Data Flow Analysis
Data Flow Analysis

• A framework for proving facts about programs

• Reasons about lots of little facts

• Little or no interaction between facts
  ▪ Works best on properties about how program computes

• Based on all paths through program
  ▪ Including infeasible paths

• Operates on control-flow graphs, typically
x := a + b;
y := a * b;
while (y > a) {
    a := a + 1;
    x := a + b
}
Control-Flow Graph w/Basic Blocks

x := a + b;
y := a * b;
while (y > a + b) {
    a := a + 1;
    x := a + b
}

• Can lead to more efficient implementations
• But more complicated to explain, so...
  ▪ We’ll use single-statement blocks in lecture today
Example with Entry and Exit

\[
x := a + b;
\]
\[
y := a * b;
\]
while (y > a) {
    \[
a := a + 1;
    \]
    \[
x := a + b
    \]
}

• All nodes without a (normal) predecessor should be pointed to by entry

• All nodes without a successor should point to exit
Notes on Entry and Exit

• Typically, we perform data flow analysis on a function body

• Functions usually have
  ▪ A unique entry point
  ▪ Multiple exit points

• So in practice, there can be multiple exit nodes in the CFG
  ▪ For the rest of these slides, we’ll assume there’s only one
  ▪ In practice, just treat all exit nodes the same way as if there’s only one exit node
Available Expressions

• An expression $e$ is available at program point $p$ if
  - $e$ is computed on every path to $p$, and
  - the value of $e$ has not changed since the last time $e$ was computed on the paths to $p$

• Optimization
  - If an expression is available, need not be recomputed
    - (At least, if it’s still in a register somewhere)
Data Flow Facts

• Is expression e available?
• Facts:
  ▪ $a + b$ is available
  ▪ $a \times b$ is available
  ▪ $a + 1$ is available
What is the effect of each statement on the set of facts?

<table>
<thead>
<tr>
<th>Stmt</th>
<th>Gen</th>
<th>Kill</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x := a + b$</td>
<td>$a + b$</td>
<td></td>
</tr>
<tr>
<td>$y := a * b$</td>
<td>$a * b$</td>
<td></td>
</tr>
<tr>
<td>$a := a + 1$</td>
<td></td>
<td>$a + 1, a + b, a * b$</td>
</tr>
</tbody>
</table>
Computing Available Expressions

\[
\emptyset \rightarrow \text{entry} \\
\{a + b\} \rightarrow x := a + b \\
\{a + b, a * b\} \rightarrow y := a * b \\
\{a + b, a * b\} \rightarrow y > a \\
\emptyset \rightarrow a := a + 1 \\
\{a + b\} \rightarrow x := a + b
\]
Terminology

• A joint point is a program point where two branches meet

• Available expressions is a forward must problem
  ▪ Forward = Data flow from in to out
  ▪ Must = At join point, property must hold on all paths that are joined
Data Flow Equations

- Let $s$ be a statement
  - $\text{succ}(s) = \{\text{immediate successor statements of } s\}$
  - $\text{pred}(s) = \{\text{immediate predecessor statements of } s\}$
  - $\text{in}(s) = \text{program point just before executing } s$
  - $\text{out}(s) = \text{program point just after executing } s$

- $\text{in}(s) = \bigcap_{s' \in \text{pred}(s)} \text{out}(s')$

- $\text{out}(s) = \text{gen}(s) \cup (\text{in}(s) - \text{kill}(s))$
  - Note: These are also called transfer functions
Liveness Analysis

• A variable $v$ is *live* at program point $p$ if
  - $v$ will be used on some execution path originating from $p$...
  - before $v$ is overwritten

• Optimization
  - If a variable is not live, no need to keep it in a register
  - If variable is dead at assignment, can eliminate assignment
Data Flow Equations

• Available expressions is a forward must analysis
  ▪ Data flow propagate in same dir as CFG edges
  ▪ Expr is available only if available on all paths

• Liveness is a backward may problem
  ▪ To know if variable live, need to look at future uses
  ▪ Variable is live if used on some path

• \( \text{out}(s) = \bigcup_{s' \in \text{succ}(s)} \text{in}(s') \)

• \( \text{in}(s) = \text{gen}(s) \cup (\text{out}(s) - \text{kill}(s)) \)
### Gen and Kill

- What is the effect of each statement on the set of facts?

