Compositional Symbolic Execution Semantics

Erik Voogd[a,*], Åsmund Aqissiaq Arild Kløvstad[a], Einar Broch Johnsen[a], Andrzej Wąsowski[b]

[a] *University of Oslo, Oslo, Norway*
[b] *IT University of Copenhagen, Copenhagen, Denmark*

Abstract

Symbolic execution is a program analysis technique to systematically explore all possible paths through a program. The technique can be formally explained by means of small-step transition systems that update symbolic states and compute a precondition corresponding to the taken execution path. In stateful transition systems behavior may depend on previous transitions, which complicates compositional reasoning about programs. To enable compositonal reasoning this paper defines a denotational semantics for symbolic execution. The proposed semantics views a program as a set of traces, each of which has a corresponding substitution — the composition of all its assignments — and a corresponding *path condition* — the conjunction of all its Boolean tests under appropriate substitution. We prove correspondence between the symbolic denotational semantics and a concrete semantics. We argue that the symbolic denotational semantics is a very natural framework to reason about symbolic execution, and use it to prove that symbolic execution computes (weakest) preconditions. We provide mechanizations in Coq for the main results.

Keywords: Symbolic Execution, Program Analysis, Denotational Semantics

1. Introduction

Major successes in program analysis, particularly for debugging, test case generation, and verification, have been achieved by *symbolic execution* [5, 15, 7, 6, 8, 9, 17, 21], a powerful simulation technique in which symbolic states represent a wide range of concrete program states. A complementary line of research investigates the meta-properties of symbolic execution [5, 35, 36]; in particular, modeling symbolic execution as a formal system and proving it correct with respect to a concrete operational semantics.

With symbolic execution, program states associate program variables to symbolic expressions rather than to concrete values. Assignments in the program are understood as updating the symbolic state through substitutions σ. Since symbolic states are abstract, no

*Corresponding author

Email addresses: `erikvoogd@live.nl` (Erik Voogd), `aaklovst@ifi.uio.no` (Åsmund Aqissiaq Arild Kløvstad), `einarj@uio.no` (Einar Broch Johnsen), `wasowski@itu.dk` (Andrzej Wąsowski)

choice can be made when encountering control-flow statements guarded by Boolean expressions. Instead, the transition system modeling symbolic execution branches in both possible directions (theoretically using nondeterminism; in practice exploring both branches), and updates its own state by storing the decision, i.e., the Boolean guard or its negation, under substitution. It thus generates the *path condition* φ, which is an aggregation of all decisions made with respect to Boolean control-flow guards. If a program p is symbolically executed and terminates in some symbolic state (σ, φ), then φ is a precondition for p to behave in a way specified by the substitution σ.

Example. Consider the following code snippet, which halves an integer-valued variable if its value is *even*, and multiplies it by three, adding one, if its value is *odd*:

$$\texttt{cz} \overset{\text{def}}{=} \texttt{if (n\%2=0) then \{ n:=n}\div\texttt{2; \} else \{ n:=3*n+1; \}}$$

Three things are monitored in the state triples (p, σ, φ) of a symbolic execution engine: (i) the remaining program p to analyze, (ii) a symbolic substitution σ that composes the assignments it encounters, and (iii) the path condition φ that gathers the Boolean tests encountered. Symbolic execution is usually initiated in the configuration $(\mathrm{id}, \top)$: the identity substitution id along with the path condition $\top$ (true) specifying the entire input space. The symbolic executions of the program cz are the following two:

$$\begin{array}{l}
(\texttt{cz}, \mathrm{id}, \top) \longrightarrow (\texttt{n:=n}\div 2,\ \mathrm{id}, \top \wedge \texttt{n\%2} = \texttt{0}) \longrightarrow (\varepsilon, \{\texttt{n}/\texttt{n}\div 2\}, \top \wedge \texttt{n\%2} = \texttt{0}) \\
(\texttt{cz}, \mathrm{id}, \top) \longrightarrow (\texttt{n:=3*n+1}, \mathrm{id}, \top \wedge \neg\texttt{(n\%2=0)}) \longrightarrow (\varepsilon, \{\texttt{n}/3 * \texttt{n} + 1\}, \top \wedge \neg\texttt{(n\%2=0)})
\end{array}$$

Here, ε is the terminated program, and $\{\texttt{n}/\texttt{n}\div 2\}$ is the substitution that maps n to $\texttt{n}\div 2$ and every other variable to itself. The first step in both executions analyzes the `if` statement and updates the path condition (under substitution, in this case identity) according to the Boolean test and the branch it explores. The second step analyzes the assignment and updates the symbolic substitution accordingly. We write $\longrightarrow^*$ for the reflexive-transitive closure of the relation $\longrightarrow$. The two symbolic executions $(\texttt{cz}, \mathrm{id}, \top) \longrightarrow^* (\varepsilon, \sigma_i, \varphi_i)$ $(i = 1, 2)$ above provide two final configurations (σ_i, φ_i) $(i = 1, 2)$. The final configurations symbolically represent paths through the program that behave exactly like the obtained substitution. Moreover, the path conditions are preconditions for their corresponding substitution.

Substitutions. There are two ways to view symbolic substitutions. First, a substitution σ is a *syntactic* operation that performs substitution of variables. Syntactic substitutions transform postconditions into preconditions. As an example, consider the postcondition $\psi \equiv x > 0$ and let σ substitute x by $x + 1$. This σ represents the assignment $x := x + 1$. The precondition of this assignment with respect to the postcondition ψ is obtained by applying σ to ψ, so $\sigma(x > 0) = x + 1 > 0$. Indeed, if $x + 1 > 0$ and the program executes $x := x + 1$, then, afterwards, $x > 0$.

Second, *semantically*, a substitution σ models a state transformation $[\![\sigma]\!]$ (defined in Definition 4) that evaluates the input state after substitution. With σ as above, a state s with x valued as 20 would be transformed by σ to a state $[\![\sigma]\!](s)$ where x is valued as 21. If symbolic execution of a deterministic program p yields a terminated symbolic configuration (σ, φ), and p is run on a concrete input state s that satisfies φ, then the output is the state $[\![\sigma]\!](s)$.

Verification. Given a postcondition ψ and a terminated symbolic configuration (σ, φ), the formula $\varphi \wedge \sigma(\psi)$ is a precondition of p with respect to ψ, because the path condition φ ensures that program behavior corresponds to $[\![\sigma]\!]$, and the output $[\![\sigma]\!](s)$ satisfies ψ if and only if the input s satisfies $\sigma(\psi)$ (as formalized in Lemma 3), in this case, $x+1 > 0$. *Syntactic* application of σ computes a precondition from a postcondition in a *backward* fashion, and, simultaneously, σ describes a *forward* state transformation given by $[\![\sigma]\!]$.

Symbolic execution is therefore a compelling technique for verification purposes: it provides preconditions $\varphi \wedge \sigma(\psi)$ specifying the inputs that ensure the program takes the path corresponding to the symbolic execution of (σ, φ), and guarantees an output satisfying ψ. Gathering more and more symbolic executions will yield a weaker and weaker precondition. In fact, the (possibly infinite) disjunction of the formulas $\varphi \wedge \sigma(\psi)$, ranging over all final configurations (σ, φ), gives precisely the *weakest* precondition—assuming the program is deterministic. Following Hoare's work on correctness reasoning [22], Dijkstra introduced *weakest preconditions* for a backward predicate transformer semantics of the guarded command language [13]. In this work, we show how symbolic execution can be used to express weakest preconditions. A formal proof of this is highly nontrivial, because it connects a logical framework for preconditions to an infinite set of arbitrarily long executions in an operational semantics. The denotational semantics of symbolic execution ameliorates exactly such complications.

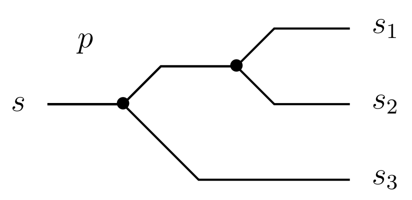
Composing symbolic executions. Historically, the benefit of denotational semantics has been its compositionality. The degree of compositionality of a program semantics is determined by how it is used to reason about programs in terms of their constituents. With operational semantics, the focus is on *execution* of the program. How symbolic executions compose in an operational semantics is not straightforward. For example, suppose we have analyzed two programs p and q and obtained

$$(p, \text{id}, \top) \longrightarrow^* (\varepsilon, \sigma_p, \varphi_p) \qquad \text{and} \qquad (q, \text{id}, \top) \longrightarrow^* (\varepsilon, \sigma_q, \varphi_q)$$

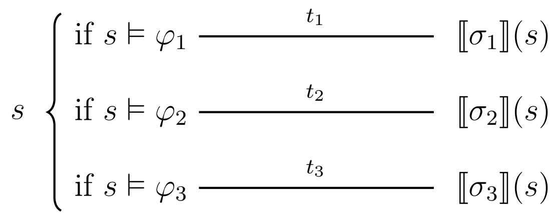
It is not obvious from an operational point of view how these two executions compose to form an execution of the sequenced program $p \mathbin{⨾} q$. This is because the second execution does not continue from the configuration where the first one ended, but starts from the initial configuration $(\text{id}, \top)$. One expects to obtain an execution $(p \mathbin{⨾} q, \text{id}, \top) \longrightarrow^* (\varepsilon, \sigma, \varphi)$ where σ is some composition of σ_p and σ_q, and the path condition φ is some combination of φ_p and $\sigma_p(\varphi_q)$ (i.e., σ_p applied to all the variables occurring in φ_q), because we continued executing q starting from σ_p instead of id. The denotational semantics turns this informal argument into formal theory, and avoids overly complex proofs in the operational semantics.

Contributions. In this work we make the following technical contributions:

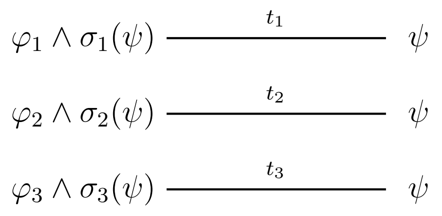
- We develop a denotational framework for symbolic execution.
- We show that symbolic substitutions, interpreted as concrete state transformers, coincide with the state-transforming trace semantics for inputs satisfying the trace path condition (Thm. 1). Moreover, we show that this result lifts to programs (Thm. 2).



(a) a nondeterministically branching program producing a set of outputs

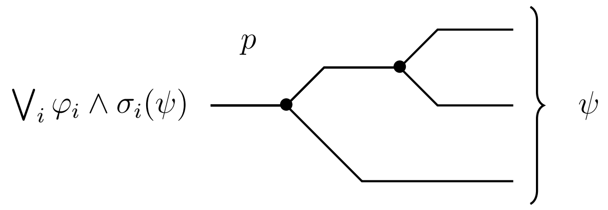


(b) the same program behavior as a set of traces t_i with their respective substitution σ_i and precondition φ_i



$\varphi_1 \wedge \sigma_1(\psi)$
t_1
ψ
$\varphi_2 \wedge \sigma_2(\psi)$
t_2
ψ
$\varphi_3 \wedge \sigma_3(\psi)$
t_3
ψ

(c) trace behavior specified by postcondition ψ, and trace substitutions and path conditions (σ_i, φ_i)



(d) program folded back together, precondition guaranteeing *existence* of an output satisfying ψ

Figure 1: Reasoning about preconditions in terms of the denotational semantics

- We prove correctness and completeness of operational semantics (i.e., symbolic execution) with respect to the denotational semantics (Thms. 3 and 4).
- We prove that symbolic execution techniques compute (weakest) preconditions (Thms. 5 and 6 and Cor. 2) using the framework of propositional dynamic logic.
- We have mechanized[1] our results in the Coq theorem prover [10]; these have been labeled with the symbol.

Expressing preconditions in dynamic logic illustrates the power of a symbolic denotational framework. A conceptual overview of how the symbolic denotational semantics is used to reason about preconditions is exhibited in Figure 1: we represent programs as traces (a to b); substitution lemmas convert this to specifying weakest preconditions of traces with respect to postconditions (b to c); and for the whole program, the disjunction of all trace preconditions will guarantee existence of a trace leading to the postcondition (c to d).

Overview. Section 2 provides a preliminary setup. Section 3 introduces *traces*, the key concept in symbolic execution. Theorem 1 shows that symbolic and concrete semantics of traces corresponds. In Section 4, we use propositional dynamic logic to express weakest (liberal) preconditions of traces. In Section 5, we discuss syntax and semantics of programs and represent them as sets of traces. In Section 6, we introduce the symbolic denotational semantics of programs, culminating in the correspondence result Theorem 2. In Section 7

[1]The mechanized theory is available at https://doi.org/0281/zenodo.14698419

we discuss symbolic operational semantics (i.e., symbolic execution), which we show to be correct and complete (Theorems 3 and 4) in terms of the denotational semantics. In Section 8, to demonstrate the power of the denotational framework, we state and prove that symbolic execution computes preconditions (Theorems 5 and 6 and Cor. 2). Section 9 discusses related work and Section 10 provides some concluding remarks.

This work extends previously published work [41] on semantics of symbolic execution. There, the symbolic semantics consisted of pairs of functions on states and sets of states. In this work, we keep the semantics purely symbolic; the semantics consists of substitutions and path conditions. This enables backward reasoning about preconditions, which is a substantial extension in this work. The earlier version [41] considered procedures as an extension, whereas here we consider probabilistic programming constructs and arrays.

