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

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?

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 op y", generates d
  - Stmt s, "d: v <- x op y", kills all other defs(v)

Gen and kill for each stmt

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

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 $n$

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

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

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

```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
```
Example (pass 1)

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

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

\[
\begin{array}{cccc}
1: & n & \leftarrow & 10 \\
2: & \text{older} & \leftarrow & 0 \\
3: & \text{old} & \leftarrow & 1 \\
4: & \text{result} & \leftarrow & 0 \\
5: & \text{if } n =\ 1 & \text{goto } & 14 \\
6: & i & \leftarrow & 2 \\
7: & \text{if } i > n & \text{goto } & 13 \\
8: & \text{result} & \leftarrow & \text{old} + \text{older} \\
9: & \text{old} & \leftarrow & \text{old} \\
10: & \text{old} & \leftarrow & \text{result} \\
11: & i & \leftarrow & i + 1 \\
12: & \text{JUMP } 7 \\
13: & \text{return result} \\
14: & \text{return } n
\end{array}
\]

Example (pass 2)

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

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

\[
\begin{array}{cccc}
1: & n & \leftarrow & 10 \\
2: & \text{older} & \leftarrow & 0 \\
3: & \text{old} & \leftarrow & 1 \\
4: & \text{result} & \leftarrow & 0 \\
5: & \text{if } n =\ 1 & \text{goto } & 14 \\
6: & i & \leftarrow & 2 \\
7: & \text{if } i > n & \text{goto } & 13 \\
8: & \text{result} & \leftarrow & \text{old} + \text{older} \\
9: & \text{old} & \leftarrow & \text{old} \\
10: & \text{old} & \leftarrow & \text{result} \\
11: & i & \leftarrow & i + 1 \\
12: & \text{JUMP } 7 \\
13: & \text{return result} \\
14: & \text{return } n
\end{array}
\]

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

Gen={1,2,3,4}
Kill={8,9,10}

Gen={6}
Kill={11}

Gen={8,9,10,11}
Kill={2,3,4,6}

In out
1,2,3,4

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
- \( F_{bb} : \text{out}[b] = (\text{in}[b] | \text{gen}[b]) \land \text{kill}[b] \)
- 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 1 1
2: old <- 0 1 1
3: old <- 1 1 1,2
4: result <- 0 1-3 1-3
5: if x <= 1 goto 14 1-4 1-4
6: i <- 2 1-4,6 1-4,6
7: if i > n goto 13 1-4,6,8-11 1-4,6,8-11
8: result <- old + older 1-4,6,8-11 1-3,6,8-11
9: older <- old 1-3,6,8-11 1-3,6,8-11
10: old <- result 1-3,6,8-11 1-3,6,8-11
11: i <- i + 1 1.6,8-11 1.8-11
12: JMP 7 1.8-11 1.8-11
13: return result 1-4,6 1-4,6
14: return n 1-4 1-4
```

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: 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

Better Constant Propagation

• What about: x <- 1
  if (y > z)
  x <- 1
  a <- x
  • Use a better lattice
  • Meet:
  a <- a ∧ top
  bot <- a ∧ bot
  c <- c ∧ c
  bot <- 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 ..
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 at 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
- In[n] = variables live before n
  = out[n] − kill[n] ∪ gen[n]
- Out[n] = variables live after n
  = \( Y \cdot \text{In}[s] \)

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?

a <- 4 + i
b <- 4 + i

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

a <- b+c
b <- a-d
c <- b+c
d <- a-d

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
- gen\[b\] = if b evals expr e and doesn't define variables used in e
- kill\[b\] = if b assigns to x, then all exprs using x are killed.
- out\[b\] = in\[b\] – kill\[b\] ∪ gen\[b\]
- in\[b\] = what to do at a join point?
- initialization?

Computing Available Expressions

- Forward
- Values: all expressions
- Lattice: available, not-avail
- 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
- out\[b\] = in\[b\] – kill\[b\] ∪ gen\[b\]
- in\[b\] = An expr is avail only if avail on ALL edges, so: in\[b\] = \cap over all p ∈ pred(b), out\[p\]
- Initialization
  - All nodes, but entry are set to ALL avail
  - Entry is set to NONE avail