**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

**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
  - \((42 \cdot y) + (z > 5 ? 12 \cdot z : z + 20)\)

- Want simpler representation for analysis
  - ...at least, for dataflow analysis

**CMSC 631 — Program Analysis and Understanding**

Fall 2006

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)

**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 with the AST

**Abstract Syntax Tree Example**

- \(x := a + b;\)
- \(y := a \cdot b;\)
- while \((y > a)\) {
  - \(a := a + 1;\)
  - \(x := a + b\)
- }

**CMSC 430**
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 $x \text{ relop } y \text{ goto } L$
  - etc.

Control-Flow Graph Example

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

Variations on CFGs

- 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

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

CFG vs. AST

- CFGs are much simpler than ASTs
  - Fewer forms, less redundancy, 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 unpars 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

- 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
  - $a \cdot b$ is available
  - $a + 1$ 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>$a + b$</td>
</tr>
<tr>
<td>$y := a \cdot b$</td>
<td>$a \cdot b$</td>
<td>$a \cdot b$</td>
</tr>
<tr>
<td>$a := a + 1$</td>
<td>$a + 1$, $a + b$, $a \cdot b$</td>
<td>$a + 1$, $a + b$, $a \cdot b$</td>
</tr>
</tbody>
</table>

Computing Available Expressions

- $\emptyset$
- $\{ a + b \}$
- $\{ a + b, a \cdot b \}$
- $\{ a + b, a \cdot b, y = a \}$
- $\{ a + b, a \cdot b, y = a \}$
- $\{ a + b \}$
- $\emptyset$
- $\{ a + b \}$
- $\{ a + b \}$
- $\{ a + b \}$

Terminology

- A *join 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
  - $\text{succ}(s) = \{ \text{immediate successor statements of } s \}$
  - $\text{pred}(s) = \{ \text{immediate predecessor statements of } s \}$
  - $\text{In}(s) = \text{program point just before executing } s$
  - $\text{Out}(s) = \text{program point just after executing } s$

- $\text{In}(s) = \bigcap_{s' \in \text{pred}(s)} \text{Out}(s')$
- $\text{Out}(s) = \text{Gen}(s) \cup (\text{In}(s) - \text{Kill}(s))$
  - 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

\[ \text{Out}(s) = \bigcup s' \in \text{succ}(s) \text{ ln}(s') \]
\[ \text{ln}(s) = \text{Gen}(s) \cup (\text{Out}(s) - \text{Kill}(s)) \]

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>$y := a \ast b$</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></td>
</tr>
</tbody>
</table>

Computing Live Variables

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

Very Busy Expressions

- An expression $e$ is very busy at point $p$ if
  - On every path from $p$, expression $e$ is used before any component of $e$ is changed

- Optimization
  - Can hoist very busy expression computation to $p$

- 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</td>
<td>Available</td>
</tr>
<tr>
<td></td>
<td>definitions</td>
<td>expressions</td>
</tr>
<tr>
<td>Backward</td>
<td>Live variables</td>
<td>Very busy</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

Partial Orders

- A partial order is a pair \((P, \leq)\) such that
  - \((\leq)\) is reflexive: \(x \leq x\)
  - \((\leq)\) is anti-symmetric: \(x \leq y \text{ and } y \leq x \implies x = y\)
  - \((\leq)\) is transitive: \(x \leq y \text{ and } y \leq z \implies x \leq z\)

Meet and Join Operations

- \(\wedge\) is the meet or greatest lower bound operation:
  - \(a \wedge b \leq a \text{ and } a \wedge b \leq b\)
  - If \(a \leq a \wedge b \text{ and } b \leq a \wedge b\), then \(a \leq a \wedge b\)
- \(\vee\) is the join or least upper bound operation:
  - \(a \vee b \geq a \text{ and } a \vee b \geq b\)
  - If \(a \geq a \vee b \text{ and } b \geq a \vee b\), then \(a \geq a \vee b\)

Lattices

- A partial order is a lattice if meet and join exist for every pair of elements in \(P\)
- A lattice has unique elements \(a\) and \(b\) such that
  - \(a \wedge b = \min\{a, b\}\)
  - \(a \vee b = \max\{a, b\}\)
- In a lattice, \(x \leq y \text{ if } \exists a : x = a \wedge y\)
- A partial order is a complete lattice if meet and join are defined on any set \(S \subseteq P\)

