Categorical Semantics of Probabilistic Symbolic Execution

JOHN M. LI, Northeastern University, USA
JACK CZENSZAK, Yale University, USA
STEVEN HOLTZEN, Northeastern University, USA

Symbolic execution has emerged as a powerful technique for scaling exact probabilistic inference to languages with more expressive features. But, this expressivity comes at a price: probabilistic programming languages based on symbolic execution are difficult to debug, optimize, and prove correct due to the many intricacies inherent to high-performance symbolic execution strategies. We aim to make it easier to work with probabilistic symbolic executors by developing *symbolic sets*, a new semantic domain that cleanly captures the notion of computation underlying symbolic execution. Just as a symbolic executor replaces ordinary execution with a lifted semantics, symbolic set theory replaces ordinary set theory with a lifted *mathematics*: the category of symbolic sets is a Grothendieck topos, which allows type theory to be used as a metalanguage for working with symbolic sets and functions. We prove a metatheorem that shows how a large class of definitional interpreters written in the internal language of symbolic sets are automatically correct for their ordinary set-theoretic interpretations. Using this metatheorem, we give the first full correctness argument for a symbolic probabilistic language with higher-order functions, type-directed state merging, pattern matching, and structural recursion.

CCS Concepts: • **Theory of computation** → **Categorical semantics**; **Program analysis**.

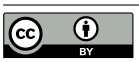
Additional Key Words and Phrases: categorical semantics, probabilistic programming, symbolic execution

ACM Reference Format:
John M. Li, Jack Czenszak, and Steven Holtzen. 2026. Categorical Semantics of Probabilistic Symbolic Execution. *Proc. ACM Program. Lang.* 10, PLDI, Article 265 (June 2026), 38 pages. https://doi.org/10.1145/3808343

1 Introduction

Symbolic execution [27] has emerged as a powerful platform for implementing probabilistic programming languages (PPLs) with exact probabilistic inference [8, 16, 23, 33, 41, 51]. The reason for this is that symbolic executors and symbolic exact probabilistic inference are at their core very similar procedures: they both work by executing programs under a nonstandard semantics that uses logical formulae to abstractly represent many concrete execution traces at once. This close correspondence between symbolic and probabilistic execution has increasingly enabled a new PPL implementation strategy: *repurposing* an existing symbolic executor to perform inference [17, 33, 51, 58]. For example, recently the Roulette [33] probabilistic programming language repurposed the Rosette symbolic evaluation platform with minimal modifications; the result is a PPL as expressive as Rosette, a significant boost in expressivity over prior work [23]. By repurposing an existing symbolic executor, PPL implementors reuse all of the work done by the symbolic execution community on building symbolic executors for general-purpose languages. This automatically does the heavy lifting of "compiling away" a large collection of language features into symbolic representations—a task that has traditionally taken significant manual effort [8, 9, 23].

Authors' Contact Information: John M. Li, Northeastern University, Boston, USA, li.john@northeastern.edu; Jack Czenszak, Yale University, New Haven, USA, jack.czenszak@yale.edu; Steven Holtzen, Northeastern University, Boston, USA, s.holtzen@northeastern.edu.



ACM 2475-1421/2026/6-ART265
https://doi.org/10.1145/3808343

While this repurposing makes it much easier to *implement* a PPL with exact inference, it significantly complicates *reasoning* about the resulting implementation. First, it is often not obvious if a repurposing is correct: probabilistic semantics can differ from normal symbolic execution in subtle ways, and proving a repurposing correct often requires an entirely new semantics and wholesale reengineering of existing proofs. Second, beyond basic correctness, symbolic executors are often heavily optimized programs that rely on careful invariants that may or may not generalize to the probabilistic setting. And finally, once a repurposed symbolic executor is in the hands of a probabilistic programmer, the programmer needs a cost model to effectively use the language. The cost model of symbolic executors is tricky to convey to programmers: for instance, the Rosette performance guide has many helpful hints on how to write performant programs that are quite unintuitive, and Roulette inherits these performance subtleties [53, Sec. 9].

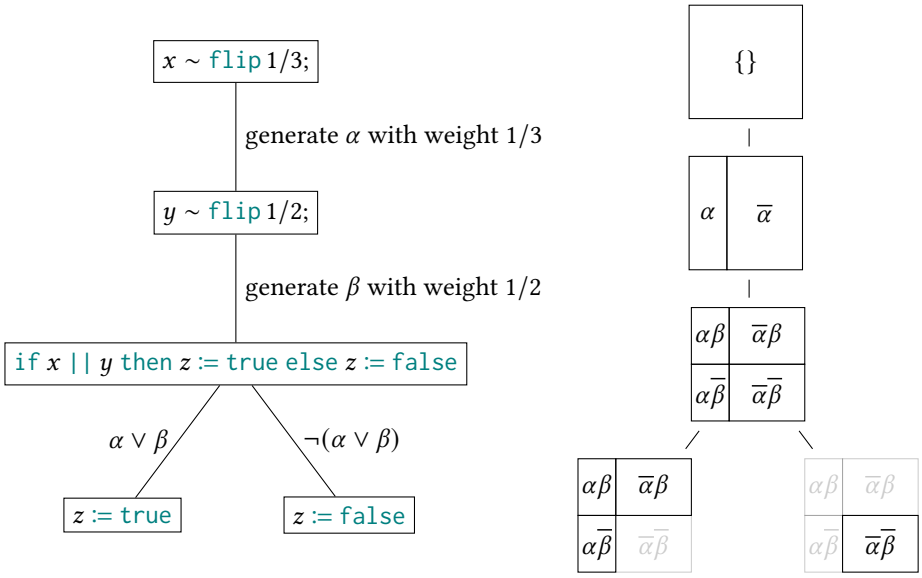
Typically, the tools needed to understand program behavior, justify optimizations, and conduct proofs are provided by a programming language's *semantics*. However, existing semantics of symbolic execution are not equipped for probabilistic adaptation: they are either missing formalizations of key features needed for probabilistic applications [18, 39, 58], are too low-level to validate high-level reasoning principles [2, 14, 15, 57], or are too high-level to capture important aspects of implementation [59]—a detailed comparison can be found in Section 9. Our main aim in this paper is to develop a new semantics that captures just the right amount of detail about symbolic execution: not so low-level that one drowns in irrelevant implementation concerns, and not so high-level that crucial facts about how symbolic execution works are lost. To do this, we identify a connection between the semantics of symbolic execution and *sheaves*. Just as repurposing a symbolic executor allows reusing the implementation effort of the symbolic execution community, recasting symbolic semantics into sheaf theory allows reusing decades of mathematics [7, 25, 28, 31]. We leverage this to build a theory of probabilistic symbolic semantics that can be used to explain behavior, justify optimizations, and prove correctness.

First, we use *presheaves* to define a new category SymSet (§3) of *symbolic sets*, which can be thought of as a semantic domain for working with values guarded by a path condition, a situation similar to the *symbolic unions* used by Rosette [54]. As a category of presheaves, SymSet inherits rich structure useful for modeling the various objects that arise during symbolic execution (§3.2.3). This already yields novel results: applying the formula for presheaf exponentials makes SymSet the first denotational model of symbolic higher-order functions. We extend this basic setup with two monads Gen and Piecewise that capture essential aspects of probabilistic symbolic execution (§3.3).

Next, using the well-studied *internal languages* of categories, we package up the categorical structure of SymSet into a *symbolic metalanguage*—a type theory that turns constructions in SymSet into ordinary functional programs (§4). The symbolic metalanguage can be thought of as a mathematical counterpart to the *solver-aided host language* concept pioneered by Rosette [54]. As a demonstration, we use the symbolic metalanguage to build a probabilistic symbolic semantics (§4.1); this semantics captures enough detail about symbolic execution to be implementable in Haskell, and comes with equational reasoning principles that validate standard optimizations (§4.3).

Moving from presheaves to *sheaves* provides us with a rich theory of symbolic execution with *state merging*—an essential ingredient for scaling in practice (§5). Using sheaves, we build a category $\mathrm{SymSet}_{\mathrm{m}}$ of *mergable symbolic sets*; this provides the first denotational account of the highly nontrivial type-directed state merging strategy of Rosette [10, 39, 55].

Moving onto correctness, we next use *gluing* – a categorical analog of logical relations – to prove two *lifting theorems* (Theorems 6.1 and 6.2) that establish the correctness of a wide class of constructions in both SymSet and $\mathrm{SymSet}_{\mathrm{m}}$. In particular, our lifting theorems imply the correctness of any symbolic metalanguage construction, and hence also the correctness of all symbolic semantics presented in this paper (§6). We again obtain novel results simply by making use of



(a) The symbolic execution tree. (b) The space of weighted models.

Fig. 1. Visualizing probabilistic symbolic execution of NoisyOr.

categorical techniques: the statement of our lifting theorem provides the first compositional account of correctness for a symbolic executor, including correctness at higher types.

Finally, as a demonstration of all of these techniques working together, we build a denotational semantics of higher-order probabilistic symbolic execution with type-directed state merging, pattern matching, and structural recursion (§7)—the first of its kind in each of these aspects. Constructing the semantics is as simple as building a definitional interpreter, and its correctness follows directly from our lifting theorem.

2 A Primer on Probabilistic Symbolic Execution

To set the stage for the ensuing development, this section briefly reviews how symbolic execution can be adapted to perform probabilistic inference. Following traditional accounts of symbolic execution [27], we will use as a running example a simple imperative probabilistic language Imp:

$$e ::= x \in \mathsf{Var} \mid \texttt{true} \mid \texttt{false} \mid \texttt{!}\, e \mid e_1 \,\texttt{\&\&}\, e_2 \mid e_1 \,\texttt{||}\, e_2$$
$$c ::= \texttt{skip} \mid x \coloneqq e \mid x \sim \texttt{flip}\, p \mid c_1 \texttt{;}\ c_2 \mid \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2$$

Imp consists of Boolean expressions e and commands c, including a probabilistic command $x \sim \texttt{flip}\, p$ whose semantics is to set x to a random Boolean that is `true` with probability p.

Probabilistic symbolic execution of Imp works by *lifting* ordinary execution to operate on logical formulae instead of concrete values. To demonstrate, consider the following Imp program:

$$\textsc{NoisyOr} := \left(x \sim \texttt{flip}\, 1/3\texttt{;}\ \ y \sim \texttt{flip}\, 1/2\texttt{;}\ \ \texttt{if}\ x\ \texttt{||}\ y\ \texttt{then}\ z \coloneqq \texttt{true}\ \texttt{else}\ z \coloneqq \texttt{false}\right)$$

Probabilistic symbolic execution of NoisyOr, visualized in Figure 1a, takes the form of a tree whose nodes are commands and whose paths correspond to paths through the original program. Symbolic execution operates on *symbolic stores*, which map variable names to (possibly) symbolic values. First, executing the two `flip` commands generates fresh symbolic variables α and β to

stand in for the random Booleans x and y. These symbolic variables are associated with *weights* that record the distribution of the random Booleans they represent. Then, the `if` splits execution in two, corresponding to the two branches of control flow. Execution proceeds in both branches with each branch guarded by a *path condition*, which is a logical formula capturing the conditions under which that branch is reachable during concrete execution. In the case of NoisyOr, the `then`-branch is run under path condition $\alpha \vee \beta$ and the `else`-branch under its complement $\neg(\alpha \vee \beta)$.

The result of executing the whole `if` is then an *amalgamation* of the results of both branches. The details of amalgamation vary depending on the symbolic execution strategy. Here we will stick to the traditional formulation, where amalgamation *enumerates* all the symbolic stores reachable under both branches, with each state guarded by a corresponding path condition; Section 5 will consider more sophisticated alternatives. This yields the following result for NoisyOr, reminiscent of a symbolic union in Rosette:

$$\Big(\Big[(\alpha \vee \beta, \underbrace{\{x \mapsto \alpha, y \mapsto \beta, z \mapsto \top\}}_{\hat{s}_t}), (\neg(\alpha \vee \beta), \underbrace{\{x \mapsto \alpha, y \mapsto \beta, z \mapsto \bot\}}_{\hat{s}_e})\Big], \underbrace{\{\alpha \mapsto 1/3, \beta \mapsto 1/2\}}_{w}\Big) \quad \text{(Res)}$$

In addition to the weight map w, Res has two reachable symbolic stores: executing the `then`-branch yields $\hat{s}_t$ guarded by $\alpha \vee \beta$, and executing the `else`-branch yields $\hat{s}_e$ guarded by $\neg(\alpha \vee \beta)$.

From the perspective of probability theory, the procedure just described uses symbolic values like $\hat{s}_t$ and $\hat{s}_e$ to represent random variables out of a sample space Ω consisting of *models*, i.e., all concrete assignments to symbolic variables. Figure 1b shows the evolution of Ω throughout execution of NoisyOr: it grows on each `flip` and splits when going under `if`s. At the start, the set of symbolic variables is empty, so $\Omega = \{0,1\}^{\varnothing}$ has exactly one element $\{\}$ corresponding to the empty assignment. The first `flip` introduces α, which expands Ω to $\{0,1\}^{\{\alpha\}}$. The probability measure on Ω is given by the weights of α and $\overline{\alpha}$, and is visualized in Figure 1b using area: the rectangle enclosing α occupies one third of the whole square. The second `flip` expands Ω to $\{0,1\}^{\{\alpha,\beta\}}$. The new probability measure is given as the product of the weights of literals: for example, $\overline{\alpha}\beta$ has probability $(1 - w(\alpha))w(\beta) = 1/3$. Finally, the `if` splits Ω into two *subspaces* Ω_t and Ω_e, corresponding to the models for the logical formulae $\alpha \vee \beta$ and $\neg(\alpha \vee \beta)$. This splitting is depicted in Figure 1b, where the models that lie outside of each branch's subspace have been grayed out.

With these subprobability spaces made explicit, the two stores $\hat{s}_t$ and $\hat{s}_e$ produced by NoisyOr can be thought of as random variables S_t on Ω_t and S_e on Ω_e that map models to concrete stores:

$$S_t(m) = \{x \mapsto m(\alpha), y \mapsto m(\beta), z \mapsto \texttt{true}\} \qquad S_e(m) = \{x \mapsto m(\alpha), y \mapsto m(\beta), z \mapsto \texttt{false}\}$$

Now, Res can itself be cast as a random variable $S : \Omega \to \text{Store}$ defined piecewise on the whole space Ω by $S(m) = S_t(m)$ if $m \models \alpha \vee \beta$ and $S(m) = S_e(m)$ if $m \models \neg(\alpha \vee \beta)$. In particular, the final value of program variable z corresponds to the random variable $Z : \Omega \to \{\texttt{true}, \texttt{false}\}$ defined piecewise by whether one lands in the left or right subspace depicted in Figure 1b; it returns `true` on the union of the three rectangles in which either α or β are true, and `false` on the single rectangle where α and β are both false.

This reduces probabilistic inference on NoisyOr to reasoning about the random variables S and Z. For instance, suppose one is interested in the probability that NoisyOr terminates with $z = \texttt{true}$. This is equivalent to the probability of the event $Z = \texttt{true}$, which in turn is equivalent to a *weighted model count*, i.e. the weighted sum of models satisfying the formula $\alpha \vee \beta$. Weighted model counting is a well-studied problem with many solvers tailored specifically for the probabilistic inference task [11, 12, 16, 36, 42]. In this way, probabilistic symbolic execution reduces the problem of implementing a PPL with exact inference into two well-studied subproblems: symbolic execution on the one hand, and weighted model counting on the other.

3 Symbolic Sets: A New Semantic Domain for Probabilistic Symbolic Execution

Now we begin the process of identifying a semantics that precisely captures the probabilistic symbolic execution strategy outlined in the previous section. A set-theoretic semantics interprets types A as sets $[\![A]\!]$ and programs $a_1 : A_1, \ldots, a_n : A_n \vdash M : B$ as functions $[\![M]\!] : [\![A_1]\!] \times \cdots \times [\![A_n]\!] \to [\![B]\!]$; in category-theoretic terms, this amounts to interpreting types and programs as objects and morphisms, respectively, in the category Set of sets and functions. However, the usual Set-theoretic semantics fails to account for a crucial aspect of symbolic execution: the space of weighted models depicted in Figure 1b and how it evolves over time. Our definition of SymSet will capture this by following a standard recipe known as *presheaf semantics*.

In general, presheaf semantics involves moving from Set to a category of functors $[\mathcal{C}; \mathrm{Set}]$ for a small category $\mathcal{C}$. The resulting interpretation function $[\![-]\!]$ sends types A to functors $[\![A]\!] : \mathcal{C} \to \mathrm{Set}$ (the so-called *presheaves*), and terms to natural transformations. The category $[\mathcal{C}; \mathrm{Set}]$ can be thought of as a proof-relevant generalization of Kripke semantics: objects of $\mathcal{C}$ are Kripke worlds, and morphisms of $\mathcal{C}$ form the Kripke accessibility relation. Each presheaf $P : \mathcal{C} \to \mathrm{Set}$ is then a proof-relevant analog of a world-indexed proposition. The object part of the functor P sends each Kripke world c to a set P_c of "proofs" or "witnesses" to the truth of P at world c. The morphism part of P captures Kripke monotonicity by ensuring that, for each morphism $f : c \to c'$ of $\mathcal{C}$ witnessing the accessibility of c' from c, there is a corresponding function $P_f : P_c \to P_{c'}$ that transforms witnesses for the truth of P at c into witnesses for the truth of P at c' [26].

The world-dependent nature of presheaf semantics makes it ideally suited to modeling computations that vary in time with respect to some ambient environment, with a wide range of applications: different choices for $\mathcal{C}$ can be used to model variation in heap shape [34, 35], typing environment [21], and the amount of steps left in a computation [6].

3.1 Symbolic subprobability spaces as Kripke worlds

Our definition of SymSet is a particular instance of presheaf semantics where $\mathcal{C}$ is set to a category $\mathbb{S}$ of Kripke worlds that models the ambient environment required for probabilistic symbolic execution. The objects of $\mathbb{S}$ are symbolic representations of subprobability spaces:

Definition 3.1. A *symbolic subprobability space* is a pair (Γ, φ) consisting of a *weight map* Γ drawn from the grammar $\Gamma ::= \cdot \mid \Gamma, \alpha : \mathrm{Ber}\ p$, where $\mathrm{Ber}\ p$ is Bernoulli distribution with bias $p \in [0, 1]$, and a *path condition* φ, drawn from $\varphi, \psi \in \mathrm{Formula} ::= \alpha \in \mathrm{vars}(\Gamma) \mid \top \mid \bot \mid \neg\varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi$.

Given a symbolic subprobability space (Γ, φ), one can extract a discrete subprobability space (Ω, μ) where Ω is the set of models $\mathrm{Mod}(\Gamma, \varphi)$ defined by $\mathrm{Mod}(\Gamma, \varphi) = \{m \mid m \vDash \varphi\}$, and $\mu : \Omega \to [0, 1]$ is a discrete subprobability measure defined by taking products of weights of literals according to Γ. We will later see how Definition 3.1 generalizes to theories other than Boolean formulae in §8. In this way, the objects of $\mathbb{S}$ form syntactic representations of subprobability spaces on models.

