






Symbolic Semantics for Probabilistic Programs

Erik Voogd¹, Einar Broch Johnsen¹, Alexandra Silva²,
Zachary J. Susag², and Andrzej Wąsowski³

¹ University of Oslo, Oslo, Norway

² Cornell University, Ithaca, New York, USA

³ IT University of Copenhagen, Copenhagen, Denmark



Abstract. We present a new symbolic execution semantics of probabilistic programs that include observe statements and sampling from continuous distributions. Building on Kozen’s seminal work, this symbolic semantics consists of a countable collection of measurable functions, along with a partition of the state space. We use the new semantics to provide a full correctness proof of symbolic execution for probabilistic programs. We also implement this semantics in the tool `symProb`, and illustrate its use on examples.

1 Introduction

Probabilistic programming languages are designed to make probabilistic computations easier to express for a broader scientific community. They can be used to model behaviour based on data that carries uncertainty or randomness, as found in, e.g., robotics [30], machine learning [12, 23], statistics [11], and cryptography [14]. Besides traditional programming constructs, a key aspect of a probabilistic language is the ability to *sample* random values, in order to represent the uncertainty that occurs in the real world. It is essential that these programming languages have a rigorous foundation, such that correctness and safety properties can be guaranteed when designing and implementing tools for probabilistic program analysis, optimization, and compilation.

Probabilistic semantics was first studied by Kozen [19], using measure theory to relate denotational and operational semantics of imperative probabilistic programs. The *denotational* semantics represents states as probability measures over program variables and programs as measure transformers. It enables one to effectively reason about programs as a whole. The *operational* semantics, on the other hand, is a computation model describing step-by-step computation in terms of a measurable function. A correctness theorem for the operational semantics states that the produced sets of outputs are distributed correctly according to the measure transformer of the denotational semantics.

To enable more detailed reasoning about probabilistic programs, we introduce a new semantics, inspired by symbolic execution techniques. For non-probabilistic programs, symbolic execution [1, 18] has been very successful in program analysis techniques such as debugging, test generation, and verification (e.g., [3, 5–8, 13, 15, 16]). Its appeal arises from the fact that one symbolic execution is an abstraction of possibly infinitely many executions in the concrete state space, that all share a single execution path through the program. Symbolic execution interprets program variables symbolically, such that assignments update a symbolic substitution and conditional statements produce so-called

path conditions that must be true for the program’s input variables to execute each path. Thus, symbolic execution generates pairs consisting of a symbolic substitution and a path condition. If the Boolean path condition holds in some initial state, then running the program is the same as applying the corresponding substitution to the initial state.

We use the new semantics to provide a full correctness proof of symbolic execution of probabilistic programs with respect to a denotational semantics à la Kozen. The correctness proof is highly nontrivial: one semantics deals with the program’s symbolic execution traces, the other interprets programs as measure transformers. Our approach is to first prove a one-to-one correspondence between traces and the elements of our new symbolic semantics. The latter is equivalent to the operational semantics first introduced by Kozen [19], and hence to the denotational semantics. The full details of this proof are included in the extended version of this paper [32].

We consider a language with *observe* statements: in some interpretations of Bayesian inference, a programmer can express to have observed a value as a sample from some discrete or continuous distribution [28]. In others, including the imperative language that we consider, observe statements are given the semantics of asserting the truth of a provided Boolean formula. Observe statements were not studied in Kozen’s work, and an operational semantics for probabilistic programming with observe statements has not been formally defined in that setting. For our correctness proof, therefore, we must extend both denotational and operational semantics with an interpretation of *observe*, and state and prove the corresponding correctness theorem anew.

Intuitively, observe statements in an operational semantics can be modelled by *rejection sampling*. That is, any execution that violates the observed formula is aborted. The probability mass will then reside in execution traces where the observe condition holds, and the probability of the set of traces where it does not hold will be annihilated. We formalize this semantics and prove its correctness with respect to a denotational semantics that interprets observe b as a measure transformer that restricts measures to the set corresponding with the Boolean formula b . This is an interpretation based on Bayes’ theorem.

Contributions and Overview. We start the paper by introducing the language using an example program and discussing the challenges in the technical development (Section 2). The subsequent technical sections of the paper present the following contributions:

- An in-depth description of symbolic execution of probabilistic programs (Section 3), which supports discrete and continuous sampling as well as Bayesian inference through observing Boolean formulas of positive-measure sets. In symbolic execution, sampled values will be represented by symbolic variables.
- As a main contribution we provide a full correctness proof of symbolic execution of probabilistic programs (Section 4) by introducing a new *symbolic* semantics for imperative probabilistic programs and proving correspondence with established semantics (which we extend to support *observe* statements).
- To showcase symbolic execution of probabilistic programs, we have developed the tool `symProb` that performs bounded symbolic execution according to our new symbolic semantics (Section 5). At the end of a symbolic execution, `symProb` reports the path condition, the substitution mapping, and the set of Boolean formulas which correspond to the program’s observe statements. We report on running `symProb` on a series of examples to show how our semantics can be applied.

2 Probabilistic Programming

We introduce the language through an example and then present its grammar.

2.1 Example Program

Consider the probabilistic program on the right, modelling the gender distribution among people taller than 200 centimeters. In the Bernoulli sampling statement `bern(0.51)` for the variable `gender` we assume that 51% of the total population is male. Among men, height is normally distributed with mean 175 and variance 72, and among women with mean 161 and variance 50. The last line conditions the distribution on people taller than 200 centimeters.

```
gender ~ bern(0.51);
if (gender = 1) {
  height ~ norm(175,72);
} else {
  height ~ norm(161,50);
}
observe (height >= 200);
```

Symbolic execution has been very effective in analysis of non-probabilistic programs. The technique builds variable substitutions by analyzing assignments, and resolves conditional branching and iteration by use of nondeterminism. The conditionals are stored under variable substitution as a Boolean formula – called the *path condition* – that needs to be satisfied by the initial state for the corresponding substitution to be a representation of the program. Some problems arise when programs have discrete and continuous sample and observe statements, such as the above.

Consider for example a Bernoulli sampling statement $x_i \sim \text{bern}(e)$. One could introduce additional sets of symbolic variables for sampling statements, but it is practically infeasible to do so for each possible bias e , especially if one allows parameters to be arithmetic expressions. In our approach, we assume a finite number of *primitive* distributions (that is, unparameterized) and encode *parameterized* distributions using these primitives. A Bernoulli sampling statement $x_i \sim \text{bern}(e)$ is then encoded by use of uniform continuous sampling from the unit interval $[0, 1]$, written $x_i \sim \text{rnd}$, as follows:

```
 $x_i \sim \text{rnd}; \text{if } (x_i < t) \{x_i := 1\} \text{ else } \{x_i := 0\}$ 
```

A denotational semantics can be used to show soundness of such encodings.

