CS 473: COMPILER DESIGN
STATIC SINGLE ASSIGNMENT
Homework 4 Questions

Top
Static Single Assignment

Many dataflow analyses are about relating variable uses to variable definitions.

What if there was exactly one definition for every variable?

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

\[
\begin{align*}
a_1 &= x + y \\
b_1 &= a_1 - 1 \\
a_2 &= y + b_1 \\
b_2 &= x \cdot 4 \\
a_3 &= a_2 + b_2
\end{align*}
\]

We can tell by looking at a use exactly which def it refers to.

**Static Single Assignment**: one defining instruction for each variable.

- *Dynamically*, the def might run multiple times, with different values each time.
Static Single Assignment

• What if there was exactly one definition for every variable?

\[
x = 0 \\
test: \\
\text{if } (x \geq 10) \text{ goto done} \\
y = x \times 4 \\
x = x + 1 \\
goto test
\]

\[
x_1 = 0 \\
test: \\
\text{if } (x_1 \geq 10) \text{ goto done} \\
y = x_1 \times 4 \\
x_2 = x_1 + 1 \\
goto test
\]

• We can tell by looking at a use exactly which def it refers to

• Static Single Assignment: one defining instruction for each variable
  – Dynamically, the def might run multiple times, with different values each time

• In general, this will be a problem at join points in the CFG, where different definitions come in along different paths
Phi Functions

• Solution: “ϕ (phi) function”
  – Fictitious operator, used only for analysis, implemented by move at assembly level
  – Chooses among different versions of a variable based on the path by which control enters the phi node

x = 0
test:
if (x >= 10) goto done
y = x * 4
x = x + 1
goto test

x₁ = 0
test:
if (x₁ >= 10) goto done
y = x₁ * 4
x₂ = x₁ + 1
goto test
Phi Functions

• Solution: “\( \phi \) (phi) function”
  – Fictitious operator, used only for analysis, implemented by move at assembly level
  – Chooses among different versions of a variable based on the path by which control enters the phi node

\[
x_1 = 0
\]

\[
\text{test:}
\]

\[
\text{if } (x \geq 10) \text{ goto done}
\]

\[
y = x \times 4
\]

\[
x = x + 1
\]

\[
goto \text{ test}
\]

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

\[
\text{if } (x_3 \geq 10) \text{ goto done}
\]

\[
y = x_3 \times 4
\]

\[
x_2 = x_3 + 1
\]

\[
goto \text{ test}
\]

• How does it know which x to use?
• Solution: “\( \phi \) (phi) function”
  – Chooses among different versions of a variable based on the path by which control enters the phi node

• How does it know which \( x \) to use?

• Answer 1: magic!
  – We’re just using this for dataflow analysis, not actually running the code
  – \( \phi(x_1, x_2) \) means “could be \( x_1 \) or \( x_2 \), we don’t know which”

• Answer 2: a move before the incoming edge
Static Single Assignment Form

- Static Single Assignment (SSA): every definition node defines a different variable
- We can tell where a use is defined just by looking at the variable name
- At join points, when multiple definitions might reach, we insert phi functions ($\phi$) to pick the right value

```plaintext
x = 0

test:
if (x >= 10) goto done
y = x * 4
x = x + 1
goto test
```

```plaintext
x_1 = 0

test:
x_3 = \phi(x_1, x_2)
if (x_3 >= 10) goto done
y = x_3 * 4
x_2 = x_3 + 1
goto test
```
Questions
Converting to SSA

- Static Single Assignment: every definition node defines a different variable
- We can tell where a use is defined just by looking at the variable name
- At join points, when multiple definitions might reach, we insert phi functions (φ) to pick the right value

- Simple transformation: insert a phi function at every join point (node with at least two predecessors), then number every def and change every use to the def that reaches it

\[ x = \phi(x) \]
Converting to SSA

• Simple transformation: insert a phi function at every join point (node with at least two predecessors), then number every def and change every use to the def that reaches it

• But every phi function is a new def, which means a new variable, which makes it harder to optimize