2. Preliminaries

Here we briefly introduce notations used in the paper, including states and expression syntax and semantics.

Concrete states. Program syntax builds on an arbitrary countable set X of variables denoted $x, y, \ldots$. Values for variables range over some domain D, which may include, e.g., integers, rationals, Boolean values, lists, arrays, etc. Programs transform *concrete states* $s\colon X \to D$ (or just *states*), which associate a value $s(x) \in D$ to every $x \in X$ during execution. We let $s[x \mapsto v]$ denote the *updated* state that maps $x \in X$ to $v \in D$ and leaves every other $y \neq x$ in X unchanged, i.e., y maps to $s(y)$. Let $S_X = \{s\colon X \to D\}$ denote the set of states.

Expressions. We use universal algebra [23] to model expressions. Let E be a signature, i.e., a set of operation symbols with $\sharp\colon E \to \mathbb{N}$ providing arities. The set of *E-terms* over a set X is the smallest set E_X such that $X \subseteq E_X$ and such that, for every operator $\mathtt{f} \in E$ with $n = \sharp(\mathtt{f})$, if $e_1, \ldots, e_n \in E_X$, then also $\mathtt{f}(e_1, \ldots, e_n) \in E_X$.

- An *E-algebra* is a pair (X, α) of a set X and a family α of maps $\alpha_{\mathtt{f}}\colon X^{\sharp(\mathtt{f})} \to X$, $\mathtt{f} \in E$ that provide an interpretation of the operators $\mathtt{f}$ in terms of elements of X.

- An *E-algebra homomorphism* between (X, α) and (X', α') is a map $g\colon X \to X'$ that preserves E-operations: $g(\alpha_{\mathtt{f}}(a_1, \ldots, a_{\sharp(\mathtt{f})})) = \alpha'_{\mathtt{f}}(g(a_1), \ldots, g(a_{\sharp(\mathtt{f})}))$ for all $\mathtt{f} \in E$.

The set E_X with the family ι_X of maps $\iota_{X,\mathtt{f}}\colon E_X^{\sharp(\mathtt{f})} \to E_X, e_1, \ldots, e_{\sharp(\mathtt{f})} \mapsto \mathtt{f}(e_1, \ldots, e_{\sharp(\mathtt{f})})$ interpreting tuples of E-terms over X as E-terms over X is an E-algebra. This is the free algebra of E-terms for the set X. For any map $g\colon X \to X'$ and an E-algebra (X', α'), the *inductive extension* of g over α' is the E-algebra homomorphism $\overline{g}\colon E_X \to X'$ inductively defined as $\overline{g}\colon a \mapsto g(a)$ for $a \in X$ and $\overline{g}\colon \mathtt{f}(e_1, \ldots, e_{\sharp(\mathtt{f})}) \mapsto \alpha'(\overline{g}(e_1), \ldots, \overline{g}(e_n))$.

The *syntax of expressions* is provided by the set E_X of E-terms when X is interpreted as the set of program variables.

The *semantics of expressions* comes from some E-algebra (D, ε) that interprets each operator $\mathtt{f} \in E$ as a map $\varepsilon_{\mathtt{f}}\colon D^{\sharp(\mathtt{f})} \to D$. Let $\overline{s}\colon E_X \to D$ denote the inductive extension of a state $s\colon X \to D$ over ε, evaluating expressions in the given state.

Predicates. The *syntax of predicates* is provided by a signature Π of predicate symbols with arities $\sharp\colon \Pi \to \mathbb{N}$. Common examples of predicate symbols include $=$, $<$, and $\in$, all binary. For a predicate symbol $\mathtt{r} \in \Pi$ with $\sharp(\mathtt{r}) = n$, a *predicate over* X is of the form $\mathtt{r}(e_1, \ldots, e_n)$ with each $e_i \in E_X$. Let P_X denote the set of predicates over X.

The *semantics of predicates* comes from an *interpretation* $I\colon \Pi \to \bigcup_{n\in\mathbb{N}} D^n$, assigning to each n-ary predicate symbol the set of n-tuples in D^n for which the predicate is true. Given the semantics ε of expressions, a state $s\colon X \to D$ *satisfies* a predicate $\mathtt{r}(e_1, \ldots, e_n) \in P_X$, written $s \vDash_{\varepsilon,I} \mathtt{r}(e_1, \ldots, e_n)$, if $(\overline{s}(e_1), \ldots, \overline{s}(e_n)) \in I(\mathtt{r})$. Here, $\overline{s}$ denotes the inductive extension of s over ε. We assume ε and I fixed and write $\vDash$ in lieu of $\vDash_{\varepsilon,I}$.

Boolean expressions. The *syntax of Boolean expressions* is generated as follows: every predicate $\pi \in P_X$ over X is a Boolean expression over X; $\bot$ (false) is always a Boolean expression; and if φ and ψ are Boolean expressions over X, then so is $\varphi \to \psi$. Write B_X for the set of Boolean expressions generated this way. We also include the following encodings: negation $\neg\varphi$ as $\varphi \to \bot$; $\top$ (true) as $\neg\bot$; disjunction $\varphi \vee \psi$ as $\neg\varphi \to \psi$; conjunction $\varphi \wedge \psi$ as $\neg(\neg\varphi \vee \neg\psi)$; and biconditional $\varphi \leftrightarrow \psi$ as $(\varphi \to \psi) \wedge (\psi \to \varphi)$.

Recall that $S_X = \{s\colon X \to D\}$. The *semantics* $m(\varphi) \subseteq S_X$ *of a Boolean expression* $\varphi \in B_X$ for the set of states S_X is inductively defined as

- $m(\mathtt{r}(e_1, \ldots, e_n)) = \{s \in S_X \mid s \vDash \mathtt{r}(e_1, \ldots, e_n)\}$;
- $m(\bot) = \emptyset$;
- $m(\varphi \to \psi) = (S_X \setminus m(\varphi)) \cup m(\psi)$.

Note that there is an implicit dependence on ε and I in the base cases $\mathtt{r}(e_1, \ldots, e_n)$ above. We write $s \vDash \varphi$ if $s \in m(\varphi)$. The derived semantic definitions of the encodings for negation, true, disjunction, conjunction, and biconditional, follow conventional propositional logic. Write $\varphi \Rightarrow \psi$ if $m(\varphi) \subseteq m(\psi)$ and $\varphi \equiv \psi$ if $m(\varphi) = m(\psi)$. We will say that a formula φ is *valid* (in m), written $\vDash \varphi$, if $s \vDash \varphi$ for all states $s \in S_X$.

Example 1. Let $D = \mathbb{Z}$ and let E contain integer arithmetic operation symbols, including symbols for integer division '$\div$' and remainder after integer division '%'. Every integer is also a constant symbol of E. Naturally, the map ε evaluates the integer symbols in E as integers in $\mathbb{Z}$, and the symbols '$\div$' and '%' (and others) as the familiar integer operations. Let Π contain symbols for equality '$=$' and inequalities ('$<$', '$\leq$', etc.). Let I model their natural interpretations of equality and the respective inequalities. Now `(3*n+1)%2=0` with $\mathtt{n} \in X$ is a Boolean expression. In a state $s\colon X \to D$ where $s(\mathtt{n}) = 5$, we have $s \vDash$ `3*n+1%2=0`, since, using ε, we have $\overline{s}($`(3*n+1)%2`$) = 0$, and $(0, 0) \in I(=)$.

Composition operators. We use many different symbols for composing different structures. For later reference, here is an overview:

$\fatsemi$	syntactic sequencing of traces and programs
$\circ$	semantic sequencing of traces (function composition)
$\odot$	semantic sequencing of programs (Kleisli composition for the powerset monad)
$\star$	symbolic sequencing of trace substitutions (substitution composition)
$\bullet$	state evaluation by symbolic substitution

3. Traces

Symbolic execution is a technique that represents finite and deterministic runs, or *traces*, of a program by a substitution and path condition. Our starting point will therefore be traces; the theory is extended to programs from Section 5 onwards. We argue that a trace defines a state transformer, and that symbolic execution computes precisely this state transformer. Furthermore, symbolic execution computes a Boolean formula which, as we show later, is a precondition for the trace.

Traces are finite lists of *assignments* of (side-effect free) expressions to variables and *assertions* of Boolean expressions, composed by sequencing. From here on out, X will be the set of program variables.

Definition 1 (Trace Syntax)**.** The set of traces T_X is defined by the grammar:

$$T_X \ni t ::= \varepsilon \mid x := e \mid \varphi? \mid t \fatsemi t$$

where $x \in X$, $e \in E_X$ is an expression and $\varphi \in B_X$ a Boolean expression.

The trace ε denotes the empty trace. Before we can equip the traces with state transformer semantics, we need to deal with failed assertions. Let $\overline{S_X} = S_X \cup \{\oslash\}$ be the extended set of states that includes the *aborted* state $\oslash$ (recall that $S_X = \{s \colon X \to D\}$). Each trace then defines a transformer of states as follows.

Definition 2 (Trace Semantics)**.** Let $t \in T_X$ range over traces. The *trace semantics* $t \colon \overline{S_X} \to \overline{S_X}$ is defined inductively on the structure of traces by

$$t \colon s \mapsto \begin{cases} s & \text{if } t = \varepsilon \\ s[x \mapsto \overline{s}(e)] & \text{if } t = x := e \\ s & \text{if } t = \varphi? \text{ and } s \vDash \varphi \\ \oslash & \text{if } t = \varphi? \text{ and } s \nvDash \varphi \\ (t_2 \circ t_1)(s) & \text{if } t = t_1 \fatsemi t_2 \end{cases}$$

for $s \in S_X$ and $t \colon \oslash \mapsto \oslash$ for $t = \varepsilon$, $t = x := e$, $t = \varphi?$, and $t = t_1 \fatsemi t_2$.

The empty trace acts as the identity and an unsatisfied assertion produces the aborted state. With this semantics, sequencing is associative since function composition is. One may therefore take it to be *right*-associative, without loss of expressiveness. Every nonempty trace is thus of the form $t \fatsemi u$ — possibly by adding a trailing $u = \varepsilon$ — where t is sequence-free, i.e., $t = \varepsilon$, $t = x := e$, or $t = \varphi?$.

Example 2. Intuitively, the cz code snippet from the introduction has exactly two traces—this intuition is made formal in Definition 11 and Lemma 9. The traces of cz are given by $t_1 = \neg(\mathtt{n}\,\%\,2\mathtt{=}0)?\,⨾\,\mathtt{n}\,\mathtt{:=}\,3*\mathtt{n}+1$ and $t_2 = (\mathtt{n}\,\%\,2\mathtt{=}0)?\,⨾\,\mathtt{n}\,\mathtt{:=}\,\mathtt{n}\div 2$.

3.1. Symbolic Substitutions

In addition to the *concrete* semantics, we give a *symbolic* semantics to traces. The symbolic semantics of a trace is given by a symbolic *substitution* that represent its concrete behavior. A symbolic substitution captures an abstract notion of state transformation by mapping variables to expressions.

Definition 3 (Symbolic Substitution)**.** A *symbolic substitution* is a map $\sigma\colon X \to E_X$ from variables to expressions over variables.

We write $\Sigma_X = \{\sigma\colon X \to E_X\}$ for the set of symbolic substitutions. Let $\mathrm{id} \in \Sigma_X$ denote the "identity substitution" $x \mapsto x$ that maps each variable to itself, and $\boldsymbol{\{}x/e\boldsymbol{\}}$ the substitution that maps $x \mapsto e$ and $y \mapsto y$ for $y \neq x$. Substitutions σ are lifted to expressions by inductive extension:

$$\overline{\sigma}\colon E_X \to E_X \qquad \mathtt{f}(e_1,\ldots,e_n) \mapsto \mathtt{f}(\overline{\sigma}(e_1),\ldots,\overline{\sigma}(e_m))$$

Constants (null-ary operators) are left unchanged by $\overline{\sigma}$. The lifting allows us to compose two substitutions $\sigma_1, \sigma_2\colon X \to E_X$ by defining $\sigma_1 \star \sigma_2\colon x \mapsto \overline{\sigma}_1(\sigma_2(x))$. This is well-defined in the sense that $(\sigma_1 \star \sigma_2)(e) = \overline{\sigma_1}(\overline{\sigma_2}(e))$ for all expressions $e \in E_X$. The composition $\star$ is associative and has identity id. Using common *update* notation, note how $\sigma \star \boldsymbol{\{}x/e\boldsymbol{\}} = \sigma[x \mapsto \sigma(e)]$ (and not $\sigma[x \mapsto e]$).

Given a substitution $\sigma\colon X \to E_X$ in Σ_X and a concrete state $s\colon X \to D$ in S_X, *evaluation after substitution* is the map $s \bullet \sigma\colon X \to D$ given by $x \mapsto \overline{s}(\sigma(x))$. This is a new concrete state that associates to every variable x the value obtained by evaluating in s the expression associated to x by the substitution σ. As a state, $s \bullet \sigma$ evaluates an expression $e \in E_X$ like any other: using the extension $\overline{s \bullet \sigma}$. This lifting commutes with application in the following sense:

Lemma 1 (Substitution Lemma for Expressions)**.** Let $\sigma \in \Sigma_X$ and $s \in S_X$. Then $\overline{(s \bullet \sigma)}(e) = \overline{s}(\overline{\sigma}(e))$ for all expressions $e \in E_X$.