The morphisms of $\mathbb{S}$ model the way that symbolic subprobability spaces can evolve over time. In Section 2 we saw that there are two ways the subprobability space can change: it grows at each `flip` through the generation of fresh symbolic variables, and shrinks when executing conditional branches through additions to the path condition. Morphisms in $\mathbb{S}$ are generated by sequences of these basic transitions. First, the generation of fresh symbolic variables is captured by the notion of *injective renamings*, following standard categorical models of name generation [47]:

Definition 3.2. A *renaming* r from Γ to Γ', written $r : \Gamma \to \Gamma'$, is an injective function sending each entry $\alpha : \mathrm{Ber}\ p$ in Γ to an entry $r(\alpha) : \mathrm{Ber}\ p$ in Γ' of the same type.

Intuitively, a renaming $r : \Gamma \to \Gamma'$ expands the set of available names from Γ to Γ'. Renamings can be naturally lifted to operate on Boolean formulae: given a path condition $\Gamma \vdash \varphi$ and a renaming

$r : \Gamma \to \Gamma'$, we will write $r(\varphi)$ for the formula, well-formed under Γ', that is given by renaming all the variables in φ according to r. We will also implicitly apply the weakening renaming to path conditions: if $\Gamma \vdash \varphi$, we will write φ for the same formula well-formed under extended context Γ, Δ.

The second kind of transition between subprobability spaces—the shrinking of the path condition φ that occurs when one goes under a conditional branch—is naturally captured by the standard notion of *entailment* for Boolean formulae. Given two path conditions φ and ψ in context Γ, we will say that φ is *stronger* than ψ, written $\Gamma \vdash \varphi \leq \psi$ (or just $\varphi \leq \psi$ if Γ is clear from context), if φ entails ψ. Under the interpretation Mod of objects of $\mathbb{S}$ as subprobability spaces on models, the entailment relation $\leq$ becomes the subspace relation on subprobability spaces. Path conditions will be considered equivalent up to bientailment: that is, $\varphi = \psi$ if and only if $\varphi \leq \psi$ and $\psi \leq \varphi$. We also make use of some standard terminology for splitting path conditions into pieces. Two path conditions will be called *disjoint*, written $\Gamma \vdash \varphi \# \psi$, if $\varphi \wedge \psi = \bot$. For two path conditions φ and ψ, we will write $\varphi + \psi$ for their disjunction in the special case where $\varphi \# \psi$. A family of path conditions $\varphi_1, \ldots, \varphi_n$ *partitions* a path condition φ if $\varphi_1 + \cdots + \varphi_n = \varphi$. Under the interpretation Mod, disjoint formulae become disjoint subprobability spaces and partitions of formulae become partitions of subprobability spaces. Putting renaming and entailment together yields the notion of accessibility required for our category $\mathbb{S}$ of symbolic Kripke worlds:

Definition 3.3. Symbolic subprobability spaces form the objects of a category $\mathbb{S}$ whose morphisms from (Γ, φ) to (Γ', φ') are renamings $r : \Gamma \to \Gamma'$ such that $\Gamma' \vdash \varphi' \leq r(\varphi)$. The identity morphism on (Γ, φ) is the identity renaming $\mathrm{id} : \Gamma \to \Gamma$. Composition of two morphisms $r : (\Gamma, \varphi) \to (\Gamma', \varphi')$ and $r' : (\Gamma', \varphi') \to (\Gamma'', \varphi'')$ is the composition $r' \circ r$ of the two renamings, well-formed because $\Gamma \vdash \varphi' \leq r(\varphi)$ and $\Gamma' \vdash \varphi'' \leq r'(\varphi')$ together imply $\Gamma'' \vdash \varphi'' \leq r'(r(\varphi))$.

Morphisms in $\mathbb{S}$ can be interpreted as functions between subprobability spaces: each morphism $r : (\Gamma, \varphi) \to (\Gamma', \varphi')$ determines a function $\mathsf{Mod}(r) : \mathsf{Mod}(\Gamma', \varphi') \to \mathsf{Mod}(\Gamma, \varphi)$ that sends each model m' of φ' to $m'|_r$, a model of φ computed by restricting m' to only the variables that are in the image of r. The following proposition summarizes the situation:

Proposition 3.1. $\mathsf{Mod}(-)$ forms a functor $\mathbb{S}^{\mathrm{op}} \to \mathrm{Set}$.

In Section 6 we will see how the functor Mod plays a crucial role in correctness. Together, the category $\mathbb{S}$ and the functor Mod are the two pieces of critical data needed for the ensuing technical development: $\mathbb{S}$ models one's choice of logical theory for representing logical formulae, and Mod models how formulae are related to their ground-truth interpretations.

3.2 Symbolic sets as presheaves

With the category $\mathbb{S}$ in hand we can define the category of symbolic sets:

Definition 3.4. The *category of symbolic sets*, written SymSet, is the functor category $[\mathbb{S}; \mathrm{Set}]$. Objects of SymSet will be called *symbolic sets* and morphisms will be called *symbolic functions*.

Defining SymSet as a category of presheaves immediately gives it rich categorical structure. Every presheaf category is a *Grothendieck topos* [31], which can be thought of as an "alternative mathematical universe"—indeed, any Grothendieck topos provides a model of dependent type theory [13, 19, 22]. In this paper, we will make use of only a small amount of this structure, which we record formally below.

Proposition 3.2. SymSet is Cartesian closed and has all limits and colimits.

Definition 3.4 and Proposition 3.2 pack quite a punch: by applying well-established formulae for exponentials, limits, and colimits in presheaves, we obtain a number of new constructions relevant

to the semantics of probabilistic symbolic execution. To demonstrate, the rest of this section is devoted to unpacking the layers of categorical abstraction involved.

3.2.1 Unpacking the definition of symbolic set. Since SymSet is the functor category $[\mathbb{S}; \mathrm{Set}]$, the objects of SymSet are functors $\mathbb{S} \to \mathrm{Set}$. Hence every symbolic set $\widehat{A}$ takes (1) each object (Γ, φ) to a set $A_{(\Gamma,\varphi)}$ and (2) each morphism $r : (\Gamma, \varphi) \to (\Gamma', \varphi')$ to a function $A_r : A_{(\Gamma,\varphi)} \to A_{(\Gamma',\varphi')}$ such that the functor laws are satisfied. Making all details explicit yields a category-free definition:

Definition 3.5. A *symbolic set* $\widehat{A}$ consists of:

- (Carrier) For each symbolic subprobability space (Γ, φ), a set $\widehat{A}_{(\Gamma,\varphi)}$
- (Renaming operation) For each renaming $r : \Gamma \to \Gamma'$, a function $r(-) : \widehat{A}_{(\Gamma,\varphi)} \to \widehat{A}_{(\Gamma', r(\varphi))}$
- (Restriction operation) For each entailment $\Gamma \vdash \varphi' \leq \varphi$, a function $(-)|_{\varphi'} : \widehat{A}_{(\Gamma,\varphi)} \to \widehat{A}_{(\Gamma,\varphi')}$

such that the following hold for all $\widehat{a} \in \widehat{A}_{(\Gamma,\varphi)}$, renamings $s : \Gamma \to \Gamma'$ and $r : \Gamma' \to \Gamma''$, and path conditions φ' and φ'' satisfying $\Gamma \vdash \varphi'' \leq \varphi' \leq \varphi$:

$$\mathrm{id}(\widehat{a}) = \widehat{a} \qquad (r \circ s)(\widehat{a}) = r(s(\widehat{a})) \qquad \widehat{a}|_{\varphi} = \widehat{a} \qquad \widehat{a}|_{\varphi'}|_{\varphi''} = \widehat{a}|_{\varphi''} \qquad r(\widehat{a}|_{\varphi'}) = r(\widehat{a})|_{r(\varphi')}$$

Just as path conditions $\Gamma \vdash \varphi$ are implicitly weakened to larger contexts $\Gamma + \Delta$, we will also implicitly coerce elements $\widehat{a} \in A_{(\Gamma,\varphi)}$ into elements of $A_{(\Gamma+\Delta,\varphi)}$ along the weakening renaming.

An intuition for this definition is that each set $\widehat{A}_{(\Gamma,\varphi)}$ behaves like a set of functions $\mathrm{Mod}(\Gamma, \varphi) \to A$ for some set A. In fact, functions of this shape provide an archetypical example of a symbolic set:

Example 3.1. Given a set A, the *symbolic set of A-valued functions*, written $\mathrm{Fn}\, A$, has carrier $(\mathrm{Fn}\, A)_{(\Gamma,\varphi)} = \mathrm{Mod}(\Gamma, \varphi) \to A$. The restriction operation is given by function restriction, and renaming by $r : \Gamma \to \Gamma'$ sends $f : \mathrm{Mod}(\Gamma, \varphi) \to A$ to $f \circ \mathrm{Mod}(r) : \mathrm{Mod}(\Gamma', \varphi) \to A$. The laws follow directly from properties of function restriction and composition.

In the probabilistic context, such functions denote random variables parameterized by the ambient sample space $\mathrm{Mod}(\Gamma, \varphi)$. For instance, $\mathrm{Fn}\, \mathbb{Z}$ is a symbolic set of integer-valued random variables. Of course, the benefit of the abstract definition is that the sets $\widehat{A}_{(\Gamma,\varphi)}$ do not *need* to be functions. For instance, the following symbolic set captures the way that Boolean formulae represent Boolean functions syntactically:

Example 3.2. The *symbolic set of formulae* $\mathbb{B}$ is defined by mapping each carrier $\mathbb{B}_{(\Gamma,\varphi)}$ to the set Formula of Boolean formulae quotiented by the equivalence relation $\psi \sim \psi'$ if and only if $\varphi \wedge \psi = \varphi \wedge \psi'$. The equivalence class of a formula ψ will be written $[\psi]$. The renaming operation is defined by renaming of variables in formulae and the restriction operation by $[\psi]|_\chi = [\psi \wedge \chi]$.

The quotient in this definition makes ψ and ψ' equal as elements of $\mathbb{B}_{(\Gamma,\varphi)}$ if and only if they entail each other under the assumption φ. This abstractly captures the intuitive idea that ψ and ψ' denote Boolean functions that are equal on the subspace defined by φ.

3.2.2 Unpacking the definition of symbolic function. So far we have only been examining the objects of SymSet. We now turn to its morphisms. As a functor category, morphisms of SymSet are natural transformations. Unpacking the definition of natural transformation in this context yields the following definition to accompany Definition 3.5:

Definition 3.6. A *symbolic function* $\widehat{f} : \widehat{A} \to \widehat{B}$ is a family of functions $\widehat{f}_{(\Gamma,\varphi)} : \widehat{A}_{(\Gamma,\varphi)} \to \widehat{B}_{(\Gamma,\varphi)}$, indexed by (Γ, φ), such that $r(\widehat{f}_{(\Gamma',\varphi)}(x)) = \widehat{f}_{(\Gamma, r(\varphi))}(r(x))$ and $\widehat{f}_{(\Gamma,\varphi)}(x)|_{\varphi'} = \widehat{f}_{(\Gamma,\varphi')}(x|_{\varphi'})$ hold for all x in $\widehat{A}_{(\Gamma,\varphi)}$, renamings $r : \Gamma \to \Gamma'$, and path conditions φ' stronger than φ.

Name	Notation	Carrier	Renaming	Restriction
Singleton	$\widehat{\mathbf{1}}$	$\widehat{\mathbf{1}}_{(\Gamma,\varphi)} = \{\star\}$	$r(\star) = \star$	$\star\|_\psi = \star$
Product	$\widehat{A} \times \widehat{B}$	$(\widehat{A} \times \widehat{B})_{(\Gamma,\varphi)} = \widehat{A}_{(\Gamma,\varphi)} \times \widehat{B}_{(\Gamma,\varphi)}$	$r(\widehat{a}, \widehat{b}) = (r(\widehat{a}), r(\widehat{b}))$	$(\widehat{a}, \widehat{b})\|_\psi = (\widehat{a}\|_\psi, \widehat{b}\|_\psi)$
	$\prod_{i\in I} \widehat{A}_i$	$(\prod_{i\in I} \widehat{A}_i)_{(\Gamma,\varphi)} = \prod_{i\in I} \widehat{A}_{i,(\Gamma,\varphi)}$	$r(\widehat{a}_i)_{i\in I} = (r(\widehat{a}_i))_{i\in I}$	$(\widehat{a}_i)_{i\in I}\|_\psi = (\widehat{a}_i\|_\psi)_{i\in I}$
Coproduct	$\widehat{A} + \widehat{B}$	$(\widehat{A} + \widehat{B})_{(\Gamma,\varphi)} = \widehat{A}_{(\Gamma,\varphi)} + \widehat{B}_{(\Gamma,\varphi)}$	$r(\mathsf{inl}\,\widehat{a}) = \mathsf{inl}(r(\widehat{a}))$ $r(\mathsf{inr}\,\widehat{b}) = \mathsf{inr}(r(\widehat{b}))$	$(\mathsf{inl}\,\widehat{a})\|_\psi = \mathsf{inl}(\widehat{a}\|_\psi)$ $(\mathsf{inr}\,\widehat{b})\|_\psi = \mathsf{inr}(\widehat{b}\|_\psi)$
	$\coprod_{i\in I} \widehat{A}_i$	$(\coprod_{i\in I} \widehat{A}_i)_{(\Gamma,\varphi)} = \coprod_{i\in I} \widehat{A}_{i,(\Gamma,\varphi)}$	$r(\mathsf{inj}_i\,\widehat{a}_i) = \mathsf{inj}_i(r(\widehat{a}_i))$	$(\mathsf{inj}_i\,\widehat{a}_i)\|_\psi = \mathsf{inj}_i(\widehat{a}_i\|_\psi)$
Exponent	$\widehat{A} \Rightarrow \widehat{B}$	$(\widehat{A} \Rightarrow \widehat{B})_{(\Gamma,\varphi)} = \mathsf{Nat}(ⓨ(\Gamma,\varphi) \times \widehat{A}, \widehat{B})$	$r(\widehat{f})(s, \widehat{a}) = \widehat{f}(s \circ r, \widehat{a})$	$\widehat{f}\|_\psi(s, \widehat{a}) = \widehat{f}(s\|_\psi, \widehat{a})$

Fig. 2. A selection of the symbolic set constructions that follow from unpacking Proposition 3.2.

In keeping with the analogy that each set $\widehat{A}_{(\Gamma,\varphi)}$ is like a set of functions $\mathsf{Model}(\Gamma,\varphi) \to A$, a symbolic function $\widehat{f} : \widehat{A} \to \widehat{B}$ is like *composition* by some function $f : A \to B$, sending functions $g : \mathsf{Mod}(\Gamma,\varphi) \to A$ to functions $f \circ g : \mathsf{Mod}(\Gamma,\varphi) \to B$. In fact, just as $\mathsf{Fn}\,A$ is an archetypical example of a symbolic set, function composition is an archetypical example of a symbolic function:

Example 3.3. For each function $f : X \to Y$ there is a symbolic function $\mathsf{Fn}\,f : \mathsf{Fn}\,X \to \mathsf{Fn}\,Y$ defined by $(\mathsf{Fn}\,f)_{(\Gamma,\varphi)}(g) = f \circ g$ for all $g : \mathsf{Mod}(\Gamma,\varphi) \to X$.

Symbolic functions do not *need* to look like function composition. The following example illustrates how the negation connective represents composition by the negation function:

Example 3.4. *Symbolic negation* is the symbolic function $\widehat{\neg} : \mathbb{B} \to \mathbb{B}$ defined by $\widehat{\neg}_{(\Gamma,\varphi)}[\psi] = [\neg\psi]$.

3.2.3 Unpacking universal constructions in symbolic sets. Having fully unpacked Definition 3.4, we now turn to Proposition 3.2. Figure 2 contains a glossary of the consequences of Proposition 3.2 that we will use in this paper.[1] From the standard formulae for computing limits in a category of presheaves [40, Prop. 3.3.9], we obtain *singleton*, *product*, and *indexed product* of symbolic sets. Their definitions operate "componentwise": for instance, elements of $(\widehat{A} \times \widehat{B})_{(\Gamma,\varphi)}$ are pairs consisting of an element in $\widehat{A}_{(\Gamma,\varphi)}$ and an element in $\widehat{B}_{(\Gamma,\varphi)}$, and renaming and restriction of pairs is defined by renaming and restricting components. As an example of products in action, each of the logical connectives on Boolean formulae lift to symbolic functions on the symbolic set $\mathbb{B}$:

Example 3.5. The following define symbolic functions on $\mathbb{B}$:

$$\widehat{\top} : \widehat{\mathbf{1}} \to \mathbb{B} \qquad \widehat{\bot} : \widehat{\mathbf{1}} \to \mathbb{B} \qquad \widehat{\wedge} : \mathbb{B} \times \mathbb{B} \to \mathbb{B} \qquad \widehat{\vee} : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$$

$$\widehat{\top}_{(\Gamma,\varphi)}(\star) = [\top] \quad \widehat{\bot}_{(\Gamma,\varphi)}(\star) = [\bot] \quad \widehat{\wedge}_{(\Gamma,\varphi)}([\psi_1],[\psi_2]) = [\psi_1 \wedge \psi_2] \quad \widehat{\vee}_{(\Gamma,\varphi)}([\psi_1],[\psi_2]) = [\psi_1 \vee \psi_2]$$

Dually, from the formulae for colimits in presheaves we get *coproducts* and *indexed coproducts*, defined as tagged unions in the way one might expect. Finally, SymSet is *Cartesian closed*, with exponential objects $\widehat{A} \Rightarrow \widehat{B}$ defined following the standard formula for exponentials; we will examine exponentials more closely in §7.

[1] The notation "ⓨ", used in the definition of exponentials, is the Yoneda embedding.

3.3 The monads Piecewise and Gen

So far we have discussed only those constructions on symbolic sets that follow directly from it being a category of presheaves. In this section we now discuss those constructions that are particular to probabilistic symbolic execution, which take the form of two monads. In the next section these monads will be used to give definitional interpreters for Imp.

The first monad, Piecewise, models how a list of guarded values can represent a piecewise function on the space of models—recall the random variable S from Section 2, which represented Res. Roughly speaking, an element of $(\mathsf{Piecewise}\,\widehat{A})_{(\Gamma,\varphi)}$ is a set of pairs $\{(\varphi_i, a_i), \dots\}_{1\le i\le n}$ where the φ_is partition φ and each a_i is an element of $\widehat{A}_{(\Gamma,\varphi_i)}$. Intuitively, each such set of pairs represents the piecewise function that takes value a_i on subspace φ_i. These will be called *piecewise definitions* of type $\widehat{A}$ at path condition (Γ, φ). The definition of Piecewise is a quotient of the set of piecewise definitions:

Definition 3.7. $\mathsf{Piecewise}\,\widehat{A}$ is the symbolic set where $(\mathsf{Piecewise}\,\widehat{A})_{(\Gamma,\varphi)}$ is the set of piecewise definitions quotiented by the equivalence relation $\sim$ generated by the following rules:

$$(1)\ \{(\bot, a)\} \sim \varnothing \quad (2)\ \{(\varphi_1 + \varphi_2, a)\} \sim \{(\varphi_1, a|_{\varphi_1}), (\varphi_2, a|_{\varphi_2})\} \quad (3)\ f \sim g \implies f \uplus h \sim g \uplus h$$

Intuitively, rule (1) states that a piecewise definition guarded by an unsatisfiable path condition is equivalent to an empty function, rule (2) states that a branch of a piecewise definition guarded by a disjunction of disjoint path conditions can be split into two individual branches, and rule (3) states that two equal piecewise definitions stay equal under the addition of additional cases h. The equivalence class of $\{(\varphi_1, a_1), \dots, (\varphi_n, a_n)\}$ will be written $[\varphi_1 : a_1, \dots, \varphi_n : a_n]$. The renaming operation is defined by $r[\varphi_1 : a_1, \dots, \varphi_n : a_n] = [r(\varphi_1) : r(a_1), \dots, r(\varphi_n) : r(a_n)]$ and the restriction operation by $[\varphi_1 : a_1, \dots, \varphi_n : a_n]|_{\varphi'} = [(\varphi_1 \wedge \varphi') : a_1|_{\varphi'}, \dots, (\varphi_n \wedge \varphi') : a_n|_{\varphi'}]$.

