Lecture Outline

1. Review
   • The DAG representation of basic blocks
   • Finding local common subexpressions
   • The use of algebraic identities
   • Peephole optimizations

2. The Data-Flow Analysis Schema
   • A data-flow value at a program point represents the set of all possible programs states that can be observed for that point, for example, all definitions in the program that can reach that point.
   • Let IN[s] and OUT[s] be the set of data-flow values before and after a statement s in a program.
   • A transfer function \( f_s \) relates the data-flow values before and after a statement s.
   • In a forward data-flow problem
     \[
     \text{OUT}[ s ] = f_s(\text{IN}[ s ])
     \]
   In a backward data-flow problem
     \[
     \text{IN}[ s ] = f_s(\text{OUT}[ s ])
     \]
   • A transfer function can be extended to a basic block by composing the transfer functions for all the statements in the block. Thus in a forward data-flow problem such as reaching definitions for a block \( B \),
     \[
     \text{OUT}[ B ] = f_B(\text{IN}[ B ])
     \]
Given a flow graph, in a forward data-flow problem the IN set of a basic block \( B \) is computed from the OUT sets of \( B \)'s predecessors:

\[
\text{IN}[B] = \bigcup_{P \text{ a predecessor of } B} \text{OUT}[P]
\]

In a backward data-flow problem such as live variable analysis:

\[
\text{IN}[B] = f_B(\text{OUT}[B])
\]

\[
\text{OUT}[B] = \bigcup_{S \text{ a successor of } B} \text{IN}[S]
\]

The data-flow problem for a flow graph is to compute the values of \( \text{IN}[B] \) and \( \text{OUT}[B] \) for all blocks \( B \) in the flow graph.

3. Reaching Definitions

A definition \( d \) reaches a program point \( p \) if there is a path from the point immediately following \( d \) to \( p \) such that \( d \) is not killed along that path.

Flow graph with \( \text{gen} \) and \( \text{kill} \) sets for each basic block:

```
ENTRY

B1

d1: i = m - 1
\text{gen}_{B1} = \{d1, d2, d3\}
\text{kill}_{B1} = \{d4, d5, d6, d7\}

B2

d4: i = i + 1
\text{gen}_{B2} = \{d4, d5\}
\text{kill}_{B2} = \{d1, d2, d7\}

B3

d2: j = n
\text{d3: a = u1}
\text{d4: i = i + 1}
\text{d5: j = j - 1}
\text{d6: a = u2}
\text{d7: i = u3}

B3

B4

\text{gen}_{B4} = \{d7\}
\text{kill}_{B4} = \{d1, d4\}

EXIT
```
• \( \text{gen}_B \) contains all definitions in block \( B \) that are visible immediately after block \( B \).

• \( \text{kill}_B \) is the union of all definitions killed by the statements in block \( B \).

4. Control-Flow Equations for Reaching Definitions

• The reaching definitions problem is defined by the following control-flow equations:

\[
\text{OUT}[\text{ENTRY}] = \text{empty} \_ \text{set}
\]

For all blocks \( B \) other than \( \text{ENTRY} \):

\[
\text{OUT}[B] = \text{\text{gen}}_B \cup (\text{IN}[B] - \text{kill}_B)
\]

\[
\text{IN}[B] = \bigcup_{\text{a predecessor of } B} \text{OUT}[P]
\]

5. Iterative Algorithm for Reaching Definitions

• Given a flow graph for which the \( \text{gen} \) and \( \text{kill} \) sets have been computed for each block, we can compute the set of definitions reaching the entry and exit of each block \( B \) using the following iterative algorithm:

\[
\text{OUT}[\text{ENTRY}] = \text{empty} \_ \text{set};
\]

\[
\text{for (each basic block } B \text{ other than ENTRY) }
\]

\[
\text{OUT}[B] = \text{empty} \_ \text{set};
\]

\[
\text{while (changes to any OUT occur)}
\]

\[
\text{for (each basic block } B \text{ other than ENTRY) }
\]

\[
\text{IN}[B] = \bigcup_{\text{a predecessor of } B} \text{OUT}[P];
\]

\[
\text{OUT}[B] = \text{\text{gen}}_B \cup (\text{IN}[B] - \text{kill}_B);
\]