For Gaussian distributed samples, the primitive distribution used is the *standard* Gaussian distribution, which has mean zero and variance one. To obtain a sample from a Gaussian distribution with mean e_1 and variance e_2 – both arithmetic expressions – a standard Gaussian sample is scaled (multiplied by $\sqrt{e_2}$) and translated (add e_1).

With these encodings, one trace through the example program above has final substitution σ and path condition ϕ , given by

$$\sigma = \{\text{gender} \mapsto 1, \text{height} \mapsto z_0 \sqrt{72} + 175\} \quad \phi \equiv y_0 < 0.51 \wedge 1 = 1$$

Here, y_0 is a symbolic variable that is uniformly distributed over the interval $[0, 1]$. As it turns out, the probability of the path condition as measured by the input measure is the *prior* probability 0.51 of the trace.

The obtained Boolean formula from *observe* statements in this symbolic execution is

$$\psi \equiv z_0 \cdot \sqrt{72} + 175 \geq 200,$$

where z_0 is a symbolic variable representing a standard Gaussian sample. Measuring the set represented by this Boolean formula with the input measure yields exactly the *likelihood* of the model we have in mind. To keep the prior and the likelihood separate, we collect Boolean conditions under *observe* statements in what we coin the *path observations*.

This paper formally describes the procedure above for general probabilistic programs, and proves correctness with respect to a denotational semantics. Building on work by Kozen [19], we choose to interpret substitutions from symbolic execution as measurable functions on the value space, and path conditions and path observations as measurable sets of values for the program variables. These can then be compared against Kozen’s denotational semantics, where program states are subprobability measures over the domain of values for the program variables, and a program p is a transformer of input to output measures $\llbracket p \rrbracket: \mathcal{M}(\mathbb{R}^n) \rightarrow \mathcal{M}(\mathbb{R}^n)$. The use of measure theory is unavoidable in order to handle continuous random variables. For example the semantics of `height \sim norm(175,72)`, given an input $\mu \in \mathcal{M}(\mathbb{R})$, is the measure:

$$\llbracket \text{height} \sim \text{norm}(175,72) \rrbracket(\mu) : A \mapsto \gamma_{175,72}(A)$$

Here, $\gamma_{175,72}$ denotes the Gaussian measure with expected value 175 and variance 72.

After defining the language formally in Section 2.2, we provide a detailed description of symbolic execution of probabilistic programs in Section 3. There, towards a correctness proof, we also introduce the *symbolic semantics*. This is a *big-step* semantics for symbolic execution—this is stated in Theorem 4. Section 4 is aimed at proving correctness with respect to a measure-transforming semantics. The correctness statement (Theorem 5) says that the sum of probabilities of all symbolic execution paths, as measured by the input measure, expressed using the final substitutions, path conditions, and path observations, is the same as the probability under the denotational semantics. The technical contribution is concluded with a proof of concept tool that implements symbolic execution and we perform some experiments with it (Section 5).

2.2 Language

To express programs like the above example, we consider a language that contains basic imperative constructs (assignments, sequential composition, conditionals, and loops) together with constructs to manipulate random variables (sampling) and observe statements that allow conditioning on an observation defined by a Boolean condition:

$\mathbb{E} \ni e ::= q \in \mathbb{Q}$	$\mathbb{B} \ni b ::= \text{False}$	$\mathbb{P} \ni p ::= \text{Skip}$
x_i	True	$x_i := e$
$\text{op}(e_1, \dots, e_{\# \text{op}})$	$e \diamond e$	$x_i \sim \text{rnd}$
	$b \parallel b$	$\text{observe } b$
	$b \&\& b$	$p \ddagger p$
$x_i \in \mathcal{X}$	$!b$	$\text{if } b \text{ } p \text{ else } p$
$\diamond \in \{<, \leq, =, ! =, \geq, >\}$		$\text{while } b \text{ } p$

It defines *expressions* $e \in \mathbb{E}$ over program variables $x_i \in \mathcal{X}$ (where $i \in \mathbb{N}$) and constants $q \in \mathbb{Q}$ using operators op of arity $\# \text{op}$. Expressions $e \in \mathbb{E}$ are interpreted as functions

$\bar{e} : \mathbb{R}^n \rightarrow \mathbb{R}$ that compute the value of e given a valuation $v : \{0, 1, \dots, n-1\} \rightarrow \mathbb{R}$ of the program variables. Formally, op is a function $\overline{\text{op}} : \mathbb{R}^m \rightarrow \mathbb{R}$ where $m = \#\text{op}$, and

$$\bar{q}(v) = q, \quad \bar{x}_i(v) = v(i), \quad \overline{\text{op}(e_1, \dots, e_{\#\text{op}})}(v) = \overline{\text{op}}(\bar{e}_1(v), \dots, \bar{e}_{\#\text{op}}(v))$$

The grammar also defines *Boolean expressions* $b \in \mathbb{B}$ that can relate expressions and apply the standard Boolean operators. We slightly overload notation and interpret Boolean expressions b as subsets \bar{b} of \mathbb{R}^n where the formula holds. For example, $\overline{\text{True}} = \mathbb{R}^n$ and if $b = e_1 \diamond e_2$ then

$$\bar{b} = \{v \in \mathbb{R}^n \mid \bar{e}_1(v) \diamond \bar{e}_2(v)\}.$$

Boolean conjunction, disjunction, and negation are respectively interpreted as set intersection, union, and complement in \mathbb{R}^n .

The Boolean expressions are used in the *if* and *while* conditions and in observe statements of *program statements* $p \in \mathbb{P}$. Whenever some program $p \in \mathbb{P}$ is fixed, there is also a fixed amount of n variables.

Sampling statements $x_i \sim \text{rnd}$ draw uniform random samples from the unit interval $[0, 1]$. For presentation purposes, we sample from only one primitive distribution in the formal grammar, and we do so at the level of statements rather than in expressions. Expressions can be made probabilistic, however, by first sampling and then using the variable in a (Boolean) expression. We also stress again that the language can be extended to support sampling from a multitude of other primitive distributions, both discrete and continuous, without affecting any of the results in this paper.

With the extensions described in Section 2.1, the probabilistic program presented there is in the language generated by our grammar.