The definition of Piecewise appears complicated compared to a more straightforward one that does not use quotients. Section 5 will explain how the quotient is actually quite natural from the perspective of sheaves, and has many pleasant consequences. We record one consequence here: the quotient implies the intuitive fact that piecewise Booleans are equivalent to logical formulae.

Proposition 3.3. $\mathsf{Piecewise}(\widehat{\mathbf{1}} + \widehat{\mathbf{1}}) \cong \mathbb{B}$ in SymSet.

Proof. Any element $[\varphi_1 : b_1, \dots, \varphi_n : b_n]$ of $\mathsf{Piecewise}(\widehat{\mathbf{1}} + \widehat{\mathbf{1}})_{(\Gamma,\varphi)}$ can be brought into a normal form $[\psi : \mathsf{inl}(\star), (\varphi \wedge \neg\psi) : \mathsf{inr}(\star)]$, where ψ is the disjunction over all the φ_is with $b_i = \mathsf{inl}(\star)$. The map $[\varphi_1 : b_1, \dots, \varphi_n : b_n] \mapsto [\psi]$ defines a bijective symbolic function $\mathsf{Piecewise}(\widehat{\mathbf{1}} + \widehat{\mathbf{1}}) \to \mathbb{B}$. □

Piecewise forms a strong monad on SymSet: the unit map sends $a \in A_{(\Gamma,\varphi)}$ to the singleton $[\varphi : a]$, the multiplication map flattens $[\varphi_i : [\psi_{ij} : a_{ij}]_{j\in J}]_{i\in I}$ into $[\varphi_i \wedge \psi_{ij} : a_{ij}]_{(i,j)\in I\times J}$, and the strength map pushes the first component of a pair $(a, [\varphi_i : b_i]_{i\in I})$ into the second to give $[\varphi_i : (a|_{\varphi_i}, b_i)]_{i\in I}$.

3.3.1 The monad Gen. The second monad that is specific to our setting models the generation of fresh symbolic variables. Its definition is heavily inspired by traditional presheaf models of name generation [47]. Roughly speaking, the elements of $(\mathsf{Gen}\,A)_{(\Gamma,\varphi)}$ are pairs (Δ, a) where Δ is a context containing newly-generated symbolic variables and $a \in A_{(\Gamma+\Delta,\varphi)}$ is an element of the symbolic set A at the extended context $\Gamma + \Delta$. This has the structure of a monad: any pure computation $a \in A_{(\Gamma,\varphi)}$ can be transformed into the pair $(\varnothing, a)$ modeling a computation that does not generate any variables, and any nested computation $(\Delta_1, (\Delta_2, a))$ can be flattened into $(\Delta_1 + \Delta_2, a)$, collecting up the variables Δ_1 and Δ_2 generated by subcomputations.

Just as with Piecewise, going from this basic idea to the definition of Gen requires a quotient in order to validate additional equational reasoning principles that one would intuitively expect

to hold of probabilistic symbolic execution. First, we would like the following equations to hold (where $\mathsf{gen}\, p$ is the effectful operation that generates a fresh symbolic variable with bias b):

$$(x \leftarrow \mathsf{gen}\, p;\ e) = e \ \ (x \notin \mathsf{FV}(e)) \qquad (x \leftarrow \mathsf{gen}\, p;\ y \leftarrow \mathsf{gen}\, c;\ e) = (y \leftarrow \mathsf{gen}\, c;\ x \leftarrow \mathsf{gen}\, p;\ e)$$

These equations state that Gen should be *affine commutative*, which means that the order that symbolic variables are generated in does not matter and that unused symbolic variables can be dropped—properties that should intuitively hold of any generative effect [47]. Second, we would like Gen to compose nicely with Piecewise, so that both the effects that they model can be used together. This leads us to a quotient by an equivalence relation that relates two computations (Δ, a) and (Δ', a') if they are piecewise equal up to renaming of freshly generated symbolic variables:

Definition 3.8. The *symbolic name generation monad*, written Gen, is the monad on SymSet that sends each symbolic set A to the symbolic set $\mathsf{Gen}\, A$ with $(\mathsf{Gen}\, A)_{(\Gamma,\varphi)} = \{(\Delta, a) \mid a \in A_{(\Gamma+\Delta,\varphi)}\}/_\sim$ where $(\Delta, a) \sim (\Delta', a')$ if and only if there is a partition $\varphi_1, \ldots, \varphi_n$ of φ such that, for each $1 \le i \le n$, there is a context Δ'' and renamings $r_i : \Delta \to \Delta''$ and $r_i' : \Delta' \to \Delta''$ such that $r_i(a|_{\varphi_i}) = r_i'(a|_{\varphi_{i'}})$.

The equivalence class of (Δ, a) will be written $\nu\, \Delta.\, a$, following the ν-calculus [38]. Intuitively, $\nu\Delta.a$ is the computation that produces a after generating fresh variables Δ. The quotient ensures that the choice of names in Δ does not matter. The renaming operation is defined by $r(\nu\, \Delta.\, a) = \nu\, \Delta.\, (r + \Delta)(a)$, where $r + \Delta$ is the renaming given by extending r with the identity renaming on Δ, and the restriction operation is defined by $(\nu\, \Delta.\, a)|_\varphi = \nu\, \Delta.\, (a|_\varphi)$. Altogether, this makes Gen a strong monad on SymSet, with unit map defined by $a \mapsto \nu\, \varnothing.\, a$, multiplication map by $\nu\, \Delta_1.\, \nu \Delta_2.\, a \mapsto \nu\, (\Delta_1 + \Delta_2).\, a$, and strength map by $(a, \nu\, \Delta.\, b) \mapsto \nu\, \Delta.\, (a, b)$.

Definition 3.9. For each $p \in [0, 1]$, let $\mathsf{gen}\, p : \widehat{\mathbf{1}} \to \mathsf{Gen}(\mathsf{Piecewise}(\widehat{\mathbf{1}} + \widehat{\mathbf{1}}))$ be the symbolic function defined by $(\mathsf{gen}\, p)_{(\Gamma,\varphi)}(\star) = \nu\, (\alpha : \mathsf{Ber}\, p).\, [\varphi \wedge \alpha : \mathsf{inl}(\star), \varphi \wedge \neg\alpha : \mathsf{inr}(\star)]$.

The quotient used to define Gen makes it affine commutative. While monads do not compose in general, the following establishes that Gen composes nicely with Piecewise (see §D for a proof):

Theorem 3.1. Both $\mathsf{Piecewise} \circ \mathsf{Gen}$ and $\mathsf{Gen} \circ \mathsf{Piecewise}$ are monads.

4 Symbolic Sets in Action

This section puts the just-defined category of symbolic sets to work. For demonstration, it will show how to use symbolic sets to derive a symbolic semantics for Imp that formalizes the informal procedure described in Section 2. Thanks to the rich structure of symbolic sets, this symbolic semantics is quick to construct (§4.1 and §4.2), and captures enough implementation detail that it can be converted into Haskell code (§4.3).

4.1 The symbolic metalanguage: symbolic semantics as functional programming

To build a symbolic semantics for Imp, one needs to define a symbolic set for each Imp semantic domain and a symbolic function for each Imp expression and command. At first this seems quite involved because symbolic sets and functions have complicated definitions. But, the rich structure of SymSet allows us to make use of its *internal language*—a type theory where types denote SymSet objects and terms denote morphisms—as a domain-specific language for categorical constructions.

With a category as well behaved as SymSet, there are many languages one could use for this purpose. In this paper, it will suffice to use a simply-typed language extended with infinitary sum and product types; Proposition 3.2 guarantees that SymSet forms a model. On top of this stock type theory, we sprinkle in the two monads Piecewise and Gen, which upgrade it with monadic constructs that are interpreted in the standard way [32].

$$\mathsf{Bool} = \mathbf{1} + \mathbf{1}$$
$$\mathsf{Store} = \prod_{x \in \mathsf{Var}} \mathsf{Piecewise}\,\mathsf{Bool}$$

(a) Interpreting types.

$$\begin{aligned}
[\![e]\!] &: \mathsf{Store} \to \mathsf{Piecewise}\,\mathsf{Bool} \\
[\![x]\!]\,(s) &= \pi_x(s) \\
[\![\texttt{true}]\!]\,(s) &= \mathsf{pure}(\mathsf{inl}\,\star) \\
[\![\texttt{false}]\!]\,(s) &= \mathsf{pure}(\mathsf{inr}\,\star) \\
[\![\texttt{!}\,e]\!]\,(s) &= v \leftarrow [\![e]\!]\,(s);\ \mathsf{pure}(\mathsf{not}\,v) \\
[\![e_1\ \texttt{\&\&}\ e_2]\!]\,(s) &= v_1 \leftarrow [\![e_1]\!]\,(s);\ v_2 \leftarrow [\![e_2]\!]\,(s);\ \mathsf{pure}(v_1\ \&\&\ v_2) \\
[\![e_1\ \texttt{||}\ e_2]\!]\,(s) &= v_1 \leftarrow [\![e_1]\!]\,(s);\ v_2 \leftarrow [\![e_2]\!]\,(s);\ \mathsf{pure}(v_1\ ||\ v_2)
\end{aligned}$$

(b) Interpreting expressions.

$$\begin{aligned}
[\![c]\!] &: \mathsf{Store} \to \mathsf{Gen}(\mathsf{Piecewise}\,\mathsf{Store}) \\
[\![\texttt{skip}]\!]\,(s) &= \mathsf{pure}\,s \\
[\![x := e]\!]\,(s) &= \mathsf{pure}\,s[x \mapsto [\![e]\!]\,(s)] \\
[\![x \sim \texttt{flip}\,p]\!]\,(s) &= v \leftarrow \mathsf{gen}\,p \\
&\quad\ \mathsf{pure}\,s[x \mapsto v] \\
[\![c_1\,;\ c_2]\!]\,(s) &= s_1 \leftarrow [\![c_1]\!]\,(s) \\
&\quad\ [\![c_2]\!]\,(s_1) \\
\left[\!\!\left[\begin{array}{l}\texttt{if}\ e \\ \texttt{then}\ c_1 \\ \texttt{else}\ c_2\end{array}\right]\!\!\right](s) &= b \leftarrow [\![e]\!]\,(s) \\
&\quad\ \mathsf{if}\ b \\
&\quad\ \mathsf{then}\,[\![c_1]\!]\,(s) \\
&\quad\ \mathsf{else}\,[\![c_2]\!]\,(s)
\end{aligned}$$

(c) Interpreting commands.

Fig. 3. Interpreting Imp in the symbolic metalanguage.

Figure 3 depicts this *symbolic metalanguage* at work. Figure 3a uses the metalanguage type formers to build the requisite semantic domains. Booleans are defined by the sum $\mathbf{1} + \mathbf{1}$, and stores are defined by an indexed product over piecewise-definable Booleans. Figures Fig. 3b and Fig. 3c use the term formers of the metalanguage to interpret expressions and commands. Expressions denote symbolic functions from stores to piecewise Booleans, using do-notation to work with the Piecewise monad in the usual way, and commands denote monadic computations from stores to piecewise stores. Notably, $x \sim \texttt{flip}\,p$ is interpreted using the effectful operation $\mathsf{gen}\,p$.

4.2 Unpacking the symbolic semantics of Imp

Just as a program written in a DSL can be macro-expanded into its host language, the interpreter in Figure 3 can be turned into ordinary math by unpacking definitions. This yields a semantics $\mathcal{S}[\![-]\!]$ for Imp where all symbolic execution details are made explicit. First, unpacking Figure 3a gives interpretations of Imp semantic types.

- Booleans are interpreted by a symbolic set $\widehat{\mathsf{Bool}}$ defined by $\widehat{\mathsf{Bool}}_{(\Gamma,\varphi)} = \{\mathsf{inl}(\star), \mathsf{inr}(\star)\}$, with trivial renaming and restriction operations $r(\widehat{b}) = \widehat{b}$ and $\widehat{b}|_\psi = \widehat{b}$.
- Stores are interpreted by a symbolic set $\widehat{\mathsf{Store}}$ where $\widehat{\mathsf{Store}}_{(\Gamma,\varphi)}$ is the set of mappings of the form $\{x \mapsto [\varphi_i : \widehat{b}_i]_{i\in I}, \dots\}$ associating each $x \in \mathsf{Var}$ to a piecewise element of $\widehat{\mathsf{Bool}}$. In light of Proposition 3.3, these are equivalent to mappings $\{x \mapsto [\varphi], \dots\}$ from variables x to equivalence classes of Boolean formulae φ. The renaming and restriction operations are defined variable-wise: $r\{x \mapsto [\varphi], \dots\} = \{x \mapsto [r(\varphi)], \dots\}$ and $\{x \mapsto [\varphi], \dots\}|_\psi = \{x \mapsto [\varphi \wedge \psi], \dots\}$.

Next, unpacking Figure 3b gives an interpretation of expressions, shown on the left in Figure 4 below. For legibility, we have converted all piecewise Booleans into logical formulae, following Proposition 3.3. Each expression e denotes a function that sends a store $\hat{s}$ to the formula given by replacing each variable x in e with the corresponding entry in $\hat{s}$ and each Boolean operation ($\texttt{true}$, $\texttt{false}$, $\texttt{!}$, $\texttt{\&\&}$, $\texttt{||}$) with an application of the corresponding symbolic function ($\widehat{\top}$, $\widehat{\bot}$, $\widehat{\neg}$, $\widehat{\wedge}$, $\widehat{\vee}$).

Lastly, unpacking Figure 3c yields an interpretation of commands, shown on right in Figure 4, as functions that take in a symbolic subprobability space and a store and produce a piecewise store under some fresh variables. We briefly describe the interesting cases. Sampling $x \sim \texttt{flip}\,p$ generates a symbolic variable α of type $\mathsf{Ber}\,p$ and sets x to α in the store $\hat{s}$. Sequencing $c_1\,;\ c_2$ first runs c_1 in the input store $\hat{s}$ to get new variables Γ' and a piecewise store $[\varphi'_i : \hat{s}'_i]_{i\in I}$, and

$$\mathcal{S}[\![e]\!]_{(\Gamma,\varphi)} : \widehat{\mathsf{Store}}_{(\Gamma,\varphi)} \to (\mathsf{Piecewise}\,\widehat{\mathsf{Bool}})_{(\Gamma,\varphi)}$$
$$\mathcal{S}[\![\mathtt{true}]\!]_{(\Gamma,\varphi)}(\hat{s}) = [\top]$$
$$\mathcal{S}[\![\mathtt{false}]\!]_{(\Gamma,\varphi)}(\hat{s}) = [\bot]$$
$$\mathcal{S}[\![x]\!]_{(\Gamma,\varphi)}(\hat{s}) = \hat{s}(x)$$
$$\mathcal{S}[\![!\,e]\!]_{(\Gamma,\varphi)}(\hat{s}) = [\neg\psi] \text{ where}$$
$$\mathcal{S}[\![e]\!]_{(\Gamma,\varphi)}(\hat{s}) = [\psi]$$
$$\mathcal{S}[\![e_1 \,\&\&\, e_2]\!]_{(\Gamma,\varphi)}(\hat{s}) = [\psi_1 \wedge \psi_2] \text{ where}$$
$$\mathcal{S}[\![e_1]\!]_{(\Gamma,\varphi)}(\hat{s}) = [\psi_1] \text{ and}$$
$$\mathcal{S}[\![e_2]\!]_{(\Gamma,\varphi)}(\hat{s}) = [\psi_2]$$
$$\mathcal{S}[\![e_1 \,||\, e_2]\!]_{(\Gamma,\varphi)}(\hat{s}) = [\psi_1 \vee \psi_2] \text{ where}$$
$$\mathcal{S}[\![e_1]\!]_{(\Gamma,\varphi)}(\hat{s}) = [\psi_1] \text{ and}$$
$$\mathcal{S}[\![e_2]\!]_{(\Gamma,\varphi)}(\hat{s}) = [\psi_2]$$

$$\mathcal{S}[\![c]\!]_{(\Gamma,\varphi)} : \widehat{\mathsf{Store}}_{(\Gamma,\varphi)} \to (\mathsf{Gen}(\mathsf{Piecewise}\,\widehat{\mathsf{Store}}))_{(\Gamma,\varphi)}$$
$$\mathcal{S}[\![\mathtt{skip}]\!]_{(\Gamma,\varphi)}(\hat{s}) = \nu\,\varnothing.\,[\varphi : \hat{s}]$$
$$\mathcal{S}[\![x \coloneqq e]\!]_{(\Gamma,\varphi)}(\hat{s}) = \nu\,\varnothing.\,[\varphi : \hat{s}[x \mapsto \mathcal{S}[\![e]\!]_{(\Gamma,\varphi)}(\hat{s})]]$$
$$\mathcal{S}[\![x \sim \mathtt{flip}\,p]\!]_{(\Gamma,\varphi)}(\hat{s}) = \nu\,(\alpha : \mathrm{Ber}\,p).\,[\varphi : \hat{s}[x \mapsto \alpha]]$$
$$\mathcal{S}[\![c_1\,;\,c_2]\!]_{(\Gamma,\varphi)}(\hat{s}) = \nu\left(\Gamma' + \textstyle\sum_{i\in I}\Gamma_i''\right).\,[\varphi_{ij}'' : \hat{s}_{ij}'']_{i,j\in I\times J} \text{ where}$$
$$\mathcal{S}[\![c_1]\!]_{(\Gamma,\varphi)}(\hat{s}) = \nu\,\Gamma'.\,[\varphi_i' : \hat{s}_i']_{i\in I}$$
$$\mathcal{S}[\![c_2]\!]_{(\Gamma+\Gamma',\varphi_i')}(\hat{s}_i) = \nu\,\Gamma_i''.\,[\varphi_{ij}'' : \hat{s}_{ij}'']_{j\in J} \text{ for all } i \in I$$
$$\mathcal{S}\left[\!\!\left[\begin{array}{l}\mathtt{if}\ e\\ \mathtt{then}\ c_1\\ \mathtt{else}\ c_2\end{array}\right]\!\!\right]_{(\Gamma,\varphi)}(\hat{s}) = \nu\,(\Gamma_1' + \Gamma_2').\,[\varphi_i' : \hat{s}_i']_{i\in I} + [\varphi_j' : \hat{s}_j']_{i\in J} \text{ where}$$
$$\mathcal{S}[\![e]\!]_{(\Gamma,\varphi)}(\hat{s}) = [\psi]$$
$$\mathcal{S}[\![c_1]\!]_{(\Gamma,\varphi\wedge\psi)}(\hat{s}|_{\varphi\wedge\psi}) = \nu\,\Gamma_1'.\,[\varphi_i' : \hat{s}_i']_{i\in I}$$
$$\mathcal{S}[\![c_2]\!]_{(\Gamma,\varphi\wedge\neg\psi)}(\hat{s}|_{\varphi\wedge\neg\psi}) = \nu\,\Gamma_2'.\,[\varphi_j' : \hat{s}_j']_{i\in J}$$

Fig. 4. Symbolic semantics of Imp expressions (left) and commands (right) unfolded from Fig. 3.

```
data Formula = Var Integer | Top | Bot | And Formula Formula | Or Formula Formula | Not Formula

true _ = Top                 skip (p,s) = pure [(p,s)]
false _ = Bot                set x e (p,s) = pure [(p,insert x (e s) s)]
var x (p,s) = find x s       flip x b (p,s) (i,c) = ([(p,insert x (Var i) s)],(i+1,insert i b c))
not e = Not . e              (c1 >>> c2) (p,s) = concat <$> (mapM c2 =<< c1 (p,s))
(&&&) = liftA2 And           ite e c1 c2 (p,s) = (++) <$> try c1 (p /\ q,s) <*> try c2 (p /\ Not q,s)
(|||) = liftA2 Or              where { q = e s; try c (p,s) = if unsat p then pure [] else c (p,s) }
```

Fig. 5. Probabilistic symbolic executor for Imp expressions (left) and commands (right) derived from Fig. 4.

then runs c_2 under each new path condition φ_i' and store $\hat{s}_i'$. The results of all these runs are then combined using the multiplication of Gen ∘ Piecewise, which takes the disjoint union of all fresh names generated by all runs of c_2 and forms a context $\sum_{i\in I}\Gamma_i''$. Conditionals first run the guard to a formula ψ, then combine the results of running the branches under new path conditions $\varphi \wedge \psi$ and $\varphi \wedge \neg\psi$ that restrict the current path condition φ along the guard e.

