Dataflow Analysis

• Last time we looked at code transformations
  - Constant propagation
  - Copy propagation
  - Common sub-expression elimination
  - ...

• Today, dataflow analysis:
  - How to determine if it is legal to perform such an optimization
  - (Not doing analysis to determine if it is beneficial)

A sample program

```c
int fib10(void) {
    int n = 10;
    int older = 0;
    int old = 1;
    int result = 0;
    int i;
    if (n <= 1) return n;
    for (i = 2; i<n; i++) {
        result = old + older;
        older = old;
        old = result;
    }
    return result;
}
```

Simple Constant Propagation

• Can we do SCP?
• How do we recognize it?
• What aren't we doing?

• Metanote:
  - keep opts simple!
  - Use combined power
Reaching Definitions

- A definition of variable v at program point d reaches program point u if there exists a path of control flow edges from d to u that does not contain a definition of v.

Reaching Definitions (ex)

- 1 reaches 5, 7, and 14
- 2 reaches 8
- Older in 8 is reached by
  - 2
  - 9
- Tells us which definitions reach a particular use (ud-info)

Calculating Reaching Definitions

- A definition of variable v at program point d reaches program point u if there exists a path of control flow edges from d to u that does not contain a definition of v.

Meta-notes:
- (almost) always conservative
- only know what we know
- Keep it simple:
  - What opt(s), if run before this would help
  - What about:
    1: x <- 0
    2: if (false) x<-1
    3: ... x ...
  - Does 1 reach 3?
  - What opt changes this?

1: n <- 10
2: older <- 0
3: old <- 1
4: result <- 0
5: if n <= 1 goto 14
6: i <- 2
7: if i > n goto 13
8: result <- old + older
9: older <- old
10: old <- result
11: i <- i + 1
12: JUMP 7
13: return result
14: return n
**Gen and kill for each stmt**

1: n <- 10
2: older <- 0
3: old <- 1
4: result <- 0
5: if n <= 1 goto 14
6: i <- 2
7: if i > n goto 13
8: result <- old + older
9: older <- old
10: old <- result
11: i <- i + 1
12: JUMP 7
13: return result
14: return n

How can we determine the defs that reach a node? We can use:
- control flow information
- gen and kill info

**Computing in[n] and out[n]**

- In[n]: the set of defs that reach the beginning of node n
- Out[n]: the set of defs that reach the end of node n

\[
in[n] = \bigcup_{p \in \text{pred}[n]} \text{out}[p]
\]
\[
\text{out}[n] = \text{gen}[n] \cdot (\text{in}[n] - \text{kill}[n])
\]

- Initialize in[n]=out[n]={} for all n
- Solve iteratively

**What is pred[n]??**

- Pred[n] are all nodes that can reach n in the control flow graph.
- E.g., pred[7] = { 6, 12 }

**What order to eval nodes?**

- Does it matter?
- Lets do: 1,2,3,4,5,14,6,7,13,8,9,10,11,12
Example:

- Order: 1,2,3,4,5,14,6,7,13,8,9,10,11,12

\[ in[n] = \bigvee_{p \in \text{pred}[n]} out[p] \quad out[n] = \text{gen}[n] Y (in[n] - \text{kill}[n]) \]

<table>
<thead>
<tr>
<th></th>
<th>Gen</th>
<th>kill</th>
<th>in</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>n &lt;- 10</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2:</td>
<td>older &lt;- 0</td>
<td>2</td>
<td>9</td>
<td>1,2</td>
</tr>
<tr>
<td>3:</td>
<td>old &lt;- 1</td>
<td>3</td>
<td>10</td>
<td>1,2,3</td>
</tr>
<tr>
<td>4:</td>
<td>result &lt;- 0</td>
<td>4</td>
<td>8</td>
<td>1-3</td>
</tr>
<tr>
<td>5:</td>
<td>if n &lt;= 1 goto 14</td>
<td>6</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>7:</td>
<td>if i &gt; n goto 13</td>
<td>8</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>8:</td>
<td>result &lt;- old + older</td>
<td>9</td>
<td>2</td>
<td>1-3,6,8,9</td>
</tr>
<tr>
<td>9:</td>
<td>older &lt;- old</td>
<td>10</td>
<td>3</td>
<td>1,3,6,8,9,16,8-10</td>
</tr>
<tr>
<td>11:</td>
<td>i &lt;- i + 1</td>
<td>11</td>
<td>6</td>
<td>1,6,8-10,18-11</td>
</tr>
<tr>
<td>12:</td>
<td>JUMP 7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13:</td>
<td>return result</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14:</td>
<td>return n</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example (pass 1)

