

Symbolic Execution II

Procedures, Globals, Arrays, Interpolation, and Probability

PhD Course

October 15, 2026

Motivation

Procedures, Data, and Path Reasoning

The previous lecture used configurations $\langle c, \hat{\sigma}, \varphi \rangle$ and

$$\gamma(\hat{\sigma}, \varphi) = \{\nu \hat{\sigma} \mid \nu \models \varphi\}.$$

This lecture asks:

- How do calls and summaries account for shared global variables?
- How do integer-valued procedure pointers determine symbolic call targets?
- What changes when values include arrays?
- What can an infeasible path teach us about other paths?

We retain integer arithmetic and the previous IMP constructs, extending the semantics explicitly at each stage.

Procedure Calls and Composition

IMP with Procedures

Fix global names G and a procedure environment

$$\Delta(f) = (u, L, c_f), \quad G \cap (\{u\} \cup L) = \emptyset.$$

```
proc f(u) {  
  local L;  
  c_f  
}
```

- Call by value, one integer argument and one integer result.
- The command c_f includes an explicit return r on every normal exit.
- The body can access G , its parameter, and its own locals.
- Globals are shared. Each call gets fresh locals, initially zero.
- No name shadowing. Return follows the body; recursive calls are allowed.

The caller may assign the result to a local or a global variable.

Procedure Calls and Composition

Local and Global Stores

The active store combines two disjoint maps:

$$\sigma = \ell \uplus g, \quad g = \sigma|_G.$$

Expressions and ordinary assignments use this combined store. Updating a global changes g ; updating a local changes ℓ .

Define call entry and caller resumption by

$$\begin{aligned} \text{enter}_f(v, g) &= \text{init}_f(v) \uplus g, \\ \text{resume}(\ell, g') &= \ell \uplus g'. \end{aligned}$$

Here $\text{init}_f(v)$ sets the parameter to v and callee locals to zero. Resumption preserves saved caller locals and takes the *new* globals.

Procedure Calls and Composition

The Small-Step Machine

Configurations are $\langle c, \sigma, \kappa \rangle$. The store contains the active locals and globals. The stack contains only return frames:

$$\kappa ::= [] \mid \text{ret}(\ell, c) :: \kappa.$$

- $\text{ret}(\ell, c)$ records only the caller's local variables and the command c to execute after return. The continuation contains a distinguished result placeholder •.

$\langle \text{skip}, \sigma, [] \rangle$ is a normal final configuration, with no outgoing transition. An infinite transition sequence represents divergence.

The following rules define $\longrightarrow_{\Delta}$ explicitly. Write $\longrightarrow_{\Delta}^*$ for zero or more steps.

Procedure Calls and Composition

Commands and Expression Evaluation

$$\begin{aligned} a &::= n \mid x \mid \bullet \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 \cdot a_2, \\ b &::= \text{true} \mid \text{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \wedge b_2, \\ c &::= \text{skip} \mid x := a \mid c_1; c_2 \mid y := f(a) \mid \text{return } a \\ &\quad \mid \text{if } b \text{ then } c_1 \text{ else } c_2 \mid \text{while } b \text{ do } c. \end{aligned}$$

Evaluate expressions in the active store:

$$\begin{aligned} \llbracket n \rrbracket_\sigma &= n, & \llbracket x \rrbracket_\sigma &= \sigma(x), \\ \llbracket a_1 \circ a_2 \rrbracket_\sigma &= \llbracket a_1 \rrbracket_\sigma \circ \llbracket a_2 \rrbracket_\sigma, & \circ &\in \{+, -, \cdot\}, \\ \llbracket a_1 \diamond a_2 \rrbracket_\sigma &= (\llbracket a_1 \rrbracket_\sigma \diamond \llbracket a_2 \rrbracket_\sigma), & \diamond &\in \{=, \leq\}. \end{aligned}$$