With the following definition, we interpret a substitution as a map on states:

Definition 4 (Symbolic Substitution Semantics)**.** The *semantics of a symbolic substitution* $\sigma \in \Sigma_X$ is the map $[\![\sigma]\!]\colon S_X \to S_X$ defined by $s \mapsto s \bullet \sigma$.

By Lemma 1, this composition is well-defined:

Lemma 2 (Composition of Symbolic Substitutions)**.** For all $\sigma, \sigma' \in \Sigma_X$: $[\![\sigma \star \sigma']\!] = [\![\sigma']\!] \circ [\![\sigma]\!]$.

Proof. Let $s\colon X \to D$ in S_X and $x \in X$ be arbitrary. Using Lemma 1, we have

$$[s \bullet (\sigma \star \sigma')](x) = \overline{s}([\sigma \star \sigma'](x)) = \overline{s}(\overline{\sigma}(\sigma'(x))) \overset{\text{Lem. 1}}{=} \overline{s \bullet \sigma}(\sigma'(x)) = [(s \bullet \sigma) \bullet \sigma'](x)$$

showing that $[\![\sigma \star \sigma']\!](s) = [\![\sigma']\!]([\![\sigma]\!](s))$ for arbitrary s. □

By defining the semantics of symbolic substitutions as state transformers, $[\![\mathrm{id}]\!]$ is the identity $s \mapsto s$ on S_X, and $[\![\{x/e\}]\!]$ is the update map $s \mapsto s[x \mapsto s(e)]$. Composition of substitution semantics is associative—simply because it is function composition.

Remark. Semantic composition $[\![\sigma']\!] \circ [\![\sigma]\!]$ is the reverse of syntactic composition $\sigma \star \sigma'$, because syntactic composition is a lifted map and performs uniform substitution of variables in expressions. For example, substituting $(x-1)$ for x in the substitution $x \mapsto 2 \cdot x$ (syntactically, $x \mapsto x-1$ "*after*" $x \mapsto 2 \cdot x$) yields a substitution that associates to x an expression whose semantics *first* computes $x-1$ and *then* multiplies by two.

Substitutions are applied to predicates as in $\sigma(\mathtt{r}(e_1,\ldots,e_n)) \overset{\text{def}}{=} \mathtt{r}(\overline{\sigma}(e_1),\ldots,\overline{\sigma}(e_n))$, and then recursively on Boolean operators, e.g., $\sigma(\varphi \to \psi) \overset{\text{def}}{=} \sigma(\varphi) \to \sigma(\psi)$.

Lemma 3 (Substitution Lemma for Boolean Expressions)**.** Let $\sigma \in \Sigma_X$ and $s \in S_X$. Then $[\![\sigma]\!](s) \vDash \varphi$ if and only if $s \vDash \sigma(\varphi)$, for all Boolean expressions $\varphi \in B_X$.

An important corollary of Lemmas 1 and 3 is that if $\varphi \Rightarrow \varphi'$ then $\sigma(\varphi) \Rightarrow \sigma(\varphi')$.

Example 3. The composition $\sigma \star \sigma'$ of $\sigma, \sigma' \in \Sigma_X$, where $\sigma\colon \mathtt{n} \mapsto 3 * \mathtt{n} + 1$ and $\sigma'\colon \mathtt{n} \mapsto \mathtt{n} \div 2$, maps $\sigma \star \sigma'\colon \mathtt{n} \mapsto (3 * \mathtt{n} + 1) \div 2$. For a state $s \in S_X$ with $s\colon \mathtt{n} \mapsto 5$, we have $[\![\sigma \star \sigma']\!](s)(\mathtt{n}) = \big(s \bullet (\sigma \star \sigma')\big)(\mathtt{n}) = \overline{s}((3 * \mathtt{n} + 1) \div 2) = 8$. This is equivalent to letting $s' = [\![\sigma]\!](s)$, for which $s'(\mathtt{n}) = 3 * 5 + 1 = 16$ and then computing $[\![\sigma']\!](s')(\mathtt{n}) = 16 \div 2 = 8$.

Thus, $[\![\sigma']\!](s') \vDash \mathtt{n} \,\%\, 2 = 0$. By Lemma 3, this is equivalent to $s' \vDash \sigma'(\mathtt{n} \,\%\, 2 = 0)$, i.e., $s' \vDash (\mathtt{n} \div 2) \,\%\, 2 = 0$. Using $s' = [\![\sigma]\!](s)$, this in turn is equivalent to $s \vDash \sigma(\sigma'(\mathtt{n} \,\%\, 2 = 0))$, so $s \vDash ((3 * \mathtt{n} + 1) \div 2) \,\%\, 2 = 0$.

3.2. Symbolic Trace Semantics

Both traces and symbolic substitutions define state transformers. In this section, we define the symbolic semantics of a trace, called the *trace substitution*, as the composition of all its assignments interpreted as substitutions.

Definition 5 (Trace Substitution)**.** Let $t \in T_X$ range over traces. The *trace substitution* $\mathsf{Sub}(t)\colon X \to E_X$ is defined inductively over the structure of traces as follows:

$$\mathsf{Sub}(t) = \begin{cases} \mathrm{id} & \text{if } t = \varepsilon \\ \{x/e\} & \text{if } t = x := e \\ \mathrm{id} & \text{if } t = \varphi? \\ \mathsf{Sub}(t_1) \star \mathsf{Sub}(t_2) & \text{if } t = t_1 \,⨾\, t_2 \end{cases}$$

For the trace semantics in Definition 2, sequencing can be taken to be right-associative without loss of expressivity, because its semantics is composition, which is associative. The same argument holds here: composition of substitutions is associative as well. That is,

$$\mathsf{Sub}((t \,⨾\, u) \,⨾\, v) = (\mathsf{Sub}(t) \star \mathsf{Sub}(u)) \star \mathsf{Sub}(v) = \mathsf{Sub}(t) \star (\mathsf{Sub}(u) \star \mathsf{Sub}(v)) = \mathsf{Sub}(t \,⨾\, (u \,⨾\, v))$$

Thus, with this substitution semantics, we may again assume that every nonempty trace is of the form $t \fatsemi u$ where t is sequence-free, and possibly by adding a trailing $u = \varepsilon$.

The semantics of $\mathsf{Sub}(t)$ coincides with the semantics of t itself (Definition 2), except for aborted computations:

Lemma 4 (). For every trace $t \in T_X$ and input $s \in S_X$, if $t(s) \neq \oslash$ then $t(s) = [\![\mathsf{Sub}(t)]\!](s)$.

Proof. Letting $t = u \fatsemi t'$ with u sequence-free, we do a case analysis on u. The interesting case is $t = x := e \fatsemi t'$. For arbitrary $s \colon X \to D$ and putting $s' = s[x \mapsto s(e)]$,

$$t(s) = t'(s') \overset{\text{IH}}{=} [\![\mathsf{Sub}(t')]\!](s') = [\![\mathsf{Sub}(t')]\!]([\![\{x/e\}]\!](s)) \overset{\text{Lemma 2}}{=} [\![\{x/e\} \star \mathsf{Sub}(t')]\!](s) = [\![\mathsf{Sub}(t)]\!](s)$$

Here we have used the fact that $s' = s[x \mapsto s(e)] = [\![\{x/e\}]\!](s)$. $\square$

Example 4. With t_1 and t_2 from Example 2 representing cz, write φ for $\mathtt{n} \,\%\, 2 = 0$. Then,

$$\begin{aligned}
\mathsf{Sub}(t_1 \fatsemi t_2) &= \mathsf{Sub}(\neg\varphi?) \star \mathsf{Sub}(\mathtt{n} := 3 * \mathtt{n} + 1) \star \mathsf{Sub}(\varphi?) \star \mathsf{Sub}(\mathtt{n} := \mathtt{n} \div 2) \\
&= \mathrm{id} \star \{\mathtt{n}/3 * \mathtt{n} + 1\} \star \mathrm{id} \star \{\mathtt{n}/\mathtt{n} \div 2\} \\
&= \{\mathtt{n}/3 * \mathtt{n} + 1\} \star \{\mathtt{n}/\mathtt{n} \div 2\} \\
&= \{\mathtt{n}/(3 * \mathtt{n} + 1) \div 2\}
\end{aligned}$$

3.3. Path Conditions

In the definition of Sub, all assertions are simply ignored, even though these provide crucial information, namely, for which inputs the trace does not lead to an aborted state. To characterize these inputs, the *path condition* collects all the assertions along a trace.

Definition 6 (Trace Path Condition)**.** The *path condition* $\mathsf{PC}(t) \in B_X$ *of a trace* $t \in T_X$ is defined inductively over the structure of traces as follows:

$$\mathsf{PC}(t) = \begin{cases} \top & \text{if } t = \varepsilon \\ \top & \text{if } t = x := e \\ \varphi & \text{if } t = \varphi? \\ \mathsf{PC}(t_1) \wedge \mathsf{Sub}(t_1)(\mathsf{PC}(t_2)) & \text{if } t = t_1 \fatsemi t_2 \end{cases}$$

Since $\varphi \wedge \top \equiv \varphi$ and $\sigma(\top) \equiv \top$, the conjuncts resulting from ε and $x := e$ may be omitted in the resulting path conditions. The substitution of t is applied to the path condition of u in the sequencing case. It follows straightforwardly from Definitions 5 and 6 that $\mathsf{PC}((t \fatsemi u) \fatsemi v) = \mathsf{PC}(t \fatsemi (u \fatsemi v))$ for all $t, u, v \in T_X$.

Example 5. Consider t_1 and t_2 from Example 2 and $\varphi \equiv \mathtt{n} \,\%\, 2 = 0$ from Example 4. Then,

$$\begin{aligned}
\mathsf{PC}(t_1 \fatsemi t_2) &= \mathsf{PC}(\neg\varphi? \fatsemi \mathtt{n} := 3 * \mathtt{n} + 1) \wedge \mathsf{Sub}(\neg\varphi? \fatsemi \mathtt{n} := 3 * \mathtt{n} + 1)(\mathsf{PC}(\varphi? \fatsemi \mathtt{n} := \mathtt{n} \div 2) \\
&= \mathsf{PC}(\neg\varphi?) \wedge \top \wedge \big(\mathsf{Sub}(\neg\varphi?) \star \mathsf{Sub}(\mathtt{n} := 3 * \mathtt{n} + 1)\big)(\varphi \wedge \top) \\
&\equiv \neg(\mathtt{n} \,\%\, 2 = 0) \wedge \{\mathtt{n}/3 * \mathtt{n} + 1\}(\mathtt{n} \,\%\, 2 = 0) \\
&= \neg(\mathtt{n} \,\%\, 2 = 0) \wedge (3 * \mathtt{n} + 1) \,\%\, 2 = 0
\end{aligned}$$

The path condition of a trace specifies precisely which initial states will *not* be aborted.

Lemma 5 (). For every trace $t \in T_X$ and input $s \colon X \to D$, $t(s) \neq \oslash$ if and only if $s \vDash \mathsf{PC}(t)$.

Proof. Assuming t of the form $u \fatsemi t'$ and letting u be free of sequencing, we do a case analysis:

- $t = \varphi? \fatsemi t'$: by IH, $t'(s) \neq \oslash$ if and only if $s \vDash \mathsf{PC}(t')$. Thus, $t(s) \neq \oslash$ if and only if $s \vDash \varphi$ and $s \vDash \mathsf{PC}(t')$. But this means exactly $s \vDash \mathsf{PC}(t)$.
- $t = x := e \fatsemi t'$: write $s' = [\![\{x/e\}]\!](s) = s[x \mapsto s(e)]$, so $t(s) = t'(s')$. By Definition 6, $s \vDash \mathsf{PC}(t)$ iff $s \vDash \{x/e\}(\mathsf{PC}(t'))$ iff (Lemma 1) $[\![\{x/e\}]\!](s) \vDash \mathsf{PC}(t')$ iff (IH) $\oslash \neq t'(s') = t(s)$.

This case analysis proves the lemma. □

Computing both the final substitution and path condition of a trace amounts to a *symbolic execution* of that trace. This provides a symbolic denotational semantics for traces:

Definition 7 (Symbolic Trace Semantics). The symbolic semantics of a trace $t \in T_X$ is the pair $(\mathsf{Sub}(t), \mathsf{PC}(t)) \in \Sigma_X \times B_X$.

The following result relating the symbolic trace semantics to the concrete trace semantics is now a straightforward corollary of Lemmas 4 and 5:

Theorem 1 (Symbolic-Concrete Correspondence for Traces). For every trace $t \in T_X$ and states $s, s' \in S_X$: $t(s) = s'$ if and only if $s \models \mathsf{PC}(t)$ and $[\![\mathsf{Sub}(t)]\!](s) = s'$.

4. Weakest Preconditions of Traces

Theorem 1 states that $\mathsf{PC}(t)$ is a precondition of t in the sense that it avoids failed assertions. The if and only if tells us that this is the *weakest* such precondition. We now ask more broadly: given a postcondition ψ, what is the weakest precondition that guarantees successful execution of t in such a way that we terminate in a state satisfying ψ? This is what we explore in this section using propositional dynamic logic (PDL) [18], a multi-modal logic in which programs constitute modalities. For now, we restrict the modalities to traces.