4.3 Symbolic sets as reasoning tool

In the previous section we saw how $\mathcal{S}[\![-]\!]$ captures quite a bit of detail relevant to implementation, such as how path conditions should be updated during symbolic execution. In fact, this semantics contains enough information to be turned into executable code: Figure 5 depicts a Haskell implementation of $\mathcal{S}[\![-]\!]$. Figure 5 is heavily condensed, but a full code listing can be found in §A. The left of Figure 5 interprets expressions as Haskell functions `e :: (Formula, Store) -> Formula` that produce a `Formula` given a pair `(p,s)` containing a path condition and a store—`Store` is defined to be the type of maps from variables to `Formula`s. The right of Figure 5 interprets commands. In converting the semantics to code, the symbolic context Γ becomes a piece of state that contains (1) an `Integer` counter for generating fresh symbolic variables and (2) a `WeightMap` that maps each variable to its probability. As a result, commands take the form `c :: (Formula, Store) -> State (Integer, WeightMap) [(Formula, Store)]`, with the piecewise stores produced by the semantics represented by lists of pairs. The symbolic semantics of each Imp construct corresponds closely with this implementation: for instance, the combinator `flip x b` implements $x \sim \mathtt{flip}\,b$ by associating the next fresh symbolic variable `i` with `b` in the context `c`, incrementing the fresh variable counter, and then setting `x` to `Var i` in the store `s`.

Crucially, the symbolic semantics only captures *essential* details of this implementation—low-level questions, such as the representation of formulae and the maintenance of the fresh variable counter, are not modeled. This is by design: the abstractness of $\mathcal{S}[\![-]\!]$ makes it a useful tool for reasoning. To demonstrate, we show how algebraic identities validated by the symbolic semantics justify two ways in which the code in Figure 5 has been optimized over a simpler implementation that naively translates the symbolic semantics. Both optimizations concern `if` statements. The first optimization is that `ite` does not restrict the store `s` to the new path conditions `p /\ q` and `p /\ Not q` before running the branches `c1` and `c2`, even though the semantics does. Intuitively, this is safe to do because `s` contains at least as much information as its restrictions, so a function expecting a restriction can be given `s` directly without losing information. This intuitive argument can be justified by equational reasoning in symbolic sets: the quotient used to define $\mathbb{B}$ ensures that $[\varphi \wedge \psi] = [\varphi]$ as elements of $\mathbb{B}_{(\Gamma,\psi)}$. Therefore, the restriction operation on stores can be simplified as follows: $\{x \mapsto [\varphi], \dots\}|_\psi = \{x \mapsto [\varphi \wedge \psi], \dots\} = \{x \mapsto [\varphi], \dots\}$. Since the `Formula`s in the implementation encode representatives φ of the equivalence classes $[\varphi]$, this equation implies store restriction can be implemented as a no-op.

The second optimization of `ite` is that it uses `try` to stop execution when the path condition is `unsat`isfiable. Intuitively, this is safe because any computation with an unsatisfiable path condition is unreachable. The following lemma justifies this intuition.

Lemma 4.1. $(\mathrm{Gen}(\mathrm{Piecewise}\,\widehat{\mathrm{Store}}))_{(\Gamma,\bot)}$ is a singleton for any unsatisfiable path condition $(\Gamma, \bot)$.

Proof. The only partitions of $\bot$ are of shape $\bot + \cdots + \bot$, so elements of $(\mathrm{Piecewise}\,\widehat{\mathrm{Store}})_{(\Gamma,\bot)}$ are of shape $[\bot : \hat{s}_1, \dots, \bot : \hat{s}_n]$ for $\hat{s}_1, \dots, \hat{s}_n \in \widehat{\mathrm{Store}}_{(\Gamma,\bot)}$. By the quotient used to define Piecewise, these elements are all equal to $[]$. This implies all elements of $(\mathrm{Gen}(\mathrm{Piecewise}\,\widehat{\mathrm{Store}}))_{(\Gamma,\bot)}$ are of shape $\nu\,\Delta.\,[]$ for some context Δ. But now $\nu\,\Delta.\,[] = \nu\,\varnothing.\,[]$ by the quotient used to define Gen. Hence $\nu\,\varnothing.\,[]$ is the unique element of $(\mathrm{Gen}(\mathrm{Piecewise}\,\widehat{\mathrm{Store}}))_{(\Gamma,\bot)}$. □

The quotients used to define Piecewise and Gen play a key role this proof. Note also that the precise form of the unique element $\nu\,\varnothing.\,[]$ of $(\mathrm{Gen}(\mathrm{Piecewise}\,\widehat{\mathrm{Store}}))_{(\Gamma,\bot)}$ directly justifies returning `pure []` in `try`—it shows that, in case the current path condition is unsatisfiable, it is semantically valid to return the monadic computation that produces the empty symbolic union `[]` without generating any new symbolic variables.

5 State Merging via Sheaves

So far we have considered a symbolic execution strategy known as *all-path symbolic execution*, which simply enumerates all of the results produced by conditional branches: recall from Section 2 how symbolic execution of NoisyOr summarized its two execution paths in the form of a symbolic union $[\alpha \vee \beta : \hat{s}_\mathrm{t}, \neg(\alpha \vee \beta) : \hat{s}_\mathrm{t}]$ containing the two symbolic stores $\hat{s}_\mathrm{t}$ and $\hat{s}_\mathrm{e}$. Under all-paths symbolic execution, any continuation of NoisyOr will be run twice—once for $\hat{s}_\mathrm{t}$ and once for $\hat{s}_\mathrm{e}$—leading to a duplication of work that can produce an exponential number of execution paths as one scales to larger programs. This *path explosion problem* is mitigated in practice by *state merging*, which combines the results of two different branches of computation in order to save work down the line [44, 54].

Take for example the symbolic stores $\hat{s}_\mathrm{t}$ and $\hat{s}_\mathrm{e}$ produced by NoisyOr. Semantically, these are elements of $\widehat{\mathrm{Store}}_{(\Gamma_{\alpha\beta},\alpha\vee\beta)}$ and $\widehat{\mathrm{Store}}_{(\Gamma_{\alpha\beta},\neg(\alpha\vee\beta))}$ respectively, where $\Gamma_{\alpha\beta}$ is the two-element context $\alpha : \mathrm{Ber}(1/3), \beta : \mathrm{Ber}(1/2)$. The information contained in $\hat{s}_\mathrm{t}$ and $\hat{s}_\mathrm{e}$ can be merged to form $\hat{s}_\mathrm{t} \sqcup_{\alpha\vee\beta,\neg(\alpha\vee\beta)} \hat{s}_\mathrm{e}$, an element of $\widehat{\mathrm{Store}}$ supported by path condition $(\alpha \vee \beta) \vee \neg(\alpha \vee \beta) = \top$, by

merging the logical formulas stored at each variable:

$$\begin{aligned}
&\hat{s}_{\mathsf{t}} \sqcup_{\alpha\vee\beta,\neg(\alpha\vee\beta)} \hat{s}_{\mathsf{e}} \\
&= \{x \mapsto \alpha, y \mapsto \beta, z \mapsto \top\} \sqcup_{\alpha\vee\beta,\neg(\alpha\vee\beta)} \{x \mapsto \alpha, y \mapsto \beta, z \mapsto \bot\} && \text{definition of } \hat{s}_{\mathsf{t}}, \hat{s}_{\mathsf{e}} \\
&= \{x \mapsto \alpha \sqcup \alpha, y \mapsto \beta \sqcup \beta, z \mapsto \mathrm{ite}(\alpha \vee \beta, \top, \bot)\} && \text{merge stores variable-wise} \\
&= \{x \mapsto \alpha, y \mapsto \beta, z \mapsto \alpha \vee \beta\} && \text{merge the logical formulas}
\end{aligned}$$

In practice, the definitions of merging operations like $\sqcup_{\alpha\vee\beta,\neg(\alpha\vee\beta)}$ depend heavily on the type of data being merged, and can get quite intricate. For example, if symbolic stores were allowed to omit entries for undefined variables, then the definition of $\sqcup_{\alpha\vee\beta,\neg(\alpha\vee\beta)}$ would be complicated by additional logic to handle the case where a variable exists in one store but not the other.

This section describes how the categorical framework developed thus far naturally accommodates merging. In particular, we will identify a subcategory $\mathsf{SymSet}_{\mathsf{m}}$ of SymSet consisting of *mergable symbolic sets*, and leverage the categorical structure of $\mathsf{SymSet}_{\mathsf{m}}$ to define a variety of symbolic objects automatically equipped with merge operations.

As foreshadowed in Section 1, defining $\mathsf{SymSet}_{\mathsf{m}}$ requires passing from presheaves to sheaves. Section 3 put forth the intuition that a category of presheaves $[C; \mathrm{Set}]$ is like a proof-relevant Kripke semantics, with objects of C representing possible worlds and morphisms representing accessibility. Sheaves additionally give each Kripke world c a *spatial* character, by allowing one to specify when a given world c can be decomposed into a family of accessible worlds $(c_i)_{i\in I}$. This ability to decompose a world into subworlds lets us model the way that subprobability spaces are decomposed into subspaces by conditionals, as depicted in Figure 1b.

Formally, decomposition of Kripke worlds $c \in C$ is captured by a *coverage* on C [25, C2.1.1], which specifies when a family of morphisms $\{f_i : c \to c_i\}_{i\in I}$ "covers" its common domain c in the sense that the accessible worlds $(c_i)_{i\in I}$ jointly contain enough information to recover the original world c. To capture conditionals, a natural choice of coverage for $\mathbb{S}$ is the one where a finite family of morphisms $\{f_i : (\Gamma, \varphi) \to (\Gamma_i, \varphi_i)\}_{1\le i\le n}$ covers a symbolic subprobability space (Γ, φ) if it represents, via Mod, a family of functions $\{\mathsf{Mod}(f_i) : \mathsf{Mod}(\Gamma_i, \varphi_i) \to \mathsf{Mod}(\Gamma, \varphi)\}_{1\le i\le n}$ whose images partition $\mathsf{Mod}(\Gamma, \varphi)$. (Note the reversal of arrows due to the contravariance of Mod). For example, consider the space of weighted models depicted at the top of Figure 1b. This space is represented by the symbolic subprobability space $(\Gamma_{\alpha\beta}, \top)$. Its splitting into two pieces, corresponding to the two paths through NoisyOr, is captured by the two morphisms $i : (\Gamma_{\alpha\beta}, \top) \to (\Gamma_{\alpha\beta}, \alpha \vee \beta)$ and $j : (\Gamma_{\alpha\beta}, \top) \to (\Gamma_{\alpha\beta}, \neg(\alpha \vee \beta))$ that witness the entailments $\alpha \vee \beta \le \top$ and $\neg(\alpha \vee \beta) \le \top$. The family $\{i, j\}$ forms a cover of $(\Gamma_{\alpha\beta}, \top)$: the corresponding functions

$$\mathsf{Mod}(\Gamma_{\alpha\beta}, \alpha \vee \beta) \xrightarrow{\mathsf{Mod}(i)} \mathsf{Mod}(\Gamma_{\alpha\beta}, \top) \text{ and } \mathsf{Mod}(\Gamma_{\alpha\beta}, \neg(\alpha \vee \beta)) \xrightarrow{\mathsf{Mod}(j)} \mathsf{Mod}(\Gamma_{\alpha\beta}, \top)$$

partition the space $\mathsf{Mod}(\Gamma, \top)$ into $\mathsf{Mod}(\Gamma, \alpha \vee \beta)$ and $\mathsf{Mod}(\Gamma, \neg(\alpha \vee \beta))$, just as depicted at the bottom of Figure 1b.

Given a coverage on a category C, the *sheaves* for that coverage are those presheaves $P : C \to \mathrm{Set}$ that "respect covers" in the sense that any family of elements of P decorating a cover of (Γ, φ) can be uniquely amalgamated into an element of $P(\Gamma, \varphi)$.[2] In the case of $\mathbb{S}$, the sheaves for the finite-partition coverage just defined above are symbolic sets that come with a merging operation. This motivates the following definition:

Definition 5.1. The category of *mergable symbolic sets*, written $\mathsf{SymSet}_{\mathsf{m}}$, is the full subcategory of SymSet consisting of sheaves for the coverage on $\mathbb{S}$ generated by finite partitions.

[2]For more details on abstract sheaf theory, see MacLane and Moerdijk [31, §II-III].

Name	Definition	Empty element	Merge operation
Singleton	$\widehat{\mathbf{1}}$	$r(\star) = \star$	$\star \sqcup \star = \star$
Product	$\widehat{A} \times \widehat{B}$	$\varnothing = (\varnothing, \varnothing)$	$(\widehat{a}, \widehat{b}) \sqcup (\widehat{a}', \widehat{b}') = (\widehat{a} \sqcup \widehat{b}, \widehat{a}' \sqcup \widehat{b}')$
	$\prod_{i \in I} \widehat{A}_i$	$\varnothing = (\varnothing)_{i \in I}$	$(\widehat{a}_i)_{i \in I} \sqcup (\widehat{a}'_i)_{i \in I} = (\widehat{a}_i \sqcup \widehat{a}'_i)_{i \in I}$
Coproduct	$\mathsf{Piecewise}(\coprod_{i \in I} \widehat{A}_i)$	$\varnothing = [\,]$	$[\varphi_i : \mathsf{inj}_i\, \widehat{a}_i]_{i \in I} \sqcup [\varphi'_i : \mathsf{inj}_i\, \widehat{a}'_i]_{i \in I} = [(\varphi_i \vee \varphi'_i) : \mathsf{inj}_i (\widehat{a}_i \sqcup \widehat{a}'_i)]_{i \in I}$
Formula	$\mathbb{B}$	$\varnothing = [\bot]$	$[\psi] \sqcup_{\varphi,\varphi'} [\psi'] = [(\psi \wedge \varphi) \vee (\psi' \wedge \varphi')]$
Exponent	$\widehat{A} \Rightarrow \widehat{B}$	$\varnothing(\_, \_) = \varnothing$	$(\widehat{f} \sqcup \widehat{g})(s, \widehat{a}) = \widehat{f}(s, \widehat{a}) \sqcup \widehat{g}(s, \widehat{a})$

Fig. 6. A selection of mergable symbolic set constructions that follow from Proposition 5.2.

As with the definition of SymSet, fully appreciating this definition requires some unpacking. The following proposition describes when a symbolic set is a sheaf in elementary terms:

Proposition 5.1. A symbolic set $\widehat{A}$ lands in the subcategory $\mathsf{SymSet}_{\mathsf{m}}$ if and only if

(1) $\widehat{A}_{(\Gamma,\bot)}$ contains a single element $\bot_\Gamma$, which will be called the *empty element at* Γ
(2) For all $\hat{a} \in \widehat{A}_{(\Gamma,\varphi)}$ and $\hat{a}' \in \widehat{A}_{(\Gamma,\varphi')}$ with φ and φ' disjoint, there exists a unique element $\hat{a} \sqcup_{\varphi,\varphi'} \hat{a}'$ of $\widehat{A}_{(\Gamma,\varphi+\varphi')}$, called the *merge* of $\hat{a}$ and $\hat{a}'$, such that $(\hat{a} \sqcup_{\varphi,\varphi'} \hat{a}')|_\varphi = \hat{a}$ and $(\hat{a} \sqcup_{\varphi,\varphi'} \hat{a}')|_{\varphi'} = \hat{a}'$.

The subscripts on $\varnothing_\Gamma$ and $\sqcup_{\varphi,\varphi'}$ will be omitted when clear from context.

In other words, a symbolic set is mergable precisely when it comes equipped with a merge operation $\sqcup$, including for the edge case of a "nullary merge" $\varnothing$. Once again, the symbolic set of functions provides a canonical example: $\mathsf{Fn}\, A$ is mergable for any set A, with $\varnothing$ given by the empty function with empty domain and $\sqcup_{\varphi,\varphi'}$ given by the unioning the graphs of functions with disjoint domains φ, φ'. But there are also more abstract examples; a particularly important one is $\mathsf{Piecewise}\, \widehat{A}$, with $\varnothing$ given by the empty piecewise definition $[\,]$ and $\sqcup$ by the disjoint union of piecewise definitions.

Like SymSet, the category $\mathsf{SymSet}_{\mathsf{m}}$ is a Grothendieck topos by definition, which immediately gives a mergable analog of Proposition 3.2:

Proposition 5.2. $\mathsf{SymSet}_{\mathsf{m}}$ is Cartesian closed and has all limits and colimits.

The monad Piecewise plays a special role in the interaction between SymSet and $\mathsf{SymSet}_{\mathsf{m}}$:

Proposition 5.3. Piecewise is the monad induced by the adjunction $\mathsf{SymSet}_{\mathsf{m}} \underset{i}{\overset{a}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}} \mathsf{SymSet}$, where i is the inclusion of sheaves into presheaves and a is *sheafification* [31, §V.3].

This shows how the somewhat-complicated definition of Piecewise arises naturally from a sheaf-theoretic perspective: $\mathsf{Piecewise}\, \widehat{A}$ is the canonical way of forcing a given symbolic set $\widehat{A}$ to be mergable. It follows that Piecewise also preserves mergability.