Useful Lattices

- \((2^S, \subseteq)\) forms a lattice for any set \(S\)
  - \(2^S\) is the powerset of \(S\) (set of all subsets)
- If \((S, \subseteq)\) is a lattice, so is \((S, \supseteq)\)
  - i.e., lattices can be flipped
- The lattice for constant propagation
**Forward Must Data Flow Algorithm**

- \( \text{Out}(s) = \text{Top} \) for all statements \( s \)
- \( \text{// Slight acceleration: Could set Out}(s) = \text{Gen}(s) \cup (\text{Top} - \text{Kill}(s)) \)
- \( W := \{ \text{all statements} \} \) (worklist)

```
repeat
  \( \text{Take } s \text{ from } W \)
  \( \text{In}(s) := n \text{ s' } \epsilon \text{ pred}(s) \text{ Out}(s') \)
  \( \text{temp} := \text{Gen}(s) \cup (\text{In}(s) - \text{Kill}(s)) \)
  \( \text{if (temp \text{~} fn\text{~ Out}(s))} \{ \)
    - Out\( (s) := \text{temp} \)
    - W := W \cup \text{succ}(s) 
  \} 
until W = \emptyset
```
Lattices ($P, \leq$), cont’d

• Live variables
  - $P =$ sets of variables
  - $S_1 \cap S_2 = S_1 \cup S_2$
  - Top = empty set

• Very busy expressions
  - $P =$ set of expressions
  - $S_1 \cap S_2 = S_1 \cap S_2$
  - Top = set of all expressions

Forward vs. Backward

\[
Out(s) = \text{Top for all } s \\
W \rightarrow \{ \text{all statements } \} \\
\text{repeat} \\
\text{Take } s \text{ from } W \\
\text{temp} := f(s) \text{ (} s' \in \text{pred}(s) \text{)} \\
\text{if (temp \neq Out(s))} \\
\text{Out}(s) := \text{temp} \\
W := W \cup \text{succ}(s) \\
\text{until } W = \emptyset \\
\]

\[
In(s) = \text{Top for all } s \\
W \rightarrow \{ \text{all statements } \} \\
\text{repeat} \\
\text{Take } s \text{ from } W \\
\text{temp} := f(s) \text{ (} s' \in \text{succ}(s) \text{)} \\
\text{if (temp \neq In(s))} \\
\text{In}(s) := \text{temp} \\
W := W \cup \text{pred}(s) \\
\text{until } W = \emptyset \\
\]

Termination Revisited

• How many times can we apply this step:
  - \[
  \text{temp} := f(s) \text{ (} s' \in \text{pred}(s) \text{)} \\
  \text{if (temp \neq Out(s))} \\
  \]
  - Claim: $Out(s)$ only “shrinks”
    - Proof: $Out(s)$ starts out as top
    - So temp = Top after first step
    - Assume $Out(s')$ shrinks for all predecessors $s'$ of $s$
    - Then $s' \in \text{pred}(s)$ shrinks
    - Since $f$ is monotonic, $f(s) \text{ (} s' \in \text{pred}(s) \text{)}$ shrinks

Least vs. Greatest Fixpoints

• Dataflow tradition: Start with Top, use meet
  - To do this, we need a complete meet semilattice with top, of finite height
  - Complete meet semilattice = meets defined for any set
  - Finite height ensures termination
  - Computes greatest fixpoint

• Denotational semantics tradition: Start with Bottom, use join
  - Computes least fixpoint

Termination Revisited (cont’d)

• A descending chain in a lattice is a sequence
  - $x_0 > x_1 > x_2 > \ldots$
  - The height of a lattice is the length of the longest descending chain in the lattice

• Then, dataflow must terminate in $O(nk)$ time
  - $n =$ # of statements in program
  - $k =$ height of lattice
  - Assumes meet operation takes $O(1)$ time

Least vs. Greatest Fixpoints

• Dataflow tradition: Start with Top, use meet
  - To do this, we need a complete meet semilattice with top, of finite height
  - Complete meet semilattice = meets defined for any set
  - Finite height ensures termination
  - Computes greatest fixpoint

