Compiler Construction 2009/2010
SSA—Static Single Assignment Form

Peter Thiemann

March 15, 2010
Outline

1. Static Single-Assignment Form
2. Converting to SSA Form
3. Optimization Algorithms Using SSA
4. Dependencies
5. Converting Back from SSA Form
Important data structure: **def-use chain**
links definitions and uses to flow-graph nodes

**Improvement: SSA form**
- Intermediate representation
- Each variable has exactly one (static) definition
Usefulness of SSA Form

- Dataflow analysis becomes simpler
- Optimized space usage for def-use chains
  \( N \) uses and \( M \) definitions of var: \( N \cdot M \) pointers required
- Uses and defs are related to dominator tree
- Unrelated uses of the same variable are made different
SSA Example

straight-line program

\[
\begin{align*}
a & \leftarrow x + y \\
b & \leftarrow a - 1 \\
a & \leftarrow y + b \\
b & \leftarrow x \cdot 4 \\
a & \leftarrow a + b
\end{align*}
\]

program in SSA form

\[
\begin{align*}
a_1 & \leftarrow x + y \\
b_1 & \leftarrow a_1 - 1 \\
a_2 & \leftarrow y + b_1 \\
b_2 & \leftarrow x \cdot 4 \\
a_3 & \leftarrow a_2 + b_2
\end{align*}
\]
\( \phi \)-Functions

CFG with a control-flow join

\[
\begin{align*}
  b & \leftarrow M[x] \\
  a & \leftarrow 0
\end{align*}
\]

\[
\text{if } b < 4
\]

\[
\begin{align*}
  a & \leftarrow b \\
  c & \leftarrow a + b
\end{align*}
\]
\( b_1 \leftarrow M[x_0] \)
\( a_1 \leftarrow 0 \)

\[ \text{if } b_1 < 4 \]

\( a_2 \leftarrow b_1 \)

\( a_3 \leftarrow \phi(a_2, a_1) \)
\( c_1 \leftarrow a_3 + b_1 \)
\[ b_1 \leftarrow M[x_0] \]
\[ a_1 \leftarrow 0 \]
\[ \text{if } b_1 < 4 \]
\[ a_2 \leftarrow b_1 \]
\[ a_3 \leftarrow \phi(a_2, a_1) \]
Program with a loop

- $a \leftarrow 0$
- $b \leftarrow a + 1$
- $c \leftarrow c + b$
- $a \leftarrow b \cdot 2$
- if $a < N$
- return $c$
\( \phi \)-Functions

… transformed to edge-split SSA form

\[
\begin{align*}
a_3 & \leftarrow \phi(a_1, a_2) \\
b_1 & \leftarrow \phi(b_0, b_2) \\
c_2 & \leftarrow \phi(c_0, c_1) \\
b_2 & \leftarrow a_3 + 1 \\
c_1 & \leftarrow c_2 + b_2 \\
a_2 & \leftarrow b_2 \cdot 2 \\
\text{if } a_2 & < N
\end{align*}
\]

\[
\begin{align*}
a_1 & \leftarrow 0 \\
return c
\end{align*}
\]
Features of SSA Form

- SSA renames variables
- SSA introduces $\phi$-functions
  - not “real” functions, just notation
  - implemented by move instruction on incoming edges
  - can often be ignored by optimization
Converting to SSA Form

- Program $\rightarrow$ CFG
- Insert $\phi$-functions
  - could add a $\phi$-function for each variable at each join point
- Rename variables
- Perform edge splitting
Inserting $\phi$-functions

The Path-Convergence Criterion

Add a $\phi$-function for variable $a$ at node $z$ of the flow graph iff

1. There is a block $x$ containing a definition of $a$.
2. There is a block $y \neq x$ containing a definition of $a$.
3. There is a non-empty path $\pi_{xz}$ from $x$ to $z$.
4. There is a non-empty path $\pi_{yz}$ from $y$ to $z$.
5. Paths $\pi_{xz}$ and $\pi_{yz}$ have only $z$ in common.
6. Node $z$ does not appear in both $\pi_{xz}$ and $\pi_{yz}$ prior to the end, but it may appear before in one of them.
Remarks
- Start node contains an implicit definition of each variable
- A $\phi$-function counts as a definition
- Compute by fixpoint iteration

