Symbolic Execution is (Not Quite) All You Need

Version 0.0.2

Sophie Smithburg

***Abstract*—Given a programming language implementation running on a host machine and a grammar, you can use symbolic execution and formal ISA semantics as oracles to extract a formal labeled transition system that simulates the implementation's behavior.**

In plain terms: You can automatically derive executable formal specs from implementations—no one writes specs, everyone writes implementations, but you can't prove things about implementations directly. This technique lets you get the spec from the implementation (given its grammar).

***Index Terms*—TODO add index terms**

I. Introduction

We can model a programming language executing on a host machine as a host labeled transition system H which, after having been fed $\mathcal{I}$, the implementation, simulates G—this is to say the PL is G, its implementation is $\mathcal{I}$, and the host machine is H. If we observe a set of traces $\tau \in \mathcal{T}$ of executions of G recorded at the level of H, we can use a specific algorithm to extract an LTS G' that is bisimilar to G.

In concrete terms, one can think of H as something like x86 or ARM, and $\mathcal{I}$ as something like CPython or V8. This matters because in practice everyone builds implementations. No one writes specs, but you can't prove things about implementations directly for the most part, and you can about specs. So in order to have an effective ratchet in the sense of the LangSec 2017 Perry Metzger talk, we want to be able to get these specs for these implementations.

To prove G' is bisimilar to G, we need to *have* G in a formalism that shares the same proof basis as G'—but in practice, this is never the case. The problem we're trying to solve practically speaking is inferring G' when no one cares to actually write out G, they're all focused on $\mathcal{I}$. So what we do instead is try to prove bisimilarity with respect to $H_{\mathcal{I}}$, the LTS H after executing $\mathcal{I}$, by way of quotienting over all the implementation details of H that aren't causally controllable by way of inputs to G in the form of program structure or codata.

II. Notation

- Σ is the concrete runtime space, and in our case, it is parametric over the definition of an ISA. It's basically the main memory, registers, CPU flags, any state that is documented for a particular CPU or host machine. In the case of the more abstract view, like a labeled transition system, which we focus on in this paper.
- X is computed indirectly, but can be defined directly. X is the set of all the state of the subsystem (like the programming language under analysis) that can be controlled by a program in that programming language. It is possible to extend this in the future to include input by way of any predefined input channel, but modeling codata introduces non-determinism, which is out of scope for the current paper.
- π is the mapping from Σ to X, which we get precisely by the indirect computation mechanism hinted at above.
- H is the host machine, it can be seen as an LTS or a bog standard ISA
- G is the true transition system, or notionally, what would be the specification of the programming language if the programming language were defined by the implementation? In some very real sense, a lot of programming languages are. Take, for example, C, Python, and the fact that it's been stated about the specifications in the Python enhancement proposal process and other documents in the standard that if one were to follow them, it's likely, without other guidance, they would end up with an entirely different language.
- $\mathcal{I}$ is the programming language implementation under analysis
- $H_{\mathcal{I}}$ is the LTS H after executing $\mathcal{I}$—i.e., an x86 machine after executing the CPython binary, or an ARM machine after executing a V8 implementation
- G' is the transition system we extract from traces $\tau \in \mathcal{T}$ of running $H_{\mathcal{I}}$, i.e. the host machine H or host labeled transition system without loss of generality. Additionally, G' is composed from R.
- Γ is the formal grammar for $\mathcal{I}$
- $\gamma \in \Gamma$ is a production in the grammar
- holes(γ) is the set of holes in production γ
- κ_h is the sentinel value for hole h in production γ
- $\mathcal{C}$ is a covering set of programs that enumerate all productions in Γ, and alternatives in each production, in a minimized way to be elaborated upon later
- $\tau \in \mathcal{T}$ is a trace, or a sequence of states and labels for the transitions between them, generated when we run the host machine H over the programming language implementation $\mathcal{I}$ and a program from $\mathcal{C}$, which implies all the values relevant to updates (it is of course more convenient when the values for updates are provided for us; we presume this, we think without loss of generality, but perhaps that's wrong!)
- $\mathcal{T}$ is the set of all traces τ
- A **Hole-to-Hole (HTH) block** is the maximal straight-line region of execution between two consecutive holes

in evaluation order—see Appendix A for visualizations
- L is the set of HTH labels; each $\ell = (\gamma, h_i, h_j)$ identifies the transition from hole h_i to hole h_j within production γ, where $h_i, h_j \in \text{holes}(\gamma)$ are consecutive in evaluation order
- $\ell \in L$ is a label for a step in our transition system, identifying which HTH region the step corresponds to
- $R_\ell(x, x') := \text{Guard}_\ell(x) \wedge \text{Update}_\ell(x, x')$ summarizes the preconditions and state transformations for a step labeled ℓ
- $R^* = \bigcup_\ell R_\ell$ and its transitive closure
- $\text{Reach}(H_\mathcal{I})$ is the set of reachable states in $H_\mathcal{I}$
- $\text{BStates}(\mathcal{I}) \subseteq \text{Reach}(H_\mathcal{I})$ is the set of *all* reachable states with branching behavior—i.e., $s \in \text{BStates}(\mathcal{I})$ iff $|\text{Alt}(s)| > 1$. The branching oracle O discovers these via symbolic execution at the Σ level.
- $\text{Alt}(s)$ is the set of *all* feasible branch outcomes from state s that are reachable by modifying $\pi(s)$. This is the completeness constraint: if outcome o is achievable by changing X-state at s, then $o \in \text{Alt}(s)$. Branches depending on state outside X are either implementation-internal (irrelevant to G') or trigger π-refinement to include that state.
- $O(s)$ is the branching oracle: given $s \in \text{BStates}(\mathcal{I})$, it produces constraints over $\pi(s)$ that achieve each alternative in $\text{Alt}(s)$
- $\text{ReplayApply}(s, c) \Downarrow o$ means replaying from state s with $\pi(s)$ modified by constraint c realizes branch outcome o
- $O^*(\mathcal{T})$ is the fixpoint up to behavioral equivalence: for each branch outcome $o \in \text{Alt}(s)$ at each s along traces in $O^*(\mathcal{T})$, we require *at least one exemplar trace* realizing o. We do not generate all traces—only one representative per behavioral pattern (same control flow, same HTH structure). Additionally, π has stabilized. This suffices for completeness: every reachable behavior has an exemplar from which we extract R_ℓ.
- $G' \preceq M$ states that G' simulates M, which is to say that all behaviors of M have corresponding behaviors in G'

III. Main Claim

We reconstruct behavior piecewise along traces and each trace is a sound reconstruction because it almost literally replays the trace we saw. And also we use a model of the host machine that knows about branches to achieve completeness by adding new traces for all uncovered paths. If you manage soundness and completeness of reconstruction that's basically the whole game.

A. Oracles

We require two kinds of oracles: one for branching behavior and one for value transformations. Neither is counterfactual—both are instantiated by real tools.