Procedure Calls and Composition

Assignment and Sequencing

$$\frac{}{\langle x := a, \sigma, \kappa \rangle \longrightarrow_{\Delta} \langle \text{skip}, \sigma[x \mapsto \llbracket a \rrbracket_{\sigma}], \kappa \rangle} \text{Assign}$$

$$\frac{}{\langle \text{skip}; c_2, \sigma, \kappa \rangle \longrightarrow_{\Delta} \langle c_2, \sigma, \kappa \rangle} \text{Seq-Done}$$

$$\frac{\langle c_1, \sigma, \kappa \rangle \longrightarrow_{\Delta} \langle c'_1, \sigma', \kappa \rangle}{\langle c_1; c_2, \sigma, \kappa \rangle \longrightarrow_{\Delta} \langle c'_1; c_2, \sigma', \kappa \rangle} \text{Seq-Step}$$

Sequencing is handled in the command syntax. Neither rule changes the stack. When the head command is a call, the dedicated call-in-sequence rule below is used; Seq-Step is not applied to call heads.

Assignment updates a local or a global according to the destination's declared scope.

Procedure Calls and Composition

Conditionals

$$\frac{\llbracket b \rrbracket_{\sigma} = \text{true}}{\langle \text{if } b \text{ then } c_1 \text{ else } c_2, \sigma, \kappa \rangle \longrightarrow_{\Delta} \langle c_1, \sigma, \kappa \rangle} \text{If-T}$$
$$\frac{\llbracket b \rrbracket_{\sigma} = \text{false}}{\langle \text{if } b \text{ then } c_1 \text{ else } c_2, \sigma, \kappa \rangle \longrightarrow_{\Delta} \langle c_2, \sigma, \kappa \rangle} \text{If-F}$$

The guard selects exactly one branch. The transition changes neither the store nor the stack.

Procedure Calls and Composition

While Loops

Let $w \equiv \text{while } b \text{ do } d$.

$$\frac{}{\langle w, \sigma, \kappa \rangle \longrightarrow_{\Delta} \langle \text{if } b \text{ then } (d; w) \text{ else skip}, \sigma, \kappa \rangle} \text{ While}$$

When the guard is true, the rules give

$$\begin{aligned} \langle w, \sigma, \kappa \rangle &\longrightarrow_{\Delta} \langle \text{if } b \text{ then } (d; w) \text{ else skip}, \sigma, \kappa \rangle \\ &\longrightarrow_{\Delta} \langle d; w, \sigma, \kappa \rangle. \end{aligned}$$

After the body, Seq-Done resumes the loop with the updated store. For a false guard, the conditional becomes skip.

Procedure Calls and Composition

Small-Step Semantics: Entering a Call

The call saves only the caller's locals and pushes a return frame. The callee starts with fresh locals and the current globals.

For $y := f(a); c$, the call rule captures c directly in the return frame:

$$\text{ret}(\sigma|_{\text{dom}(\sigma) \setminus G}, y := \bullet; c) :: \kappa.$$

The callee must return before the caller executes c . The corresponding rule is:

$$\frac{\Delta(f) = (u, L, c_f) \quad v = \llbracket a \rrbracket_\sigma}{\langle y := f(a); c, \sigma, \kappa \rangle \longrightarrow_\Delta \langle c_f, \text{enter}_f(v, \sigma|_G), \text{ret}(\sigma|_{\text{dom}(\sigma) \setminus G}, y := \bullet; c) :: \kappa \rangle} \text{Call-Seq}$$

Procedure Calls and Composition

Small-Step Semantics: Returning

$$\frac{w = \llbracket r \rrbracket_{\sigma}}{\langle \text{return } r, \sigma, \text{ret}(\ell, c) :: \kappa \rangle \longrightarrow_{\Delta} \langle c[w/\bullet], \text{resume}(\ell, \sigma|_G), \kappa \rangle} \text{Return}$$

Restore caller locals, retain the callee's globals, substitute the return value into the saved continuation, and execute it. If the continuation assigns a global, that assignment updates the shared global. The next rule depends on the newly exposed stack frame.

For $f(u)$ with global *ticks*:

```
u := u + 1;  
ticks := ticks + 1;  
return u
```

From $x = 4$, $ticks = 0$, the call $y := f(x)$ gives $x = 4$, $y = 5$, $ticks = 1$.