- Order: 1,2,3,4,5,14,6,7,13,8,9,10,11,12

\[ in[n] = \bigvee_{p \in \text{pred}[n]} out[p] \quad out[n] = \text{gen}[n] Y (in[n] - \text{kill}[n]) \]

<table>
<thead>
<tr>
<th></th>
<th>Gen</th>
<th>kill</th>
<th>in</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>n &lt;- 10</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2:</td>
<td>older &lt;- 0</td>
<td>2</td>
<td>9</td>
<td>1,2</td>
</tr>
<tr>
<td>3:</td>
<td>old &lt;- 1</td>
<td>3</td>
<td>10</td>
<td>1,2,3</td>
</tr>
<tr>
<td>4:</td>
<td>result &lt;- 0</td>
<td>4</td>
<td>8</td>
<td>1-3</td>
</tr>
<tr>
<td>5:</td>
<td>if n &lt;= 1 goto 14</td>
<td>6</td>
<td>11</td>
<td>1-4</td>
</tr>
<tr>
<td>7:</td>
<td>if i &gt; n goto 13</td>
<td>8</td>
<td>4</td>
<td>1-4,6</td>
</tr>
<tr>
<td>9:</td>
<td>result &lt;- old + older</td>
<td>9</td>
<td>2</td>
<td>1-3,6,8,9</td>
</tr>
<tr>
<td>10:</td>
<td>old &lt;- result</td>
<td>10</td>
<td>3</td>
<td>1,3,6,8,9,16,8-10</td>
</tr>
<tr>
<td>11:</td>
<td>i &lt;- i + 1</td>
<td>11</td>
<td>6</td>
<td>1,6,8-10,18-11</td>
</tr>
<tr>
<td>12:</td>
<td>JUMP 7</td>
<td></td>
<td></td>
<td>1-8</td>
</tr>
<tr>
<td>13:</td>
<td>return result</td>
<td></td>
<td></td>
<td>1-4,6</td>
</tr>
<tr>
<td>14:</td>
<td>return n</td>
<td></td>
<td></td>
<td>1-4</td>
</tr>
</tbody>
</table>

Example (pass 2)

- Order: 1,2,3,4,5,14,6,7,13,8,9,10,11,12

\[ in[n] = \bigvee_{p \in \text{pred}[n]} out[p] \quad out[n] = \text{gen}[n] Y (in[n] - \text{kill}[n]) \]

<table>
<thead>
<tr>
<th></th>
<th>Gen</th>
<th>kill</th>
<th>in</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>n &lt;- 10</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2:</td>
<td>older &lt;- 0</td>
<td>2</td>
<td>9</td>
<td>1,2</td>
</tr>
<tr>
<td>3:</td>
<td>old &lt;- 1</td>
<td>3</td>
<td>10</td>
<td>1,2,3</td>
</tr>
<tr>
<td>4:</td>
<td>result &lt;- 0</td>
<td>4</td>
<td>8</td>
<td>1-3</td>
</tr>
<tr>
<td>5:</td>
<td>if n &lt;= 1 goto 14</td>
<td>6</td>
<td>11</td>
<td>1-4,6,8-11</td>
</tr>
<tr>
<td>7:</td>
<td>if i &gt; n goto 13</td>
<td>8</td>
<td>4</td>
<td>1-4,6,8-11</td>
</tr>
<tr>
<td>9:</td>
<td>older &lt;- old</td>
<td>9</td>
<td>2</td>
<td>1-3,6,8-11</td>
</tr>
<tr>
<td>10:</td>
<td>old &lt;- result</td>
<td>10</td>
<td>3</td>
<td>1,3,6,8-11,16,8-11</td>
</tr>
<tr>
<td>11:</td>
<td>i &lt;- i + 1</td>
<td>11</td>
<td>6</td>
<td>1,6,8-11,18-11</td>
</tr>
<tr>
<td>12:</td>
<td>JUMP 7</td>
<td></td>
<td>1-8,11</td>
<td></td>
</tr>
<tr>
<td>13:</td>
<td>return result</td>
<td></td>
<td>1-4,6</td>
<td></td>
</tr>
<tr>
<td>14:</td>
<td>return n</td>
<td></td>
<td>1-4</td>
<td>1-4</td>
</tr>
</tbody>
</table>

An Improvement: Basic Blocks

- No need to compute this one stmt at a time
- For straight line code:
  - In[s1; s2] = in[s1]
  - Out[s1; s2] = out[s2]
- Can we combine the gen and kill sets into one set per BB?

