Program Analysis

Data Flow Analysis (Part 3)
Outline

- First example: Available expressions
- Basic principles
- More examples
- Solving data flow problems
- Inter-procedural analysis
- Sensitivities
Data Flow Analyses

- Seen previously
  - Available expressions
- Next
  - Reaching definitions
  - Very busy expressions
  - Live variables
- Course project
  - Taint analysis
Reaching Definitions Analysis

Goal: For each program point, compute which assignments may have been made and may not have been overwritten

- Useful in various program analyses
- E.g., to compute a data flow graph
Example

```javascript
var x = 5;
var y = 1;
while (x > 1) {
    y = x * y;
    x = x - 1;
}
```
Example

```javascript
var x = 5;
var y = 1;
while (x > 1) {
    y = x * y;
    x = x - 1;
}
```

Definition reaches entry of this statement
Example

```javascript
var x = 5;
var y = 1;
while (x > 1) {
y = x * y;
x = x - 1;
}
```

All definitions reach the entry of this statement
Example

```
var x = 5;
var y = 1;
while (x > 1) {
    y = x * y;
    x = x - 1;
}
```

Three definitions reach entry of this statement
Defining the Analysis

- **Domain**: Definitions (i.e., assignments) in the code
  - Set of pairs \((v, s)\) of variables and stmts.
  - \((v, s)\) means a definition of \(v\) at \(s\)

- **Direction**: Forward

- **Meet operator**: Union
  - Because we care about definitions that *may* reach a program point
Defining the Analysis (2)

- **Transfer function:**
  
  \[ RD_{exit}(s) = (RD_{entry}(s) \setminus kill(s)) \cup gen(S) \]

- **Function** \( gen(s) \)
  
  - If \( s \) is assignment to \( v \): \( (v, s) \)
  
  - Otherwise: Empty set

- **Function** \( kill(s) \)
  
  - If \( s \) is assignment to \( v \): \( (v, s') \) for all \( s' \) that define \( v \)
  
  - Otherwise: Empty set
**Defining the Analysis (3)**

- **Boundary condition:** Entry node starts with all variables undefined
  - Special "statement" for undefined variables: ?
  - $RD_{entry}(entryNode) = \{(v, ?) | v \in Vars\}$

- **Initially, all nodes have no reaching definitions**
Data flow equations

**RDentry (1)** = \{ (x, ?), (y, ?) \}

**RDentry (2)** = **RDexit (1)**

**RDentry (3)** = **RDexit (2)** \ U \ **RDexit (5)**

**RDentry (4)** = **RDexit (3)**

**RDentry (5)** = **RDexit (4)**

**RDexit (1)** = \( (\text{RDentry (1)} \setminus \{ (x, 1), (x, 5), (x, ?) \}) \)

\ U \ \{ (x, 1) \}

**RDexit (2)** = \( (\text{RDentry (2)} \setminus \{ (y, 2), (y, 4), (y, ?) \}) \)

\ U \ \{ (y, 2) \}

**RDexit (3)** = **RDentry (3)**

**RDexit (4)** = \( (\text{RDentry (4)} \setminus \{ (y, 2), (y, 4), (y, ?) \}) \)

\ U \ \{ (y, 4) \}

**RDexit (5)** = \( (\text{RDentry (5)} \setminus \{ (x, 1), (x, 5), (x, ?) \}) \)

\ U \ \{ (x, 5) \}

**Solution**

<table>
<thead>
<tr>
<th>\text{RDentry (1)}</th>
<th>\text{RDexit (1)}</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ (x, ?), (y, ?) }</td>
<td>{ (x, 1), (y, ?) }</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>\text{RDentry (2)}</th>
<th>\text{RDexit (2)}</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ (x, ?), (y, ?) }</td>
<td>{ (y, 2), (y, 4), (y, ?) }</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>\text{RDentry (3)}</th>
<th>\text{RDexit (3)}</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ (x, 1), (y, 2), (y, 4), (y, ?) }</td>
<td>{ (x, 5) }</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>\text{RDentry (4)}</th>
<th>\text{RDexit (4)}</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ (x, 1), (y, 4) }</td>
<td>{ (y, 1), (y, 4) }</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>\text{RDentry (5)}</th>
<th>\text{RDexit (5)}</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ (x, 1), (x, 5), (x, ?) }</td>
<td>{ (y, 2), (y, 4), (y, ?) }</td>
</tr>
</tbody>
</table>
Outline

- First example: Available expressions
- Basic principles
- More examples
- Solving data flow problems
- Inter-procedural analysis
- Sensitivities
Very Busy Expression Analysis

Goal: For each program point, find expressions that must be very busy

■ ”Very busy”: On all future paths, expression will be used before any of the variables in it are redefined

■ Useful for program optimizations, e.g., hoisting

□ Hoisting an expression: Pre-compute it, e.g., before entering a block, for later use
Example

```java
if (a > b) {
    x = b - a;
    y = a - b;
} else {
    y = b - a;
    x = a - b;
}
```
Example

if (a > b) {
    x = b - a;
    y = a - b;
} else {
    y = b - a;
    x = a - b;
}
Defining the Analysis

