Generalizing Data-flow Analysis

Announcements
- Read the data-flow analysis handout

Today
- Other types of data-flow analysis
  - Reaching definitions, available expressions, reaching constants
  - Abstracting data-flow analysis
- What’s common among the different analyses?

Reaching Definitions

Definition
- A definition (statement) \( d \) of a variable \( v \) reaches node \( n \) if there is a path from \( d \) to \( n \) such that \( v \) is not redefined along that path

Uses of reaching definitions
- Build use/def chains
- Constant propagation
- Loop invariant code motion

A Better Formulation of Reaching Definitions

Problem
- Reaching definitions gives you a set of definitions (nodes)
- Doesn’t tell you what variable is defined
- Expensive to find definitions of variable \( v \)

Solution
- Reformulate to include variable
  \( \text{e.g., } \) Use a set of \( (\text{var}, \text{def}) \) pairs

Computing Reaching Definitions

Assumption
- At most one definition per node
- We can refer to definitions by their node “number”

Gen\( [n] \): Definitions that are generated by node \( n \) (at most one)
Kill\( [n] \): Definitions that are killed by node \( n \)

Defining Gen and Kill for various statement types

<table>
<thead>
<tr>
<th>statement</th>
<th>Gen( [s] )</th>
<th>Kill( [s] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s: \text{t = b op c} )</td>
<td>{s}</td>
<td>{s, def[t] } - {s}</td>
</tr>
<tr>
<td>( s: \text{t = M[b]} )</td>
<td>{s}</td>
<td>{s, def[t] } - {s}</td>
</tr>
<tr>
<td>( s: \text{M[a] = b} )</td>
<td>{s}</td>
<td>{s}</td>
</tr>
<tr>
<td>( s: \text{if a op b goto L} )</td>
<td>{s}</td>
<td>{s}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>statement</th>
<th>Gen( [s] )</th>
<th>Kill( [s] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s: \text{goto L} )</td>
<td>{s}</td>
<td>{}</td>
</tr>
<tr>
<td>( s: \text{L:} )</td>
<td>{s}</td>
<td>{}</td>
</tr>
<tr>
<td>( s: \text{f(a, ...)} )</td>
<td>{s}</td>
<td>{}</td>
</tr>
<tr>
<td>( s: \text{t=f(a, ..., s) de[t] = [s]} )</td>
<td>{s}</td>
<td>{s}</td>
</tr>
</tbody>
</table>
Recall Liveness Analysis

**Definition**
- A variable is *live* at a particular point in the program if its value at that point will be used in the future (dead, otherwise).

**Uses of Liveness**
- Register allocation
- Dead-code elimination

```
1 a = . . .;
2 b = . . .;
3 ...
4 x = f(b);
```

Available Expressions

**Definition**
- An expression, \( x+y \), is *available* at node \( n \) if every path from the entry node to \( n \) evaluates \( x+y \), and there are no definitions of \( x \) or \( y \) after the last evaluation.

```
...x+y...
...
...x+y...
...x+y...
```

Available Expressions for CSE

**How is this information useful?**

**Common Subexpression Elimination (CSE)**
- If an expression is available at a point where it is evaluated, it need not be recomputed

**Example**

```
1 i := j
   a := 4 * i

2 i := i + 1
   b := 4 * i

3 c := 4 * i
```

Aspects of Data-flow Analysis

**Must or may Information**
- guaranteed or possible

**Direction**
- forward or backward

**Flow values**
- variables, definitions, ...

**Initial guess**
- universal or empty set

**Kill**
- due to semantics of stmt what is removed from set

**Gen**
- due to semantics of stmt what is added to set

**Merge**
- how sets from two control paths compose
### Must vs. May Information

<table>
<thead>
<tr>
<th>Safe</th>
<th>Overly large set</th>
<th>Overly small set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Desired information</td>
<td>Small set</td>
<td>Large set</td>
</tr>
<tr>
<td>Gen</td>
<td>Add everything that might be true</td>
<td>Add only facts that are guaranteed to be true</td>
</tr>
<tr>
<td>Kill</td>
<td>Remove only facts that are guaranteed to be true</td>
<td>Remove everything that might be false</td>
</tr>
<tr>
<td>Merge</td>
<td>Union</td>
<td>Intersection</td>
</tr>
<tr>
<td>Initial guess</td>
<td>Empty set</td>
<td>Universal set</td>
</tr>
</tbody>
</table>

### Defining Available Expressions Analysis

<table>
<thead>
<tr>
<th>Must or may Information?</th>
<th>Must</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direction?</td>
<td>Forward</td>
</tr>
<tr>
<td>Flow values?</td>
<td>Sets of expressions</td>
</tr>
<tr>
<td>Initial guess?</td>
<td>Universal set</td>
</tr>
<tr>
<td>Kill?</td>
<td>Set of expressions killed by statement s</td>
</tr>
<tr>
<td>Gen?</td>
<td>Set of expressions evaluated by s</td>
</tr>
<tr>
<td>Merge?</td>
<td>Intersection</td>
</tr>
</tbody>
</table>

### Reaching Definitions: Must or May Analysis?

Consider constant propagation

![Diagram showing control flow and reaching definitions](image)

---

### Reaching Constants

**Goal**
- Compute value of each variable at each program point (if possible)

**Flow values**
- Set of (variable, constant) pairs

**Merge function**
- Intersection

**Data-flow equations**
- Effect of node \( n \ x = c \)
  - \( \text{kill}[n] = \{(x,d) \mid \forall d \} \)
  - \( \text{gen}[n] = \{(x,c)\} \)
- Effect of node \( n \ x = y + z \)
  - \( \text{kill}[n] = \{(x,c) \mid \forall c \} \)
  - \( \text{gen}[n] = \{(x,c) \mid c = \text{valy} + \text{valz}, (y, \text{valy}) \in \text{in}[n], (z, \text{valz}) \in \text{in}[n] \} \)
Reality Check!

Some definitions and uses are ambiguous

- We can’t tell whether or what variable is involved
  
  \[ \text{e.g., } *p = x; /* what variable are we assigning?! */ \]
  
  - Unambiguous assignments are called strong updates
  - Ambiguous assignments are called weak updates

Solutions

- Be conservative
  
  - Sometimes we assume that it could be everything
    
    \[ \text{e.g., Defining } *p \text{ (generating reaching definitions)} \]
  
  - Sometimes we assume that it is nothing
    
    \[ \text{e.g., Defining } *p \text{ (killing reaching definitions)} \]
  
  - Try to figure it out: alias/pointer analysis (more later)

Concepts

Data-flow analyses are distinguished by

- Flow values (initial guess, type)
  
  - May/must
  - Direction
  - Gen
  - Kill
  - Merge

Complication

- Ambiguous references (strong/weak updates)

Next Time

Lecture

- Lattice theoretic foundation for data-flow analysis