Dataflow Analysis

17-654/17-754
Analysis of Software Artifacts

Jonathan Aldrich

Overview: Analyses We’ve Seen

• AST walker analyses
  • e.g. assignment inside an if statement
  • Very approximate, very local
    • Misses case where accidental assignment is done outside an if

• Hoare logic
  • Useful for proving correctness
  • Requires a lot of work (even for ESC/Java)
  • Automated tool is unsound
    • So is manual proof, without a proof checker
Motivation: Dataflow Analysis

- Catch interesting errors
  - Non-local: x is null, x is written to y, y is dereferenced
- Optimize code
  - Reduce run time, memory usage...
- Soundness required
  - Safety-critical domain
    - Assure lack of certain errors
  - Cannot optimize unless it is proven safe
    - Correctness comes before performance
- Automation required
  - Dramatically decreases cost
  - Makes cost/benefit worthwhile for far more purposes

Dataflow analysis

- Tracks value flow through program
  - Can distinguish order of operations
    - Did you read the file after you closed it?
    - Does this null value flow to that dereference?
  - Differs from AST walker
    - Walker simply collects information or checks patterns
      - Tracking flow allows more interesting properties
- Abstracts values
  - Chooses abstraction particular to property
    - Is a variable null?
    - Is a file open or closed?
    - Could a variable be 0?
    - Where did this value come from?
  - More specialized than Hoare logic
    - Hoare logic allows any property to be expressed
      - Specialization allows automation and soundness
Zero Analysis

- Could variable \( x \) be 0?
  - Useful to know if you have an expression \( y/x \)
  - In C, useful for null pointer analysis
- Program semantics
  - \( \eta \) maps every variable to an integer
- Semantic abstraction
  - \( \sigma \) maps every variable to non zero (NZ), zero(Z), or maybe zero (MZ)
  - Abstraction function for integers \( \alpha_{\mathbb{Z}} \):
    - \( \alpha_{\mathbb{Z}}(0) = \mathbb{Z} \)
    - \( \alpha_{\mathbb{Z}}(n) = \text{NZ for all } n \neq 0 \)
  - We may not know if a value is zero or not
    - Analysis is always an approximation
    - Need MZ option, too

Zero Analysis Example

\[ \sigma = [] \]

\[
\begin{align*}
  x & := 10; \\
  y & := x; \\
  z & := 0; \\
  \text{while } y > -1 \text{ do} \\
  & \quad x := x / y; \\
  & \quad y := y-1; \\
  & \quad z := 5;
\end{align*}
\]
Zero Analysis Example

\[ \sigma = [\] \]

\[ x := 10; \]
\[ y := x; \]
\[ z := 0; \]

while \( y > -1 \) do
  \[ x := x / y; \]
  \[ y := y-1; \]
  \[ z := 5; \]
Zero Analysis Example

\( \sigma = [] \)

\( x := 10; \quad \sigma = [x \mapsto \text{NZ}] \)

\( y := x; \quad \sigma = [x \mapsto \text{NZ}, y \mapsto \text{NZ}] \)

\( z := 0; \quad \sigma = [x \mapsto \text{NZ}, y \mapsto \text{NZ}, z \mapsto Z] \)

while \( y > -1 \) do

\( x := x \div y; \quad \sigma = [x \mapsto \text{NZ}, y \mapsto \text{NZ}, z \mapsto Z] \)

\( y := y - 1; \quad \sigma = [x \mapsto \text{NZ}, y \mapsto \text{MZ}, z \mapsto Z] \)

\( z := 5; \quad \sigma = [x \mapsto \text{NZ}, y \mapsto \text{MZ}, z \mapsto \text{NZ}] \)

Zero Analysis Example

\( \sigma = [] \)

\( x := 10; \quad \sigma = [x \mapsto \text{NZ}] \)

\( y := x; \quad \sigma = [x \mapsto \text{NZ}, y \mapsto \text{NZ}] \)

\( z := 0; \quad \sigma = [x \mapsto \text{NZ}, y \mapsto \text{NZ}, z \mapsto Z] \)

while \( y > -1 \) do

\( x := x \div y; \quad \sigma = [x \mapsto \text{NZ}, y \mapsto \text{NZ}, z \mapsto Z] \)

