Generalizing Data-flow Analysis

Last Time
– Introduction to data-flow analysis

Today
– Other examples of data-flow analysis
– Abstracting data-flow analysis
  – What’s common among these different analyses?

Time Complexity

Consider a program of size N
– Has N nodes in the flow graph and at most N variables
– Each live-in or live-out set has at most N elements
– Each set-union operation takes $O(N)$ time
– The for loop body
  – constant # of set operations per node
  – $O(N)$ nodes $\Rightarrow O(N^2)$ time for the loop
– Each iteration of the repeat loop can only make the set larger
– Each set can contain at most N variables $\Rightarrow 2N^2$ iterations

Worst case: $O(N^3)$
Typical case: 2 to 3 iterations with good ordering & sparse sets
  $\Rightarrow O(N)$ to $O(N^2)$
More Performance Considerations

**Basic blocks**
- Decrease the size of the CFG by merging nodes that have a single predecessor and a single successor into **basic blocks**

**One variable at a time**
- Instead of computing data-flow information for all variables at once using sets, compute a (simplified) analysis for each variable separately

**Representation of sets**
- For dense sets, use a bit vector representation
- For sparse sets, use a sorted list (e.g., linked list)

Conservative Approximation

<table>
<thead>
<tr>
<th>node #</th>
<th>use</th>
<th>def</th>
<th>X</th>
<th>in</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td></td>
<td></td>
<td>c</td>
<td>ac</td>
</tr>
<tr>
<td>2</td>
<td>a</td>
<td>b</td>
<td></td>
<td>ac</td>
<td>bc</td>
</tr>
<tr>
<td>3</td>
<td>bc</td>
<td>c</td>
<td></td>
<td>bc</td>
<td>bc</td>
</tr>
<tr>
<td>4</td>
<td>b</td>
<td>a</td>
<td></td>
<td>bc</td>
<td>ac</td>
</tr>
<tr>
<td>5</td>
<td>a</td>
<td></td>
<td></td>
<td>ac</td>
<td>ac</td>
</tr>
<tr>
<td>6</td>
<td>c</td>
<td></td>
<td></td>
<td>c</td>
<td></td>
</tr>
</tbody>
</table>

**Solution X**
- Our solution as computed on previous slides

February 2, 2015 Data-flow Analysis 3

February 2, 2015 Data-flow Analysis 4
### Conservative Approximation (cont)

#### Solution Y
- Carries variable d uselessly around loop
- Does Y solve the equations? Yes
- Is d live? No
- Does Y lead to a correct program? Yes

Imprecise conservative solutions $\Rightarrow$ sub-optimal but correct programs

#### Solution Z
- Does not identify c as live in all cases
- Does Z solve the equations? No
- Does Z lead to a correct program? No

Non-conservative solutions $\Rightarrow$ incorrect programs
The Need for Approximations

Static vs. Dynamic Liveness
- In the following graph, \( b^2 \) is always non-negative, so \( c \geq b \) is always true and \( a \)'s value will never be used after node 2.

1. \( a := b \times b \)
2. \( c := a + b \)
3. \( c \geq b ? \)
4. \( \text{return } a \)
5. \( \text{return } c \)

Rule (2) for computing liveness
- Since \( a \) is live-in at node 4, it is live-out at nodes 3 and 2
- This rule ignores actual control flow

No compiler can statically know all of a program’s dynamic properties!

Generalizing Data-flow Analysis

Last Time
Introduction to data-flow analysis

Today
- Other examples of data-flow analysis
- Abstracting data-flow analysis
- What’s common among these 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

To determine whether it's legal to move statement 4 out of the loop, we need to ensure that there are no reaching definitions of \( a \) or \( b \) inside the loop.

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

February 4, 2015  Generalizing Data-flow Analysis  9
**Data-flow Equations for Reaching Definitions**

**The In set**
- A definition reaches the beginning of a node if it reaches the end of any of the predecessors of that node

**The Out set**
- A definition reaches the end of a node if (1) the node itself generates the definition or if (2) the definition reaches the beginning of the node and the node does not kill it

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

**Recall Liveness Analysis**

**Data-flow equations for liveness**
\[
\begin{align*}
\text{in}[n] &= \text{use}[n] \cup (\text{out}[n] - \text{def}[n]) \\
\text{out}[n] &= \bigcup_{s \in \text{succ}[n]} \text{in}[s]
\end{align*}
\]

**Liveness equations in terms of Gen and Kill**
\[
\begin{align*}
\text{in}[n] &= \text{gen}[n] \cup (\text{out}[n] - \text{kill}[n]) \\
\text{out}[n] &= \bigcup_{s \in \text{succ}[n]} \text{in}[s]
\end{align*}
\]

- A use of a variable generates liveness
- A def of a variable kills liveness

**Can define almost any data-flow analysis in terms of Gen and Kill**

**Gen:** New information that’s added at a node
**Kill:** Old information that’s removed at a node
Direction of Flow

Backward data-flow analysis
- Information at a node is based on what happens later in the flow graph
  i.e., \(\text{in}[n]\) is defined in terms of \(\text{out}[n]\)
  \[
  \text{in}[n] = \text{gen}[n] \cup (\text{out}[n] - \text{kill}[n])
  \]
  \[
  \text{out}[n] = \bigcup_{s \in \text{succ}[n]} \text{in}[s]
  \]

