Lecture 21: Review of Program Analysis

17-355/17-665/17-819: Program Analysis

Rohan Padhye

December 4, 2025

* Course materials developed with Jonathan Aldrich and Claire Le Goues

What ~~is~~ was this course about?

- Program analysis is *the systematic examination of a program to determine its properties.*
- From 30,000 feet, this requires:
  - Precise program representations
  - Tractable, systematic ways to reason over those representations.
- ~~We will learn~~: What we learned:
  - How to unambiguously define the meaning of a program, and a programming language.
  - How to prove theorems about the behavior of particular programs.
  - How to use, build, and extend tools that do the above, automatically.

What ~~is~~ was this course about?

- Program analysis is *the systematic examination of a program to determine its properties*.
- Principal techniques:
  - **Dynamic:**
    - **Testing:** Direct execution of code on test data in a controlled environment.
    - **Analysis:** Tools extracting data from test runs.
  - **Static:**
    - **Inspection:** Human evaluation of code, design documents (specs and models), modifications.
    - **Analysis:** Tools reasoning about program behavior without executing it.
    - **Inference**: Statistical models of code (e.g., AI / ML)
  - …and their combination.

The Bad News: Rice's Theorem

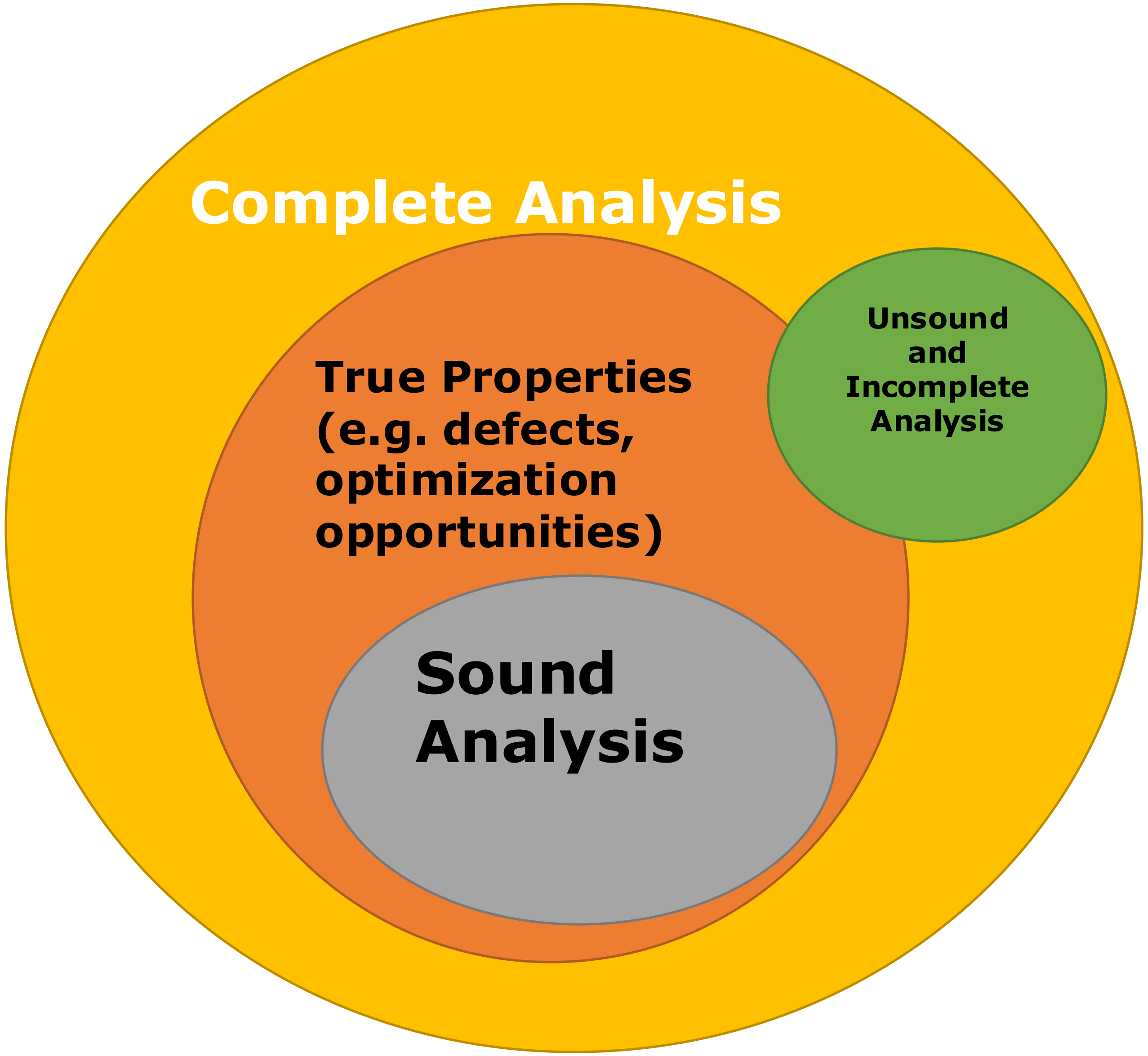
"Any nontrivial property about the language recognized by a Turing machine is undecidable.“

Henry Gordon Rice, 1953

Soundness and Completeness

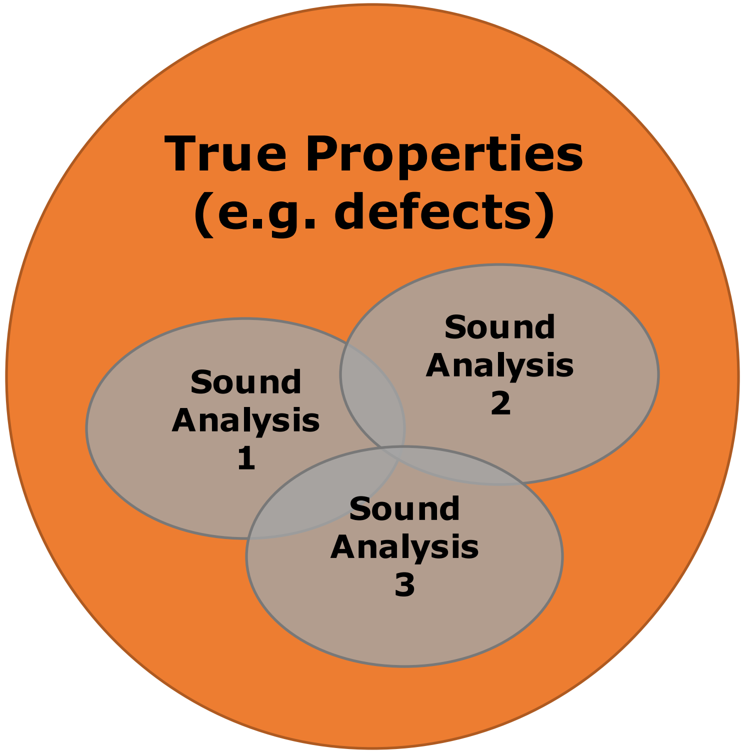
- An analysis is “sound” if every claim it makes is true
- An analysis is “complete” if it makes every true claim

- Soundness/Completeness correspond to under/over-approximation depending on context.
  - E.g. compilers and verification tools treat “soundness” as over-approximation since they make claims over all possible inputs
  - E.g. code quality tools often treat “sound” analyses as under-approximation because they make claims about existence of bugs


Complete Analysis
True Properties (e.g. defects, optimization opportunities)
Unsound and Incomplete Analysis
Sound Analysis

Soundness and Completeness Tradeoffs

- Sound + Complete is impossible in general (Rice's theorem)
- Most practical tools attempt to be either sound or complete for some specific application, using approximation
- Multiple classes of sound/complete techniques may exist, with trade-offs for accuracy and performance.
- Program analysis is a rich field because of the constant and never-ending battle to balance these trade-offs with ever-increasing software complexity



Fundamental concepts

- Abstraction
  - Elide details of a specific implementation.
  - Capture semantically relevant details; ignore the rest.
- The importance of semantics.
  - We prove things about analyses with respect to the semantics of the underlying language.
- Program proofs as inductive invariants.
- Implementation
  - You do not understand analysis until you have written several.

What you were supposed to get

- Beautiful and elegant theory
  - Mostly discrete mathematics, symbolic reasoning, inductive proofs
  - This is traditionally a “white-board” course [may not always use slides]
- Build awesome tools
  - Engineering of program analyses, compilers, and bug finding tools make great use of many fundamental ideas from computer science and software engineering
- New way to think about programs
  - Representations, control/data-flow, input state space
- Appreciate the limits and achievements in the space
  - What tools are *impossible* to build?
  - What tools are *impressive* that they exist at all?
  - When is it appropriate to use a particular analysis tool versus another?
  - How to interpret the results of a program analysis tool?

The WHILE language – Example program

```
y := x;
z := 1;
if y > 0 then
  while y > 1 do
    z := z * y;
    y := y - 1
else
  skip
```

- Sample program computes z = *x!* using *y* as a temp variable.
- WHILE uses assignment statements, if-then-else, while loops.
- All vars are integers.
- Expressions only arithmetic (for vars) or relational (for conditions).
- No I/O statements. Inputs and outputs are implicit.
  - Later on, we may use extensions with explicit `read x` and `print x`.