Procedure Calls and Composition

Symbolic Procedure Call

Symbolic configurations extend the concrete machine with a path condition: $\langle c, \hat{\sigma}, \varphi, \kappa \rangle$. Let $t = \llbracket a \rrbracket_{\hat{\sigma}}$, $\ell = \hat{\sigma}|_{\text{dom}(\hat{\sigma}) \setminus G}$, and $\hat{\sigma} = \text{enter}_f(t, \hat{\sigma}|_G)$. Then

$$\frac{\Delta(f) = (u, L, c_f)}{\langle y := f(a); c, \hat{\sigma}, \varphi, \kappa \rangle \longrightarrow_s \langle c_f, \hat{\sigma}, \varphi, \text{ret}(\ell, y := \bullet; c) :: \kappa \rangle} \text{ S-Call-Seq}$$

The argument and current globals become symbolic inputs to the callee. The caller locals and continuation are retained in the return frame. No path constraint changes at call entry.

Procedure Calls and Composition

Symbolic Procedure Return

At the return command, evaluate the return expression in the callee's symbolic store. Let $t = \llbracket r \rrbracket_{\hat{\sigma}}$. Then

$$\frac{}{\langle \text{return } r, \hat{\sigma}, \varphi, \text{ret}(\ell, c) :: \kappa \rangle \longrightarrow_s \langle c[t/\bullet], \text{resume}(\ell, \hat{\sigma}|_G), \varphi, \kappa \rangle} \text{S-Return}$$

The transition restores caller locals, keeps the callee's symbolic globals, and substitutes the symbolic return term into the saved continuation. Return also leaves the path condition unchanged.

Symbolic Procedure Execution

A Generic Symbolic Run of a Procedure

Analyze f once from symbolic inputs rather than from a particular caller. Let α be a fresh parameter symbol and let $(\alpha_g)_{g \in G}$ be a fresh tuple of global-input symbols, one for each global variable. Start from

$$\langle c_f, \text{enter}_f(\alpha, \{g \mapsto \alpha_g\}), \phi_f, \epsilon \rangle.$$

Here $\phi_f = \llbracket P_f \rrbracket_{\{u \mapsto \alpha, g \mapsto \alpha_g\}}$ is the precondition assumed while analyzing f . The tuple α_g is the global component of the initial symbolic store, not a program variable and not a store by itself. Each feasible symbolic path ℓ reaches a return configuration:

$$\langle \text{return } r_\ell, \hat{\sigma}_\ell, \varphi_\ell, \epsilon \rangle$$

Symbolic Procedure Execution

Instantiating a Procedure Run at a Call

Suppose the executor reaches

$$\langle y := f(a); c, \hat{\sigma}, \psi, \kappa \rangle.$$

Instantiate the procedure-input symbols using the symbolic expressions at this call:

$$\theta(\alpha) = \llbracket a \rrbracket_{\hat{\sigma}}, \quad \theta(\alpha_g) = \hat{\sigma}(g) \quad (g \in G).$$

Reuse the analysis only after checking its assumed precondition:

$$\psi \Rightarrow \phi_f \theta.$$

For each procedure path ℓ ending in

$$\langle \text{return } r_\ell, \hat{\sigma}_\ell, \varphi_\ell, \epsilon \rangle,$$

the call instantiates it to

$$\langle \text{return } r_\ell, \hat{\sigma}_\ell \theta, \psi \wedge \llbracket \varphi_\ell \rrbracket_\theta, \kappa \rangle.$$

Symbolic Procedure Execution

Example: Reusing a Symbolic Branch

Consider

```
proc addPositive(u) {  
  if u > 0 then g := g + u else skip;  
  return g  
}
```

For the global g , write α_g for its fresh input symbol. Analyze under $P_f(\alpha, \alpha_g) \equiv \alpha \geq 0$. The two final leaves are

$$\langle \text{return } g, \{u \mapsto \alpha, g \mapsto \alpha_g + \alpha\}, \alpha \geq 0 \wedge \alpha > 0, \epsilon \rangle$$

$$\langle \text{return } g, \{u \mapsto \alpha, g \mapsto \alpha_g\}, \alpha \geq 0 \wedge \alpha \leq 0, \epsilon \rangle.$$

Symbolic Procedure Execution

Example: Instantiating the Leaves

At a call site with $\psi \equiv X \geq 2 \wedge \Gamma = 5$, $\hat{\sigma}(x) = X$, $\hat{\sigma}(g) = \Gamma$, and $y := \text{addPositive}(x + 1)$,

$$\theta(\alpha) = X + 1, \quad \theta(\alpha_g) = \Gamma, \quad \psi \Rightarrow X + 1 \geq 0.$$

The positive leaf becomes

$$\langle \text{return } g, \{u \mapsto X + 1, g \mapsto \Gamma + X + 1\}, X \geq 2 \wedge \Gamma = 5 \wedge X + 1 > 0, \kappa \rangle.$$

The else leaf has path condition $X \geq 2 \wedge \Gamma = 5 \wedge X + 1 \leq 0$, which is unsatisfiable and is discarded.