\( y := y - 1; \quad \sigma = [x \mapsto \text{NZ}, y \mapsto \text{MZ}, z \mapsto Z] \)

\( z := 5; \quad \sigma = [x \mapsto \text{NZ}, y \mapsto \text{MZ}, z \mapsto \text{NZ}] \)
Zero Analysis Example

\[
\begin{align*}
\sigma &= [] \\
\sigma &= [x\mapsto NZ] \\
\sigma &= [\top] \\
\end{align*}
\]

x := 10; 
\sigma = [x\mapsto NZ] 
\sigma = [\top] 
y := x; 
\sigma = [x\mapsto NZ, y\mapsto NZ] 
\sigma = [\top] 
z := 0; 
\sigma = [x\mapsto NZ, y\mapsto NZ, z\mapsto \top] 
\sigma = [\top] 
while y > -1 do 
\sigma = [x\mapsto NZ, y\mapsto MZ, z\mapsto MZ] 
\sigma = [\top] 
x := x / y; 
\sigma = [x\mapsto NZ, y\mapsto MZ, z\mapsto MZ] 
\sigma = [\top] 
y := y - 1; 
\sigma = [x\mapsto NZ, y\mapsto MZ, z\mapsto MZ] 
\sigma = [\top] 
z := 5; 
\sigma = [x\mapsto NZ, y\mapsto MZ, z\mapsto NZ] 
\sigma = [\top] 
\]

Nothing more happens!
Zero Analysis Termination

- The analysis values will not change, no matter how many times we execute the loop
  - Proof: our analysis is deterministic
  - We run through the loop with the current analysis values, none of them change
  - Therefore, no matter how many times we run the loop, the results will remain the same
  - Therefore, we have computed the dataflow analysis results for any number of loop iterations
- Why does this work
  - If we simulate the loop, the data values could (in principle) keep changing indefinitely
    - There are an infinite number of data values possible
    - Not true for 32-bit integers, but might as well be true
    - Counting to $2^{32}$ is slow, even on today’s processors
  - Dataflow analysis only tracks 2 possibilities!
    - So once we’ve explored them all, nothing more will change
    - This is the secret of abstraction
- We will make this argument more precise later

Using Zero Analysis

- Visit each division in the program
- Get the results of zero analysis for the divisor
- If the results are definitely zero, report an error
- If the results are possibly zero, report a warning
Defining Dataflow Analyses

- **Lattice**
  - Describes program data abstractly
  - Abstract equivalent of environment
- **Abstraction function**
  - Maps concrete environment to lattice element
- **Flow functions**
  - Describes how abstract data changes
  - Abstract equivalent of expression semantics
- **Control flow graph**
  - Determines how abstract data propagates from statement to statement
  - Abstract equivalent of statement semantics

---

Lattice

- A lattice is a tuple \((L, \sqsubseteq, \sqcup, \bot, \top)\)
  - \(L\) is a set of abstract elements
  - \(\sqsubseteq\) is a partial order on \(L\)
    - Means at least as precise as
  - \(\sqcup\) is the least upper bound of two elements
    - Must exist for every two elements in \(L\)
    - Used to merge two abstract values
  - \(\bot\) (bottom) is the least element of \(L\)
    - Means we haven’t yet analyzed this yet
    - Will become clear later
  - \(\top\) (top) is the greatest element of \(L\)
    - Means we don’t know anything
  - \(L\) may be infinite
    - Typically should have finite height
      - All paths from \(\bot\) to \(\top\) should be finite
      - We’ll see why later

---

Analysis of Software Artifacts - Spring 2006
Is this a lattice?

- A lattice is a tuple \((L, \leq, \bot, \top)\)
- \(L\) is a set of abstract elements
- \(\leq\) is a partial order on \(L\)
- \(\bot\) is the least upper bound of two elements
  - must exist for every two elements in \(L\)
- \(\bot\) (bottom) is the least element of \(L\)
- \(\top\) (top) is the greatest element of \(L\)
- Yes!

---

Is this a lattice?

- A lattice is a tuple \((L, \leq, \bot, \top)\)
- \(L\) is a set of abstract elements
- \(\leq\) is a partial order on \(L\)
- \(\bot\) is the least upper bound of two elements
  - must exist for every two elements in \(L\)
- \(\bot\) (bottom) is the least element of \(L\)
- \(\top\) (top) is the greatest element of \(L\)
- No!
  - No bottom element
    - \(\bot\) is not least in the lattice order
    - It is mis-named
Is this a lattice?

