HARVARD
John A. Paulson
School of Engineering
and Applied Sciences

Symbolic Execution

CS252r Fall 2015

Contains content from slides by Jeff Foster

Static analysis

- Static analysis allows us to reason about all possible executions of a program
  - Gives assurance about any execution, prior to deployment
  - Lots of interesting static analysis ideas and tools
- But difficult for developers to use
  - Commercial tools spend a lot of effort dealing with developer confusion, false positives, etc.
    - See "A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World" in CACM 53(2), 2010
      - http://bit.ly/aedM3k

One issue is abstraction

- Abstraction lets us scale and model all possible runs
  - But must be conservative
  - Try to balance precision and scalability
    - Flow-sensitive, context-sensitive, path-sensitivity, …
- And static analysis abstractions do not cleanly match developer abstractions

Testing

- Fits well with developer intuitions
- In practice, most common form of bug-detection
- But each test explores only one possible execution of the system
  - Hopefully, test cases generalize

Symbolic execution

- King, CACM 1976.
- Key idea: generalize testing by using unknown symbolic variables in evaluation

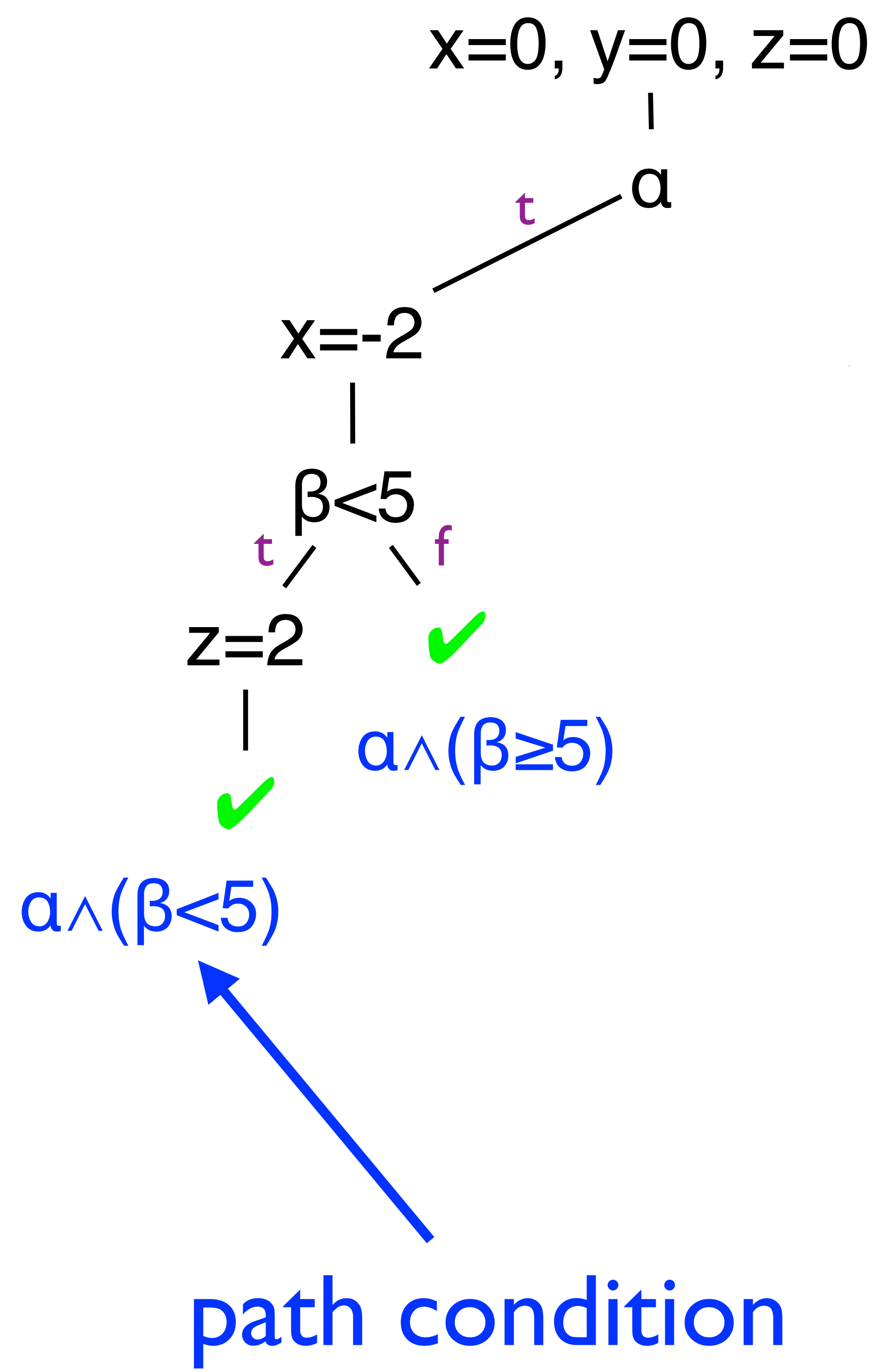
- Symbolic executor executes program, tracking symbolic state.
- If execution path depends on unknown, we fork symbolic executor
    - at least, conceptually

