Dataflow Analysis

- **Properties**
  - Compile-time reasoning about the run-time flow of values
  - Represents facts about run-time behavior
  - Describes effect of executing each basic block on sets of facts
  - Propagate facts around control flow graph (CFG)

- **Limitations**
  - Answers must be conservative.
  - Often needs to approximate information
  - Assumes all possible paths may be taken

Global Data-flow Problem

- Represents specific facts about run-time behavior
  - Set of facts form a lattice \( L \)
  - Lattice is used to describe relation between values
  - Facts can also be represented as a bit or bit vector

- Within basic blocks
  - Describe effect of "executing" basic block on facts
  - Propagation functions: \( f_{\text{bb}}: L \rightarrow L \)

- Between basic blocks
  - Formulated as a set of simultaneous equations
    - Sets attached to nodes and edges of CFG
  - Solve equations using
    - Iterative framework
    - Methods based on program structure

Classical Data Flow Problems

- **Problems**
  - Reaching definitions (RD)
  - Live uses of variables (LV)
  - Available expressions (AVAIL)
  - Very Busy Expressions (VBE)

- **Def-Use and Use-Def chains, built from RD**
  - AVAIL enables global common subexpression elimination
  - VBE can be used for code motion

Reaching Definitions (RD)

A definition of a variable \( x \) is a statement that may modify the value of variable \( x \).

A definition of a variable \( x \) at node \( k \) reaches node \( n \) if there is a definition-free path from \( k \) to \( n \).

Live Variables (LV)

**Use:**
An appearance of a variable \( x \) as a RHS operand that results in reading its value.

A use of a variable \( x \) is **live on exit** from node \( k \) if there is a definition-clear path for \( x \) from \( k \) to a note \( n \) that uses \( x \).
DU and UD Chains

**UD-Chain:** Links each use of variable \( x \) to definitions which reach that use.

**DU-Chain:** Links each definition of variable \( x \) to those uses which that definition can reach.

Optimizations that can use DU and UD Chains

- **Optimization** Uses
- Dead code elimination (DU)
- Code motion (UD)
- Strength reduction (UD)
- Constant propagation (UD/PU)
- Forward substitution (Copy propagation) (DU)

Classification of Data Flow Problems: Propagation

**IN(B)** : data flow information valid on entry to basic block \( B \)

**OUT(B)** = \( GEN(B) \cup [IN(B) - KILL(B)] \)

**GEN** and **KILL** describe the effect of basic block \( B \) on data flow information

**OUT(B)** : data flow information valid on exit from basic block \( B \)

Classification of Data Flow Problems: Flow Direction

**IN(B)** = \( \wedge (GEN(B) \cup [IN(B) - KILL(B)] \) )

\( Bi \in PREV(B) \)

**IN(B)** : data flow information valid on entry to basic block \( B \)

**OUT(B)** : data flow information valid on exit from basic block \( B \)

Classification of Data Flow Problems: Flow Direction

**For forward data flow problems**
- Solution for basic block applies to beginning of basic block
  - Examples
    - Reaching definitions
    - Available expressions
    - Constant propagation

**For backward data flow problems**
- Solution for basic block applies to the end of basic block
  - Examples
    - Live variables
    - Very busy expressions
Classification of Data Flow Problems: Merging Information

\[ \text{IN}(B_1) \cap \text{IN}(B_2) \cap \ldots \cap \text{IN}(B_n) \]

\[ \bigcup_{B \in \text{IN}(B)} \]

Merging data flow information

Classification of Data Flow Problems: Examples

<table>
<thead>
<tr>
<th></th>
<th>forward</th>
<th>backward</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>may:</strong></td>
<td>there exists a path (union)</td>
<td>( \text{RD} )</td>
</tr>
<tr>
<td><strong>must:</strong></td>
<td>for all paths (intersection)</td>
<td>( \text{AVAIL} )</td>
</tr>
</tbody>
</table>

Reaching Definitions

A definition of a variable \( x \) is a statement that may modify the value of variable \( x \).

A definition of variable \( x \) is killed if \( x \) is reassigned by another definition; the "killing" definition must occur.

\[ \text{RD}(B) = \bigcup_{B_i \in \text{PRE}(B)} (\text{GEN}(B_i) \cup \{ \text{RD}(B_i) - \text{KILL}(B_i) \}) \]

\( \text{GEN}(B_i) \): All definitions in \( B_i \) that are not killed by subsequent definitions in \( B_i \)

\( \text{KILL}(B_i) \): All definitions of variables \( x \) defined in \( B_i \)

Reaching Definitions Example

Universe of facts? All possible subsets of \((B_i, \text{VAR})\)

pairs, where \( B_i \) is a basic block and \( \text{VAR} \) is a program variable

\( \text{GEN} \) and \( \text{KILL} \) sets for each basic block?

Initial values of \( \text{RD}(B) \) before propagation starts?

\( \text{IN}(B_i) = \emptyset \)

Reaching Definitions Example 2

\[ \text{RD}(B) = \bigcup_{B_i \in \text{PRE}(B)} (\text{GEN}(B_i) \cup \{ \text{RD}(B_i) - \text{KILL}(B_i) \}) \]

\( \text{RD}(B_2) = \bigcup_{B_i \in \text{PRE}(B_2)} (\text{GEN}(B_i) \cup \{ \text{RD}(B_i) - \text{KILL}(B_i) \}) \)

\( \text{RD}(B_3) = \bigcup_{B_i \in \text{PRE}(B_3)} (\text{GEN}(B_i) \cup \{ \text{RD}(B_i) - \text{KILL}(B_i) \}) \)

\( \text{RD}(B_4) = \bigcup_{B_i \in \text{PRE}(B_4)} (\text{GEN}(B_i) \cup \{ \text{RD}(B_i) - \text{KILL}(B_i) \}) \)

Round Robin Iterative Algorithm (Forward Problem)

for \( j = 1, n \) do initialize \( \text{IN}(B_j) \) with \( T \); change := true;

while ( change )

\( \text{new} := \bigwedge \{ (\text{GEN}(B_i) \cup \{ \text{IN}(B_i) - \text{KILL}(B_i) \}) \} \)

if ( new \# \text{IN}(B_j) )

\( \text{IN}(B_j) := \text{new}, \text{change} := \text{true} \)

}
POSTORDER and Reverse POSTORDER

Step 1: POSTORDER

- Main()
  count = 1;
  Visit (root)

- Visit(n)
  mark n as visited
  for each successor s of n not yet visited
    Visit(s);
  \[ \text{POSTORDER}(n) = \text{count}; \]
  \[ \text{count} = \text{count} + 1; \]

Step 2: rPOSTORDER

- For each node n
  \[ \text{rPOSTORDER}(n) = \text{NumNodes} + 1 - \text{POSTORDER}(n) \]

Iterative Data-flow Solver

- Iterates over each basic block
  - Solving data-flow equations
  - Until data-flow solutions converge

- Data-flow solutions converge faster if
  - Compute solutions for predecessors before current node
    - Since solution for node only changes if solution for some predecessor changes (for forward problem)

- Visiting nodes in reverse postorder
  - More likely to solve predecessors first
  - Not possible to always solve all predecessors (e.g., loops)