<table>
<thead>
<tr>
<th></th>
<th>Gen</th>
<th>kill</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>i &lt;- 1</td>
<td>1,8,4</td>
</tr>
<tr>
<td>2:</td>
<td>j &lt;- 2</td>
<td>2</td>
</tr>
<tr>
<td>3:</td>
<td>k &lt;- 3 + i</td>
<td>3,11</td>
</tr>
<tr>
<td>4:</td>
<td>i &lt;- j</td>
<td>4,18</td>
</tr>
<tr>
<td>5:</td>
<td>m &lt;- i + k</td>
<td>5</td>
</tr>
</tbody>
</table>

- Gen[BB]={2,3,4,5}
- Kill[BB]={1,8,11}
- Gen[s1;s2]=
- Kill[s1;s2]=
**BB sets**

<table>
<thead>
<tr>
<th></th>
<th>Gen</th>
<th>kill</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>1,2,3,4</td>
<td>8,9,10</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>11</td>
</tr>
<tr>
<td>7</td>
<td>8,11</td>
<td>2,4,6</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>6</td>
</tr>
<tr>
<td>12</td>
<td>8-11</td>
<td>2-4,6</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Forward Dataflow**

- Reaching definitions is a forward dataflow problem:
  It propagates information from preds of a node to the node.
- Defined by:
  - Basic attributes: (gen and kill)
  - Transfer function: $\text{out}[b] = F_{bb}(\text{in}[b])$
  - Meet operator: $\text{in}[b] = M(\text{out}[p])$ for all $p \in \text{pred}(b)$
  - Set of values (a lattice, in this case powerset of program points)
  - Initial values for each node $b$
- Solve for fixed point solution
How to implement?

- Values?
- Gen?
- Kill?
- \( F_{bb} \)
- Order to visit nodes?
- When are we done?
  - In fact, do we know we terminate?

Implementing RD

- Values: bits in a bit vector
- Gen: 1 in each position generated, otherwise 0
- Kill: 0 in each position killed, otherwise 1
- \( F_{bb}: \) out\[b\] = (in\[b\] | gen\[b\]) & kill\[b\]
- Init in\[b\]=out\[b\]=0

- When are we done?
- What order to visit nodes? Does it matter?

RD Worklist algorithm

Initialize: \( \text{in}[B] = \text{out}[b] = \emptyset \)
Initialize: \( \text{in}[\text{entry}] = \emptyset \)
Work queue, \( W = \) all Blocks in topological order
while (\(|W| \neq 0\) ) { remove \( b \) from \( W \)
old = \( \text{out}[b] \)
in\[b\] = \{over all \( \text{pred}(p) \in b\}\} \cup \text{out}[p]
out\[b\] = \( \text{gen}[b] \cup (\text{in}[b] - \text{kill}[b]) \)
if (old \( \neq \) out\[b\]) \( W = W \cup \text{succ}(b) \)
}

Storing Rd information

- Use-def chains: for each use of var \( x \) in \( s \), a list of definitions of \( x \) that reach \( s \)

1: \( n \leftarrow 10 \)
2: \( \text{older} \leftarrow 0 \)
3: \( \text{old} \leftarrow 1 \)
4: \( \text{result} \leftarrow 0 \)
5: if \( n \leq 1 \) goto 14
6: \( i \leftarrow 2 \)
7: if \( i > n \) goto 13
8: \( \text{result} \leftarrow \text{old} + \text{older} \)
9: \( \text{older} \leftarrow \text{old} \)
10: \( \text{old} \leftarrow \text{result} \)
11: \( i \leftarrow i + 1 \)
12: JUMP 7
13: return \( \text{result} \)
14: return \( n \)
Def-use chains are valuable too