3 Symbolic Execution

The inductive rules that define a transition system implementing symbolic execution of probabilistic programs are presented in fig. 1. The *symbolic states* in this transition system are quintuples consisting of the following data: (i) a *program* p that is to be executed; (ii) a *substitution* σ of program variables to symbolic expressions (defined shortly), which captures past program behavior; (iii) a *sampling index*¹ $k \in \mathbb{N}$; (iv) the *path condition* as a precondition for this symbolic trace; and (v) the *path observation* as a means to bookkeep which states will be accepted by *observe* statements throughout execution. Progression of the system depends only on the program syntax (i). Below we provide a detailed description of the components (ii)-(v). The substitutions (ii) and path conditions (iv) follow established methods from symbolic execution [3]; the notion of path observation (v) is novel.

The program `Skip` cannot make a transition and represents the *terminated* program. The system is nondeterministic due to the pairs of rules for *if* and *while* statements. The system has infinite symbolic traces due to Rule `iter-T`. Customarily, $\xrightarrow{*}$ denotes the reflexive-transitive closure of \longrightarrow .

¹ One for each primitive distribution: we use one in this paper for presentation purposes. Two sampling indices are used in the tool `symProb`.

$\frac{}{(x_i := e, \sigma, k, \phi, \psi) \longrightarrow (\text{Skip}, \sigma[x_i \mapsto \sigma e], k, \phi, \psi)}$	asgn
$\frac{}{(x_i \sim \text{rnd}, \sigma, k, \phi, \psi) \longrightarrow (\text{Skip}, \sigma[x_i \mapsto y_k], k + 1, \phi, \psi)}$	smp1
$\frac{}{(\text{observe } b, \sigma, k, \phi, \psi) \longrightarrow (\text{Skip}, \sigma, k, \phi, \psi \wedge \sigma b)}$	obs
$\frac{}{(\text{Skip} \ ; \ p, \sigma, k, \phi, \psi) \longrightarrow (p, \sigma, k, \phi, \psi)}$	seq-0
$\frac{(p, \sigma, k, \phi, \psi) \longrightarrow (p', \sigma', k', \phi', \psi')}{(p \ ; \ q, \sigma, k, \phi, \psi) \longrightarrow (p' \ ; \ q, \sigma', k', \phi', \psi')}$	seq-n
$\frac{}{(\text{if } b \ p_1 \ \text{else } p_2, \sigma, k, \phi, \psi) \longrightarrow (p_1, \sigma, k, \phi \wedge \sigma b, \psi)}$	if-T
$\frac{}{(\text{if } b \ p_1 \ \text{else } p_2, \sigma, k, \phi, \psi) \longrightarrow (p_2, \sigma, k, \phi \wedge \sigma!b, \psi)}$	if-F
$\frac{}{(\text{while } b \ p, \sigma, k, \phi, \psi) \longrightarrow (\text{Skip}, \sigma, k, \phi \wedge \sigma!b, \psi)}$	iter-F
$\frac{}{(\text{while } b \ p, \sigma, k, \phi, \psi) \longrightarrow (p \ ; \ \text{while } b \ p, \sigma, k, \phi \wedge \sigma b, \psi)}$	iter-T

Fig. 1: Inductive transition rules for symbolic execution

3.1 Symbolic Substitutions

To capture the behavior of a symbolic trace, a *substitution* assigns to every program variable a *symbolic expression*. Symbolic expressions are generated by the following grammar (recall that $\#_{\text{op}}$ denotes the arity of op):

$$\mathbb{S}\mathbb{E} \ni \mathbb{E} ::= i \mid q \in \mathbb{Q} \mid x_i \in \mathcal{X} \mid y_k \in \mathcal{Y} \mid \text{op}(E, \dots, E_{\#_{\text{op}}})$$

$\mathbb{S}\mathbb{E}$ extends \mathbb{E} with the set $\mathcal{Y} = \{y_0, y_1, \dots\}$ as base cases, representing the random samples that may be drawn during execution. When the language samples from more than one primitive distribution, a set of symbolic variables $\mathcal{Y}_{\mathcal{D}}$ is used for every primitive distribution \mathcal{D} . For each such distribution \mathcal{D} , one will need a sampling index $k_{\mathcal{D}}$ in the transition system. We use $\mathcal{Z} = \{z_0, z_1, \dots\}$ for symbolic variables representing standard normal samples in our examples and in the tool.

Thus, formally, substitutions are maps $\sigma : \mathcal{X} \rightarrow \mathbb{S}\mathbb{E}$. We sometimes write $\sigma(i)$ in lieu of $\sigma(x_i)$. The *updated* substitution $\sigma[i \mapsto E]$ denotes the substitution σ' such that $\sigma'(j) = \sigma(j)$ for $j \neq i$ and $\sigma'(i) = E$. Any substitution σ inductively extends to expressions $e \in \mathbb{E}$ by $\sigma(\text{op}(e_1, \dots, e_{\#_{\text{op}}})) := \text{op}(\sigma(e_1), \dots, \sigma(e_{\#_{\text{op}}}))$.

If symbolic execution is to conform with a denotational semantics, the symbolic substitutions have to be interpreted as concrete state transformers. For this, first, symbolic *expressions* are interpreted as functions $\mathbb{R}^{n+\omega} \rightarrow \mathbb{R}$, where the first n arguments are

the values of the program variables, and the rest is an infinite stream of samples to be drawn. Here, one may use extra streams of sample spaces for any additional primitive distributions in the language. The mapping $|\cdot| : \mathbb{R}^{n+\omega} \rightarrow \mathbb{R}$ equips symbolic expressions with a formal interpretation as follows:

$$\begin{aligned} |q| : \rho \mapsto q, & & |x_i| : \rho \mapsto \rho_i, & & |y_k| : \rho \mapsto \rho_{n+k}, \\ |\text{op}(E_1, \dots, E_{\# \text{op}})| : \rho \mapsto \overline{\text{op}}(|E_1|(\rho), \dots, |E_{\# \text{op}}|(\rho)) \end{aligned}$$

The symbols y_k thus pick the k -th sample available in \mathbb{R}^ω . Symbolic expressions free of y_k are just program expressions, and their interpretations agree in the following way:

Lemma 1 (Substitution lemma for expressions). *For all $e \in \mathbb{E}$, $|e|(\rho) = \bar{e}(\rho|_n)$.*

This interpretation of $E \in \mathbb{SE}$ as a function $\mathbb{R}^{n+\omega} \rightarrow \mathbb{R}$ extends naturally to symbolic substitutions $\sigma \in \mathbb{SE}^n$ as functions $\mathbb{R}^{n+\omega} \rightarrow \mathbb{R}^n$, by mere point-wise application after substitution. However, since we want to compose substitutions (their interpretations) due to sequencing, the codomain must be extended from \mathbb{R}^n to $\mathbb{R}^{n+\omega}$:

Definition 2 (Interpretation of symbolic substitutions). *Let $\sigma : \mathcal{X} \rightarrow \mathbb{SE}$ be a substitution and $k \in \mathbb{N}$ a sampling index. The interpretation of σ at sampling index k , denoted $|\sigma|_k$, is the function $\mathbb{R}^{n+\omega} \rightarrow \mathbb{R}^{n+\omega}$ defined by*

$$|\sigma|_k : \rho \mapsto (|\sigma(x_0)|(\rho), \dots, |\sigma(x_{n-1})|(\rho), \rho_{n+k}, \rho_{n+k+1}, \rho_{n+k+2}, \dots)$$

The first n values of $|\sigma|_k(\rho)$ are just pointwise applications of $|\cdot|$. The remaining elements of $|\sigma|_k(\rho)$ are the same as those of ρ , but left-shifted k positions. This formalizes the idea that $|\sigma|_k$ has already drawn k samples.