1) Branching: The branching oracles we require are in no way counterfactual. Some aspects are partially captured by symbolic execution and then the rest of that is fully captured by formal ISA semantics.

a) Completeness.: For all $\mathcal{I}$-relevant branch states and all feasible outcomes, the oracle can produce a constraint that achieves that outcome:

$$\forall s \in \text{BStates}(\mathcal{I}).\ \forall o \in \text{Alt}(s).\ \exists c \in O(s).\ \text{ReplayApply}(s, c) \Downarrow o$$

2) Value Transformation: What we need for value transformation inference is provided fully by symbolic execution engines. When symbolically executing a basic block, the engine produces symbolic expressions showing how input values map to output values—this directly gives us the $\text{Update}(x, x')$ component of each R_ℓ.

B. Proof Structure

1) **Given**: H (host), $\mathcal{I}$ (implementation), Γ (grammar)
2) **Given**: O (branching oracle)—instantiated by SSE or formal ISA semantics
3) **Define**: $\mathcal{T}_0$ = initial traces from covering set $\mathcal{C}$
4) **Define**: $O^*(\mathcal{T}_0)$ = fixpoint (apply O until no new traces)
5) **Assume**: Symbolic execution recovers Guard $\wedge$ Update from HTH regions
6) **Construct**: G' from $\{R_\ell \mid \ell \text{ derived from } O^*(\mathcal{T}_0)\}$
7) **Prove**: $G' \preceq H_\mathcal{I}$

C. Soundness and Completeness

a) Soundness.: Each R_ℓ is computed as the relational transformer associated with a hole-to-hole basic block. Our relational transformer is as sound as the symbolic execution engine we use to compute said transformer.

b) Completeness.: This follows basically from the definition of the branching oracle and the fixpoint. Since by definition the branching oracle can detect all branching behavior at the level of H, were there to be any state reachable in $H_\mathcal{I}$ that we did not find at least one exemplar trace for, then it's a contradiction with the assumption that the oracle is complete.

D. Scope of This Paper

This current paper attempts to prove that it is possible to do such a reconstruction. We leave it for future work to fully implement such an algorithm and prove it correct. We don't consider it necessary to prove that the oracles exist because they do in the form of formal specifications of ISAs and symbolic execution engines. Our intent is simply to prove that, given such tools, it should be possible to extract semantics from the implementations and an algorithm corresponding to this sketch.

We're only focused on ISAs, like Industrial ISAs, running industrial programming languages. We are not yet aiming to tackle them in their full generality, but rather to model as large of a slice of an industrial implementation as is possible. To this end, we make some radical choices in terms of scope to make proving any result possible, knowing that we must return to each of these in future work.

Out of scope for this paper are:

- non-determinism
- concurrency
- lazy evaluation

- typing or any static analysis or codata

Each of these scope limitations would likely warrant their own follow-up paper in themselves. All of that being said, we think that this is still useful work, as the core semantics of a large class of industrially interesting programming languages can be captured this way.

IV. Methodology

We use sentinels κ_h in templates $P \in \mathcal{C}$ to determine the correspondence from a program with a particular AST construct executing and the resulting trace—particularly the mapping from sentinel holes to their corresponding values in the trace. We use this both to learn a mapping from syntactic representation to runtime representation, as well as to delineate the bounds of hole-to-hole basic blocks.

A. Structured Programming and Lexical Scope

[NOTE: Sophie flagged this may need to move earlier]

To explain what a hole-to-hole basic block is, we must first introduce the requirement of lexical scope in structured programming languages. Structured programming is necessary because it gives us the property that jumps are always lexically bounded with respect to the AST. For κ_{h_1} to κ_{h_2}, that basic block, we identify the points in the trace that correspond to that. From those points in the trace, we can extract information like memory regions to feed to a selective symbolic execution engine to selectively symbolically execute just that region between those two holes.

B. Learning the Configuration

We are also using this to learn π. The idea is that we learn what our configuration should look like. Our configuration is defined by the bindings from AST paths to runtime values, so if something isn't AST-bindable or AST-referable, then it can't end up in our projection π.

Differential causality testing. To learn π concretely, we run paired experiments. For each hole h in template P, we execute P with two different sentinel values: $P[h \leftarrow \kappa_1]$ and $P[h \leftarrow \kappa_2]$. Comparing the resulting traces τ_1 and τ_2, we identify the positions where they differ. These differing positions represent state that is *causally influenced* by the value at hole h—and therefore must be included in X.

The key insight is that X is exactly the *transitive closure of AST-bound state*. Nothing enters X without first being directly bound to an AST position (by determinism—all state changes trace back to program structure). A variable's value is in X because it was assigned from an expression; that expression's value is in X because it was computed from subexpressions bound to AST nodes; and so on. The differential test reveals this transitive closure: if changing a sentinel at hole h eventually affects some state S in the trace, then S is transitively reachable from h's AST binding, and belongs in X.

C. Control Flow and Value Transformation

We extract control flow and value transformation along this same part of the technique. The selective symbolic execution gives us path constraints. Say we're doing the if-then-else situation where we are going from `cond` to the then-branch. We'll have a path constraint induced by this that `cond` must evaluate to something truthy. We can flip that path constraint, and that gives us the other thing we have to explore. We would have already gotten that by way of our covering set in the case of truthy and falsy—at least for literal `true` and `false` we would have. What's key is that we get an SMT formulation in terms of things we have determined bindings for in terms of our AST—we've extracted the guard conditions for our labeled transition system.

For value inference: since these hole-to-hole blocks do not involve branching, we can extract the relational transformer along this basic block. We use the symbolic statement of inputs in the configuration mapping to symbolic expressions in the output. We can use those, plus the mapping from syntactic literal to runtime type, to define the value transformation aspect of semantics.

D. Bootstrapping and Co-Refinement

The boundary detection problem. Finding HTH regions requires knowing where each hole's computation "happens" in the trace. But we cannot simply look for where sentinel *values* appear—values can be inlined, constant-folded, or otherwise transformed by the implementation. We need to detect where *computation* happens, not just where values appear.

Moreover, we cannot use syntactic heuristics like "this AST node is named `if`" or "this node has two children." The entire point is to infer semantics from implementation behavior. If we hardcode syntactic patterns, we assume what we are trying to learn.

Sentinel expressions. Instead of sentinel values, we use sentinel *expressions* that force observable computation—e.g., `(x := sentinel; sentinel)` where the assignment leaves a trace signature. When the assignment appears in the trace, we know that hole was evaluated. But to design such expressions, we must know which operations have observable sequencing.

Learning algebraic laws from R_ℓ. Here is the key insight: R_ℓ is an SMT formula produced by symbolic execution. SMT formulas use SMT-LIB operations: `bvadd`, `bvmul`, `store`, etc. These operations have *precisely defined algebraic laws*—this is the SMT-LIB standard. We do not pattern-match against heuristics; we read exact algebraic properties from the formula structure to identify what *weakens* ordering detection:

- R_ℓ contains $\texttt{bvadd}(x, y) \Rightarrow$ commutative $\Rightarrow$ *cannot* use this to detect evaluation order
- R_ℓ contains $\texttt{bvsub}(x, y) \Rightarrow$ non-commutative $\Rightarrow$ *can* use this to detect order
- R_ℓ contains `store` $\Rightarrow$ has observable side effects $\Rightarrow$ *can* use this to detect order

Sequencing analysis. Commutative operations cannot detect evaluation order: if $a + b = b + a$, we cannot tell from the

result whether a or b was evaluated first. Non-commutative operations and side effects *can* detect order. By reading the algebraic laws from R_ℓ, we identify which operations are safe for sequencing detection.

