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;
    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

```c
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
```
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 \( v \).

![Control flow graph]

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)

Calculation 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 \( v \).

  - Build up RD stmt by stmt
  - Stmt \( s \), “d: v <- x op y”, generates \( d \)
  - Stmt \( s \), “d: v <- x op y”, kills all other \( \text{defs}(v) \)
  - Or,
    - \( \text{Gen}[s] = \{ d \} \)
    - \( \text{Kill}[s] = \text{defs}(v) - \{ d \} \)
Gen and kill for each stmt

```
Gen   kill
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
```

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

\[
in[n] = \bigcup_{p \in \text{pred}[n]} \text{out}[p]
\]

\[
\text{out}[n] = \text{gen}[n] \cup (\text{in}[n] - \text{kill}[n])
\]

- 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

\[
\begin{align*}
in[n] &= \bigcup_{p \in \text{pred}[n]} \text{out}[p] \\
\text{out}[n] &= \text{gen}[n] \cup (in[n] - \text{kill}[n])
\end{align*}
\]

<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>1</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>6:</td>
<td>i &lt;- 2</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>8:</td>
<td>result &lt;- old + older</td>
<td>9</td>
<td>2</td>
<td>1-3,6,8</td>
</tr>
<tr>
<td>9:</td>
<td>old &lt;- old</td>
<td>10</td>
<td>3</td>
<td>1,3,6,8,9</td>
</tr>
<tr>
<td>10:</td>
<td>old &lt;- result</td>
<td>11</td>
<td>6</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>1,6,8-10,1,8-11</td>
</tr>
<tr>
<td>12:</td>
<td>JUMP 7</td>
<td>12:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13:</td>
<td>return result</td>
<td>13:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14:</td>
<td>return n</td>
<td>14:</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

\[
\begin{align*}
in[n] &= \bigcup_{p \in \text{pred}[n]} \text{out}[p] \\
\text{out}[n] &= \text{gen}[n] \cup (in[n] - \text{kill}[n])
\end{align*}
\]

<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>1</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>6:</td>
<td>i &lt;- 2</td>
<td>6</td>
<td>11</td>
<td>1-4,6</td>
</tr>
<tr>
<td>7:</td>
<td>if i &gt; n goto 13</td>
<td>8</td>
<td>4</td>
<td>1-4,6,1-3,6,8,9</td>
</tr>
<tr>
<td>8:</td>
<td>result &lt;- old + older</td>
<td>9</td>
<td>2</td>
<td>1-3,6,8,9,16,8-10</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>10:</td>
<td>old &lt;- result</td>
<td>11</td>
<td>6</td>
<td>1,6,8-10,1,8-11</td>
</tr>
<tr>
<td>11:</td>
<td>i &lt;- i + 1</td>
<td>11</td>
<td>6</td>
<td>1,6,8-10,1,8-11</td>
</tr>
<tr>
<td>12:</td>
<td>JUMP 7</td>
<td>12:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13:</td>
<td>return result</td>
<td>13:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14:</td>
<td>return n</td>
<td>14:</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example (pass 2):

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

\[
\begin{align*}
in[n] &= \bigcup_{p \in \text{pred}[n]} \text{out}[p] \\
\text{out}[n] &= \text{gen}[n] \cup (in[n] - \text{kill}[n])
\end{align*}
\]