WHILE abstract syntax

S	statements
a	arithmetic expressions (AExp)
x, y	program variables (Vars)
n	number literals
b	boolean expressions (BExp)

We'll use these meta-variables frequently for ease of notation

$$
\begin{array}{lcl}
S & ::= & x := a \\
 & | & \text{skip} \\
 & | & S_1;\ S_2 \\
 & | & \text{if } b \text{ then } S_1 \text{ else } S_2 \\
 & | & \text{while } b \text{ do } S
\end{array}
\qquad
\begin{array}{lcl}
b & ::= & \text{true} \\
 & | & \text{false} \\
 & | & \text{not } b \\
 & | & b_1\ op_b\ b_2 \\
 & | & a_1\ op_r\ a_2
\end{array}
\qquad
\begin{array}{lcl}
a & ::= & x \\
 & | & n \\
 & | & a_1\ op_a\ a_2
\end{array}
\qquad
\begin{array}{lcl}
op_b & ::= & \text{and} \mid \text{or} \\
op_r & ::= & < \ \mid\ \leqslant \ \mid\ = \\
 & | & > \ \mid\ \geqslant \\
op_a & ::= & + \mid - \mid * \mid /
\end{array}
$$

Our first static analysis: AST walking

- One way to find “bugs” is to walk the AST, looking for particular patterns.
  - Traverse the AST, look for nodes of a particular type
  - Check the neighborhood of the node for the pattern in question.
  - Basically, a glorified “grep” that knows about the syntax but not semantics of a language.

CodeQL

- A language for querying code. Developed by GitHub.
- Supports many common languages.
- Library of common programming patterns and optimizations.

CodeQL queries 1.23

Dashboard / Java queries

Inefficient empty string test

Created by Documentation team, last modified on Mar 28, 2019

Name: Inefficient empty string test

Description: Checking a string for equality with an empty string is inefficient.

ID: java/inefficient-empty-string-test

Kind: problem

Severity: recommendation

Precision: high

Query: InefficientEmptyStringTest.ql › Expand source

When checking whether a string `s` is empty, perhaps the most obvious solution is to write something like `s.equals("")` (or `"".equals(s)`). However, this actually carries a fairly significant overhead, because `String.equals` performs a number of type tests and conversions before starting to compare the content of the strings.

Recommendation

The preferred way of checking whether a string `s` is empty is to check if its length is equal to zero. Thus, the condition is `s.length() == 0`. The `length` method is implemented as a simple field access, and so should be noticeably faster than calling `equals`.

Note that in Java 6 and later, the `String` class has an `isEmpty` method that checks whether a string is empty. If the codebase does not need to support Java 5, it may be better to use that method instead.

Operational Semantics of WHILE

- The meaning of WHILE expressions depend on the values of variables
  - What does x+5 mean? It depends on x.
  - If $x = 8$ at some point, we expect x+5 to mean 13
- The value of integer variables at a given moment is abstracted as a function:

$$E : Var \to Z$$

- We will augment our notation of big-step evaluation to include state:

$$\langle E, a \rangle \Downarrow n$$

- So, if $\{x \mapsto 8\} \in E$, then $\langle E, x + 5 \rangle \Downarrow 13$

Big-Step Semantics for WHILE expressions

$$\frac{}{\langle E, n \rangle \Downarrow n} \; \textit{big-int} \qquad \frac{}{\langle E, x \rangle \Downarrow E(x)} \; \textit{big-var}$$

$$\frac{\langle E, a_1 \rangle \Downarrow n_1 \quad \langle E, a_2 \rangle \Downarrow n_2}{\langle E, a_1 + a_2 \rangle \Downarrow n_1 + n_2} \; \textit{big-add}$$

- Similarly for other arithmetic and boolean expressions

Big-Step Semantics for WHILE statements

$$\frac{\langle E, b\rangle \Downarrow \texttt{false}}{\langle E, \texttt{while}\ b\ \texttt{do}\ S\rangle \Downarrow E}\ \textit{big-whilefalse}$$

$$\frac{\langle E, b\rangle \Downarrow \texttt{true} \quad \langle E, S; \texttt{while}\ b\ \texttt{do}\ S\rangle \Downarrow E'}{\langle E, \texttt{while}\ b\ \texttt{then}\ S\rangle \Downarrow E'}\ \textit{big-whiletrue}$$

Small-Step Semantics for WHILE statements

$$\frac{\langle E, S_1\rangle \to \langle E', S_1'\rangle}{\langle E, S_1; S_2\rangle \to \langle E', S_1'; S_2\rangle}\ \textit{small-seq-congruence}$$

$$\frac{}{\langle E, \texttt{skip}; S_2\rangle \to \langle E, S_2\rangle}\ \textit{small-seq}$$

Small-Step Semantics for WHILE statements

$$\frac{\langle E, b \rangle \rightarrow_b b'}{\langle E, \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2 \rangle \rightarrow \langle E, \texttt{if } b' \texttt{ then } S_1 \texttt{ else } S_2 \rangle}\ \textit{small-if-congruence}$$

$$\frac{}{\langle E, \texttt{if true then } S_1 \texttt{ else } S_2 \rangle \rightarrow \langle E, S_1 \rangle}\ \textit{small-iftrue}$$

Proofs by Structural Induction

Example. Let $L(a)$ be the number of literals and variable occurrences in some expression a and $O(a)$ be the number of operators in a. Prove by induction on the structure of a that $\forall a \in \texttt{Aexp} \,.\, L(a) = O(a) + 1$:

Base cases:

- Case $a = n$. $L(a) = 1$ and $O(a) = 0$
- Case $a = x$. $L(a) = 1$ and $O(a) = 0$

Inductive case 1: Case $a = a_1 + a_2$

- By definition, $L(a) = L(a_1) + L(a_2)$ and $O(a) = O(a_1) + O(a_2) + 1$.
- By the induction hypothesis, $L(a_1) = O(a_1) + 1$ and $L(a_2) = O(a_2) + 1$.
- Thus, $L(a) = O(a_1) + O(a_2) + 2 = O(a) + 1$.

The other arithmetic operators follow the same logic.

Proofs by Structural Induction

- Prove that WHILE is *deterministic*. That is, if the program terminates, it evaluates to a unique value.

$$\forall a \in \texttt{Aexp} \,.\quad \forall E \,.\, \forall n, n' \in \mathbb{N} \,.\quad \langle E, a\rangle \Downarrow n \wedge \langle E, a\rangle \Downarrow n' \Rightarrow n = n'$$

$$\forall P \in \texttt{Bexp} \,.\quad \forall E \,.\, \forall b, b' \in \mathcal{B} \,.\quad \langle E, P\rangle \Downarrow b \wedge \langle E, P\rangle \Downarrow b' \Rightarrow b = b'$$

$$\forall S \,.\quad \forall E, E', E'' \,.\quad \langle E, S\rangle \Downarrow E' \wedge \langle E, S\rangle \Downarrow E'' \Rightarrow E' = E''$$

Rule for while is recursive; doesn't depend only on subexpressions

- Can prove for expressions via induction over syntax, but not for statements.
- But there's still a way.

To prove: $\boxed{\forall S . \qquad \forall E, E', E'' . \qquad \langle E, S\rangle \Downarrow E' \wedge \langle E, S\rangle \Downarrow E'' \Rightarrow E' = E''}$

Structural Induction over Derivations

Base case: the one rule with no premises, `skip`:

let $D :: \langle E, S\rangle \Downarrow E'$, and let $D' :: \langle E, S\rangle \Downarrow E''$

$$D ::= \overline{\langle E, \texttt{skip}\rangle \Downarrow E}$$

By inversion, the last rule used in D' (which, again, produced E'') must also have been the rule for `skip`. By the structure of the `skip` rule, we know $E'' = E$.

Inductive cases: We need to show that the property holds when the last rule used in D was each of the possible non-skip WHILE commands. I will show you one representative case; the rest are left as an exercise. If the last rule used was the `while-true` statement:

$$D ::= \frac{D_1 :: \langle E, b\rangle \Downarrow \texttt{true} \quad D_2 :: \langle E, S\rangle \Downarrow E_1 \quad D_3 :: \langle E_1, \texttt{while}\ b\ \texttt{do}\ S\rangle \Downarrow E'}{\langle E, \texttt{while}\ b\ \texttt{do}\ S\rangle \Downarrow E'}$$

Pick arbitrary E'' such that $D' :: \langle E, \texttt{while}\ b\ \texttt{do}\ S\rangle \Downarrow E''$

By inversion, D' must use either the `while-true` or the `while-false` rule. However, having proved that boolean expressions are deterministic (via induction on syntax), and given that D contains the judgment $\langle E, b\rangle \Downarrow \texttt{true}$, we know that D' cannot be the `while-false` rule, as otherwise it would have to contain a contradicting judgment $\langle E, b\rangle \Downarrow \texttt{false}$.

So, we know that D' is also using `while-true` rule. In its derivation, D' must also have subderivations $D'_2 :: \langle E, S\rangle \Downarrow E'_1$ and $D'_3 :: \langle E'_1, \texttt{while}\ b\ \texttt{do}\ S\rangle \Downarrow E''$. By the induction hypothesis on D_2 with D'_2, we know $E_1 = E'_1$. Using this result and the induction hypothesis on D_3 with D'_3, we have $E'' = E'$.