The bootstrapping loop. This creates a productive cycle:

1) Start with rough HTH regions from initial trace differences and observable side effects
2) Extract R_ℓ via selective symex on each region
3) Read algebraic laws from R_ℓ (exact, from SMT-LIB operations)
4) Use laws to refine sequencing analysis—design better sentinel expressions
5) Get more precise HTH boundaries
6) Extract more precise R_ℓ
7) Repeat until fixpoint

Convergence. This process terminates because: (1) the grammar Γ is finite, so there are finitely many operations to analyze; (2) algebraic law knowledge is monotonic—we only learn new laws, never unlearn; (3) HTH precision increases or stays the same with each iteration. The fixpoint is reached when no new algebraic knowledge yields more precise boundaries.

Three-way co-refinement. The full technique involves co-refinement across three dimensions:

1) **Configuration refinement (π):** Learning what state is program-relevant by observing causal influence of sentinels
2) **Region refinement (HTH):** Learning computation boundaries using sequencing-sensitive expressions, identified by their non-commutative semantics
3) **Semantic refinement (R_ℓ):** Extracting relational transformers via selective symex, which reveals algebraic properties that inform sequencing detection

These three refine together: better $\pi \to$ better HTH detection $\to$ better $R_\ell \to$ better algebraic knowledge $\to$ better HTH $\to \ldots$ until all three stabilize together with O^*.

E. Soundness and Completeness of the Technique

This gives us a representation of individual paths that is as sound as either our symex engine and/or our formal semantics, to the extent we depend on either of those oracles and/or both. It is complete as both of those oracles as well. Since it's lexical scope, we don't have to do any scope inference. [That's the whole technique.]

V. Proof

This section proves a conditional result: given sound and complete oracles for grammar coverage, control-flow branching, and value transformation, the reconstructed semantics G' simulates $H_\mathcal{I}$ within the constrained semantic domain defined in the Scope section.

A. Oracle Decomposition

We decompose the reconstruction's correctness into three oracle dependencies:

1) **Grammar oracle** (Γ): Provides full syntactic coverage of the language fragment. This may be a known grammar or one inferred via grammar-mining techniques. The grammar oracle delimits the syntax space but is not a source of semantic uncertainty.
2) **Branching oracle** (O): The critical completeness dependency. We require *local completeness* over branching behavior relevant to each reconstructed HTH region. Selective symbolic execution serves as a branching discriminator, ensuring all feasible successor regions for a given syntactic construct are discovered.
3) **Value-transformation oracle**: Provides relational transformers $R_\ell(x, x')$ for basic blocks. May be instantiated by symbolic execution, relational symbolic execution, or trusted formal ISA semantics. This oracle primarily affects soundness: incorrect transformers yield unusable semantics, while partial transformers under-approximate behavior.

B. Setup

Fix a host machine H, implementation $\mathcal{I}$, and grammar Γ. Let:

- $\mathcal{C}$ be a covering set of programs that exercises every production in Γ
- $\mathcal{T}_0 = \{\tau_P \mid P \in \mathcal{C}\}$ be the initial traces from executing $\mathcal{C}$ on $H_\mathcal{I}$
- O be a branching oracle satisfying the completeness axiom (Section III-A)
- $O^*(\mathcal{T}_0)$ be the fixpoint of applying O to traces until no new traces are discovered and π has stabilized

Labeling HTH regions. Each template $P \in \mathcal{C}$ is derived from some production $\gamma \in \Gamma$. The trace region between the computation of hole h_i and hole h_j—where h_i and h_j are consecutive in that trace's evaluation order—constitutes an HTH region, and we assign it the label $\ell = (\gamma, h_i, h_j)$. For traces generated by exploring branches, different control flow paths may reach different holes, yielding different labels. The mechanism for detecting these boundaries—using sentinel expressions refined via algebraic laws—is described in Section IV-D.[1]

For each trace $\tau \in O^*(\mathcal{T}_0)$, the value-transformation oracle produces relational summaries $R_\ell(x, x') := \text{Guard}_\ell(x) \wedge \text{Update}_\ell(x, x')$ for each HTH region labeled ℓ within τ.

Define $G' := \langle X, L, \{R_\ell\}_{\ell \in L}, s_0 \rangle$ where:

- X is the learned configuration space (the image of π)
- L is the set of HTH labels (see Notation)
- $s_0 = \pi(\sigma_0)$ for some initial $H_\mathcal{I}$ state σ_0

[1] In lazy languages, demand patterns determine which holes are evaluated and when, introducing completeness concerns that sentinels and symbolic execution cannot address—one would need to instrument demand propagation itself. This is why we restrict to eager evaluation.

C. Main Theorem

Theorem (Conditional Simulation). If:

1) $\mathcal{C}$ covers all productions in Γ (grammar completeness)
2) O satisfies the completeness axiom: $\forall s \in \text{BStates}(\mathcal{I}).\ \forall o \in \text{Alt}(s).\ \exists c \in O(s).\ \text{ReplayApply}(s, c) \Downarrow o$
3) The value-transformation oracle is sound (produces valid R_ℓ for each HTH region)

then $G' \preceq H_{\mathcal{I}}$.

Proof. We must show that every behavior of $H_{\mathcal{I}}$ (within the constrained scope) has a corresponding behavior in G'.

Soundness. Each R_ℓ in G' is computed as the relational transformer associated with an observed HTH region from some trace $\tau \in O^*(\mathcal{T}_0)$. By hypothesis (3), each R_ℓ correctly characterizes the state transformation along that region. Since τ is a valid trace of $H_{\mathcal{I}}$, the behavior captured by R_ℓ is a valid $H_{\mathcal{I}}$ behavior. Therefore, every step of G' corresponds to a valid step of $H_{\mathcal{I}}$ (projected through π).

Completeness. Suppose for contradiction that some reachable $H_{\mathcal{I}}$ behavior is not captured in G'. Then there exists a state $\sigma \in \text{Reach}(H_{\mathcal{I}})$ and a transition from σ that is not represented by any R_ℓ.

Case 1: If σ lies along some trace $\tau \in O^*(\mathcal{T}_0)$, then the HTH region containing σ was processed by the value-transformation oracle, yielding some R_ℓ—contradiction.

Case 2: If σ is not on any trace in $O^*(\mathcal{T}_0)$, then consider the path from the initial state to σ. At some point, this path must diverge from all traces in $O^*(\mathcal{T}_0)$. Let s be the first divergence point and o the branch outcome leading toward σ. Since σ is reachable, $o \in \text{Alt}(s)$. By hypothesis (2), there exists $c \in O(s)$ such that $\text{ReplayApply}(s, c) \Downarrow o$, meaning the oracle would generate a trace covering this branch. But $O^*(\mathcal{T}_0)$ is the fixpoint of applying O, so this trace would be in $O^*(\mathcal{T}_0)$—contradiction.

Therefore $G' \preceq H_{\mathcal{I}}$. □

D. Remarks

On oracle instantiation. The branching oracle is not hypothetical. Selective symbolic execution engines (S2E, angr) and formal ISA semantics (Sail for ARM/RISC-V) provide exactly this capability. The value-transformation oracle is similarly instantiated by symbolic execution's production of path conditions and symbolic stores.