Symbolic execution example

```
int a = α, b = β, c = γ;
                    // symbolic
int x = 0, y = 0, z = 0;
if (a) {
  x = -2;
}
if (b < 5) {
  if (!a && c)  { y = 1; }
  z = 2;
}
assert(x+y+z!=3)
```

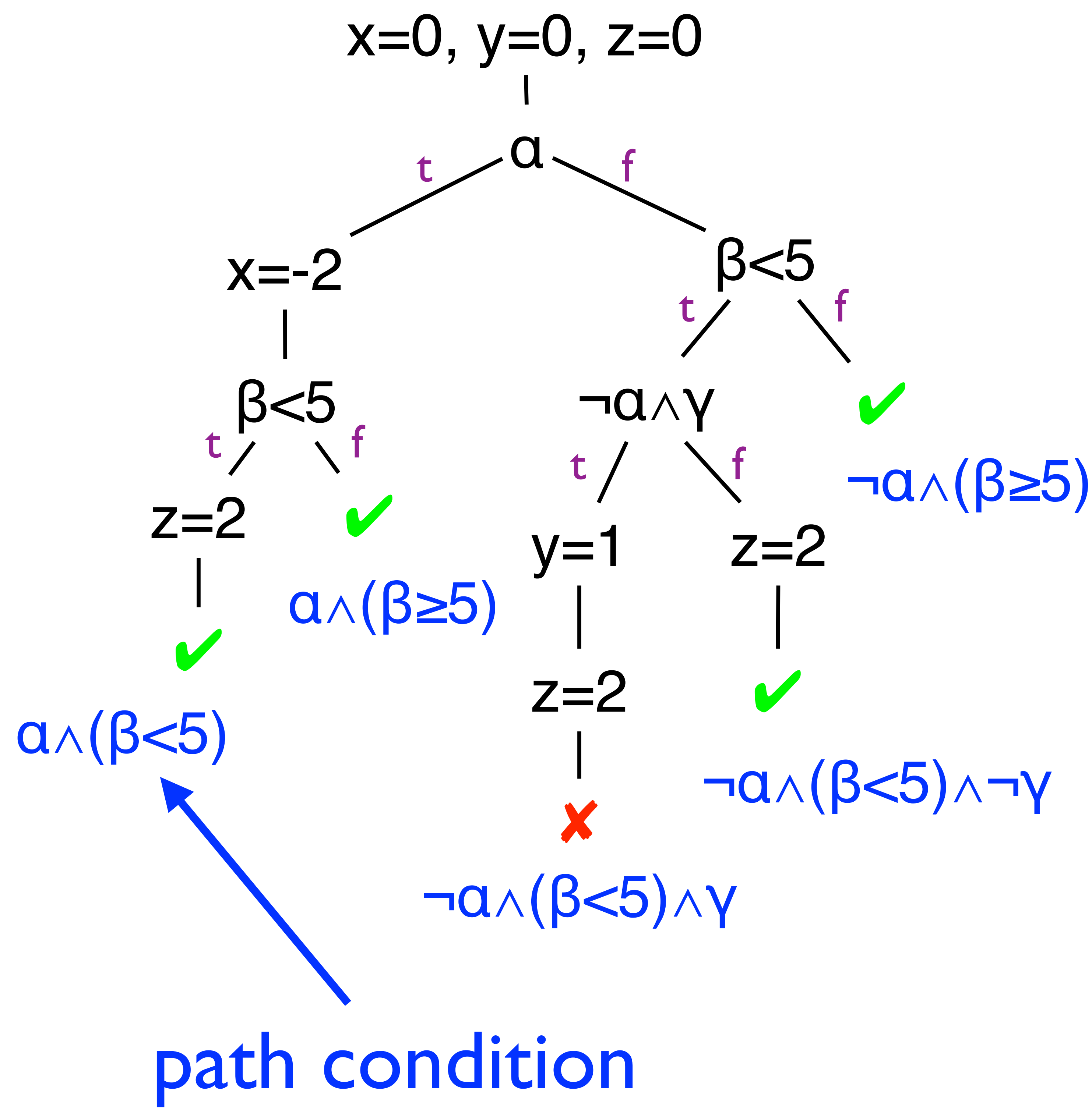
Symbolic execution example

```
int a = α, b = β, c = γ;
                 // symbolic
int x = 0, y = 0, z = 0;
if (a) {
  x = -2;
}
if (b < 5) {
  if (!a && c)  { y = 1; }
  z = 2;
}
assert(x+y+z!=3)
```



