CMSC 631 — Program Analysis and Understanding
Fall 2004

Data Flow Analysis

Compiler Structure

- Source code parsed to produce AST
- AST transformed to CFG
- Data flow analysis operates on control flow graph (and other intermediate representations)

Abstract Syntax Tree (AST)

- Programs are written in text
  - I.e., sequences of characters
  - Awkward to work with
- First step: Convert to structured representation
  - Use lexer (like flex) to recognize tokens
    - Sequences of characters that make words in the language
  - Use parser (like bison) to group words structurally
    - And, often, to produce AST

Abstract Syntax Tree Example

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

ASTs

- ASTs are abstract
  - They don’t contain all information in the program
    - E.g., spacing, comments, brackets, parentheses
  - Any ambiguity has been resolved
    - E.g., \(a + b + c\) produces the same AST as \((a + b) + c\)
- For more info, see CMSC 430
  - In this class, we will generally begin at the AST level

Disadvantages of ASTs

- AST has many similar forms
  - E.g., for, while, repeat...until
  - E.g., if, ?:, switch
- Expressions in AST may be complex, nested
  - \((42 \times y) + (z > 5 \ ? 12 \times z : z + 20)\)
- Want simpler representation for analysis
  - ...at least, for dataflow analysis
**Control-Flow Graph (CFG)**

- A directed graph where
  - Each node represents a statement
  - Edges represent control flow

- Statements may be
  - Assignments $x := y \text{ op } z$ or $x := \text{ op } z$
  - Copy statements $x := y$
  - Branches $\text{ goto } L$ or if $x \text{ relop } y$ goto $L$
  - etc.

**Control-Flow Graph Example**

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

**Variations on CFGs**

- We usually don’t include declarations (e.g., int $x$;)
  - But there’s usually something in the implementation

- May want a unique entry and exit node
  - Won’t matter for the examples we give

- May group statements into basic blocks
  - A sequence of instructions with no branches into or out of the block

**Control-Flow Graph w/Basic Blocks**

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

**CFG vs. AST**

- CFGs are much simpler than ASTs
  - Fewer forms, less redundancy, only simple expressions

- But...AST is a more faithful representation
  - CFGs introduce temporaries
  - Lose block structure of program

- So for AST,
  - Easier to report error + other messages
  - Easier to explain to programmer
  - Easier to unpause to produce readable code

**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
  - \( a \times b \) is available
  - \( a + 1 \) is available

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

Gen and Kill

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

<table>
<thead>
<tr>
<th>Some</th>
<th>Gen</th>
<th>Kill</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x := a + b )</td>
<td>( a + b )</td>
<td>( a + b )</td>
</tr>
<tr>
<td>( y := a \times b )</td>
<td>( a \times b )</td>
<td>( a \times b )</td>
</tr>
<tr>
<td>( a := a + 1 )</td>
<td>( a + 1 ), ( a + b ), ( a \times b )</td>
<td>( a + b )</td>
</tr>
<tr>
<td>( x := a + b )</td>
<td>( a + b )</td>
<td>( a + b )</td>
</tr>
</tbody>
</table>

Terminology

- A \textit{joint point} is a program point where two branches meet

- Available expressions is a \textit{forward must} problem
  - Forward = Data flow from \textit{in} to \textit{out}
  - Must = At join point, property must hold on all paths that are joined
**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 available on some path
    - $\text{In}(s) = \text{Gen}(s) \cup (\text{Out}(s) - \text{Kill}(s))$
    - $\text{Out}(s) = \bigcup_{s' \in \text{succ}(s)} \text{In}(s')$

**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>$y := a * b$</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 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

**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? 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

**Data Flow Facts and Lattices**

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

```
    a+b, a*b, a+1
    ————
    1 2 3
```

**Lattices**

- A partial order is a lattice if meet and join are defined on any set:
  - \( \sqcap \) is the meet or greatest lower bound operation:
    - \( x \sqcap y \leq x \) and \( x \sqcap y \leq y \)
    - if \( x \leq z \) and \( y \leq z \), then \( z \leq x \sqcap y \)
  - \( \sqcup \) is the join or least upper bound operation:
    - \( x \leq x \sqcup y \) and \( y \leq x \sqcup y \)
    - if \( x \leq z \) and \( y \leq z \), then \( x \sqcup y \leq z \)

**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
  - \( x \sqcap \top = x \)
  - \( x \sqcup \bot = x \)

In a lattice,
- \( x \leq y \) iff \( x \sqcap y = x \)
- \( x \leq y \) iff \( x \sqcup y = y \)

**Useful Lattices**

- \( (2^S, \subseteq) \) forms a lattice for any set \( S \)
  - \( 2^S \) is the powerset of \( S \) (set of all subsets)

- If \( (S, \leq) \) is a lattice, so is \( (S, \geq) \)
  - i.e., lattices can be flipped

- The lattice for constant propagation

```
    1
   /\ 2
  /   \ 3
   \   /  ...
    \ / \
     \bot
```
**Forward Must Data Flow Algorithm**

- $\text{Out}(s) = \text{Gen}(s)$ for all statements $s$
  - Or, if you want, $\text{Out}(s) = \text{Top}$
- $W := \{ \text{all statements} \}$ (worklist)
- repeat
  - Take $s$ from $W$
  - $\text{In}(s) := \cap_{s' \in \text{pred}(s)} \text{Out}(s')$
  - $\text{temp} := \text{Gen}(s) \cup (\text{In}(s) - \text{Kill}(s))$
  - if ($\text{temp} \neq \text{Out}(s)$) {
      - $\text{Out}(s) := \text{temp}$
      - $W := W \cup \text{succ}(s)$
  }
- 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
  - $\text{In}(s) := \cap_{s' \in \text{pred}(s)} \text{Out}(s')$
  - $\text{temp} := \text{Gen}(s) \cup (\text{In}(s) - \text{Kill}(s))$
  - Putting these two together,
    - $\text{temp} := f_s(\cap_{s' \in \text{pred}(s')} \text{Out}(s'))$

**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