P423/P523 Compilers
Single Static Assignment

Based on material from Static Single Assignment Book

Deyaaeldeen Almahallawi\textsuperscript{1}

\textsuperscript{1}dalmahal@indiana.edu
Indiana University

April 23, 2015
History

- First used for efficient computation of dataflow problems such as global value numbering, congruence of variables, aggressive deadcode removal, and constant propagation with conditional branches.
- Currently used by GCC, Suns HotSpot JVM, IBMs RVM, Chromium V8, Mono, and LLVM.
A program is defined to be in SSA form if each variable is a target of exactly one assignment statement in the program text.
Consider the following code:

\begin{verbatim}
x = 1;
y = x + 1;
x = 2;
z = x + 1;
\end{verbatim}
Consider the following code:

\[
\begin{align*}
x & = 1; \\
y & = x + 1; \\
x & = 2; \\
z & = x + 1; \\
\end{align*}
\]

\[
\begin{align*}
x1 & = 1; \\
y & = x1 + 1; \\
x2 & = 2; \\
z & = x2 + 1; \\
\end{align*}
\]
What about this code?

```python
x = input();
if (x == 42)
    y = 1;
else
    y = x + 2;
end
print(y);
```
\begin{verbatim}
x = input();
if (x == 42)
  then
  y1 = 1;
else
  y2 = x + 2;
end
y3 = \phi (y1, y2);
print(y3);
\end{verbatim}
x = 0;
y = 0;

while (x < 10) {
    y = y + x;
    x = x + 1;
}

print(y)
Introduction
x1 = 0;
y1 = 0;

x2 = \phi(x1, x3)
y2 = \phi(y1, y3)
while (x2 < 10){
    y3 = y2 + x2;
x3 = x2 + 1;
}

print(y2)
Properties

- Since there is only a single definition for each variable in the program text, a variables value is independent of its position in the program.
- Almost free use-def chains.
- Simplifies def-use chains.
Properties (Def-use)

\[
x \leftarrow 1 \quad x \leftarrow 2
\]

\[
y \leftarrow x + 1 \quad z \leftarrow x + 2
\]

\[
x_1 \leftarrow 1 \quad x_2 \leftarrow 2
\]

\[
x_3 \leftarrow \phi(x_1, x_2)
\]

\[
y \leftarrow x_3 + 1 \quad z \leftarrow x_3 + 2
\]
Properties

- Since there is only a single definition for each variable in the program text, a variable's value is independent of its position in the program.
- Almost free use-def chains.
- Simplifies def-use chains.
- No program point can be reached by two definitions of the same variable (First phase).
Properties

Single reaching-definition property

A definition $D$ of variable $v$ reaches a point $p$ in the CFG if there exists a path from $D$ to $p$ that does not pass through another definition of $v$. 
Minimal SSA Construction

The minimality of the number of inserted $\phi$-functions.

- $\phi$-function insertion: performs live-range splitting to ensure that any use of a given variable $v$ is reached by exactly one definition of $v$.
- Variable renaming: assigns a unique variable name to each live-range.
Minimal SSA Construction

Minimality property

The minimality of the number of inserted $\phi$-functions.
Minimal SSA Construction

Minimality property

The minimality of the number of inserted $\phi$-functions.

- $\phi$-function insertion:
Minimality property

The minimality of the number of inserted $\phi$-functions.

- $\phi$-function insertion: performs live-range splitting
Minimal SSA Construction

Minimality property

The minimality of the number of inserted $\phi$-functions.

- $\phi$-function insertion: performs live-range splitting to ensure that any use of a given variable $v$ is reached by exactly one definition of $v$. 
Minimal SSA Construction

Minimality property

The minimality of the number of inserted $\phi$-functions.

- $\phi$-function insertion: performs live-range splitting to ensure that any use of a given variable $v$ is reached by exactly one definition of $v$.
- Variable renaming:
Minimal SSA Construction

Minimality property

The minimality of the number of inserted \( \phi \)-functions.

- \( \phi \)-function insertion: performs live-range splitting to ensure that any use of a given variable \( v \) is reached by exactly one definition of \( v \).
- Variable renaming: assigns a unique variable name to each live-range.
Background

- Join sets: For a given set of nodes $S$ in a CFG, the join set $S(S)$ is the set of nodes in $S$ that can be reached by two (or more) distinct elements of $S$ using disjoint paths.
- Dominance: $d \text{ dom } i$ if all paths from entry to node $i$ include $d$
- Strict dominance: $d \text{ sdom } i$ if $d \text{ dom } i$ and $d \neq i$
- Dominance frontier: $DF(n)$ is the border of the CFG region that is dominated by $n$, i.e. it contains all nodes $x$ such that $n$ dominates a predecessor of $x$ but $n$ does not strictly dominate $x$. 