On the conditional nature. The nontrivial content of this result lies in making the reconstruction mechanism explicit, not in claiming stronger guarantees than the oracles permit. Soundness holds insofar as the value-transformation oracle is sound; completeness holds insofar as the branching oracle enumerates all feasible control-flow alternatives.

On tractability. This result holds for the constrained scope defined earlier: structured control flow, lexical scope, eager evaluation, no concurrency, no nondeterminism. Within this scope, memory aliasing is bounded, control flow is analyzable at basic-block level, and symbolic execution is feasible when used selectively and locally.

On co-refinement convergence. The fixpoint $O^*(\mathcal{T}_0)$ involves co-refinement: π (the projection to configuration space X) and the trace set are refined together. A natural concern is whether this process can deadlock—could discovering a branch require *both* a π-update and an O-update simultaneously, with neither able to proceed first?

The answer is no, because O operates at the Σ level (full host state), not the X level (projected configuration). Symbolic execution sees all concrete state, not just the projection. When O discovers a branch whose condition involves state S:

- If $S \in X$ already: the branch is expressible in the current π, proceed normally
- If $S \notin X$: O still *discovers* the branch (symex sees Σ), but the branch condition references state we are not yet tracking. We refine π to include S.

Crucially, O is never blocked waiting for π—it sees everything at the host level. The π refinement determines what we *include in* G', not what O can discover. Since X grows monotonically (we only add state, never remove), branching is finite, and state is finitely representable within our scope constraints, the co-refinement converges.

Acknowledgments

References

[1] T. A. Henzinger, R. Majumdar, and J.-F. Raskin, "A classification of symbolic transition systems," in *Proceedings of STACS 2000*, ser. Lecture Notes in Computer Science, vol. 1770. Springer, 2000, pp. 13–34.

[2] R. J. van Glabbeek, "The linear time – branching time spectrum," in *CONCUR '90: Theories of Concurrency: Unification and Extension*, ser. Lecture Notes in Computer Science, vol. 458. Springer, 1990, pp. 278–297.

[3] ——, "The linear time – branching time spectrum II: The semantics of sequential systems with silent moves," in *CONCUR '93: 4th International Conference on Concurrency Theory*, ser. Lecture Notes in Computer Science, vol. 715. Springer, 1993, pp. 66–81.

[4] G. Roşu and T. F. Şerbănuţă, "An overview of the K semantic framework," *Journal of Logic and Algebraic Programming*, vol. 79, no. 6, pp. 397–434, 2010.

[5] J. Craaijo, F. Verbeek, and B. Ravindran, "libLISA: Instruction discovery and analysis on x86-64," *Proceedings of the ACM on Programming Languages*, vol. 8, no. OOPSLA2, pp. 1–29, 2024.

[6] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *IEEE Transactions on Software Engineering*, vol. 27, no. 2, pp. 99–123, 2001.

APPENDIX
HTH BLOCK VISUALIZATIONS

This appendix provides detailed visualizations of Hole-to-Hole (HTH) blocks and worked examples demonstrating the semantics extraction process.

[DRAFT] Unified HTH Visualization - Arithmetic

h_1 left h_2 right h_0 target

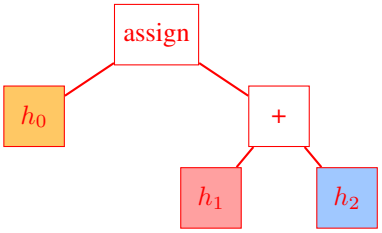
Code:

```
z = a + b
```

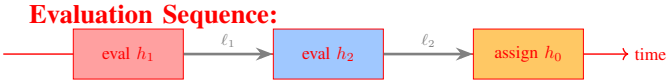
Grammar:

```
stmt ::= NAME '=' expr
expr ::= expr '+' expr
```

AST Tree:



Evaluation Sequence:



Relational Transformers:

$$R_{\ell_1} := x' = x$$
$$R_{\ell_2} := x'[h_0] = x[h_1] + x[h_2]$$

[DRAFT] Unified HTH Visualization - Control Flow

h_1	h_2	h_3	h_0
cond	then	else	target

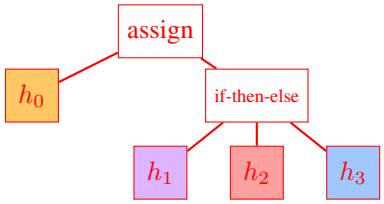
Code:

```
y = if c then t else f
```

Grammar:

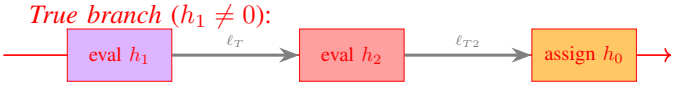
```
stmt ::= NAME '=' expr
expr ::= 'if' expr 'then' expr
         'else' expr
```

AST Tree:

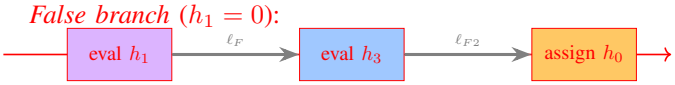


Evaluation Sequences (branching):

True branch ($h_1 \neq 0$):



False branch ($h_1 = 0$):



Relational Transformers:

$$R_{\ell_T}(x, x') := x[h_1] \neq 0 \wedge x' = x \quad \text{(true branch guard)}$$

$$R_{\ell_F}(x, x') := x[h_1] = 0 \wedge x' = x \quad \text{(false branch guard)}$$

$$R_{\ell_{T2}}(x, x') := \text{true} \wedge x'[h_0] = x[h_2] \quad \text{(assign from then)}$$

$$R_{\ell_{F2}}(x, x') := \text{true} \wedge x'[h_0] = x[h_3] \quad \text{(assign from else)}$$

[DRAFT] Hole-to-Hole (HTH) Blocks

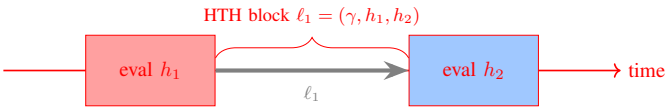
What is an HTH Block?

A **hole-to-hole (HTH) block** is a maximal straight-line region of execution between two consecutive holes in evaluation order. Within an HTH block, there is no branching—it is a *basic block* at the semantic level.

The key insight is that programming language semantics can be decomposed into these atomic units. Each HTH block represents a single "step" in the abstract semantics—the minimal unit of semantic behavior between syntactically-identifiable points.

Visualizing a Single HTH Block

Consider the expression `a + b`. Between evaluating the left operand (h_1) and the right operand (h_2), there is exactly one HTH block:



Key insight: The HTH block is the region *between* hole evaluations—the arrow, not the boxes. The colored boxes represent hole evaluation points; the arrow represents the HTH block itself.

The HTH Label

Each HTH block is identified by a label $\ell = (\gamma, h_i, h_j)$ where:

Comp.	Name	Meaning
γ	Production	Grammar rule
h_i	Source	Hole just evaluated
h_j	Dest.	Hole evaluated next

This triple uniquely identifies each HTH block in the language. Different productions have different holes; different evaluation orders within a production yield different HTH blocks.

