8. Static Single Assignment Form

Marcus Denker
Roadmap

> Static Single Assignment Form (SSA)
> Converting to SSA Form
> Examples
> Transforming out of SSA
Static Single Assignment Form

> Goal: simplify procedure-global optimizations

> Definition:

Program is in SSA form if every variable is only assigned once
Why Static?

- Why Static?
  - We only look at the static program
  - One assignment per variable in the program

> At runtime variables are assigned multiple times!
**Example: Sequence**

> Easy to do for sequential programs:

Original

\[
\begin{align*}
  a &:= b + c \\
  b &:= c + 1 \\
  d &:= b + c \\
  a &:= a + 1 \\
  e &:= a + b
\end{align*}
\]

SSA

\[
\begin{align*}
  a_1 &:= b_1 + c_1 \\
  b_2 &:= c_1 + 1 \\
  d_1 &:= b_2 + c_1 \\
  a_2 &:= a_1 + 1 \\
  e_1 &:= a_2 + b_2
\end{align*}
\]
Example: Condition

> Conditions: what to do on control-flow merge?

Original

\[
\begin{align*}
\text{if } B \text{ then} \\
a &:= b \\
\text{else} \\
a &:= c \\
\text{end} \\
\ldots \ a \ldots
\end{align*}
\]

SSA

\[
\begin{align*}
\text{if } B \text{ then} \\
a_1 &:= b \\
\text{else} \\
a_2 &:= c \\
\text{End} \\
\ldots \ a? \ldots
\end{align*}
\]
**Solution: $\Phi$-Function**

> Conditions: what to do on control-flow merge?

**Original**

```plaintext
if B then
  a := b
else
  a := c
end
... a ...
```

**SSA**

```plaintext
if B then
  a_1 := b
else
  a_2 := c
End
a_3 := \Phi(a_1, a_2)
... a_3 ...
```
The \( \Phi \)-Function

> \( \Phi \)-functions are always at the beginning of a basic block

> Select between values depending on control-flow

> \( a_1 := \Phi(a_1 \ldots a_k) \): the block has \( k \) preceding blocks

\textit{PHI-functions are all evaluated simultaneously.}
SSA and CFG

- SSA is normally done for control-flow graphs (CFG)

- Basic blocks are in 3-address form
> A CFG models *transfer of control* in a program
  — nodes are *basic blocks* (straight-line blocks of code)
  — edges represent *control flow* (loops, if/else, goto …)
SSA: a Simple Example

if B then
  a1 := 1
else
  a2 := 2
End
a3 := PHI(a1,a2)
... a3 ...

Diagram:
- B
- a1 := 2
- a2 := 2
- a3 := PHI(a1,a2)
  ... a3 ...
Repeat: IR

- front end produces IR
- optimizer transforms IR to more efficient program
- back end transform IR to target code
SSA as IR

source code → frontend → optimizer (IR) → backend (IR) → machine code

GenSSA → OPT → Remove SSA → SSA → SSA

SSA

© Marcus Denker
Transforming to SSA

> Problem: Performance / Memory

  — Minimize number of inserted $\Phi$-functions
  — Do not spend to much time

> Many relatively complex algorithms

  — We do not go too much into details
  — See literature!
Minimal SSA

> Two steps:
  — Place $\Phi$-functions
  — Rename Variables

> Where to place $\Phi$-functions?

> We want minimal amount of needed $\Phi$
  — Save memory
  — Algorithms will work faster
Path Convergence Criterion

> There should be a $\Phi$ for $a$ at node $Z$ if:

1. There is a block $X$ containing a definition of $a$.
2. There is a block $Y$ ($Y \neq X$) containing a definition of $a$.
3. There is a nonempty path $P_{xz}$ of edges from $X$ to $Z$.
4. There is a nonempty path $P_{yz}$ of edges from $Y$ to $Z$.
5. Path $P_{xz}$ and $P_{yz}$ do not have any nodes in common other than $Z$.
6. The node $Z$ does not appear within both $P_{xz}$ and $P_{yz}$ prior to the end (although it may appear in one or the other).
Iterated Path-Convergence

> Inserted $\Phi$ is itself a definition!

```
While there are nodes X,Y,Z satisfying conditions 1-5
    and Z does not contain a phi-function for a
  do
    insert PHI at node Z.
```

A bit slow, other algorithms used in practice
Example (Simple)

1. block X containing a definition of a
2. block Y (Y \( \neq \) X) containing a definition of a.
3. path \( P_{xz} \) of edges from X to Z.
4. path \( P_{yz} \) of edges from Y to Z.