3.2 Path Conditions and Path Observations

The Boolean formulas for the condition in *branching* and *iteration* statements are aggregated under substitution to build the *path condition* of a symbolic trace. The path condition represents the unique part of the input space that triggers the symbolic trace. Similarly, *observed* Boolean formulas under substitution make up the *path observation*, and represent the part of the input space that will lead to acceptance of *observe* statements in this trace.

Path conditions and observations are expressed as *symbolic Boolean expressions*:

$$\mathbb{SB} \ni \mathbb{B} ::= \perp \mid \top \mid E \diamond E \mid \mathbb{B} \vee \mathbb{B} \mid \mathbb{B} \wedge \mathbb{B} \mid \neg \mathbb{B}$$

Substitutions $\sigma : \mathcal{X} \rightarrow \mathbb{SE}$ extend through $\mathbb{E} \rightarrow \mathbb{SE}$ further to Booleans, as in $\mathbb{B} \rightarrow \mathbb{SB}$, in a completely trivial way. For example, $\sigma(e_1 \diamond e_2 \ \&\& \ \text{True}) = \sigma(e_1) \diamond \sigma(e_2) \wedge \top$.

Path conditions and path observations – their symbolic Boolean expressions – are interpreted as subsets of $\mathbb{R}^{n+\omega}$ where the formula is satisfied. The mapping $|\cdot|$ equips symbolic Boolean expressions with a formal interpretation as follows:

$$\begin{aligned} |\perp| &= \emptyset, & |\mathbb{B}_1 \vee \mathbb{B}_2| &= |\mathbb{B}_1| \cup |\mathbb{B}_2|, \\ |\top| &= \mathbb{R}^{n+\omega}, & |\mathbb{B}_1 \wedge \mathbb{B}_2| &= |\mathbb{B}_1| \cap |\mathbb{B}_2|, \\ |\neg \mathbb{B}| &= \mathbb{R}^{n+\omega} \setminus |\mathbb{B}|, & |E_1 \diamond E_2| &= \{\rho \in \mathbb{R}^{n+\omega} \mid |E_1|(\rho) \diamond |E_2|(\rho)\}. \end{aligned}$$

Here we overload notation of $|\cdot|$ to use it for both expressions and Boolean expressions.

3.3 Final Configurations

Let σ_0 denote the *initial* substitution $\{x_i \mapsto x_i\}_{x_i \in \mathcal{X}}$. We call the quadruple (σ, k, ϕ, ψ) the (symbolic) *configuration* of a state $(p, \sigma, k, \phi, \psi)$ and $(\sigma_0, 0, \top, \top)$ is the *initial configuration*. The configurations we are mostly interested in result from a finite symbolic execution trace starting from the initial configuration:

$$\Gamma_p := \{(\sigma, k, \phi, \psi) \mid (p, \sigma_0, 0, \top, \top) \xrightarrow{*} (\text{Skip}, \sigma, k, \phi, \psi)\}$$

is called the set of *final configurations* of p .

For a program $p \in \mathbb{P}$ and a configuration $(\sigma, k, \phi, \psi) \in \Gamma_p$, p transforms inputs $\rho \in |\phi| \cap |\psi|$ to the output $|\sigma|_k(\rho)$. That is, p behaves like $|\sigma|_k$ on $|\phi| \cap |\psi|$. Execution of p on inputs from $|\phi| \setminus |\psi|$ leads to an unsatisfied *observe* statement.

Example 3. Consider the example program in Section 2.1 (call it p) that models the gender distribution among people taller than 200 centimeters. Due to Bernoulli sampling, it contains an additional *if* statement hidden in the encoding. The program has thus *four* symbolic traces; their respective final configurations $\gamma_1, \gamma_2, \gamma_3, \gamma_4 \in \Gamma_p$ are:

Final	Substitution σ	k_y	k_z	Path condition ϕ	Path observation ψ
γ_1	$\{g \mapsto 1, h \mapsto z_0\sqrt{72} + 175\}$	1	1	$y_0 < 0.51 \wedge 1 = 1$	$z_0\sqrt{72} + 175 \geq 200$
γ_2	$\{g \mapsto 1, h \mapsto z_0\sqrt{50} + 161\}$	1	1	$y_0 < 0.51 \wedge 1 \neq 1$	$z_0\sqrt{50} + 161 \geq 200$
γ_3	$\{g \mapsto 0, h \mapsto z_0\sqrt{72} + 175\}$	1	1	$y_0 \geq 0.51 \wedge 0 = 1$	$z_0\sqrt{72} + 175 \geq 200$
γ_4	$\{g \mapsto 0, h \mapsto z_0\sqrt{50} + 161\}$	1	1	$y_0 \geq 0.51 \wedge 0 \neq 1$	$z_0\sqrt{50} + 161 \geq 200$

The final configurations γ_2 and γ_3 have unsatisfiable path conditions. Path conditions represent the *priors* in the model and path observations represent *likelihoods*.

3.4 Symbolic Semantics

Now we introduce symbolic semantics for probabilistic programs with the main purpose of proving correctness of symbolic execution, which we just described. The new semantics is a set of functions, and can in fact be considered a denotational semantics for symbolic execution of probabilistic programs. Defined directly on the syntax of programs, the semantics consists of the interpretations of all final symbolic configurations—this is Theorem 4 below.