- Def-use chain: for each definition of var x, a list of all uses of that definition
- Computed from liveness analysis, a backward dataflow problem
- Def-use and use-def are different

```
x <- 1
z > y
y <- x + 1
z <- x + 3
```

Better Constant Propagation

- What about:
  ```
x <- 1
if (y > z)
x <- 1
a <- x
```


```
1: n <- 10
2: older <- 0
3: old <- 1
4: result <- 0
5: if n <= 1 goto 14
6: i <- 2
7: if i > n goto 13
8: result <- old + older
9: older <- old
10: old <- result
11: i <- i + 1
12: JUMP 7
13: return result
14: return n
```

Better Constant Propagation

- What about:
  ```
x <- 1
if (y > z)
x <- 1
a <- x
```

- Lattice
  ```
  -inf ... -2 -1 0 1 2 ... inf
  ⊥
  T
  ⊥
  ⊥ <- a ∧ ⊥
c <- c ∧ c
⊥ <- c ∧ d (if c ≠ d)
```

- Init all vars to: bot or top?
Loop Invariant Code Motion

- When can expression be moved out of a loop?

```
x <- y + z
```

```
a <- x ..
```

Liveness (def-use chains)

- A variable $x$ is live-out of a stmt $s$ if $x$ can be used along some path starting a $s$, otherwise $x$ is dead.
- Why is this important?
- How can we frame this as a dataflow problem?

Liveness as a dataflow problem

- This is a backwards analysis
  - A variable is live out if used by a successor
  - Gen: For a use: indicate it is live coming into $s$
  - Kill: Defining a variable $v$ in $s$ makes it dead before $s$ (unless $s$ uses $v$ to define $v$)
  - Lattice is just live (top) and dead (bottom)
- Values are variables
- $\text{In}[n] = \text{variables live before } n$
  - $\text{out}[n] - \text{kill}[n] \cup \text{gen}[n]$
- $\text{Out}[n] = \text{variables live after } n$
  - $\bigcup_{s \in \text{succ}(n)} \text{In}[s]$
Dead Code Elimination

- Code is dead if it has no effect on the outcome of the program.
- When is code dead?

When can we do CSE?

Available Expressions

- X+Y is “available” at statement S if
  - X+Y is computed along every path from the start to S AND
  - neither X nor Y is modified after the last evaluation of X+Y

\[
\begin{align*}
  a &\leftarrow 4 + i \\
  b &\leftarrow 4 + i \\
  \text{?} &
\end{align*}
\]
Computing Available Expressions

• Forward or backward?
• Values?
• Lattice?
• gen[b] =
  • if b evals expr e and doesn’t define variables used in e
• kill[b] =
  • if b assigns to x, exprs(x) are killed
• in[b] =
  • what to do at a join point?
• out[b] =
  • initialization?
• out[b] = in[b] – kill[b] \cup gen[b]
• in[b] = in[b] \cap \forall p \in \text{pred}(b), out[p]
• Initialization
  - All nodes, but entry are set to ALL avail
  - Entry is set to NONE avail

Constructing Gen & Kill

<table>
<thead>
<tr>
<th>Stmt</th>
<th>Gen</th>
<th>Kill</th>
</tr>
</thead>
<tbody>
<tr>
<td>t &lt;- x \ op y</td>
<td>(x \ op y)-kill[s]</td>
<td>{exprs containing t}</td>
</tr>
<tr>
<td>t &lt;- M[a]</td>
<td>{M[a]}-kill[s]</td>
<td></td>
</tr>
<tr>
<td>M[a] &lt;- b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>f(a, ...)</td>
<td></td>
<td>{M[x] for all x}</td>
</tr>
<tr>
<td>t &lt;- f(a,...)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Constructing Gen & Kill

<table>
<thead>
<tr>
<th>Stmt</th>
<th>Gen</th>
<th>Kill</th>
</tr>
</thead>
<tbody>
<tr>
<td>t &lt;- x op y</td>
<td>{x op y}-kill[s]</td>
<td>{exprs containing t}</td>
</tr>
<tr>
<td>t &lt;- M[a]</td>
<td>{M[a]}-kill[s]</td>
<td>{exprs containing t}</td>
</tr>
<tr>
<td>M[a] &lt;- b</td>
<td>{}</td>
<td>{for all x, M[x]}</td>
</tr>
<tr>
<td>f(a, ...)</td>
<td>{}</td>
<td>{for all x, M[x]}</td>
</tr>
<tr>
<td>t &lt;- f(a,...)</td>
<td>{}</td>
<td>{exprs containing t for all x, M[x]}</td>
</tr>
</tbody>
</table>