Deyaaaldeen Almahallawi
Background

- **Join sets:** For a given set of nodes $S$ in a CFG, the join set $\mathcal{J}(S)$ is the set of nodes in $S$ that can be reached by two (or more) distinct elements of $S$ using disjoint paths.
Background

- **Join sets**: For a given set of nodes $S$ in a CFG, the join set $\mathcal{J}(S)$ is the set of nodes in $S$ that can be reached by two (or more) distinct elements of $S$ using disjoint paths.
- **Dominance**: $d \text{ dom } i$ if all paths from entry to node $i$ include $d$.
- **Strict dominance**: $d \text{ sdom } i$ if $d \text{ dom } i$ and $d \neq i$.
- **Dominance frontier**: $\text{DF}(n)$ is the border of the CFG region that is dominated by $n$, i.e. it contains all nodes $x$ such that $n$ dominates a predecessor of $x$ but $n$ does not strictly dominate $x$. 
Background

- **Join sets:** For a given set of nodes $S$ in a CFG, the join set $\mathcal{J}(S)$ is the set of nodes in $S$ that can be reached by two (or more) distinct elements of $S$ using disjoint paths.

- **Dominance:** $d \text{ dom } i$ if all paths from entry to node $i$ include $d$

- **Strict dominance:** $d \text{ sdom } i$ if $d \text{ dom } i$ and $d \neq i$
Background

- Join sets: For a given set of nodes $S$ in a CFG, the join set $\mathcal{J}(S)$ is the set of nodes in $S$ that can be reached by two (or more) distinct elements of $S$ using disjoint paths.
- Dominance: $d \text{ dom } i$ if all paths from entry to node $i$ include $d$
- Strict dominance: $d \text{ sdom } i$ if $d \text{ dom } i$ and $d \neq i$
- Dominance frontier: $DF(n)$ is the border of the CFG region that is dominated by $n$, i.e. it contains all nodes $x$ such that $n$ dominates a predecessor of $x$ but $n$ does not strictly dominate $x$. 
Minimal SSA Construction

Background

- Join sets: For a given set of nodes $S$ in a CFG, the join set $\mathcal{J}(S)$ is the set of nodes in $S$ that can be reached by two (or more) distinct elements of $S$ using disjoint paths.
- Dominance: $d \text{ dom } i$ if all paths from entry to node $i$ include $d$
- Strict dominance: $d \text{ sdom } i$ if $d \text{ dom } i$ and $d \neq i$
- Dominance frontier: $DF(n)$ is the border of the CFG region that is dominated by $n$, i.e. it contains all nodes $x$ such that $n$ dominates a predecessor of $x$ but $n$ does not strictly dominate $x$. 
Dominance Frontier

What is the border frontier of $y$ in blocks $B$ and $C$?
Constructing minimal SSA form requires for each variable $v$, the insertion of $\phi$-functions at $I(\text{Defs}(v))$
Constructing minimal SSA form requires for each variable \( \nu \), the insertion of \( \phi \)-functions at \( \mathcal{S}(\text{Defs} (\nu)) \), where \( \text{Defs}(\nu) \) is the set of nodes that have definitions of \( \nu \).
\( \phi \)-function Insertion

\begin{verbatim}
1 \( F \leftarrow \{\}; \quad /* \text{set of basic blocks where } \phi \text{ is added} */
2 \textbf{for } v: \text{variable names in original program do}
3 \quad W \leftarrow \{\}; \quad /* \text{set of basic blocks} */
4 \quad \textbf{for } d \in \text{Defs}(v) \text{ do}
5 \quad \quad \text{let } B \text{ be the basic block containing } d;
6 \quad \quad W \leftarrow W \cup \{B\};
7 \quad \textbf{end}
8 \quad \textbf{while } W \neq \{\} \text{ do}
9 \quad \quad \text{remove a basic block } X \text{ from } W;
10 \quad \quad \textbf{for } Y: \text{basic block } \in \text{DF}(X) \text{ do}
11 \quad \quad \quad \textbf{if } Y \notin F \text{ then}
12 \quad \quad \quad \quad \text{add } v \leftarrow \phi(...) \text{ at entry of } Y;
13 \quad \quad \quad \quad F \leftarrow F \cup \{Y\};
14 \quad \quad \quad \textbf{if } Y \notin \text{Defs}(v) \text{ then}
15 \quad \quad \quad \quad W \leftarrow W \cup \{Y\};
16 \quad \quad \quad \textbf{end}
17 \quad \quad \textbf{end}
18 \quad \textbf{end}
19 \textbf{end}
\end{verbatim}
Because a $\phi$-function is itself a definition,
Notes

- Because a $\phi$-function is itself a definition, it may require further $\phi$-functions to be inserted.
Because a $\phi$-function is itself a definition, it may require further $\phi$-functions to be inserted.