For a triple (F, B, \mathcal{O}) in the definition below, F is the final substitution of a trace, B is the path condition, and \mathcal{O} is the path observation. Recall that $\mathcal{B}(X)$ is the Borel σ -algebra of X and let $\mathcal{B}(X, Y)$ denote the space of Borel measurable functions $X \rightarrow Y$. Then, for programs $p \in \mathbb{P}$ in n variables, the sets

$$\mathcal{F}_p \subset \mathcal{B}(\mathbb{R}^{n+\omega}, \mathbb{R}^{n+\omega}) \times \mathcal{B}(\mathbb{R}^{n+\omega}) \times \mathcal{B}(\mathbb{R}^{n+\omega})$$

are defined inductively on the structure of p as follows:

- For *inaction* Skip, the state remains unaltered and there is no restriction on the path condition or the path observation:

$$\mathcal{F}_{\text{skip}} := \{(\rho \mapsto \rho, \mathbb{R}^{n+\omega}, \mathbb{R}^{n+\omega})\}$$

- An *assignment* has no restriction on the precondition, but the state is updated according to the assignment:

$$\mathcal{F}_{x_i := e} := \{(\rho \mapsto \rho[i \mapsto \bar{e}(\rho|_n)], \mathbb{R}^{n+\omega}, \mathbb{R}^{n+\omega})\}$$

Only the first n values $\rho|_n$ of the state ρ are needed to evaluate e , and $\rho[i \mapsto a]$ denotes the state ρ' where $\rho'(j) = \rho(j)$ if $j \neq i$ and $\rho'(i) = a$.

- When *sampling* we also merely perform an appropriate state update:

$$\mathcal{F}_{x_i \sim \text{rnd}} := \{(\rho \mapsto \text{sample}_i(\rho), \mathbb{R}^{n+\omega}, \mathbb{R}^{n+\omega})\}$$

Here, $\text{sample}_i(\rho)$ is an updated stream ρ' that has drawn one sample:

$$\text{sample}_i(\rho) = (\rho_0, \dots, \rho_{i-1}, \rho_n, \rho_{i+1}, \dots, \rho_{n-1}, \rho_{n+1}, \rho_{n+2}, \dots)$$

- *Observing* a Boolean formula only updates the path observation:

$$\mathcal{F}_{\text{observe } b} := \{(\rho \mapsto \rho, \mathbb{R}^{n+\omega}, \bar{b} \times \mathbb{R}^\omega)\}$$

- When *sequencing* two programs p_1 and p_2 , range over all pairs of executions $(F_1, B_1, \mathcal{O}_1) \in \mathcal{F}_{p_1}$ and $(F_2, B_2, \mathcal{O}_2) \in \mathcal{F}_{p_2}$ and compose them. The first path condition B_1 should be satisfied and, after executing the first component F_1 , the second path condition B_2 should be satisfied (and sim. for the path observation):

$$\mathcal{F}_{p_1 \& p_2} := (F_2 \circ F_1, B_1 \cap F_1^{-1}[B_2], \mathcal{O}_1 \cap F_1^{-1}[\mathcal{O}_2]) \mid (F_i, B_i, \mathcal{O}_i) \in \mathcal{F}_{p_i}, i = 1, 2\}$$

- The two branches of an *if* statement are put together in a binary union of sets. The path conditions are updated accordingly ($-^c$ denotes complement):

$$\begin{aligned} \mathcal{F}_{\text{if } b \text{ p else } q} := & \{(F, B \cap (\bar{b} \times \mathbb{R}^\omega), \mathcal{O}) \mid (F, B, \mathcal{O}) \in \mathcal{F}_p\} \\ & \cup \{(F, B \cap (\bar{b}^c \times \mathbb{R}^\omega), \mathcal{O}) \mid (F, B, \mathcal{O}) \in \mathcal{F}_q\} \end{aligned}$$

- In a *while* statement, the union is over every possible number of iterations m . For $m = 0$, the behavior is that of Skip and the precondition is the negation of the Boolean formula. Every next number $m + 1$ of loop iterations takes all possible executions of m iterations, pre-composes all possible additional iterations, and updates the preconditions accordingly:

$$\mathcal{F}_{\text{while } b \text{ p}} := \bigcup_{m=0}^{\infty} \mathbb{F}_{b,p}^m \{(v \mapsto v, \bar{b}^c, \mathbb{R}^{n+\omega})\},$$

where $\mathbb{F}_{b,p}^m$ denotes m applications of the mapping $\mathbb{F}_{b,p}$ from $\mathcal{B}(\mathbb{R}^{n+\omega}, \mathbb{R}^{n+\omega}) \times \mathcal{B}(\mathbb{R}^{n+\omega}) \times \mathcal{B}(\mathbb{R}^{n+\omega})$ to itself that pre-composes an additional iteration of the loop:

$$\begin{aligned} \mathbb{F}_{b,p} : \mathcal{F} \mapsto & \{(F \circ F_q, (\bar{b} \times \mathbb{R}^\omega) \cap B_q \cap F_q^{-1}[B], \mathcal{O}_q \cap F_q^{-1}[\mathcal{O}]) \\ & \mid (F, B, \mathcal{O}) \in \mathcal{F}, (F_q, B_q, \mathcal{O}_q) \in \mathcal{F}_q\} \end{aligned}$$

The sets \mathcal{F}_p form a big-step semantics for symbolic execution of probabilistic programs:

Theorem 4. *For any program $p \in \mathbb{P}$, there is a one-to-one correspondence between final configurations $(\sigma, k, \phi, \psi) \in \Gamma_p$ and triples $(F, B, \mathcal{O}) \in \mathcal{F}_p$ such that $F = |\sigma|_k$, $B = |\phi|$, and $\mathcal{O} = |\psi|$.*

In this correspondence, we consider all final configurations $(\sigma, k, \phi, \psi) \in \Gamma_p$ with unsatisfiable path condition, i.e., $|\phi| = \emptyset$, equivalent. Similarly, all triples (F, B, \mathcal{O}) for which $B = \emptyset$ are considered equivalent.

4 Correctness

The symbolic execution engine described in the previous section is now proven correct with respect to a denotational semantics. Following Kozen [19], probabilistic programs are mappings of measures on the Borel measurable space of \mathbb{R}^n , where n is the number of program variables. Observe statements were not considered in his work. They *have* been studied [4] in this context of measure transformer semantics, but in the absence of unbounded loops.

To be fully precise, we need to recall some definitions from measure theory. A *measurable space* (X, Σ) is a set X equipped with a σ -algebra Σ , i.e., a set $\Sigma \subseteq \mathcal{P}(X)$ that (i) contains the emptyset, (ii) is closed under complements in X , and (iii) is closed under countable union. Elements of Σ are called *measurable sets*. The *Borel* σ -algebra $\mathcal{B}(X)$ is the one generated by the open sets of X . Whenever we say that functions or sets are measurable, we mean with respect to the Borel σ -algebras. A *(sub)probability measure* on (X, Σ) is a function $\mu : \Sigma \rightarrow [0, 1]$ such that $\mu(\emptyset) = 0$ and $\mu(\bigcup_{i \in I} A_i) = \sum_{i \in I} \mu(A_i)$ for any countable disjoint family of sets $(A_i)_{i \in I} \subseteq \Sigma$. We let $\mathcal{M}(X)$ denote the set of measures on the Borel σ -algebra of X ; this makes $\mathcal{M}(\mathbb{R}^n)$ the state space in the denotational semantics.