Example

Entry:

- c <- a+b  
- d <- a*c  
- e <- d*d  
- i <- 1

Gen={a+b,a*c,d*d}
Kill={c>d,c*2,i>10,i+1}

Gen={a+b,c*d}
Kill={c*2,M[x],a*c}

Gen={d*d}
Kill={M[x]}

Gen={a*c}
Kill={M[x]}

I<-> Entry

Exit:

W={1}

Example

Entry:

- c <- a+b  
- d <- a*c  
- e <- d*d  
- i <- 1

Gen={a+b,a*c,d*d}
Kill={c>d,c*2,i>10,i+1}

Gen={a+b,c*d}
Kill={c*2,M[x],a*c}

Gen={d*d}
Kill={M[x]}

Gen={a*c}
Kill={M[x]}

Gen={i>10}
Kill={i+1}

I<-> Entry

Exit:

W={1}
**Example**

$$c \leftarrow a + b$$
$$d \leftarrow a \cdot c$$
$$i \leftarrow 1$$

$$f[i] \leftarrow a + b$$
$$c \leftarrow c \cdot 2$$
$$br \ c > d$$

$$g[i] \leftarrow a \cdot c$$
$$g[i] \leftarrow d \cdot d$$

$$i \leftarrow i + 1$$
$$br \ i > 10$$

**Gen**={a+b,a*c,d*d}
**Kill**={c>d,c*2,i>10,i+1}

**Example**

$$c \leftarrow a + b$$
$$d \leftarrow a \cdot c$$
$$i \leftarrow 1$$

$$f[i] \leftarrow a + b$$
$$c \leftarrow c \cdot 2$$
$$br \ c > d$$

$$g[i] \leftarrow a \cdot c$$
$$g[i] \leftarrow d \cdot d$$

$$i \leftarrow i + 1$$
$$br \ i > 10$$

**Gen**={a+b,a*c,d*d}
**Kill**={c>d,c*2,i>10,i+1}

**Example**

$$c \leftarrow a + b$$
$$d \leftarrow a \cdot c$$
$$i \leftarrow 1$$

$$f[i] \leftarrow a + b$$
$$c \leftarrow c \cdot 2$$
$$br \ c > d$$

$$g[i] \leftarrow a \cdot c$$
$$g[i] \leftarrow d \cdot d$$

$$i \leftarrow i + 1$$
$$br \ i > 10$$

**Gen**={a+b,a*c,d*d}
**Kill**={c>d,c*2,i>10,i+1}

**Example**

$$c \leftarrow a + b$$
$$d \leftarrow a \cdot c$$
$$i \leftarrow 1$$

$$f[i] \leftarrow a + b$$
$$c \leftarrow c \cdot 2$$
$$br \ c > d$$

$$g[i] \leftarrow a \cdot c$$
$$g[i] \leftarrow d \cdot d$$

$$i \leftarrow i + 1$$
$$br \ i > 10$$

**Gen**={a+b,a*c,d*d}
**Kill**={c>d,c*2,i>10,i+1}

**Example**

$$c \leftarrow a + b$$
$$d \leftarrow a \cdot c$$
$$i \leftarrow 1$$

$$f[i] \leftarrow a + b$$
$$c \leftarrow c \cdot 2$$
$$br \ c > d$$

$$g[i] \leftarrow a \cdot c$$
$$g[i] \leftarrow d \cdot d$$

$$i \leftarrow i + 1$$
$$br \ i > 10$$

**Gen**={a+b,a*c,d*d}
**Kill**={c*d,e*2,f*3,i*10,i+1}
CSE

- Calculate Available expressions
- For every stmt in program
  If expression, \( x \text{ op } y \), is available {
    Compute reaching expressions for \( x \text{ op } y \)
    at this stmt
    foreach stmt in RE of the form \( t \leftarrow x \text{ op } y \)
    rewrite at: \( t' \leftarrow x \text{ op } y \)
    \( t \leftarrow t' \)
  }
  replace \( x \text{ op } y \) in stmt with \( t' \)