Why HTH Blocks Matter

HTH blocks are the *atoms* of semantic extraction:

- **No internal branching:** Within an HTH block, execution is deterministic. This makes symbolic analysis tractable.
- **Syntactically anchored:** The boundaries are defined by grammar holes, not arbitrary program points. This connects runtime behavior to syntax.
- **Composable:** The full semantics G' is built by composing HTH blocks according to the grammar structure.

The entire technique rests on this decomposition: we extract the relational transformer R_ℓ for each HTH block separately, then compose them to get the full language semantics.

[DRAFT] The Branching Oracle

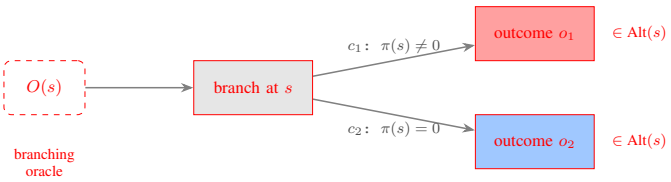
Purpose

The **branching oracle** O discovers all feasible control-flow alternatives at branch points. It answers the question: "Given a state s where execution could go multiple ways, what are all the possible outcomes, and how do we reach each one?"

This oracle is essential for *completeness*—without it, we would only observe the branches that happen to execute in our test cases, missing potentially important semantic behaviors.

How It Works

Given a state s at a branch point, $O(s)$ produces constraints over $\pi(s)$ (the projected configuration) that achieve each alternative outcome:



What It Recovers

- **BStates**$(\mathcal{I})$: The set of all reachable states where branching occurs—states with multiple possible successors.
- **Alt**(s): For each branch state s, all feasible outcomes that can be reached by varying the configuration $\pi(s)$.
- **Constraints**: For each outcome $o \in \text{Alt}(s)$, a constraint $c \in O(s)$ such that replaying from s with $\pi(s)$ modified according to c produces outcome o.

Formally: $\forall o \in \text{Alt}(s).\ \exists c \in O(s).\ \text{ReplayApply}(s, c) \Downarrow o$

Instantiation

The branching oracle is not hypothetical—it is instantiated by real tools:

- **Symbolic execution engines** (S2E, angr, KLEE): These explore program paths by treating inputs as symbolic values. When they encounter a branch, they generate constraints for each direction.
- **Formal ISA semantics** (Sail for ARM/RISC-V): These provide precise models of instruction behavior, including branch conditions.

The key property is that the oracle operates at the *host level* (Σ), not the projected level (X). This means it can discover branches even before we know which state is relevant to the language semantics.

[DRAFT] The Value Transformation Oracle

Purpose

The **value transformation oracle** recovers the relational transformer $R_\ell(x, x')$ for each HTH block. It answers the question: "What happens to the configuration as we traverse this HTH block?"

This oracle is essential for *soundness*—the extracted semantics are only as correct as the relational transformers we compute.

The Relational Transformer

Each HTH block has an associated relational transformer $R_\ell(x, x')$ that relates the input configuration x to the output configuration x'. This decomposes into two parts:

Component	Form	Meaning
$\text{Guard}_\ell(x)$	pred. on x	When taken?
$\text{Update}_\ell(x, x')$	rel. on x, x'	State change?
$R_\ell(x, x')$	Guard $\wedge$ Update	Full summary

Example: Sequencing

For the HTH block ℓ_1 (sequencing from h_1 to h_2 in addition `a + b`):

$$R_{\ell_1}(x, x') := \underbrace{\text{true}}_{\text{Guard}} \wedge \underbrace{x' = x}_{\text{Update}}$$

- **Guard = true**: This block is always taken (no condition guards sequencing).
- **Update =** $x' = x$: No state change—we just move from evaluating h_1 to evaluating h_2.

Example: Computation

For the HTH block that computes the addition result and stores it:

$$R_{\ell_2}(x, x') := \text{true} \wedge x'[\,h_0\,] = x[\,h_1\,] + x[\,h_2\,]$$

- **Guard = true**: Always taken after both operands evaluated.
- **Update**: The target hole h_0 receives the sum of the values at h_1 and h_2.

Instantiation

The value transformation oracle is instantiated by **selective symbolic execution**:

1) Given an HTH region in a concrete trace, snapshot the memory state at the region's start.
2) Make the hole values symbolic (replace concrete sentinels with symbolic variables).
3) Symbolically execute just that region.
4) The resulting path condition becomes the Guard; the symbolic store becomes the Update.

Because HTH blocks have no internal branching, this symbolic execution is tractable—we follow exactly one path.

[DRAFT] Covering Sets and Extraction

What is a Covering Set?

A **covering set** $\mathcal{C}$ is a minimal collection of program templates that exercises every production in the grammar Γ. Each template $P \in \mathcal{C}$ contains **sentinel holes**—placeholders filled with distinguishable values κ_h.

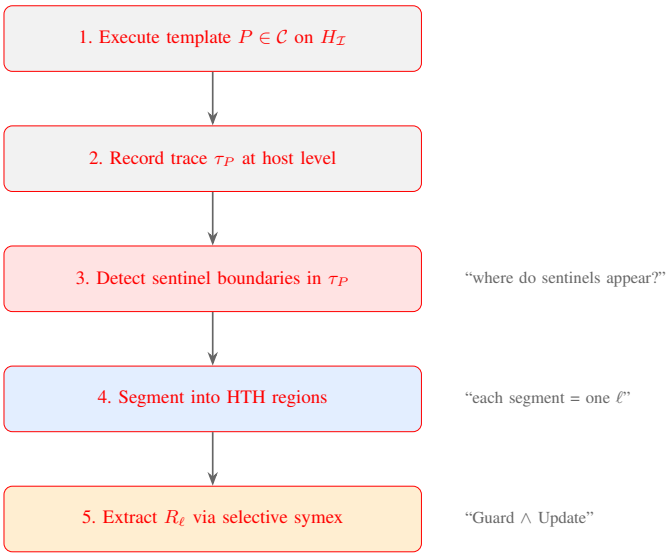
Example: To cover the production `expr ::= expr '+' expr`, we need a template:

$$\boxed{\kappa_1 \; + \; \kappa_2}$$

where κ_1 and κ_2 are sentinel values (e.g., distinctive integers like `0xDEAD` and `0xBEEF`).

The Extraction Process

Given a covering set $\mathcal{C}$, we extract HTH blocks and their relational transformers:



Sentinel Detection

Sentinels mark hole boundaries. When we see κ_1 computed in the trace, we know h_1 was just evaluated. The region *between* sentinel appearances is the HTH block.

[DRAFT] Covering Sets and Extraction (continued)

Selective Symbolic Execution

Once we identify an HTH region in the trace, we invoke **selective symbolic execution** on just that region:

Input	Concrete trace segment, memory snapshot at region start
Process	Symbolically execute the region with symbolic inputs for hole values
Output	$R_\ell(x, x') = \text{Guard}_\ell(x) \wedge \text{Update}_\ell(x, x')$

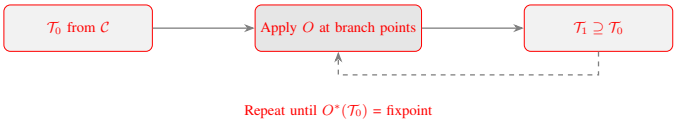
The symbolic execution engine produces:

- **Guard:** The path condition accumulated during execution. For HTH blocks (no branching), this is typically true.
- **Update:** The symbolic store showing how each output location relates to input locations. This captures the actual computation performed.

