Suppose we want to compute an analysis over CFGs. We have two possible algorithms.

Algorithm A is simple but has worst-case $O(N^2)$ where a CFG has $N$ nodes and $E$ edges.

Algorithm B is more complicated but has worst-case complexity $O(N + \log(E))$.

Which algorithm should we use? Why?
Announcements

• Project 6 due today
• Project 7 out
  • Due Thursday Nov 29 (9 days)
• Project 8 out
  • Due Saturday Dec 8 (18 days)
• Final exam: Wed December 12, 9am-12pm, Emerson 305
  • Covers everything except guest lectures
    ‣ Lec 1-21, 23, 24, and all projects are fair game!
  • 30 multiple choice questions
  • Open book, open note, open laptop
  • No internet (except to look up notes, etc.),
    ‣ No looking up answers, no communicating with anyone
Today

- Static Single Assignment form
  - What and why
  - SSA to CFG
  - CFG to SSA
    - Dominance frontiers
  - Optimization algorithms using SSA
Pure vs Imperative

• Consider CFG available expression analysis

<table>
<thead>
<tr>
<th>Stmt</th>
<th>Gen</th>
<th>Kill</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{x := v}</td>
<td>{ v }</td>
<td>{ e \mid x \text{ in } e }</td>
</tr>
</tbody>
</table>

• If variables are immutable (i.e., are assigned exactly once) analysis simplifies!

<table>
<thead>
<tr>
<th>Stmt</th>
<th>Gen</th>
<th>Kill</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{x := v}</td>
<td>{ v }</td>
<td></td>
</tr>
</tbody>
</table>

• Empty kill set!
Almost all data flow analyses simplify when variables are defined once.
- no kills in dataflow analysis
- can interpret as either functional or imperative

Our monadic form had this property, which made many of the optimizations simpler.
- e.g., just keep around a set of available definitions that we keep adding to

On the other hand, imperative form (i.e., CFGs) allowed us to have control-flow graphs, not just trees.
Best of Both Worlds

• Static Single Assignment (SSA)
  • CFGs but with immutable variables
  • Plus a slight “hack” to make graphs work out
  • Now widely used (e.g., LLVM)
  • Intra-procedural representation only
    • An SSA representation for whole program is possible (i.e., each global variable and memory location has static single assignment), but difficult to compute
Idea Behind SSA

• Start with CFG code
• Give each definition a fresh name
• Propagate fresh name to subsequent uses

```
x := n
y := m
x := x + y
return x
```

```
x0 := n
y0 := m
x1 := x0 + y0
return x1
```
The Problem...

• What about control flow merges?

```
x := n
y := m
if x < y
x := x + 1
y := y - 1
y := x + 2
z := x * y
return z
```
The Problem...

• What about control flow merges?

```plaintext
x0 := n
y0 := m
if x0 < y0
  x1 := x0 + 1
  y1 := y0 - 1
  y2 := x0 + 2
  z0 := x? * y?
  return z0
```

The Solution

- Insert “phony” expressions for the merge

- A **phi node** is a phony “use” of a variable
  - As if an oracle chooses to set $x_2$ to either $x_0$ or $x_1$ based on which control flow edge was used to get to here

```plaintext
x_0 := n
y_0 := m
if x_0 < y_0
  x_1 := x_0 + 1
  y_1 := y_0 - 1
x_2 := \phi(x_1, x_0)
y_2 := x_0 + 2
y_3 := \phi(y_1, y_2)
z_0 := x_2 * y_3
return z_0
```
Wait, Remind Me Why Is This Useful

• Data-flow analysis and optimizations become simpler if each variable has 1 definition

• Compilers often build def-use chains
  • Connects definitions of variables with uses of them
  • Propagate dataflow facts directly from defs to uses, rather than through control flow graph
  • In SSA form, def-use chains are linear in size of original program; in non-SSA form may be quadratic

• Is relationship between SSA form and dominator structure of CFG
  • Simplifies algs such as interference graph construction
  • More info soon....