Symbolic execution example

```
int a = α, b = β, c = γ;
                   // symbolic
int x = 0, y = 0, z = 0;
if (a) {
  x = -2;
}
if (b < 5) {
  if (!a && c)  { y = 1; }
  z = 2;
}
assert(x+y+z!=3)
```



What's going on here?

- During symbolic execution, we are trying to determine if certain formulas are satisfiable
  - E.g., is a particular program point reachable?
    - Figure out if the path condition is satisfiable
  - E.g., is array access a[i] out of bounds?
    - Figure out if conjunction of path condition and $i<0 \vee i \geq a.length$ is satisfiable
  - E.g., generate concrete inputs that execute the same paths
- This is enabled by powerful SMT/SAT solvers
  - SAT = Satisfiability
  - SMT = Satisfiability modulo theory = SAT++
  - E.g. Z3, Yices, STP

SMT

- Satisfiability Modulo Theory
- SMT instance is a formula in first-order logic, where some function and predicate symbols have additional meaning
- The “additional meaning” depends on the theory being used
  - E.g., Linear inequalities
    - Symbols with extra meaning include the integers, +, -, ×, ≤
  - A richer modeling language than just Boolean SAT
  - Some commonly supported theories: Uninterpreted functions; Linear real and integer arithmetic; Extensional arrays; Fixed-size bit-vectors; Quantifiers; Scalar types; Recursive datatypes, tuples, records; Lambda expressions; Dependent types
- A lot of recent success using SMT solvers
  - In symbolic execution and otherwise...

Predicate transformer semantics

- Predicate transformer semantics give semantics to programs as relations from logical formulas to logical formulas
  - Strongest post-condition semantics: if formula φ is true before program c executes, then formula ψ is true after c executes
    - Like forward symbolic execution of program
  - Weakest pre-condition semantics: if formula φ is true after program c executes, then formula ψ must be true before c executes
    - Like backward symbolic execution of program

Predicate transformer semantics

- Predicate transformers operationalize Hoare Logic
- **Hoare Logic is a deductive system**
  - Axioms and inference rules for deriving proofs of Hoare triples (aka partial correctness assertion)
  - $\{ \varphi \}$ c $\{ \psi \}$ says that if φ holds before execution of program c and c terminates, then ψ holds after c terminates
- Predicate transformers provide a way of producing valid Hoare triples

Hoare logic

- First we need a language for the assertions
  - E.g., first order logic

assertions	$P, Q \in \mathbf{Assn}$	$P ::= \textsf{true} \mid \textsf{false} \mid a_1 < a_2$ $\mid P_1 \land P_2 \mid P_1 \lor P_2 \mid P_1 \Rightarrow P_2 \mid \neg P$ $\mid \forall i.\ P \mid \exists i.\ P$
arithmetic expressions	$a \in \mathbf{Aexp}$	$a ::= \ \dots$
logical variables	$i, j \in \mathbf{LVar}$	

- We also need a semantics for assertions
  - For state σ:Var→Int and interpretation I:LVar→Int we write σ, I ⊨ P if P is true when interpreted under σ, I