The denotational semantics of a program p is a function $\llbracket p \rrbracket : \mathcal{M}(\mathbb{R}^n) \rightarrow \mathcal{M}(\mathbb{R}^n)$, pushing a measure corresponding to the current program state forward, according to the statement being interpreted. The inductive definition is as follows:

- *Skip* does not change the given measure: $\llbracket \text{Skip} \rrbracket(\mu) = \mu$.
- For an *assignment*, let α_e^i be the function that updates the i -th value appropriately:

$$\alpha_e^i : \mathbb{R}^n \rightarrow \mathbb{R}^n, \quad (x_1, \dots, x_n) \mapsto (x_1, \dots, x_{i-1}, \bar{e}(x_1, \dots, x_n), x_{i+1}, \dots, x_n)$$

This function is measurable, so its preimages can be measured by μ . Thus, the semantics of assignments as *pushforward* measures $\llbracket x_i := e \rrbracket(\mu) := \mu \circ (\alpha_e^i)^{-1}$ is well-defined. Intuitively, the measure $\llbracket x_i := e \rrbracket(\mu)$ measures with μ the set of values in \mathbb{R}^n that lead to appropriately updated values for the i -th variable.

- *Sampling* updates the measure component of the corresponding variable with the distribution measure λ to be sampled from:

$$\llbracket x_i \sim \text{rnd} \rrbracket(\mu) : A_1 \times \dots \times A_n \mapsto \mu(A_1 \times \dots \times A_{i-1} \times \mathbb{R} \times A_{i+1} \times \dots \times A_n) \cdot \lambda(A_i).$$

For λ , we use the Lebesgue measure on the unit interval for uniform continuous sampling. Other measures can be used for other primitive sampling statements. The measure $\llbracket x_i \sim \text{rnd} \rrbracket(\mu)$ here is defined on the rectangles of \mathbb{R}^n , and by Carathéodory's extension theorem, defines a unique measure on all of \mathbb{R}^n .

- When *observing*, we restrict the measure to the observed measurable set. For measurable $B \subseteq \mathbb{R}^n$, let $e_B(\mu)$ denote the subprobability measure $A \mapsto \mu(A \cap B)$. Then $\llbracket \text{observe } b \rrbracket(\mu) := e_{\bar{b}}(\mu)$. Note that this measure is not normalized.
- *Sequencing* is just composition: $\llbracket p \ ; \ q \rrbracket = \llbracket q \rrbracket \circ \llbracket p \rrbracket$.
- *Branching* restricts the measure of one branch (resp. the other) to the measurable subset where the condition is true (resp. false). Formally,

$$\llbracket \text{if } b \ p_1 \ \text{else } p_2 \rrbracket(\mu) = (\llbracket p_1 \rrbracket \circ e_{\bar{b}})(\mu) + (\llbracket p_2 \rrbracket \circ e_{\bar{\bar{b}}})(\mu)$$

This sum of measures is setwise. The measure is first restricted by $e_{\bar{b}}$ (or $e_{\bar{\bar{b}}}$) to a subprobability measure over the part of the state space where the condition is true (or false) and then passed on to be transformed by $\llbracket p_1 \rrbracket$ (or $\llbracket p_2 \rrbracket$).

- Interpreting *iterations* as repeated unfolding of if statements, the infinite sum

$$\llbracket \text{while } b \ p \rrbracket(\mu) = \left(\sum_{m=0}^{\infty} e_{\bar{b}} \circ (\llbracket p \rrbracket \circ e_{\bar{b}})^m \right)(\mu)$$

describes the semantics of while loops.

Now $\llbracket p \rrbracket$ is a positive operator of norm at most one for all p defined by the grammar in Section 2.2. This means that any subprobability measure is mapped to a subprobability measure [2]. Without while and observe statements, this norm would be exactly one. Intuitively, this is because programs would then always terminate, and no probability mass would ever be lost either by diverging while loops or by conditional probability. The measure semantics is not normalized.

Recall that Γ_p is the set of final configurations of symbolic execution of a program p .

Theorem 5 (Correctness of Symbolic Execution of Probabilistic Programs). *Let $p \in \mathbb{P}$ be a program, $\mu \in \mathcal{M}(\mathbb{R}^n)$ a distribution measure over the input variables, and $A \subseteq \mathbb{R}^n$ a measurable set. Then*

$$\llbracket p \rrbracket(\mu)(A) = \sum_{(\sigma, k, \phi, \psi) \in \Gamma_p} (\mu \otimes \lambda^\omega)(|\sigma|_k^{-1}[A \times \mathbb{R}^\omega] \cap |\phi| \cap |\psi|)$$

A friendly reminder that the measure λ was some chosen measure implicitly used in the denotational semantics; λ^ω denotes the product measure of infinitely many copies of it.

Proof (Sketch). For every program p , there is a function f_p such that

- $(\mu \otimes \lambda^\omega)(f_p^{-1}[A \times \mathbb{R}^\omega]) = \llbracket p \rrbracket(\mu)(A)$, and
- $(\mu \otimes \lambda^\omega)(f_p^{-1}[A \times \mathbb{R}^\omega]) = \sum_{(F, B, \mathcal{O}) \in \mathcal{F}_p} (\mu \otimes \lambda^\omega)(F^{-1}[A \times \mathbb{R}^\omega] \cap B \cap \mathcal{O})$.

The function f_p is basically the operational semantics defined in [19], but extended to an observe construct that rejects unsatisfied observed formulas. The theorem is proven by chaining these two equalities and applying Theorem 4. See [32] for the full proof.

Case Study	Number of Paths		Samples	Lines	Time (sec.)
	Actual	Discarded			
BurglarAlarm	4	12	4	26	0.31
DieCond	20	0	20	17	2.15
Grass	28	36	6	21	0.80
MurderMystery	2	2	2	12	0.06
NeighborAge	4	4	4	14	0.10
NeighborBothBias	7	5	4	19	0.24
NoisyOr	256	–	8	37	2.33
Piranha	3	1	2	12	0.07
Random Z2 Walk (1)	16	–	4	14	0.19
Random Z2 Walk (2)	52	–	6	14	0.60
Random Z2 Walk (4)	712	–	10	14	8.37
Random Z2 Walk (8)	159,436	–	18	14	2167.90
SecuritySynthesis	1	0	8	14	0.04
TrueSkillFigure9	1	0	9	20	0.03
TwoCoins	3	1	2	6	0.05

Table 1: Performance metrics for `symProb` on a series of case studies. For the “Random Z2 Walk (i)” cases, i refers to the number of times the main `while` loop was unrolled.