Proposition 5.3 lets us use standard formulae for computing limits and colimits in sheaves to unpack Proposition 5.2, just as we did for SymSet. The result is shown in Figure 6. First, limits in sheaves are the same as in presheaves. This implies the singleton, product, and indexed product in $\mathsf{SymSet}_{\mathsf{m}}$ are the same as in SymSet; their empty elements and merge operations all work componentwise. Next, colimits in sheaves are the sheafification of colimits in presheaves. Hence indexed coproducts in $\mathsf{SymSet}_{\mathsf{m}}$ are piecewise indexed coproducts in SymSet. Each element of the indexed coproduct is a piecewise definition $[\varphi_i : \hat{a}_i]_{i \in I}$ with exactly one case $\varphi_i : \hat{a}_i$ for each i in the indexing family I and at most finitely many elements not equal to $\varnothing$. Merging of such elements operates case by case. The formula for coproducts also shows that $\mathbb{B}$ is the coproduct of $\widehat{\mathbf{1}}$ with itself in $\mathsf{SymSet}_{\mathsf{m}}$, in light of Proposition 3.3; the empty element of $\mathbb{B}$ is the unsatisfiable

$$\begin{aligned}
&\mathsf{Bool} = \mathbf{1} + \mathbf{1}\\
&\mathsf{Store} = \textstyle\prod_{x \in \mathsf{Var}} \mathsf{Bool}\\
&[\![e]\!] : \mathsf{Store} \to \mathsf{Bool}\\
&[\![c]\!] : \mathsf{Store} \to \mathsf{Gen}\ \mathsf{Store}
\end{aligned}$$

$$\begin{aligned}
&\mathcal{S}_{\mathsf{m}}[\![\mathsf{Bool}]\!] = \mathbb{B}\\
&\mathcal{S}_{\mathsf{m}}[\![\mathsf{Store}]\!] = \textstyle\prod_{x \in \mathsf{Var}} \mathbb{B}\\
&\mathcal{S}_{\mathsf{m}}[\![\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2]\!]_{(\Gamma,\varphi)}(\hat{s}) = \nu\,(\Gamma_1 + \Gamma_2).\,(\hat{s}_1 \sqcup_{\varphi\wedge\psi,\varphi\wedge\neg\psi} \hat{s}_2) \text{ where}\\
&\quad \mathcal{S}_{\mathsf{m}}[\![e]\!]_{(\Gamma,\varphi)}(\hat{s}) = [\psi]\\
&\quad \mathcal{S}_{\mathsf{m}}[\![c_1]\!]_{(\Gamma,\varphi\wedge\psi)}(\hat{s}) = \nu\,\Gamma_1.\,\hat{s}_1\\
&\quad \mathcal{S}_{\mathsf{m}}[\![c_2]\!]_{(\Gamma,\varphi\wedge\neg\psi)}(\hat{s}) = \nu\,\Gamma_2.\,\hat{s}_2\\
&\quad \{x \mapsto [\varphi], \dots\} \sqcup_{\psi,\psi'} \{x \mapsto [\varphi'], \dots\} = \{x \mapsto [(\varphi \wedge \psi) \vee (\varphi' \wedge \psi')], \dots\}
\end{aligned}$$

Fig. 7. Alternative model of Imp in the symbolic metalanguage (left) and its unfolding into $\mathsf{SymSet}_{\mathsf{m}}$ (right).

formula and merging is defined in terms of disjunction. Finally, sheaves form an exponential ideal in presheaves. This implies that the exponential $\widehat{A} \Rightarrow \widehat{B}$ in symbolic sets is mergable if $\widehat{B}$ is; we will return to exponentials in Section 7.

The monad Gen also interacts nicely with mergable symbolic sets: it preserves sheafhood, justifying the use of Gen in the symbolic metalanguage for building symbolic semantics with state merging. (A proof can be found in §F.)

Proposition 5.4. If $\widehat{A}$ is mergable then so is $\mathsf{Gen}\,\widehat{A}$.

5.1 Imp with merging

This section concludes with a demonstration of how $\mathsf{SymSet}_{\mathsf{m}}$ can be used in place of SymSet to obtain a probabilistic symbolic semantics for Imp that performs state merging.

Propositions 5.2, 5.3, and 5.4 together establish that $\mathsf{SymSet}_{\mathsf{m}}$ is Cartesian closed, has all limits and colimits, and supports both the monads Gen and Piecewise. This makes for a model of the symbolic metalanguage where types denote *mergable* symbolic sets. The left side of Figure 7 sketches how the symbolic metalanguage can be used to build a state-merging semantics for Imp. Unlike the interpretation in Section 4, the metatype Store of stores is defined to be an indexed product over Bool rather than Piecewise Bool; similarly, the return type of computations has been simplified to Gen Store. This is because the semantics being built is designed to be interpreted in sheaves; Piecewise, being sheafification, has no effect in this context. Given the interpretation of Imp semantic types, the interpretations of expressions and commands are completely standard. The full details can be found in §B.1.

Unfolding the metalanguage interpretation into $\mathsf{SymSet}_{\mathsf{m}}$ gives a state-merging analog $\mathcal{S}_{\mathsf{m}}[\![-]\!]$ of the semantics $\mathcal{S}[\![-]\!]$ derived in Section 4.2. The most interesting part of this new semantics is shown on the right of Figure 7: the semantics of `if` combines the stores produced by both branches into a single store using $\sqcup$. This merge operation $\sqcup$ is in turn mechanically derived from the definition of the metatype Store as an indexed product of coproducts; as one might expect, it combines symbolic stores variable-wise, using the merge operation on $\mathbb{B}$ to combine the values bound to each variable.

6 Correctness via Gluing

In the previous sections, we saw how SymSet and $\mathsf{SymSet}_{\mathsf{m}}$ can be used to quickly build probabilistic symbolic semantics via the symbolic metalanguage. In this section we augment this framework for semantics with a framework for proving correctness.

Intuitively, a symbolic semantics is correct if, for all commands c, the symbolic semantics $\mathcal{S}[\![c]\!]$ simulates the effect of running the *concrete semantics* $C[\![c]\!]$ of c on all concrete instantiations for symbolic variables. This property is formalized by the following commutative square, originally

$$\left(\widehat{A}_{(\Gamma,\varphi)} \times \mathsf{Mod}(\Gamma,\varphi) \xrightarrow{i_{(\Gamma,\varphi)}} A\right)_{(\Gamma,\varphi)\in\mathbb{S}} \overset{\text{curry}}{\rightsquigarrow} \left(\widehat{A} \xrightarrow{i} \mathsf{Fn}\, A\right) \overset{\text{make relational}}{\rightsquigarrow} \left(\widehat{A} \xleftarrow{i} \widehat{R} \xrightarrow{j} \mathsf{Fn}\, A\right)$$

Fig. 8. The steps leading from the traditional notion of instantiation to our relational formulation.

introduced by King [27]:

$$\begin{array}{ccc} (\hat{s}, m) & \xmapsto{\text{symbolic execution}} & (\mathcal{S}[\![c]\!](\hat{s}), m) \\ \Big\downarrow{\scriptstyle \text{instantiate symbolic variables}} & & \Big\downarrow{\scriptstyle \text{instantiate symbolic variables}} \\ \hat{s}[m] & \xmapsto[\text{concrete execution}]{} & C[\![c]\!](\hat{s}[m]) \end{array} \tag{King}$$

This commutative square, which has since become the standard correctness criterion for symbolic executors [2, 14, 15, 59], states that if one symbolically executes a command c under initial store $\hat{s}$ and then instantiates all symbolic variables in the result with m, this gives the same store as concretely executing c under the instantiation $\hat{s}[m]$.

While this correctness property is intuitive, proving it takes quite a bit of work. First one needs to fill in the horizontal arrows of (King), which includes a concrete semantics $C[\![-]\!]$ to judge the correctness of $\mathcal{S}[\![-]\!]$ against. Then one needs a notion of *instantiation* to fill in the vertical arrows of (King). This is nontrivial to define, as it requires specifying how each symbolic object $\hat{x}$ can be instantiated against a given model m to obtain a concrete counterpart x; for Imp, this means defining instantiation for Booleans, piecewise Booleans, formulas, stores, piecewise stores, and computations that may generate fresh names. Finally, there remains the matter of actually proving that the square commutes. This is also nontrivial, as one must systematically check that the symbolic semantics of each language construct properly simulates its concrete counterpart.

The goal of this section is to validate the following reasoning principle:

The Lifting Principle. Every probabilistic symbolic semantics constructed using the symbolic metalanguage is automatically correct, no matter whether it is interpreted in SymSet or $\mathsf{SymSet}_{\mathsf{m}}$.

The Lifting Principle makes the symbolic metalanguage useful not just for *building* a probabilistic symbolic semantics, but also for proving it correct. This saves a significant amount of work—for instance, it automatically completes all of the steps described above for the symbolic semantics of Imp from Section 4.2, as well as for the merging semantics from Section 5.

Our justification of the Lifting Principle follows the pattern laid out in Sections 3 and 5: we will introduce a new category and show that it has enough structure to interpret all metalanguage constructs. This new category, called Inst (for *instantiation span*, to be defined below), is in essence a category of *proof-relevant binary logical relations* that connects SymSet-based interpretations of metalanguage constructs to their concrete counterparts. To build Inst, we make use of *gluing* [52, §7.7], a categorical analog of logical relations. The theory of gluing lets us quickly prove a *Lifting Theorem* (Theorem 6.1) which shows that the symbolic metalanguage always yields a correct symbolic semantics when interpreted in SymSet. We then similarly use gluing to build a *mergable* analog of Inst and to prove an analogous lifting theorem for $\mathsf{SymSet}_{\mathsf{m}}$ (Theorem 6.2).

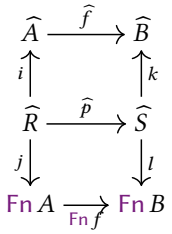
In essence, gluing is a technique for reducing complicated proofs to well-established facts about comma categories. To make gluing applicable, we translate the key notion of *instantiation* needed to specify (King) into the shape of a comma category. This translation process is shown in Figure 8. On the far left is the naive notion of instantiation—a family of functions $i_{(\Gamma,\varphi)}$ that send symbolic values $\hat{a}$ and models m to concrete values $i_{(\Gamma,\varphi)}(\hat{a}, m)$. First, we *curry* this family into a family of functions $\widehat{A}_{(\Gamma,\varphi)} \to (\mathsf{Mod}(\Gamma,\varphi) \to A)$, which now fits the shape of a *symbolic function* from $\widehat{A}$ to

$\mathsf{Fn}\,A$. Then, we move from functions to *proof-relevant relations*, replacing the morphism i with a *span* (i, j) as shown on the right of Figure 8. These spans form a comma category:

Definition 6.1. The category Inst of *instantiation spans* is the comma category $\mathrm{id}_{\mathsf{SymSet}} \downarrow \mathsf{span}$, where $\mathsf{span} : \mathsf{SymSet} \times \mathsf{Set} \to \mathsf{SymSet}$ is the functor $(\widehat{A}, A) \mapsto \widehat{A} \times \mathsf{Fn}\,A$.

Objects of Inst are (isomorphic to) tuples $(\widehat{A}, A, \widehat{R}, i, j)$ that form spans as shown on the right of Figure 8. This span "decorates" the pair of objects $(\widehat{A}, A)$ with the tuple $(\widehat{R}, i, j)$, which one can think of as a binary logical relation between the symbolic set $\widehat{A}$ and its concrete counterpart A: each element $r \in \widehat{R}_{(\Gamma,\varphi)}$ is a "proof" that $i_{(\Gamma,\varphi)}(r)$ is related to $j_{(\Gamma,\varphi)}(r)$.

Morphisms of Inst have the form shown immediately to the right. A morphism $(\widehat{f}, f, \widehat{p})$ from $(\widehat{A}, A, \widehat{R}, i, j)$ to $(\widehat{B}, B, \widehat{S}, k, l)$ consists of (1) a symbolic function $\widehat{f} : \widehat{A} \to \widehat{B}$, (2) an ordinary function $f : A \to B$, and (3) a symbolic function $\widehat{p} : \widehat{R} \to \widehat{S}$ such that the diagram commutes. This can be thought of as a pair of morphisms $(\widehat{f}, f)$ that is decorated by a proof $\widehat{p}$ that $(\widehat{f}, f)$ satisfies the fundamental property of binary logical relations; namely, that two inputs related by $\widehat{R}$ are sent by $\widehat{f}$ and $\mathsf{Fn}\,f$ respectively to outputs related by $\widehat{S}$.



The objects and morphisms of Inst work together to provide a crisp formulation of correctness: a symbolic semantics $\mathcal{S}[\![-]\!]$ in SymSet is correct with respect to a concrete semantics $C[\![-]\!]$ in Set if the pair $(\mathcal{S}[\![-]\!], C[\![-]\!])$ can be decorated with enough structure so as to constitute a model of the language in Inst—a categorical analog of the fundamental theorem of logical relations. This can be made precise by asking for a *lifting* of the pair $(\mathcal{S}[\![-]\!], C[\![-]\!])$ along the following functor that forgets all decorations:

Definition 6.2. Inst comes with a canonical projection $p_{\mathsf{Inst}} : \mathsf{Inst} \to \mathsf{SymSet} \times \mathsf{Set}$ defined by sending each object $(\widehat{A}, A, \widehat{R}, i, j)$ to $(\widehat{A}, A)$ and each morphism $(\widehat{f}, f, \widehat{p})$ to $(\widehat{f}, f)$.

Validating the Lifting Principle for SymSet requires lifting all symbolic metalanguage constructs. This reduces to lifting limits, colimits, exponentials, Gen, and Piecewise. Liftings for the first three of these follow from general facts about comma categories [52, §7.7]:

Proposition 6.1. Exponentials, limits, and colimits lift along p_{Inst}.

This immediately shows that all the constructions described in Section 3.2.3 are automatically correct with respect to their concrete counterparts. We note in passing that Proposition 6.1 already goes beyond existing frameworks for proving correctness of a symbolic semantics: its lifting of exponentials gives the first account of correctness for a symbolic semantics at higher type.

To obtain the Lifting Principle for SymSet, it only remains to lift the monads Piecewise and Gen. These liftings take the form of monads on Inst that relate symbolic computations and random variables. We sketch the definitions of these lifted monads here; details can be found in §E. The lifting for Piecewise relates symbolic unions to the piecewise functions they represent: an element $[\varphi_i : \hat{a}_i]_{i \in I}$ of $(\mathsf{Piecewise}\,\widehat{A})_{(\Gamma,\varphi)}$ is related to a function $f : \mathsf{Mod}(\Gamma, \varphi) \to A$ if each $\hat{a}_i$ is related to the corresponding subfunction $f|_{\varphi_i}$. The lifting for Gen relates symbolic computations running in Gen to concrete computations running in the discrete probability monad Dist[3]: an element $\nu\,\Delta.\,\widehat{a}$ of $(\mathsf{Gen}\,\widehat{A})_{(\Gamma,\varphi)}$ is related to a function $f : \mathsf{Mod}(\Gamma, \varphi) \to \mathsf{Dist}\,A$ if there exists a function $f' : \mathsf{Mod}(\Gamma + \Delta, \varphi) \to A$ such that (1) $\widehat{a} \in \widehat{A}_{(\Gamma+\Delta,\varphi)}$ is related to f' and (2) f is the Kleisli composition of f' with the distribution generated by sampling all the variables in Δ according to their types. These definitions, in combination with Proposition 3.2, yield our first lifting theorem:

[3]Concretely, Dist sends a set A to the set of finitely-supported functions $A \to [0, 1]$ that sum to 1.

Theorem 6.1 (Lifting for SymSet)**.** The following structure in $\mathsf{SymSet} \times \mathsf{Set}$ lifts along p_{Inst}:

(1) Exponentials, limits, and colimits
(2) The monads $(\mathsf{Piecewise}, \mathsf{id}_{\mathsf{Set}})$ and $(\mathsf{Gen}, \mathsf{Dist})$, as well as all of their composites
(3) The morphism $(\mathsf{gen}\, p, \mathsf{Ber}\, p) : (\widehat{\mathbf{1}}, 1) \to (\mathsf{Gen}(\mathsf{Piecewise}(\widehat{\mathbf{1}} + \widehat{\mathbf{1}})), \mathsf{Dist}(1 + 1))$ for $p \in [0, 1]$

We will write $\overline{\mathsf{Piecewise}}$ and $\overline{\mathsf{Gen}}$ for the liftings of Piecewise and Gen respectively. The same procedure works with $\mathsf{SymSet}_{\mathsf{m}}$ in place of SymSet, yielding our second lifting theorem:

Theorem 6.2 (Lifting for $\mathsf{SymSet}_{\mathsf{m}}$)**.** The functor span lands in the subcategory $\mathsf{SymSet}_{\mathsf{m}} \times \mathsf{Set}$ of $\mathsf{SymSet} \times \mathsf{Set}$, and the same structure listed in Theorem 6.1 lifts along the canonical projection $(\mathsf{id}_{\mathsf{SymSet}_{\mathsf{m}}} \downarrow \mathsf{span}) \to \mathsf{SymSet}_{\mathsf{m}} \times \mathsf{Set}$.

Our final theorem shows that our relational formulation of correctness smoothly generalizes the traditional one: it implies that any symbolic semantics built using only the first-order type formers of the symbolic metalanguage is automatically correct in the sense of (King).

Theorem 6.3. The category $\mathsf{id}_{\mathsf{SymSet}} \downarrow \mathsf{Fn}$, whose objects are (curried) instantiation operations and whose morphisms are (curried) squares of shape (King), embeds fully and faithfully into Inst. Moreover, this embedding preserves limits and colimits, and picks out a subcategory of Inst that is closed under applications of the lifted monads $\overline{\mathsf{Gen}}$ and $\overline{\mathsf{Piecewise}}$.

To demonstrate the Lifting Principle in action, we briefly unpack what it gives us when applied to the semantics of Imp given in Figure 3. First, it yields a concrete semantics $C[\![-]\!]$ that reinterprets Figure 3 in Set with Piecewise interpreted by the identity monad and Gen by Dist. This recovers the standard semantics of Imp where an expression denotes a function $\mathsf{Store} \to \{0, 1\}$ and a command denotes a function $\mathsf{Store} \to \mathsf{Dist}\, \mathsf{Store}$. Second, the Lifting Principle yields instantiation operations $i^{\widehat{A}}_{(\Gamma,\varphi)} : \widehat{A}_{(\Gamma,\varphi)} \times \mathsf{Mod}(\Gamma, \varphi) \to A$ that connect each symbolic set $\widehat{A}$ used in the symbolic semantics with its counterpart A in the concrete semantics. Precise definitions for these operations are given by unpacking Theorems 6.1 and 6.3. This yields:

- An operation $i^{\mathbb{B}}_{(\Gamma,\varphi)} : \mathbb{B}_{(\Gamma,\varphi)} \times \mathsf{Mod}(\Gamma, \varphi) \to 1 + 1$ for formulas that sends each pair $([\psi], m)$ to the result of evaluating the formula ψ at a given model m;
- An operation $i^{\widehat{\mathsf{Store}}}_{(\Gamma,\varphi)} : \widehat{\mathsf{Store}}_{(\Gamma,\varphi)} \times \mathsf{Mod}(\Gamma, \varphi) \to \mathsf{Store}$ for stores, defined variable-wise by sending each pair $(\hat{s}, m)$ to the concrete store $\{x \mapsto i^{\mathbb{B}}_{(\Gamma,\varphi)}(\hat{s}(x), m)\}$;
- An operation $i^{\mathsf{Piecewise}\,\widehat{\mathsf{Store}}}_{(\Gamma,\varphi)} : (\mathsf{Piecewise}\,\widehat{\mathsf{Store}})_{(\Gamma,\varphi)} \times \mathsf{Mod}(\Gamma, \varphi) \to \mathsf{Store}$ for piecewise stores that is defined by cases, sending $([\varphi_j : \hat{s}_j]_{j\in J}, m)$ to the concrete store $i^{\widehat{\mathsf{Store}}}_{(\Gamma,\varphi_j)}(\hat{s}_j, m)$ in case $m \models \varphi_j$, for all $j \in J$;
- An operation $i^{\mathsf{Gen}(\mathsf{Piecewise}\,\widehat{\mathsf{Store}})}_{(\Gamma,\varphi)} : (\mathsf{Gen}(\mathsf{Piecewise}\,\widehat{\mathsf{Store}}))_{(\Gamma,\varphi)} \times \mathsf{Mod}(\Gamma, \varphi) \to \mathsf{Dist}\,\mathsf{Store}$ for computations that sends each $(\nu\Delta.\, [\varphi_j : \hat{s}_j]_{j\in J}, m)$ to the distribution over concrete stores given by sampling a model m_Δ from Δ and then producing $i^{\mathsf{Piecewise}\,\widehat{\mathsf{Store}}}_{(\Gamma+\Delta,\varphi)}([\varphi_j : \hat{s}_j]_{j\in J}, m + m_\Delta)$.