5. Path \( P_{xz} \) and \( P_{yz} \) do not have any nodes in common other than Z.
6. The node Z does not appear within both \( P_{xz} \) and \( P_{yz} \) prior to the end.
Dominance Property of SSA

> Dominance: node $D$ dominates node $N$ if every path from the start node to $N$ goes through $D$.

(“strictly dominates”: $D \neq N$)

Dominance Property of SSA:

1. If $x$ is used in a Phi-function in block $N$, then the definition of $x$ dominates every predecessor of $N$.
2. If $x$ is used in a non-Phi statement in $N$, then the definition of $x$ dominates $N$

“Definition dominates use”
Dominance and SSA Creation

> Dominance can be used to efficiently build SSA

> $\phi$-Functions are placed in all basic blocks of the Dominance Frontier.

> **Dominance Frontier:** the set of all nodes $N$ such that $D$ dominates an immediate predecessor of $N$ but does not strictly dominate $N$. 
Dominance and SSA Creation

DF(D) = the set of all nodes N such that D dominates an immediate predecessor of N but does not strictly dominate N.

Intuition: Nodes at the border of a region of dominance
Dominance and SSA Creation

DF(D) = the set of all nodes N such that D dominates an immediate predecessor of N but does not strictly dominate N.
Dominance and SSA Creation

\[ DF(D) = \text{the set of all nodes } N \text{ such that } D \text{ dominates an immediate predecessor of } N \text{ but does not strictly dominate } N. \]

**Intuition:**

*Nodes at the border of a region of dominance*
Dominance and SSA Creation
Dominance and SSA Creation

5 Dominates all nodes in the gray area
Targets of edges from the dominates by 5 to the region not strictly dominated by 5.

DF(5) = \{4, 5, 12, 13\}
Simple Example

DF(B1)=
DF(B2)=
DF(B3)=
DF(B4)=

B

B1

B2

a := 1

B3

a := 2

a

B4
Simple Example

DF(B1)={?}
DF(B2)=
DF(B3)=
DF(B4)=

DF(B1)={?}
DF(B2)=
DF(B3)=
DF(B4)=
Simple Example

DF(B1) = {}  
DF(B2) =  
DF(B3) =  
DF(B4) =  

 SSA
Simple Example

DF(B1)={}
DF(B2)={?}
DF(B3)=
DF(B4)=

B

B1

B2

a := 1

B3

a := 2

B4

a

SSA

© Marcus Denker
Simple Example

DF(B1) = {}  
DF(B2) = {B4}  
DF(B3) =  
DF(B4) =
Simple Example

DF(B1)=\{
DF(B2)={B4}
DF(B3)={B4}
DF(B4)=

Simple Example

DF(B1)={}
DF(B2)={B4}
DF(B3)={B4}
DF(B4)={}

SSA
DF(B1)={}  
DF(B2)={B4}  
DF(B3)={B4}  
DF(B4)={}

PHI-Function needed in B4 (for a)
Properties of SSA

> Simplifies many optimizations
  — Every variable has only one definition
  — Every use knows its definition, every definition knows its uses
  — Unrelated variables get different names

> **Examples:**
  — Constant propagation
  — Value numbering
  — Invariant code motion and removal
  — Strength reduction
  — Partial redundancy elimination

Next Week!
SSA in the Real World

> Invented end of the 80s, a lot of research in the 90s

> Used in many modern compilers

   — ETH Oberon 2
   — LLVM
   — GNU GCC 4
   — IBM Jikes Java VM
   — Java Hotspot VM
   — Mono
   — Many more…
Transforming out-of SSA

> Processor cannot execute $\Phi$-Function

> How do we remove it?
Simple Copy Placement

1. \( a_1 := b \)
2. \( a_2 := 2 \)
3. \( a_3 := \text{PHI}(a_1, a_2) \)  
   ... \( a_3 .... \)

1. \( a_1 := b \)
2. \( a_3 := a_1 \)
3. \( a_2 := 2 \)
4. \( a_3 := a_2 \)
5. ... \( a_3 .... \)
> Problems:
  — Copies need to be removed
  — Wrong in some cases after reordering of code
**Φ-Congruence**

Idea: transform program so that all variables in $Φ$ are the same:

$$a_1 = Φ(a_1, a_1) \quad ---> \quad a_1 = a_1$$

> Insert Copies
> Rename Variables
Φ-Congruence: Definitions

Φ-connected(x):

\[ a_3 = \Phi(a_1, a_2) \]
\[ a_5 = \Phi(a_3, a_4) \]

--> a1, a4 are connected

Φ-congruence-class:
Transitive closure of Φ-connected(x).
Φ-Congruence Property

Φ-congruence property:

All variables of the same congruence class can be replaced by one representative variable without changing the semantics.