Data-Flow Analysis

Computes universal properties about program state at specific program points. (e.g. will *x* be zero at line 7?)

- About program state
  - About data store (e.g. variables, heap memory)
  - Not about control (e.g. termination, performance)
- At program points
  - Statically identifiable (e.g. line 7, or when foo() calls bar())
  - Not dynamically computed (E.g. when x is 12 or when foo() is invoked 12 times)
- Universal
  - Reasons about all possible executions (always/never/maybe)
  - Not about specific program paths (see: symbolic execution, testing)

WHILE3ADDR: An Intermediate Representation

Simpler, more uniform than WHILE syntax

Categories:

$I \in$ **Instruction** — instructions

$x, y \in$ **Var** — variables

$n \in$ **Num** — number literals

Syntax:

I ::= x := n | x := y | x := y op z
| goto n | if x op_r 0 goto n

op_a ::= + | - | * | / | …

op_r ::= < | ≤ | = | > | ≥ | ...

P ∈ **Num** → I

$$\frac{P(n) = x := m}{P \vdash \langle E, n \rangle \leadsto \langle E[x \mapsto m], n+1 \rangle} \textit{ step-const}$$

$$\frac{P[n] = x := y}{P \vdash \langle E, n \rangle \leadsto \langle E[x \mapsto E(y)], n+1 \rangle} \textit{ step-copy}$$

$$\frac{P(n) = x := y \; op \; z \quad E(y) \; \mathbf{op} \; E(z) = m}{P \vdash \langle E, n \rangle \leadsto \langle E[x \mapsto m], n+1 \rangle} \textit{ step-arith}$$

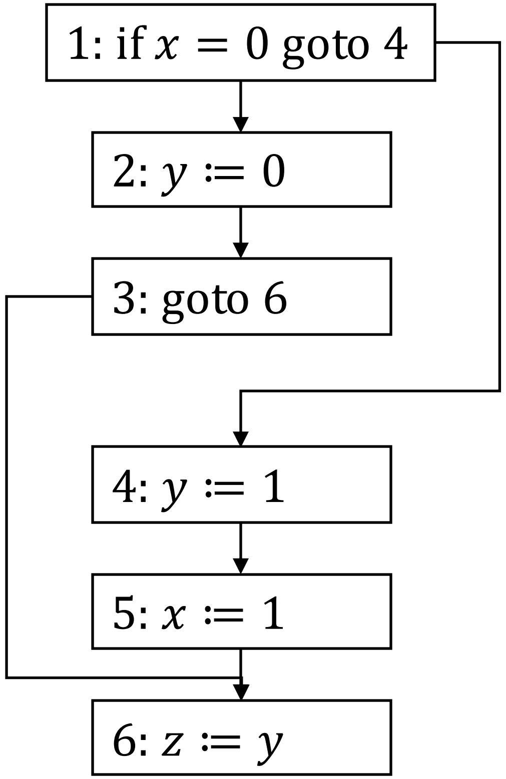
$$\frac{P(n) = \text{goto } m}{P \vdash \langle E, n \rangle \leadsto \langle E, m \rangle} \textit{ step-goto}$$

$$\frac{P(n) = \text{if } x \; op_r \; 0 \text{ goto } m \quad E(x) \; \mathbf{op_r} \; 0 = true}{P \vdash \langle E, n \rangle \leadsto \langle E, m \rangle} \textit{ step-iftrue}$$

$$\frac{P(n) = \text{if } x \; op_r \; 0 \text{ goto } m \quad E(x) \; \mathbf{op_r} \; 0 = false}{P \vdash \langle E, n \rangle \leadsto \langle E, n+1 \rangle} \textit{ step-iffalse}$$

Control-flow Graphs

$$
\begin{aligned}
1 &: \quad \text{if } x = 0 \text{ goto } 4\\
2 &: \quad y := 0\\
3 &: \quad \text{goto } 6\\
4 &: \quad y := 1\\
5 &: \quad x := 1\\
6 &: \quad z := y
\end{aligned}
$$



Nodes = Statements

Edges = (s1, s2) is an edge iff s1 and s2 can be executed consecutively aka "control flow"

Common properties of CFGs:

- Weakly connected
- Only one entry node
- Only one exit (terminal) node

Classic Data-Flow Analyses

- Zero Analysis
- Integer Sign Analysis
- Constant Propagation
- Reaching Definitions
- Live Variables Analysis

- Available Expressions
- Very Busy Expressions
- …

Partial Order & Join on set L

$l_1 \sqsubseteq l_2 \quad : \quad l_1$ is at least as precise as l_2

reflexive: $\forall l : l \sqsubseteq l$
transitive: $\forall l_1, l_2, l_3 : l_1 \sqsubseteq l_2 \wedge l_2 \sqsubseteq l_3 \Rightarrow l_1 \sqsubseteq l_3$
anti-symmetric: $\forall l_1, l_2 : l_1 \sqsubseteq l_2 \wedge l_2 \sqsubseteq l_1 \Rightarrow l_1 = l_2$

$l_1 \sqcup l_2$: **join** or *least-upper-bound*… "most precise generalization"

L is a *join-semilattice* iff: $l_1 \sqcup l_2$ always exists and is unique $\forall l_1, l_2 \in L$

$\top$ ("top") is the maximal element

Fixed point of Flow Functions

$$(\sigma_0, \sigma_1, \sigma_2, \dots, \sigma_n) \xrightarrow{f_z} (\sigma'_0, \sigma'_1, \sigma'_2, \dots, \sigma'_n)$$

$$\sigma'_0 = \sigma_0$$

$$\sigma'_1 = f_z[\![x := 10]\!](\sigma_0)$$

$$\sigma'_2 = f_z[\![y := 0]\!](\sigma_1)$$

$$\sigma'_3 = \sigma_2 \sqcup \sigma_7$$

$$\sigma'_4 = f_z[\![\text{if } x = 10 \text{ goto } 7]\!]_F(\sigma_3)$$

$$\vdots$$

$$\sigma'_8 = f_z[\![\text{if } x = 10 \text{ goto } 7]\!]_T(\sigma_3)$$

$$\sigma'_9 = f_z[\![x := y]\!](\sigma_8)$$

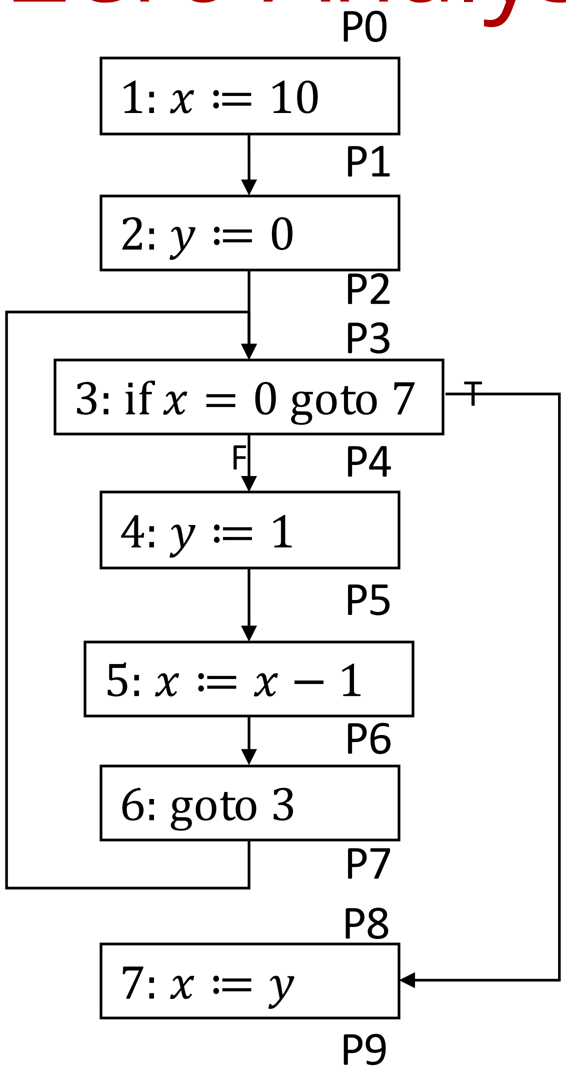
Fixed point!

$$\boxed{(\sigma_0, \sigma_1, \sigma_2, \dots, \sigma_n) = f_z(\sigma_0, \sigma_1, \sigma_2, \dots, \sigma_n)}$$

Correctness theorem:
If data-flow analysis is well designed*, then any fixed point of the analysis is sound.