Rules of Hoare Logic

$$\textsc{Skip}\ \frac{}{\{P\}\ \textbf{skip}\ \{P\}}$$

$$\textsc{Assign}\ \frac{}{\{P[a/x]\}\ x := a\ \{P\}}$$

$$\textsc{Seq}\ \frac{\{P\}\ c_1\ \{R\} \qquad \{R\}\ c_2\ \{Q\}}{\{P\}\ c_1; c_2\ \{Q\}}$$

$$\textsc{If}\ \frac{\{P \land b\}\ c_1\ \{Q\} \qquad \{P \land \neg b\}\ c_2\ \{Q\}}{\{P\}\ \textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2\ \{Q\}}$$

$$\textsc{Consequence}\ \frac{\vDash (P \Rightarrow P') \qquad \{P'\}\ c\ \{Q'\} \qquad \vDash (Q' \Rightarrow Q)}{\{P\}\ c\ \{Q\}}$$

$$\textsc{While}\ \frac{\{P \land b\}\ c\ \{P\}}{\{P\}\ \textbf{while}\ b\ \textbf{do}\ c\ \{P \land \neg b\}}$$

Soundness and completeness of Hoare Logic

- Semantics of Hoare Triples
  - $\sigma, I \vDash \{P\}\ c\ \{Q\} \quad \triangleq \quad$ if $\sigma, I \vDash P$ and $[\![c]\!]\sigma = \sigma'$, then $\sigma', I \vDash P$
  - $\vDash \{P\}\ c\ \{Q\} \quad \triangleq \quad$ for all σ, I we have $\sigma, I \vDash \{P\}\ c\ \{Q\}$
- Soundness: If there is a proof of $\{P\}\ c\ \{Q\}$, then $\vDash \{P\}\ c\ \{Q\}$
- Relative completeness: If $\vDash \{P\}\ c\ \{Q\}$ then there is a proof of $\{P\}\ c\ \{Q\}$
  - (assuming you can prove the implications in the rule of consequence).

Back to predicate transformers

- Weakest pre-condition semantics
  - Function wp takes command c and assertion Q and returns assertion P such that $\vDash \{P\}c\{Q\}$
  - wp(c, Q) is the **weakest** such condition
    - $\vDash \{P\}c\{Q\}$ if and only if $P \Rightarrow wp(c, Q)$
  - $wp(skip, Q) = Q$
  - $wp(x:=a, Q) = Q[a/x]$
  - $wp(c_1;c_2, Q) = wp(c_1, wp(c_2, Q))$
  - $wp(\text{if } b \text{ then } c_1 \text{ else } c_2, Q) = (b \Rightarrow wp(c_1, Q) \wedge (\neg b \Rightarrow wp(c_2, Q))$

What about loops?

- Two possibilities: do we want the weakest precondition to guarantee termination of the loop?
- **Weakest liberal precondition**: does not guarantee termination
  - Corresponds to **partial** correctness of Hoare triples
  - $\text{wp}(\text{while } b \text{ do } c, Q) = \forall i \in \text{Nat}.\ L_i(Q)$
    where $L_0(Q) = \text{true}$
    $L_{i+1}(Q) = (\neg b \Rightarrow Q) \land (b \Rightarrow \text{wp}(c, L_i(Q)))$
    - Ensures loop terminates in a state that satisfies Q or runs forever

What about loops?

- **Weakest precondition**: guarantees termination
  - Corresponds to **total** correctness of Hoare triples
  - wp(while b do c, Q) = $\exists i \in \text{Nat}.\ L_i(Q)$
    where $L_0(Q) = \text{false}$
    $L_{i+1}(Q) = (\neg b \Rightarrow Q) \land (b \Rightarrow wp(c, L_i(Q)))$
    - Ensures loop terminates in a state that satisfies Q

Strongest post condition

- Function sp takes command c and assertion P and returns assertion Q such that $\vDash \{P\}c\{Q\}$
- sp(c, P) is the **strongest** such condition
  - $\vDash \{P\}c\{Q\}$ if and only if $sp(c, P) \Rightarrow Q$