SSA without optimizations has Φ-congruence property

Variables of the congruence class never live at the same time (by construction)
A variable $v$ is \textit{live} on edge $e$ if there is a path from $e$ to a use of $v$ not passing through a definition of $v$.

\begin{itemize}
  \item \textbf{a}\hspace{1cm}a:= 0
  \item \textbf{b}\hspace{1cm}b := a + 1
  \item \textbf{c}\hspace{1cm}c := c + b
  \item \textbf{a}\hspace{1cm}a := b * 2
  \item \textbf{a < N}\hspace{1cm}a < N
  \item \textbf{return c}
\end{itemize}

\begin{itemize}
  \item \textbf{b}\hspace{1cm}a:= 0
  \item \textbf{b}\hspace{1cm}b := a + 1
  \item \textbf{c}\hspace{1cm}c := c + b
  \item \textbf{a}\hspace{1cm}a := b * 2
  \item \textbf{a < N}\hspace{1cm}a < N
  \item \textbf{return c}
\end{itemize}

\begin{itemize}
  \item \textbf{c}\hspace{1cm}a:= 0
  \item \textbf{b}\hspace{1cm}b := a + 1
  \item \textbf{c}\hspace{1cm}c := c + b
  \item \textbf{a}\hspace{1cm}a := b * 2
  \item \textbf{a < N}\hspace{1cm}a < N
  \item \textbf{return c}
\end{itemize}

\textit{a and b are never live at the same time, so two registers suffice to hold a, b and c.}
Interference

A variable $v$ is *live* on edge $e$ if there is a path from $e$ to a use of $v$ not passing through a definition of $v$.

---

**a, c live at the same time: interference**
Φ-Removal: Big picture

CSSA: SSA with Φ-congruence-property.
  - directly after SSA generation
  - no interference

TSSA: SSA without Φ-congruence-property.
  - after optimizations
  - interference

1. Transform TSSA into CSSA (fix interference)
2. Rename Φ-variables
3. Delete Φ
Example: Problematic case

X2 and X3 interfere

Solution: Break up

\[ x_1 = \]

\[ x_2 = \text{phi}(x_1, x_3) \]
\[ x_3 = x_2 + 1 \]

\[ y = \text{phi}(x_1, x_3) \]
\[ x_2 = y \]
\[ x_3 = x_2 + 1 \]
SSA and Register Allocation

> Idea: remove $\Phi$ as late as possible

> Variables in $\Phi$-function never live at the same time!
   — *Can be stored in the same register*

> Do register allocation on SSA!
SSA: Literature

Books:
- SSA Chapter in Appel
  Modern Compiler Impl. In Java
- Chapter 8.11 Muchnik:
  Advanced Compiler Construction

SSA Creation:
Cytron et. al: *Efficiently computing Static Single Assignment Form and the Control Dependency Graph* (TOPLAS, Oct 1991)

PHI-Removal: Sreedhar et at. *Translating out of Static Single Assignment Form* (LNCS 1694)
Summary

> SSA, what it is and how to create it
  — Where to place $\Phi$-functions?

> Transformation out of SSA
  — Placing copies
  — Remove $\Phi$

Next Week: Optimizations
What you should know!

- When a program has SSA form.
- What is a $\Phi$-function.
- When do we place $\Phi$-functions
- How to remove $\Phi$-functions
Can you answer these questions?

✎ Why can we not directly generate executable code from SSA?
✎ Why do we use 3-address code and CFG for SSA?
License

> http://creativecommons.org/licenses/by-sa/2.5/

You are free:
- to copy, distribute, display, and perform the work
- to make derivative works
- to make commercial use of the work

Under the following conditions:

**Attribution.** You must attribute the work in the manner specified by the author or licensor.

**Share Alike.** If you alter, transform, or build upon this work, you may distribute the resulting work only under a license identical to this one.

- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.