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 $v$.

#### 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 $v$.

  - Build up RD stmt by stmt
  - Stmt $s$, "$d: v <- x \text{ op } y$", generates $d$
  - Stmt $s$, "$d: v <- x \text{ op } y$", kills all other defs($v$)

  Or,
  - Gen[$s$] = { $d$ }
  - Kill[$s$] = defs($v$) - { $d$ }

### Gen and kill for each stmt

<table>
<thead>
<tr>
<th>Stmt</th>
<th>Gen</th>
<th>Kill</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n &lt;- 10$</td>
<td>$1$</td>
<td></td>
</tr>
<tr>
<td>$older &lt;- 0$</td>
<td>$2$</td>
<td>$9$</td>
</tr>
<tr>
<td>$old &lt;- 1$</td>
<td>$3$</td>
<td>$10$</td>
</tr>
<tr>
<td>$result &lt;- 0$</td>
<td>$4$</td>
<td>$8$</td>
</tr>
<tr>
<td>$if n &lt;= 1$ goto $14$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$i &lt;- 2$</td>
<td>$6$</td>
<td>$11$</td>
</tr>
<tr>
<td>$if i &gt; n$ goto $13$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$result &lt;- old + older$</td>
<td>$8$</td>
<td>$4$</td>
</tr>
<tr>
<td>$old &lt;- old$</td>
<td>$9$</td>
<td>$2$</td>
</tr>
<tr>
<td>$old &lt;- result$</td>
<td>$10$</td>
<td>$3$</td>
</tr>
<tr>
<td>$i &lt;- i + 1$</td>
<td>$11$</td>
<td>$6$</td>
</tr>
<tr>
<td>$JUMP$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$return result$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$return n$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Computing $\text{in}[n]$ and $\text{out}[n]$

- $\text{In}[n]$: the set of defs that reach the beginning of node $n$
- $\text{Out}[n]$: the set of defs that reach the end of node $n$

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

- Initialize $\text{in}[n]=\text{out}[n]=\emptyset$ for all $n$
- Solve iteratively

What is $\text{pred}[n]$?

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

Example:

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

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:

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

- $\text{Gen}$: kill in out
- $\text{Gen}$: $\text{kill}$ in out
- $\text{Gen}$: kill in out
- $\text{Example}$: $\text{Gen}$ kill in out
- $\text{Gen}$ kill in out
Example (pass 1)

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

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

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

Example (pass 2)

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

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

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

An Improvement: Basic Blocks

- No need to compute this one stmt at a time
- For straight line code:
  - \text{In}[s1; s2] = \text{in}[s1]
  - \text{Out}[s1; s2] = \text{out}[s2]
- Can we combine the gen and kill sets into one set per BB?
- \text{Gen}[BB] = \{2, 3, 4, 5\}
- \text{Kill}[BB] = \{1, 8, 11\}
- \text{Gen}[s1; s2] =
- \text{Kill}[s1; s2] =

BB sets

<table>
<thead>
<tr>
<th>\text{Gen}</th>
<th>\text{kill}</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: \text{n} \leftarrow 10</td>
<td>1, 2, 3, 4, 8, 9, 10</td>
</tr>
<tr>
<td>2: \text{older} \leftarrow 0</td>
<td>2, 9</td>
</tr>
<tr>
<td>3: \text{old} \leftarrow 1</td>
<td>3, 10</td>
</tr>
<tr>
<td>4: \text{result} \leftarrow 0</td>
<td>4, 8</td>
</tr>
<tr>
<td>5: \text{if n} \leqslant 1 \text{ goto 14}</td>
<td>1, 2, 3, 4, 8, 9, 10</td>
</tr>
<tr>
<td>6: \text{i} \leftarrow 2</td>
<td>6, 11</td>
</tr>
<tr>
<td>7: \text{if i} &gt; \text{n goto 13}</td>
<td>1-4, 6, 8-11</td>
</tr>
<tr>
<td>8: \text{result} \leftarrow \text{old} + \text{older}</td>
<td>8, 4</td>
</tr>
<tr>
<td>9: \text{older} \leftarrow \text{old}</td>
<td>9, 2</td>
</tr>
<tr>
<td>10: \text{old} \leftarrow \text{result}</td>
<td>10, 3</td>
</tr>
<tr>
<td>11: \text{i} \leftarrow \text{i} + 1</td>
<td>11, 6</td>
</tr>
<tr>
<td>12: \text{JUMP 7}</td>
<td>8-11</td>
</tr>
<tr>
<td>13: \text{return result}</td>
<td>2-4, 6</td>
</tr>
<tr>
<td>14: \text{return n}</td>
<td>1-4</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: in\([B]\) = out\([b]\) = ∅
Initialize: in\([entry]\) = ∅

Work queue, W = all Blocks in topological order
while (|W| != 0) {
    remove b from W
    old = out\([b]\)
    in\([b]\) = \{over all pred(p) ∈ b\} ∪ out\([p]\)
    out\([b]\) = gen\([b]\) ∪ (in\([b]\) - kill\([b]\))
    if (old != out\([b]\)) W = W ∪ 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 ← 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
```

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 is NOT symmetric to use-def

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

• Use a better lattice

Better Constant Propagation

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

• Use a better lattice

Loop Invariant Code Motion

• When can expression be moved out of a loop?

  x <- y + z
  a <- .. x ..
Loop Invariant Code Motion

- When can expression be moved out of a loop?
- 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?
Dead Code Elimination

- Code is dead if it has no effect on the outcome of the program.
- When is code dead?
  - When the definition is dead, and
  - When the instruction has no side effects
- So:
  - run liveness
  - Construct def-use chains
  - Any instruction which has no users and has no side effects can be eliminated

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

Computing Available Expressions

- Forward or backward?
- Values?
- Lattice?
- gen[b] =
- kill[b] =
- in[b] =
- out[b] =
- initialization?
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) )-kill[s]</td>
<td>( {\text{exprs containing} \ t} )</td>
</tr>
<tr>
<td>( t \leftarrow M[a] )</td>
<td>( M[a] )-kill[s]</td>
<td></td>
</tr>
<tr>
<td>( M[a] \leftarrow b )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( f(a, ...) )</td>
<td>( {M[x]} ) for all ( x )</td>
<td></td>
</tr>
<tr>
<td>( t \leftarrow f(a, ...) )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
W={6}

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}

Exit

W={3,7}

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}

Exit

CSE

- Calculate Available expressions
- For every stmt in program
  - If expression, `x op y`, is available {
    - Compute reaching expressions for `x op y`
    - 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