Lecture 4
Introduction to Data Flow Analysis

I  Structure of data flow analysis
II  Example 1: Reaching definition analysis
III  Example 2: Liveness analysis
IV  Generalization

Reference: Chapter 8, 8.1-4

Data Flow Analysis

• Local analysis (e.g. value numbering)
  - analyze effect of each instruction
  - compose effects of instructions to derive information
    from beginning of basic block to each instruction

• Data flow analysis
  - analyze effect of each basic block
  - compose effects of basic blocks to derive information
    at basic block boundaries
  - (from basic block boundaries,
    apply local technique to generate information on instructions)

Effects of a basic block

• Effect of a statement: a = b+c
  - Uses variables (b, c)
  - Kills an old definition (old definition of a)
  - new definition (a)

• Compose effects of statements -> Effect of a basic block
  - A locally exposed use in a b.b. is a use of a data item which is
    not preceded in the b.b. by a definition of the data item
  - any definition of a data item in the basic block kills all definitions
    of the same data item reaching the basic block.
  - A locally available definition = last definition of data item in b.b.

Across Basic Blocks

• Static program vs. dynamic execution

  • Statically: Finite program
    Dynamically: Potentially infinite possible execution paths
  • Can reason about each possible path
    as if all instructions executed are in one basic block
  • Data flow analysis:
    Associate with each static point in the program
    information true of
    the set of dynamic instances of that program point

Effects of a basic block

- a = b+c
- Uses variables (b, c)
- Kills an old definition (old definition of a)
- new definition (a)
- Compose effects of statements -> Effect of a basic block
- A locally exposed use in a b.b. is a use of a data item which is
  not preceded in the b.b. by a definition of the data item
- any definition of a data item in the basic block kills all definitions
  of the same data item reaching the basic block.
- A locally available definition = last definition of data item in b.b.

Across Basic Blocks

- Static program vs. dynamic execution

  - Statically: Finite program
    Dynamically: Potentially infinite possible execution paths
  - Can reason about each possible path
    as if all instructions executed are in one basic block
  - Data flow analysis:
    Associate with each static point in the program
    information true of
    the set of dynamic instances of that program point
II. Reaching Definitions

- A definition of a variable $x$ is a statement that assigns, or may assign, a value to $x$.
- A definition $d$ reaches a point $p$ if there exists a path from the point immediately following $d$ to $p$ such that $d$ is not killed along that path.

II.1 Reaching Definitions

- Problem statement
  - For each basic block $b$, determine if each definition in the program reaches $b$.
  - A representation:
    - $IN[b]$, $OUT[B]$: a bit vector, one bit for each definition

A1.3.1 Example

- $a = x$
- $b = a$
- $a = y$
- $d_1: a = 10$
- $d_2: b = 11$
- $if$ $e$
- $d_3: a = 1$
- $d_4: b = 2$
- $d_5: c = a$
- $d_6: a = 4$

II.2 Describing Effects of the Nodes (basic blocks)

- A transfer function $f_b$ of a basic block $b$:
  - $OUT[b] = f_b(IN[b])$
- incoming reaching definitions -> outgoing reaching definitions

- A basic block $b$
  - generate definitions: $Gen[b]$, set of locally available definitions in $b$
  - propagate definitions: $in[b]$ - $Kill[b]$, where $Kill[b]$=set of defs (in rest of program) killed by defs in $b$
  - $out[b] = Gen[b] U (in(b) - Kill[b])$

- Out[b] = $f_b(IN[b])$

II.3 Effects of the Edges (acyclic)

- Equations still hold
  - $out[b] = f_b(IN[b])$
  - in[b] = out[p_1] U out[p_2] U ... U out[p_n], where $p_1$, ..., $p_n$ are all predecessors of $b$

II.4 Cyclic Graphs

- Equations still hold
  - $out[b] = f_b(IN[b])$
  - $in[b] = out[p_1] U out[p_2] U ... U out[p_n]$, $p_1$, ..., $p_n$ pred.
  - Solve for fixed point solution
Reaching Definitions: Worklist Algorithm

input: control flow graph CFG = (N, E, Entry, Exit)

// Initialize
out[Entry] = \Ø // can set out[Entry] to special def
// if reaching then undefined use
For all nodes i
out[i] = \Ø // can optimize by out[i]=gen[i]
ChangedNodes = N

// iterate
While ChangedNodes ≠ \Ø{
    Remove i from Changed Nodes
    in[i] = U (out[p]), for all predecessors p of i
    oldout = out[i]
    out[i] = \{i(in[i]) // out[i]=gen[i]U(in[i]-kill[i])
    if oldout ≠ out[i] {
        for all successors s of i
        add s to ChangedNodes
    }
}

Example

III. Live Variable Analysis

- Definition
  - A variable v is live at point p if the value of v is used along some path in the flow graph starting at p.
  - Otherwise, the variable is dead.