- A lattice is a tuple \((L, \sqsubseteq, \sqcup, \bot, \top)\)
  - \(L\) is a set of abstract elements
  - \(\sqsubseteq\) is a partial order on \(L\)
  - \(\sqcup\) is the least upper bound of two elements
    - must exist for every two elements in \(L\)
  - \(\bot\) (bottom) is the least element of \(L\)
  - \(\top\) (top) is the greatest element of \(L\)

Definition: Least Upper Bounds

- \(x \sqcup y = z\) iff
  - \(z\) is an upper bound of \(x\) and \(y\)
    - \(x \sqsubseteq z\) and \(y \sqsubseteq z\)
  - \(z\) is the least such bound
    - \(\forall \forall w \in L\) such that \(x \sqsubseteq w\) and \(y \sqsubseteq w\) we have \(z \sqsubseteq w\)
  - Also called a join

Not a lattice

- What is \(c \sqcup d\)?
  - \(a, b,\) and \(\top\) are upper bounds
    - Assume \(\sqsubseteq\) is transitive
  - None is least upper bound
Is this a lattice?

- A lattice is a tuple \((L, \sqsubseteq, \sqcup, \bot, \top)\)
  - \(L\) is a set of abstract elements
  - \(\sqsubseteq\) is a partial order on \(L\)
  - \(\sqcup\) is the least upper bound of two elements
    - must exist for every two elements in \(L\)
  - \(\bot\) (bottom) is the least element of \(L\)
  - \(\top\) (top) is the greatest element of \(L\)
- Yes!

---

Zero Analysis Lattice

- Integer zero lattice
  - \(L_{\mathbb{Z}} = \{ \bot, Z, \mathbb{N}Z, MZ \}\)
  - \(\bot \in \mathbb{Z}, \bot \subseteq \mathbb{N}Z, \mathbb{N}Z \subseteq MZ, Z \subseteq MZ\)
    - \(\bot \subseteq \mathbb{N}Z\) holds by transitivity
  - \(\sqcup\) defined as join for \(\sqsubseteq\)
    - \(x \sqcup y = z\) iff
      - \(z\) is an upper bound of \(x\) and \(y\)
      - \(z\) is the least such bound
    - Obeys laws: \(\bot \sqcup \mathcal{X} = \mathcal{X}, \top \sqcup \mathcal{X} = \top, \mathcal{X} \sqcup \mathcal{X} = \mathcal{X}\)
      - \(\mathbb{N}Z \sqcup \mathbb{N}Z = \mathbb{N}Z\)
  - \(\bot = \bot\)
    - \(\forall \mathcal{X}, \bot \subseteq \mathcal{X}\)
  - \(\top = MZ\)
    - \(\forall \mathcal{X}, \mathcal{X} \subseteq \top\)
Zero Analysis Lattice

- Integer zero lattice
  - $L_{ZI} = \{ \bot, Z, NZ, MZ \}$
  - $\bot \subseteq Z, \bot \subseteq NZ, NZ \subseteq MZ, Z \subseteq MZ$
  - $\cup$ defined as join for $\subseteq$
  - $\bot = \bot$
  - $T = MZ$

- Program lattice is a tuple lattice
  - $L_Z$ is the set of all maps from $\text{Var}$ to $L_{ZI}$
  - $\sigma_1 \equiv_Z \sigma_2$ iff $\forall x \in \text{Var} \sigma_1(x) \equiv_{ZI} \sigma_2(x)$
  - $\sigma_1 \cup_Z \sigma_2 = \{ x \mapsto \sigma_1(x) \cup_{ZI} \sigma_2(x) \mid x \in \text{Var} \}$
  - $\bot = \{ x \mapsto \bot_{ZI} \mid x \in \text{Var} \}$
  - $T = \{ x \mapsto T_{ZI} \mid x \in \text{Var} \} = \{ x \mapsto MZ \mid x \in \text{Var} \}$
  - Can produce a tuple lattice from any base lattice
    - Just define as above