Dynamic logic syntax. We extend our Boolean expressions to trace PDL by

$$\widehat{B}_1 \ni \varphi ::= \mathtt{r}(e_1, \ldots, e_n) \mid \bot \mid \varphi \to \varphi \mid [t]\,\varphi$$

where $\mathtt{r}$ ranges over predicate operators in Π, each e_i over expressions in E_X, and t ranges over traces in T_X. We retain encodings of negation, truth, disjunction, conjunction, and biconditional from Boolean expressions. The dual of the box modality is the *diamond* modality encoded by $\langle t\rangle\,\varphi \overset{\text{def}}{=} \neg[t]\,\neg\varphi$.

Dynamic logic semantics. The semantics $m(\varphi) \subseteq S_X$ as defined for predicates, false, conditionals, and the propositional logic encodings is as before. The semantics of the box modality is given by

$$m([t]\,\varphi) \overset{\text{def}}{=} \{s \in S_X \mid t(s) \in m(\varphi) \text{ or } t(s) = \oslash\}$$

Box modalities $[t]\,\varphi$ specify states on which t either aborts or produces an output satisfying φ. This is known as the *weakest liberal precondition* of φ with respect to t.

Using the semantics of negation and box, diamond semantics is derived as

$$m(\langle t\rangle\,\varphi) = S_X \setminus \{s \in S_X \mid t(s) \in m(\neg\varphi) \text{ or } t(s) = \oslash\} = \{s \in S_X \mid t(s) \in m(\varphi)\}$$

Diamond modalities $\langle t\rangle\,\varphi$ specify states on which t *must* produce an output satisfying φ (and not abort). This is known as the *weakest precondition* of φ with respect to t.

Weakest preconditions of traces. By definition, box and diamond modalities are respectively the weakest liberal precondition and weakest precondition of a trace with respect to a postcondition. Symbolic execution computes precisely the necessary information in the form of a path condition and final substitution. Since postconditions are usually given as (modality-free) propositional formulas, and since Lemma 3 was proved in absence of modalities, we consider postconditions to be Boolean formulas $\psi \in B_X$.

Lemma 6 (Weakest Preconditions for Traces)**.** For all traces $t \in T_X$ and Boolean expressions $\psi \in B_X$:

- $[t]\,\psi \equiv \mathsf{PC}(t) \to \mathsf{Sub}(t)(\psi)$ and
- $\langle t\rangle\,\psi \equiv \mathsf{PC}(t) \wedge \mathsf{Sub}(t)(\psi)$.

Proof. By straightforward induction on the structure of t and using Lemmas 4 and 5. $\square$

Example 6. For t_1 defined as $\neg(\mathtt{n} \,\%\, 2 = 0)? \fatsemi \mathtt{n} := 3 * \mathtt{n} + 1$ as in Example 2 and $\psi \equiv \mathtt{n} = 16$, we have the following weakest preconditions:

$$[t_1]\,\psi \equiv (\mathtt{n} \,\%\, 2 = 0) \vee \mathtt{n} = 5 \qquad \langle t_1\rangle\,\psi \equiv \mathtt{n} = 5$$

Dynamic logic theorems. Recall that a formula φ is *valid* if $s \vDash \varphi$ for all $s \in S_X$.

Proposition 1 ()**.** For all variables $x \in X$, expressions $e \in E_X$, formulas $\varphi, \psi \in B_X$, and traces $t, u \in T_X$, the following formulas are valid:

T1. $[\varepsilon]\,\varphi \leftrightarrow \varphi$

T2. $[x := e]\,\varphi \leftrightarrow \{x/e\}(\varphi)$

T3. $[\psi?]\,\varphi \leftrightarrow (\psi \to \varphi)$

T4. $[t \fatsemi u]\,\varphi \leftrightarrow [t]\,[u]\,\varphi$

Proof. We prove validity of **T3** and **T4**. Showing validity of $\varphi \leftrightarrow \psi$, i.e., $\models \varphi \leftrightarrow \psi$, amounts to showing $m(\varphi \leftrightarrow \psi) = S_X$, which is equivalent to $m(\varphi) = m(\psi)$.

T3. If $s \in m([\psi?]\,\varphi)$ then either $\psi?(s) = s \in m(\varphi)$ or $\psi?(s) = \oslash$. By definition, the former means $s \in m(\varphi)$. The latter, by definition, happens iff $s \notin m(\psi)$. Either way, $s \in m(\neg\psi) \cup m(\varphi)$, so $s \in m(\psi \to \varphi)$. The converse implication is proved symmetrically.

T4. If $s \in m([t \mathbin{;} u]\,\varphi)$ then either $u(t(s)) \in m(\varphi)$ or $u(t(s)) = \oslash$. By definition, the former means $t(s) \in m([u]\,\varphi)$ and so $s \in m([t]\,[u]\,\varphi)$; the latter means one of two cases: if $t(s) = \oslash$ then $s \in m([t]\,\psi)$ for any ψ and, in particular, $s \in m([t]\,[u]\,\varphi)$. Otherwise, if $t(s) = s'$ for some $s' \in S_X$ then $u(s') = \oslash$ and $t(s) \in m([u]\,\varphi)$ so $s \in m([t]\,[u]\,\varphi)$. Either way, $s \in m([t]\,[u]\,\varphi)$ and so $m([t \mathbin{;} u]\,\varphi) \subseteq m([t]\,[u]\,\varphi)$. The converse is proved similarly. $\square$

The statements T1-T4 are often used as axioms, which, together with some rules of deduction, may form a system of inference for PDL that breaks down the trace modalities and leaves us with a propositional logic formula. The proposition above tells us that using T1-T4 as axioms for our deduction system would be *sound*, since they are semantically valid. To have a complete system of deduction, the proof system also contains axioms for propositional logic and the following two deduction rules:

R1. Modus ponens: if φ and $\varphi \to \psi$ are theorems, then ψ is also a theorem;

R2. Generalization: if φ is a theorem then $[t]\,\varphi$ is a theorem for all traces $t \in T_X$.

The proofs of soundness and completeness are similar to those in a similar context by Harel et al. [18].

5. Programs

We have introduced substitutions $\mathsf{Sub}(t)$ and path conditions $\mathsf{PC}(t)$ for traces in Section 3, and showed that $\mathsf{Sub}(t)(\psi) \wedge \mathsf{PC}(t)$ is the weakest precondition of a trace with respect to a postcondition ψ in Section 4. We will now extend our formalization to programs that include sequencing, choice, and Kleene-star iteration. The symbolic denotational semantics of programs will be a so-called *collective semantics* [12]; it is given in terms of *sets* of traces. We also discuss the While language: a deterministic subclass of programs.

5.1. Programs

The set of programs over program variables X is denoted Q_X and defined by extending the trace syntax from Definition 1 with operators for choice '+' and iteration '*'.

Definition 8 (Program Syntax)**.** The set of programs $p \in Q_X$ over X is defined by the grammar:

$$Q_X \ni p ::= \varepsilon \mid x := e \mid \varphi? \mid p \mathbin{;} p \mid p + p \mid p^*$$

where $x \in X$, $e \in E_X$ is an expression over X and $\varphi \in B_X$ a Boolean expression.

Iteration binds strongest and sequencing binds stronger than choice, i.e., $p + q \fatsemi r$ means $p + (q \fatsemi r)$. Note that traces are a subclass of programs, i.e., $T_X \subseteq Q_X$.

Programs generate multiple outputs at the same time due to the nondeterministic constructs, and so the semantics of programs will be to produce *sets* of concrete states. Thus the natural choice of semantics for p is a function $p\colon S_X \to \mathcal{P}S_X$. Since $S_X \neq \mathcal{P}S_X$, we cannot directly compose the functions. Instead, we employ *Kleisli composition* [28]. For functions $f\colon X \to \mathcal{P}Y$ and $g\colon Y \to \mathcal{P}Z$, the Kleisli composition is defined as

$$g \odot f\colon X \to \mathcal{P}Z \qquad x \mapsto \{z \in g(y) \mid y \in f(x)\}$$

Kleisli composition is associative, and identity is given by singleton inclusion $x \mapsto \{x\}$. Note that defining composition in this way amounts to working in the Kleisli category for the powerset monad [28]. We are now ready to give semantics to programs.

Definition 9 (Program Semantics)**.** The semantics of programs $p \in Q_X$ is inductively defined as

$$p\colon S_X \to \mathcal{P}S_X, \quad s \mapsto \begin{cases} \{t(s)\} \setminus \{\oslash\} & \text{if } p = t \in T_X \\ q(s) \cup r(s) & \text{if } p = q + r \\ \bigcup_{m=0}^{\infty} q^m(s) & \text{if } p = q^* \\ r \odot q & \text{if } p = q \fatsemi r \end{cases}$$

By removing the aborted state $\oslash$ in the first case, aborted runs are discarded. In the case of iteration, q^m stands for m-fold Kleisli composition of q; for $m = 0$, q^m is the identity $s \mapsto \{s\}$. Given q and s, the definition of iteration is equivalent to the least fixed point of the operator $\lambda W \,.\, \{s\} \cup \{s'' \in q(s') \mid s' \in W\}$, where $W \subseteq S_X$ and the partial order is that of subset inclusion. If programs happen to be traces, trace and program semantics coincide in the case of sequencing, i.e., $(p \fatsemi q)(s) = \{(t \fatsemi u)(s)\}$ if $p \fatsemi q = t \fatsemi u$.

A program $p \in Q_X$ is called *deterministic* if, for all inputs $s \in S_X$, there is *at most* one state in the set $p(s)$. A deterministic program p *diverges* on state s if there are *no* outputs in $p(s)$. If p is deterministic and does not diverge on input s, then there is a unique state $s' \in p(s)$; we write $p(s) = s'$ in this case and say that p *terminates* on input s.

Programs obey several expected laws. Iteration can be unfolded to a choice of the empty program and sequencing, and sequencing distributes over choice.

Lemma 7 ()**.** *For all programs $p, q, r \in Q_X$: $p^* = \varepsilon + p \fatsemi p^*$ and $(p + q) \fatsemi r = p \fatsemi r + q \fatsemi r$.*

Proof. We prove iteration unfolding. The sets $p^*(s) = (\varepsilon + p \fatsemi p^*)(s)$ are equal for all s. Indeed, if $s' \in p^*(s)$ then there is $m \geq 0$ such that $s' \in p^m(s)$. *Either* (i) $m = 0$ and $s' = s$ *or* (ii) $m > 0$ and $s' \in (p^{m-1} \odot p)(s)$. Case (i) means $s' \in \varepsilon(s)$ and in case (ii) $s' \in (p \fatsemi p^*)(s)$. Hence, $s' \in (\varepsilon + p \fatsemi p^*)(s)$. Reverse all implications to prove the converse inclusion. $\square$

5.2. The While Language

Conventional *if* and *while* statements can be encoded in the language as follows:

$$\texttt{if}\ \varphi\ p\ q \overset{\text{def}}{=} (\varphi? \fatsemi p) + (\neg\varphi? \fatsemi q) \qquad\qquad \texttt{while}\ \varphi\ p \overset{\text{def}}{=} (\varphi? \fatsemi p)^* \fatsemi \neg\varphi?$$

By Lemma 7, associativity and unit ε of sequencing, and commutativity of choice,

$$\begin{aligned}\texttt{while}\ \varphi\ p = &\ (\varepsilon + (\varphi? \mathbin{⨾} p) \mathbin{⨾} (\varphi? \mathbin{⨾} p)^*) \mathbin{⨾} \neg\varphi? \\ = &\ \neg\varphi? + \varphi? \mathbin{⨾} p \mathbin{⨾} (\varphi? \mathbin{⨾} p)^* \mathbin{⨾} \neg\varphi? \\ = &\ \texttt{if}\ \varphi\ (p \mathbin{⨾} \texttt{while}\ \varphi\ p)\ \varepsilon\end{aligned}$$

While programs are a subclass of programs that are built exclusively from *inactions*, *assignments*, *sequencing*, and `if` and `while` statements—any assumption, nondeterministic choice, or iteration happens only due to an `if` or `while` statement.

Definition 10 (While Syntax)**.** The set of While programs over X is defined by:

$$W_X \ni p ::= \varepsilon \mid x := e \mid p \mathbin{⨾} p \mid \texttt{if}\ \varphi\ p\ p \mid \texttt{while}\ \varphi\ p$$

where $x \in X$, $e \in E_X$ is an expression over X and $\varphi \in B_X$ a Boolean expression.

Since $W_X \subseteq Q_X$, the semantics of While programs is already defined.

Lemma 8 ( )**.** Every While program $p \in W_X$ is deterministic.

Proof. By induction on the structure of While programs. We look at two cases.

- Let $p = \texttt{if}\ \varphi\ q\ r$, where $q, r \in W_X$, so deterministic (IH). For arbitrary $s \in S_X$, without loss of generality, $s \vDash \varphi$, so $(\neg\varphi? \mathbin{⨾} r)(s)$ diverges, and $(\varphi? \mathbin{⨾} q)(s)$ is deterministic, since $\varphi?(s) = \{s\}$ and q is deterministic. Hence, $p(s)$ contains at most one output.
- Let $p = \texttt{while}\ \varphi\ q$ with $q \in W_X$ and $s \in S_X$ arbitrary. By definition, $p(s) = \bigcup_{m\in\mathbb{N}} F_m$ with $F_m = \big(\neg\varphi? \odot (q \odot \varphi?)^m\big)(s)$. For all m, F_m is either empty or a singleton (IH). Also, $F_m = \{s'\}$ if and only if there are $s_0, s_1, \ldots, s_m \in S_X$ where $s_0 = s$ and $s_m = s'$ such that, for all $j < m$, $s_j \vDash \varphi$ and $s_{j+1} = q(s_j)$ (IH), and $s_m \vDash \neg\varphi$. Hence, if $F_m = \{s'\}$ then $F_\ell = \emptyset$ for all $\ell \neq m$, and we are done.