Algorithm

while there are nodes $x$, $y$, $z$ satisfying conditions 1–5 and $z$ does not contain a $\phi$-function for $a$
do insert $a \leftarrow \phi(a_1, \ldots, a_p)$
where $p =$ number of predecessors of $z$
Dominance Property of SSA Form

In SSA, each definition dominates all its uses

1. If $x$ is the $i$th argument of a $\phi$-function in block $n$, then the definition of $x$ dominates the $i$th predecessor of node $n$.

2. If $x$ is used in a non-$\phi$ statement in block $n$, then the definition of $x$ dominates node $n$. 
The Dominance Frontier
A more efficient algorithm for placing $\phi$-functions

**Conventions**
- $x$ strictly dominates $y$ if $x$ dominates $y$ and $x \neq y$.
- Successor and predecessor for graph edges.
- Parent and child for dominance tree edges, ancestor for paths.
- The dominance frontier of a node $x$ is the set of all nodes $w$ such that $x$ dominates a predecessor of $w$, but does not strictly dominate $w$.

**Dominance Frontier Criterion**
If node $x$ contains a definition for some variable $a$, then any node $z$ in the dominance frontier of $x$ needs a $\phi$-function for $a$. 
Domiance Frontier

Consider node 5
The dominance frontier criterion must be iterated: each inserted $\phi$-function counts as a new definition.

**Theorem**

The iterated dominance frontier criterion and the iterated path-convergence criterion specify the same set of nodes for placing $\phi$-functions.
Computing the Dominance Frontier

- $DF[n]$, the dominance frontier of $n$, can be computed in one pass through the dominator tree.
- $DF_{local}[n]$ successors of $n$ not strictly dominated by $n$.
  \[ DF_{local}[n] = \{ y \in succ[n] \mid idom(y) \neq n \} \]
- $DF_{up}[n, c]$ nodes in the dominance frontier of $c$ that are not strictly dominated by $c$’s immediate dominator $n$.
  \[ DF_{up}[n, c] = \{ y \in DF[c] \mid idom(y) \neq n \} \]
- It holds that
  \[ DF[n] = DF_{local}[n] \cup \bigcup_{c \in children[n]} DF_{up}[n, c] \]
Computing the Dominance Frontier

Algorithm

\[ \text{computeDF}[n] = \]
\[ S \leftarrow \emptyset \]
\[ \text{for each node } y \in \text{succ}[n] \text{ do } \{ \text{compute } DF_{\text{local}}(n) \} \]
\[ \text{if } \text{idom}(y) \neq n \text{ then} \]
\[ S \leftarrow S \cup \{ y \} \]
\[ \text{for each child } c \text{ with } \text{idom}(c) = n \text{ do } \{ \text{compute } DF_{\text{up}}(n, c) \} \]
\[ \text{computeDF}[c] \]
\[ \text{for each } w \in DF[c] \text{ do} \]
\[ \text{if } n = w \text{ or } n \text{ does not dominate } w \text{ then} \]
\[ S \leftarrow S \cup \{ w \} \]
Inserting $\phi$-Functions

Place-$\phi$-Functions =

for each node $n$ do

for each variable $a \in A_{\text{orig}}[n]$ do

$\text{defsites}[a] \leftarrow \text{defsites}[a] \cup \{n\}$

for each variable $a$ do

$W \leftarrow \text{defsites}[a]$

while $W \neq \emptyset$ do

remove some node $n$ from $W$

for each $y \in DF[n]$ do

if $a \not\in A_{\phi}[y]$ then

insert statement $a \leftarrow \phi(a, \ldots, a)$ at top of block $y$,
where the number of arguments is $|\text{pred}[y]|$