- Motivation
  - e.g. register allocation
    for i = 0 TO n
      i
    for i = 0 to n
      i

- Problem statement
  - For each basic block
    determine if each variable is live in each basic block
  - Size of bit vector: one bit for each variable
Effects of a Basic Block (Transfer Function)

- Observation: Trace uses backwards to the definitions
  
  A basic block \( b \) can
  
  - generate live variables: \( \text{Use}[b] \), set of locally exposed uses in \( b \)
  
  - propagate incoming live variables: \( \text{OUT}[b] - \text{Def}[b] \), where \( \text{Def}[b] = \) set of variables defined in \( b \).
  
  - transfer function for block \( b \):
    \[
    \text{in}[b] = \text{Use}[b] \cup (\text{out}(b) - \text{Def}[b])
    \]

Flow Graph

- \( \text{in}[b] = f_b(\text{out}[b]) \)
- Join node: a node with multiple successors
- meet operator:
  \[
  \text{out}[b] = \text{in}[s_1] \cup \text{in}[s_2] \cup \ldots \cup \text{in}[s_n],
  \]
  where \( s_1, \ldots, s_n \) are all successors of \( b \)

Live Variable: Worklist Algorithm

- input: control flow graph \( \text{CFG} = (N, E, \text{Entry}, \text{Exit}) \)

```plaintext
// Initialize
\text{in}[\text{Exit}] = \emptyset //local variables
For all nodes \( i \)
\text{in}[i] = \emptyset //can optimize by \text{in}[i] = \text{use}[i]
\text{ChangedNodes} = N

// iterate
While \( \text{ChangedNodes} \neq \emptyset \) {
  Remove \( i \) from \( \text{ChangedNodes} \)
  \text{out}[i] = \text{U} (\text{in}[s]), \text{for all successors } s \text{ of } i
  \text{oldin} = \text{in}[i]
  \text{in}[i] = f_i(\text{out}[i]) //\text{in}[i] = \text{use}[i] \cup \text{out}[i] - \text{def}[i]
  \text{if} \ \text{oldin} \neq \text{in}[i] \{
    \text{for all predecessors } p \text{ of } i
    \text{add } p \text{ to } \text{ChangedNodes}
  \}
}
```

Example
IV. Framework

<table>
<thead>
<tr>
<th></th>
<th>Reaching Definitions</th>
<th>Live Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain</td>
<td>Sets of definitions</td>
<td>Sets of variables</td>
</tr>
<tr>
<td>Transfer function</td>
<td>f_b(x)</td>
<td></td>
</tr>
<tr>
<td>Generate</td>
<td>Gen_b (Gen: definitions in b)</td>
<td>Use_b (Use: var. used in b)</td>
</tr>
<tr>
<td>Propagate</td>
<td>in[b]-Kill_b (Kill_b: killed defs)</td>
<td>out[b]-Def_b (Def_b: var defined)</td>
</tr>
<tr>
<td>Merge operation</td>
<td>U (in[b]=U out[predecessors])</td>
<td>U (out[b]= U in[successors])</td>
</tr>
<tr>
<td>Initialization</td>
<td>out[entry] = ∅</td>
<td>in[exit] = ∅</td>
</tr>
<tr>
<td></td>
<td>out[b] = ∅</td>
<td>in[b] = ∅</td>
</tr>
</tbody>
</table>

Questions

- Correctness
  - equations are satisfied, if the program terminates.

- Precision: how good is the answer?
  - is the answer ONLY a union of all possible executions?

- Convergence: will the analysis terminate?
  - or, will there always be some nodes that change?

- Speed: how fast is the convergence?
  - how many times will we visit each node?