<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>1</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>6:</td>
<td>i &lt;- 2</td>
<td>6</td>
<td>11</td>
<td>1-4,6</td>
</tr>
<tr>
<td>7:</td>
<td>if i &gt; n goto 13</td>
<td>8</td>
<td>4</td>
<td>1-4,6,1-3,6,8,9</td>
</tr>
<tr>
<td>8:</td>
<td>result &lt;- old + older</td>
<td>9</td>
<td>2</td>
<td>1-3,6,8,9,16,8-10</td>
</tr>
<tr>
<td>9:</td>
<td>old &lt;- old</td>
<td>10</td>
<td>3</td>
<td>1,3,6,8,9,16,8-10</td>
</tr>
<tr>
<td>10:</td>
<td>old &lt;- result</td>
<td>11</td>
<td>6</td>
<td>1,6,8-10,1,8-11</td>
</tr>
<tr>
<td>11:</td>
<td>i &lt;- i + 1</td>
<td>11</td>
<td>6</td>
<td>1,6,8-10,1,8-11</td>
</tr>
<tr>
<td>12:</td>
<td>JUMP 7</td>
<td>12:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13:</td>
<td>return result</td>
<td>13:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14:</td>
<td>return n</td>
<td>14:</td>
<td></td>
<td></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, 1,8</td>
</tr>
<tr>
<td>5:</td>
<td>m &lt;- i + k</td>
<td>5</td>
</tr>
<tr>
<td>6:</td>
<td>old &lt;- old</td>
<td>1-4,6</td>
</tr>
<tr>
<td>7:</td>
<td>old &lt;- result</td>
<td>1-4,6</td>
</tr>
<tr>
<td>8:</td>
<td>result &lt;- old + older</td>
<td>1-4,6</td>
</tr>
<tr>
<td>9:</td>
<td>older &lt;- old</td>
<td>1-4,6</td>
</tr>
<tr>
<td>10:</td>
<td>old &lt;- result</td>
<td>1-4,6</td>
</tr>
<tr>
<td>11:</td>
<td>i &lt;- i + 1</td>
<td>1-4,6</td>
</tr>
<tr>
<td>12:</td>
<td>JUMP 7</td>
<td>1-4,6</td>
</tr>
<tr>
<td>13:</td>
<td>return result</td>
<td>1-4,6</td>
</tr>
<tr>
<td>14:</td>
<td>return n</td>
<td>1-4,6</td>
</tr>
</tbody>
</table>
BB sets

<table>
<thead>
<tr>
<th>In</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gen={1,2,3,4}</td>
<td>1,2,3,4</td>
</tr>
<tr>
<td>Kill={8,9,10}</td>
<td></td>
</tr>
<tr>
<td>Gen={6}</td>
<td>1,2,3,4</td>
</tr>
<tr>
<td>Kill={11}</td>
<td></td>
</tr>
<tr>
<td>Gen={8,9,10,11}</td>
<td>1-4,6,8-11</td>
</tr>
<tr>
<td>Kill={2,3,4,6}</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: out[b]=F_{bb}(in[b])
  - Meet operator: in[b]=M(out[p]) for all p \in 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
- \( F_{bb} \): \( out[b] = (in[b] \mid gen[b]) \& kill[b] \)
- Init \( in[b] = out[b] = 0 \)
- When are we done?
- What order to visit nodes? Does it matter?

RD Worklist algorithm

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

Storing Rd information

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

```plaintext
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
```
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

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: old <- 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

- Meet:
  a <- a ∧ 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?

\[ x \leftarrow y + z \]

\[ a \leftarrow \ldots x \ldots \]

• When all reaching definitions of operands are outside of loop, expression is loop invariant

• Use ud-chains to detect

• Can du-chains be helpful?

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?


dead

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 $X+Y$

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

- Forward or backward?
- Values?
- Lattice?
- \( \text{gen}[b] = \) if \( b \) evals expr \( e \) and doesn’t define variables used in \( e \)
- \( \text{kill}[b] = \) if \( b \) assigns to \( x \), exprs(\( x \)) are killed
- \( \text{in}[b] = \) An expr is avail only if avail on ALL edges, so: \( \text{in}[b] = \bigcap \) over all \( p \in \text{pred}(b) \), \( \text{out}[p] \)
- Initialization
  - All nodes, but entry are set to ALL avail
  - Entry is set to NONE avail

Computing Available Expressions

- Forward
- Values: all expressions
- Lattice: available, not-avail
- \( \text{gen}[b] = \) if \( b \) evals expr \( e \) and doesn’t define variables used in \( e \)
- \( \text{kill}[b] = \) if \( b \) assigns to \( x \), then all exprs using \( x \) are killed.
- \( \text{out}[b] = \text{in}[b] - \text{kill}[b] \cup \text{gen}[b] \)
- \( \text{in}[b] = \) what to do at a join point?
- Initialization