Tuple Lattices Visually

- For $\text{Var} = \{ x, y \}$
Abstraction Function

- Maps each concrete program state to a lattice element
  - For tuple lattices, the function can be defined for values and lifted to tuples
- Integer Zero abstraction function $\alpha_{ZI}$:
  - $\alpha_{ZI}(0) = Z$
  - $\alpha_{ZI}(n) = NZ$ for all $n \neq 0$
- Zero Analysis abstraction function $\alpha_{ZA}$:
  - $\alpha_{ZA}(\eta) = \{ x \mapsto \alpha_{ZI}(\eta(x)) \mid x \in \textbf{Var} \}$
  - This is just the tuple form of $\alpha_{ZI}(n)$
    - Can be done for any tuple lattice
Control Flow Graph (CFG)

- Shows order of statement execution
  - Determines where data flows
- Decomposes expressions into primitive operations
  - Crystal: One CFG node per “useful” AST node
    - constants, variables, binary operations, assignments, if, while…
  - Loops are written out
    - Form a loop in the CFG
  - Benefit: analysis is defined one operation at a time

Intuition for Building a CFG

- Connect nodes in order of operation
  - Defined by language
- Java order of operation
  - Expressions, assignment, sequence
    - Evaluate subexpressions left to right
    - Evaluate node after children (postfix)
  - While, If
    - Evaluate condition first, then if/while
    - if branches to else and then
    - while branches to loop body and exit
Control Flow Graph Example

while $i \times 2 < 10$ do
    if $x < i + 2$
        then $x := x + 5$
    else $i := i + 1$
END

Flow Functions

- Compute dataflow information after a statement from dataflow information before the statement
  - Formally, map a lattice element and a CFG node to a new lattice element
- Analysis performed on 3-address code
  - inspired by 3 addresses in assembly language: add $x,y,z$
- Convert complex expressions to 3-address code
  - Each subexpression represented by a temporary variable
  - $x + 3 \times y \rightarrow t_1 := 3; t_2 := t_1 \times y; t_3 := x + t_2$
Zero Analysis Flow Functions

- \( f_{ZA}(\sigma, [x := y]) = [x \mapsto \sigma(y)] \sigma \)
- \( f_{ZA}(\sigma, [x := n]) = \) if \( n=0 \)
  - then \([x \mapsto Z]\sigma\)
  - else \([x \mapsto NZ]\sigma\)
- \( f_{ZA}(\sigma, [x := \ldots]) = [x \mapsto \text{MZ}] \sigma \)
  - Could be more precise, e.g.
    \( f_{ZA}(\sigma, [x := y + z]) = \)
    - if \( \sigma[y] = Z \) \&\& \( \sigma[z] = Z \)
    - then \([x \mapsto Z]\sigma\) else \([x \mapsto \text{MZ}]\sigma\)
- \( f_{ZA}(\sigma, /*\text{any non-assignment}*/\sigma) = \sigma \)
Zero Analysis Example

x := 0;
while x > 3 do
  x := x + 1

Intuition:
We know nothing about initial variable values. We could use a precondition if we had one.
Zero Analysis Example

\[ \sigma_i = \{ x \to \text{MZ} \mid x \in \text{Var} \} \]

BEGIN \qquad \text{END}

\[ \sigma_2 = f_{ZA}(\sigma, [t_2 := 0]) \]
\[ = [t_2 \to \text{Z}] \sigma_i \]

BEGIN \qquad \text{END}

\[ f_{ZA}(\sigma, [x := n]) = \]
\[ \text{if } n = 0 \]
\[ \text{then } [x \to \text{Z}]\sigma \]
\[ \text{else } [x \to \text{NZ}]\sigma \]

Zero Analysis Example

\[ \sigma_i = \{ x \to \text{MZ} \mid x \in \text{Var} \} \]

BEGIN \qquad \text{END}

\[ \sigma_2 = [t_2 \to \text{Z}] \sigma_i \]

BEGIN \qquad \text{END}

\[ \sigma_3 = f_{ZA}(\sigma_2, [x := t_2]) \]
\[ = [x \to \sigma_2(t_2)] \sigma_2 \]
\[ = [x \to \text{Z}] \sigma_2 \]
\[ = [x \to \text{Z}, t_2 \to \text{Z}] \sigma \]