<table>
<thead>
<tr>
<th>Stmt</th>
<th>Gen</th>
<th>Kill</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x := a + b$</td>
<td>$a, b$</td>
<td>$x$</td>
</tr>
<tr>
<td>$y := a \times b$</td>
<td>$a, b$</td>
<td>$y$</td>
</tr>
<tr>
<td>$y &gt; a$</td>
<td>$a, y$</td>
<td></td>
</tr>
<tr>
<td>$a := a + 1$</td>
<td>$a$</td>
<td>$a$</td>
</tr>
</tbody>
</table>
Computing Live Variables

\{a, b\} → x := a + b

\{x, a, b\} → y := a \times b

\{x, y, a, b\} → y > a

\{y, a, b\} → a := a + 1

\{y, a, b\} → x := a + b

\{x, y, a, b\}
Very Busy Expressions

• An expression $e$ is very busy at point $p$ if
  - On every path from $p$, expression $e$ is evaluated before the value of $e$ is changed

• Optimization
  - Can hoist very busy expression computation

• What kind of problem?
  - Forward or backward? backward
  - May or must? must
Reaching Definitions

• A *definition* of a variable $v$ is an assignment to $v$
• A definition of variable $v$ reaches point $p$ if
  - There is no intervening assignment to $v$

• Also called def-use information

• What kind of problem?
  - Forward or backward? *forward*
  - May or must? *may*
Space of Data Flow Analyses

<table>
<thead>
<tr>
<th></th>
<th>May</th>
<th>Must</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward</td>
<td>Reaching definitions</td>
<td>Available expressions</td>
</tr>
<tr>
<td>Backward</td>
<td>Live variables</td>
<td>Very busy expressions</td>
</tr>
</tbody>
</table>

- Most data flow analyses can be classified this way
  - A few don’t fit: bidirectional analysis
- Lots of literature on data flow analysis
Solving data flow equations

• Let’s start with forward may analysis
  ▪ Dataflow equations:
    - \( \text{in}(s) = \bigcup_{s' \in \text{pred}(s)} \text{out}(s') \)
    - \( \text{out}(s) = \text{gen}(s) \cup (\text{in}(s) - \text{kill}(s)) \)

• Need algorithm to compute \text{in} and \text{out} at each stmt

• Key observation: \text{out}(s) is monotonic in \text{in}(s)
  ▪ \text{gen}(s) and \text{kill}(s) are fixed for a given \( s \)
  ▪ If, during our algorithm, \text{in}(s) grows, then \text{out}(s) grows
  ▪ Furthermore, \text{out}(s) and \text{in}(s) have max size

• Same with \text{in}(s)
  ▪ in terms of \text{out}(s') for precedessors \( s' \)
Solving data flow equations (cont’d)

• Idea: fixpoint algorithm
  ▪ Set \texttt{out(entry)} to emptyset
    - E.g., we know no definitions reach the entry of the program
  ▪ Initially, assume \texttt{in(s)}, \texttt{out(s)} empty everywhere else, also
  ▪ Pick a statement \texttt{s}
    - Compute \texttt{in(s)} from predecessors’ \texttt{out}’s
    - Compute new \texttt{out(s)} for \texttt{s}
  ▪ Repeat until nothing changes

• Improvement: use a worklist
  ▪ Add statements to worklist if their \texttt{in(s)} might change
  ▪ Fixpoint reached when worklist is empty
Forward May Data Flow Algorithm

\[
\text{out(entry)} = \emptyset \\
\text{for all other statements } s \\
\quad \text{out}(s) = \emptyset \\
W = \text{all statements} \quad // \text{worklist} \\
\text{while } W \text{ not empty} \\
\quad \text{take } s \text{ from } W \\
\quad \text{in}(s) = \bigcup_{s' \in \text{pred}(s)} \text{out}(s') \\
\quad \text{temp} = \text{gen}(s) \cup (\text{in}(s) - \text{kill}(s)) \\
\quad \text{if temp} \neq \text{out}(s) \text{ then} \\
\quad \quad \text{out}(s) = \text{temp} \\
\quad \quad W := W \cup \text{succ}(s) \\
\quad \text{end} \\
\text{end}
\]
## Generalizing