• We’d like to insert phi functions only where necessary – that is, where two or more different definitions of a variable reach the same point

• We can figure this out using the dominator tree!
The *dominance frontier* of a node \( n \) is the set of all nodes \( w \) such that \( n \) dominates a predecessor of \( w \), but does not strictly dominate \( w \).
The dominance frontier of a node $n$ is the set of all nodes $w$ such that $n$ dominates a predecessor of $w$, but does not strictly dominate $w$, i.e., the set of nodes we can reach by leaving the area dominated by $n$ (possibly including $n$).
• Observation: If node $n$ has a definition of variable $a$, then all the nodes in $n$’s dominance frontier need phi functions for $a$
Converting to SSA

• We’d like to insert phi functions only where necessary – that is, where two or more different definitions of a variable reach the same point
• Observation: If node $n$ has a definition of variable $a$, then all the nodes in $n$’s dominance frontier need phi functions for $a$
• So we can compute the dominance frontiers for the whole graph (using the dominator tree), then add phi functions in the dominance frontier for each node, then repeat (since each phi function is also a definition)

• Once we’ve inserted all the phi functions, we walk through the dominator tree, renaming each variable and all its uses ($a$ to $a_1$, $a_2$, etc.)
Converting to SSA

• Step 1: compute the dominator tree and dominance frontiers
• Step 2: insert phi functions in the dominance frontiers, repeat as necessary
• Step 3: rename every definition so each variable has exactly one def

```
x = 0
test:
if (x >= 10) goto done
y = x * 4
x = x + 1
goto test
```

```
x_1 = 0
test:
x_3 = \phi(x_1, x_2)
if (x_3 >= 10) goto done
y = x_3 * 4
x_2 = x_3 + 1
goto test
```
Questions

Top
SSA Optimizations

• In SSA, every use has exactly one definition
  – And every definition dominates its uses!

• This makes lots of optimizations easy:

  • Dead assignment elimination: if a def has no uses, delete it
  • Constant propagation: if a def is $x = c$, where $c$ is a constant, then delete it and replace all uses of $x$ with $c$
  • Copy propagation: if a def is $x = y$, where $y$ is a variable, then delete it and replace all uses of $x$ with $y$

• Without SSA, we’d have to do reaching definition analysis, and only replace uses when we know the def is the only one that reaches

• In SSA, we’ve done the analysis once and for all!
  – Each of these optimizations preserves SSA form
Implementing SSA Optimizations

• In SSA, every use has exactly one definition
  – And every definition dominates its uses!
• We can keep a list of (pointers to) the uses for each definition, so we can easily find the associated CFG nodes
• If the list is empty, the definition is dead
• When we delete dead code, we might eliminate uses of other variables:

\[
y = a + b \\
\ldots \\
\ldots \\
x = y + z \\
\ldots \\
// \text{no uses of } x
\]

\[
y = a + b \\
\ldots \\
\ldots \\
// \text{one fewer use of } y \text{ and } z!
\]
Static Single Assignment: Summary

• Instead of computing relationships between uses and definitions for every optimization, we can encode it in the IR!

• One assignment statement per variable, so every use is immediately tied to its single definition

• Need to insert phi functions at join points, to resolve places where a use might refer to multiple definitions

• Makes analysis and optimization a lot easier, and is used in many compiler IRs (most notably LLVM)
Questions

Top
Compilation in a Nutshell

Source Code
(Character stream)
if (b == 0) { a = 1; }

Token stream:
if ( b == 0 ) { a = 0 ; }

Abstract Syntax Tree:
```
If
  Eq
    b
    0
  Assn
    a
    1
  None
```

Assembly Code
```
11:
  cmpq %eax, $0
  jeq 12
  jmp 13
12:
  ...
```
Compilation in a Nutshell

Now you know how to translate from the programs you write to the programs that processors actually run!

Abstract Syntax Tree:

```
If
  Eq
  Assn
  None

b 0 a 1
```

Assembly Code

```
11:
  cmpq %eax, $0
  jeq 12
  jmp 13
12:
  store i64* %a, 1
  br label %13
13:
...
```