Strongest post condition

- $sp(\text{skip}, P) = P$
- $sp(x{:=}a, P) = \exists n.\ x{=}a[n/x] \wedge P[n/x]$
- $sp(c_1;c_2, P) = sp(c_2, sp(c_1, P))$
- $sp(\text{if } b \text{ then } c_1 \text{ else } c_2, P) = sp(c_1, b\wedge P) \vee sp(c_2, \neg b\wedge P))$
- $sp(\text{while } b \text{ do } c, P) = \neg b\wedge\exists i.\ L_i(P)$
  where $L_0(P) = P$
  $L_{i+1}(P) = sp(c, b\wedge L_i(P))$
- Weakest preconditions are typically easier to use than strongest postconditions

Symbolic execution

- Symbolic execution can be viewed as a predicate transformation semantics
- Symbolic state and path condition correspond to a formula that is true at a program point
  - e.g., Symbolic state $[x \mapsto \alpha, y \mapsto \beta+7]$ and path condition $\alpha>0$ may correspond to $\alpha>0 \wedge x=\alpha \wedge y=\beta+7$
- Strongest post condition transformations gives us a **forward** symbolic execution of a program
- Weakest pre condition transformations gives us a **backward** symbolic execution of a program

Symbolic execution

- Recall
  - $sp(x{:=}e, P) = \exists n.\ x{=}e[n/x] \wedge P[y/x]$
  - $sp(c_1;c_2, P) = sp(c_2, sp(c_1, P))$
  - $sp(\text{if } b \text{ then } c_1 \text{ else } c_2, P) = sp(c_1, b{\wedge}P) \vee sp(c_2, \neg b{\wedge}P))$
  - $sp(\text{while } b \text{ do } c, P) = \neg b{\wedge}\exists i.\ L_i(P)$
    where $L_0(P) = \text{true}$
    $L_{i+1}(P) = sp(c, b{\wedge}L_i(P))$
- Disjunction encoded by multiple states
  - $\langle \text{if } b \text{ then } c_1 \text{ else } c_2, P\rangle \Downarrow \langle \text{skip}, \{b{\wedge}P, \neg b{\wedge}P\}\rangle\rangle$
  - or equivalently with non-deterministic semantics?
    - $\langle \text{if } b \text{ then } c_1 \text{ else } c_2, P\rangle \mapsto \langle c_1, b{\wedge}P)\rangle$ and
      $\langle \text{if } b \text{ then } c_1 \text{ else } c2, P\rangle \mapsto \langle c_2, \neg b{\wedge}P)\rangle$
- While loops simply unrolled (may fail to terminate)

Symbolic execution and abstract interpretation

- Can we use logical formulas as an abstract domain?
  - Yes! Known as **logical abstract interpretation**
  - Also makes use of SMT solvers
- Can perhaps be seen as an abstract semantics for a concrete predicate transformer semantics?

Back to symbolic execution...

- What about the details?

What values to treat symbolically?

- Primitive values, like ints, floats, and chars seem reasonable
- Strings? Or treat them as arrays of chars?
  - There are theories for strings, but somewhat limited in their reasoning ability
- Pointers?
  - Yeah, you want to have symbolic pointers.
  - But can complicate, e.g., deallocation
- Memory objects?
  - Symbolic regions of memory?
  - Symbolic data structures (e.g., linked lists, trees, hash maps, ...)?
- Files?
- When to concretize?
  - See Klee's behavior on pointers and files. Concretizes a pointer based on each memory object it may point to.
  - Keep a symbolic ternary value e.g., ($\alpha<0$? 0 : β) or fork execution?
    - *Mixing type checking and symbolic execution*. Khoo, Chang, and Foster, PLDI 2010.

What are the sources of symbolic values?

- Inputs to the program? Which?
- Environment?
  - Environment variables? File system? Network messages?

Efficient implementation

- How to efficiently handle many forked executions?
  - Use underlying process abstraction?
  - How to take advantage of lots of shared state between forked executions?
  - How to take advantage of lots of shared/similar queries between forked executions
  - When to concretize?
- Order of evaluating executions?
  - How to explore unexplored code paths?
  - How to avoid getting stuck in "fork bombs"?
- How to reduce the number of SMT queries?

Summary

- Symbolic execution
  - Predicate transformation semantics
  - Allows us to reason about multiple concrete executions
    - But may not allow us to reason about all possible executions
  - Enabled by recent advances in SMT solvers