5 Implementation and Experiments

We have developed `symProb` as a prototype implementation of the symbolic execution technique presented in Section 3². `symProb` takes as input a probabilistic program written in the language described in Section 2.2 (up to some natural imperative-style syntactic additions such as keywords for *else* and delimiter use of parentheses and braces). `symProb` performs *bounded* symbolic execution, meaning that all while loops are unrolled a finite number of times, to ensure termination. Finite loops are fully unrolled while all other loops are unrolled a configurable, yet fixed, number of times. `symProb` reports the final configuration: the final substitution σ , the amount of samples, the path condition ϕ , and the path observation ψ . All symbolic configurations reported by `symProb` are expressed as uninterpreted symbolic expressions. `symProb` is written in Rust in around 2,000 lines of code and uses Z3 [22] for real numbers to determine branch satisfiability.

We have executed `symProb` on a series of examples sourced from PSI [10], R2 [24], “Fun” programs from Infer.NET [21], and Barthe et al. [2]. We summarize our findings in Table 1. All the experiments were done on a machine with 3.3GHz Intel Core i7-5820K and 32 GB of RAM, running Linux 6.3.2-arch1-1.

One feat of symbolic execution of probabilistic programs that `symProb` implements, is that paths with an unsatisfiable *observe* statement can be filtered out, since they have zero likelihood in the probabilistic model. `symProb` therefore *discards* such paths.

For each experiment, we report the following metrics: the number of traces explored, the number of traces which were discarded due to a failed observe statement (‘–’ if there were no observe statements in the program), the maximum number of random samples drawn, the number of lines of code, and the time `symProb` took to explore all paths.

² Source code for `symProb` and all experiments are available on Zenodo [31].

We make a few general observations about the results. Outside of the DieCond, Grass, NoisyOr, and Random Z2 Walk (4,8) examples, symProb terminates in under a second. While these are relatively small examples, they still showcase a range of path counts and number of random samples drawn. Additionally, of the case studies that had observe statements present, many paths were discarded due to reaching an unsatisfiable observe statement. The Grass case study, for example, which involves six Bernoulli samples about weather conditions and such, had 36 paths discarded. In Section 7, we discuss how we might utilize this information in future work to optimize the performance of probabilistic programs with observe statements.

6 Related Work

Symbolic transition systems as described in this work have recently been formalized in a non-probabilistic setting by de Boer and Bonsangue [3]. From that starting point, this work extends to sampling in probabilistic programming by incorporating symbolic random variables $\{y_0, y_1, \dots\}$ in the symbolic substitutions and path conditions. Furthermore, we introduced *path observations* to keep track of observe statements.

Denotational semantics for Bayesian inference is an active research area [9, 17, 27, 28]. Mostly, the focus has been on *discrete* probabilistic programs [17, 25], whereas we support sampling of both discrete *and* continuous distributions, thereby generalizing the probabilistic choice based on discrete sampling used in these works.

Staton [27, 28] describes *observe-from* statements in his denotational semantics as an encoding for a *score* construct used to attach a likelihood scoring to an execution trace. The construct observe e from \mathcal{D} , where \mathcal{D} is some distribution, is then sugar for score $f_{\mathcal{D}}(e)$, where $f_{\mathcal{D}}$ is the probability density function for the distribution \mathcal{D} . The score value in that semantics is therefore akin to the probability measure of our path observation. Staton’s work considers language semantics of a functional nature, where the difficulty mainly lies in handling higher-order functions. In an imperative language, this is generally not a major concern. This allows us to consider the more general construct of observing Boolean formulas (of positive measure), rather than observing fixed samples (which may have zero measure). To the best of our knowledge, our work provides the first semantics for Boolean observe statements as a forward state transformer (as proposed by Kozen [19]) for an imperative probabilistic programming language in the presence of unbounded loops.

Sampson et al. [26] used symbolic execution to transform probabilistic programs into a Bayesian network. Their semantics was aimed at verification of certain probabilistic assertions, however, and lacked a formal foundation in measure theory. Luckow et al. combined symbolic execution with model counting to analyze programs with discrete distributions and nondeterministic choice [20]. Their work used schedulers to reduce Markov Decision Processes to Markov Chains, in contrast to our work with continuous distributions and observe statements, founded in measure theory. Susag et al. [29] considered symbolic execution for programs with random sampling to automatically verify quantitative correctness properties over unknown inputs. While they also used symbolic random variables, they were more focused on verifying correctness properties of “real-world” programs (e.g., written in C++) that use random sampling. They solely considered discrete distributions and did not support observe statements.

Gehr et al. [10] developed PSI, *probabilistic symbolic inference*, a tool that enables programmers to perform posterior distribution, expectation, and assertion queries through symbolic inference. Their symbolic reasoning engine works on symbolic representations of probability distributions, whereas we have symbolic terms which are interpreted as *random variables* of which we do not know the distribution in principle. By default, PSI computes a symbolic representation of the joint posterior distribution represented by the given probabilistic program. In contrast, `symProb` explores all paths through a given probabilistic program and reports on the path condition, substitution map, and path observation of each path. We see these two tools as complementary, as inference is only one aspect of probabilistic programming.

7 Conclusion and Future Work

We have defined new symbolic semantics for imperative probabilistic programs supporting forward execution and conditioning (Bayesian inference), which we proved to be a big-step semantics for symbolic execution of probabilistic programs. To support Bayesian inference, we extended Kozen’s denotational semantics to include `observe` statements of positive-measure events. Significantly, the symbolic semantics thus theoretically supports implementation of a symbolic executor for imperative probabilistic programs with continuous domain variables. We introduced *path observations* to keep track of the part of the initial state space that lead to acceptance of observe statements for each symbolic trace. The symbolic transition system is implemented in our prototype tool `symProb`. Its effectiveness has been demonstrated on example programs, producing results with path conditions and path observations that correctly represent (prior) path probabilities and likelihoods of the symbolic traces, as expected in light of Theorem 5. Since path conditions represent priors and path observations represent likelihoods, we believe it to be beneficial to conceptually separate the two in a symbolic executor.

Interestingly, our semantics and its accompanying correctness results enable one to prove the correctness of certain program transformations. More generally, we believe the symbolic semantics has potential applications in model checking tools, deductive proof systems, optimizations, and reasoning about termination. For example, one can exploit this theory to commute sample and observe statements under appropriate substitutions. In complex probabilistic models, the placement of observe statements may then become an optimization problem. We plan to explore such program transformations both theoretically and experimentally in the future. Moreover, we intend to investigate the limiting behavior of our correctness result, which contains an infinite sum in the presence of unbounded loops. One naturally wonders how fast the sum approximates the true posterior of the program, and how different structures in a program influence the speed of this approximation.