Arrays and Concretization

IMP with Array Values

Return to the original IMP model of Lecture I: configurations are $\langle c, \sigma \rangle$, with no procedures, globals, or call stack. Add a distinct sort of array variables A, B :

$$\text{Array} = \mathbb{Z} \rightarrow \mathbb{Z}, \quad a ::= \dots \mid A[a], \quad c ::= \dots \mid A[a] := a \mid A := B.$$

A typed store maps scalar variables to integers and array variables to array values.

- Arrays are total functions. There are no implicit bounds or allocation faults.
- For a bounded array, introduce a length n and check $0 \leq i < n$.
- $A := B$ copies an array *value*. Later updates of A do not change B .

These are functional arrays, not mutable heap references with aliasing.

Arrays and Concretization

The Theory of Arrays

Array terms use reads and functional updates:

$$\text{select}(M, i), \quad \text{store}(M, i, v).$$

$$\text{select}(\text{store}(M, i, v), i) = v,$$

$$i \neq j \Rightarrow \text{select}(\text{store}(M, i, v), j) = \text{select}(M, j).$$

Extensionality identifies arrays that agree at every index:

$$(\forall i. \text{select}(M, i) = \text{select}(N, i)) \Rightarrow M = N.$$

Also:

$$i \neq j \Rightarrow \text{store}(\text{store}(M, i, v_1), j, v_2) = \text{store}(\text{store}(M, j, v_2), i, v_1)$$

$$\text{store}(\text{store}(M, i, v_1), i, v_2) = \text{store}(M, i, v_2)$$

Arrays and Concretization

Concrete and Symbolic Array Updates

Concrete evaluation reads the current array value:

$$\llbracket A[a] \rrbracket_{\sigma} = \sigma(A)(\llbracket a \rrbracket_{\sigma}).$$

For $i = \llbracket a_1 \rrbracket_{\sigma}$, $v = \llbracket a_2 \rrbracket_{\sigma}$:

$$\langle A[a_1] := a_2, \sigma \rangle \longrightarrow \langle \text{skip}, \sigma[A \mapsto \sigma(A)[i \mapsto v]] \rangle.$$

Symbolically, let $t_i = \llbracket a_1 \rrbracket_{\hat{\sigma}}$, $t_v = \llbracket a_2 \rrbracket_{\hat{\sigma}}$:

$$\langle A[a_1] := a_2, \hat{\sigma}, \varphi \rangle \longrightarrow_s \langle \text{skip}, \hat{\sigma}[A \mapsto \text{store}(\hat{\sigma}(A), t_i, t_v)], \varphi \rangle.$$

A symbolic read is $\llbracket A[a] \rrbracket_{\hat{\sigma}} = \text{select}(\hat{\sigma}(A), \llbracket a \rrbracket_{\hat{\sigma}})$.

Arrays and Concretization

Array Copying

Copying a whole array updates the active store:

$$\frac{}{\langle A := B, \sigma \rangle \longrightarrow \langle \text{skip}, \sigma[A \mapsto \sigma(B)] \rangle} \text{Array-Copy}$$

Symbolically:

$$\langle A := B, \hat{\sigma}, \varphi \rangle \longrightarrow_s \langle \text{skip}, \hat{\sigma}[A \mapsto \hat{\sigma}(B)], \varphi \rangle.$$

The concrete and symbolic rules extend the original procedure-free IMP semantics. Arrays are values. A later update of A produces a new value without modifying B .

Arrays and Concretization

A Symbolic Read After a Write

```
A[i] := 7;  
x := A[j]
```

Initially, let $A = M$, $i = \alpha_i$, $j = \alpha_j$. Then

$$\hat{\sigma}'(x) = \text{select}(\text{store}(M, \alpha_i, 7), \alpha_j) = \text{ite}(\alpha_i = \alpha_j, 7, \text{select}(M, \alpha_j)).$$

The obligation $x = 7$ holds under $\alpha_i = \alpha_j$. Without it, a counterexample has $\alpha_i = 0, \alpha_j = 1, M(1) = 2$.