Example of Zero Analysis: Looping Code

$$
\begin{array}{ll}
1: & x := 10 \\
2: & y := 0 \\
3: & \text{if } x = 0 \text{ goto } 7 \\
4: & y := 1 \\
5: & x := x - 1 \\
6: & \text{goto } 3 \\
7: & x := y
\end{array}
$$



	x	y	
P0	$\top$	$\top$	
P1	N	$\top$	
P2	N	Z	
P3	$\top$	$\top$	*join*
P4	N_F	$\top$	*updated*
P5	N	N	*already at fixed point*
P6	$\top$	N	*already at fixed point*
P7	$\top$	N	*already at fixed point*
P8	Z_T	$\top$	*updated*
P9	$\top$	$\top$	*updated*

Kildall's Algorithm

```
worklist = ∅
for  Node n in cfg
    input[n] = output[n] = ⊥
    add n to worklist
input[0] = initialDataflowInformation

while worklist is not empty
    take a Node n off the worklist
    output[n] = flow(n, input[n])
    for Node j in succs(n)
          newInput = input[j] ⊔ output[n]
          if newInput ≠ input[j]
                input[j] = newInput
                add j to worklist
```

Worklist Algorithm Terminates at Fixed Point

At the fixed point, we therefore have the following equations satisfied:

$$\sigma_0 \sqsubseteq \sigma_1$$

$$\forall i \in P : \left(\bigsqcup_{j \in \mathtt{preds}(i)} f[\![P[j]]\!](\sigma_j) \right) \sqsubseteq \sigma_i$$

The worklist algorithm shown above computes a fixed point when it terminates. We can prove this by showing that the following loop invariant is maintained:

$$\forall i \,.\, (\exists j \in \mathit{preds}(i) \text{ such that } f[\![P[j]]\!](\sigma_j) \not\sqsubseteq \sigma_i) \Rightarrow i \in \mathtt{worklist}$$

Program Traces and DataFlow Soundness

A trace T of a program P is a potentially infinite sequence $\{c_0, c_1, ...\}$ of program configurations, where $c_0 = E_0, 1$ is called the initial configuration, and for every $i \geqslant 0$ we have $P \vdash c_i \leadsto c_{i+1}$

The result $\langle \sigma_n \mid n \in P \rangle$ of a program analysis running or program P is sound iff, for all traces T of P, for all i such that $0 \leqslant i < \mathit{length}(T)$, $\alpha(c_i) \sqsubseteq \sigma_{n_i}$

Fixed Point Theorem

Theorem 2 (A fixed point of a locally sound analysis is globally sound). *If a dataflow analysis's flow function f is monotonic and locally sound, and for all traces T we have $\alpha(c_0) \sqsubseteq \sigma_0$ where σ_0 is the initial analysis information, then any fixed point $\{\sigma_n \mid n \in P\}$ of the analysis is sound.*

Proof. To show that the analysis is sound, we must prove that for all program traces, every program configuration in that trace is correctly approximated by the analysis results. We consider an arbitrary program trace T and do the proof by induction on the program configurations $\{c_i\}$ in the trace.

Least Fixed Point (LFP)

The least fixed point solution of a composite flow function $\mathcal{F}$ is the fixed-point result Σ^* such that $\mathcal{F}(\Sigma^*) = \Sigma^*$ and $\forall \Sigma : (\mathcal{F}(\Sigma) = \Sigma) \Rightarrow (\Sigma^* \sqsubseteq \Sigma)$.

Merge Over Paths (MOP)

We first enumerate all paths π of the form $\pi = n_1, n_2, \ldots$ in the control-flow graph, where n_i are the instructions (nodes) in the path. For each such path π, we successively apply flow functions to form the sequence of tuples $\Pi = \langle\sigma_1, n_1\rangle, \langle\sigma_2, n_2\rangle, \ldots$ such that $\sigma_{\Pi_j} = f[\![P[n_{\Pi_j}]]\!](\sigma_{\Pi_{j-1}})$, where Π_j is the j-th tuple in the sequence and σ_0 is the initial data flow information. We then join over all σ values computed for an instruction i to get the MOP:

$$\text{MOP}(i) = \bigsqcup\{\sigma \mid \langle\sigma, i\rangle \in \text{Some } \Pi \text{ for } P\}$$

The MOP solution is the most precise result if we consider all possible program paths through the CFG, even though it may be less precise than the optimal solution due to the consideration of infeasible paths. The MOP is computable when flow functions are *distributive* over join.

Distributivity A function f is *distributive* iff $f(\sigma_1) \sqcup f(\sigma_2) = f(\sigma_1 \sqcup \sigma_2)$

Reaching Definitions

$$
\begin{aligned}
f_{RD}[\![I]\!](\sigma) &= \sigma - KILL_{RD}[\![I]\!] \cup GEN_{RD}[\![I]\!] \\
\mathrm{KILL}_{RD}[\![n\colon\ x := \ldots]\!] &= \{x_m \mid x_m \in \mathsf{DEFS}(x)\} \\
\mathrm{KILL}_{RD}[\![I]\!] &= \emptyset \qquad \text{if } I \text{ is not an assignment} \\
\mathrm{GEN}_{RD}[\![n\colon\ x := \ldots]\!] &= \{x_n\} \\
\mathrm{GEN}_{RD}[\![I]\!] &= \emptyset \qquad \text{if } I \text{ is not an assignment}
\end{aligned}
$$

Live Variables

Flow functions map backward! (out --> in)

$$\text{KILL}_{LV}[\![I]\!] \quad = \{x \mid I \text{ defines } x\}$$

$$\text{GEN}_{LV}[\![I]\!] \quad = \{x \mid I \text{ uses } x\}$$

Constant Propagation

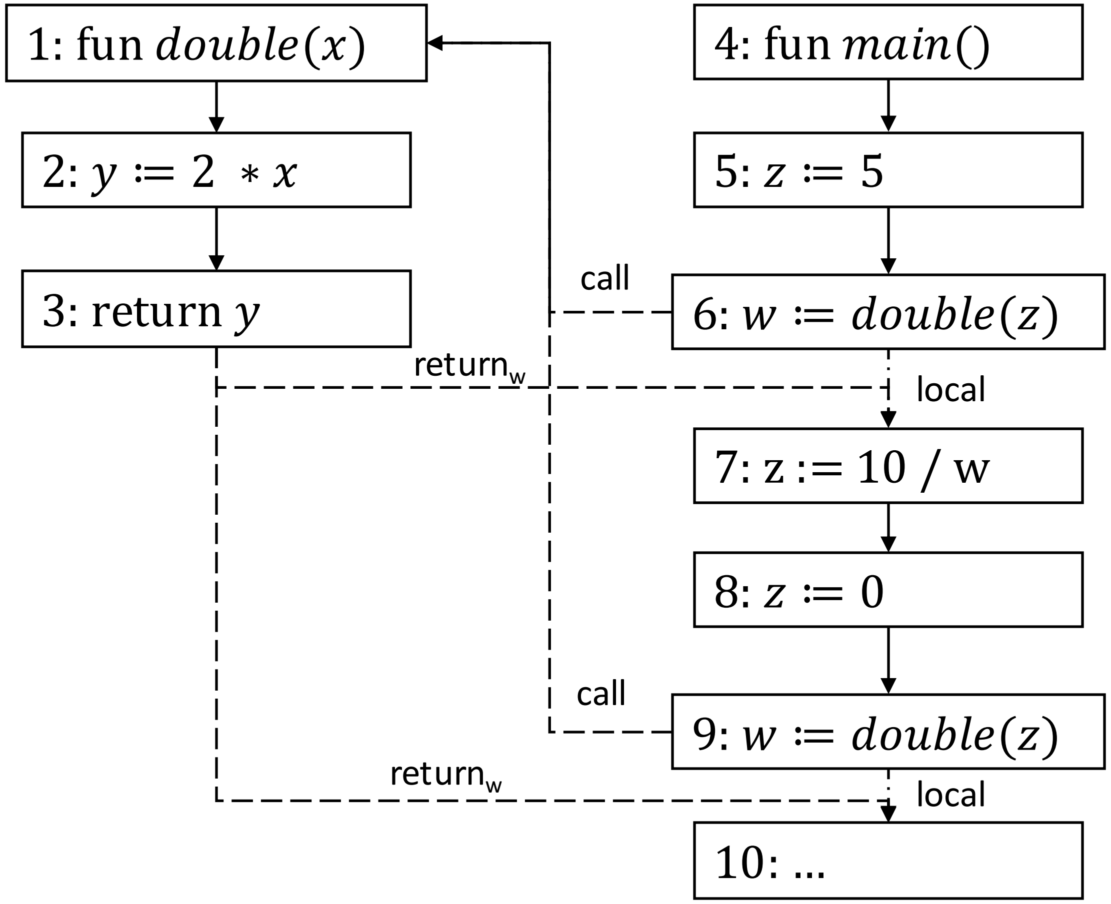
$$
\begin{array}{rcl}
\sigma & \in & \mathit{Var} \rightarrow L_{CP} \\
\sigma_1 \sqsubseteq_{lift} \sigma_2 & \mathit{iff} & \forall x \in \mathit{Var} : \sigma_1(x) \sqsubseteq \sigma_2(x) \\
\sigma_1 \sqcup_{lift} \sigma_2 & = & \{x \mapsto \sigma_1(x) \sqcup \sigma_2(x) \mid x \in \mathit{Var}\} \\
\top_{lift} & = & \{x \mapsto \top \mid x \in \mathit{Var}\} \\
\bot_{lift} & = & \{x \mapsto \bot \mid x \in \mathit{Var}\} \\
\alpha_{CP}(n) & = & n \\
\alpha_{lift}(E) & = & \{x \mapsto \alpha_{CP}(E(x)) \mid x \in \mathit{Var}\} \\
\sigma_0 & = & \top_{lift}
\end{array}
$$

Extend WHILE3ADDR with functions

$$F \quad ::= \quad \text{fun } f(x) \; \{ \; \overline{n : I} \; \}$$
$$I \quad ::= \quad \ldots \mid \text{return } x \mid y := f(x)$$

$$
\begin{array}{ll}
1: & \text{fun } \mathit{double}(x) : \mathit{int} \\
2: & \quad y := 2 * x \\
3: & \quad \text{return } y \\
4: & \text{fun } \mathit{main}() : \mathit{void} \\
5: & \quad z := 0 \\
6: & \quad w := \mathit{double}(z)
\end{array}
$$

