Lecture 2

Introduction to Data Flow Analysis

I. Introduction
II. Example: Reaching definition analysis
III. Example: Liveness analysis
IV. A General Framework
   (Theory in next lecture)

Reading: Chapter 9.2
Overview of Lectures 2-5

• High-level programming languages generate a lot of redundancy
• Many useful optimizations independently developed originally
  – Constant propagation, common subexpressions, loop invariant code motion, dead code elimination
• Formulate individual optimizations in the same DataFlow framework: equations with respect to nodes in a graph
  – Theory: prove properties on the framework
  – Software engineering: implement / debug / optimize framework once
• Partial redundancy elimination (PRE)
  – Subsumes multiple optimizations by setting up 4 DataFlow problems
• Practice today:
  – Many compilers use SSA (static single assignment) - an abstraction on top of dataflow
  – Idea to be covered by the homework
  – Useful for many optimizations, but cannot naturally support PRE
• Plan: L2: basic examples; L3: theory; L4: full examples; L5: PRE
I. Compiler Organization

1. Program
   Front end
   Abstract Syntax Tree
   High-level IR
   Machine-Independent Intermediate Representations

   High-level optimization
   Parallelization
   Loop transformations

   Low-level IR
   Low-level optimization
   Redundancy elimination

   Code generation
   Machine code

   Register allocation
   Instruction scheduling
Flow Graph

• Basic block = a maximal sequence of consecutive instructions s.t.
  – flow of control only enters at the beginning
  – flow of control can only leave at the end
    (no halting or branching except perhaps at end of block)

• Flow Graphs
  – Nodes: basic blocks
  – Edges
    • $B_i \rightarrow B_j$, iff $B_j$ can follow $B_i$ immediately in execution
What is Data Flow Analysis?

- Data flow analysis:
  - Flow-sensitive: sensitive to the control flow in a function
  - Intraprocedural analysis
- Examples of optimizations:
  - Constant propagation
  - Common subexpression elimination
  - Dead code elimination

Value of x?
Which “definition” defines x?
Is the definition still meaningful (live)?
Static Program vs. Dynamic Execution

- **Statically**: Finite program
- **Dynamically**: Can have infinitely many possible execution paths
- **Data flow analysis abstraction**:
  - For each point in the program:
    - combines information of all the instances of the same program point.
- **Example of a data flow question**:
  - Which definition defines the value used in statement “b = a”?

![Flowchart with nodes B1, B2, and B3]

- B1: a = 10
- B2: if input() -> exit
- B3: b = a
  - a = 11
Reaching Definitions

• Every assignment is a definition
• A definition $d$ reaches a point $p$ if there exists path from the point immediately following $d$ to $p$ such that $d$ is not killed (overwritten) along that path.
• Problem statement
  – For each point in the program, determine if each definition in the program reaches the point
  – A bit vector per program point, vector-length = #defs
Data Flow Analysis Schema

- Build a flow graph (nodes = basic blocks, edges = control flow)
- Set up a set of equations between in[b] and out[b] for all basic blocks b
  - Effect of code in basic block:
    - Transfer function $f_b$ relates in[b] and out[b], for same b
  - Effect of flow of control:
    - relates out[b₁], in[b₂] if $b₁$ and $b₂$ are adjacent
- Find a solution to the equations
Effects of a Statement

- \( f_s \): A transfer function of a statement
  - abstracts the execution with respect to the problem of interest
- For a statement \( s \) (\( d: x = y + z \))
  \( \text{out}[s] = f_s(\text{in}[s]) = \text{Gen}[s] \cup (\text{in}[s]-\text{Kill}[s]) \)
  - \( \text{Gen}[s] \): definitions generated: \( \text{Gen}[s] = \{d\} \)
  - \( \text{Propagated} \) definitions: \( \text{in}[s] - \text{Kill}[s] \),
    where \( \text{Kill}[s] \)=set of all other defs to \( x \) in the rest of program
Effects of a Basic Block

\( \text{out[B0]} = \text{transfer function of a basic block B} = f_B(\text{in[B]}) = f_d1 \cdot f_d0 \)

- Transfer function of a statement \( s \):
  - \( \text{out}[s] = f_s(\text{in}[s]) = \text{Gen}[s] \cup (\text{in}[s] - \text{Kill}[s]) \)
- Transfer function of a basic block \( B \):
  - Composition of transfer functions of statements in \( B \)
  - \( \text{out}[B] = f_B(\text{in}[B]) = f_d1 \cdot f_d0(\text{in}[B]) = \text{Gen}[d_1] \cup (\text{Gen}[d_0] \cup (\text{in}[B] - \text{Kill}[d_0])) - \text{Kill}[d_1]) = (\text{Gen}[d_1] \cup (\text{Gen}[d_0] - \text{Kill}[d_1])) \cup \text{in}[B] - (\text{Kill}[d_0] \cup \text{Kill}[d_1]) = \text{Gen}[B] \cup (\text{in}[B] - \text{Kill}[B]) \)

\( \text{Gen}[B] \): locally exposed definitions (available at end of bb)
\( \text{Kill}[B] \): set of definitions killed by \( B \)
**Effects of the Edges (acyclic)**

- **Join node**: a node with multiple predecessors
- **meet operator** ($\wedge$): $\bigvee$
  
  $$\text{in}[b] = \text{out}[p_1] \cup \text{out}[p_2] \cup \ldots \cup \text{out}[p_n],$$
  where $p_1, \ldots, p_n$ are all predecessors of $b$
Cyclic Graphs

