td>
<td>The DF computation does work proportional to the size (number of edges) of the original graph, plus the size of the dominance frontiers it computes. In practice, this is usually linear in the size of the graph.</td>
</tr>
<tr>
<td>evenSum = φ(evenSum, evenSum)</td>
<td>The DF computation does work proportional to the size (number of edges) of the original graph, plus the size of the dominance frontiers it computes. In practice, this is usually linear in the size of the graph.</td>
</tr>
<tr>
<td>i++;</td>
<td>The DF computation does work proportional to the size (number of edges) of the original graph, plus the size of the dominance frontiers it computes. In practice, this is usually linear in the size of the graph.</td>
</tr>
<tr>
<td>i = φ(i, i);</td>
<td>The DF computation does work proportional to the size (number of edges) of the original graph, plus the size of the dominance frontiers it computes. In practice, this is usually linear in the size of the graph.</td>
</tr>
<tr>
<td>while(i&lt;1000000)</td>
<td>Speed of SSA Conversion</td>
</tr>
<tr>
<td>return;</td>
<td>Speed of SSA Conversion</td>
</tr>
<tr>
<td>evenSum = 0;</td>
<td>Speed of SSA Conversion</td>
</tr>
<tr>
<td>i1 = 0;</td>
<td>Speed of SSA Conversion</td>
</tr>
</tbody>
</table>

Numbering

Just done for variable i in this example

<table>
<thead>
<tr>
<th>Numbering</th>
<th>Speed of SSA Conversion</th>
</tr>
</thead>
<tbody>
<tr>
<td>evenSum+=i2;</td>
<td>Speed of SSA Conversion</td>
</tr>
<tr>
<td>if(i2%2 == 0)</td>
<td>Speed of SSA Conversion</td>
</tr>
<tr>
<td>evenSum = φ(evenSum, evenSum)</td>
<td>Speed of SSA Conversion</td>
</tr>
<tr>
<td>i3 = i2 + 1;</td>
<td>Speed of SSA Conversion</td>
</tr>
<tr>
<td>i2 = φ(i1, i3);</td>
<td>Speed of SSA Conversion</td>
</tr>
<tr>
<td>while(i2&lt;1000000)</td>
<td>Speed of SSA Conversion</td>
</tr>
<tr>
<td>return;</td>
<td>Speed of SSA Conversion</td>
</tr>
<tr>
<td>evenSum1 = 0;</td>
<td>Speed of SSA Conversion</td>
</tr>
<tr>
<td>i1 = 0;</td>
<td>Speed of SSA Conversion</td>
</tr>
</tbody>
</table>

Numbering

Just done for variable i in this example

<table>
<thead>
<tr>
<th>Numbering</th>
<th>Speed of SSA Conversion</th>
</tr>
</thead>
<tbody>
<tr>
<td>rename(1)</td>
<td>Speed of SSA Conversion</td>
</tr>
<tr>
<td>for each successor of block 1: {2}</td>
<td>Speed of SSA Conversion</td>
</tr>
<tr>
<td>for each successor in second statement</td>
<td>Speed of SSA Conversion</td>
</tr>
<tr>
<td>for each use of variable x, use the top of the stack’s number</td>
<td>Speed of SSA Conversion</td>
</tr>
<tr>
<td>for each statement, if it’s not a φ-function, replace the corresponding parameter of the φ-function with the current subscripted version</td>
<td>Speed of SSA Conversion</td>
</tr>
<tr>
<td>Rename(1)</td>
<td>Speed of SSA Conversion</td>
</tr>
<tr>
<td>Rename(2)</td>
<td>Speed of SSA Conversion</td>
</tr>
<tr>
<td>Rename(3)</td>
<td>Speed of SSA Conversion</td>
</tr>
<tr>
<td>Rename(4)</td>
<td>Speed of SSA Conversion</td>
</tr>
<tr>
<td>Rename(5)</td>
<td>Speed of SSA Conversion</td>
</tr>
</tbody>
</table>

Speed of SSA Conversion

The DF computation does work proportional to the size (number of edges) of the original graph, plus the size of the dominance frontiers it computes. In practice, this is usually linear in the size of the graph.

The placing of phi functions algorithm does a constant amount of work for
1. each node and edge in the CFG,
2. each statement in the program,
3. each element of every dominance frontier, and
4. each inserted phi-function.

For a program of size N:
• the amounts (1) and (2) are proportional to N,
• (3) is usually approximately linear in N
• (4) could be N^2 in the worst case, but empirical measurement has shown that it is usually proportional to N.
Converting out of SSA

After program transformations and optimization, a program in SSA form must be translated into some executable representation without $\phi$-functions.

The definition $y \leftarrow \phi(x_1, x_2, x_3)$ can be translated as:
- move $y \leftarrow x_1$ if arriving along predecessor edge 1,
- move $y \leftarrow x_2$ if arriving along predecessor edge 2, and
- move $y \leftarrow x_3$ if arriving along predecessor edge 3.

It is tempting simply to assign $x_1$ and $x_2$ the same register if they were derived from the same variable $x$. However, transformations on SSA form may make live ranges interfere.

Instead, we rely on coalescing in the register allocator to eliminate almost all of the move instructions.