$\square$

Example 7. The While program `cz` defined as `if` $(\mathtt{n} \,\%\, 2 = 0)$ $\{\mathtt{n} := \mathtt{n} \div 2\}$ $\{\mathtt{n} := 3 * \mathtt{n} + 1\}$ from the running example introduced in Section 1 is deterministic. The While program (hence, deterministic) $\mathsf{Cz} \overset{\text{def}}{=} \texttt{while}\ \neg(\mathtt{n} = 1)\ \{\mathtt{cz}\}$ repeats `cz` until the value of `n` becomes one. Whether Cz terminates on all positive integer inputs is unknown—a problem known as the *Collatz conjecture*.

5.3. Programs as Traces

Crucial to this work, in this section, a program is represented as a set of traces.

Definition 11 (Program Traces)**.** Let $p \in Q_X$ range over programs. The sets $\mathcal{T}_p \subseteq T_X$ of *traces through* p are inductively defined on the structure of programs as follows:

$$\mathcal{T}_p := \begin{cases} \{t\} & \text{if } p = t \in T_X \\ \mathcal{T}_q \cup \mathcal{T}_r & \text{if } p = q + r \\ \bigcup_{m=0}^{\infty} \mathcal{T}_{q^m \fatsemi \varepsilon} & \text{if } p = q^* \\ \{t \fatsemi u \mid t \in \mathcal{T}_q, u \in \mathcal{T}_r\} & \text{if } p = q \fatsemi r \end{cases}$$

Here, $q^m \fatsemi \varepsilon$ means q sequenced m times followed by a trailing ε, which is just ε when $m = 0$. This is equivalent to the least fixed point of the operator $\lambda W \,.\, \{\varepsilon\} \cup \{u \fatsemi t \mid u \in \mathcal{T}_q, t \in W\}$, where $W \subseteq T_X$ and the partial order is that of subset inclusion. Thus, $\mathcal{T}_{q^*} = \mathcal{T}_\varepsilon \cup \mathcal{T}_{q \fatsemi q^*}$. This equation mirrors Lemma 7. Note that, in general, two programs can have different sets of traces, even though they may produce the same outputs.

The set $\mathcal{T}_p$ of traces of a program p is a correct representation:

Lemma 9 ()**.** For all programs $p \in Q_X$ and every input $s \in S_X$, $p(s) = \{t(s) \mid t \in \mathcal{T}_p\} \setminus \{\oslash\}$.

Proof. Straightforward induction on the structure of programs. $\square$

Example 8. Let cz and Cz be the programs in Example 7. Then $\mathcal{T}_{\mathsf{cz}} = \{t_1, t_2\}$ where $t_1 = \neg(\mathtt{n}\,\%\,2\mathtt{=}0)? \fatsemi \mathtt{n}\mathtt{:=}3 * \mathtt{n} + 1$ and $t_2 = (\mathtt{n}\,\%\,2\mathtt{=}0)? \fatsemi \mathtt{n}\mathtt{:=}\mathtt{n} \div 2$. For s with $\mathtt{n} \mapsto 5$, $p(s) = t_1(s)$ with $t_1(s)(\mathtt{n}) = 16$, and for s with $\mathtt{n} \mapsto 16$, $p(s) = t_2(s)$ with $t_2(s)(\mathtt{n}) = 8$. Traces $t \in \mathcal{T}_{\mathsf{Cz}}$ of Cz are all of the form $\neg(\mathtt{n} = 1)? \fatsemi t^{(1)} \fatsemi \neg(\mathtt{n} = 1)? \fatsemi t^{(2)} \fatsemi \ldots \fatsemi \neg(\mathtt{n} = 1)? \fatsemi t^{(m)} \fatsemi (\mathtt{n} = 1)?$ for some $m \geq 0$ where each $t^{(j)} \in \mathcal{T}_{\mathsf{cz}}$.

6. Denotational Symbolic Semantics

We have shown how the concrete semantics of a single trace is captured by a symbolic execution (Lemma 1), and how a program is represented by a set of traces (Lemma 9). Combining these results, we obtain a natural description of symbolic execution as a technique that traverses all the traces, all the while computing each trace substitution and path condition. The denotational symbolic semantics of traces is generalized to programs as follows.

Definition 12 (Symbolic Program Semantics)**.** The denotational symbolic semantics of a program $p \in Q_X$ is the set

$$\mathcal{F}_p = \{(\mathsf{Sub}(t), \mathsf{PC}(t)) \mid t \in \mathcal{T}_p\}$$

For example, $(\mathrm{id}, \top)$, $(\{x/e\}, \top)$, and (id, φ), are the singleton elements of $\mathcal{F}_\varepsilon$, $\mathcal{F}_{x\mathtt{:=}e}$, and $\mathcal{F}_{\varphi?}$, respectively. We moreover have the following characterizations showcasing the compositional nature of the symbolic semantics:

Lemma 10 (Symbolic Semantics Properties)**.** For all programs $p, q \in Q_X$:

(i) Sequencing: $\mathcal{F}_{p \mathbin{;} q} = \{(\sigma \star \sigma', \varphi \wedge \sigma(\varphi')) \mid (\sigma, \varphi) \in \mathcal{F}_p, (\sigma', \varphi') \in \mathcal{F}_q\}$;

(ii) Choice $\mathcal{F}_{p+q} = \mathcal{F}_p \cup \mathcal{F}_q$;

(iii) Induction: $\mathcal{F}_{p^*} = \bigcup_{m \in \mathbb{N}} \mathcal{F}_{p^m \mathbin{;} \varepsilon}$.

(iv) Unfolding: $\mathcal{F}_{p^*} = \mathcal{F}_\varepsilon \cup \mathcal{F}_{p \mathbin{;} p^*}$; and

Proof. Items (i) to (iii) are immediate from Definition 11; Item (i) also relies on Definitions 5 and 6. *Unfolding* follows from the least fixed point characterization $\mathcal{T}_{p^*} = \mathcal{T}_\varepsilon \cup \mathcal{T}_{p \mathbin{;} p^*}$. □

The sets $\mathcal{F}_p$ and $\mathcal{F}_{p \mathbin{;} \varepsilon}$ are not syntactically identical, because of an extra conjunct $\top$ originating from the PC of the trailing ε. We write $\mathcal{F}_p \equiv \mathcal{F}_q$ if there is a one-to-one correspondence between $\mathcal{F}_p$ and $\mathcal{F}_q$ such that $(\sigma, \varphi) \in \mathcal{F}_p$ corresponds to $(\sigma, \varphi') \in \mathcal{F}_q$ with $\varphi \equiv \varphi'$. Let $(\sigma, \varphi) \mathrel{\sqsubseteq} \mathcal{F}_p$ denote that there exists $\varphi' \in B_X$ such that $(\sigma, \varphi') \in \mathcal{F}_p$ and $\varphi \equiv \varphi'$.

Example 9. Consider the deterministic While program Cz with the while loop. The Collatz conjecture (see Example 7) states that the program terminates for any initial state. We will express this conjecture using the symbolic semantics. Letting $X = \{\mathtt{n}\}$ and $D = \mathbb{N}_+$ (positive integers), the set of states $S_X = \{s \colon X \to D\}$ is isomorphic to $\mathbb{N}_+$. Thus, we can identify states in S_X with values in $\mathbb{N}_+$ for the unique variable $\mathtt{n}$.

Assume the conjecture is true and let $n \in \mathbb{N}_+$ be an arbitrary initial state. By the conjecture, and using Lemma 9, there is $n' \in \mathbb{N}_+$ and $t \in \mathcal{T}_{\mathsf{Cz}}$ such that $n' = t(n)$. Subsequently applying Lemma 5, we have $s \vDash \mathsf{PC}(t)$, and therefore $s \vDash \bigvee_{(\sigma,\varphi) \in \mathcal{F}_{\mathsf{Cz}}} \varphi$. Thus, if the Collatz conjecture holds, then $n \vDash \bigvee_{(\sigma,\varphi) \in \mathcal{F}_{\mathsf{Cz}}} \varphi$ for every initial state $n \in \mathbb{N}_+$.

Conversely, if there is $(\sigma, \varphi) \in \mathcal{F}_{\mathsf{Cz}}$ such that $n \vDash \varphi$, then, by Lemma 5, there is $n' \in \mathbb{N}_+$ and $t \in \mathcal{T}_{\mathsf{Cz}}$ such that $t(n) = n'$, i.e., Cz terminates on n. Therefore, the Collatz conjecture is equivalent to the statement that $\bigvee_{(\sigma,\varphi) \in \mathcal{F}_{\mathsf{Cz}}} \varphi$ holds for all initial states $n \in \mathbb{N}_+$.

The symbolic semantics corresponds to the concrete semantics in the following way:

Theorem 2 (Concrete-Symbolic Correspondence)**.** Let $p \in Q_X$ be a program and $s, s' \in S_X$ concrete states.

(i) If $s' \in p(s)$ then there is $(\sigma, \varphi) \in \mathcal{F}_p$ such that $s \vDash \varphi$ and $[\![\sigma]\!](s) = s'$.

(ii) If $(\sigma, \varphi) \in \mathcal{F}_p$ and $s \vDash \varphi$ then $[\![\sigma]\!](s) \in p(s)$.

The first part (i) expresses completeness: every concrete, non-aborted computation has a corresponding symbolic computation. Conversely, the second part (ii) expresses correctness: every symbolic computation describes the concrete executions for initial states that satisfy the path condition.

Proof. By Lemma 9, $s' \in p(s)$ iff there is $t \in \mathcal{T}_p$ such that $t(s) = s'$. By Lemmas 4 and 5, this s' is a non-aborted state if and only if $s \vDash \mathsf{PC}(t)$ and $[\![\mathsf{Sub}(t)]\!](s) = s'$. □

$$\frac{}{(x := e, \sigma, \varphi) \longrightarrow (\varepsilon, \sigma[x \mapsto \sigma(e)], \varphi)}\ \textsf{asgn} \qquad \frac{}{(\varphi?, \sigma, \varphi) \longrightarrow (\varepsilon, \sigma, \varphi \wedge \sigma(\varphi))}\ \textsf{asm}$$

$$\frac{}{(\varepsilon \fatsemi p, \sigma, \varphi) \longrightarrow (p, \sigma, \varphi)}\ \textsf{seq-0} \qquad \frac{}{(p + q, \sigma, \varphi) \longrightarrow (p, \sigma, \varphi)}\ \textsf{ndet-L}$$

$$\frac{(p, \sigma, \varphi) \longrightarrow (p', \sigma', \varphi')}{(p \fatsemi q, \sigma, \varphi) \longrightarrow (p' \fatsemi q, \sigma', \varphi')}\ \textsf{seq-n} \qquad \frac{}{(p + q, \sigma, \varphi) \longrightarrow (q, \sigma, \varphi)}\ \textsf{ndet-R}$$

$$\frac{}{(p^*, \sigma, \varphi) \longrightarrow (\varepsilon, \sigma, \varphi)}\ \textsf{halt} \qquad \frac{}{(p^*, \sigma, \varphi) \longrightarrow (p \fatsemi p^*, \sigma, \varphi)}\ \textsf{unfold}$$

Figure 2: Symbolic Execution

7. Operational Symbolic Semantics: Symbolic Execution

The denotational symbolic semantics defines the symbolic substitutions and path conditions for each program trace. Symbolic execution *systems* are small-step transition systems that compute these substitutions and path conditions operationally. The rules for a system implementing symbolic execution for programs $p \in Q_X$ are shown in Figure 2. In this system, states (p, σ, φ) consist of (i) a program p to be executed; (ii) a symbolic substitution σ recording trace behavior; and (iii) the trace precondition φ, called the *path condition*. Recall that $\longrightarrow^*$ denotes the reflexive-transitive closure of $\longrightarrow$. A chain $(p, \sigma, \varphi) \longrightarrow^* (p', \sigma', \varphi')$ is called a *symbolic execution*.

In the transition system, sequencing is evaluated from left to right, meaning a program $p \fatsemi q \fatsemi r$ is read as $p \fatsemi (q \fatsemi r)$. The *updated* $\sigma[x \mapsto \sigma(e)]$ in Rule $\textsf{asgn}$ equals the composition $\sigma \star \{x/e\}$. A pair (σ, φ) is called a *configuration*.

We write $(p, \sigma, \varphi) \Longrightarrow^* (p', \sigma', \varphi')$ if there is a path condition φ'' such that $(p, \sigma, \varphi) \longrightarrow^* (p', \sigma', \varphi'')$ and $\varphi'' \equiv \varphi'$. Similar to '$\sqsubseteq$', we introduce the notation '$\Longrightarrow^*$' because of possible syntactic mismatches of conjuncts $\top$ between denotational and operational semantics. For example, $(\varphi?, \mathrm{id}, \top) \longrightarrow (\varepsilon, \mathrm{id}, \top \wedge \varphi)$, but we do *not* have $(\mathrm{id}, \top \wedge \varphi) \in \mathcal{F}_{\varphi?}$. However, we *do* have $(\mathrm{id}, \top \wedge \varphi) \sqsubseteq \mathcal{F}_{\varphi?}$. Conversely, $(\mathrm{id}, \varphi) \in \mathcal{F}_{\varphi?}$ but *not* $(\varphi?, \mathrm{id}, \top) \longrightarrow^* (\varepsilon, \mathrm{id}, \varphi)$. However, we *do* have $(\varphi?, \mathrm{id}, \top) \Longrightarrow^* (\varepsilon, \mathrm{id}, \varphi)$.