Finally, the Lifting Principle establishes that (King) commutes for this notion of instantiation. When applied to the NoisyOr program from Section 2, this commutativity property says precisely that the final symbolic union $[\alpha \vee \beta : \hat{s}_{\mathsf{t}}, \neg(\alpha \vee \beta) : \hat{s}_{\mathsf{e}}]$ produced by symbolic execution represents, via weighted model counting, the expected distribution over concrete stores.

7 From Imp to Fun

So far all of our examples have focused on the toy language Imp. This final section demonstrates how the framework we have developed scales to a host of realistic language features, by building a correct-by-construction probabilistic symbolic semantics for a very different language Fun.

$$
\begin{array}{lll}
\sigma, \tau ::= \texttt{Unit} & [\![\texttt{Unit}]\!] = \mathbf{1} & \mathcal{S}_{\mathrm{m}}[\![\texttt{Unit}]\!] = \widehat{\mathbf{1}} \\
\mid \texttt{Bool} & [\![\texttt{Bool}]\!] = \mathbf{1} + \mathbf{1} & \mathcal{S}_{\mathrm{m}}[\![\texttt{Bool}]\!] = \mathbb{B} \\
\mid \sigma + \tau & [\![\sigma + \tau]\!] = [\![\sigma]\!] + [\![\tau]\!] & \mathcal{S}_{\mathrm{m}}[\![\sigma + \tau]\!] = \mathrm{Piecewise}(\mathcal{S}_{\mathrm{m}}[\![\sigma]\!] + \mathcal{S}_{\mathrm{m}}[\![\tau]\!]) \\
\mid \sigma \times \tau & [\![\sigma \times \tau]\!] = [\![\sigma]\!] \times [\![\tau]\!] & \mathcal{S}_{\mathrm{m}}[\![\sigma \times \tau]\!] = \mathcal{S}_{\mathrm{m}}[\![\sigma]\!] \times \mathcal{S}_{\mathrm{m}}[\![\tau]\!] \\
\mid \sigma \to \tau & [\![\sigma \to \tau]\!] = [\![\sigma]\!] \Rightarrow [\![\tau]\!] & \mathcal{S}_{\mathrm{m}}[\![\sigma \to \tau]\!] = \mathcal{S}_{\mathrm{m}}[\![\sigma]\!] \Rightarrow \mathcal{S}_{\mathrm{m}}[\![\tau]\!] \\
\mid \texttt{List}\ \tau & [\![\texttt{List}\ \tau]\!] = \coprod_{n \in \mathbb{N}} [\![\tau]\!]^n & \mathcal{S}_{\mathrm{m}}[\![\texttt{List}\ \tau]\!] = \mathrm{Piecewise}(\coprod_{n \in \mathbb{N}} \mathcal{S}_{\mathrm{m}}[\![\tau]\!]^n) \\
\mid \texttt{Prob}\ \tau & [\![\texttt{Prob}\ \tau]\!] = \mathrm{Gen}\ [\![\tau]\!] & \mathcal{S}_{\mathrm{m}}[\![\texttt{Prob}\ \tau]\!] = \mathrm{Gen}\ \mathcal{S}_{\mathrm{m}}[\![\tau]\!]
\end{array}
$$

Fig. 9. Syntax (left), metalanguage interpretations (middle), and mergable semantics (right) of Fun types.

The syntax of Fun types is shown on left in Figure 9. Fun is a higher-order functional language with `List`s and a `Prob`ability monad. The terms and typing rules for Fun are standard.The semantics of Fun, defined using the symbolic metalanguage, is shown in the middle of Figure 9. Note that lists are defined as an indexed coproduct of finite approximations. This is an instance of a general recipe: the symbolic metalanguage readily supports other algebraic data types in a similar manner.

Unfolding these metalanguage definitions into $\mathrm{SymSet}_{\mathrm{m}}$ gives the semantics of Fun types shown on right in Figure 9. The Fun function space is interpreted by the exponential $\Rightarrow$ in $\mathrm{SymSet}_{\mathrm{m}}$, defined by the formula in Figure 2. Elements f of $(\widehat{A} \Rightarrow \widehat{B})_{(\Gamma,\varphi)}$ are functions that take in a "future world" (Γ', φ') reachable from the "current" one via a renaming $r : (\Gamma, \varphi) \to (\Gamma', \varphi')$, alongside an input $\widehat{a} \in \widehat{A}_{(\Gamma',\varphi')}$, and produce an element $f_{(\Gamma',\varphi')}(r, \widehat{a})$ of $\widehat{B}_{(\Gamma',\varphi')}$. Lists are interpreted by the formula for mergable indexed coproducts from Figure 6. Using $\mathrm{SymSet}_{\mathrm{m}}$ as the semantic domain gives each semantic type its own merge operation. This makes Figure 9 the first denotational semantics of *type-directed state merging*, a merging strategy pioneered by Rosette [54]. The merging operations derived via $\mathrm{SymSet}_{\mathrm{m}}$ recover their Rosette implementations. For instance, elements of $\mathcal{S}_{\mathrm{m}}[\![\texttt{List}\ \tau]\!]_{(\Gamma,\varphi)}$ are of shape $[\varphi_i : \mathsf{inj}_{n_i}(\vec{v}_i)]_{i \in I}$ for some finite set I, which naturally organizes symbolic lists $\vec{v}_i$ by their length n_i, with merge defined accordingly; this exactly matches Rosette's representation of lists, which turns out to be important for achieving good performance in practice.

Figure 10 shows selected cases of the semantics of Fun terms. The metalanguage definitions are shown on left, and their unfoldings on right. The semantics of $\lambda x.\, e$ is a function that, given an input v at future world (Γ', φ') reachable by r, runs e with environment γ coerced into the new world via renaming and restriction and then extended by $x \mapsto \hat{v}$. Dually, $e_1\ e_2$ immediately invokes e_1 at the current world, reachable via $\mathsf{id}_{(\Gamma,\varphi)}$, with e_2 as input. List operations are interpreted length-wise: `nil` produces a singleton with the empty list, `cons` inserts e_1 into every list $\vec{v}_i$ produced by e_2, and `rec` folds over every list $\vec{v}_i$ of length n_i produced by e_1 and then merges the results.

This symbolic semantics is correct by construction, via the Lifting Principle.

8 Discussion

From symbolic semantics to implementation. While the symbolic semantics of Imp in Figure 4 and its implementation in Figure 5 are quite similar, there is still a gap between the two. Going from semantics to code requires making many representation decisions: for instance, Figure 5 represents the store $\hat{s}$ as a finite map data structure, path conditions φ as abstract syntax trees, and symbolic contexts Γ as a weight map and an integer counter. More realistic implementations would use sophisticated data structures, such as binary decision diagrams to represent formulae compactly [16, 23, 33], and may contain additional optimizations for good performance. Consequently, using the present framework to establish end-to-end correctness of a practical implementation will require showing that such low-level details faithfully encode the symbolic semantics.

$$[\![\Gamma \vdash e : \tau]\!] : [\![\Gamma]\!] \to [\![\tau]\!]$$
$$[\![\lambda x.\, e]\!](\gamma) = \lambda v.\, [\![e]\!](\gamma, v)$$
$$[\![e_1\ e_2]\!](\gamma) = [\![e_1]\!](\gamma)([\![e_2]\!](\gamma))$$
$$[\![\texttt{flip}\ p]\!](\gamma) = \texttt{gen}\ p$$
$$[\![\texttt{nil}]\!](\gamma) = \mathrm{inj}_0()$$
$$[\![\texttt{cons}(e_1, e_2)]\!](\gamma) =$$
$$\quad \text{case } [\![e_2]\!](\gamma) \text{ of } \{\mathrm{inj}_n(\vec{v}) \Rightarrow \mathrm{inj}_{n+1}(v, \vec{v})\}_{n\in\mathbb{N}}$$
$$\quad \text{where } [\![e_1]\!](\gamma) = v$$
$$[\![\texttt{rec}(e_1, e_2, xyz.e_3)]\!](\gamma) =$$
$$\quad \text{case } [\![e_2]\!](\gamma) \text{ of } \{\mathrm{inj}_n(\vec{v}) \Rightarrow \mathrm{go}_n(\vec{v})\}_{n\in\mathbb{N}}$$
$$\quad \text{where } \mathrm{go}_0() = [\![e_2]\!](\gamma)$$
$$\mathrm{go}_{n+1}(v, \vec{v}) = [\![e_3]\!](\gamma[x \mapsto v, y \mapsto \vec{v}, z \mapsto \mathrm{go}_n(\vec{v})])$$

$$\mathcal{S}_{\mathrm{m}}[\![\Gamma \vdash e : \tau]\!] : \mathcal{S}_{\mathrm{m}}[\![\Gamma]\!] \to \mathcal{S}_{\mathrm{m}}[\![\tau]\!]$$
$$\mathcal{S}_{\mathrm{m}}[\![\lambda x.\, e]\!]_{(\Gamma,\varphi)}(\gamma) = \left((r, v) \mapsto \mathcal{S}_{\mathrm{m}}[\![e]\!]_{(\Gamma',\varphi')}(r(\gamma)|_{\varphi'}[x \mapsto v])\right)_{(\Gamma',\varphi')}$$
$$\mathcal{S}_{\mathrm{m}}[\![e_1\ e_2]\!]_{(\Gamma,\varphi)}(\gamma) = \mathcal{S}_{\mathrm{m}}[\![e_1]\!]_{(\Gamma,\varphi)}(\mathrm{id}_{(\Gamma,\varphi)}, \mathcal{S}_{\mathrm{m}}[\![e_2]\!]_{(\Gamma,\varphi)}(\gamma))$$
$$\mathcal{S}_{\mathrm{m}}[\![\texttt{flip}\ p]\!]_{(\Gamma,\varphi)}(\gamma) = \nu(\alpha : \mathrm{Ber}\, p).\, [\alpha]$$
$$\mathcal{S}_{\mathrm{m}}[\![\texttt{nil}]\!]_{(\Gamma,\varphi)}(\gamma) = [\varphi : \mathrm{inj}_0()]$$
$$\mathcal{S}_{\mathrm{m}}[\![\texttt{cons}(e_1, e_2)]\!]_{(\Gamma,\varphi)}(\gamma) = [\varphi_i : \mathrm{inj}_{n_i+1}(v|_{\varphi_i}, \vec{v}_i)]_{i\in I} \text{ where}$$
$$\quad \mathcal{S}_{\mathrm{m}}[\![e_1]\!]_{(\Gamma,\varphi)} = v$$
$$\quad \mathcal{S}_{\mathrm{m}}[\![e_2]\!]_{(\Gamma,\varphi_i)} = [\varphi_i : \mathrm{inj}_{n_i}(\vec{v}_i)]_{i\in I}$$
$$\mathcal{S}_{\mathrm{m}}[\![\texttt{rec}(e_1, e_2, xyz.e_3)]\!](\gamma) = \bigsqcup_{i\in I} \mathrm{go}_{n_i,(\Gamma,\varphi_i)}(\vec{v}_i) \text{ where}$$
$$\quad \mathcal{S}_{\mathrm{m}}[\![e_1]\!](\gamma) = [\varphi_i : \mathrm{inj}_{n_i}(\vec{v}_i)]_{i\in I}$$
$$\quad \mathrm{go}_n : \mathcal{S}_{\mathrm{m}}[\![\sigma]\!]^n \to \mathcal{S}_{\mathrm{m}}[\![\tau]\!]$$
$$\quad \mathrm{go}_{0,(\Gamma,\varphi)}() = \mathcal{S}_{\mathrm{m}}[\![e_2]\!]_{(\Gamma,\varphi)}(\gamma)$$
$$\quad \mathrm{go}_{n+1,(\Gamma,\varphi)}(v, \vec{v}) = \mathcal{S}_{\mathrm{m}}[\![e_3]\!]_{(\Gamma,\varphi)}(\gamma[x \mapsto v, y \mapsto \vec{v}, z \mapsto \mathrm{go}_{n,(\Gamma,\varphi)}(\vec{v})])$$

Fig. 10. Selected cases of the metalanguage intepretation (left) and unfolded semantics (right) of Fun terms.

In future work, we hope to close this gap using *realizability*, a well-studied technique for relating program values to the mathematical objects they represent [56]. Recent work has shown how realizability can be used for modular verification of tricky low-level code [4, 5]; this so-called *realistic* realizability forms the basis for modern approaches to specifying language interoperability at both the source and application-binary level [37, 60]. We hope to similarly use realizability to relate the mathematical objects appearing in our denotational semantics to the low-level operational behavior of an implementation.

Generalizations of symbolic sets. While we have restricted attention to finite discrete probability, our setup readily generalizes to other settings. This is because the three categories SymSet, $\mathrm{SymSet}_{\mathrm{m}}$, and Inst central to our framework were constructed in a generic manner: their definitions depend only on the category $\mathbb{S}$ of symbolic subprobability spaces and the functor Mod reflecting how symbolic subprobability spaces are related to actual ones. This gives our setup significant flexibility. For instance, suppose one replaced the objects of $\mathbb{S}$ with pairs (Γ, φ), where φ is a formula of integer arithmetic and Γ maps variables to distributions over integers, and replaced Mod with the functor that interprets each formula φ as a subset of the space of integer-valued assignments over Γ in the standard way. This yields a version of symbolic sets that models random variables with potentially-infinite support (e.g., Poisson), and a version of instantiation spans that validates the corresponding Lifting Principle. Replacing the objects of $\mathbb{S}$ with formulas of real arithmetic, and Mod with the functor that interprets each real arithmetic formula φ as a measurable subset of a measurable space, yields a version of symbolic sets that supports continuous probability, with a Lifting Principle that guarantees correctness with respect to a measure-theoretic concrete semantics.

9 Related work

Semantics of symbolic execution. The semantics of symbolic execution has a long history going back to King [27]. Traditionally, these semantics are operational in flavor and restricted to first-order imperative languages [14, 15]. Voogd et al. [59] are the first to give a denotational semantics of symbolic execution, by modeling symbolic values as functions out of the space of models. This semantics is well suited for high-level reasoning, but less suited to capturing implementation details

in the same way that symbolic sets do, and does not account for the generation of fresh symbolic variables or symbolic higher-order functions. Formal models of real-scale symbolic execution platforms cover these more advanced languages features [18, 39], but they are tailor-made for proving the correctness of an implementation; as such, they contain many low-level details that hinder high-level reasoning and probabilistic adaptation. While general frameworks for correct-by-construction symbolic semantics exist [2, 3, 57], they are restricted to first order, and yield symbolic semantics given by operational rules that make it difficult to validate reasoning principles.

Semantics of probabilistic symbolic execution. Many probabilistic adaptations of symbolic execution come with a probabilistic adaptation of symbolic semantics and a manual proof of correctness [33, 51, 58]. Of these, our framework is most related to Roulette [33], which is a probabilistic adaptation of Rosette [39, 54]. In particular, the semantics of Fun given in Section 7 models the typed and purely functional fragment of Roulette [33]. Compared to the semantics of Roulette, Section 7 is most notably missing higher-order mutable references—a feature known to be difficult for denotational methods. On the other hand, our semantics captures fine-grained details of Rosette's merging strategy and is correct by construction, whereas Roulette's semantics abstracts over merging by modeling symbolic values as functions and has a highly nontrivial correctness argument. Finally, there are several other semantic models and implementations of PPLs that strongly resemble symbolic execution, even if they do not make explicit mention of it [8, 16, 20, 29, 41, 45, 48–50].

Sheaves as a model of branching computation. Using sheaves to capture computational effects of a branching nature has precedent in both logic and programming language semantics. An early example is Scott's Boolean-valued models of non-standard analysis [43], which uses sheaves on a complete Boolean algebra of events to capture the idea that measure-theoretic concepts are ambiently parameterized by a measurable space. This setup has since been generalized by Jackson [24] to give a synthetic account of measure theory relative to an arbitrary locale. A separate but related application comes from Altenkirch et al. [1], who use sheaves to establish normalization for simply-typed λ-calculus with an η rule for coproducts, and from Srinivas [46]'s sheaf-theoretic formulation of pattern matching. Our mergable symbolic sets are similar in spirit to these uses of sheaf theory for modelling branching behavior, but tailored to the setting of symbolic execution.

10 Conclusion

Presheaves capture the way that probabilistic symbolic execution manipulates an ambient probability space (§3), and sheaves capture type-directed state merging (§5). Using presheaves and sheaves, we introduced two new semantic domains, SymSet and $\mathrm{SymSet}_{\mathrm{m}}$, for probabilistic symbolic execution. The rich structure of presheaves and sheaves, along with their close connection to categorical techniques like gluing, gives rise to a rich symbolic metalanguage that can be leveraged to quickly build symbolic semantics (§4), justify optimizations (§4.3), and prove correctness (§6). As a demonstration, we developed and proved correct the first symbolic semantics for a higher-order probabilistic language with type-directed state merging, pattern matching, and structural recursion. We hope our framework can unify existing strands of work on probabilistic symbolic semantics and act as a reusable language-invariant framework for giving semantics and proving their correctness.

11 Acknowledgements

We thank Cameron Moy, Joshua Gancher and the reviewers for their careful feedback. This work was supported by the National Science Foundation under Grants #CCF-2220408 and #CCF-2442685.

Data Availability Statement

The artifact for this paper can be found at https://doi.org/10.5281/zenodo.15049041 [30].

References

[1] Thorsten Altenkirch, Peter Dybjer, Martin Hofmann, and Philip Scott. 2001. Normalization by Evaluation for Typed Lambda Calculus with Coproducts. In *Logic in Computer Science (LICS)*. doi:10.1109/LICS.2001.932506

[2] Andrei Arusoaie. 2014. *A Generic Framework for Symbolic Execution: Theory and Applications*. Ph. D. Dissertation. Alexandru Ioan Cuza, University of Iasi.

[3] Andrei Arusoaie, Dorel Lucanu, and Vlad Rusu. 2013. A Generic Framework for Symbolic Execution. In *International Conference on Software Language Engineering*. Springer. doi:10.1016/j.jsc.2016.07.012

[4] Nick Benton. 2006. Abstracting Allocation: The New New Thing. In *International Workshop on Computer Science Logic (CSL)*. Springer.

[5] Nick Benton and Nicolas Tabareau. 2009. Compiling Functional Types to Relational Specifications for Low Level Imperative Code. In *International Workshop on Types in Language Design and Implementation*. doi:10.1145/1481861.1481864

[6] Lars Birkedal, Rasmus Ejlers Møgelberg, Jan Schwinghammer, and Kristian Støvring. 2012. First Steps in Synthetic Guarded Domain Theory: Step-Indexing in the Topos of Trees. *Logical Methods in Computer Science* (2012). doi:10.2168/LMCS-8(4:1)2012

[7] Francis Borceux. 1994. *Handbook of Categorical Algebra: Volume 3, Sheaf Theory*. Vol. 3. Cambridge University Press.