Interprocedural CFG



$$
\begin{array}{ll}
1: & \text{fun } double(x) : int \\
2: & \quad y := 2 * x \\
3: & \quad \text{return } y \\
4: & \text{fun } main() \\
5: & \quad z := 5 \\
6: & \quad w := double(z) \\
7: & \quad z := 10/w \\
8: & \quad z := 0 \\
9: & \quad w := double(z)
\end{array}
$$

$$f_Z[\![x := g(y)]\!]_{local}(\sigma) = \sigma \setminus (\{x\} \cup \ Globals)$$

$$f_Z[\![x := g(y)]\!]_{call}(\sigma) = \{v \mapsto \sigma(v) |\ v \in Globals\} \cup \{formal(g) \mapsto \sigma(y)\}$$

$$f_Z[\![\text{return } y]\!]_{return_x}(\sigma) = \{v \mapsto \sigma(v) |\ v \in Globals\} \cup \{x \mapsto \sigma(y)\}$$

Context-Sensitive Analysis Example

```
fun double(x) : int
    y := 2 * x
    return y

fun main()
    z := 5
    w := double(z)
    z := 10/w
    z := 0
    w := double(z)
```

Context	σ_{in}	σ_{out}
<main, T>	T	{w->Z, Z->Z}
<double, N>	{x->N}	{x->N, y->N}
<double, Z>	{x->Z}	{x->Z, y->Z}

Key Idea: Worklist of Contexts

val *worklist* : *Set*[*Context*]
val *analyzing* : *Set*[*Context*]
val *results* : *Map*[*Context*, *Summary*]
val *callers* : *Map*[*Context*, *Set*[*Context*]]

```
function ANALYZEPROGRAM
    initCtx ← GETCTX(main, nil, 0, ⊤)
    worklist ← {initCtx}
    results[initCtx] ← Summary(⊤, ⊥)
    while NOTEMPTY(worklist) do
        ctx ← REMOVE(worklist)
        ANALYZE(ctx, results[ctx].input)
    end while
end function
```

```
function ANALYZE(ctx, σ_in)
    σ_out ← results[ctx].output
    ADD(analyzing, ctx)
    σ'_out ← INTRAPROCEDURAL(ctx, σ_in)
    REMOVE(analyzing, ctx)
    if σ'_out ⋢ σ_out then
        results[ctx] ← Summary(σ_in, σ_out ⊔ σ'_out)
        for c ∈ callers[ctx] do
            ADD(worklist, c)
        end for
    end if
    return σ'_out
end function
```

function $\textsc{ResultsFor}(ctx, \sigma_{in})$
  if $ctx \in \mathrm{dom}(results)$ **then**
    if $\sigma_{in} \sqsubseteq results[ctx].input$ **then**
      return $results[ctx].output$ ▻ existing results are good
    else
      $results[ctx].input \leftarrow results[ctx].input \sqcup \sigma_{in}$ ▻ keep track of more general input
    end if
  else
    $results[ctx] = Summary(\sigma_{in}, \bot)$ ▻ initially optimisti
  end if
  if $ctx \in analyzing$ **then**
    return $results[ctx].output$ ▻ $\bot$ if it hasn't been analyzed yet; otherwis
  else
    return $\textsc{Analyze}(ctx, results[ctx].input)$
  end if
end function

function $\textsc{Analyze}(ctx, \sigma_{in})$
  $\sigma_{out} \leftarrow results[ctx].output$
  $\textsc{Add}(analyzing, ctx)$
  $\sigma'_{out} \leftarrow \textsc{Intraprocedural}(ctx, \sigma_{in})$
  $\textsc{Remove}(analyzing, ctx)$
  if $\sigma'_{out} \not\sqsubseteq \sigma_{out}$ **then**
    $results[ctx] \leftarrow Summary(\sigma_{in}, \sigma_{out} \sqcup \sigma'_{out})$
    for $c \in callers[ctx]$ **do**
      $\textsc{Add}(worklist, c)$
    end for
  end if
  return σ'_{out}
end function

function $\textsc{Flow}([\![n\colon x := f(y)]\!], ctx, \sigma_n)$
  $\sigma_{in} \leftarrow [formal(f) \mapsto \sigma_n(y)]$ ▻
  $calleeCtx \leftarrow \textsc{GetCtx}(f, ctx, n, \sigma_{in})$
  $\sigma_{out} \leftarrow \textsc{ResultsFor}(calleeCtx, \sigma_{in})$
  $\textsc{Add}(callers[calleeCtx], ctx)$
  return $\sigma_n[x \mapsto \sigma_{out}[result]]$

Extending WHILE3ADDR with Pointers

$$
\begin{array}{rcll}
I & ::= & \ldots & \\
& | & p := \&x & \text{taking the address of a variable} \\
& | & p := q & \text{copying a pointer from one variable to another} \\
& | & *p := q & \text{assigning through a pointer} \\
& | & p := *q & \text{dereferencing a pointer}
\end{array}
$$

Andersen’s Analysis

$$\frac{}{[\![p := \&x]\!] \hookrightarrow l_x \in p}\ \textit{address-of}$$

$$\frac{l_x \in q}{\in p}\ \textit{copy}$$

$$\frac{}{[\![p := q]\!] \hookrightarrow p \supseteq q}\ \textit{copy}$$

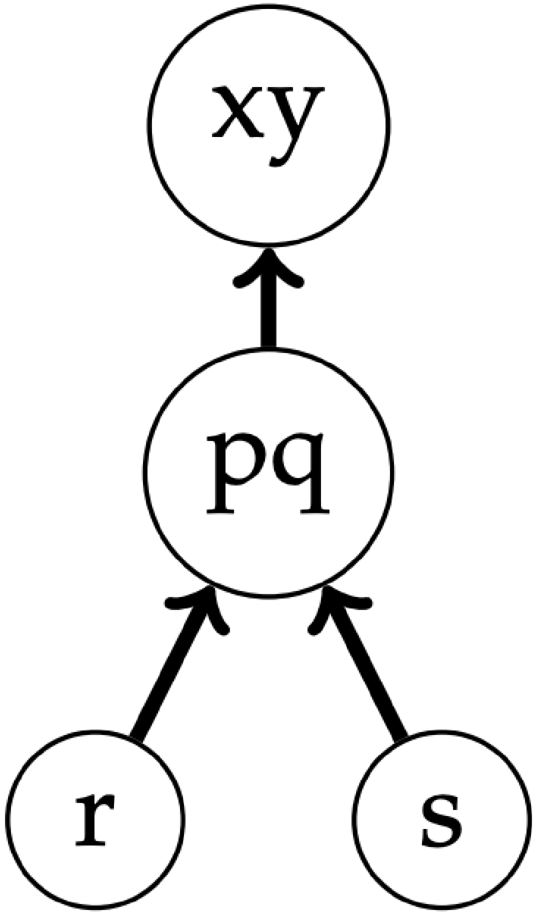
$$\frac{p \quad l_x \in q}{r}\ \textit{assign}$$

$$\frac{}{[\![*p := q]\!] \hookrightarrow *p \supseteq q}\ \textit{assign}$$

$$\frac{}{[\![p := *q]\!] \hookrightarrow p \supseteq *q}\ \textit{dereference} \quad \frac{l_x \in r}{}\ \textit{dereference}$$

Steensgaard's Analysis - Example

$$
\begin{aligned}
1:&\quad p := \&x\\
2:&\quad r := \&p\\
3:&\quad q := \&y\\
4:&\quad s := \&q\\
5:&\quad r := s
\end{aligned}
$$



Steensgaard's Analysis

$$\frac{}{[\![p := q]\!] \hookrightarrow join(*p, *q)}\ copy$$

$$\frac{}{[\![p := \&x]\!] \hookrightarrow join(*p, x)}\ address\text{-}of$$

$$\frac{}{[\![p := *q]\!] \hookrightarrow join(*p, **q)}\ dereference$$

$$\frac{}{[\![*p := q]\!] \hookrightarrow join(**p, *q)}\ assign$$

```
join(ℓ1,ℓ2)
    if (find(ℓ1) == find(ℓ2))
        return
    n1 ← *ℓ1
    n2 ← *ℓ2
    union(ℓ1,ℓ2)
    join(n1,n2)
```

Hoare Triple

$$\{ P \} \; S \; \{ Q \}$$

- P is the precondition
- Q is the postcondition
- S is any statement (in WHILE, at least for our class)

- Semantics: if P holds in some state E and if $<S; E> \Downarrow E'$, then Q holds in E'
  - This is *partial correctness*: termination of S is not guaranteed
  - *Total correctness* additionally implies termination, and is written $[P] \; S \; [Q]$

Semantics of Hoare Triples

- A partial correctness assertion $\models \{P\}\, S\, \{Q\}$ is defined formally to mean:

$$\forall E. \forall E'. (E \models P \wedge \langle E, S \rangle \Downarrow E') \Rightarrow E' \models Q$$

- How would we define total correctness $[P]\, S\, [Q]$?

- This is a good formal definition—but it doesn't help us prove many assertions because we have to reason about all environments. How can we do better?