Analyzing the rules in Figure 2, using the Substitution Lemmas 1 and 3, it can be shown that, if $(p, \sigma, \varphi) \Longrightarrow^* (p', \sigma', \varphi') \longrightarrow (p'', \sigma'', \varphi'')$ then $(p, \sigma, \varphi) \Longrightarrow^* (p'', \sigma'', \varphi'')$. By induction, if $(p, \sigma, \varphi) \Longrightarrow^* (p', \sigma', \varphi') \longrightarrow^* (p'', \sigma'', \varphi'')$ then $(p, \sigma, \varphi) \Longrightarrow^* (p'', \sigma'', \varphi'')$.

Theorem 3 (Correctness)**.** If $(p, \mathrm{id}, \top) \longrightarrow^* (\varepsilon, \sigma, \varphi)$ then $(\sigma, \varphi) \sqsubseteq \mathcal{F}_p$.

Theorem 4 (Completeness)**.** If $(\sigma, \varphi) \in \mathcal{F}_p$ then $(p, \mathrm{id}, \top) \Longrightarrow^* (\varepsilon, \sigma, \varphi)$.

The proofs of Theorems 3 and 4 are provided below. The proof of correctness is by induction on the length of the transition chain. The statement of correctness concerns symbolic executions that are both *terminating* (the final state contains the inactive program ε) and *canonical* (starting from the *initial configuration* $(\mathrm{id}, \top)$). Observe that induction on transition chains that are both terminating and canonical is a challenging task, since the IH cannot be applied to transition chains of length one step smaller: this will be either non-canonical (analyzing the first step of the chain) or non-terminating (analyzing the last step). To make the proof work, we analyze the first step, and the remaining transition chain must be reconfigured to the canonical symbolic execution. The canonical execution starting from a program p typifies *all* symbolic executions from p in the following way:

Proposition 2 (Canonical Symbolic Excecution)**.** For all programs $p, q \in Q_X$ and configurations $(\sigma, \varphi), (\sigma', \varphi') \in \Sigma_X \times B_X$, if $(p, \sigma, \varphi) \longrightarrow^* (q, \sigma', \varphi')$ then there exists (σ_1, φ_1) such that $(p, \mathrm{id}, \top) \longrightarrow^* (q, \sigma_1, \varphi_1)$, $\sigma' = \sigma_1 \star \sigma$, and $\varphi' \equiv \varphi \wedge \sigma(\varphi_1)$.

Proof. By induction on the length of the transition chain, analyzing the last execution step in the chain. The base case (length zero) is trivial. For the inductive step, we have $(p, \sigma, \varphi) \longrightarrow^* (q, \sigma', \varphi') \longrightarrow (q'', \sigma'', \varphi'')$. The IH states that there is $(p, \mathrm{id}, \top) \longrightarrow^* (q, \sigma_1, \varphi_1)$ such that $\sigma' = \sigma \star \sigma_1$ and $\varphi' \equiv \varphi \wedge \sigma(\varphi_1)$. The goal is to show that $(q, \sigma_1, \varphi_1) \longrightarrow (q'', \sigma_2, \varphi_2)$ such that $\sigma'' = \sigma \star \sigma_2$ and $\varphi'' \equiv \varphi \wedge \sigma(\varphi_2)$.

- If the last transition was justified using Rule `asgn` then $\sigma'' = \sigma' \star \{x/e\}$ and $\varphi'' = \varphi'$. We have a matching transition $(q, \sigma_1, \varphi_1) \longrightarrow (q'', \sigma_2, \varphi_2)$ with $\sigma_2 = \sigma_1 \star \{x/e\}$ and $\varphi_2 = \varphi_1$. By IH, $\sigma'' = (\sigma \star \sigma_1) \star \{x/e\} = \sigma \star \sigma_2$, and clearly $\varphi'' \equiv \varphi \wedge \sigma(\varphi_2)$.
- If the last transition was justified using Rule `asm` then $\sigma'' = \sigma'$ and $\varphi'' = \varphi' \wedge \sigma'(\varphi)$ for some $\varphi \in B_X$. Clearly, $\sigma'' = \sigma' = \sigma \star \sigma_1 = \sigma \star \sigma_2$. Moreover, $\varphi'' \equiv \varphi' \wedge \sigma'(\varphi) \equiv (\varphi \wedge \sigma(\varphi_1)) \wedge (\sigma \star \sigma_1)(\varphi) \equiv \varphi \wedge \sigma(\varphi_1 \wedge \sigma_1(\varphi)) \equiv \varphi \wedge \sigma(\varphi_2)$.
- For any other rule justifying the last transition, except for Rule `seq-n`, the configuration remains unchanged. That is, $(\sigma'', \varphi'') = (\sigma', \varphi')$ and also $(q, \sigma_1, \varphi_1) \longrightarrow (q'', \sigma_2, \varphi_2)$ such that $(\sigma_2, \varphi_2) = (\sigma_1, \varphi_1)$. In these cases, the result is immediate from IH.
- For Rule `seq-n` let $(q \fatsemi r, \sigma', \varphi') \longrightarrow (q'' \fatsemi r, \sigma'', \varphi'')$ for some trailing program r be the last transition, justified by $(q, \sigma', \varphi') \longrightarrow (q'', \sigma'', \varphi'')$. Similarly, $(q \fatsemi r, \sigma_1, \varphi_1) \longrightarrow (q'' \fatsemi r, \sigma_2, \varphi_2)$ must be justified by $(q, \sigma_1, \varphi_1) \longrightarrow (q'', \sigma_2, \varphi_2)$. Since sequencing is performed from left to right, the transitions $(q, \sigma', \varphi') \longrightarrow (q'', \sigma'', \varphi'')$ and $(q, \sigma_1, \varphi_1) \longrightarrow (q'', \sigma_2, \varphi_2)$ are *not* justified by Rule `seq-n`. Hence, a subsequent analysis of the other rules, as in the previous three cases, shows that $\sigma'' = \sigma \star \sigma_2$ and $\varphi'' = \varphi \wedge \sigma(\varphi_2)$.

Induction on the transition chain length is finished, and the result is established. □

Towards a full correctness proof, we show how each single step in the transition system is consistent with the denotational semantics:

Lemma 11 (Canonical One-Step Correctness)**.** For all programs $p \in Q_X$, if $(p, \mathrm{id}, \top) \longrightarrow (q, \sigma, \varphi)$ and $(\sigma', \varphi') \in \mathcal{F}_q$, then $(\sigma \star \sigma', \varphi \wedge \sigma(\varphi')) \sqsubseteq \mathcal{F}_p$.

By induction on the length of the chain, this extends to symbolic executions in general:

Corollary 1 (Canonical Correctness)**.** For all programs $p, q \in Q_X$, if $(p, \mathrm{id}, \top) \longrightarrow^* (q, \sigma, \varphi)$ and $(\sigma', \varphi') \in \mathcal{F}_q$, then $(\sigma \star \sigma', \varphi \wedge \sigma(\varphi')) \sqsubseteq \mathcal{F}_p$.

Proof (of Lemma 11). We proceed by induction on the structure of programs p, analyzing all possible transitions. Inaction ε has no outgoing transitions and the two other base cases (assignment $x := e$ and assertion $\varphi?$) are mechanically verified.

- For sequencing with Rule **seq-0**, we have $(\varepsilon \fatsemi q, \mathrm{id}, \top) \longrightarrow (q, \mathrm{id}, \top)$. Observe that $(\mathrm{id} \star \sigma', \top \wedge \mathrm{id}(\varphi')) \equiv (\sigma', \varphi')$ for all $(\sigma', \varphi') \in \mathcal{F}_q$ and that $\mathcal{F}_q \equiv \mathcal{F}_{\varepsilon \fatsemi q}$, so $(\sigma', \varphi') \sqsubseteq \mathcal{F}_{\varepsilon \fatsemi q}$.
- Assume $(p \fatsemi r, \mathrm{id}, \top) \longrightarrow (q \fatsemi r, \sigma, \varphi)$ is justified using Rule **seq-n** by $(p, \mathrm{id}, \top) \longrightarrow (q, \sigma, \varphi)$, and let $(\sigma', \varphi') \in \mathcal{F}_{q \fatsemi r}$. By Lemma 10(i), $\sigma' = \sigma_1 \star \sigma_2$ and $\varphi' = \varphi_1 \wedge \sigma_1(\varphi_2)$ for some $(\sigma_1, \varphi_1) \in \mathcal{F}_q$ and $(\sigma_2, \varphi_2) \in \mathcal{F}_r$. By IH (p), $(\sigma \star \sigma_1, \varphi \wedge \sigma(\varphi_1)) \sqsubseteq \mathcal{F}_p$. Then, again using Lemma 10(i), it follows that $([\sigma \star \sigma_1] \star \sigma_2, [\varphi \wedge \sigma(\varphi_1)] \wedge (\sigma \star \sigma_1)(\varphi_2)) \sqsubseteq \mathcal{F}_{p \fatsemi r}$, and this is equivalent to $(\sigma \star \sigma', \varphi \wedge \sigma(\varphi'))$, so we are done. This is the only case using IH.
- For $p + q$, apply Rule **ndet-L** or Rule **ndet-R**; the result is immediate from Lemma 10(ii).
- For p^* and analyzing Rules **halt** and **unfold**, the result follows from Lemma 10(iv).

Induction for the proof of Lemma 11 is now finished.

The proof of Corollary 1 is by induction on the length of the transition chain, analyzing the *first* transition in the chain. For $(p, \mathrm{id}, \top) \longrightarrow (q, \sigma_1, \varphi_1) \longrightarrow^* (p', \sigma', \varphi')$, the IH cannot be applied directly to $(q, \sigma_1, \varphi_1) \longrightarrow^* (p', \sigma', \varphi')$, because it does not start from the initial configuration $(\mathrm{id}, \top)$. Instead, the inductive step is finished by reconfiguring the chain with Proposition 2, applying IH, and using Lemma 11 to prepone $(p, \mathrm{id}, \top) \longrightarrow (q, \sigma_1, \varphi_1)$. $\square$

In the proof of Lemma 11, IH is used only for sequencing. This reveals how traces (i.e., sequences) are at the heart of operational semantics. Choice and iteration, operationally speaking, merely perform internal reprogramming, picking a trace to continue with.

Proof (of Theorems 3 and 4). Correctness is a special case of Corollary 1, taking $q = \varepsilon$. The proof of completeness proceeds by induction on the structure of programs. The base cases are mechanically verified, e.g., $\mathcal{F}_p$ for $p = x := e$ we have $(\sigma, \varphi) = (\{x/e\}, \top)$ and $(x := e, \mathrm{id}, \top) \longrightarrow^* (\varepsilon, \{x/e\}, \top)$. The case for nondeterministic choice is immediate from the IHs, since the transition $(p + q, \sigma, \varphi) \longrightarrow (p, \sigma, \varphi)$ does not change the configuration. Two cases remain:

- For $(\sigma, \varphi) \in \mathcal{F}_{p \fatsemi q}$ we know that $(\sigma, \varphi) = (\sigma_1 \star \sigma_2, \varphi_1 \wedge \sigma_1(\varphi_2))$ for some $(\sigma_1, \varphi_1) \in \mathcal{F}_p$ and $(\sigma_2, \varphi_2) \in \mathcal{F}_q$. Apply IH (p) to get $(p, \mathrm{id}, \top) \Longrightarrow^* (\varepsilon, \sigma_1, \varphi_1)$. Sequence the whole chain with q to get $(p \fatsemi q, \mathrm{id}, \top) \Longrightarrow^* (\varepsilon \fatsemi q, \sigma_1, \varphi_1)$ and transition to (q, σ_1, φ_1). Next, apply IH (q) to get $(q, \mathrm{id}, \top) \Longrightarrow^* (\varepsilon, \sigma_2, \varphi_2)$. Reconfigure with Proposition 2 to the symbolic execution $(q, \sigma_1, \varphi_1) \Longrightarrow^* (\varepsilon, \sigma', \varphi')$ such that $\sigma' = \sigma_1 \star \sigma_2$ and $\varphi' \equiv \varphi_1 \wedge \sigma_1(\varphi_2)$. Observe that $(\sigma, \varphi) \equiv (\sigma', \varphi')$, so we are done.
- For $(\sigma, \varphi) \in \mathcal{F}_{p^*} = \bigcup_{m \in \mathbb{N}} \mathcal{F}_{p^m \fatsemi \varepsilon}$ we prove by induction on the number of iterations m that, for all m: for all $(\sigma, \varphi) \in \mathcal{F}_{p^m \fatsemi \varepsilon}$ we have $(p^*, \mathrm{id}, \top) \Longrightarrow^* (\varepsilon, \sigma, \varphi)$. The base case, $m = 0$ is immediate, since $\mathcal{F}_\varepsilon = \{(\mathrm{id}, \top)\}$ and $(p^*, \mathrm{id}, \top) \longrightarrow (\varepsilon, \mathrm{id}, \top)$ by Rule **halt**. If $m > 0$, observe that $(\sigma, \varphi) \in \mathcal{F}_{p \fatsemi p^{m-1} \fatsemi \varepsilon}$ so that $\sigma = \sigma_1 \star \sigma'$ and $\varphi = \varphi_1 \wedge \sigma_1(\varphi')$ for some

$(\sigma_1, \varphi_1) \in \mathcal{F}_p$ and $(\sigma', \varphi') \in \mathcal{F}_{p^{m-1} \fatsemi \varepsilon}$. Apply IH (p) to get $(p, \mathrm{id}, \top) \Longrightarrow^* (\varepsilon, \sigma_1, \varphi_1)$ and IH $(m-1)$ to get $(p^*, \mathrm{id}, \top) \Longrightarrow^* (\varepsilon, \sigma', \varphi')$. Using Rule unfold and the same sequencing and reconfiguring technique as in the previous item, we obtain $(p^*, \mathrm{id}, \top) \Longrightarrow^* (\varepsilon, \sigma, \varphi)$.