Calculating RE

- Could be dataflow problem, but not needed enough, so ...
- To find RE for \( x \text{ op } y \) at stmt \( S \)
  - traverse cfg backward from \( S \) until
    - reach \( t \leftarrow x + y \) (& put into RE)
    - reach definition of \( x \) or \( y \)
Dataflow Summary

<table>
<thead>
<tr>
<th>Union</th>
<th>intersection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reaching defs</td>
<td>Available exprs</td>
</tr>
<tr>
<td>Forward</td>
<td>Backward</td>
</tr>
<tr>
<td>Live variables</td>
<td></td>
</tr>
</tbody>
</table>

Later in course we look at bidirectional dataflow

Dataflow Framework

- Lattice
- Universe of values
- Meet operator
- Basic attributes (e.g., gen, kill)
- Traversal order
- Transfer function

Def-Use chains are expensive

```c
foo(int i, int j) {
    ...
    switch (i) {
        case 0: x=3; break;
        case 1: x=1; break;
        case 2: x=6; break;
        case 3: x=7; break;
        default: x = 11;
    }
    switch (j) {
        case 0: y=x+7; break;
        case 1: y=x+4; break;
        case 2: y=x-2; break;
        case 3: y=x+1; break;
        default: y=x+9;
    }
    ...
}
```

In general, \(N\)defs \(M\)uses \(\Rightarrow O(NM)\) space and time

A solution is to limit each var to ONE def site
Def-Use chains are expensive

foo(int i, int j) {
    ...
    switch (i) {
        case 0: x=3; break;
        case 1: x=1; break;
        case 2: x=6;
        case 3: x=7;
        default: x = 11;
    }
    x1 is one of the above x's
    switch (j) {
        A solution is to limit each
        var to ONE def site
        case 0: y=x1+7;
        case 1: y=x1+4;
        case 2: y=x1-2;
        case 3: y=x1+1;
        default: y=x1+9;
    }
    ...
}

Advantages of SSA

- Makes du-chains explicit
- Makes dataflow analysis easier
- Improves register allocation
  - Automatically builds Webs
  - Makes building interference graphs easier
- For most programs reduces space/time
  requirements

SSA

- Static single assignment is an IR where every
  variable is assigned a value at most once in the
  program text
- Easy for a basic block:
  - assign to a fresh variable at each stmt.
  - Each use uses the most recently defined var.
  - (Similar to Value Numbering)

Straight-line SSA

\[
\begin{align*}
  a &\leftarrow x + y \\
  b &\leftarrow a + x \\
  a &\leftarrow b + 2 \\
  c &\leftarrow y + 1 \\
  a &\leftarrow c + a 
\end{align*}
\]
Straight-line SSA

\[
\begin{align*}
a &\leftarrow x + y \\
b &\leftarrow a + x \\
a &\leftarrow b + 2 \\
c &\leftarrow y + 1 \\
a &\leftarrow c + a \\
a_1 &\leftarrow x + y \\
b_1 &\leftarrow a_1 + x \\
a_2 &\leftarrow b_1 + 2 \\
c_1 &\leftarrow y + 1 \\
a_3 &\leftarrow c_1 + a_2
\end{align*}
\]

SSA

- Static single assignment is an IR where every variable is assigned a value at most once in the program text
- Easy for a basic block:
  - assign to a fresh variable at each stmt.
  - Each use uses the most recently defined var.
  - (Similar to Value Numbering)
- What about at joins in the CFG?

Merging at Joins

\[
\begin{align*}
c &\leftarrow 12 \\
\text{if (i) } &\{ \\
a &\leftarrow x + y \\
b &\leftarrow a + x \\
\} \text{ else } &\{ \\
a &\leftarrow b + 2 \\
c &\leftarrow y + 1 \\
\} \\
a &\leftarrow c + a \\
c_1 &\leftarrow 12 \\
\text{if (i)}
\end{align*}
\]