Achieving Completeness

The covering set $\mathcal{C}$ gives us *one* trace per production. But productions with branching (like `if-then-else`) have multiple HTH blocks depending on which branch executes.

The **branching oracle** O completes the picture:



At fixpoint $O^*(\mathcal{T}_0)$:

- Every reachable HTH block has at least one exemplar trace
- Every R_ℓ has been extracted via selective symbolic execution
- The reconstructed semantics $G' = \langle X, L, \{R_\ell\}_{\ell \in L}, s_0 \rangle$ is complete

---

Summary: From Templates to Semantics

1) **Cover** the grammar with sentinel-filled templates $\mathcal{C}$
2) **Execute** each template, recording traces at the host level
3) **Detect** HTH boundaries via sentinel appearances
4) **Extract** R_ℓ for each HTH region via selective symbolic execution
5) **Iterate** using the branching oracle until all reachable HTH blocks are covered
6) **Compose** the R_ℓ into the complete semantics G'

[DRAFT] Worked Example: GCD

The Algorithm

```
def gcd(a, b):
1:   while b != 0:
2:     t = b
3:     b = a % b
4:     a = t
5:   return a
```

Color Key by HTH Block Type

Comp.	Assign	Ctrl	Seq
ℓ_1, ℓ_6	$\ell_4, \ell_7, \ell_9, \ell_{11}$	ℓ_2, ℓ_3	$\ell_5, \ell_8, \ell_{10}$

HTH Blocks in This Program

Each production's operational semantics defines jumps between holes. Every computation, every assignment, and every control flow jump is its own HTH block.

#	Label	Description
1	ℓ_1	compute : b != 0
2	ℓ_2	control : line 1→2, enter body
3	ℓ_3	control : line 1→5, exit loop
4	ℓ_4	assign : t = b
5	ℓ_5	sequence : line 2→3
6	ℓ_6	compute : a % b
7	ℓ_7	assign : b = result
8	ℓ_8	sequence : line 3→4
9	ℓ_9	assign : a = t
10	ℓ_{10}	sequence : line 4→1
11	ℓ_{11}	assign : return a

Total: 11 unique HTH blocks. Note that ℓ_2 and ℓ_3 are alternatives (branching)—only one is taken per evaluation of the condition.

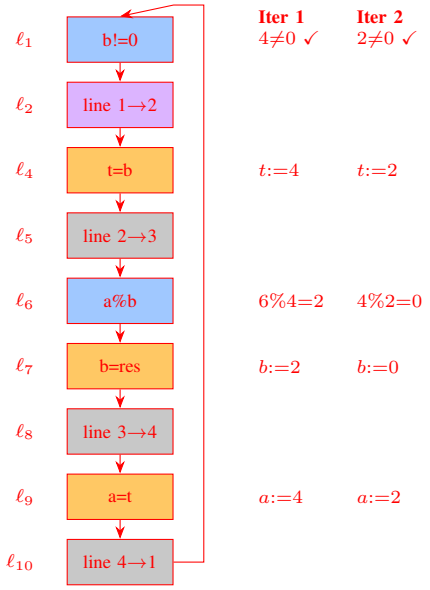
[DRAFT] GCD Execution Trace: gcd(6, 4)

Initial State

$a = 6, \quad b = 4$

Loop Body Trace

Each iteration traverses these 9 HTH blocks (values shown for both iterations):

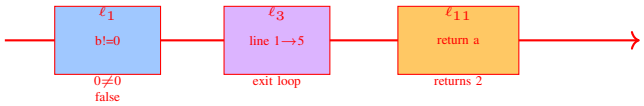


	Iter 1	**Iter 2**
	$4 \neq 0$ ✓	$2 \neq 0$ ✓
ℓ_4	$t := 4$	$t := 2$
ℓ_6	$6 \% 4 = 2$	$4 \% 2 = 0$
ℓ_7	$b := 2$	$b := 0$
ℓ_9	$a := 4$	$a := 2$

State after Iter 1: $a = 4, b = 2, t = 4$ **After Iter 2:** $a = 2, b = 0, t = 2$

Exit

3 HTH blocks for the exit path:

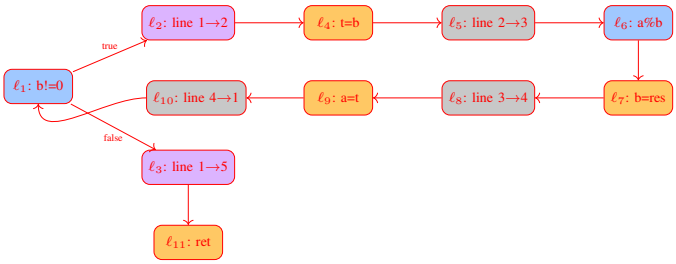


Result: $\gcd(6, 4) = 2$

[DRAFT] GCD: The Complete HTH Graph

All 11 HTH Blocks

The HTH blocks form a graph. Loops create cycles; branches create forks.



Relational Transformers (all 11)

$$
\begin{aligned}
R_{\ell_1} &:= x'[\text{cmp}] = (x[\text{b}] \neq 0) \\
R_{\ell_2} &:= x[\text{cmp}] \wedge x' = x \\
R_{\ell_3} &:= \neg x[\text{cmp}] \wedge x' = x \\
R_{\ell_4} &:= x'[\text{t}] = x[\text{b}] \\
R_{\ell_5} &:= x' = x \\
R_{\ell_6} &:= x'[\text{tmp}] = x[\text{a}] \,\%\, x[\text{b}] \\
R_{\ell_7} &:= x'[\text{b}] = x[\text{tmp}] \\
R_{\ell_8} &:= x' = x \\
R_{\ell_9} &:= x'[\text{a}] = x[\text{t}] \\
R_{\ell_{10}} &:= x' = x \\
R_{\ell_{11}} &:= x'[\text{ret}] = x[\text{a}]
\end{aligned}
$$

Key Observations

- **Same HTH blocks, different values:** Iterations 1 and 2 traverse the *same* 9 HTH blocks $(\ell_1, \ell_2, \ell_4, \ell_5, \ell_6, \ell_7, \ell_8, \ell_9, \ell_{10})$ but with different concrete values. The HTH blocks are the *schema*; the values are the *instantiation*.
- **Every step is an HTH block:** Computations (ℓ_1, ℓ_6), assignments (ℓ_4, ℓ_7, ℓ_9), control flow jumps (ℓ_2, ℓ_3, ℓ_{10}), and sequencing (ℓ_5, ℓ_8) are *all* HTH blocks. There is no distinction—evaluation order *is* the sequence of HTH blocks.
- **Branching creates alternatives:** ℓ_2 (enter body) and ℓ_3 (exit loop) are alternatives at the same branch point. Their guards are mutually exclusive: $x[\text{cmp}] = \text{true}$ vs $x[\text{cmp}] = \text{false}$.
- **The loop is a cycle:** ℓ_{10} (loop back) connects the end of the body back to ℓ_1 (condition), creating a cycle in the HTH graph.
- **Composition gives semantics:** The full GCD semantics is the composition of these 11 R_ℓ along all paths through the HTH graph.