[8] Maddy Bowers, Alexander K. Lew, Joshua B. Tenenbaum, Armando Solar-Lezama, and Vikash K. Mansinghka. 2025. Stochastic Lazy Knowledge Compilation for Inference in Discrete Probabilistic Programs. *Programming Language Design and Implementation (PLDI)* (2025). doi:10.1145/3729325

[9] William X Cao, Poorva Garg, Ryan Tjoa, Steven Holtzen, Todd Millstein, and Guy Van den Broeck. 2023. Scaling Integer Arithmetic in Probabilistic Programs. In *Uncertainty in Artificial Intelligence*. PMLR.

[10] Stephen Chang, Alex Knauth, and Emina Torlak. 2017. Symbolic Types for Lenient Symbolic Execution. *Principles of Programming Languages (POPL)* (2017). doi:10.1145/3158128

[11] Mark Chavira and Adnan Darwiche. 2008. On Probabilistic Inference by Weighted Model Counting. *Artificial Intelligence* (2008). doi:10.1016/j.artint.2007.11.002

[12] Arthur Choi, Doga Kisa, and Adnan Darwiche. 2013. Compiling Probabilistic Graphical Models Using Sentential Decision Diagrams. In *European Conference on Symbolic and Quantitative Approaches to Reasoning and Uncertainty*. Springer. doi:10.1007/978-3-642-39091-3_11

[13] Thierry Coquand. 2013. Presheaf Model of Type Theory. *Unpublished note* (2013). https://www.cse.chalmers.se/~coquand/presheaf.pdf

[14] Frank S. de Boer and Marcello Bonsangue. 2019. On the Nature of Symbolic Execution. In *Formal Methods - The Next 30 Years* (Porto, Portugal). Springer-Verlag, Berlin, Heidelberg. doi:10.1007/978-3-030-30942-8_6

[15] Frank S de Boer and Marcello Bonsangue. 2021. Symbolic Execution Formally Explained. *Formal Aspects of Computing* (2021). doi:10.1007/s00165-020-00527-y

[16] Daan Fierens, Guy Van den Broeck, Joris Renkens, Dimitar Shterionov, Bernd Gutmann, Ingo Thon, Gerda Janssens, and Luc De Raedt. 2015. Inference and Learning in Probabilistic Logic Programs Using Weighted Boolean Formulas. *Theory and Practice of Logic Programming* (2015). doi:10.1017/S1471068414000076

[17] Antonio Filieri, Corina S. Păsăreanu, and Willem Visser. 2013. Reliability Analysis in Symbolic PathFinder. In *International Conference on Software Engineering (ICSE)*. doi:10.1109/ICSE.2013.6606608

[18] José Fragoso Santos, Petar Maksimović, Sacha-Élie Ayoun, and Philippa Gardner. 2020. Gillian, Part I: A Multi-Language Platform for Symbolic Execution. In *Programming Language Design and Implementation (PLDI)*. doi:10.1145/3385412.3386014

[19] Daniel Gratzer, Michael Shulman, and Jonathan Sterling. 2025. Strict Universes for Grothendieck Topoi. *Theory and Applications of Categories* (2025). doi:10.17863/CAM.123919

[20] Chris Heunen, Ohad Kammar, Sam Staton, and Hongseok Yang. 2017. A Convenient Category for Higher-Order Probability Theory. In *Logic in Computer Science (LICS)*. IEEE. doi:10.1109/lics.2017.8005137

[21] Martin Hofmann. 1999. Semantical Analysis of Higher-Order Abstract Syntax. In *Logic in Computer Science (LICS)*. IEEE. doi:10.1109/LICS.1999.782616

[22] Martin Hofmann and Thomas Streicher. 1997. Lifting Grothendieck Universes. *Unpublished note* (1997). https://www2.mathematik.tu-darmstadt.de/~streicher/NOTES/lift.pdf

[23] Steven Holtzen, Guy Van den Broeck, and Todd Millstein. 2020. Scaling Exact Inference for Discrete Probabilistic Programs. *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (2020). doi:10.1145/3428208

[24] Matthew Jackson. 2006. *A Sheaf Theoretic Approach to Measure Theory*. Ph. D. Dissertation. University of Pittsburgh.

[25] Peter Johnstone. 2002. *Sketches of an Elephant: A Topos Theory Compendium*. Oxford University Press.

[26] Alex Kavvos. 2024. Two-Dimensional Kripke Semantics I: Presheaves. In *Formal Structures for Computation and Deduction (FSCD)*. doi:10.4230/LIPIcs.FSCD.2024.14

[27] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* (1976). doi:10.1145/360248.360252

[28] Joachim Lambek and Philip J Scott. 1988. *Introduction to Higher-Order Categorical Logic*. Vol. 7. Cambridge University Press.

[29] John M Li, Jon Aytac, Philip Johnson-Freyd, Amal Ahmed, and Steven Holtzen. 2024. A Nominal Approach to Probabilistic Separation Logic. In *Logic in Computer Science (LICS)*. doi:10.1145/3661814.3662135

[30] John M. Li, Jack Czenszak, and Steven Holtzen. 2026. *Artifact for "Categorical Semantics of Probabilistic Symbolic Execution"*. doi:10.5281/zenodo.19485098

[31] Saunders MacLane and Ieke Moerdijk. 2012. *Sheaves in Geometry and Logic: A First Introduction to Topos Theory*. Springer Science & Business Media. doi:10.1007/978-1-4612-0927-0

[32] Eugenio Moggi. 1988. *Computational Lambda-Calculus and Monads*. University of Edinburgh, Department of Computer Science, Laboratory for Foundations of Computer Science. doi:10.1109/LICS.1989.39155

[33] Cameron Moy, Jack Czenszak, John M. Li, Brianna Marshall, and Steven Holtzen. 2025. Roulette: A Language for Expressive, Exact, and Efficient Discrete Probabilistic Programming. *Programming Language Design and Implementation (PLDI)* (2025). doi:10.1145/3729334

[34] Frank Joseph Oles. 1983. *A Category-Theoretic Approach to the Semantics of Programming Languages*. Ph.D. Dissertation. Syracuse University.

[35] Frank Joseph Oles. 1985. Type Algebras, Functor Categories, and Block Structure. *Algebraic Methods in Semantics* (1985).

[36] Umut Oztok, Arthur Choi, and Adnan Darwiche. 2016. Solving $\mathrm{PP}^{\mathrm{PP}}$-Complete Problems Using Knowledge Compilation. In *Principles of Knowledge Representation and Reasoning (KR)*. doi:10.1613/jair.1.11201

[37] Daniel Patterson, Andrew Wagner, and Amal Ahmed. 2023. Semantic Encapsulation Using Linking Types. In *International Workshop on Type-Driven Development*. doi:10.1145/3609027.3609405

[38] Andrew M Pitts. 2013. *Nominal Sets: Names and Symmetry in Computer Science*. Cambridge University Press. doi:10.5555/2512979

[39] Sorawee Porncharoenwase, Luke Nelson, Xi Wang, and Emina Torlak. 2022. A Formal Foundation for Symbolic Evaluation with Merging. *Principles of Programming Languages (POPL)* (2022). doi:10.1145/3498709

[40] Emily Riehl. 2017. *Category Theory in Context*. Courier Dover Publications.

[41] Feras A Saad, Martin C Rinard, and Vikash K Mansinghka. 2021. SPPL: Probabilistic Programming with Fast Exact Symbolic Inference. In *Programming Language Design and Implementation (PLDI)*. doi:10.1145/3453483.3454078

[42] Tian Sang, Paul Beame, and Henry A Kautz. 2005. Performing Bayesian Inference by Weighted Model Counting. In *Association for the Advancement of Artificial Intelligence (AAAI)*.

[43] Dana Scott. 1969. Boolean Models and Nonstandard Analysis. *Applications of Model Theory to Algebra, Analysis, and Probability* (1969).

[44] Koushik Sen, George Necula, Liang Gong, and Wontae Choi. 2015. Multise: Multi-Path Symbolic Execution Using Value Summaries. In *Foundations of Software Engineering*. doi:10.1145/2786805.2786830

[45] Alex Simpson. 2017. Probability Sheaves and the Giry Monad. In *Conference on Algebra and Coalgebra in Computer Science (CALCO 2017)*. doi:10.4230/LIPIcs.CALCO.2017.1

[46] Yellamraju V Srinivas. 1993. A Sheaf-Theoretic Approach to Pattern Matching and Related Problems. *Theoretical Computer Science* (1993). doi:10.1016/0304-3975(93)90239-P

[47] Ian Stark. 1996. Categorical Models for Local Names. *Lisp and Symbolic Computation* (1996). doi:10.1007/BF01806033

[48] Dario Stein. 2025. Random Variables, Conditional Independence and Categories of Abstract Sample Spaces. In *Logic in Computer Science (LICS)*. doi:10.1109/LICS65433.2025.00042

[49] Dario Stein and Sam Staton. 2021. Compositional Semantics for Probabilistic Programs with Exact Conditioning. In *Logic in Computer Science (LICS)*. doi:10.1109/LICS52264.2021.9470552

[50] Dario Stein and Sam Staton. 2024. Probabilistic Programming with Exact Conditions. *The Association for Computing Machinery (ACM)* (2024). doi:10.1145/3632170

[51] Zachary Susag, Sumit Lahiri, Justin Hsu, and Subhajit Roy. 2022. Symbolic Execution for Randomized Programs. *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (2022). doi:10.1145/3563344

[52] Paul Taylor. 1999. *Practical Foundations of Mathematics*. Cambridge University Press.

[53] Emina Torlak. [n. d.]. The Rosette Guide. https://docs.racket-lang.org/rosette-guide/index.html. Online; accessed Nov 13, 2025.

[54] Emina Torlak and Rastislav Bodik. 2013. Growing Solver-Aided Languages with Rosette. In *Onward! Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. doi:10.1145/2509578.2509586

[55] Emina Torlak and Rastislav Bodik. 2014. A Lightweight Symbolic Virtual Machine for Solver-Aided Host Languages. In *Programming Language Design and Implementation (PLDI)*. doi:10.1145/2594291.2594340

[56] Jaap Van Oosten. 2008. *Realizability: An Introduction to its Categorical Side*. Elsevier.

[57] Erik Voogd, Einar Broch Johnsen, Åsmund Aqissiaq Arild Kløvstad, Jurriaan Rot, and Alexandra Silva. 2025. Correct and Complete Symbolic Execution for Free. In *Integrated Formal Methods*, Nikolai Kosmatov and Laura Kovács (Eds.).

doi:10.1007/978-3-031-76554-4_13
[58] Erik Voogd, Einar Broch Johnsen, Alexandra Silva, Zachary J Susag, and Andrzej Wąsowski. 2023. Symbolic Semantics for Probabilistic Programs. In *International Conference on Quantitative Evaluation of Systems*. Springer. doi:10.1007/978-3-031-43835-6_23
[59] Erik Voogd, Åsmund Aqissiaq Arild Kløvstad, and Einar Broch Johnsen. 2023. Denotational Semantics for Symbolic Execution. In *International Conference on Theoretical Aspects of Computing (ICTAC)*. doi:10.1007/978-3-031-47963-2_22
[60] Andrew Wagner, Zachary Eisbach, and Amal Ahmed. 2024. Realistic Realizability: Specifying ABIs You Can Count On. *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (2024). doi:10.1145/3689755

A Haskell implementation of Imp

The full code listing for the Haskell implementation of Imp can be found below. Relative to the golfed implementation shown in Figure 5, the main difference is that the data type of Boolean formulas is implemented using the `decision-diagrams` library[4] so that it is efficient to check for unsatisfiability of a formula.

```
{-# LANGUAGE PatternSynonyms, BlockArguments, LambdaCase #-}
module Main where

import Prelude hiding (map, lookup, not, flip)
import Data.Map (Map, empty, singleton, insert, lookup, map, unionWith,
                 fromList, fromListWith, toList, foldr', foldrWithKey', insertWith)
import Data.Maybe (fromMaybe, isNothing)
import Control.Monad.State (State, state, runState)

import Data.DecisionDiagram.BDD (BDD, AscOrder)
import qualified Data.DecisionDiagram.BDD as BDD

--------------------------------------------------------------------------------
-- Definitions needed for SE

type SymVar = Int
newtype Formula = Formula { toBDD :: BDD BDD.AscOrder } deriving (Eq, Show)
type WeightMap = Map SymVar Rational

unsat :: Formula -> Bool
unsat = isNothing . BDD.anySat . toBDD

(/\) :: Formula -> Formula -> Formula
Formula p /\ Formula q = Formula (p BDD..&&. q)

(\/) :: Formula -> Formula -> Formula
Formula p \/ Formula q = Formula (p BDD..||. q)

symVar :: SymVar -> Formula
symVar = Formula . BDD.var

notB :: Formula -> Formula
notB (Formula p) = Formula (BDD.notB p)

iteB :: Formula -> Formula -> Formula -> Formula
iteB (Formula p) (Formula q) (Formula r) = Formula (BDD.ite p q r)

top :: Formula
top = Formula BDD.true
```

[4] https://hackage.haskell.org/package/decision-diagrams-0.2.0.0

```
bot :: Formula
bot = Formula BDD.false

--------------------------------------------------------------------------------
-- Signatures for each language construct

true :: Expr
false :: Expr
var :: Var -> Expr
not :: Expr -> Expr
(.&&) :: Expr -> Expr -> Expr
(.||) :: Expr -> Expr -> Expr

skip :: Command
set :: Var -> Expr -> Command
flip :: Var -> Rational -> Command
(>>>) :: Command -> Command -> Command
ite :: Expr -> Command -> Command -> Command

--------------------------------------------------------------------------------
-- Interpreting types

type Var = String
type Memory = Map Var Formula
type Expr = (Formula, Memory) -> Formula
type Command = (Formula, Memory) -> (State (SymVar, WeightMap)) [(Formula, Memory)]

--------------------------------------------------------------------------------
-- Interpreting expressions

true _ = top
false _ = bot
(var x) (p, m) = find x m where find x = fromMaybe top . lookup x
(not e) (p, m) = notB (e (p, m))
(.&&) = liftA2 (/\)
(.||) = liftA2 (\/)

--------------------------------------------------------------------------------
-- Interpreting commands

skip (p, m) = pure [(p, m)]
(set x e) (p, m) = pure [(p, insert x (e (p, m)) m)]
(c1 >>> c2) (p, m) = concat <$> (mapM c2 =<< c1 (p, m))
(flip x b) (p, m) = state \(i, w) -> ([(p, insert x (symVar i) m)], (succ i, insert i b w))
(ite e c1 c2) (p, m) = (++) <$> try c1 (p /\ q, m) <*> try c2 (p /\ notB q, m)
  where { q = e (p, m); try c (p, m) = if unsat p then pure [] else c (p, m) }

--------------------------------------------------------------------------------

run :: Command -> ([(Formula, Memory)], (SymVar, WeightMap))
run c = c (top, empty) `runState` (0, empty)

printLongLine :: IO ()
printLongLine = putStrLn (replicate 80 '-')

main :: IO ()
main = do
```

```
let noisyOr = flip "x" (1/2) >>> flip "y" (1/3) >>> ite (var "x" .|| var "y") (set "z" true) (set "z" false)
print $ run noisyOr
pure ()
```

B Imp with state merging

B.1 Symbolic semantics

The semantics of expressions is identical to the one given in Figure 4. The semantics of commands is shown below.

$$
\begin{aligned}
&\mathcal{S}_{\mathrm{m}}[\![\texttt{skip}]\!]_{(\Gamma,\varphi)}(\hat{s}) = \nu\,\varnothing.\,\hat{s}\\
&\mathcal{S}_{\mathrm{m}}[\![x \coloneqq e]\!]_{(\Gamma,\varphi)}(\hat{s}) = \nu\,\varnothing.\,\hat{s}[x \mapsto \mathcal{S}_{\mathrm{m}}[\![e]\!]_{(\Gamma,\varphi)}(\hat{s})]\\
&\mathcal{S}_{\mathrm{m}}[\![x \sim \texttt{flip}\,p]\!]_{(\Gamma,\varphi)}(\hat{s}) = \nu\,(\alpha \colon \mathrm{Ber}\,p).\,\hat{s}[x \mapsto [\alpha]]\\
&\mathcal{S}_{\mathrm{m}}[\![c_1\texttt{;}\ c_2]\!]_{(\Gamma,\varphi)}(\hat{s}) = \nu\,(\Gamma_1 + \Gamma_2).\,\hat{s}_2 \text{ where}\\
&\quad \mathcal{S}_{\mathrm{m}}[\![c_1]\!]_{(\Gamma,\varphi)}(\hat{s}) = \nu\,\Gamma_1.\,\hat{s}_1\\
&\quad \mathcal{S}_{\mathrm{m}}[\![c_2]\!]_{(\Gamma+\Gamma_1,\varphi)}(\hat{s}_1) = \nu\,\Gamma_2.\,\hat{s}_2\\
&\mathcal{S}_{\mathrm{m}}[\![\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2]\!]_{(\Gamma,\varphi)}(\hat{s}) = \nu\,(\Gamma_1 + \Gamma_2).\,(\hat{s}_1 \sqcup_{\varphi\wedge\psi,\varphi\wedge\neg\psi} \hat{s}_2) \text{ where}\\
&\quad \mathcal{S}_{\mathrm{m}}[\![e]\!]_{(\Gamma,\varphi)}(\hat{s}) = [\psi]\\
&\quad \mathcal{S}_{\mathrm{m}}[\![c_1]\!]_{(\Gamma,\varphi\wedge\psi)}(\hat{s}) = \nu\,\Gamma_1.\,\hat{s}_1\\
&\quad \mathcal{S}_{\mathrm{m}}[\![c_2]\!]_{(\Gamma,\varphi\wedge\neg\psi)}(\hat{s}) = \nu\,\Gamma_2.\,\hat{s}_2
\end{aligned}
$$

The merging operation on stores is defined by:

$$\{x \mapsto [\psi], \dots\} \sqcup_{\varphi,\varphi'} \{x \mapsto [\psi'], \dots\} = \{x \mapsto [(\psi \wedge \varphi) \vee (\psi' \wedge \varphi')], \dots\}$$

C Proof of Proposition 5.4

It suffices to show that the unit map η of Piecewise satisfies the following universal property: if $f : \widehat{A} \to \widehat{B}$ is a symbolic function from a symbolic set $\widehat{A}$ to a mergable symbolic set $\widehat{B}$, then there exists a unique symbolic function $\overline{f} : \mathrm{Piecewise}\,\widehat{A} \to \widehat{B}$ such that $\overline{f} \circ \eta = f$.

The requisite symbolic function $\overline{f}$ can be defined by $\overline{f}_{(\Gamma,\varphi)}[\varphi_i : \hat{a}_i]_{i\in I} = \bigsqcup_{i\in I} f_{(\Gamma,\varphi)}(\hat{a}_i)$. This function is well-defined: it respects the quotient used to define Piecewise by induction on I, using the uniqueness of amalgamations computed by $\varnothing, \sqcup$ in $\widehat{B}$, and is natural because f is natural and $\sqcup$ commutes with restriction and renaming.