BEGIN \qquad \text{END}
Zero Analysis Example

\[
\begin{align*}
\sigma_i &= \{ x \mapsto \text{MZ} \mid x \in \text{Var} \} \quad \text{BEGIN} \\
\sigma_3 &= [x \mapsto Z, t_2 \mapsto Z] \sigma_i \\
\sigma_{12} &= \perp \\
\sigma_5 &= f_{ZA}(\sigma_3 \cup \sigma_{12}, [t_5 := 3]) \\
&= f_{ZA}(\sigma_3 \cup \perp, [t_5 := 3]) \\
&= f_{ZA}(\sigma_3, [t_5 := 3]) \\
&= [t_2 \mapsto \text{NZ}] \sigma_5 \\
f_{ZA}(\sigma, [x := n]) &= \\
\text{if } n = 0 \\
&\quad \text{then } [x \mapsto Z] \sigma \\
\text{else } [x \mapsto \text{NZ}] \sigma \\
\end{align*}
\]

Input to \([3]_5\) comes from \([=]_3\) and \([=]_{12}\)
Input should be \(\sigma_3 \cup \sigma_{12}\)
What is \(\sigma_{12}\)?
Solution: assume \(\perp\)
Benefit: \(\sigma_3 \cup \perp = \sigma_3\)
Same result as ignoring back edge first time
Zero Analysis Example

\[
\sigma_1 = \{ x \rightarrow \text{MZ} \mid x \in \text{Var} \} \\
\sigma_3 = [x \rightarrow Z, t_2 \rightarrow Z] \sigma_1 \\
\sigma_{12} = \perp \\
\sigma_5 = [t_6 \rightarrow \text{NZ}] \sigma_3 \\
\sigma_6 = f_{ZA}(\sigma_6, [t_6 := x < t_6]) \\
\quad = \sigma_5 \\
\quad = [t_6 \rightarrow \text{NZ}] \sigma_3 \\
\]

\[f_{ZA}(\sigma, \text{ /* any other */ }) = \sigma\]

Zero Analysis Example

\[
\sigma_1 = \{ x \rightarrow \text{MZ} \mid x \in \text{Var} \} \\
\sigma_3 = [x \rightarrow Z, t_2 \rightarrow Z] \sigma_1 \\
\sigma_{12} = \perp \\
\sigma_5 = [t_6 \rightarrow \text{NZ}] \sigma_3 \\
\]

Skipping similar nodes...
Zero Analysis Example

\[ \sigma_1 = \{ x \mapsto \text{MZ} \mid x \in \text{Var} \} \]
\[ \sigma_3 = [x \mapsto Z, t_2 \mapsto Z] \sigma_1 \]
\[ \sigma_{12} = \perp \]
\[ \sigma_{10} = [t_{10} \mapsto \text{NZ}, \ldots] \sigma_3 \]
\[ \sigma_{11} = f_{ZA}(\sigma_{10}, [t_{11} \mapsto x + t_{10}]) \]
\[ = [t_{11} \mapsto \text{MZ}] \sigma_{10} \]
\[ f_{ZA}(\sigma, [x \mapsto y \ op \ z]) = [x \mapsto \text{MZ}] \sigma \]

Zero Analysis Example

\[ \sigma_1 = \{ x \mapsto \text{MZ} \mid x \in \text{Var} \} \]
\[ \sigma_3 = [x \mapsto Z, t_2 \mapsto Z] \sigma_1 \]
\[ \sigma_{12} = \perp \]
\[ \sigma_{11} = [t_{10} \mapsto \text{NZ}, t_{11} \mapsto \text{MZ}, \ldots] \sigma_3 \]
\[ \sigma_{12} = f_{ZA}(\sigma_{11}, [x \mapsto t_{11}]) \]
\[ = [x \mapsto \sigma_{11}(t_{11})] \sigma_{11} \]
\[ = [x \mapsto \text{MZ}] \sigma_{11} \]
\[ = [x \mapsto \text{MZ}, \ldots] \sigma_3 \]
\[ f_{ZA}(\sigma, [x \mapsto y]) = [x \mapsto \sigma(y)] \sigma \]
Zero Analysis Example