$A_{\phi}[y] \leftarrow A_{\phi}[y] \cup \{a\}$

if $a \not\in A_{\text{orig}}[y]$ then

$W \leftarrow W \cup \{y\}$
Renaming Variables

- Top-down traversal of the dominator tree
- Rename the different definitions (including $\phi$) of variable $a$ to $a_1, a_2, \ldots$
- Rename each use of $a$ in a statement to the closest definition of an $a$ that is above $a$ in the dominator tree
- For $\phi$-functions look ahead in the successor nodes
Some analyses and transformations are simpler if no control flow edge leads from a node with multiple successors to one with multiple predecessors.

Edge splitting achieves the unique successor or predecessor property.

If there is a control-flow edge $a \rightarrow b$ where $|\text{succ}[a]| > 1$ and $|\text{pred}[b]| > 1$, then create new, empty node $z$ and replace edge $a \rightarrow b$ by $a \rightarrow z$ and $z \rightarrow b$. 
There are efficient, almost linear-time algorithms for computing the dominator tree [Lengauer, Tarjan 1979] [Harel 1985] [Buchsbaum 1998] [Alstrup 1999].

But there are easy variations of the naive algorithm that perform better in practice. [Cooper, Harvey, Kennedy 2006]
Optimization Algorithm Using SSA
Representation of SSA Form

**Statement**  assignment, \( \phi \)-function, fetch, store, branch.
Fields: containing block, previous/next statement in block, variables defined, variables used

**Variable**  definition site, list of use sites

**Block**  list of statements, ordered list of predecessors, one or more successors
SSA: Dead-Code Elimination

SSA Liveness
A variable definition is live iff its list of uses is non-empty.

Algorithm

\[ W \leftarrow \text{list of all variables in SSA program} \]

\[ \textbf{while } W \neq \emptyset \textbf{ do} \]

remove some variable \( v \) from \( W \)

\[ \textbf{if } v \text{'s list of uses is empty } \textbf{then} \]

let \( S \) be \( v \text{'s defining statement} \)

\[ \textbf{if } S \text{ has no side effects other than the assignment to } v \textbf{ then} \]

delete \( S \) from program

\[ \textbf{for each variable } x_i \text{ used by } S \textbf{ do} \]

delete \( S \) from list of uses of \( x_i \) \{in constant time\}

\[ W \leftarrow W \cup \{x_i\} \]
SSA: Simple Constant Propagation

- If \( v \) is defined by \( v \leftarrow c \) (a constant) then each use of \( v \) can be replaced by \( c \).
- The \( \phi \)-function \( v \leftarrow \phi(c, \ldots, c) \) can be replaced by \( v \leftarrow c \).

**Algorithm**

\[
W \leftarrow \text{list of all statements in SSA program}
\]

\[
\textbf{while } W \neq \emptyset \textbf{ do}
\]

\[
\begin{align*}
&\text{remove some statement } S \text{ from } W \\
&\textbf{if } S \text{ is } v \leftarrow \phi(c, \ldots, c) \text{ for constant } c \text{ then} \\
&\quad \text{replace } S \text{ by } v \leftarrow c \\
&\textbf{if } S \text{ is } v \leftarrow c \text{ for constant } c \text{ then} \\
&\quad \text{delete } S \\
&\text{for each statement } T \text{ that uses } v \text{ do} \\
&\quad \text{substitute } c \text{ for } v \text{ in } T \\
&W \leftarrow W \cup \{T\}
\end{align*}
\]
SSA: Further Linear-Time Transformations

Copy propagation
If some $S$ is $x \leftarrow \phi(y)$ or $x \leftarrow y$, then remove $S$ and substitute $y$ for every use of $x$.