• Unrelated uses of same variable becomes different variables
Example

• Unrelated uses of same variable:

```plaintext
i := 0

i < N

A[i] := 0
i := i + 1

s := s + B[i]
i := i + 1

s := 0

i := 0

i < N

A[i] := 0

i := i + 1

i1 := 0

i3 := φ(i1, i2)
i3 < N

i2 := i3 + 1

i4 := 0

s1 := 0

i6 := φ(i4, i5)
i6 < N

s3 := φ(s1, s2)
i6 < N

s2 := s2 + B[i6]
i5 := i6 + 1
```
Remaining Issues

• How do we generate SSA from CFG representation?
  • In order to get benefits of SSA form

• How do we generate CFG (or MIPS) from SSA?
  • In order to take SSA form and continue with code generation
**SSA Back to CFG**

- Simply insert assignments corresponding to phi nodes on the edges
- Coalescing register allocation will get rid of copies...

```
x0 := n
y0 := m
if x0 < y0

x1 := x0 + 1
y1 := y0 - 1
y2 := x0 + 2
x2 := φ(x1, x0)
y3 := φ(y1, y2)
z0 := x2 * y3
return z0
```

```
x0 := n
y0 := m
if x0 < y0

x1 := x0 + 1
y1 := y0 - 1
x2 := x1
y3 := y1
y2 := x0 + 2
x2 := x0
y3 := y2
x2 := φ(x1, x0)
y3 := φ(y1, y2)
z0 := x2 * y3
return z0
```
• Insert phi nodes in each basic block except the start node.
  • Could limit insertion to nodes with >1 predecessor, but for simplicity we will insert phi nodes everywhere.

• Calculate the dominator tree.

• Traverse the dominator tree in a breadth-first fashion:
  • give each definition of $x$ a fresh index
  • propagate that index to all of the uses
    • each use of $x$ that is not killed by a subsequent definition.
    • propagate the last definition of $x$ to the successors’ phi nodes.
Example

\begin{align*}
x &:= n \\
y &:= m \\
a &:= 0
\end{align*}

\begin{align*}
\text{if } x > 0
\end{align*}

\begin{align*}
a &:= a + y \\
x &:= x - 1
\end{align*}

\begin{align*}
z &:= a + y \\
\text{return } z
\end{align*}
Example

- Insert phi nodes

A

\[
x := n
y := m
a := 0
\]

B

\[
x := \varphi(x, x)
y := \varphi(y, y)
a := \varphi(a, a)
if \ x > 0
\]

C

\[
x := \varphi(x)
y := \varphi(y)
a := \varphi(a)
a := a + y
x := x - 1
\]

D

\[
x := \varphi(x)
y := \varphi(y)
a := \varphi(a)
z := a + y
return z
\]
Example

- Dominators:

```
x := n
y := m
a := 0
```

```
x := φ(x, x)
y := φ(y, y)
a := φ(a, a)
if x > 0
```

```
x := φ(x)
y := φ(y)
a := φ(a)
a := a + y
x := x - 1
```

```
x := φ(x)
y := φ(y)
a := φ(a)
z := a + y
return z
```
• In breadth-first order:
  • give each definition of var a fresh index
  • propagate that index to each use within block
  • propagate to successor’s phi node

\[
\begin{align*}
A & : \ x := n \\
 & \ y := m \\
 & \ a := 0 \\

B & : \ x := \phi(x,x) \\
 & \ y := \phi(y,y) \\
 & \ a := \phi(a,a) \\
 & \text{if } x > 0
\end{align*}
\]

\[
\begin{align*}
C & : \ x := \phi(x) \\
 & \ y := \phi(y) \\
 & \ a := \phi(a) \\
 & \ a := a + y \\
 & \ x := x - 1
\end{align*}
\]

\[
\begin{align*}
D & : \ x := \phi(x) \\
 & \ y := \phi(y) \\
 & \ a := \phi(a) \\
 & \ z := a + y \\
 & \text{return } z
\end{align*}
\]
Example

• In breadth-first order:
  • give each definition of var a fresh index
  • propagate that index to each use within block
  • propagate to successor’s phi node

\[
\begin{align*}
  x_0 &:= n \\
y_0 &:= m \\
a_0 &:= 0
\end{align*}
\]

\[
\begin{align*}
x &:= \varphi(x_0, x) \\
y &:= \varphi(y_0, y) \\
a &:= \varphi(a_0, a) \\
\text{if } x > 0
\end{align*}
\]

\[
\begin{align*}
x &:= \varphi(x) \\
y &:= \varphi(y) \\
a &:= \varphi(a) \\
a &:= a + y \\
x &:= x - 1
\end{align*}
\]

\[
\begin{align*}
x &:= \varphi(x) \\
y &:= \varphi(y) \\
a &:= \varphi(a) \\
z &:= a + y \\
\text{return } z
\end{align*}
\]
Example