The proof of completeness is thus concluded. $\square$

We conclude this section by operationally characterizing `if` and `while` statements.

Proposition 3 (If and While Statements)**.** For all programs p, q, Boolean expressions φ, and symbolic configurations (σ, φ), the following transition chains are always provable:

(i) $(\texttt{if}\ \varphi\ p\ q, \sigma, \varphi) \longrightarrow^* (p, \sigma, \varphi \wedge \sigma(\varphi))$;

(ii) $(\texttt{if}\ \varphi\ p\ q, \sigma, \varphi) \longrightarrow^* (q, \sigma, \varphi \wedge \sigma(\neg\varphi))$;

(iii) $(\texttt{while}\ \varphi\ p, \sigma, \varphi) \longrightarrow^* (p \fatsemi \texttt{while}\ \varphi\ p, \sigma, \varphi \wedge \sigma(\varphi))$; and

(iv) $(\texttt{while}\ \varphi\ p, \sigma, \varphi) \longrightarrow^* (\varepsilon, \sigma, \varphi \wedge \sigma(\neg\varphi))$.

Proof. For Item (i) the first transition is $(\texttt{if}\ \varphi\ p\ q, \sigma, \varphi) \longrightarrow (\varphi? \fatsemi p, \sigma, \varphi)$, justified by Rule ndet-L. The Rules asm and seq-n prove that $(\varphi? \fatsemi p, \sigma, \varphi) \longrightarrow (\varepsilon \fatsemi p, \sigma, \varphi \wedge \sigma(\varphi))$, and this transitions to $(p, \sigma, \varphi \wedge \sigma(\varphi))$ by Rule seq-0. Item (ii) is proved in the same way but with Rule ndet-R. Item (iii) is proved by sequencing Rule unfold, yielding $\varphi? \fatsemi p \fatsemi (\varphi? \fatsemi p)^*$, with Rule asm. Item (iv) is proved with Rule halt and then sequencing with a $\neg\varphi$-assumption. $\square$

8. Weakest Preconditions

In this section we generalize the propositional dynamic logic of Section 4 from traces to programs. The syntax is identical; except that the modalities now contain *programs* rather than traces.

Definition 13 (PDL for Programs)**.** The syntax of PDL is given by:

$$\widehat{B}_1 \ni \varphi \ ::= \ \mathtt{r}(e_1, \ldots, e_n) \ \mid \ \bot \ \mid \ \varphi \to \varphi \ \mid \ [p]\,\varphi$$

where $\mathtt{r}$ ranges over predicate operators in Π, each e_i over expressions in E_X, and p ranges over programs in Q_X.

The semantics m of predicates, false, and conditionals is as defined in Section 2. The semantics

$$m([p]\,\varphi) \overset{\text{def}}{=} \{s \mid \forall s' \in p(s)\colon s' \in m(\varphi)\}$$

supersedes the semantics of trace modalities: if p happens to be a trace then the trace box modality and program box modality coincide. By definition, $[p]\,\varphi$ is the weakest liberal precondition of p with respect to φ. We retain all previous encodings for negation, true, disjunction, conjunction and again add $\langle p\rangle\,\varphi$ encoded as the dual $\neg[p]\,\neg\varphi$ of the box modality. The diamond modality's derived semantics is

$$m(\langle p\rangle\,\varphi) = \{s \mid \exists s' \in p(s)\colon s' \in m(\varphi)\}$$

If p happens to be a trace then the trace diamond modality and program diamond modality coincide. For *deterministic* programs p, and in particular While programs, the formula $\langle p\rangle\,\psi$ expresses the weakest precondition of p with respect to ψ.

Theorem 5 (Weakest Liberal Precondition)**.** For all programs $p \in Q_X$ and modality-free formulas $\psi \in B_X$:

$$[p]\,\psi \equiv \bigwedge_{(\sigma,\varphi)\in\mathcal{F}_p} \varphi \to \sigma(\psi)$$

Proof. Let $s \vDash [p]\,\psi$ and $(\sigma,\varphi) \in \mathcal{F}_p$. If $s \vDash \neg\varphi$ then $s \vDash \varphi \to \sigma(\psi)$. If $s \vDash \varphi$ then $[\![\sigma]\!](s) \in p(s)$ by Theorem 2, and hence, $[\![\sigma]\!](s) \vDash \psi$ because $s \vDash [p]\,\psi$; equivalently, using Lemma 1, $s \vDash \sigma(\psi)$; hence $s \vDash \varphi \to \sigma(\psi)$. We have shown $[p]\,\psi \Rightarrow (\varphi \to \sigma(\psi))$ and since (σ,φ) was arbitrary, $[p]\,\psi \Rightarrow \bigwedge_{(\sigma,\varphi\in\mathcal{F}_p)} \varphi \to \sigma(\psi)$. The converse is proved similarly. □

Using De Morgan laws for box and diamond modalities and for disjunction and conjunction, the following result is immediate:

Theorem 6 ()**.** For all programs $p \in Q_X$ and modality-free formulas $\psi \in B_X$:

$$\langle p\rangle\,\psi \equiv \bigvee_{(\sigma,\varphi)\in\mathcal{F}_p} \sigma(\psi) \wedge \varphi$$

In these results, we have extended PDL with arbitrary conjunction and disjunction: finitary or countable-infinitary, depending on the size of $\mathcal{F}_p$.

A disjunction of $\sigma(\psi) \wedge \varphi$ for the (σ,φ) from a finite subset of $\mathcal{F}_p$ will still give a precondition guaranteeing the postcondition for ψ, but it may not be the weakest.

Corollary 2 (Arbitrary Preconditions)**.** For any subset $W \subseteq \mathcal{F}_p$ of symbolic executions, $\left(\bigvee_{(\sigma,\varphi)\in W} \sigma(\psi) \wedge \varphi\right) \Rightarrow \langle p\rangle\,\psi$.

This does not hold for the weakest liberal precondition, seeing as we are taking a conjunction there, and the formula may be *too* weak.

For deterministic programs, and While programs $p \in W_X$ in particular, for which the weakest precondition is given by the diamond modality, Theorem 6 expresses the weakest precondition in terms of the symbolic executions.

9. Related Work

In this section we discuss related interesting research developments regarding semantics of symbolic execution.

In an earlier formal description of symbolic execution [5], De Boer and Bonsangue proved correctness and completeness of symbolic execution with respect to an operational-style semantics modeling concrete execution. In contrast, we relate concrete and symbolic *denotational* semantics (Theorem 2), by semanticizing the symbolic substitutions as concrete state transformers. This semantics is very natural, since substitutions are syntactic objects describing a transformation for each variable. Nevertheless, making this semantics precise is crucial in relating forward state transformation — using the semantics of the substitution — to backward predicate transformation — by applying the substitution syntactically.

De Boer and Bonsangue [5] also view symbolic execution as extracting traces from programs. They extract traces in an operational semantics, whereas we take a denotational approach and define them inductively, directly on the program. In our context, correctness (Theorem 3) of symbolic execution states that the operational semantics computes exactly the elements of the denotational semantics $\mathcal{F}_p$; completeness (Theorem 4) that all trace representations $(\sigma, \varphi) \in \mathcal{F}_p$ will eventually be reached in the operational semantics. In contrast to De Boer and Bonsangue [5], both results stay within the symbolic realm, and no concrete semantics is necessary. The proofs of correctness and completeness cannot be done using syntactic equality: the conjuncts appearing in the corresponding path conditions are different, but equivalent.

Lucanu et al. develop a language-independent coinductive description of symbolic execution [2, 27], based on term rewriting [4], which can be extended into a deductive verification system for *Reachability Logic* [35]. The authors prove correctness properties (called coverage and precision, which are similar to De Boer and Bonsangue's notions of correctness and completeness discussed above) for their symbolic semantics with respect to a concrete semantics. They use derivatives of formulas in matching logic and reachability logic to express semantics as rewrite rules, and show how this approach can be used for tool-based reasoning about programs in the $\mathbb{K}$ framework [33]. In contrast to this term rewriting approach based on forward reasoning, our work is concerned with a compositional, trace-based denotational semantics and we show how symbolic execution can also be used for weakest precondition (backwards) reasoning. Lucanu et al. [2, 27] further consider a construction from concrete to symbolic semantics in terms of transformation steps on the rules: linearization, term abstraction into symbolic variables and explicitation of path conditions. Recent work by the authors [39] propose a rule format to automatically generate both concrete and symbolic semantics with built-in correctness properties in terms of a syncrete bisimulation relation. This line of research on constructing symbolic semantics complements the work in our paper.

Kneuper [26] gives a denotational semantics of symbolic execution based on sets of sequences of symbolic states and a function extending these sequences. Steinhöfel [37, Ch. 3] describes a more general approach based on *concretization* of symbolic states. A similar approach is taken by Porncharoenwase et al. [32], who describe symbolic execution of a Scheme dialect through big-step semantics. These approaches operationalize the exploration of symbolic states. In the case of Kneuper as sets of traces that are incrementally extended, and in the case of Steinhöfel as changing sets of reached states. On the other hand, the present work defines operational symbolic semantics by implicitly constructing a tree of symbolic states and relating this tree to the (non-branching) concrete semantics, leaving the exploration of the tree unspecified.

Several works exist at the intersection of symbolic execution and Coq mechanizations. Correnson and Steinhöfel [11] build on De Boer and Bonsangue's small-step semantics to produce a verified symbolic bug finder. Kløvstad et al. [25] mechanized proofs of compositional correctness and completeness for symbolic execution of parallel programs, but not in a denotational setting.

Owens et al. [31] mechanize what they call a *functional* big-step semantics for a toy language called FOR, which is similar to ours, but uses *for* loops instead of *while* loops, and models assignments as side-effects of expressions. Their functional big-step semantics is essentially identical to our concrete semantics, but we moreover take a perspective on the semantics from a symbolic point of view. Implementing symbolic execution for the FOR language and reason about weakest preconditions should be straightforward.

Nakata and Uustalu [30] explored four different trace-based coinductive operational semantics for the While language: big-step and small-step, functional and relational—all of them only for *concrete* execution, while we include *symbolic*. Similar to our work, the semantics of Nakata and Uustalu are concerned with sequential composition. It is challenging to define denotational semantics for parallel composition operators; concurrency leads to interference, which is difficult to capture compositionally. Recent work by Din et al. [14] combine a locally symbolic big-step semantics with a globally concrete medium-step semantics to obtain a denotational trace semantics for concurrent languages. In their work, the granularity of the denotational, symbolic semantics correspond to the atomic regions of the parallel operator, so parallel composition in fact lazily unfolds and stitches together denotational fragments. Whereas their approach is partly symbolic, the resulting global trace semantics is in fact concrete. Compared to this line of work, the present work can be seen as a functional big-step semantics for symbolic execution, following the terminology of Nakata, Owens and co-authors [30, 31]. We deemed "denotational" more appropriate, as the purpose of our work is to elucidate the *denotation* of the syntactic objects generated in symbolic execution, and to enable compositional reasoning; this has historically been the use of denotational semantics.

The KeY project [1] is a semi-automatic deductive verification tool for Java that employs forward reasoning with symbolic execution for debugging and verification condition generation. KeY's logical framework is based on a dynamic logic for Java with *updates*. Related combinations of symbolic execution and dynamic logic have also recently been developed for, e.g., smart contracts [3, 34] and probabilistic programs [24]. Like our traces, these updates represent finite, deterministic and side effect-free program fragments, which are computed symbolically and applied to formulas. While this line of work is focused on using symbolic execution to compute strongest postconditions, our paper further justifies the use of symbolic execution to compute preconditions by semanticizing symbolic substitutions as state transformers. Gordon and Collavizza [16] discuss the merits of strongest postcondition with forward reasoning compared to weakest precondition with backward reasoning; the authors claim that computing weakest preconditions is simpler, but that strongest postcondition reasoning benefits from being formulated with symbolic execution. Our work shows that weakest precondition reasoning may in fact also benefit from symbolic execution frameworks.

Hoare and Jifeng's Unifying Theories of Programming (UTP) [20] also seek to unify operational, denotational and axiomatic semantics, but in a concrete setting. In their framework programs represent *relations* between concrete states, rather than state transformers. Other work in the UTP project are more closely related to ours. In particular Jifeng [19] develops both denotational and operational semantics from an algebra of programs and shows how to derive a *predicate* transformer semantics and weakest preconditions. Jifeng's approach has recently been mechanized by Mu and Li [29]. However, none of these works define a denotational semantics for symbolic execution.