Constant folding
If $S$ is $v \leftarrow c \oplus d$ where $c$ and $d$ are constants, then compute $e = c \oplus d$ at compile time and replace $S$ by $b \leftarrow e$. 
SSA: Further Linear-Time Transformations

**Constant conditions**

Let \( \textbf{if } a \neq b \textbf{ goto } L_1 \textbf{ else } L_2 \) be at the end of block \( L \) with \( a \) and \( b \) constants and \( \neq \) a comparison operator.

- Replace the conditional branch by \textbf{goto } L_1 \textbf{ or } \textbf{goto } L_2 depending on the compile-time value of \( a \neq b \)
- Delete the control flow edge \( L \rightarrow L_2 \) (\( L_1 \) respectively)
- Adjust the \( \phi \) functions in \( L_2 \) (\( L_1 \)) by removing the argument associated to predecessor \( L \).

**Unreachable code**

Deleting an edge from a predecessor may cause block \( L_2 \) to become unreachable.

- Delete all statements of \( L_2 \), adjusting the use lists of the variables used in these statements.
- Delete block \( L_2 \) and the edges to its successors.
Conditional Constant Propagation

\[ \begin{align*}
  &i \leftarrow 1 \\
  &j \leftarrow 1 \\
  &k \leftarrow 0 \\
  &\text{if } k < 100 \\
  &\text{if } j < 20 \\
  &j \leftarrow i \\
  &k \leftarrow k + 1 \\
  &\text{return } j \\
  &j \leftarrow k \\
  &k \leftarrow k + 2 \\
  &k \leftarrow k + 1
\end{align*} \]
Conditional Constant Propagation

- does not assume that a block can be executed until there is evidence for it
- does not assume a variable is non-constant until there is evidence for it
### Conditional Constant Propagation

**Data Structures**

**Constant Propagation Lattice**

- $V[v] = \perp$ no assignment to $v$ has been seen (initially)
- $V[v] = c$ an assignment $v \leftarrow c$ (constant) has been seen
- $V[v] = \top$ conflicting assignments have been seen

**Block Reachability**

- $E[B] = false$ no control transfer to $B$ has been seen (initially)
- $E[B] = true$ a control transfer to $B$ has been seen
Least upper bound operation
\[ \perp \sqcup \alpha = \alpha \sqcup \perp = \alpha \]
\[ T \sqcup \alpha = \alpha \sqcup T = T \]
\[ a \sqcup b = \begin{cases} a & a = b \\ \top & a \neq b \end{cases} \]

Primitive operation
\[ \perp \hat{\oplus} \alpha = \alpha \hat{\oplus} \perp = \perp \]
\[ T \hat{\oplus} \alpha = \alpha \hat{\oplus} T = T \]
\[ a \hat{\oplus} b = (a \oplus b) \]
Conditional Constant Propagation
Algorithm Initialization

1. Initialize $V[v] = \bot$ for all variables $v$ and $E[B] = false$ for all blocks $B$
2. If $v$ has no definition, then set $V[v] \leftarrow \top$ (must be input or uninitialized)
3. The entry block is reachable: $E[B_0] \leftarrow true$
For each $B$ with $E[B]$ and $B$ has only one successor $C$, then set $E[C] = true$.

For each reachable assignment $v \leftarrow x \oplus y$ set $V[v] \leftarrow V[x] \oplus V[y]$.

For each reachable assignment $v \leftarrow \phi(x_1, \ldots, x_p)$ set $V[v] \leftarrow \bigcup\{V[x_j] \mid j$th predecessor is reachable$\}$

For each reachable assignment $v \leftarrow M[\ldots]$ or $v \leftarrow CALL(\ldots)$ set $V[v] \leftarrow \top$

For each reachable branch if $x\#y$ goto $L_1$ else $L_2$ consider $\beta = V[x] \oplus V[y]$.