Derivation Rules for Hoare Logic

- Judgment form $\vdash \{P\}\ S\ \{Q\}$ means "we can prove the Hoare triple $\{P\}\ S\ \{Q\}$"

$$\frac{}{\vdash \{P\}\ \mathrm{skip}\ \{P\}}\ \mathit{skip} \qquad \frac{}{\vdash \{[a/x]P\}\ x{:=}a\ \{P\}}\ \mathit{assign}$$

$$\frac{\vdash \{P\}\ S_1\ \{P'\} \qquad \vdash \{P'\}\ S_2\ \{Q\}}{\vdash \{P\}\ S_1;\ S_2\ \{Q\}}\ \mathit{seq} \qquad \frac{\vdash \{P \wedge b\} S_1 \{Q\} \qquad \vdash \{P \wedge \neg b\}\ S_2\ \{Q\}}{\vdash \{P\}\ \mathrm{if}\ b\ \mathrm{then}\ S_1\ \mathrm{else}\ S_2\ \{Q\}}\ \mathit{if}$$

$$\frac{\vdash P' \Rightarrow P \qquad \vdash \{P\}\ S\ \{Q\} \qquad \vdash Q \Rightarrow Q'}{\vdash \{P'\}\ S\ \{Q'\}}\ \mathit{consq}$$

Hoare Triples and Weakest Preconditions

- Theorem: {P} S {Q} holds if and only if P $\Rightarrow$ *wp*(S,Q)
  - In other words, a Hoare Triple is still valid if the precondition is stronger than necessary, but not if it is too weak
  - Can use this to prove {P} S {Q} by computing wp(S,Q) and checking implication.

- Question: Could we state a similar theorem for a strongest postcondition function?
  - e.g. {P} S {Q} holds if and only if *sp*(S,P) $\Rightarrow$ Q
  - A: Yes, but it's harder to compute (see text for why)

Proving loops correct

- First consider *partial correctness*
  - The loop may not terminate, but if it does, the postcondition will hold
- {P} while B do S {Q}
  - Find an invariant Inv such that:
    - P $\Rightarrow$ Inv
      - The invariant is initially true
    - { Inv && B } S {Inv}
      - Each execution of the loop preserves the invariant
    - (Inv && $\neg$B) $\Rightarrow$ Q
      - The invariant and the loop exit condition imply the postcondition

What if we just went forwards?

$$\{P\}$$
$$x := e_1$$
$$x := e_2$$
$$\{Q\}$$

Generate "fresh" math variables for every mutable program variable

Proof Obligation:

$$\forall x_n : ([x_0/x]P \wedge x_1 = [x_0/x]e_1 \wedge x_2 = ([x_1/x]e_2)) \Rightarrow [x_2/x]Q$$

Dealing with conditional paths

$$\{true\}$$
$$\textbf{if}\ (x < 0)\ :$$
$$\quad y := -x$$
$$\textbf{else}\ :$$
$$\quad y := x$$
$$\{y \geqslant 0\}$$

Dynamic Symbolic Execution:

$$\forall x_0, y_0 \in \mathbb{Z} : (x_0 < 0 \wedge y_0 = -x_0) \Rightarrow y_0 \geqslant 0$$

$$\forall x_0, y_0 \in \mathbb{Z} : (x_0 \geqslant 0 \wedge y_0 = x_0) \Rightarrow y_0 \geqslant 0$$

Static Symbolic Execution:

$$\forall x_0, y_0 \in \mathbb{Z} : ((x_0 < 0 \Rightarrow y_0 = -x_0) \wedge (x_0 \geqslant 0 \Rightarrow y_0 = x_0)) \Rightarrow y_0 \geqslant 0$$

Symbolic Execution of Statements (DSE)

$$\frac{}{\langle g, \Sigma, \texttt{skip} \rangle \Downarrow \langle g, \Sigma \rangle} \textit{ big-skip}$$

$$\frac{\langle g, \Sigma, s_1 \rangle \Downarrow \langle g', \Sigma' \rangle \quad \langle g', \Sigma', s_2 \rangle \Downarrow \langle g'', \Sigma'' \rangle}{\langle g, \Sigma, s_1; s_2 \rangle \Downarrow \langle g'', \Sigma'' \rangle} \textit{ big-seq}$$

$$\frac{\langle \Sigma, a \rangle \Downarrow a_s}{\langle g, \Sigma, x := a \rangle \Downarrow \langle g, \Sigma[x \mapsto a_s] \rangle} \textit{ big-assign}$$

Symbolic Execution with Branching (DSE)

$$\frac{\langle \Sigma, b\rangle \Downarrow g' \quad g \wedge g' \mathrm{SAT} \quad \langle g \wedge g', \Sigma, s_1\rangle \Downarrow \langle g'', \Sigma'\rangle}{\langle g, \Sigma, \mathtt{if}\ b\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2, \rangle \Downarrow \langle g'', \Sigma'\rangle}\ \textit{big-iftrue}$$

$$\frac{\langle \Sigma, b\rangle \Downarrow g' \quad g \wedge \neg g' \mathrm{SAT} \quad \langle g \wedge \neg g', \Sigma, s_2\rangle \Downarrow \langle g'', \Sigma'\rangle}{\langle g, \Sigma, \mathtt{if}\ b\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2, \rangle \Downarrow \langle g'', \Sigma'\rangle}\ \textit{big-iffalse}$$

Symbolic Execution of Loops

Bounded exploration (*k*-limited)

$$\frac{k > 0 \quad \langle \Sigma, b \rangle \Downarrow g' \quad g \wedge g' \mathrm{SAT} \quad \langle g \wedge g', \Sigma, s; \mathtt{while_{k-1}}\ b\ \mathtt{do}\ s \rangle \Downarrow \langle g'', \Sigma' \rangle}{\langle g, \Sigma, \mathtt{while_k}\ b\ \mathtt{do}\ s, \rangle \Downarrow \langle g'', \Sigma' \rangle}\ \textit{big-whiletrue}$$

$$\frac{\langle \Sigma, b \rangle \Downarrow g' \quad g \wedge \neg g' \mathrm{SAT}}{\langle g, \Sigma, \mathtt{while_k}\ b\ \mathtt{do}\ s, \rangle \Downarrow \langle g \wedge \neg g', \Sigma \rangle}\ \textit{big-whilefalse}$$

Satisfiability (SAT) solving

- Let's start by considering Boolean formulas:
  variables connected with $\wedge$ $\vee$ $\neg$
- First step: convert to conjuctive normal form (CNF)
  - A conjunction of disjunctions of (possibly negated) variables

$$(a \vee \neg b) \wedge (\neg a \vee c) \wedge (b \vee c)$$

- If formula is not in CNF, we transform it: use De Morgan's laws, the double negative law, and the distributive laws:

$$\begin{aligned}
\neg(P \vee Q) &\iff \neg P \wedge \neg Q \\
\neg(P \wedge Q) &\iff \neg P \vee \neg Q \\
\neg\neg P &\iff P \\
(P \wedge (Q \vee R)) &\iff ((P \wedge Q) \vee (P \wedge R)) \\
(P \vee (Q \wedge R)) &\iff ((P \vee Q) \wedge (P \vee R))
\end{aligned}$$

Concolic Execution

```
int double (int v) {
    return 2*v;
}

void bar(int x, int y) {
    z = double (y);
    if (z == x) {
        if (x > y+10) {
            ERROR;
        }
    }
}
```

1. Input: x=0, y=1
   - Path: (2*y != x)
   - Next: (2*y == x) :: **SAT**
2. Input: x=2, y=1
   - Path: (2*y == x) && (x <= y+10)
   - Next: (2*y == x) && (x > y+10) :: **SAT**
3. Input: x=22, y=11
   - Path: (2*y == x) && (x > y+10)
   - **Bug found!!**