SSA

- Static single assignment is an IR where every variable is assigned a value at most once in the program text
- Easy for a basic block:
  - assign to a fresh variable at each stmt.
  - Each use uses the most recently defined var.
  - (Similar to Value Numbering)
- What about at joins in the CFG?
  - Use a notional fiction: A Φ function
Merging at Joins

\[ c_1 \leftarrow 12 \]
\[ \text{if (i)} \]
\[ a_1 \leftarrow x + y \]
\[ b_1 \leftarrow a_1 + x \]
\[ a_2 \leftarrow b + 2 \]
\[ c_2 \leftarrow y + 1 \]
\[ a_3 \leftarrow \Phi(a_1, a_2) \]
\[ c_3 \leftarrow \Phi(c_1, c_2) \]
\[ b_2 \leftarrow \Phi(b_1, ?) \]
\[ a_4 \leftarrow c_3 + a_3 \]

The \( \Phi \) function

- \( \Phi \) merges multiple definitions along multiple control paths into a single definition.
- At a BB with \( p \) predecessors, there are \( p \) arguments to the \( \Phi \) function.
  \[ x_{\text{new}} \leftarrow \Phi(x_1, x_1, x_1, \ldots, x_p) \]

- How do we choose which \( x_i \) to use?
  - We don’t really care!
  - If we care, use moves on each incoming edge

"Implementing" \( \Phi \)

- Each assignment generates a fresh variable.
- At each join point insert \( \Phi \) functions for all live variables.

Trivial SSA

- Way too many \( \Phi \) functions inserted.
Minimal SSA

- Each assignment generates a fresh variable.
- At each join point insert $\Phi$ functions for all variables with multiple outstanding defs.

\[
\begin{align*}
x \leftarrow 1 \\
y \leftarrow x \\
y \leftarrow 2 \\
z \leftarrow y + x
\end{align*}
\]

\[
\begin{align*}
x_1 \leftarrow 1 \\
y_1 \leftarrow x_2 \\
y_2 \leftarrow 2 \\
y_3 \leftarrow \Phi(y_1, y_2) \\
z_1 \leftarrow y_3 + x_1
\end{align*}
\]

Another Example

\[
\begin{align*}
a \leftarrow 0 \\
b \leftarrow a + 1 \\
c \leftarrow c + b \\
a \leftarrow b \times 2 \\
a < N
\end{align*}
\]

\[
\begin{align*}
\text{return } c
\end{align*}
\]

Another Example

\[
\begin{align*}
a \leftarrow 0 \\
b \leftarrow a + 1 \\
c \leftarrow c + b \\
a \leftarrow b \times 2 \\
a < N
\end{align*}
\]

\[
\begin{align*}
\text{return } c
\end{align*}
\]

Let's optimize the following:

\[
\begin{align*}
i = 1; \\
j = 1; \\
k = 0; \\
\text{while } (k < 100) \{ \\
\quad \text{if } (j < 20) \{ \\
\quad \quad i = i; \\
\quad \quad k++; \\
\quad \}\text{else } \{ \\
\quad \quad j = k; \\
\quad \quad k += 2; \\
\quad \}\}
\}\text{return } j;
\end{align*}
\]

\[
\begin{align*}
i \leftarrow 1 \\
j \leftarrow 1 \\
k \leftarrow 0 \\
k < 100? \\
\quad j < 20? \\
\quad \quad i \leftarrow i \\
\quad \quad j \leftarrow i \\
\quad \quad k \leftarrow k + 1 \\
\quad j < k? \\
\quad \quad j \leftarrow k \\
\quad \quad k \leftarrow k + 2 \\
\quad \quad \text{return } j
\end{align*}
\]
First, turn into SSA

```
1 i ← 1
j ← 1
k ← 0

2 k < 100?

3 j < 20?
return j

4 j ← i
k ← k + 1

5 j ← k + 1
k ← k + 2

6 j ← i
k ← k + 1

7 k < 100?
```

Properties of SSA

```
1 i1 ← 1
j1 ← 1
k1 ← 0

2 j2 ← Φ(j4,j1)
k2 ← Φ(k4,k1)
k2 < 100?

3 j2 ← 20?
return j2

4 j3 ← i1
k3 ← k2 + 1

5 j3 ← j1
k3 ← k2 + 1

6 j5 ← k2
k5 ← k2 + 2

7 j4 ← Φ(j3,j5)
k4 ← Φ(k3,k5)
```

Constant Propagation

- If “v ← c”, replace all uses of v with c
- If “v ← Φ(c,c,c)” replace all uses of v with c

```
W ← list of all defs
while !W.isEmpty {
    Stmt S ← W.removeOne
    if S has form “v ← Φ(c,…,c)”
        replace S with V ← c
    if S has form “v ← c” then
        delete S
    foreach stmt U that uses v,
        replace v with c in U
    W.add(U)
}
```

Other stuff we can do?