Dominance frontiers of distinct nodes may intersect,
Notes

- Because a $\phi$-function is itself a definition, it may require further $\phi$-functions to be inserted.
- Dominance frontiers of distinct nodes may intersect, but once a $\phi$-function for a particular variable has been inserted at a node, there is no need to insert another.
Example: \( x \)

\[
\begin{array}{cccc}
\text{while loop} & X & \text{DF}(X) & F \\
\hline
& - & - & - & \{\} & \{B, C, D\}
\end{array}
\]
<table>
<thead>
<tr>
<th>while loop #</th>
<th>$X$</th>
<th>DF($X$)</th>
<th>$F$</th>
<th>$W$</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>$\emptyset$</td>
<td>${B, C, D}$</td>
</tr>
<tr>
<td>1</td>
<td>$B$</td>
<td>${D}$</td>
<td>${D}$</td>
<td>${C, D}$</td>
</tr>
<tr>
<td>while loop #</td>
<td>X</td>
<td>DF(X)</td>
<td>F</td>
<td>W</td>
</tr>
<tr>
<td>--------------</td>
<td>----</td>
<td>-------</td>
<td>-----</td>
<td>-------</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>{}</td>
<td>{B, C, D}</td>
</tr>
<tr>
<td>1</td>
<td>B</td>
<td>{D}</td>
<td>{D}</td>
<td>{C, D}</td>
</tr>
<tr>
<td>2</td>
<td>C</td>
<td>{D, E}</td>
<td>{D, E}</td>
<td>{D, E}</td>
</tr>
<tr>
<td>while loop #</td>
<td>X</td>
<td>DF(X)</td>
<td>F</td>
<td>W</td>
</tr>
<tr>
<td>-------------</td>
<td>----</td>
<td>-------</td>
<td>-----</td>
<td>------</td>
</tr>
<tr>
<td>1</td>
<td>B</td>
<td>{D}</td>
<td>{D}</td>
<td>{C, D}</td>
</tr>
<tr>
<td>2</td>
<td>C</td>
<td>{D, E}</td>
<td>{D, E}</td>
<td>{D, E}</td>
</tr>
<tr>
<td>3</td>
<td>D</td>
<td>{E, A}</td>
<td>{D, E, A}</td>
<td>{E, A}</td>
</tr>
<tr>
<td>while loop #</td>
<td>$X$</td>
<td>DF($X$)</td>
<td>$F$</td>
<td>$W$</td>
</tr>
<tr>
<td>--------------</td>
<td>-----</td>
<td>---------</td>
<td>---------</td>
<td>-----------</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>{}</td>
<td>{B, C, D}</td>
</tr>
<tr>
<td>1</td>
<td>$B$</td>
<td>{D}</td>
<td>{D}</td>
<td>{C, D}</td>
</tr>
<tr>
<td>2</td>
<td>$C$</td>
<td>{D, E}</td>
<td>{D, E}</td>
<td>{D, E}</td>
</tr>
<tr>
<td>3</td>
<td>$D$</td>
<td>{E, A}</td>
<td>{D, E, A}</td>
<td>{E, A}</td>
</tr>
<tr>
<td>4</td>
<td>$E$</td>
<td>{}</td>
<td>{D, E, A}</td>
<td>{A}</td>
</tr>
<tr>
<td>while loop #</td>
<td>X</td>
<td>DF(X)</td>
<td>F</td>
<td>W</td>
</tr>
<tr>
<td>-------------</td>
<td>-----</td>
<td>-------</td>
<td>------</td>
<td>----------------</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>{}</td>
<td>{B, C, D}</td>
</tr>
<tr>
<td>1</td>
<td>B</td>
<td>{D}</td>
<td>{D}</td>
<td>{C, D}</td>
</tr>
<tr>
<td>2</td>
<td>C</td>
<td>{D, E}</td>
<td>{D, E}</td>
<td>{D, E}</td>
</tr>
<tr>
<td>3</td>
<td>D</td>
<td>{E, A}</td>
<td>{D, E, A}</td>
<td>{E, A}</td>
</tr>
<tr>
<td>4</td>
<td>E</td>
<td>{}</td>
<td>{D, E, A}</td>
<td>{A}</td>
</tr>
<tr>
<td>5</td>
<td>A</td>
<td>{A}</td>
<td>{D, E, A}</td>
<td>{}</td>
</tr>
</tbody>
</table>
Example: $x$
Computing Dominance Frontier