- If $\beta = true$, then set $E[L_1] \leftarrow true$.
- If $\beta = false$, then set $E[L_2] \leftarrow true$.
- If $\beta = \top$, then set $E[L_1], E[L_2] \leftarrow true$. 
Conditional Constant Propagation

Example

\[
\begin{align*}
  \mathit{i}_1 & \leftarrow 1 \\
  \mathit{j}_1 & \leftarrow 1 \\
  \mathit{k}_1 & \leftarrow 0 \\
  \mathit{j}_2 & \leftarrow \phi(\mathit{j}_4, \mathit{j}_1) \\
  \mathit{k}_2 & \leftarrow \phi(\mathit{k}_4, \mathit{k}_1) \\
  & \text{if } \mathit{k}_2 < 100 \\
  & \text{if } \mathit{j}_2 < 20 \\
  \mathit{j}_3 & \leftarrow \mathit{j}_1 \\
  \mathit{k}_3 & \leftarrow \mathit{k}_2 + 1 \\
  \mathit{j}_5 & \leftarrow \mathit{k}_2 \\
  \mathit{k}_5 & \leftarrow \mathit{k}_2 + 2 \\
  \mathit{j}_4 & \leftarrow \phi(\mathit{j}_3, \mathit{j}_5) \\
  \mathit{k}_4 & \leftarrow \phi(\mathit{k}_3, \mathit{k}_5) \\
  & \text{return } \mathit{j}_2
\end{align*}
\]
Conditional Constant Propagation

Example after propagation

\[ k_2 \leftarrow \phi(k_4, 0) \]
\[ \text{if } k_2 < 100 \]
\[ \text{return 1} \]

\[ k_3 \leftarrow k_2 + 1 \]

\[ k_4 \leftarrow \phi(k_3) \]
Conditional Constant Propagation

Example after cleanup

\[ k_3 \leftarrow k_2 + 1 \]

\[ k_2 \leftarrow \phi(k_3, 0) \]

if \( k_2 < 100 \)

return 1
Dependencies Between Statements

B depends on A

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read-after-write</td>
<td>A defines variable v and B uses v</td>
</tr>
<tr>
<td>Write-after-write</td>
<td>A defines variable v and B defines v</td>
</tr>
<tr>
<td>Write-after-read</td>
<td>A uses v and then B defines v</td>
</tr>
<tr>
<td>Control</td>
<td>A controls whether B executes</td>
</tr>
</tbody>
</table>

In SSA form

- all dependencies are Read-after-write or Control
- Read-after-write is evident from SSA graph
- Control needs to be analyzed
Memory Dependence

- Memory does not enjoy the single assignment property
- Consider

\[ M[i] \leftarrow 4 \]
\[ x \leftarrow M[j] \]
\[ M[k] \leftarrow j \]

Depending on the values of \( i, j, \) and \( k \)
- 2 may have a read-after-write dependency with 1 (if \( i = j \))
- 3 may have a write-after-write dependency with 1 (if \( i = k \))
- 3 may have a write-after-read dependency with 2 (if \( j = k \))

so 2 and 3 may not be exchanged

Approach

- No attempt to track memory dependencies
- Store instructions always live
- No attempt to reorder memory instructions
Control Dependence Graph

Control Dependence

- Node $y$ is control dependent on $x$ if
  1. $x$ has successors $u$ and $v$
  2. there exists a path from $u$ to exit that avoids $y$
  3. every path from $v$ to exit goes through $y$

- The control-dependence graph (CDG) has an edge from $x$ to $y$ if $y$ is control dependent on $x$.

- $y$ postdominates $v$ if $y$ is on every path from $v$ to exit, i.e., if $y$ dominates $v$ in the reverse CFG.
Construction of the CDG

Let $G$ be a CFG

1. Add new entry node $r$ to $G$ with edge $r \rightarrow s$ (the original start node) and an edge $r \rightarrow \text{exit}$.