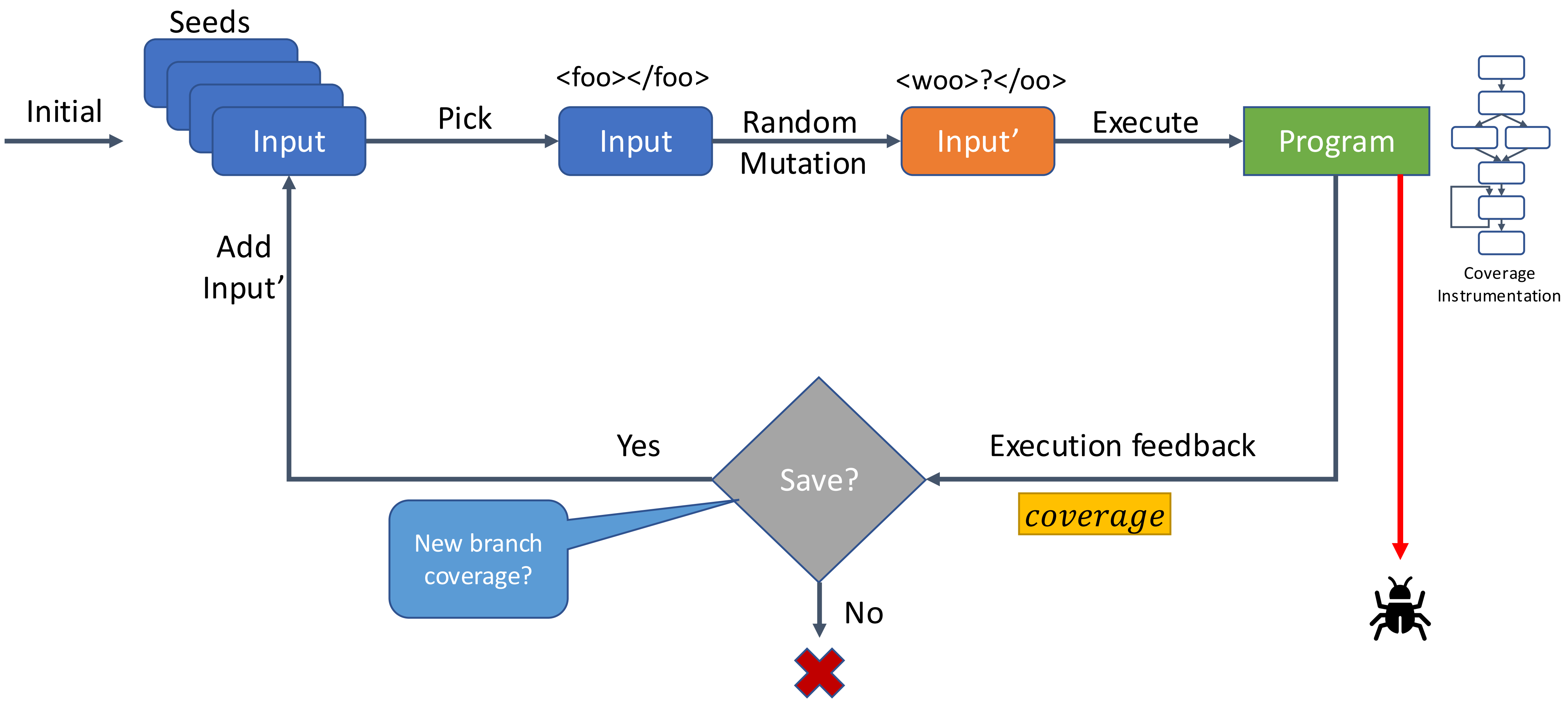
Dynamic analysis

- Observe program behavior during *execution* on *one or more inputs*.
- Examples:
  - Code coverage (→ Greybox fuzzing, fault localization)
  - Performance Profiling
    - Code profiling, memory profiling, algorithmic profiling
  - Invariant Generation
  - Concolic Execution
  - Data structure analysis
  - Concurrency analysis: Race detection
  - Concurrency analysis: Deadlock detection
  - Taint Analysis (→ Security & Privacy)
  - … (many many more)

Collecting execution info

- What to collect? *Only what's necessary*
- Key idea (again): *Abstraction*
- Examples:
  - Code coverage → Log branches
  - Profiling → Log loops, function calls, allocations, frees, etc.
  - Invariant generation → Log predicates over vars in scope
  - Concolic execution → Track symbolic values; log branch constraints
  - Race detection → Track locks, vector clocks; log accesses

Coverage-Guided Fuzzing with AFL



Mutation testing measures test effectiveness on artificial bugs

```
bool is_even(int x) {
   int a = x / 2;
   int b = a * 2;
   return x == b;
}
```

```
void test_even() {
   assert(is_even(4) == true);
}
```

Kills mutant 1 & 3 but not 2

```
bool is_even(int x) {
   int a = x / 2;
   int b = a * 2;
   return x != b;
}
```

Mutant 1

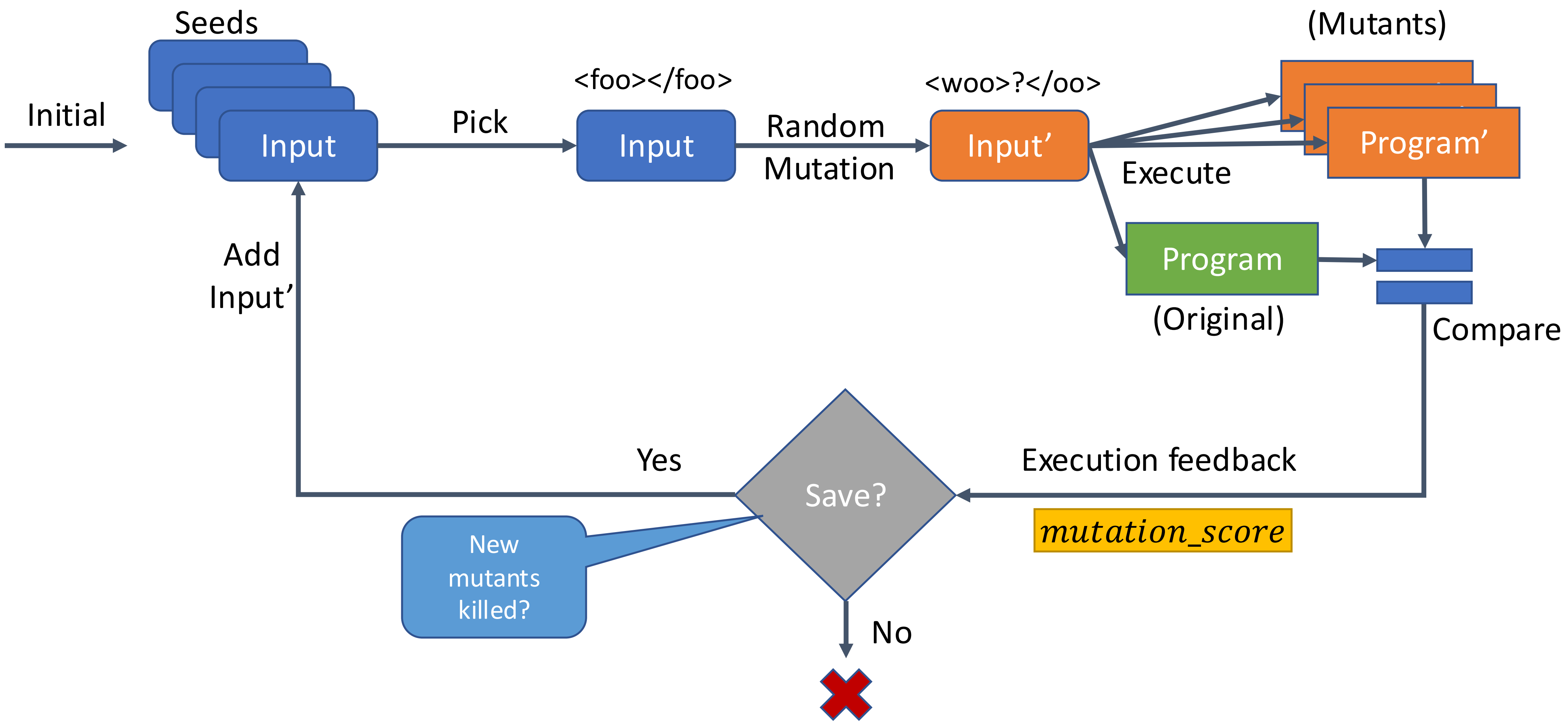
```
bool is_even(int x) {
   int a = x / 2;
   int b = a * 2;
   return b == b;
}
```

Mutant 2

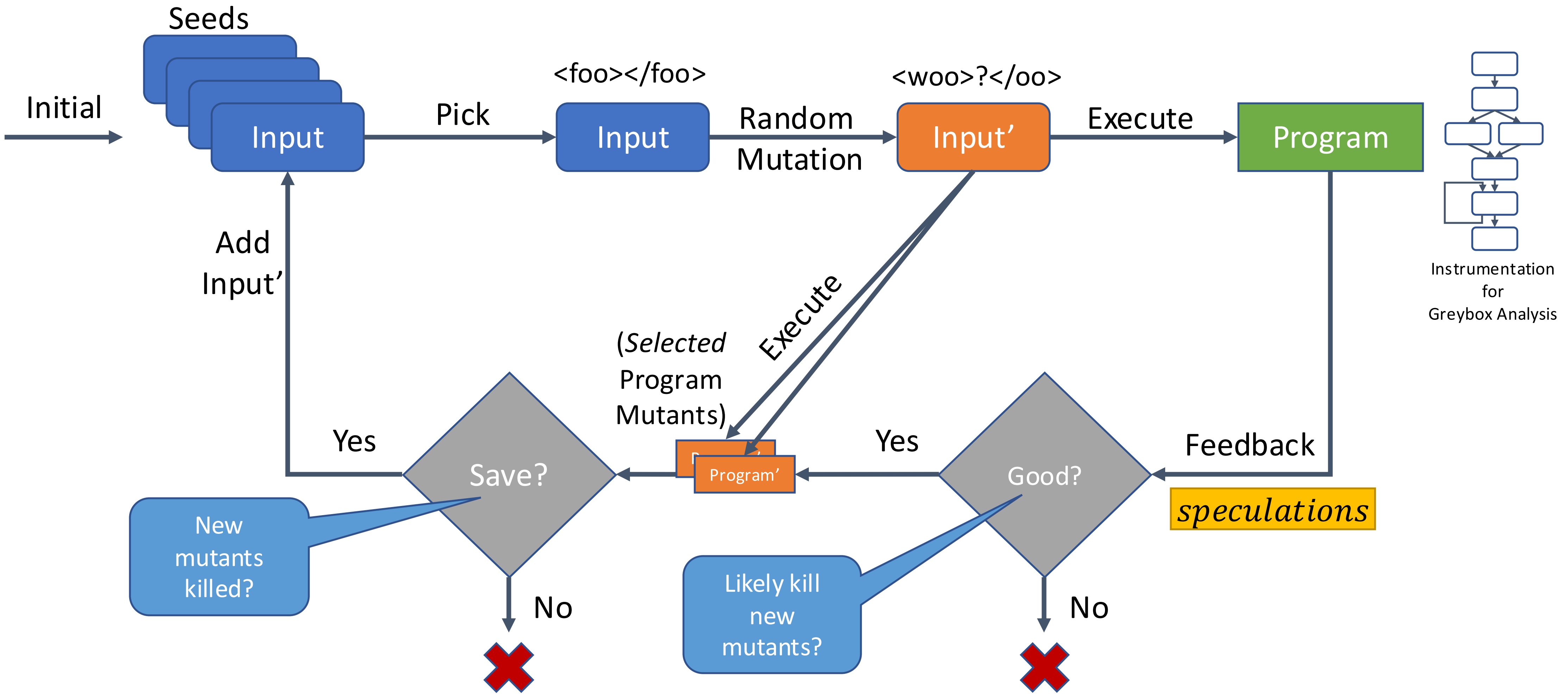
```
bool is_even(int x) {
   int a = x / 2;
   int b = a * 2;
   return x == a;
}
```