Arrays and Concretization

Concretization with Array Symbols

A valuation now maps scalar symbols to integers and array symbols to functions $\mathbb{Z} \rightarrow \mathbb{Z}$.

$$\gamma(\hat{\sigma}, \varphi) = \{\nu \hat{\sigma} \mid \nu \models_{\text{Int+Array}} \varphi\}.$$

Interpret `select` as function application and `store` as functional update. The evaluation lemma still holds:

$$\llbracket A[a] \rrbracket_{\nu \hat{\sigma}} = \llbracket \text{select}(\hat{\sigma}(A), \llbracket a \rrbracket_{\hat{\sigma}}) \rrbracket_{\nu}.$$

An array model must assign the entire array, even if a printed model describes it with finitely many exceptions to a default value.

Arrays and Concretization

Choosing Concrete Values During Analysis

Suppose a solver model suggests $\alpha_i = k$. To specialize the index soundly for bug finding:

$$\varphi' = \varphi \wedge \alpha_i = k, \quad \gamma(\hat{\sigma}, \varphi') \subseteq \gamma(\hat{\sigma}, \varphi).$$

Under this constraint, rewriting $\text{select}(M, \alpha_i)$ to $\text{select}(M, k)$

- Loses executions with other indices.
- Replacing the index but omitting $\alpha_i = k$ can mismatch the input and the read.
- Naming a read by fresh β with $\beta = \text{select}(M, \alpha_i)$ loses no precision.

Procedure Pointers

Procedures as Distinct Integers

For indirect calls, fix a finite target set $\mathcal{F}$ of declared procedures with the same signature: one integer argument and one integer result.

$$\text{id} : \mathcal{F} \hookrightarrow \mathbb{Z}, \quad C = \{\text{id}(f) \mid f \in \mathcal{F}\}.$$

Add $y := \text{call } p(a)$, where p is any integer expression.

- Procedure identifiers are ordinary integer values, with no closures or captured local environments.
- They may be assigned to variables, passed as arguments, returned, or stored in arrays.
- A value outside C causes an explicit invalid-call error.

Typed array procedures are outside this scalar target set. We do not model code addresses, provenance, or dynamic loading.

Procedure Pointers

Concrete Dispatch and Calls

$$\frac{f \in \mathcal{F} \quad \llbracket p \rrbracket_{\sigma} = \text{id}(f)}{\langle y := \text{call } p(a); c, \sigma, \kappa \rangle \longrightarrow_{\Delta} \langle y := f(a); c, \sigma, \kappa \rangle} \text{ Dispatch-Seq}$$
$$\frac{\llbracket p \rrbracket_{\sigma} \notin C}{\langle y := \text{call } p(a); c, \sigma, \kappa \rangle \longrightarrow_{\Delta} \text{error}} \text{ Invalid-Call-Seq}$$

The direct Call rule evaluates the argument and enters the selected body. Dispatch itself leaves the store unchanged.

error has no outgoing transition and is not normal termination. The error transition discards the current command and all pending return frames.

Procedure Pointers

Symbolic Target Resolution

Suppose the executor reaches

$$\langle y := \text{call } p(a); c, \hat{\sigma}, \psi, \kappa \rangle.$$

For each $f \in \mathcal{F}$, retain the target branch

$$\frac{f \in \mathcal{F}}{\langle y := \text{call } p(a); c, \sigma, \psi, \kappa \rangle \longrightarrow \langle y := f(a); c, \sigma, \psi \wedge \llbracket p \rrbracket_{\hat{\sigma}} = \text{id}(f), \kappa \rangle} \text{ S-Dispatch-Seq}$$

The invalid-target branch has condition $\psi \wedge \bigwedge_{f \in \mathcal{F}} \llbracket p \rrbracket_{\hat{\sigma}} \neq \text{id}(f)$.

$$\frac{}{\langle y := \text{call } p(a); c, \sigma, \psi, \kappa \rangle \longrightarrow \langle \text{error}, \sigma, \psi \wedge \bigwedge_{f \in \mathcal{F}} \llbracket p \rrbracket_{\hat{\sigma}} \neq \text{id}(f), \kappa \rangle} \text{ Invalid-Call-Seq}$$

Prune only proved-infeasible branches.