[DRAFT] Synthesis: HTH-Based Semantics Extraction

This section integrates all visualizations to show how we extract executable semantics from interpreter traces.

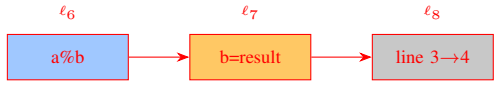
Core Concept: HTH Blocks

A **Hole-to-Hole (HTH) block** is an atomic unit of semantic behavior—the smallest step the interpreter takes between observable points. Every program execution is a sequence of HTH blocks.

The four types of HTH blocks:

Compute	Assign	Control	Sequence
evaluates expressions	stores results	branches on condition	advances to next stmt

Example: For `gcd(a, b)`, the statement `b = a % b` decomposes into:



From Traces to Relational Transformers

Each HTH block ℓ has an associated **relational transformer** $R_\ell(x, x')$ that relates the input configuration x to the output configuration x':

$$R_\ell(x, x') \;:=\; \text{Guard}_\ell(x) \;\wedge\; \text{Update}_\ell(x, x')$$

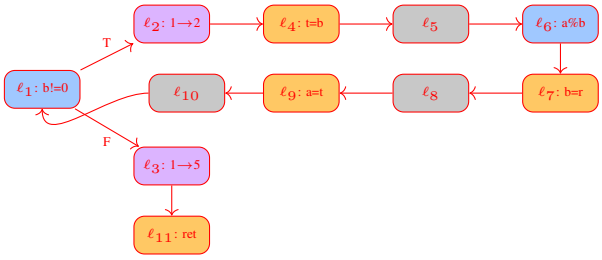
- **Guard**: Under what conditions is this block taken?
- **Update**: How does the state change?

Complete Example: GCD

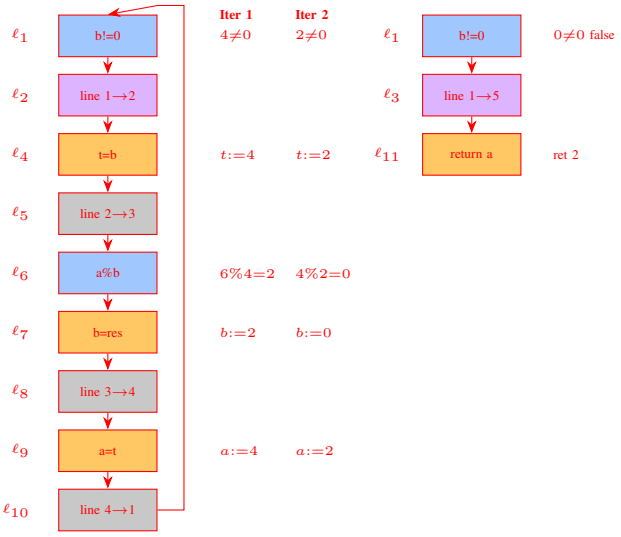
The algorithm:

```
def gcd(a, b):
1:   while b != 0:
2:     t = b
3:     b = a % b
4:     a = t
5:   return a
```

The HTH graph shows all 11 blocks and their connections. Loops create cycles; branches create forks:



Execution trace for gcd(6, 4): Two loop iterations, then exit.



Key insight: Both iterations traverse the *same* 9 HTH blocks with *different* values. The blocks are the **schema**; the values are the **instantiation**. This is why we can extract a single set of relational transformers that work for all inputs.

The Extracted Semantics

The relational transformers, composed along paths through the HTH graph, give us the complete operational semantics:

$$\begin{aligned}
R_{\ell_1} &:= x'[\text{cmp}] = (x[\text{b}] \neq 0) && \text{(compute condition)}\\
R_{\ell_2} &:= x[\text{cmp}] \wedge x' = x && \text{(enter body if true)}\\
R_{\ell_3} &:= \neg x[\text{cmp}] \wedge x' = x && \text{(exit loop if false)}\\
R_{\ell_4} &:= x'[\text{t}] = x[\text{b}] && \text{(save b to t)}\\
R_{\ell_6} &:= x'[\text{r}] = x[\text{a}] \,\%\, x[\text{b}] && \text{(compute remainder)}\\
R_{\ell_7} &:= x'[\text{b}] = x[\text{r}] && \text{(update b)}\\
R_{\ell_9} &:= x'[\text{a}] = x[\text{t}] && \text{(update a)}\\
R_{\ell_{11}} &:= x'[\text{ret}] = x[\text{a}] && \text{(return result)}
\end{aligned}$$

(Sequence blocks $\ell_5, \ell_8, \ell_{10}$ have $R_\ell := x' = x$—they advance control without changing state.)

From Programs to Languages

GCD is a worked example, but our goal is extracting **language semantics**, not program semantics. The covering set $\mathcal{C}$ contains templates that exercise every grammar production—not every program.

Covering sets and control flow. For a production like `if-then-else`, the covering set includes templates where the condition hole is filled with:

- **Low-cardinality types**: All values exhaustively (e.g., `True`, `False`)
- **High-frequency literals**: Common values like `0`, `1`, `-1`, `None`, `""`, `[]`

These first-round traces already exercise both branches for simple conditions. The branching oracle handles computed/-complex conditions.

Truthiness and method dispatch. In Python, `if x:` doesn't compare `x` to `True`—it calls `x.__bool__()`. How do we capture this?

Because we trace at the *interpreter level*, method dispatch is visible. The HTH decomposition for `if cond:` is not a single "evaluate and branch" block, but a sequence:

The `__bool__` call is just more HTH blocks. We see it in the trace because function entry/exit are observable points. The covering set for `__bool__`'s return type is trivially $\{\texttt{True}, \texttt{False}\}$, giving us branch coverage automatically.

Method dispatch isn't special—it's HTH blocks all the way down.

Summary: The Extraction Pipeline

1) **Execute** the interpreter on test inputs, recording traces
2) **Detect** HTH block boundaries via sentinel values
3) **Segment** traces into HTH regions
4) **Extract** R_ℓ for each block via selective symbolic execution
5) **Compose** the R_ℓ along the HTH graph to get full semantics

The result is a **labeled transition system** that is bisimilar to the original interpreter—same behavior, but now we have explicit, composable semantic rules.

[DRAFT] Covering Set Examples (Python)

A **covering set** $\mathcal{C}$ contains template programs that exercise every grammar production. Each template has **sentinel holes** filled with values chosen to maximize behavioral coverage. Below are concrete examples for several Python productions.

---

1. If-Then-Else

Production: `stmt ::= 'if' expr ':' suite ('else' ':' suite)?`
Holes: condition expression, then-body, else-body
Covering set templates:

```
if True:   x = 1      # cond = True
else:      x = 2

if False:  x = 1      # cond = False
else:      x = 2

if 0:      x = 1      # cond = 0 (falsy int)
else:      x = 2

if 1:      x = 1      # cond = 1 (truthy int)
else:      x = 2

if []:     x = 1      # cond = [] (falsy list)
else:      x = 2

if None:   x = 1      # cond = None
else:      x = 2
```

These exercise both branches and reveal truthiness for multiple types.