2. Let $G'$ be the reverse control-flow graph with the same nodes as $G$, all edges reversed, and with start node exit.

3. Construct the dominator tree of $G'$ with root exit.

4. Calculate the dominance frontiers $DF_{G'}$ of $G'$.

5. The CDG has edge $x \rightarrow y$ if $x \in DF_{G'}[y]$. 
Use of the CDG

A must be executed before B if there is a path $A \rightarrow B$ using SSA use-def edges and CDG edges. I.e., there are data- and control dependencies that require A to be executed before B.
Construction of the CDG

Example

\[
\begin{align*}
    &i \leftarrow 1 \\
    &j \leftarrow 1 \\
    &k \leftarrow 0
\end{align*}
\]

if \( k < 100 \)

\[
\begin{align*}
    &i \leftarrow 1 \\
    &j \leftarrow 1 \\
    &k \leftarrow 0
\end{align*}
\]

if \( j < 20 \)

\[
\begin{align*}
    &j \leftarrow i \\
    &k \leftarrow k + 1
\end{align*}
\]

return \( j \)

\[
\begin{align*}
    &j \leftarrow k \\
    &k \leftarrow k + 2
\end{align*}
\]
Construction of the CDG

CFG and reverse CFG
Construction of the CDG
Postdominators and CDG
Aggressive Dead-Code Elimination

- Application of the CDG
- Consider

\[ k_2 \leftarrow \phi(k_3, 0) \]
\[ \text{if } k_2 < 100 \]

\[ k_3 \leftarrow k_2 + 1 \]

return 1

- \( k_2 \) is live because it is used in defining \( k_3 \)
- \( k_3 \) is live because it is used in defining \( k_2 \)
Aggressive Dead-Code Elimination

**Algorithm**

Exhaustively mark a live any statement that

1. Performs I/O/, stores into memory, returns from the function, calls another function that may have side effects.
2. Defines some variable $v$ that is used by another live statement.
3. Is a conditional branch, on which some other live statement is control dependent.

Then delete all unmarked statements.

- Result on example: return 1; loop is deleted
Outline

1. Static Single-Assignment Form
2. Converting to SSA Form
3. Optimization Algorithms Using SSA
4. Dependencies
5. Converting Back from SSA Form
Converting Back from SSA Form

- \( \phi \)-functions are not executable and must be replaced to generate code
- \( y \leftarrow \phi(x_1, x_2, x_3) \) is interpreted as
  - move \( x_1 \) to \( y \) if arriving from predecessor \#1
  - move \( x_2 \) to \( y \) if arriving from predecessor \#2
  - move \( x_3 \) to \( y \) if arriving from predecessor \#3
- Insert these instructions at the end of the respective predecessor (possible due to edge-split assumption)
- Next step: register allocation
Liveness Analysis for SSA

LivenessAnalysis() =
   for each variable \( v \) do
      \( M \leftarrow \emptyset \)
      for each statement \( s \) using \( v \) do
         if \( s \) is a \( \phi \)-function with \( i \)th argument \( v \) then
            let \( p \) be the \( i \)th predecessor of \( s \)'s block
            LiveOutAtBlock(\( p, v \))
         else
            LiveInAtStatement(\( s, v \))
      \( \)\( \)\( \)
      LiveOutAtBlock(\( n, v \)) =
      \{ \( v \) is live-out at \( n \)\}
      if \( n \not\in M \) then
         \( M \leftarrow M \cup \{ n \} \)
         let \( s \) be the last statement in \( n \)
         LiveOutAtStatement(\( s, v \))
Liveness Analysis for SSA

LiveInAtStatement(s, v) =
{v is live-in at s}
if s is first statement of block n then
{v is live-in at n}
for each p ∈ pred[n] do
    LiveOutAtBlock(p, v)
else
    let s’ be the statement preceding s
    LiveOutAtStatement(s’, v)

LiveOutAtStatement(s, v) =
{v is live-out at s}
let W be the set of variables defined in s
for each variable w ∈ W \ {v} do
    add (v, w) to interference graph {needed if v defined?}
if v ∉ W then
    LiveInAtStatement(s, v)