• Example: Let us represent a set of definitions in the flow graph above by a bit vector. Thus \( 1110000 \) represents the set \( \{ d_1, d_2, d_3 \} \). The following table represents the values taken on by the IN and OUT sets after each iteration of the while-loop of this algorithm. The superscript denotes the iteration. The initial values of OUT, computed by the second statement of the algorithm, are indicated by the superscript 0.
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>000 0000</td>
<td>000 0000</td>
<td>111 0000</td>
<td>000 0000</td>
<td>111 0000</td>
</tr>
<tr>
<td>B2</td>
<td>000 0000</td>
<td>111 0000</td>
<td>001 1100</td>
<td>111 0111</td>
<td>001 1110</td>
</tr>
<tr>
<td>B3</td>
<td>000 0000</td>
<td>001 1100</td>
<td>000 1110</td>
<td>001 1110</td>
<td>000 1110</td>
</tr>
<tr>
<td>B4</td>
<td>000 0000</td>
<td>001 1110</td>
<td>001 0111</td>
<td>001 1110</td>
<td>001 0111</td>
</tr>
<tr>
<td>EXIT</td>
<td>000 0000</td>
<td>001 0111</td>
<td>001 0111</td>
<td>001 0111</td>
<td>001 0111</td>
</tr>
</tbody>
</table>

IN and OUT sets for the basic blocks of the flowgraph in Section 2.

6. Live-Variable Analysis

- In live-variable analysis we want to determine for each variable \( x \) and each program point \( p \) whether the value \( x \) at \( p \) could be used along some path in the flow graph starting at \( p \). If so, we say \( x \) is live at \( p \); if not, \( x \) is dead at \( p \). Live-variable analysis is crucial for register allocation.

- Live-variable analysis is an example of a backwards data-flow problem.

- Define \( \text{def}_B \) as the set of variables defined in \( B \) prior to any use of that variable in \( B \). In the flow graph above \( \text{def}_{B2} = \{ i, j \} \).

  Define \( \text{use}_B \) as the set of variables whose values may be used in \( B \) prior to any definition of the variable. In the flow graph above \( \text{use}_{B2} = \{ i, j \} \).

- Data-flow equations for live-variable analysis:

  \[
  \text{IN}[\text{EXIT}] = \text{empty_set}
  \]

  For all blocks \( B \) other than \( \text{EXIT} \):

  \[
  \text{IN}[B] = \text{use}_B \cup (\text{OUT}[B] \setminus \text{def}_B)
  \]

  \[
  \text{OUT}[B] = \bigcup_{S \text{ a successor of } B} \text{IN}[S]
  \]
• Given a flow graph for which the def and use sets have been computed for each block, we can compute the set of variables live on entry and exit of each block $B$ using the following iterative algorithm:

\[
\begin{align*}
\text{IN}[\text{EXIT}] &= \text{empty}\_set; \\
\text{for (each basic block } B \text{ other than EXIT)} & \\
\text{IN}[B] &= \text{empty}\_set; \\
\text{while (changes to any IN occur)} & \\
& \quad \text{for (each basic block } B \text{ other than EXIT)} \\
& \quad \quad \text{OUT}[B] = \bigcup \text{ of a successor of } B \text{ IN}[S]; \\
& \quad \quad \text{IN}[B] = \text{use}_B \bigcup (\text{OUT}[B] - \text{def}_B); \\
\end{align*}
\]

• Unlike reaching definitions, the information flow for liveness travels backward in the data-flow graph, opposite to the direction of control flow.

• However, as for reaching definitions, live-variable analysis uses union as the “meet” operator. We are interested in whether any path with the desired properties exists, not whether something is true along all paths.

• Also as for reaching definitions, the solution to the data-flow equation is not necessarily unique. We want the solution with the smallest sets of live variables.

7. Reading

• ALSU, Section 9.2.

aho@cs.columbia.edu