• Denotational semantics tradition: Start with Bottom, use join
  - Computes least fixpoint

Distributive Data Flow Problems

• By monotonicity, we also have
  \[
  f(c \uplus g) \leq f(c) f(1) f(g) \\
  \]

• A function $f$ is distributive if
  \[
  f(c \uplus g) = f(c) f(1) f(g) \\
  \]
Benefit of Distributivity

• Joins lose no information

\[ f \rightarrow^h \rightarrow g \]

Accuracy of Data Flow Analysis

• Ideally, we would like to compute the meet over all paths (MOP) solution:

\[ MOP(s) = \bigcap_{p \in \text{paths}} f_p(s) \]

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

\[ x := 2 \rightarrow x := 1 \rightarrow y := 2 \rightarrow y := 1 \rightarrow z := x + y \]

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
  - 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
  - Let $Q = \max$ # back edges on cycle free path
    - Nesting depth
    - Back edge is from node to ancestor on DFS tree
  - Then if $2Q/(Q + 1) \leq 2$ (sufficient, but not necessary)
    - Running time is $\frac{|E|Q + |E|}{Q + 1}$
      - Note direction of req't depends on top vs. bottom

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
    - `t x : int` "x" : int "t"` x = ... x : int "t"

Terminology Review

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

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

Lattices of Functions

- Let $(P, \leq)$ be a lattice
- Let $M$ be the set of monotonic functions on $P$
- Define $f \leq g$ if for all $x$, $f(x) = g(x)$
- Define the function $f \sqcap g$ as
  - $(f \sqcap g)(x) = f(x) \sqcap g(x)$

- Claim: $(M, \leq)$ forms a lattice
Elimination Methods: Conditionals

\[
\text{ite} = (\text{then} \circ \text{if}) \uparrow (\text{else} \circ \text{if})
\]
\[
\text{Out(if)} = f_{\text{if}}(\text{In(ite))})
\]
\[
\text{Out(then)} = (\text{then} \circ \text{if})(\text{In(ite))})
\]
\[
\text{Out(else)} = (\text{else} \circ \text{if})(\text{In(ite))})
\]

Elimination Methods: Loops

\[
\text{while} = f_{\text{head}}^{\uparrow}
\]
\[
\text{while} \circ f_{\text{body}} \circ f_{\text{while}} \circ f_{\text{head}}
\]

Elimination Methods: Loops (cont’d)

• Let \( f = f_0 \circ \ldots \circ f_i \) (i times)
  \[ \hat{f} = \text{id} \]
  \[ \hat{f} = f_0 \]
• Let \( g(j) = \text{while} \circ f_{\text{head}} \circ f_{\text{body}}^{\uparrow} \circ f_{\text{while}} \)
• Need to compute limit as \( j \) goes to infinity
  \[ \text{Does such a thing exist?} \]
  \[ \text{Observe: } g(j+1) \leq g(j) \]

Height of Function Lattice

• Assume underlying lattice \((P, \leq)\) has finite height
  \[ \text{What is height of lattice of monotonic functions?} \]
  \[ \text{Claim: finite (see homework)} \]
• Therefore, \( g(j) \) converges

Non-Reducible Flow Graphs

• Elimination methods usually only applied to reducible flow graphs
  \[ \text{Ones that can be collapsed} \]
  \[ \text{Standard constructs yield only reducible flow graphs} \]
• Unrestricted goto can yield irreducible graphs

Comments

• Can also do backwards elimination
  \[ \text{Not quite as nice (regions are usually single entry but often not single exit)} \]
• For bit vector problems, elimination efficient
  \[ \text{Easy to compose functions, compute meet, etc.} \]
• Elimination originally seemed like it might be faster than iteration
  \[ \text{Not really the case} \]
**Data Flow Analysis and Functions**

- What happens at a function call?
  - Lots of 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: `*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

**Data Flow Analysis and Optimization**

- Moore’s Law: Hardware advances double computing power every 18 months.
- Proebsting’s Law: Compiler advances double computing power every 18 years.
- We’ll focus on other uses of data flow analysis in this class (later in the semester)