CMSC 631 — Program Analysis and Understanding
Fall 2003

Data Flow Analysis

Compiler Structure

- Source code parsed to produce AST
- AST transformed to CFG
  - Simpler representation of the program
  - Fewer forms to reason about
- Data flow analysis operates on control flow graph (and other intermediate representations)

Control-Flow Graph

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

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

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 is computed on 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 * b is available
  - y > a + b is available
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></td>
</tr>
<tr>
<td>y := a * b</td>
<td>a * b</td>
<td></td>
</tr>
<tr>
<td>y &gt; a + b</td>
<td>y &gt; a + b</td>
<td></td>
</tr>
<tr>
<td>a := a + 1</td>
<td></td>
<td>a + b</td>
</tr>
</tbody>
</table>

Data Flow Equations

- Let s be a statement
  - succ(s) = { immediate successor statements of s }
  - pred(s) = { immediate predecessor statements of s }
  - In(s) = program point just before executing s
  - Out(s) = program point just after executing s

\[
\text{In}(s) = \bigcap_{s' \in \text{pred}(s)} \text{Out}(s')
\]

- Out(s) = Gen(s) U (In(s) - 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

Computing Available Expressions

- 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 available on some path

\[
\text{In}(s) = \text{Gen}(s) \cup \text{Out}(s) - \text{Kill}(s)
\]

- Out(s) = \( \bigcup_{s' \in \text{succ}(s)} \text{In}(s') \)

Optimization

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

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 * b</td>
<td>a, b</td>
<td>y</td>
</tr>
<tr>
<td>y &gt; a + b</td>
<td>y, a, b</td>
<td></td>
</tr>
<tr>
<td>a := a + 1</td>
<td>a</td>
<td>a</td>
</tr>
</tbody>
</table>

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 available on some path

\[
\text{In}(s) = \text{Gen}(s) \cup \text{Out}(s) - \text{Kill}(s)
\]

- Out(s) = \( \bigcup_{s' \in \text{succ}(s)} \text{In}(s') \)
Computing Live Variables

\[ \{a, b\} \]

\[ x := a + b \]

\[ \{x, a, b\} \]

\[ y := a \times b \]

\[ y > a + b \]

\[ \{y, a, b\} \]

\[ a := a + 1 \]

\[ \{x\} \]

\[ \{x, y, a, b\} \]

\[ \{x, y, a, b\} \]

\[ \{y, a, b\} \]

\[ \{y, a, b\} \]

\[ \{x, a, b\} \]

\[ \{a, b\} \]

Very Busy Expressions

- An expression \( e \) is very busy at point \( p \) if
  - On every path from \( p \), \( e \) is evaluated before the value of \( e \) is changed
  - Optimization
    - Can hoist very busy expression computation
  - What kind of problem?
    - Forward or backward! \( \text{backward} \)
    - May or must? \( \text{must} \)

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! \( \text{forward} \)
  - May or must? \( \text{may} \)

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>

Partial Orders

- A partial order is a pair \( (P, \leq) \) such that
  - \( \leq \subseteq P \times P \)
  - \( \leq \) is reflexive: \( x \leq x \)
  - \( \leq \) is anti-symmetric: \( x \leq y \) and \( y \leq x \Rightarrow x = y \)
  - \( \leq \) is transitive: \( x \leq y \) and \( y \leq z \Rightarrow x \leq z \)

Lattices

- A partial order is a lattice if \( \cap \) and \( \cup \) are defined on any set \( S \):
  - \( \cap \) is the meet or greatest lower bound operation:
    - \( x \cap y \leq x \) and \( x \cap y \leq y \)
    - if \( z \leq x \) and \( z \leq y \), then \( z \leq x \cap y \)
  - \( \cup \) is the join or least upper bound operation:
    - \( x \leq x \cup y \) and \( y \leq x \cup y \)
    - if \( x \leq z \) and \( y \leq z \), then \( x \cup y \leq z \)
### Lattices (cont’d)

- A finite partial order is a lattice if meet and join exist for every pair of elements.
- A lattice has unique elements \(\bot\) and \(\top\) such that:
  - \(x \cap \bot = \bot\)
  - \(x \cup \bot = x\)
  - \(x \cap \top = x\)
  - \(x \cup \top = \top\)

- In a lattice,
  - \(x \leq y \iff x \cap y = x\)
  - \(x \leq y \iff x \cup y = y\)

### Data Flow Facts and Lattices

- Typically, data flow facts form a lattice:
  - Example: Available expressions

\[
\begin{align*}
\text{a+b, a*b, y > a+b} \\
\text{a+b, a*b} \\
\text{y > a+b} \\
\text{(none)}
\end{align*}
\]

### Forward May Data Flow Algorithm

- \(\text{Out}(s) = \text{Gen}(s)\) for all statements \(s\)
  - Or, if you like, \(\text{Out}(s) = \top\)
  - \(W := \{\text{all statements}\}\) (worklist)
  - repeat
    - Take \(s\) from \(W\)
    - \(\text{In}(s) := \cap \{s \in \text{pred}(s) \mid \text{Out}(s)\}\)
    - \(\text{temp} := \text{Gen}(s) \cup \text{In}(s) - \text{Kill}(s)\)
    - if (temp \(!=\) Out(s)) {
      - \(\text{Out}(s) := \text{temp}\)
      - \(W := W \cup \text{succ}(s)\)
    }
  - until \(W = \emptyset\)

