Program Analysis

The Purpose of this Course

• How are the following related?
  - Program analysis
  - Model checking (as applied to software)
  - Theorem proving (as applied to software)

• But program analysis itself has sub-disciplines

What is Program Analysis?

• A collection of communities:
  - Dataflow analysis
  - Abstract interpretation
  - Type inference
  - Constraint-based analysis

• The relationships among these are not completely clear

What is Program Analysis For?

• Historically: Optimizing compilers

• More recently:
  - Influencing language design
  - Finding bugs

Culture

• Emphasis on low-complexity techniques
  - Because of emphasis on usage in tools
  - High-complexity techniques also studied, but often don’t survive

• Emphasis on complete automation

• Driven by language features
  - Particular languages and features give rise to their own sub-disciplines

Dataflow Analysis

Part 1
Control-Flow Graphs

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

Control-flow graphs are state-transition systems.

Notation

- \(s\) is a statement
- \(\text{succ}(s) = \{\ \text{successor statements of } s\}\)
- \(\text{pred}(s) = \{\ \text{predecessor statements of } s\}\)
- \(\text{write}(s) = \{\ \text{variables written by } s\}\)
- \(\text{read}(s) = \{\ \text{variables read by } s\}\)

Note: In literature, \(\text{write} = \text{kill}\) and \(\text{read} = \text{gen}\)

Available Expressions

- For each program point \(p\), which expressions must have already been computed, and not later modified, on all paths to \(p\).
- Optimization: Where available, expressions need not be recomputed.

Dataflow Equations

\[
A_b(s) = \begin{cases} 
\emptyset & \text{if } \text{pred}(s) = \emptyset \\
\bigcap_{s' \in \text{pred}(s)} A_{b'}(s') & \text{otherwise}
\end{cases}
\]

\[
A_{\text{out}}(s) = (A_b(s) \setminus \{a \in S \mid \text{write}(s) \cap V(a) = \emptyset\}) \\
\cup \{S \mid \text{write}(s) \setminus \text{read}(s) = \emptyset\}
\]

Example

Liveness Analysis

- For each program point \(p\), which of the variables defined at that point are used on some execution path?
- Optimization: If a variable is not live, no need to keep it in a register.
Dataflow Equations

\[ L_{in}(s) = (L_{out}(s) - \text{write}(s)) \cup \text{read}(s) \]

\[ L_{out}(s) = \begin{cases} \emptyset & \text{if } \text{succ}(s) = \emptyset \\ \bigcup_{s' \in \text{succ}(s)} L_{in}(s') & \text{otherwise} \end{cases} \]

Example

Available Expressions Again

\[ A_{in}(s) = \begin{cases} \emptyset & \text{if } \text{pred}(s) = \emptyset \\ \bigcap_{s \in \text{pred}(s)} A_{out}(s') & \text{otherwise} \end{cases} \]

\[ A_{out}(s) = (A_{in}(s) - \{ a \in S | \text{write}(s) \cap V(a) \neq \emptyset \}) \cup \{ s | \text{write}(s) \cap \text{read}(s) = \emptyset \} \]

Available Expressions: Schematic

\[ A_{in}(s) = \begin{cases} \emptyset & \text{if } \text{pred}(s) = \emptyset \\ \bigcap_{s \in \text{pred}(s)} A_{out}(s') & \text{otherwise} \end{cases} \]

\[ A_{out}(s) = \bigcup_{s \in \text{pred}(s)} A_{in}(s') \]

Transfer function:

Must analysis: property holds on all paths
Forwards analysis: from inputs to outputs

Live Variables Again

\[ L_{in}(s) = (L_{out}(s) - \text{write}(s)) \cup \text{read}(s) \]

\[ L_{out}(s) = \begin{cases} \emptyset & \text{if } \text{succ}(s) = \emptyset \\ \bigcup_{s' \in \text{succ}(s)} L_{in}(s') & \text{otherwise} \end{cases} \]

Live Variables: Schematic

\[ L_{in}(s) = \bigcup_{s' \in \text{succ}(s)} L_{in}(s') \]

\[ L_{out}(s) = \bigcup_{s' \in \text{succ}(s)} L_{in}(s') \]

Transfer function:

May analysis: property holds on some path
Backwards analysis: from outputs to inputs
**Very Busy Expressions**

- An expression $e$ is very busy at program point $p$ if every path from $p$ must evaluate $e$ before any variable in $e$ is redefined
- Optimization: hoisting expressions
- A must-analysis
- A backwards analysis

**Reaching Definitions**

- For a program point $p$, which assignments made on paths reaching $p$ have not been overwritten
- Connects definitions with uses (use-def chains)
- A may-analysis
- A forwards analysis