Forward data-flow analysis
- Information at a node is based on what happens earlier in the flow graph
  i.e., \(\text{out}[n]\) is defined in terms of \(\text{in}[n]\)
  \[
  \text{in}[n] = \bigcup_{p \in \text{pred}[n]} \text{out}[p]
  \]
  \[
  \text{out}[n] = \text{gen}[n] \cup (\text{in}[n] - \text{kill}[n])
  \]

Some problems need both forward and backward analysis (rare)
- e.g., Partial redundancy elimination

Data-flow Equations for Reaching Definitions

Symmetry between reaching definitions and liveness
- Swap \(\text{in}[]\) and \(\text{out}[]\) and swap the directions of the arcs

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

Liveness
- \(\text{out}[n] = \bigcup_{s \in \text{succ}[n]} \text{in}[s]\)
- \(\text{in}[n] = \text{gen}[n] \cup (\text{out}[n] - \text{kill}[n])\)
Liveness vs. Reaching Definitions

What are the key differences?
- Direction of flow
- Flow values
  - Liveness operates on variables
  - Reaching definitions operates on definitions (statements)

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 \( v \) (Why?)

Solution
- Reformulate to include the variable
  e.g., Use a set of (var, def) pairs

\[
\text{a} \xrightarrow{x=a} \text{b} \xrightarrow{y=b} \text{c} \xrightarrow{\text{in}[n]} (x,a),(y,b),...)
\]
Merging Flow Values

Liveness and reaching definitions
- Merge \textit{flow values} via set union

\textbf{Reaching Definitions} \hspace{1cm} \textbf{Liveness}
\begin{align*}
in[n] &= \bigcup_{p \in \text{pred}(n)} \text{out}[s] \\
\text{out}[n] &= \text{gen}[n] \cup (\text{in}[n] - \text{kill}[n])
\end{align*}
\begin{align*}
\text{out}[n] &= \bigcup_{s \in \text{succ}[n]} \text{in}[s] \\
\text{in}[n] &= \text{gen}[n] \cup (\text{out}[n] - \text{kill}[n])
\end{align*}

Why Use Union?
When might this be inappropriate?

Available Expressions

\textbf{Definition}
- An expression, $x+y$, is \textit{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

\begin{tikzpicture}
  \node[draw] (entry) at (0,0) {entry};
  \node[draw] (n) at (2,0) {$n$};
  \node[draw] (x) at (2,-1) {$x$};
  \node[draw] (y) at (2,-2) {$y$};
  \draw[->] (entry) -- (n);
  \draw[->] (n) -- (x);
  \draw[->] (n) -- (y);
  \draw[->] (x) -- (y);
\end{tikzpicture}
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

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

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

```
c := 4 * i
```

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

```
i := i + 1
t := 4 * i
b := t
```

```
c := t
```

Must vs. May Information

**Must information**
– Implies a guarantee

**May information**
– Identifies possibilities

<table>
<thead>
<tr>
<th>Liveness?</th>
<th>Available expressions?</th>
</tr>
</thead>
<tbody>
<tr>
<td>May</td>
<td>Must</td>
</tr>
<tr>
<td>safe</td>
<td>overly large set</td>
</tr>
<tr>
<td>desired information</td>
<td>small set</td>
</tr>
<tr>
<td>Gen</td>
<td>add everything that might be true</td>
</tr>
<tr>
<td>Kill</td>
<td>remove only facts that are guaranteed to be false</td>
</tr>
<tr>
<td>merge</td>
<td>union</td>
</tr>
<tr>
<td>initial guess</td>
<td>empty set</td>
</tr>
</tbody>
</table>
Reaching Definitions: Must or May Analysis?

Consider constant propagation

We need to know if d’ might reach node n

Exercise: Define 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
Available Expressions (cont)

**Data-Flow Equations**

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

**Plug it in to our general DFA algorithm**

for each node \( n \)

\[
\text{in}[n] = \emptyset; \quad \text{out}[n] = \emptyset
\]

repeat

for each \( n \)

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

until \( \text{in}'[n]=\text{in}[n] \) and \( \text{out}'[n]=\text{out}[n] \) for all \( n \)

Exercise: Reaching Constants

**Goal**

– Compute value of each variable at each program point (if possible)

**Flow values**

– Set of (variable,constant) pairs

**Merge function**

– Intersection

**Gen and Kill**

– Effect of node \( n \) \( x = c \)
  – \( \text{kill}[n] = \{(x,d) | \forall d\} \)
  – \( \text{gen}[n] = \{(x,c)\} \)

– Effect of node \( n \) \( x = y + z \)
  – \( \text{kill}[n] = \{(x,c) | \forall c\} \)
  – \( \text{gen}[n] = \{(x,c) | c=\text{val}y+\text{val}z, (y, \text{val}y) \in \text{in}[n], (z, \text{val}z) \in \text{in}[n]\} \)