• In breadth-first order:
  • give each definition of var a fresh index
  • propagate that index to each use within block
  • propagate to successor’s phi node

x := φ(x1)
y := φ(y1)
a := φ(a1)
a := a + y
x := x - 1

x0 := n
y0 := m
a0 := 0

x1 := φ(x0, x)
y1 := φ(y0, y)
a1 := φ(a0, a)
if x1 > 0

x := φ(x1)
y := φ(y1)
a := φ(a1)
z := a + y
return z
Example

- In breadth-first order:
  - give each definition of var a fresh index
  - propagate that index to each use within block
  - propagate to successor’s phi node

```plaintext
x0 := n
y0 := m
a0 := 0

x1 := φ(x0, x)
y1 := φ(y0, y)
a1 := φ(a0, a)
if x1 > 0

x2 := φ(x1)
y2 := φ(y1)
a2 := φ(a1)
a3 := a2 + y2
x3 := x2 - 1

x := φ(x1)
y := φ(y1)
a := φ(a1)
z := a + y
return z
```
Example

- In breadth-first order:
  - give each definition of var a fresh index
  - propagate that index to each use within block
  - propagate to successor’s phi node

\[
\begin{align*}
x_0 &:= n \\
y_0 &:= m \\
a_0 &:= 0
\end{align*}
\]

\[
\begin{align*}
x_1 &:= \varphi(x_0, x_3) \\
y_1 &:= \varphi(y_0, y_2) \\
a_1 &:= \varphi(a_0, a_3) \\
\text{if } x_1 &> 0
\end{align*}
\]

\[
\begin{align*}
x_2 &:= \varphi(x_1) \\
y_2 &:= \varphi(y_1) \\
a_2 &:= \varphi(a_1) \\
a_3 &:= a_2 + y_2 \\
x_3 &:= x_2 - 1
\end{align*}
\]

\[
\begin{align*}
x &:= \varphi(x_1) \\
y &:= \varphi(y_1) \\
a &:= \varphi(a_1) \\
z &:= a + y \\
\text{return } z
\end{align*}
\]
Example

- In breadth-first order:
  - give each definition of `var` a fresh index
  - propagate that index to each use within block
  - propagate to successor’s phi node

```
x0 := n
y0 := m
a0 := 0
```

```
x1 := \phi(x0, x3)
y1 := \phi(y0, y2)
a1 := \phi(a0, a3)
if x1 > 0
```

```
x2 := \phi(x1)
y2 := \phi(y1)
a2 := \phi(a1)
a3 := a2 + y2
x3 := x2 - 1
```

```
x4 := \phi(x1)
y3 := \phi(y1)
a4 := \phi(a1)
z0 := a4 + y3
return z0
```
Example

- Could clean up using copy propagation and dead code elimination.

```
x0 := n
y0 := m
a0 := 0

A

x1 := φ(x0, x3)
y1 := φ(y0, y2)
a1 := φ(a0, a3)
if x1 > 0

B

x2 := φ(x1)
y2 := φ(y1)
a2 := φ(a1)
a3 := a2 + y2
x3 := x2 - 1

C

x4 := φ(x1)
y3 := φ(y1)
a4 := φ(a1)
z0 := a4 + y3
return z0

D

• Could clean up using copy propagation and dead code elimination
```
- Could clean up using copy propagation and dead code elimination
Smarter Algorithm for CFG to SSA

• Compute the **dominance frontier**
• Use dominance frontier to place phi nodes
  • Whenever block $n$ defines $x$, put a phi node for $x$ in every block in the dominance frontier of $n$
• Do renaming pass using dominator tree
Dominance Frontier

• Definition: $d$ dominates $n$ if every path from the start node to $n$ must go through $d$.

• Definition: if $d$ dominates $n$ and $d \neq n$, we say $d$ strictly dominates $n$.