- **Domain**: All non-trivial expressions occurring in the code
- **Direction**: Backward
- **Meet operator**: Intersection
  - Because we care about very busy expressions that *must* be used
Defining the Analysis (2)

Transfer function:

\[ V_{B_{\text{entry}}}(s) = (V_{B_{\text{exit}}}(s) \setminus \text{kill}(s)) \cup \text{gen}(S) \]

- Backward analysis: Returns expressions that are very busy expressions at entry of statement
- Function \( \text{gen}(s) \)
  - All expressions \( e \) that appear in \( s \)
- Function \( \text{kill}(s) \)
  - If \( s \) assigns to \( x \), all expressions in which \( x \) occurs
  - Otherwise: Empty set
Defining the Analysis (3)

- **Boundary condition**: Final node starts with no very busy expressions
  \[ V_{B_{exit}}(finalNode) = \emptyset \]
- **Initially, all nodes have no very busy expressions**
Example: Very busy expr. analysis

```
<table>
<thead>
<tr>
<th>s</th>
<th>gen(s)</th>
<th>hlu(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Ø</td>
<td>Ø</td>
</tr>
<tr>
<td>2</td>
<td>{b-a}</td>
<td>Ø</td>
</tr>
<tr>
<td>3</td>
<td>{a-b}</td>
<td>Ø</td>
</tr>
<tr>
<td>4</td>
<td>{b-a}</td>
<td>Ø</td>
</tr>
<tr>
<td>5</td>
<td>{a-b}</td>
<td>Ø</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>s</th>
<th>VBentry(s)</th>
<th>VBexit(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{a-b, b-a}</td>
<td>{a-b, b-a}</td>
</tr>
<tr>
<td>2</td>
<td>{a-b, b-a}</td>
<td>{a-b}</td>
</tr>
<tr>
<td>3</td>
<td>{a-b}</td>
<td>Ø</td>
</tr>
<tr>
<td>4</td>
<td>{a-b, b-a}</td>
<td>{a-b}</td>
</tr>
<tr>
<td>5</td>
<td>{a-b}</td>
<td>Ø</td>
</tr>
</tbody>
</table>
```
Live Variables Analysis

Goal: For each statement, find variables that are *may be “live”* at the exit from the statement

- "Live": The variable is *used before being redefined*
- Useful, e.g., for identifying *dead code*
  - Bug detection: Dead assignments are typically unexpected
  - Optimization: Remove dead code
Example

```java
x = 2;
y = 4;
x = 1;
if (y > x) {
    z = y;
} else {
    z = y * y;
x = z;
}
```
Example

\[
x = 2; \\
y = 4; \\
x = 1; \\
if (y > x) \{ \\
    z = y; \\
\} \text{ else } \{ \\
    z = y \times y; \\
x = z; \\
\}
\]

\text{x is not live after this statement}
Example

\[
x = 2;
y = 4;
\textbf{x} = 1;
\text{if} \ (y > x) \ \{ \\
\quad z = y; \\
\} \ \text{else} \ \{ \\
\quad z = y \times y; \\
\quad x = z; \\
\} 
\]

Both \( x \) and \( y \) are live after this statement.
Defining the Analysis

- **Domain**: All variables occurring in the code
- **Direction**: Backward
- **Meet operator**: Union
  
  - Because we care about whether a variable *may* be used
Defining the Analysis (2)

Transfer function:

\[ LV_{entry}(s) = (LV_{exit}(s) \setminus \text{kill}(s)) \cup \text{gen}(S) \]

- Backward analysis: Returns set of variables that are live at entry of statement

- Function \( \text{gen}(s) \)
  - All variables \( v \) that are used in \( s \)

- Function \( \text{kill}(s) \)
  - If \( s \) assigns to \( x \), then it kills \( x \)
  - Otherwise: Empty set
Defining the Analysis (3)

- **Boundary condition**: Final node starts with no live variables
  - \(LV_{exit}(finalNode) = \emptyset\)

- **Initially, all nodes have no live variables**
Quiz: Live Variables

\begin{verbatim}
x = 2;
y = 4;
x = 1;
if (y > x) {
    z = y;
} else {
    z = y * y;
    x = z;
}
\end{verbatim}

Compute the live variables before and after every statement.
<table>
<thead>
<tr>
<th>s</th>
<th>$LV_{entry}(s)$</th>
<th>$LV_{exit}(s)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>2</td>
<td>$\emptyset$</td>
<td>${5}$</td>
</tr>
<tr>
<td>3</td>
<td>${5}$</td>
<td>${x,5}$</td>
</tr>
<tr>
<td>4</td>
<td>${x,y}$</td>
<td>${5}$</td>
</tr>
<tr>
<td>5</td>
<td>${5}$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>6</td>
<td>${5}$</td>
<td>${z}$</td>
</tr>
<tr>
<td>7</td>
<td>${z}$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>