Constructing Gen & Kill

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

Example

<table>
<thead>
<tr>
<th>Entry</th>
<th>Gen={a+b, a<em>c, d</em>d}</th>
<th>Kill={c<em>d, c</em>2, i&gt;10, i+1}</th>
</tr>
</thead>
<tbody>
<tr>
<td>c \leftarrow a+b\</td>
<td>{}\</td>
<td>{}\</td>
</tr>
<tr>
<td>d \leftarrow a*c\</td>
<td>{}\</td>
<td>{}\</td>
</tr>
<tr>
<td>e \leftarrow d*d\</td>
<td>{}\</td>
<td>{}\</td>
</tr>
<tr>
<td>i \leftarrow 1\</td>
<td>{}\</td>
<td>{}\</td>
</tr>
<tr>
<td>f[i] \leftarrow a+b\</td>
<td>{}\</td>
<td>{}\</td>
</tr>
<tr>
<td>c \leftarrow c*2\</td>
<td>{}\</td>
<td>{}\</td>
</tr>
<tr>
<td>i \leftarrow i+1\</td>
<td>{}\</td>
<td>{}\</td>
</tr>
<tr>
<td>g[i] \leftarrow a*c\</td>
<td>{}\</td>
<td>{}\</td>
</tr>
<tr>
<td>g[i] \leftarrow d*d\</td>
<td>{}\</td>
<td>{}\</td>
</tr>
<tr>
<td>i \leftarrow i+1\</td>
<td>{}\</td>
<td>{}\</td>
</tr>
<tr>
<td>br i&gt;10\</td>
<td>{}\</td>
<td>{}\</td>
</tr>
</tbody>
</table>

Example

<table>
<thead>
<tr>
<th>Entry</th>
<th>Gen={a+b, a<em>c, d</em>d}</th>
<th>Kill={c<em>d, c</em>2, i&gt;10, i+1}</th>
</tr>
</thead>
<tbody>
<tr>
<td>c \leftarrow a+b\</td>
<td>{}\</td>
<td>{}\</td>
</tr>
<tr>
<td>d \leftarrow a*c\</td>
<td>{}\</td>
<td>{}\</td>
</tr>
<tr>
<td>e \leftarrow d*d\</td>
<td>{}\</td>
<td>{}\</td>
</tr>
<tr>
<td>i \leftarrow 1\</td>
<td>{}\</td>
<td>{}\</td>
</tr>
<tr>
<td>f[i] \leftarrow a+b\</td>
<td>{}\</td>
<td>{}\</td>
</tr>
<tr>
<td>c \leftarrow c*2\</td>
<td>{}\</td>
<td>{}\</td>
</tr>
<tr>
<td>i \leftarrow i+1\</td>
<td>{}\</td>
<td>{}\</td>
</tr>
<tr>
<td>g[i] \leftarrow a*c\</td>
<td>{}\</td>
<td>{}\</td>
</tr>
<tr>
<td>g[i] \leftarrow d*d\</td>
<td>{}\</td>
<td>{}\</td>
</tr>
<tr>
<td>i \leftarrow i+1\</td>
<td>{}\</td>
<td>{}\</td>
</tr>
<tr>
<td>br i&gt;10\</td>
<td>{}\</td>
<td>{}\</td>
</tr>
</tbody>
</table>

Example

<table>
<thead>
<tr>
<th>Entry</th>
<th>Gen={a+b, a<em>c, d</em>d}</th>
<th>Kill={c<em>d, c</em>2, i&gt;10, i+1}</th>
</tr>
</thead>
<tbody>
<tr>
<td>c \leftarrow a+b\</td>
<td>{}\</td>
<td>{}\</td>
</tr>
<tr>
<td>d \leftarrow a*c\</td>
<td>{}\</td>
<td>{}\</td>
</tr>
<tr>
<td>e \leftarrow d*d\</td>
<td>{}\</td>
<td>{}\</td>
</tr>
<tr>
<td>i \leftarrow 1\</td>
<td>{}\</td>
<td>{}\</td>
</tr>
<tr>
<td>f[i] \leftarrow a+b\</td>
<td>{}\</td>
<td>{}\</td>
</tr>
<tr>
<td>c \leftarrow c*2\</td>
<td>{}\</td>
<td>{}\</td>
</tr>
<tr>
<td>i \leftarrow i+1\</td>
<td>{}\</td>
<td>{}\</td>
</tr>
<tr>
<td>g[i] \leftarrow a*c\</td>
<td>{}\</td>
<td>{}\</td>
</tr>
<tr>
<td>g[i] \leftarrow d*d\</td>
<td>{}\</td>
<td>{}\</td>
</tr>
<tr>
<td>i \leftarrow i+1\</td>
<td>{}\</td>
<td>{}\</td>
</tr>
<tr>
<td>br i&gt;10\</td>
<td>{}\</td>
<td>{}\</td>
</tr>
</tbody>
</table>