10. Conclusions and Future Work

We have defined a symbolic denotational semantics as a set of symbolic substitutions along with their respective preconditions; one such pair for every trace of a program. The denotational semantics enabled compositional reasoning, as demonstrated in Lemma 10. An interesting aspect of our work is that this compositionality is not obvious. Indeed, for the substitutions, the semantic composition is the reverse of the syntactic composition, and composition of path conditions uses substitution. We exploit the symbolic execution compositionality results to formally show how symbolic execution computes (weakest) preconditions.

We have mechanized our results in the theorem prover Coq.

We have disallowed program modalities in postconditions and Boolean tests in the program—this is the *poor*-test version of PDL as opposed to the *rich*-test version—since it would invalidate Rule asm in the symbolic operational semantics of Section 7. Indeed, it is unclear what the effect of a symbolic substitution on a formula including a program modality would have to be. Moreover, the resulting symbolic execution would leave us with a program modality in the precondition, and the whole point is to get rid of those. It may be theoretically interesting to explore techniques of symbolically executing modalities in rich-test PDL, but the practical implications are very limited.

The denotational semantics extends easily to more language constructs. For example, other work [40] illustrates the use of a denotational semantics for proof techniques involving *probabilistic* language constructs such as sampling and observe statements. Denotational semantics for such language constructs are straightforward extensions of the semantics presented here.

A highly interesting extension of the work in this paper is to incorporate parallelization; compositional correctness and completeness of a small-step symbolic semantics for parallel programs has recently been mechanized [25]. In a denotational setting, parallelization can be addressed by means of trace semantics and corresponding coinductive techniques (e.g., [38]); furthermore, concurrency is very context-sensitive, which makes assigning a denotational (or functional big-step) semantics challenging. In the future we plan to study a trace-based denotational semantics of symbolic execution, allowing parallelization as well as non-termination.

References

[1] Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (Eds.), 2016. Deductive Software Verification - The KeY Book - From Theory to Practice. volume 10001 of *Lecture Notes in Computer Science*. Springer. URL: https://doi.org/10.1007/978-3-319-49812-6, doi:10.1007/978-3-319-49812-6.

[2] Arusoaie, A., Lucanu, D., Rusu, V., 2013. A generic framework for symbolic execution, in: Erwig, M., Paige, R.F., Wyk, E.V. (Eds.), Proc. 6th International Conference on Software Language Engineering (SLE 2013), Springer. pp. 281–301. doi:10.1007/978-3-319-02654-1_16.

[3] Arvay, B., Doan, T.T.H., Thiemann, P., 2024. A dynamic logic for symbolic execution for the smart contract programming language michelson, in: Aldrich, J., Salvaneschi, G. (Eds.), Proc. 38th European Conference on Object-Oriented Programming(ECOOP 2024), Schloss Dagstuhl - Leibniz-Zentrum für Informatik. pp. 3:1–3:26. doi:10.4230/LIPICS.ECOOP.2024.3.

[4] Baader, F., Nipkow, T., 1998. Term rewriting and all that. Cambridge University Press.

[5] de Boer, F.S., Bonsangue, M.M., 2021. Symbolic execution formally explained. Formal Aspects of Computing 33, 617–636. doi:10.1007/S00165-020-00527-Y.

[6] Cadar, C., Dunbar, D., Engler, D.R., 2008. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs, in: Draves, R., van Renesse, R. (Eds.), Proc. 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08), USENIX Association. pp. 209–224.

[7] Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R., 2006. EXE: automatically generating inputs of death, in: Juels, A., Wright, R.N., di Vimercati, S.D.C. (Eds.), Proc. 13th ACM Conference on Computer and Communications Security (CCS'06), ACM. pp. 322–335. doi:10.1145/1180405.1180445.

[8] Cadar, C., Godefroid, P., Khurshid, S., Pasareanu, C.S., Sen, K., Tillmann, N., Visser, W., 2011. Symbolic execution for software testing in practice: preliminary assessment, in: Taylor, R.N., Gall, H.C., Medvidovic, N. (Eds.), Proc. 33rd International Conference on Software Engineering (ICSE 2011), ACM. pp. 1066–1071. doi:10.1145/1985793.1985995.

[9] Cadar, C., Sen, K., 2013. Symbolic execution for software testing: three decades later. Commun. ACM 56, 82–90. doi:10.1145/2408776.2408795.

[10] Coq Development Team, 2022. The Coq proof assistant. doi:10.5281/zenodo.7313584.

[11] Correnson, A., Steinhöfel, D., 2023. Engineering a formally verified automated bug finder, in: Chandra, S., Blincoe, K., Tonella, P. (Eds.), Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023, ACM. pp. 1165–1176. doi:10.1145/3611643.3616290.

[12] Cousot, P., Cousot, R., 1992. Abstract Interpretation Frameworks. Journal of Logic and Computation 2, 511–547. doi:10.1093/logcom/2.4.511.

[13] Dijkstra, E.W., 1997. A Discipline of Programming. 1st ed., Prentice Hall PTR, USA.

[14] Din, C.C., Hähnle, R., Henrio, L., Johnsen, E.B., Pun, V.K.I., Tapia Tarifa, S.L., 2024. Locally abstract, globally concrete semantics of concurrent programming languages. ACM Trans. Program. Lang. Syst. 46, 3:1–3:58. doi:10.1145/3648439.

[15] Godefroid, P., Klarlund, N., Sen, K., 2005. DART: directed automated random testing, in: Sarkar, V., Hall, M.W. (Eds.), Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05), ACM. pp. 213–223. doi:10.1145/1065010.1065036.

[16] Gordon, M., Collavizza, H., 2010. Forward with Hoare, in: Roscoe, A.W., Jones, C.B., Wood, K.R. (Eds.), Reflections on the Work of C. A. R. Hoare. Springer, pp. 101–121. doi:10.1007/978-1-84882-912-1_5.

[17] de Gouw, S., Rot, J., de Boer, F.S., Bubel, R., Hähnle, R., 2015. OpenJDK's Java.utils.collection.sort() is broken: The good, the bad and the worst case, in: Kroening, D., Pasareanu, C.S. (Eds.), Proc. 27th International Conference on Computer Aided Verification (CAV 2015), Springer. pp. 273–289. doi:10.1007/978-3-319-21690-4_16.

[18] Harel, D., Tiuryn, J., Kozen, D., 2000. Dynamic Logic. MIT Press, Cambridge, MA, USA.

[19] He, J., 2016. A new roadmap for linking theories of programming, in: Bowen, J.P., Zhu, H. (Eds.), Unifying Theories of Programming - 6th International Symposium, UTP 2016, Reykjavik, Iceland, June 4-5, 2016, Revised Selected Papers, Springer. pp. 26–43. doi:10.1007/978-3-319-52228-9_2.

[20] He, J., Hoare, C.A.R., 1998. Unifying theories of programming, in: Orlowska, E., Szalas, A. (Eds.), Participants Copies for Relational Methods in Logic, Algebra and Computer Science, 4th International Seminar RelMiCS, Warsaw, Poland, Septermber 14-20, 1998, pp. 97–99.

[21] Hentschel, M., Bubel, R., Hähnle, R., 2019. The symbolic execution debugger (SED): a platform for interactive symbolic execution, debugging, verification and more. Int. J. Softw. Tools Technol. Transf. 21, 485–513. doi:10.1007/S10009-018-0490-9.

[22] Hoare, C.A.R., 1969. An axiomatic basis for computer programming. Commun. ACM 12, 576–580. doi:10.1145/363235.363259.

[23] Jacobs, B., Rutten, J., 1997. A tutorial on (co) algebras and (co) induction. Bulletin-European Association for Theoretical Computer Science 62, 222–259.

[24] Johnsen, E.B., Kamburjan, E., Pardo, R., Voogd, E., Wasowski, A., 2024. Towards a proof system for probabilistic dynamic logic, in: Jansen, N., Junges, S., Kaminski, B.L., Matheja, C., Noll, T., Quatmann, T., Stoelinga, M., Volk, M. (Eds.), Principles of Verification: Cycling the Probabilistic Landscape - Essays Dedicated to Joost-Pieter

Katoen on the Occasion of His 60th Birthday, Part I, Springer. pp. 322–338. doi:10.1007/978-3-031-75783-9_13.

[25] Kløvstad, Å.A.A., Kamburjan, E., Johnsen, E.B., 2023. Compositional correctness and completeness for symbolic partial order reduction, in: Pérez, G.A., Raskin, J. (Eds.), Proc. 34th International Conference on Concurrency Theory (CONCUR 2023), Schloss Dagstuhl - Leibniz-Zentrum für Informatik. pp. 9:1–9:16. doi:10.4230/LIPICS.CONCUR.2023.9.

[26] Kneuper, R., 1991. Symbolic execution: a semantic approach. Science of computer programming 16, 207–249. doi:10.1016/0167-6423(91)90008-L.

[27] Lucanu, D., Rusu, V., Arusoaie, A., 2017. A generic framework for symbolic execution: A coinductive approach. Journal of Symbolic Computation 80, 125–163. doi:10.1016/J.JSC.2016.07.012.

[28] Mac Lane, S., 2013. Categories for the working mathematician. volume 5. Springer.

[29] Mu, R., Li, Q., 2023. A Coq implementation of the program algebra in Jifeng He's new roadmap for linking theories of programming, in: Bowen, J.P., Li, Q., Xu, Q. (Eds.), Theories of Programming and Formal Methods - Essays Dedicated to Jifeng He on the Occasion of His 80th Birthday, Springer. pp. 395–412. doi:10.1007/978-3-031-40436-8_15.

[30] Nakata, K., Uustalu, T., 2009. Trace-based coinductive operational semantics for While: big-step and small-step, relational and functional styles, in: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (Eds.), Proc. 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2009), Springer. pp. 375–390. doi:10.1007/978-3-642-03359-9_26.

[31] Owens, S., Myreen, M.O., Kumar, R., Tan, Y.K., 2016. Functional big-step semantics, in: Thiemann, P. (Ed.), Proc. 25th European Symposium on Programming (ESOP 2016), Springer. pp. 589–615. doi:10.1007/978-3-662-49498-1_23.

[32] Porncharoenwase, S., Nelson, L., Wang, X., Torlak, E., 2022. A formal foundation for symbolic evaluation with merging. Proc. ACM Program. Lang. 6. doi:10.1145/3498709.

[33] Rosu, G., 2017. $\mathbb{K}$: A semantic framework for programming languages and formal analysis tools, in: Pretschner, A., Peled, D., Hutzelmann, T. (Eds.), Dependable Software Systems Engineering. IOS Press. volume 50 of *NATO Science for Peace and Security Series*, pp. 186–206. doi:10.3233/978-1-61499-810-5-186.

[34] Schiffl, J., Ahrendt, W., Beckert, B., Bubel, R., 2020. Formal analysis of smart contracts: Applying the KeY system, in: Ahrendt, W., Beckert, B., Bubel, R., Hähnle,

R., Ulbrich, M. (Eds.), Deductive Software Verification: Future Perspectives - Reflections on the Occasion of 20 Years of KeY. Springer. volume 12345 of *Lecture Notes in Computer Science*, pp. 204–218. doi:10.1007/978-3-030-64354-6_8.

[35] Stefanescu, A., Ciobâcua, Ş., Mereuta, R., Moore, B.M., Serbanuta, T., Rosu, G., 2014. All-path reachability logic, in: Dowek, G. (Ed.), Proc. Joint International Conference on Rewriting and Typed Lambda Calculi (RTA-TLCA 2014), Springer. pp. 425–440. doi:10.1007/978-3-319-08918-8_29.

[36] Stefanescu, A., Ciobâcua, Ş., Mereuta, R., Moore, B.M., Serbanuta, T., Rosu, G., 2019. All-path reachability logic. Log. Methods Comput. Sci. 15. doi:10.23638/LMCS-15(2:5)2019.

[37] Steinhöfel, D., 2020. Abstract execution: automatically proving infinitely many programs. Ph.D. thesis. Technische Universität Darmstadt. URL: http://tuprints.ulb.tu-darmstadt.de/8540/.

[38] Uustalu, T., 2013. Coinductive big-step semantics for concurrency, in: Yoshida, N., Vanderbauwhede, W. (Eds.), Proc. 6th Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software (PLACES 2013), pp. 63–78. doi:10.4204/EPTCS.137.6.

[39] Voogd, E., Johnsen, E.B., Kløvstad, Å.A.A., Rot, J., Silva, A., 2024. Correct and complete symbolic execution for free, in: Kosmatov, N., Kovács, L. (Eds.), Proc. 19th International Conference on Integrated Formal Methods (IFM 2024), Springer. pp. 237–255. doi:10.1007/978-3-031-76554-4_13.

[40] Voogd, E., Johnsen, E.B., Silva, A., Susag, Z.J., Wąsowski, A., 2023a. Symbolic semantics for probabilistic programs, in: Jansen, N., Tribastone, M. (Eds.), Proc. 20th International Conference on Quantitative Evaluation of Systems (QEST 2023), Springer. pp. 329–345. doi:10.1007/978-3-031-43835-6_23.

[41] Voogd, E., Kløvstad, Å.A.A., Johnsen, E.B., 2023b. Denotational semantics for symbolic execution, in: Ábrahám, E., Dubslaff, C., Tapia Tarifa, S.L. (Eds.), Proc. 20th International Colloquium on Theoretical Aspects of Computing (ICTAC 2023), Springer. pp. 370–387. doi:10.1007/978-3-031-47963-2_22.