---

2. While Loop

Production: `stmt ::= 'while' expr ':' suite`
Holes: condition expression, loop body
Covering set templates:

```
while False: x = 1    # cond false: skip body

while True:           # cond true: enter body
    x = 1
    break
```

Just two templates suffice! We see all HTH transitions: condition eval, true/false guards, body entry, exit. The "loop back" edge is structural—we don't need multiple iterations to learn it.

---

3. Binary Arithmetic (+)

Production: `expr ::= expr '+' expr`
Holes: left operand, right operand
Covering set templates:

```
x = 1 + 2             # int addition
x = "a" + "b"         # string concat
x = [1] + [2]         # list concat
```

*Key insight: these produce **different HTH traces**! Each type dispatches to its own `__add__`. Concrete execution reveals this polymorphism directly—no type system modeling required.*

---

4. Indexing

Production: `expr ::= expr '[' expr ']'`
Holes: container, index
Covering set templates:

```
x = [10, 20, 30][0]    # list, first
x = [10, 20, 30][-1]   # list, negative
x = {"a": 1}["a"]      # dict
x = "hello"[0]         # string
```

The container hole is filled from `expr`'s covering set, which includes various types. This reveals type-specific `__getitem__` dispatch.

---

5. Function Definition and Call

Productions:

- `stmt ::= 'def' NAME '(' params ')' ':' suite`
- `expr ::= NAME '(' args ')'`

Covering set templates:

```
def f(): return 1     # 0-arity
x = f()

def g(a): return a    # 1-arity
x = g(42)

def h(a, b):          # 2-arity
    return a + b
x = h(1, 2)

def k(a=10): return a # default arg
x = k()
x = k(20)
```

Exercises function entry/exit, argument passing, defaults.

---

6. Attribute Access

Production: `expr ::= expr '.' NAME`
Holes: object, attribute name
Covering set templates:

```
class C:
    x = 10
    def m(self): return 1

obj = C()
y = obj.x               # attr read
z = obj.m()             # method call
obj.x = 20              # attr write
```

Reveals `__getattr__`/`__setattr__` and method binding.

---

7. For Loop

Production: `stmt ::= 'for' NAME 'in' expr ':' suite`
Holes: loop variable, iterable, body
Covering set templates:

```
for x in []: y = x       # empty (0 iters)
for x in [1]: y = x      # single element
for x in [1,2]: y = x    # multiple
for x in "ab": y = x     # string iter
```

Exercises iterator protocol (`__iter__`, `__next__`).

---

8. Try-Except

Production: `stmt ::= 'try' ':' suite 'except' ... ':' suite`
Covering set templates:

```
try:                # no exception
    x = 1
except:
    x = 2

try:                # exception raised
    x = 1 / 0
except:
    x = 2

try:                # typed exception
    x = int("bad")
except ValueError:
    x = 0
```

Exercises normal completion, exception raise, exception matching.

Summary

Each production's covering set is designed to:

1) Exercise all control flow paths (branches, iterations, exceptions)
2) Include boundary values (0, 1, -1, empty, single-element)
3) Reveal type-dependent dispatch (`__add__`, `__bool__`, etc.)
4) Cover literals that trigger special behavior

The union of all production covering sets forms $\mathcal{C}$. Executing these generates traces that, segmented into HTH blocks, yield the complete language semantics.

Why concrete execution first? Two reasons:

1) **Boundary detection**: Sentinel values in traces let us *find* HTH block boundaries empirically. The grammar tells us syntactic structure, but not evaluation order or implicit dispatches.
2) **Type dispatch discovery**: Different types produce different traces for the same production (e.g., + on ints vs strings). Concrete execution reveals this polymorphism without modeling the type system.

After boundary detection, we use selective symbolic execution to generalize each HTH segment into its relational transformer R_ℓ.

Algorithm 1: Syntax-Only Covering Set Generation

Reference: tree-sitter-python `grammar.js`

Input: A grammar production P

For each production P:

1) **Identify holes**—the field positions referencing other rules.
   Example for `if_statement`:

```
if_statement: $ => seq(
  'if',
  field('condition', $.expression),
  ':',
  field('consequence', $.block),
  optional(field('alternative', $.
      else_clause)),
)
```

Field	Sort
`condition`	`expression`
`consequence`	`block`
`alternative`	`else_clause` (optional)

2) **Generate minimal fillers** for each sort:

Sort	Minimal Fillers
`expression`	`0`, `1`, `""`, `"x"`, `True`, `False`, `None`, `[]`, `[1]`
`block`	`pass`, `x = 1`
`identifier`	`f` (fresh)
`integer`	`0`, `1`, `-1`
`string`	`""`, `"x"`
`list`	`[]`, `[1]`, `[1, 2]`

3) **Combine** to create covering templates:

```
if 0: pass            # falsy integer
if 1: pass            # truthy integer
if "": pass           # falsy string
if "x": pass          # truthy string
if []: pass           # falsy list
if [1]: pass          # truthy list
if None: pass         # None (falsy)
if True: pass         # bool True
if False: pass        # bool False
```

Output: Set of syntactically valid programs covering production P.

Algorithm 2: Identifier Resolution Discovery

Reference: tree-sitter-python `grammar.js`

Input: Grammar G, token `identifier`

Step 1: Collect identifier-containing productions

Production	Identifier Field	Role
`function_definition`	`name`	binder
`class_definition`	`name`	binder
`assignment`	`left`	binder
`primary_expression`	(direct)	reference
`call`	`function`	reference
`attribute`	`attribute`	reference
`global_statement`	(direct)	scope mod
`nonlocal_statement`	(direct)	scope mod

Step 2: Generate single-identifier test programs

Using fresh identifier `f`:

```
f                       # primary_expression (
    reference)
f()                     # call (reference)
f = 1                   # assignment (binder)
def f(): pass           # function_definition (
    binder)
class f: pass           # class_definition (
    binder)
```

Step 3: Execute and record outcomes

Test Program	Outcome
`f`	`NameError`
`f()`	`NameError`
`f = 1`	OK
`def f(): pass`	OK
`class f: pass`	OK
`f = 1; f`	OK
`def f(): pass; f()`	OK
`f = 1; f()`	`TypeError`
`def f(): pass; f = 1; f()`	`TypeError`

Step 4: Infer structure

- **Binders**: `function_definition`, `class_definition`, `assignment`
- **References**: `identifier` (in `primary_expression`), `call`
- **Ordering constraint**: binder $\prec$ reference
- **Namespace**: unified (function/class/variable share namespace, can shadow)
- **Type dispatch**: `call` requires callable value

Output:

- Binder/reference classification per production
- Ordering constraints (binding must precede reference)
- Namespace structure (unified vs. separate)

Key tree-sitter-python Sorts

Sort	Description
`identifier`	Name token
`expression`	Any expression
`primary_expression`	Atomic: id, literals, call, subscript, attr
`block`	Indented body
`call`	Function/method call
`subscript`	Indexing `x[i]`
`attribute`	Attribute access `x.y`
`assignment`	Binding `x = e`
`function_definition`	`def f(): block`
`class_definition`	`class C: block`
`if_statement`	`if cond: block`
`while_statement`	`while cond: block`
`for_statement`	`for x in e: block`
`integer`	Integer literal
`string`	String literal
`list`	List literal `[...]`
`true`/`false`/`none`	Boolean/None literals