### Monotonicity

- A function \(f\) on a partial order is monotonic if
  - \(x \leq y \Rightarrow f(x) \leq f(y)\)

- Easy to check that operations to compute \(\text{In}\) and \(\text{Out}\) are monotonic:
  - \(\cap \{s \in \text{pred}(s) \mid \text{Out}(s)\}\)
  - \(\text{Gen}(s) \cup \text{In}(s) - \text{Kill}(s)\)

### Termination

- We know the algorithm terminates because:
  - The lattice has finite height
  - The operations to compute \(\text{In}\) and \(\text{Out}\) are monotonic
  - On every iteration, we remove a statement from the worklist and/or move down the lattice.
Distributive Data Flow Problems

• By monotonicity, we also have
  \[ f(x \sqcap y) \leq f(x) \sqcap f(y) \]

• A function \( f \) is distributive if
  \[ f(x \sqcap y) = f(x) \sqcap f(y) \]

Benefit of Distributivity

• Joins lose no information

\[ k(h(f(T)) \sqcap g(T))) = k(h(f(T))) \sqcap k(h(g(T))) \]

Accuracy of Data Flow Analysis

• Ideally, we would like to compute the meet over all paths (MOP) solution:
  ■ Let \( f_s \) be the transfer function for statement \( s \)
  ■ If \( p \) is a path \( \{s_1, \ldots, s_n\} \), let \( f_p = f_{s_n} \ldots f_{s_1} \)
  ■ Let path(\( s \)) be the set of paths from the entry to \( s \)
  \[ \text{MOP}(s) = \sqcap_{p \in \text{path}(s)} f_p(T) \]

• If a data flow problem is distributive, then solving the data flow equations in the standard way yields the MOP solution

What Problems are Distributive?

• Analyses of how the program computes
  ▪ Live variables
  ▪ Available expressions
  ▪ Reaching definitions
  ▪ Very busy expressions

• All Gen/Kill problems are distributive

A Non-Distributive Example

• Constant propagation

  \[
  \begin{array}{ccc}
  x & = & 1 \\
  y & = & 2 \\
  z & = & x + y \\
  \end{array}
  \]

• In general, analysis of what the program computes in not distributive

Practical Implementation

• Data flow facts = assertions that are true or false at a program point

  ▪ Represent set of facts as bit vector
    ▪ Fact, represented by bit \( i \)
    ▪ Intersection = bitwise and, union = bitwise or, etc

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

- 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 practical data flow implementations,
  - Compute Gen/Kill for each basic block
    - Compose transfer functions
  - Store only In/Out for each basic block
  - Typical basic block ~5 statements

Order Matters

- Assume forward data flow problem

- If G acyclic, visit in topological order
  - Visit head before tail of edge
  - Running time $O(|E|)$
  - No matter what size the lattice

Another Approach: Elimination

- Recall in practice, one transfer function per basic block

- Why not generalize this idea beyond a basic block?
  - “Collapse” larger constructs into smaller ones, combining data flow equations
  - Eventually program collapsed into a single node!
  - “Expand out” back to original constructs, rebuilding information

Complexity

- Data flow algorithm works for any order in which nodes are visited
  - Let $G = (V,E)$ be the CFG
  - Let $k$ be the height of the lattice
  - Running time is $O(k|E|)$

- If $k$ non-trivial, can do better by choosing order carefully

Order Matters — Cycles

- If $G$ has cycles, visit in reverse postorder
  - Order from depth-first search
  - Let $Q = \max \#$ back edges on cycle-free path
    - Nesting depth
    - Back edge is from node to ancestor on DFS tree
  - Then if $\forall x. f(x) \leq x$ (sufficient, but not necessary)
    - Running time is $O((Q + 1)|E|)$
      - Slightly better bound than Kam + Ullman paper
      - Note direction of req’t depends on top vs. bottom

Elimination Methods: Conditionals

$$f_{ite} = (f_{then} \circ f_{if}) \cap (f_{else} \circ f_{if})$$

$$\text{Out}(\text{if}) = f_{if}(\text{In}(\text{ite}))$$
$$\text{Out}(\text{then}) = (f_{then} \circ f_{if})(\text{In}(\text{ite}))$$
$$\text{Out}(\text{else}) = (f_{else} \circ f_{if})(\text{In}(\text{ite}))$$
Non-Reducible Flow Graphs

- Elimination methods only work on reducible flow graphs
  - Ones that can be collapsed
  - Standard constructs yield only reducible flow graphs
- Unrestricted goto can yield non-reducible graphs

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 */
    ```

Terminology Review

- Must vs. May
  - (Not always followed in literature)
- Forwards vs. Backwards
- Flow-sensitive vs. Flow-insensitive
- Distributive vs. Non-distributive

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...guess?
  - Note: global analysis means “more than one basic block,” but still within a function

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: \(^\ast\)x := e
  - Assume all data flow facts killed (!)
  - Or, assume x may affect any variable whose address has not been taken
- In general, hard to analyze pointers