Procedure Pointers

Dispatch with Different Global Effects

Let $G = \{g\}$, $\text{id}(\text{bump}) = 1$, and $\text{id}(\text{keep}) = 2$. Both generic analyses have precondition $\top$. Their final leaves are

$$\begin{aligned}\text{bump} : & \langle \text{return } u + g, \{u \mapsto \alpha, g \mapsto \alpha_g + 1\}, \top, \epsilon \rangle, \\ \text{keep} : & \langle \text{return } u, \{u \mapsto \alpha, g \mapsto \alpha_g\}, \top, \epsilon \rangle.\end{aligned}$$

At $y := \text{call } p(x)$, the target branches use $\psi_{\text{bump}} = \psi \wedge \llbracket p \rrbracket_{\hat{\sigma}} = 1$ and $\psi_{\text{keep}} = \psi \wedge \llbracket p \rrbracket_{\hat{\sigma}} = 2$. For $x = 4, g = 0$, instantiating the leaves gives:

p	y	g after the call
1	5	1
2	4	0

Separately prove $\llbracket p \rrbracket_{\hat{\sigma}} = 1 \vee \llbracket p \rrbracket_{\hat{\sigma}} = 2$ to exclude the invalid-call branch.

Procedure Pointers

Array Dispatch and Concretization

A table can contain integer procedure identifiers:

```
T[0] := 1;  
T[1] := 2;  
p := T[i];  
y := call p(x)
```

At a call configuration $\langle y := \text{call } p(x); c, \hat{\sigma}, \psi, \kappa \rangle$, let $\hat{\sigma}(i) = \alpha_i$. Read-over-write gives

$$\llbracket p \rrbracket_{\hat{\sigma}} = \text{ite}(\alpha_i = 0, 1, 2).$$

The two target branches are

$$\psi_{\text{bump}} = \psi \wedge \alpha_i = 0, \quad \psi_{\text{keep}} = \psi \wedge \alpha_i = 1.$$

Choosing one model value for p is an underapproximation. Without $\alpha_i = 0 \vee \alpha_i = 1$, also explore the invalid-target branch.

Craig Interpolation

Nested Conditionals and an Error Path

Consider a program with a common assertion after two nested conditionals:

```
if x >= 0 then
  if x <= 10 then y := x + 1 else y := 1
else
  if x >= -10 then y := 1 - x else y := 1;
assert y > 0
```

Take the candidate error path through the first two true branches and then the failing assertion.
Its prefix and suffix formulas are

$$F \equiv \alpha \geq 0 \wedge \alpha \leq 10 \wedge y = \alpha + 1, \quad G \equiv y \leq 0.$$

The candidate is infeasible because $F \wedge G$ is unsatisfiable.

Craig Interpolation

Craig Interpolants

For an unsatisfiable $F \wedge G$, an interpolant I satisfies

$$F \Rightarrow I, \quad I \wedge G \Rightarrow \text{false}, \quad \text{sym}(I) \subseteq \text{sym}(F) \cap \text{sym}(G).$$

The symbol condition concerns nonlogical symbols.

- I records enough about the prefix to exclude the erroneous suffix.
- Prefix-only intermediate names disappear from the interface.

Craig Interpolation

An Interpolant at the Assertion

For this infeasible path, choose

$$I \equiv y \geq 1.$$

Verification:

$$F \Rightarrow y = \alpha + 1 \geq 1,$$

$$I \wedge G \Rightarrow \text{false}.$$

The shared name is y . The input symbol α is local to the prefix, so the interpolant records only the fact the assertion needs.

Craig Interpolation

Reusing the Interpolant

The other true branch of the outer conditional has prefix

$$F' \equiv \alpha > 10 \wedge y = 1.$$

It also implies $I \equiv y \geq 1$. Thus the executor can reuse I at the join point and prune the same suffix $G \equiv y \leq 0$, without solving a new prefix–suffix query for this branch.

This is the practical use of interpolation: retain a small fact that blocks a whole family of impossible continuations.