<table>
<thead>
<tr>
<th></th>
<th>May</th>
<th>Must</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Forward</strong></td>
<td>in(s) = ∪_{s' ∈ pred(s)} out(s')</td>
<td>in(s) = ∩_{s' ∈ pred(s)} out(s')</td>
</tr>
<tr>
<td></td>
<td>out(s) = gen(s) ∪ (in(s) - kill(s))</td>
<td>out(s) = gen(s) ∪ (in(s) - kill(s))</td>
</tr>
<tr>
<td></td>
<td>out(entry) = ∅</td>
<td>out(entry) = ∅</td>
</tr>
<tr>
<td></td>
<td>initial out elsewhere = ∅</td>
<td>initial out elsewhere = {all facts}</td>
</tr>
<tr>
<td><strong>Backward</strong></td>
<td>out(s) = ∪_{s' ∈ succ(s)} in(s')</td>
<td>out(s) = ∩_{s' ∈ succ(s)} in(s')</td>
</tr>
<tr>
<td></td>
<td>in(s) = gen(s) ∪ (out(s) - kill(s))</td>
<td>in(s) = gen(s) ∪ (out(s) - kill(s))</td>
</tr>
<tr>
<td></td>
<td>in(exit) = ∅</td>
<td>in(exit) = ∅</td>
</tr>
<tr>
<td></td>
<td>initial in elsewhere = ∅</td>
<td>initial in elsewhere = {all facts}</td>
</tr>
</tbody>
</table>
### Forward Analysis

<table>
<thead>
<tr>
<th>May</th>
<th>Must</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{out(entry)} = \emptyset)</td>
<td>(\text{out(entry)} = \emptyset)</td>
</tr>
<tr>
<td>for all other statements (s)</td>
<td>for all other statements (s)</td>
</tr>
<tr>
<td>(\text{out}(s) = \emptyset)</td>
<td>(\text{out}(s) = all \text{ facts})</td>
</tr>
<tr>
<td>(W = all \text{ statements} \quad // \text{worklist})</td>
<td>(W = all \text{ statements})</td>
</tr>
<tr>
<td>while (W) not empty</td>
<td>while (W) not empty</td>
</tr>
<tr>
<td>take (s) from (W)</td>
<td>take (s) from (W)</td>
</tr>
<tr>
<td>(\text{in}(s) = \bigcup_{s' \in \text{pred}(s)} \text{out}(s'))</td>
<td>(\text{in}(s) = \bigcap_{s' \in \text{pred}(s)} \text{out}(s'))</td>
</tr>
<tr>
<td>temp = (\text{gen}(s) \cup (\text{in}(s) - \text{kill}(s)))</td>
<td>temp = (\text{gen}(s) \cup (\text{in}(s) - \text{kill}(s)))</td>
</tr>
<tr>
<td>if temp (\neq) (\text{out}(s)) then</td>
<td>if temp (\neq) (\text{out}(s)) then</td>
</tr>
<tr>
<td>(\text{out}(s) = \text{temp})</td>
<td>(\text{out}(s) = \text{temp})</td>
</tr>
<tr>
<td>(W := W \cup \text{succ}(s))</td>
<td>(W := W \cup \text{succ}(s))</td>
</tr>
<tr>
<td>end</td>
<td>end</td>
</tr>
<tr>
<td>end</td>
<td>end</td>
</tr>
</tbody>
</table>
Backward Analysis

\[
\text{in(EXIT)} = \emptyset \quad \text{for all other statements } s \\
\text{in(s)} = \emptyset \\
W = \text{all statements} \\
\text{while } W \text{ not empty} \\
\text{take } s \text{ from } W \\
\text{out(s)} = \bigcup_{s' \in \text{succ}(s)} \text{in(s')} \\
\text{temp} = \text{gen}(s) \cup (\text{out}(s) - \text{kill}(s)) \\
\text{if } \text{temp} \neq \text{in(s)} \text{ then} \\
\text{in(s)} = \text{temp} \\
W := W \cup \text{pred}(s) \\
\text{end} \\
\text{end}
\]