Example

<table>
<thead>
<tr>
<th>Entry</th>
<th>Gen={a+b, a<em>c, d</em>d}</th>
<th>Kill={c<em>d, c</em>2, i&gt;10, i+1}</th>
</tr>
</thead>
<tbody>
<tr>
<td>c \leftarrow a+b\</td>
<td>{}\</td>
<td>{}\</td>
</tr>
<tr>
<td>d \leftarrow a*c\</td>
<td>{}\</td>
<td>{}\</td>
</tr>
<tr>
<td>e \leftarrow d*d\</td>
<td>{}\</td>
<td>{}\</td>
</tr>
<tr>
<td>i \leftarrow 1\</td>
<td>{}\</td>
<td>{}\</td>
</tr>
<tr>
<td>f[i] \leftarrow a+b\</td>
<td>{}\</td>
<td>{}\</td>
</tr>
<tr>
<td>c \leftarrow c*2\</td>
<td>{}\</td>
<td>{}\</td>
</tr>
<tr>
<td>i \leftarrow i+1\</td>
<td>{}\</td>
<td>{}\</td>
</tr>
<tr>
<td>g[i] \leftarrow a*c\</td>
<td>{}\</td>
<td>{}\</td>
</tr>
<tr>
<td>g[i] \leftarrow d*d\</td>
<td>{}\</td>
<td>{}\</td>
</tr>
<tr>
<td>i \leftarrow i+1\</td>
<td>{}\</td>
<td>{}\</td>
</tr>
<tr>
<td>br i&gt;10\</td>
<td>{}\</td>
<td>{}\</td>
</tr>
</tbody>
</table>
CSE

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

Calculating RE

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

<table>
<thead>
<tr>
<th>Forward</th>
<th>Backward</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reaching defs</td>
<td>Live variables</td>
</tr>
<tr>
<td>Available exprs</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, \[ N \text{defs} \times M \text{uses} \Rightarrow O(NM) \text{space and time} \]

A solution is to limit each var to ONE def site

```c
foo(int i, int j) {
    ...
    switch (i) {
        case 0: x=3;
        case 1: x=1;
        case 2: x=6;
        case 3: x=7;
        default: x = 11;
    }
    switch (j) {
        case 0: y=x+7;
        case 1: y=x+4;
        case 2: y=x-2;
        case 3: y=x+1;
        default: y=x+9;
    }
    ...
    ...
}```
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;
        case 3: x=7;
        default: x = 11;
    }
    x1 is one of the above x's
    switch (j) { 
        case 0: y=x1+7;
        case 1: y=x1+4;
        case 2: y=x1-2;
        case 3: y=x1+1;
        default: y=x1+9;
    }
}
```

A solution is to limit each var to ONE def site

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

```plaintext
a ← x + y
b ← a + x
a ← b + 2
c ← y + 1
a ← c + a
```
**Straight-line SSA**

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

**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**

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

**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 \( \Phi \) 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" \( \Phi \)

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

Trivial SSA

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

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

```
x ← 1
y ← x
y ← 2
z ← y + x
```

```
x1 ← 1
y1 ← x1
y2 ← 2
y3 ← $\Phi(y_1, y_2)
z1 ← y3 + x1
```