---

**One Cut at the Dataflow Design Space**

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

---

**The Literature**

- Vast literature of dataflow analyses
- 90+% can be described by
  - Forwards or backwards
  - May or must
- Some oddballs, but not many
  - Bidirectional analyses

---

**Another Cut at Dataflow Design**

- What theory are we dealing with?
- Review our schemas:
  $$ A_n(s) = \bigcap_{s' \in \text{pred}(s)} A_w(s') $$
  $$ L_n(s) = L_{\text{next}}(s) - C_1 \cup C_2 $$
  $$ A_w(s) = A_n(s) - C_1 \cup C_2 $$
  $$ L_{\text{next}}(s) = \bigcup_{s' \in \text{succ}(s)} L_n(s') $$

---

**Essential Features**

- Set variables $L_n(s)$, $L_{\text{next}}(S)$
- Set operations: union, intersection
  - Restricted complement ($\cdot$ constant)
- Domain of atoms
  - E.g., variable names
- Equations with single variable on lhs
Dataflow Problems

- Many dataflow equations are described by the grammar:
  \[ E = E \cap E | E \cup E | v \ | a \]
  \[ EQS \rightarrow v = E; EQS | e \]

- \( v \) is a variable
- \( a \) is an atom
- Note: More general than most problems...

Solving Dataflow Equations

- Simple worklist algorithm:
  - Initially let \( S(v) = 0 \) for all \( v \)
  - Repeat until \( S(v) = S(E) \) for all equations
  - Pick any \( v = E \) such that \( S(v) = S(E) \)
  - Set \( S = S(v) \cap S(E) \)

Termination

- How do we know the algorithm terminates?
- Because
  - operations are monotonic
  - the domain is finite

Monotonicity

- Operation \( f \) is monotonic if
  \[ X \subseteq Y \Rightarrow f(x) \subseteq f(y) \]
- We require that all operations be monotonic
  - Easy to check for the set operations
  - Easy to check for all transfer functions; recall:
    \[ L_0(s) = L_{\text{pre}}(s) - C_1 \cup C_2 \]

Termination again

- To see the algorithm terminates
  - All variables start empty
  - Variables and rhs's only increase with each update
    - By induction on \# of updates, using monotonicity
    - Sets can only grow to a max finite size
- Together, these imply termination

The Rest of the Lecture

- Distributive Problems
- Flow Sensitivity
- Context Sensitivity
  - Or interprocedural analysis
- What are the limits of dataflow analysis?
Distributive Dataflow Problems

- Monotonicity implies for a transfer function $f$:
  $$ f(x \uplus y) \geq f(x) \uplus f(y) $$
- Distributive dataflow problems satisfy a stronger property:
  $$ f(x \uplus y) = f(x) \uplus f(y) $$

Meet Over All Paths

- If a dataflow problem is distributive, then the (least) solution of the dataflow equations is equivalent to the analyzing every path (including infinite ones) and combining the results.
- Says joins cause no loss of information.

What Problems are Distributive?

- Many analyses of program structure are distributive:
  - E.g., live variables, available expressions, reaching definitions, very busy expressions
  - Properties of how the program computes.

Distributivity Example

- The analysis of the graph is equivalent to combining the analysis of each path.

Distributivity Again

- Obtaining the meet over all paths solution is a very powerful guarantee.
- Says that dataflow analysis is really as good as you can do for a distributive problem.
- Alternatively, can be viewed as saying distributive problems are very easy indeed . . .

Liveness Example Revisited

- Many analyses of program structure are distributive:
  - E.g., live variables, available expressions, reaching definitions, very busy expressions
  - Properties of how the program computes.
Constant Folding

- Ordering $i+j$ for any integer $i$
- $j||k = \top$ if $j = k$
- Example transfer function:
  \[
  C(v = e \cdot e)\sigma = (v \leftarrow C(e)\sigma) \otimes C(e)\sigma
  \]
  where $a \otimes b = \begin{cases} a \times b & \text{if } a, b \text{ constants} \\ \top & \text{otherwise} \end{cases}$
- Consider:
  \[
  C(x = y \cdot y)\sigma\{y = 1\} \cup C(x = y \cdot y)\sigma\{y = -1\}
  \]
  \[
  C(x = y \cdot y)\sigma\{y = 1\} \cup \{y = -1\}\}
  \]

What Problems are Not Distributive?

- Analyses of what the program computes
  - The output is (a constant, positive, ...)