• Definition: the dominance frontier of $n$ is the set of all nodes $w$ such that:
  1. $n$ dominates a predecessor of $w$
  2. $n$ does not strictly dominate $w$
Example

- Node 5
  - dominates 5, 6, 7, 8
  - strictly dominates 6, 7, 8
- Dominance frontier of 5 is 4, 5, 12, 13
  - Targets of edges from nodes dominated to nodes not strictly dominated
  - Dominance frontier of \( n \): where we transition from being dominated by \( n \) to being not strictly dominated
Example

- Recall alg:
  - Whenever block $n$ defines $x$, put a phi node for $x$ in every block in the dominance frontier of $n$
- Block $B$ strictly dominates $C,D$
- Dominance frontier of $B$ is $B$

```plaintext
x0 := n  
y0 := m  
a0 := 0

x1 := \phi(x0, x3)  
a1 := \phi(a0, a3)  
if x1 > 0

a3 := a1 + y0  
x3 := x1 - 1

z0 := a1 + y1  
return z0
```
Notes

• Adding a phi node for variable x is a new definition of x
  • Need to iterate until we satisfy the dominance frontier criterion:
    • Whenever block n defines x, put a phi node for x in every block in the dominance frontier of n

• Algorithm does work proportional to number of edges in control flow graph + size of the dominance frontiers.
  • Pathological cases can lead to quadratic behavior.
  • In practice, linear

• Computing dominator tree using iterative dataflow algorithm
  • With careful engineering, worst case complexity is quadratic, but in practice linear
  • See “A Simple, Fast Dominance Algorithm” by Cooper, Harvey, and Kennedy, Software Practice & Experience 4, 2001
    • Faster than an $O(N+\log(E))$ algorithm for CFGs with <30,000 nodes
We promised some optimization algorithms were simpler in SSA! Let’s look at some...

Assume that our compiler data structures include:

- Statement
- Variable: has definition site (statement) and list of use sites
- Block: has list of statements, ordered list of predecessors, successor(s)
Dead-Code Elimination

• Recall: Variable \( x \) is **live** at program point \( p \) is there is a path from \( p \) to a use of variable \( x \)

• A variable is live at its definition site if and only if its list of uses is non empty
  • Thanks SSA! Definition site dominates all uses, so there is a path from definition site to use site

• Iterative alg for removing dead code:
  • While there is a variable \( x \) with no uses and the statement that defines \( x \) has no other side effects:
    • Delete the statement that defines \( x \)
Work-list Algorithm for DCE

\( W \leftarrow \text{all variables in SSA program} \)

while \( W \) is not empty:

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

if \( v \)’s list of uses is empty:

let \( S \) be \( v \)’s statement of definition

if \( S \) has no side effects other than assignment to \( v \):

delete \( S \) from program

for each \( x_i \) used by \( S \):

delete \( S \) from list of uses of \( x_i \)

\( W \leftarrow W \cup \{ x_i \} \)
More Aggressive DCE

- Consider program
  
  ```
  a := 0;
  for (int i = 0; i < N; i++) {
    a := a+i;
  }
  return 1
  ```

- Variables are live at definition site, but doesn’t contribute to result of program!
More Agressive DCE

- Mark live any statement that:
  - 1. stores into mem, performs I/O, returns from function, calls function that may have side effects
  - 2. defines variable that is used in a live statement
  - 3. is a conditional branch that affects whether a live statement is executed (i.e., live statement is control dependent on the branch)
- Remove all unmarked statements

```plaintext
a0 := 0
i0 := 0

a1 := φ(a0,a2)
i1 := φ(i0,i2)
a2 := a1 + i1
i2 := i1 + 1
if i2 < N
    return 1
```
More Agressive DCE

• Mark live any statement that:
  • 1. stores into mem, performs I/O, returns from function, calls function that may have side effects
  • 2. defines variable that is used in a live statement
  • 3. is a conditional branch that affects whether a live statement is executed (i.e., live statement is control dependent on the branch)

• Remove all unmarked statements
Simple Constant Propagation

• Any statement $x := c$ for constant $c$: can replace uses of $x$ with $c$
• Any phi node $x := \varphi(c, \ldots, c)$ can be replaced with $x := c$
• Easy to detect and implement with SSA form!

\[ W \leftarrow \text{all statements in SSA program} \]
while $W$ is not empty:
  remove some $S$ from $W$
  if $S$ is of form $x := \varphi(c, \ldots, c)$:
    replace $S$ with $x := c$
  if $S$ is of form $x := c$:
    delete $S$ from program
  for each statement $T$ that uses $x$
    substitute $c$ for $x$ in $T$
\[ W \leftarrow W \cup \{ T \} \]