- Equations still hold
  - \( \text{out}[b] = f_b(\text{in}[b]) \)
  - \( \text{in}[b] = \text{out}[p_1] \cup \text{out}[p_2] \cup \ldots \cup \text{out}[p_n], \ p_1, \ldots, p_n \ \text{pred.} \)
- Find: fixed point solution
Reaching Definitions: Iterative Algorithm

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

// Boundary condition
$\text{out}[\text{Entry}] = \emptyset$

// Initialization for iterative algorithm
For each basic block $B$ other than Entry
$\text{out}[B] = \emptyset$

// iterate
While (Changes to any $\text{out[]}[]$ occur) {
  For each basic block $B$ other than Entry {
    $\text{in}[B] = \cup (\text{out}[p])$, for all predecessors $p$ of $B$
    $\text{out}[B] = \text{f}_B(\text{in}[B])$  // $\text{out}[B]=\text{gen}[B] \cup (\text{in}[B]-\text{kill}[B])$
  }
}
### Summary of Reaching Definitions

<table>
<thead>
<tr>
<th>Domain</th>
<th>Reaching Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Domain</strong></td>
<td>Sets of definitions</td>
</tr>
<tr>
<td><strong>Transfer function</strong></td>
<td></td>
</tr>
</tbody>
</table>
| $f_b(x)$                | forward: $\text{out}[b] = f_b(\text{in}[b])$  
|                         | $f_b(x) = \text{Gen}_b \cup (x \setminus \text{Kill}_b)$  
|                         | $\text{Gen}_b$: definitions in b  
|                         | $\text{Kill}_b$: killed defs  
| **Meet Operation**      | $\text{in}[b] = \cup \text{out}[\text{predecessors}]$  
| **Boundary Condition**  | $\text{out}[\text{entry}] = \emptyset$  
| **Initial interior points** | $\text{out}[b] = \emptyset$  |
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**.

• **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 back to the definitions

\[ \text{in}[b] = f_b(\text{out}[b]) \]

• Direction: backward: \( \text{in}[b] = f_b(\text{out}[b]) \)

• Transfer function for statement \( s: x = y + z \)
  • generate live variables: \( \text{Use}[s] = \{y, z\} \)
  • propagate live variables: \( \text{out}[s] - \text{Def}[s], \text{Def}[s] = x \)
  • \( \text{in}[s] = \text{Use}[s] \cup (\text{out}(s)-\text{Def}[s]) \)

• Transfer function for basic block \( b \):
  • \( \text{in}[b] = \text{Use}[b] \cup (\text{out}(b)-\text{Def}[b]) \)
  • \( \text{Use}[b] \), set of locally exposed uses in \( b \), uses not covered by definitions in \( b \)
  • \( \text{Def}[b] \), set of variables defined in \( b \).
Across Basic Blocks

• **Meet operator (\(\wedge\))**:  
  - \(\text{out}[b] = \text{in}[s_1] \cup \text{in}[s_2] \cup \ldots \cup \text{in}[s_n]\), \(s_1, \ldots, s_n\) are successors of \(b\)

• **Boundary condition:**
Example

\[
\begin{align*}
\text{entry} & \quad \text{if } g \\
\text{in}[1] & \quad n = p \\
\text{out}[1] & \\
\text{in}[2] & \quad r = n + r \\
\text{out}[2] & \\
\text{in}[3] & \quad m = n + q \\
\text{out}[3] & \\
\text{exit} & \quad p = m \\
\end{align*}
\]
Liveness: Iterative Algorithm

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

// Boundary condition
in[Exit] = ∅

// Initialization for iterative algorithm
For each basic block B other than Exit
  in[B] = ∅

// iterate
While (Changes to any in[] occur) {
  For each basic block B other than Exit {
    out[B] = ∪ (in[s]), for all successors s of B
    in[B] = f_B(out[B]) // in[B]=Use[B]∪(out[B]-Def[B])
  }
}
## 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>Direction</td>
<td>forward:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>out[b] = ( f_b(in[b]) )</td>
<td></td>
</tr>
<tr>
<td></td>
<td>in[b] = ( \land out[pred(b)] )</td>
<td>backward:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>in[b] = ( f_b(out[b]) )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>out[b] = ( \land in[succ(b)] )</td>
</tr>
<tr>
<td>Transfer function</td>
<td>( f_b(x) = Gen_b \cup (x -Kill_b) )</td>
<td>( f_b(x) = Use_b \cup (x -Def_b) )</td>
</tr>
<tr>
<td>Meet Operation (( \land ))</td>
<td>( \cup )</td>
<td>( \cup )</td>
</tr>
<tr>
<td>Boundary Condition</td>
<td>out[entry] = ( \emptyset )</td>
<td>in[exit] = ( \emptyset )</td>
</tr>
<tr>
<td>Initial interior points</td>
<td>out[b] = ( \emptyset )</td>
<td>in[b] = ( \emptyset )</td>
</tr>
</tbody>
</table>
Thought Problem 1. “Must-Reach” Definitions

- A definition $D (a = b+c)$ must reach point $P$ iff
  - $D$ appears at least once along on all paths leading to $P$
  - $a$ is not redefined along any path after last appearance of $D$ and before $P$
- How do we formulate the data flow algorithm for this problem?
Problem 2: A legal solution to (May) Reaching Def?

- Will the worklist algorithm generate this answer?
Problem 3. What are the algorithm properties?

• Correctness

• Precision: how good is the answer?

• Convergence: will the analysis terminate?

• Speed: how long does it take?