Generalizing Data-flow Analysis

Announcements
– Read Section 9.3 in the book

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
– abstracting data-flow analysis

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

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

\[
\begin{align*}
1 & \quad a = \ldots; \\
2 & \quad b = \ldots; \\
3 & \\
4 & \quad x = f(b) ; \\
\end{align*}
\]

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.

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
\[
\begin{align*}
1 & \quad i := 3 \\
2 & \quad a := 4 * i
\end{align*}
\]

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></th>
<th>May</th>
<th>Must</th>
</tr>
</thead>
<tbody>
<tr>
<td>safe</td>
<td>overly large set</td>
<td>overly small set</td>
</tr>
<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>

**Liveness? Available expressions?**

- **Must information**
  - Implies a guarantee
- **May information**
  - Identifies possibilities

**Liveness? Available expressions?**

- **Must**: Forward
  - Sets of expressions
  - Universal set
  - Set of expressions killed by statement s
  - Set of expressions evaluated by s
  - Intersection

**Reaching Definitions: Must or May Analysis?**

Consider uses of reaching definitions

![Diagram showing reaching definitions](image)

**Defining Available Expressions Analysis**

- **Must or may Information?** Must
- **Direction?** Forward
- **Flow values?** Sets of expressions
- **Initial guess?** Universal set
- **Kill?** Set of expressions killed by statement s
- **Gen?** Set of expressions evaluated by s
- **Merge?** Intersection

**Reaching Constants (aka Constant Propagation)**

**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 \( n \ x = y + z \)
  - \( \text{kill}[n] = \{(x,c) \mid y = \text{val}(y) + \text{val}(z), (y,\text{val}(y)) \in \text{in}[n], (z,\text{val}(z)) \in \text{in}[n]\} \)
Reaching Constants Example

Must or may info?

Direction?

Initial guess?

Reality Check!

Some definitions and uses are ambiguous
- We can’t tell whether or what variable is involved
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
e.g., Defining *p (generating reaching definitions)
- Sometimes we assume that it is nothing
e.g., Defining *p (killing reaching definitions)
- Try to figure it out: alias/pointer analysis (more later)

Side Effects

What happens at function calls?
- For example, the call foo(&x) might use or define
  - any local or heap variable x that has been passed by address/reference
  - any global variable

Solution
- How do we handle this for liveness used for register allocation?
- In general
  - Be conservative: assume all globals and all vars passed by address/reference may be used and/or modified
  - Figure it out: calculate side effects (example of an interprocedural analysis)

Concepts

Data-flow analyses are distinguished by
- Flow values (initial guess, type)
- May/must
- Direction
- Gen
- Kill
- Merge

Complication
- Ambiguous references (strong/weak updates)
- Side effects
Next Time

Lecture
- Lattice theoretic foundation for data-flow analysis

Suggested Exercises
- exercises from book: 9.2.1, 9.2.2, 9.2.6