Mutant 3

Mutation-analysis guided fuzzing



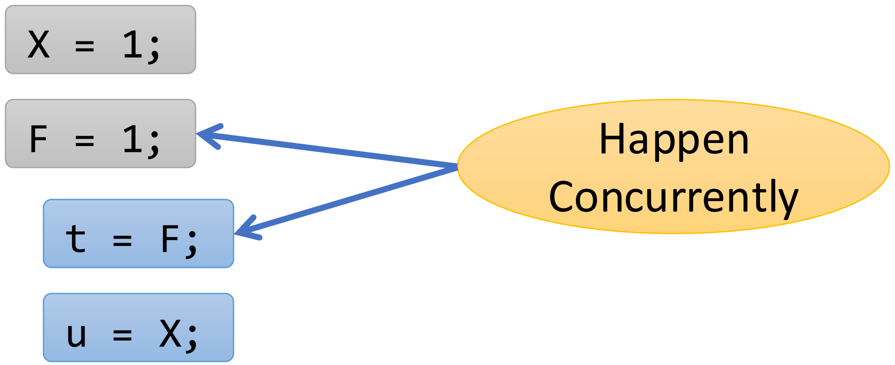
Speculative mutation analysis



Data Races

- A data race is a pair of conflicting accesses that happen concurrently

```
X = 1;        t = F;
F = 1;        u = X;
```



Data Races Can Break Sequentially Consistent Semantics

Init: X = Y = 0;

```
L1: t = X*5;
L2: u = Y;
L3: v = X*5;
```

```
M1: X = 1;
M2: Y = 1;
```

u == 1 ➔ v == 5

Data Race

Init: X = Y = 0;

```
L1: t = X*5;
L2: u = Y;
L3: v = t;
```

```
M1: X = 1;
M2: Y = 1;
```

possibly u == 1 && v == 0

Lockset Algorithm Overview

- Checks a sufficient condition for data-race-freedom
- Consistent locking discipline
  - Every data structure is protected by a single lock
  - All accesses to the data structure made while holding the lock
- Example:

```
// Remove a received packet
AcquireLock( RecvQueueLk );
pkt = RecvQueue.RemoveTop();
ReleaseLock( RecvQueueLk );

… // process pkt

// Insert into processed
AcquireLock( ProcQueueLk );
ProcQueue.Insert(pkt);
ReleaseLock( ProcQueueLk );
```

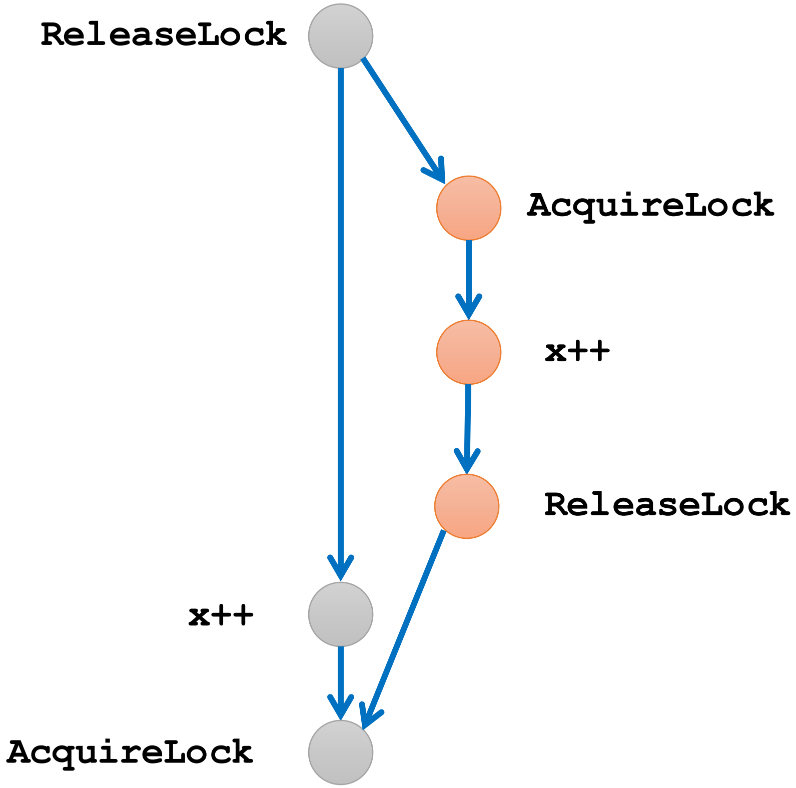
RecvQueue is consistently protected by RecvQueueLk

ProcQueue is consistently protected by ProcQueueLk

Happens-Before Relation And Data Races

- If all conflicting accesses are ordered by happens-before
- → data-race-free execution
- → All linearizations of partial-order are valid program executions

- If there exists conflicting accesses not ordered
- → a data race



Vector Clocks

VC_A		VC_B		L_m		W_x		R_x	
4	1	2	8	2	1	3	0	0	1

`x = 0`

Write-Write Check: $W_x \sqsubseteq VC_A$?

$[3 \; 0] \sqsubseteq [4 \; 1]$? Yes

Read-Write Check: $R_x \sqsubseteq VC_A$?

$[0 \; 1] \sqsubseteq [4 \; 1]$? Yes

O(n) time

Guest Lectures




```
Thread 1:

🔒_1[b].lock()
r1=AtomicRead(b)
🔒_1[b].unlock();
if(r1>=amount)
  🔒_1[b].lock()
  r1=AtomicRead(b)
  🔒_1[b].unlock();
```

```
Thread 2:

🔒_2[b].lock()
r1=AtomicRead(b)
🔒_2[b].unlock();
if(r1>=amount)
  🔒_2[b].lock()
  r1=AtomicRead(b)
  🔒_2[b].unlock();
```

```
Lock Status:
🔒_1[b]: Held by Fray
🔒_2[b]: Held by Fray
```

Thread 1

r1=balance

How useful is uReview at Uber?

We establish usefulness using a few different metrics:

- **Retrospective Detection Rate:** uReview was able to detect a good fraction of historic bugs that lead to past production incidents
- **Preventable Incident Count:** uReview detected many bugs before they reached production*
- **Developer Satisfaction Rate:** Median developer feedback was very positive*
- **Comment Addressal Rate:** ~65% of the posted comments were resolved

Considerations: AI-based Dynamic Analysis in practice

Now, we can automatically (and easily) do things that required substantial manual effort. **This substantially increases the bug-finding surface**!

Things to consider:

- Why are we able to find more bugs? Is it the program analysis technique? Are the LLMs just really capable?
- How would the next LLM impact the dynamic analysis system you've built?
- How do we properly report these bugs? See Google vs. ffmpeg battle

Static vs Dynamic Analysis

- Over-approximation vs Under-approximation
- When is one better than other? Tradeoffs!
  - Soundness/Completeness
    - Static analysis often “sound” for over-approximate reasoning (e.g. verification)
    - Dynamic Analysis can be ”sound” for under-approximate reasoning (e.g. hot spots or bugs).
    - Neither technique is complete in general.
  - Scalability
    - Static analysis often scales super-linearly with program size
    - Dynamic analysis *tries* to scale linearly with execution length
  - Feasibility
    - Static analysis may be impossible with incomplete information (e.g. dynamically loaded code, dependency injection, multi-language code, hardware interaction)
    - Dynamic analysis is only useful when appropriate program inputs are available

Course Evaluations – **cmu.smartevals.com**

??

17-355 (Undergrad)

17-665 (Masters)

17-819 (PhD)

Next Steps – Course Project

- Project discussions today (Dec 4)
- Recitation tomorrow (Dec 5) reserved for project discussions
- No Office Hours next week
- Project Presentations (Dec 11)
  - In-person 1-4pm at GHC 4215
  - Bring your laptops and adapters, if any
  - 6 min talks + 2min Q&A (= 8 min firm cut-off)
- Project Slides + Report due Dec 11 at midnight on Gradescope