- Copy propagation
  delete “x ← Φ(y)” and replace all x with y
- Constant Folding
  (Also, constant conditions too!)
- Unreachable Code
  Remember to delete all edges from unreachable block
Constant Propagation

1. \(i_1 \leftarrow 1\)
   \(j_1 \leftarrow 1\)
   \(k_1 \leftarrow 0\)

2. \(j_2 \leftarrow \Phi(j_1, 1)\)
   \(k_2 \leftarrow \Phi(k_4, k_1)\)
   \(k_z < 100?\)

3. \(j_2 < 20?\)
   \(\text{return } j_2\)

4. \(j_3 \leftarrow 1\)
   \(k_3 \leftarrow k_2 + 1\)

5. \(j_5 \leftarrow k_2\)
   \(k_5 \leftarrow k_2 + 2\)

6. \(j_4 \leftarrow \Phi(j_3, j_5)\)
   \(k_4 \leftarrow \Phi(k_3, k_5)\)

But, so what?
You will have to wait until next time :)
Summary

- Dataflow framework
  - Lattice, meet, direction, transfer function, initial values
- Du-chains, ud-chains
- CSE
- SSA
  - One static definition per variable
  - $\Phi$-functions

Conditional Constant Propagation

Tracks:
- Blocks (assume unexecuted until proven otherwise)
- Variables (assume not executed, only with proof of assignments of a non-constant value do we assume not constant)

Use a lattice for variables:

TOP = we have evidence that variable can hold different values at different times
integers = we have seen evidence that the var has been assigned a constant with the value
BOT = not executed

Lecture 2 15-745 © Seth Copen Goldstein 2005-8
Conditional Constant Propagation

1. \(i_1 \leftarrow 1\)
2. \(j_1 \leftarrow 1\)
3. \(k_1 \leftarrow 0\)

2. \(j_2 \leftarrow \Phi(j_4, 1)\)
3. \(k_2 \leftarrow \Phi(k_4, 0)\)
4. \(k_2 < 100?\)
5. \(j_2 < 20?\)
6. \(j_3 \leftarrow 1\)
7. \(k_3 \leftarrow k_2 + 1\)
8. \(j_2 \leftarrow \Phi(j_5, 1)\)
9. \(k_2 \leftarrow \Phi(k_5, 2)\)
10. \(k_2 < 100?\)
11. \(j_3 \leftarrow 1\)
12. \(k_3 \leftarrow k_2 + 1\)
13. \(j_4 \leftarrow \Phi(j_5, 1)\)
14. \(k_4 \leftarrow \Phi(k_5, 2)\)
Conditional Constant Propagation

1. \( i_1 \leftarrow 1 \)
   \( j_1 \leftarrow 1 \)
   \( k_1 \leftarrow 0 \)

2. \( j_2 \leftarrow \Phi(j_1, 1) \)
   \( k_2 \leftarrow \Phi(k_1, 0) \)
   \( k_2 < 100? \)

3. \( j_3 \leftarrow 1 \)
   \( k_3 \leftarrow k_2 + 1 \)
   \( j_4 \leftarrow \Phi(1, j_3) \)

4. \( j_5 \leftarrow k_2 \)
   \( k_5 \leftarrow k_2 + 2 \)

CCP

1. \( i_1 \leftarrow 1 \)
   \( j_1 \leftarrow 1 \)
   \( k_1 \leftarrow 0 \)

2. \( j_2 \leftarrow \Phi(j_1, j_1) \)
   \( k_2 \leftarrow \Phi(k_1, k_1) \)
   \( k_2 < 100? \)

3. \( j_3 \leftarrow i_1 \)
   \( k_3 \leftarrow k_2 + 1 \)

4. \( j_4 \leftarrow \Phi(j_3, j_3) \)

5. \( j_5 \leftarrow k_2 \)
   \( k_5 \leftarrow k_2 + 2 \)

6. return \( k_2 \leftarrow \Phi(k_3, 0) \)
   \( k_3 \leftarrow k_2 + 1 \)
   return 1