May

\[
\text{in(EXIT)} = \emptyset \quad \text{for all other statements } s \\
\text{in(s)} = \text{all facts} \\
W = \text{all statements} \\
\text{while } W \text{ not empty} \\
\text{take } s \text{ from } W \\
\text{out(s)} = \bigcap_{s' \in \text{succ}(s)} \text{in(s')} \\
\text{temp} = \text{gen}(s) \cup (\text{out}(s) - \text{kill}(s)) \\
\text{if } \text{temp} \neq \text{in(s)} \text{ then} \\
\text{in(s)} = \text{temp} \\
W := W \cup \text{pred}(s) \\
\text{end} \\
\text{end}
\]

Must
Practical Implementation

• Represent set of facts as bit vector
  ■ Fact$_i$ represented by bit i
  ■ Intersection = bitwise and, union = bitwise or, etc

• “Only” a constant factor speedup
  ■ But very useful in practice
Basic Blocks

• Recall a basic block is a sequence of statements s.t.
  - No statement except the last in a branch
  - There are no branches to any statement in the block except the first

• In some data flow implementations,
  - Compute gen/kill for each basic block as a whole
    - Compose transfer functions
  - Store only in/out for each basic block
  - Typical basic block ~5 statements
    - At least, this used to be the case...
Order Matters

• Assume forward data flow problem
  ▪ Let $G = (V, E)$ be the CFG
  ▪ Let $k$ be the height of the lattice

• If $G$ acyclic, visit in topological order
  ▪ Visit head before tail of edge

• Running time $O(|E|)$
  ▪ No matter what size the lattice
Order Matters — Cycles

• If $G$ has cycles, visit in reverse postorder
  ▪ Order from depth-first search
  ▪ (Reverse for backward analysis)

• Let $Q =$ max # back edges on cycle-free path
  ▪ Nesting depth
  ▪ Back edge is from node to ancestor in DFS tree

• In common cases, running time can be shown to be $O((Q+1)|E|)$
  ▪ Proportional to structure of CFG rather than lattice
Flow-Sensitivity

• Data flow analysis is *flow-sensitive*
  ▪ The order of statements is taken into account
  ▪ I.e., we keep track of facts per program point

• Alternative: *Flow-insensitive* analysis
  ▪ Analysis the same regardless of statement order
  ▪ Standard example: types
    - /* x : int */ x := ... /* x : int */
Data Flow Analysis and Functions

• What happens at a function call?
  ▪ Lots of proposed solutions in data flow analysis literature

• In practice, only analyze one procedure at a time

• Consequences
  ▪ Call to function kills all data flow facts
  ▪ May be able to improve depending on language, e.g., function call may not affect locals
More Terminology

• An analysis that models only a single function at a time is *intraprocedural*

• An analysis that takes multiple functions into account is *interprocedural*

• An analysis that takes the whole program into account is *whole program*

• Note: *global* analysis means “more than one basic block,” but still within a function
  ▪ Old terminology from when computers were slow...
Data Flow Analysis and The Heap

• Data Flow is good at analyzing local variables
  ▪ But what about values stored in the heap?
  ▪ Not modeled in traditional data flow

• In practice: *x := e
  ▪ Assume all data flow facts killed (!)
  ▪ Or, assume write through x may affect any variable whose address has been taken

• In general, hard to analyze pointers
Proebsting’s Law

• Moore’s Law: Hardware advances double computing power every 18 months.

• Proebsting’s Law: Compiler advances double computing power every 18 years.
  ▪ Not so much bang for the buck!
DFA and Defect Detection

- LCLint - Evans et al. (UVa)
- METAL - Engler et al. (Stanford, now Coverity)
- ESP - Das et al. (MSR)
- FindBugs - Hovemeyer, Pugh (Maryland)
  - For Java. The first three are for C.

- Many other one-shot projects
  - Memory leak detection
  - Security vulnerability checking (tainting, info. leaks)