A shortcoming of our semantics for *observe* statements is that zero-measure events cannot be observed in our language. Denotationally, the measure would then always yield zero; operationally, *almost all* simulations will be aborted. Zero-measure observed Boolean conditions may be included in future work, for example by considering *measure couplings* and *disintegrations* [9].

Data availability. The source code for symProb and the experiments we performed with it are available on Zenodo [31].

References

1. Baldoni, R., Coppa, E., D’Elia, D.C., Demetrescu, C., Finocchi, I.: A survey of symbolic execution techniques. *ACM Comput. Surv.* **51**(3), 50:1–50:39 (2018)
2. Barthe, G., Katoen, J.P., Silva, A.: *Foundations of Probabilistic Programming*. Cambridge University Press (2020)
3. de Boer, F.S., Bonsangue, M.: Symbolic execution formally explained. *Formal Aspects of Computing* **33**(4), 617–636 (2021)
4. Borgström, J., Gordon, A.D., Greenberg, M., Margetson, J., Van Gael, J.: Measure transformer semantics for Bayesian machine learning. *Log. Methods Comput. Sci.* **9**(3) (2013)
5. Cadar, C., Dunbar, D., Engler, D.R.: 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)*. pp. 209–224. USENIX Association (2008)
6. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: 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)*. pp. 322–335. ACM (2006)
7. Cadar, C., Godefroid, P., Khurshid, S., Pasareanu, C.S., Sen, K., Tillmann, N., Visser, W.: 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)*. pp. 1066–1071. ACM (2011)
8. Cadar, C., Sen, K.: Symbolic execution for software testing: three decades later. *Commun. ACM* **56**(2), 82–90 (2013)
9. Dahlqvist, F., Silva, A., Danos, V., Garnier, I.: Borel kernels and their approximation, categorically. In: Staton, S. (ed.) *Proceedings of the Thirty-Fourth Conference on the Mathematical Foundations of Programming Semantics (MFPS 2018)*. *Electronic Notes in Theoretical Computer Science*, vol. 341, pp. 91–119. Elsevier (2018)
10. Gehr, T., Misailovic, S., Vechev, M.T.: PSI: exact symbolic inference for probabilistic programs. In: Chaudhuri, S., Farzan, A. (eds.) *Proc. 28th International Conference on Computer Aided Verification (CAV 2016)*. *Lecture Notes in Computer Science*, vol. 9779, pp. 62–83. Springer (2016)
11. Gelman, A., Carlin, J.B., Stern, H.S., Rubin, D.B.: *Bayesian Data Analysis*. Chapman and Hall/CRC, 2 edn. (2004)
12. Ghahramani, Z.: Probabilistic machine learning and artificial intelligence. *Nature* **521**(7553), 452–459 (2015)
13. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. In: Sarkar, V., Hall, M.W. (eds.) *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI’05)*. pp. 213–223. ACM (2005)
14. Goldwasser, S., Micali, S.: Probabilistic encryption. *Journal of Computer and System Sciences* **28**(2), 270–299 (1984)
15. de Gouw, S., Rot, J., de Boer, F.S., Bubel, R., Hähnle, R.: OpenJDK’s `Java.util.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)*. *Lecture Notes in Computer Science*, vol. 9206, pp. 273–289. Springer (2015)

16. Hentschel, M., Bubel, R., Hähnle, R.: The symbolic execution debugger (SED): a platform for interactive symbolic execution, debugging, verification and more. *Int. J. Softw. Tools Technol. Transf.* **21**(5), 485–513 (2019)
17. Holtzen, S., Van den Broeck, G., Millstein, T.: Scaling exact inference for discrete probabilistic programs. *Proc. ACM Program. Lang.* **4**(OOPSLA), 1–31 (2020)
18. King, J.C.: Symbolic execution and program testing. *Commun. ACM* **19**(7), 385–394 (1976)
19. Kozen, D.: Semantics of probabilistic programs. In: 20th Annual Symposium on Foundations of Computer Science (SFCS 1979). pp. 101–114. IEEE (1979)
20. Luckow, K., Păsăreanu, C.S., Dwyer, M.B., Filieri, A., Visser, W.: Exact and approximate probabilistic symbolic execution for nondeterministic programs. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE 2014). pp. 575–586 (2014)
21. Minka, T., Winn, J., Guiver, J., Zaykov, Y., Fabian, D., Bronskill, J.: Infer.net 0.3 (2018)
22. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) *Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science*, vol. 4963, pp. 337–340. Springer (2008)
23. Murphy, K.P.: *Probabilistic Machine Learning: An Introduction*. MIT Press (2022)
24. Nori, A., Hur, C.K., Rajamani, S., Samuel, S.: R2: An efficient mcmc sampler for probabilistic programs. *Proceedings of the AAAI Conference on Artificial Intelligence* **28** (2014)
25. Olmedo, F., Gretz, F., Jansen, N., Kaminski, B.L., Katoen, J.P., McIver, A.: Conditioning in probabilistic programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)* **40**(1), 1–50 (2018)
26. Sampson, A., Panckheka, P., Mytkowicz, T., McKinley, K.S., Grossman, D., Ceze, L.: Expressing and verifying probabilistic assertions. In: O’Boyle, M.F.P., Pingali, K. (eds.) *Proc. 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI’14)*. pp. 112–122. ACM (2014)
27. Staton, S.: Commutative semantics for probabilistic programming. In: Yang, H. (ed.) *Proceedings of the 26th European Symposium on Programming (ESOP 2017). Lecture Notes in Computer Science*, vol. 10201, pp. 855–879. Springer (2017)
28. Staton, S., Yang, H., Wood, F.D., Heunen, C., Kammar, O.: Semantics for probabilistic programming: higher-order functions, continuous distributions, and soft constraints. In: Grohe, M., Koskinen, E., Shankar, N. (eds.) *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science (LICS’16)*. pp. 525–534. ACM (2016)
29. Susag, Z., Lahiri, S., Hsu, J., Roy, S.: Symbolic execution for randomized programs. *Proc. ACM Program. Lang.* **6**(OOPSLA) (Oct 2022)
30. Thrun, S.: Probabilistic robotics. *Commun. ACM* **45**(3), 52–57 (2002)
31. Voogd, E., Johnsen, E.B., Silva, A., Susag, Z.J., Wąsowski, A.: *Artifact for Symbolic Semantics for Probabilistic Programs* (2023). <https://doi.org/10.5281/zenodo.8139552>
32. Voogd, E., Johnsen, E.B., Silva, A., Susag, Z.J., Wąsowski, A.: *Symbolic Semantics for Probabilistic Programs (extended version)* (2023), to appear