\[ \sigma_1 = \{ x \rightarrow \text{MZ} \mid x \in \text{Var} \} \]
\[ \sigma_3 = [x \rightarrow Z, t_2 \rightarrow Z] \sigma_1 \]
\[ \sigma_{12} = [x \rightarrow \text{MZ}, \ldots] \sigma_3 \]
\[ \sigma_5 = f_{ZA}(\sigma_5 \cup \sigma_{12}, [t_5 := 3]) \]
\[ = f_{ZA}([x \rightarrow \text{MZ}] \sigma_5, [t_5 := 3]) \]
\[ = [t_5 \rightarrow \text{NZ}] [x \rightarrow \text{MZ}, \ldots] \sigma_3 \]
\[ = [t_5 \rightarrow \text{NZ}, x \rightarrow \text{MZ}, \ldots] \sigma_3 \]
\[ f_{ZA}(\sigma, [x]_k) = [t_k \rightarrow \sigma(x)] \sigma \]

Zero Analysis Example

\[ \sigma_1 = \{ x \rightarrow \text{MZ} \mid x \in \text{Var} \} \]
\[ \sigma_3 = [x \rightarrow Z, t_2 \rightarrow Z] \sigma_1 \]
\[ \sigma_{12} = [x \rightarrow \text{MZ}, \ldots] \sigma_3 \]

Propagation of \( x \rightarrow \text{MZ} \) continues
\( \sigma_{12} \) does not change, so no need to iterate again
Worklist Dataflow Analysis Algorithm

\[ \text{worklist} = \text{new Set}(); \]
for all node indexes \( i \) do
\[ \text{results}[i] = \bot_{\text{A}}; \]
\[ \text{results}[\text{entry}] = \iota_{\text{A}}; \]
\[ \text{worklist}.\text{add}(\text{all nodes}); \]
while \((!\text{worklist}.\text{isEmpty}()) \) do
\[ i = \text{worklist}.\text{pop}(); \]
before = \( \bigsqcup_{k \in \text{pred}(i)} \text{results}[k] \);
after = \( f_{\text{A}}(\text{before}, \text{node}(i)) \);
if \((!(\text{after} \subseteq \text{results}[i]))\)
\[ \text{results}[i] = \text{after}; \]
for all \( k \in \text{succ}(i) \) do
\[ \text{worklist}.\text{add}(k); \]

Example of Worklist

\[ [a := 0]_1 \]
\[ [b := 0]_2 \]
while \([a < 2]_3 \) do
\[ [b := a]_4; \]
\[ [a := a + 1]_5; \]
\[ [a := 0]_6 \]

<table>
<thead>
<tr>
<th>Position</th>
<th>Worklist</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>MZ</td>
<td>MZ</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>Z</td>
<td>MZ</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>Z</td>
<td>Z</td>
</tr>
<tr>
<td>3</td>
<td>4, 6</td>
<td>Z</td>
<td>Z</td>
</tr>
<tr>
<td>4</td>
<td>5, 6</td>
<td>Z</td>
<td>Z</td>
</tr>
<tr>
<td>5</td>
<td>3, 6</td>
<td>MZ</td>
<td>Z</td>
</tr>
<tr>
<td>3</td>
<td>4, 6</td>
<td>MZ</td>
<td>Z</td>
</tr>
<tr>
<td>4</td>
<td>5, 6</td>
<td>MZ</td>
<td>MZ</td>
</tr>
<tr>
<td>5</td>
<td>3, 6</td>
<td>MZ</td>
<td>MZ</td>
</tr>
<tr>
<td>3</td>
<td>4, 6</td>
<td>MZ</td>
<td>MZ</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>MZ</td>
<td>MZ</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>Z</td>
<td>MZ</td>
</tr>
</tbody>
</table>

Control Flow Graph

1 → 2 → 3 → 6
4 → 5
Worklist Algorithm Performance

- Performance
  - Visits node whenever input gets less precise
    - up to $h$ = height of lattice
  - Propagates data along control flow edges
    - up to $e$ = max outbound edges per node
  - Assume lattice operation cost is $o$
  - Overall, $O(h*e*o)$
    - Typically $h,o,e$ bounded by $n$ = number of statements in program
    - $O(n^3)$ for many data flow analyses
    - $O(n^2)$ if you assume a number of edges per node is small
  - Good enough to run on a function
    - Usually not run on an entire program at once, because $n$ is too big