(a) CFG

(b) DJ-graph

(c) DF-graph

(d) DF+-graph
for \((a, b) \in \text{CFG edges}\) do
\[
x \leftarrow a;
\]
while \(x\) does not strictly dominate \(b\) do
\[
\text{DF}(x) \leftarrow \text{DF}(x) \cup b;
\]
\[
x \leftarrow \text{immediate dominator}(x);
\]
end
end
Variable Renaming (Second Phase)

```plaintext
foreach v : Variable do
    v.reachingDef ← ⊥;
end

foreach BB : basic Block in depth-first search preorder traversal of the dominance tree do
    foreach i : instruction in linear code sequence of BB do
        foreach v : variable used by non-φ-function i do
            updateReachingDef(v, i);
            replace this use of v by v.reachingDef in i;
        end
        foreach v : variable defined by i (may be a φ-function) do
            updateReachingDef(v, i);
            create fresh variable v';
            replace this definition of v by v' in i;
            v'.reachingDef ← v.reachingDef;
            v.reachingDef ← v';
        end
    end
    foreach φ : φ-function in a successor of BB do
        foreach v : variable used by φ do
            updateReachingDef(v, φ);
            replace this use of v by v.reachingDef in φ;
        end
    end
end
```
Procedure updateReachingDef(v, i) Utility function for SSA renaming

Data: v : variable from program
Data: i : instruction from program

/* search through chain of definitions for v until we find the closest 
definition that dominates i, then update v.reachingDef in-place with 
this definition */

1. \( r \leftarrow v.\text{reachingDef}; \)
2. while not \( (r \equiv \bot \text{ or definition}(r) \text{ dominates } i) \) do
3. \hspace{1cm} \( r \leftarrow r.\text{reachingDef}; \)
4. end
5. \( v.\text{reachingDef} \leftarrow r; \)
What’s next?

- No we have a nice code in SSA form, but
What’s next?

- No we have a nice code in SSA form, but it has \( \phi \)-functions all over the place that we do not know how to implement them.
What’s next?

- No we have a nice code in SSA form, but it has $\phi$-functions all over the place that we do not know how to implement them.
- Let’s remove them!
What’s next?

- No we have a nice code in SSA form, but it has $\phi$-functions all over the place that we do not know how to implement them.
- Let’s remove them!
- How?
SSA Destruction

What’s next?

- No we have a nice code in SSA form, but it has $\phi$-functions all over the place that we do not know how to implement them.
- Let’s remove them!
- How? rename all $\phi$-related variables ($\phi$-web) to one unique name.
Finding $\phi$-webs

```
1 begin
2   for each variable $v$ do
3     phiweb($v$) ← \{v\};
4   end
5   for each instruction of the form $a_{\text{dest}} = \phi(a_1, \ldots, a_n)$ do
6     for each source operand $a_i$ in instruction do
7       union(phiweb($a_{\text{dest}}$), phiweb($a_i$))
8   end
9 end
10 end
```
Is it really minimal?

It can insert a $\phi$-function to merge two values that are never used after the merge.

Solution

Construct pruned SSA that uses global data-flow analysis to decide where values are live, so it only inserts $\phi$-function at those merge points where the analysis indicates that the value is potentially live.
Is it really minimal?

It can insert a $\phi$-function to merge two values that are never used after the merge.
Is it really minimal?

It can insert a $\phi$-function to merge two values that are never used after the merge.

Solution
Is it really minimal?

It can insert a \( \phi \)-function to merge two values that are never used after the merge.

Solution

Construct pruned SSA that uses global data-flow analysis to decide where values are live, so it only inserts \( \phi \)-function at those merge points where the analysis indicates that the value is potentially live.
Pruned SSA

Is it really the solution?

• Time-consuming, since computing the live ranges is not trivial.
• Space-consuming, it increases the space requirements for the build process.
Pruned SSA

Is it really the solution?

- Time-consuming
Pruned SSA

Is it really the solution?

- Time-consuming, since computing the live ranges is not trivial.
Is it really the solution?

- Time-consuming, since computing the live ranges is not trivial.
- Space-consuming
Is it really the solution?

- Time-consuming, since computing the live ranges is not trivial.
- Space-consuming, it increases the space requirements for the build process.
SSA Flavors Example

Original Code

Minimal SSA

Semi-pruned SSA

Pruned SSA
Thank you!