**Another Example**

```
a ← 0
b ← a + 1
c ← c + b
a ← b * 2
a < N
```

```
return c
```

**Another Example**

```
a ← 0
```

```
b ← a + 1
c ← c + b
```

```
a ← b * 2
```

```
a < N
```

**Let's optimize the following:**

```
i=1;
j=1;
k=0;
while (k<100) {
    if (j<20) {
        j=i;
k++;
    } else {
        j=k;
k+=2;
    }
} else {
    j=k;
k+=2;
}
return j;
```
**First, turn into SSA**

1. \( i \leftarrow 1 \)
2. \( j \leftarrow 1 \)
3. \( k \leftarrow 0 \)
4. \( k < 100? \)
5. \( j < 20? \)
6. \( j \leftarrow i \)
7. \( k \leftarrow k + 1 \)

**Properties of SSA**

- Only 1 assignment per variable
- Definitions dominate uses
- Can we use this to help with constant propagation?

**Constant Propagation**

- If "\( v \leftarrow c \)" replace all uses of \( v \) with \( c \)
- If "\( v \leftarrow \Phi(c,c,c) \)" replace all uses of \( v \) with \( c \)

```java
W <- list of alldefs
while !W.isEmpty {
    Stmt S <- W.removeOne
    if S has form "\( v \leftarrow \Phi(c,...,c) \)"
        replace S with \( V \leftarrow c \)
    if S has form "\( v \leftarrow c \)"
        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 "\( x \leftarrow \Phi(y) \)" and replace all \( x \) with \( y \)
delete "\( x \leftarrow y \)" and replace all \( x \) with \( y \)
- Constant Folding(Also, constant conditions too!)
- Unreachable CodeRemember to delete all edges from unreachable block
Constant Propagation

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

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

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

4. \( j_3 \leftarrow i_1 \)
   \( k_3 \leftarrow k_2 + 1 \)
   \( j_3 < 20? \)

5. \( j_3 \leftarrow k_2 \)
   \( k_3 \leftarrow k_2 + 2 \)

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

But so what?

You will have to wait til next time :)
Summary

• Dataflow framework
  - Lattice, meet, direction, transfer function, initial values
• Du-chains, ud-chains
• CSE
  - 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
Conditional Constant Propagation

\begin{align*}
i_1 & \leftarrow 1 \\
j_1 & \leftarrow 1 \\
k_1 & \leftarrow 0
\end{align*}

\begin{align*}
j_2 & \leftarrow \Phi(i_1,1) \\
k_2 & \leftarrow \Phi(k_1,0) \\
k_2 & < 100? \\
j_3 & \leftarrow 1 \\
k_3 & \leftarrow k_2 + 1 \\
j_4 & \leftarrow \Phi(j_3,1) \\
k_4 & \leftarrow \Phi(k_3,k_2)
\end{align*}

\begin{align*}
j_4 & \leftarrow \Phi(j_4,1) \\
k_4 & \leftarrow \Phi(k_4,0) \\
k_2 & < 100? \\
j_2 & < 20? \\
\text{return } j_2 \\
j_3 & \leftarrow 1 \\
k_3 & \leftarrow k_2 + 1 \\
j_5 & \leftarrow k_2 \\
k_5 & \leftarrow k_2 + 2
\end{align*}
Conditional Constant Propagation

1. $i_1 ← 1$
2. $j_1 ← 1$
3. $k_1 ← 0$

2. $j_2 ← \Phi(j_1, 1)$
3. $k_2 ← \Phi(k_1, 0)$
4. $k_2 < 100?$
5. $j_3 ← 1$
6. $k_3 ← k_2 + 1$
7. $j_4 ← \Phi(1, j_3)$
8. $k_4 ← \Phi(k_3, k)$

CCP

1. $i_1 ← 1$
2. $j_1 ← 1$
3. $k_1 ← 0$
4. $j_2 ← \Phi(j_1, 1)$
5. $k_2 ← \Phi(k_1, 0)$
6. $k_2 < 100?$
7. $j_3 ← 1$
8. $k_3 ← k_2 + 1$
9. $j_4 ← \Phi(j_3, j_5)$
10. $k_4 ← \Phi(k_3, k_5)$

Lecture 2 15-745 © Seth Copen Goldstein 2005-8