Flow Sensitivity

- Flow sensitive analyses
  - The order of statements matters
  - Need a control flow graph
    - Or transition system, ...
- Flow insensitive analyses
  - The order of statements doesn’t matter
  - Analysis is the same regardless of statement order

Example Flow Insensitive Analysis

- What variables does a program fragment modify?
  \[
  G(x := e) = \{x\}
  \]
  \[
  G(s; s_r) = G(s_r) \cup G(s_l)
  \]
- Note $G(s; s_r) = G(s_l; s)$

The Advantage

- Flow-sensitive analyses require a model of program state at each program point
  - E.g., liveness analysis, reaching definitions, ...
- Flow-insensitive analyses require only a single global state
  - E.g., for $G$, the set of all variables modified

Notes on Flow Sensitivity

- Flow insensitive analyses seem weak, but:
- Flow sensitive analyses are hard to scale to very large programs
  - Additional cost: state size $X \#$ of program points
- Beyond 1000’s of lines of code, only flow insensitive analyses have been shown to scale
**Context-Sensitive Analysis**

- What about analyzing across procedure boundaries?
  
  \[ \text{Def } f(x)[..] \]
  \[ \text{Def } g(y)[..f(a)[..] \]
  \[ \text{Def } h(z)[..f(b)[..] \]

- Goal: Specialize analysis of \( f \) to take advantage of
  
  - \( f \) is called with \( a \) by \( g \)
  - \( f \) is called with \( b \) by \( h \)

**Control-Flow Graphs Again**

- How do we extend control-flow graphs to procedures?
  
  - Idea: Model procedure call \( f(a) \) by:
    - Edge from point before call to entry of \( f \)
    - Edge from exit(s) of \( f \) to point after call

**Example**

- Edges from
  
  - before \( f(a) \) to entry of \( f \)
  - Exit of \( f \) to after \( f(a) \)
  - Before \( f(b) \) to entry of \( f \)
  - Exit of \( f \) to after \( f(b) \)

- Has the correct flows for \( g \)

**Example**

- Edges from
  
  - before \( f(a) \) to entry of \( f \)
  - Exit of \( f \) to after \( f(a) \)
  - Before \( f(b) \) to entry of \( f \)
  - Exit of \( f \) to after \( f(b) \)

- Has the correct flows for \( h \)

**Example**

- But also has flows we don’t want
  
  - One path captures a call to \( g \) returning at \( h \)

- So-called “infeasible paths”
What to do?

- Must distinguish calls to \( f \) in different contexts
- Three techniques
  - Assumptions
  - Lifting
  - Context-free reachability
  - Call strings
  - Today

Call Strings

- Observation:
  - At run time, different calls to \( f \) are distinguished by the call stack
- Problem:
  - The stack is unbounded
- Idea:
  - Use the last \( k \) calls on the stack to distinguish context
  - Represent a call by the name of the calling procedure

Example Revisited

- Use call strings of length 1
- Context is name of calling procedure

Experience with Call Strings

- Very expensive
  - Multiplies \# of abstract values by (\# of procedures \( \times \) length of call string)
  - Hard to contemplate call strings \( > 1 \)
- Fragile
  - Very sensitive to organization of procedures
- Well-studied, but not much used in practice

Review of Terminology

- Must vs. May
- Forwards vs. Backwards
- Flow-sensitive vs. Flow-insensitive
- Context-sensitive vs. Context-insensitive
- Distributive vs. non-Distributive

Where is Dataflow Analysis Useful?

- Best for flow-sensitive, context-insensitive, distributive problems on small pieces of code
  - E.g., the examples we've seen and many others
- Extremely efficient algorithms are known
  - Use different representation than control-flow graph, but not fundamentally different
  - More on this in a minute...
Where is Dataflow Analysis Weak?

- Lots of places

Data Structures

- Not good at analyzing data structures
- Works well for atomic values
  - Labels, constants, variable names
- Not easily extended to arrays, lists, trees, etc.
  - Work on shape analysis

The Heap

- Good at analyzing flow of values in local variables
- No notion of the heap in traditional dataflow applications
- In general, very hard to model anonymous values accurately
  - Aliasing
  - The "strong update" problem

Context Sensitivity

- Standard dataflow techniques for handling context sensitivity don’t scale well
- Brittle under common program edits
- E.g., call strings

Flow Sensitivity (Beyond Procedures)

- Flow sensitive analyses are standard for analyzing single procedures
- Not used (or not aware of uses) for whole programs
  - Too expensive

The Call Graph

- Dataflow analysis requires a call graph
  - Or something close
- Inadequate for higher-order programs
  - First class functions
  - Object-oriented languages with dynamic dispatch
- Call-graph hinders algorithmic efficiency
  - Desire to keep executable specification is limiting
Forwards vs. Backwards

- Restriction to forwards/backwards reachability
  - Very constraining
  - Many important problems not easy to fit into this mold

Next Time: Abstract Interpretation

- Theory
  - Lots
- Examples
  - Lots
- Focus on contrast with traditional dataflow analysis