Program Analysis
What is Program Analysis For?

• Historically: Optimizing compilers

• More recently:
  - 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

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

$s$ is a statement

\begin{align*}
succ(s) &= \{ \text{successor statements of } s \} \\
pred(s) &= \{ \text{predecessor statements of } s \} \\
write(s) &= \{ \text{variables written by } s \} \\
read(s) &= \{ \text{variables read by } s \} \\
\end{align*}

Note: In literature write = kill and read = 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

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

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

if $y > a + b$

$a := a + 1$

$x := a + b$

$y := a \times b$

$\text{if } y > a + b$

$a := a + 1$

$x := a + b$
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.

```plaintext
if y > a + b
    x := a + b
    y := a * b
    a := a + 1
    x := a + b
x is not live here
```
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

if \( y > a + b \)
\[
\begin{align*}
    \text{a} & := \text{a} + 1 \\
    \text{x} & := \text{a} + \text{b} \\
    \text{y} & := \text{a} \times \text{b} \\
\end{align*}
\]
Available Expressions Again

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

\[ A_{out}(s) = (A_{in}(s) - \{ a \in S \mid \text{write}(s) \cap V(a) \neq \emptyset \}) \cup \{ s \mid \text{write}(s) \cap \text{read}(s) = \emptyset \} \]
Available Expressions: Schematic

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

Transfer function:
\[ A_{out}(s) = A_{in}(s) - C_1 \cup C_2 \]

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

Transfer function:

\[ L_{\text{in}}(s) = L_{\text{out}}(s) - C_1 \cup C_2 \]

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

*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><strong>Forwards</strong></td>
<td>Reaching definitions</td>
<td>Available expressions</td>
</tr>
<tr>
<td><strong>Backwards</strong></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
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?

\[ G(x := e) = \{x\} \]
\[ G(s_1; s_2) = G(s_1) \cup G(s_2) \]

- Note \( G(s_1; s_2) = G(s_2; s_1) \)
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 × # 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?

  Def f(x){…}
  Def g(y){…f(a)…}
  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 $f(b)$
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”

• Must distinguish calls to $f$ in different contexts
Review of Terminology

- **Must vs. May**
- **Forwards vs. Backwards**
- **Flow-sensitive vs. Flow-insensitive**
- **Context-sensitive vs. Context-insensitive**
Where is Dataflow Analysis Useful?

• Best for flow-sensitive, context-insensitive 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
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