The equation $\overline{f} \circ \eta = f$ holds by the following calculation: $\overline{f}_{(\Gamma,\varphi)}(\eta_{(\Gamma,\varphi)}(\hat{a})) = \overline{f}_{(\Gamma,\varphi)}[\varphi : \hat{a}] = f_{(\Gamma,\varphi)}(\hat{a})$. To show uniqueness of $\overline{f}$, let $\widetilde{f}$ be an arbitrary symbolic function satisfying $\overline{f} \circ \eta = f$. Let $[\varphi_i : \hat{a}_i]_{i\in I}$ be an arbitrary element of $\mathrm{Piecewise}\,\widehat{A}_{(\Gamma,\varphi)}$. Then for all $i \in I$ it holds that $\left(\widetilde{f}_{(\Gamma,\varphi)}[\varphi_i : \hat{a}_i]_{i\in I}\right)|_{\varphi_i} = \widetilde{f}_{(\Gamma,\varphi_i)}\left([\varphi_i : \hat{a}_i]_{i\in I}\Big|_{\varphi_i}\right) = \widetilde{f}_{(\Gamma,\varphi_i)}[\varphi_i : \hat{a}_i] = \widetilde{f}_{(\Gamma,\varphi_i)}(\eta_{(\Gamma,\varphi_i)}(\hat{a}_i)) = \hat{a}_i$. This implies that $\widetilde{f}_{(\Gamma,\varphi_i)}[\varphi_i : \hat{a}_i]_{i\in I}$ computes the unique amalgamation in $\widehat{B}$ of the family $(\hat{a}_i)_{i\in I}$, which is equal to $\overline{f}_{(\Gamma,\varphi_i)}[\varphi_i : \hat{a}_i]_{i\in I}$ by definition.

D Proof of Theorem 3.1

We construct two distributive laws

(1) $\delta_1 : \mathrm{Gen} \circ \mathrm{Piecewise} \to \mathrm{Piecewise} \circ \mathrm{Gen}$
(2) $\delta_2 : \mathrm{Piecewise} \circ \mathrm{Gen} \to \mathrm{Gen} \circ \mathrm{Piecewise}$

from which it follows that Piecewise ∘ Gen and Gen ∘ Piecewise are both monads.

D.1 The distributive law δ_1

The distributive law δ_1 is defined as follows:

$$\delta_{1,A,(\Gamma,\varphi)} : (\mathsf{Gen}(\mathsf{Piecewise}\,\widehat{A}))_{(\Gamma,\varphi)} \to (\mathsf{Piecewise}(\mathsf{Gen}\,\widehat{A}))_{(\Gamma,\varphi)}$$
$$\delta_{1,A,(\Gamma,\varphi)}(\nu\,\Delta.\,[\varphi_i : \widehat{a}_i]_{i\in I}) = [\varphi_i : \nu\,\Delta.\,\widehat{a}_i]_{i\in I}$$

Next we show the axioms of a distributive law hold. Letting $G = \mathsf{Gen}$ and $P = \mathsf{Piecewise}$, we have:

- The canonical maps $G \to GP \to PG$ and $G \to PG$ are equal: tracing the path of a canonical element $\nu\,\Delta.\,\hat{a}$ of $(G\widehat{A})_{(\Gamma,\varphi)}$ through them gives
$$\nu\,\Delta.\,\hat{a} \mapsto \nu\,\Delta.\,[\varphi : \hat{a}] \mapsto [\varphi : \nu\,\Delta.\,\hat{a}] \text{ and}$$
$$\nu\,\Delta.\,\hat{a} \mapsto [\varphi : \nu\,\Delta.\,\hat{a}] \text{ respectively.}$$
- The canonical maps $P \to GP \to PG$ and $P \to PG$ are equal: given a canonical element $[\varphi_i : \hat{a}_i]_{i\in I}$ of $(P\widehat{A})_{(\Gamma,\varphi)}$, we have the two routes
$$[\varphi_i : \hat{a}_i]_{i\in I} \mapsto \nu\,\varnothing.\,[\varphi_i : \hat{a}_i]_{i\in I} \mapsto [\varphi_i : \nu\,\varnothing.\,\hat{a}_i] \text{ and}$$
$$[\varphi_i : \hat{a}_i]_{i\in I} \mapsto [\varphi_i : \nu\,\varnothing.\,\hat{a}_i].$$
- The canonical maps $GPP \to PGP \to PPG \to PG$ and $GPP \to GP \to PG$ are equal: given $\nu\,\Delta.\,[\varphi_i : [\varphi_{ij} : \hat{a}_{ij}]_{j\in J}]_{i\in I}$ in $(GPP\widehat{A})_{(\Gamma,\varphi)}$, we have
$$\begin{aligned}&\nu\,\Delta.\,[\varphi_i : [\varphi_{ij} : \hat{a}_{ij}]_{j\in J}]_{i\in I}\\ &\mapsto [\varphi_i : \nu\,\Delta.\,[\varphi_{ij} : \hat{a}_{ij}]_{j\in J}]_{i\in I}\\ &\mapsto [\varphi_i : [\varphi_{ij} : \nu\,\Delta.\,\hat{a}_{ij}]_{j\in J}]_{i\in I}\\ &\mapsto [\varphi_{ij} : \nu\,\Delta.\,\hat{a}_{ij}]_{i\in I, j\in J}\end{aligned}$$
and $\nu\,\Delta.\,[\varphi_i : [\varphi_{ij} : \hat{a}_{ij}]_{j\in J}]_{i\in I} \mapsto \nu\,\Delta.\,[\varphi_{ij} : \hat{a}_{ij}]_{i\in I, j\in J} \mapsto [\varphi_{ij} : \nu\,\Delta.\,\hat{a}_{ij}]_{i\in I, j\in J}$.
- The canonical maps $GGP \to GPG \to PGG \to PG$ and $GGP \to GP \to PG$ are equal: given $\nu\,\Delta_1.\,\nu\,\Delta_2.\,[\varphi_i : \hat{a}_i]_{i\in I}$ in $(GGP\widehat{A})_{(\Gamma,\varphi)}$, we have
$$\begin{aligned}&\nu\,\Delta_1.\,\nu\,\Delta_2.\,[\varphi_i : \hat{a}_i]_{i\in I}\\ &\mapsto \nu\,\Delta_1.\,[\varphi_i : \nu\,\Delta_2.\,\hat{a}_i]_{i\in I}\\ &\mapsto [\varphi_i : \nu\,\Delta_1.\,\nu\,\Delta_2.\,\hat{a}_i]_{i\in I}\\ &\mapsto [\varphi_i : \nu\,(\Delta_1 + \Delta_2).\,\hat{a}_i]_{i\in I}\end{aligned}$$
and $\nu\,\Delta_1.\,\nu\,\Delta_2.\,[\varphi_i : \hat{a}_i]_{i\in I} \mapsto \nu\,(\Delta_1 + \Delta_2).\,[\varphi_i : \hat{a}_i]_{i\in I} \mapsto [\varphi_i : \nu\,(\Delta_1 + \Delta_2).\,\hat{a}_i]_{i\in I}$.

D.2 The distributive law δ_2

The distributive law δ_2 is defined as follows:

$$\delta_{2,A,(\Gamma,\varphi)} : (\mathsf{Piecewise}(\mathsf{Gen}\,\widehat{A}))_{(\Gamma,\varphi)} \to (\mathsf{Gen}(\mathsf{Piecewise}\,\widehat{A}))_{(\Gamma,\varphi)}$$
$$\delta_{2,A,(\Gamma,\varphi)}[\varphi_i : \nu\,\Delta_i.\,\hat{a}_i]_{i\in I} = \nu\,(\textstyle\sum_{i\in I}\Delta_i).\,[\varphi_i : \hat{a}_i]_{i\in I}$$

Note the implicit weakening of each φ_i and $\hat{a}_i$ from context $\Gamma + \Gamma_i$ to $\Gamma + \sum_{i\in I}\Gamma_i$. Well-definedness of δ_2 depends crucially on the quotient used to define Gen: the definition just given respects the rule $\{\varphi_1 + \varphi_2 : \hat{a}\} \sim \{\varphi_1 : \hat{a}|_{\varphi_1}, \varphi_2 : \hat{a}|_{\varphi_2}\}$ of the quotient used to define Piecewise because elements of $\mathsf{Gen}\,\widehat{A}$ are quotiented by piecewise agreement. In particular, given any two equal piecewise definitions $[\varphi_i : \nu\,\Delta_i.\,\hat{a}_i]_{i\in I} = [\varphi'_{i'} : \nu\,\Delta_{i'}.\,\hat{a}_{i'}]_{i'\in I'}$, there exists a common refinement $(\psi_j)_{j\in J}$ of the

two partitions $(\varphi_i)_{i\in I}$ and $(\varphi'_{i'})_{i'\in I'}$ given by taking pairwise intersections, and the results produced by δ_2 agree piecewise on this common refinement.

We now establish the distributive law axioms. Letting $G = \mathsf{Gen}$ and $P = \mathsf{Piecewise}$,

- The maps $P \to PG \to GP$ and $P \to GP$ are equal: given $[\varphi_i : \hat{a}_i]_{i\in I}$ in $(\widehat{PA})_{(\Gamma,\varphi)}$, we have
$$[\varphi_i : \hat{a}_i]_{i\in I} \mapsto [\varphi_i : \nu\,\varnothing.\,\hat{a}_i]_{i\in I} \mapsto \nu\,\varnothing.\,[\varphi_i : \hat{a}_i] \text{ and}$$
$$[\varphi_i : \hat{a}_i]_{i\in I} \mapsto \nu\,\varnothing.\,[\varphi_i : \hat{a}_i]_{i\in I}.$$
- The maps $G \to PG \to GP$ and $G \to GP$ are equal: given $\nu\,\Delta.\,\hat{a}$ in $(\widehat{GA})_{(\Gamma,\varphi)}$, we have
$$\nu\,\Delta.\,\hat{a} \mapsto [\varphi : \nu\,\Delta.\,\hat{a}] \mapsto \nu\,\Delta.\,[\varphi : \hat{a}] \text{ and}$$
$$\nu\,\Delta.\,\hat{a} \mapsto \nu\,\Delta.\,[\varphi : \hat{a}].$$
- The maps $PGG \to GPG \to GGP \to GP$ and $PGG \to PG \to GP$ are equal: given an element $[\varphi_i : \nu\,\Delta_{i1}.\,\nu\,\Delta_{i2}.\,\hat{a}_i]_{i\in I}$ of $(\widehat{PGGA})_{(\Gamma,\varphi)}$, we have
$$\begin{aligned}&[\varphi_i : \nu\,\Delta_{i1}.\,\nu\,\Delta_{i2}.\,\hat{a}_i]_{i\in I}\\ &\mapsto \nu\,(\textstyle\sum_{i\in I}\Delta_{i1}).\,[\varphi_i : \nu\,\Delta_{i2}.\,\hat{a}_i]_{i\in I}\\ &\mapsto \nu\,(\textstyle\sum_{i\in I}\Delta_{i1}).\,\nu\,(\sum_{i\in I}\Delta_{i2}).\,[\varphi_i :\ \hat{a}_i]_{i\in I}\\ &\mapsto \nu\,(\textstyle\sum_{i\in I}\Delta_{i1} + \sum_{i\in I}\Delta_{i2}).\,[\varphi_i :\ \hat{a}_i]_{i\in I}\end{aligned}$$
and
$$[\varphi_i : \nu\,\Delta_{i1}.\,\nu\,\Delta_{i2}.\,\hat{a}_i]_{i\in I} \mapsto [\varphi_i : \nu\,(\Delta_{i1}+\Delta_{i2}).\,\hat{a}_i]_{i\in I} \mapsto \nu\,(\textstyle\sum_{i\in I}(\Delta_{i1}+\Delta_{i2})).\,[\varphi_i :\ \hat{a}_i]_{i\in I},$$
which produce equal outputs via the isomorphism $\sum_{i\in I}\Delta_{i1} + \sum_{i\in I}\Delta_{i2} \cong \sum_{i\in I}(\Delta_{i1}+\Delta_{i2})$.
- The maps $PPG \to PGP \to GPP \to PG$ and $PPG \to PG \to GP$ are equal: given an element $[\varphi_i : [\varphi_{ij} : \nu\,\Delta_{ij}.\,\hat{a}_{ij}]_{j\in J}]_{i\in I}$ of $(\widehat{PPGA})_{(\Gamma,\varphi)}$, we have
$$\begin{aligned}&[\varphi_i : [\varphi_{ij} : \nu\,\Delta_{ij}.\,\hat{a}_{ij}]_{j\in J}]_{i\in I}\\ &\mapsto [\varphi_i : \nu\,(\textstyle\sum_{j\in J}\Delta_{ij}).\,[\varphi_{ij} : \hat{a}_{ij}]_{j\in J}]_{i\in I}\\ &\mapsto \nu\,(\textstyle\sum_{i\in I}\sum_{j\in J}\Delta_{ij}).\,[\varphi_i : [\varphi_{ij} : \hat{a}_{ij}]_{j\in J}]_{i\in I}\\ &\mapsto \nu\,(\textstyle\sum_{i\in I}\sum_{j\in J}\Delta_{ij}).\,[\varphi_{ij} : \hat{a}_{ij}]_{i\in I, j\in J}\end{aligned}$$
and
$$[\varphi_i : [\varphi_{ij} : \nu\,\Delta_{ij}.\,\hat{a}_{ij}]_{j\in J}]_{i\in I} \mapsto [\varphi_{ij} : \nu\,\Delta_{ij}.\,\hat{a}_{ij}]_{i\in I,j\in J} \mapsto \nu\,(\textstyle\sum_{i\in I,j\in J}\Delta_{ij}).\,[\varphi_{ij} : \hat{a}_{ij}]_{i\in I,j\in J}$$
which produce equal outputs via the isomorphism $\sum_{i\in I}\sum_{j\in J}\Delta_{ij} \cong \sum_{i\in I,j\in J}\Delta_{ij}$.

E Proof of Theorem 6.1

The lifting of exponentials, limits, and colimits follow from the general theory of gluing; lifting of limits follows from the key fact that the functor span used to construct Inst preserves limits. The remainder of the proof is devoted to relating the monads Piecewise and Gen to their concrete counterparts $\mathsf{id}_{\mathsf{Set}}$ and Dist. The proof has three parts.

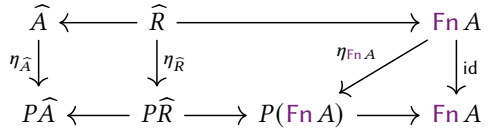
(1) Section E.1 lifts (Piecewise, $\mathsf{id}_{\mathsf{Set}}$) to a monad $\overline{\mathsf{P}}$iecewise on Inst.
(2) Section E.2 lifts (Gen, Dist) to a monad $\overline{\mathsf{G}}$en on Inst.
(3) Section E.3 lifts (gen p, Ber p) to a morphism $\overline{\mathsf{g}}\mathsf{en}\ p$ for all $p \in [0,1]$.
(4) Section E.4 lifts the distributive laws δ_1, δ_2 from Section D to corresponding distributive laws $\overline{\delta}_1, \overline{\delta}_2$, ensuring that all composites of (Piecewise, $\mathsf{id}_{\mathsf{Set}}$) and (Gen, Dist) lift to composites of $\overline{\mathsf{P}}$iecewise and $\overline{\mathsf{G}}$en.

E.1 Lifting (Piecewise, $\mathsf{id}_{\mathsf{Set}}$)

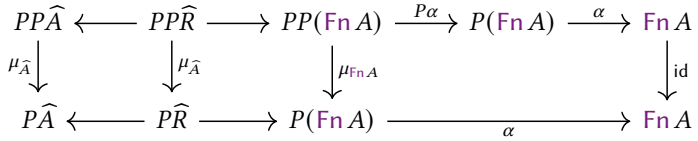
The monad $\overline{\mathsf{P}}$iecewise sends each instantiation span $(\widehat{A} \xleftarrow{i} \widehat{R} \xrightarrow{j} \mathsf{Fn}\, A)$ to the span

$$\mathsf{Piecewise}\,\widehat{A} \xleftarrow{\mathsf{Piecewise}(i)} \mathsf{Piecewise}\,\widehat{R} \xrightarrow{\mathsf{Piecewise}(j)} \mathsf{Piecewise}(\mathsf{Fn}\,A) \xrightarrow{\alpha} \mathsf{Fn}\,A,$$

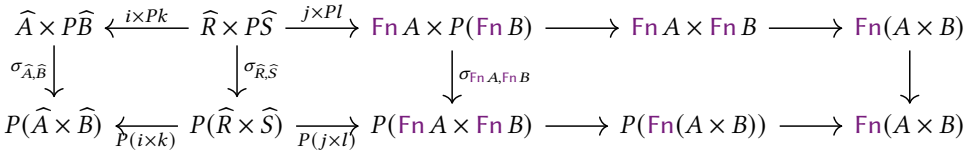
where α is the algebra map witnessing mergability of $\mathsf{Fn}\,A$. (Concretely, α sends each piecewise definition $[\varphi_i : f_i]_{i\in I}$ of a function to $\bigcup_{i\in I} f_i$, the actual function that it denotes.) The monad structure on $\overline{\mathsf{P}}$iecewise is inherited from the monad structure on Piecewise. Letting $P = \mathsf{Piecewise}$, the unit is given by the commutative diagram



where all the squares commute by naturality and the right-most triangle commutes because α is an algebra map. The multiplication is given by



where all the squares commute by naturality and the right-most rectangle commutes because α is an algebra map. Finally, given two instantiation spans $\widehat{A} \xleftarrow{i} \widehat{R} \xrightarrow{j} \mathsf{Fn}\,A$ and $\widehat{B} \xleftarrow{k} \widehat{S} \xrightarrow{l} \mathsf{Fn}\,B$, the strength is given by



where the squares commute by naturality and the rectangle commutes because, for any function $f : \mathsf{Mod}(\Gamma, \varphi) \to A$ and piecewise definition $[\varphi_i : g_i]_{i\in I}$ consisting of functions $(g_i : \mathsf{Mod}(\Gamma, \varphi_i) \to B)_{i\in I}$, the function given by first collating the g_is into a function $g = \bigcup_{i\in I} g_i$ and then pairing it with f to yield $\langle f, g\rangle$ is the same as the function given by collating the piecewise definition $[\varphi_i : \langle f|_{\mathsf{Mod}(\Gamma,\varphi_i)}, g_i\rangle]_{i\in I}$.

Because the unit, multiplication, and strength of $\overline{\mathsf{P}}$iecewise were defined using the corresponding components of Piecewise and $\mathsf{id}_{\mathsf{Set}}$, $\overline{\mathsf{P}}$iecewise is a lifting of (Piecewise, $\mathsf{id}_{\mathsf{Set}}$) by construction, and the monad laws follow from the monad laws for Piecewise and $\mathsf{id}_{\mathsf{Set}}$.

E.2 Lifting (Gen, Dist)

The monad $\overline{\mathsf{G}}$en is defined in terms of the following *concretization* map:

$$\gamma_{A,(\Gamma,\varphi)} : (\mathsf{Gen}(\mathsf{Fn}\,A))_{(\Gamma,\varphi)} \to (\mathsf{Fn}(\mathsf{Dist}\,A))_{(\Gamma,\varphi)}$$
$$\gamma_{A,(\Gamma,\varphi)}(v\,(\alpha : \mathsf{Ber}\,p, \dots).\,f) = \lambda\, m.\,\big(v \leftarrow \mathsf{Ber}\,p;\ \dots;\ \mathsf{pure}(f(m[\alpha \mapsto v, \dots]))\big)$$

In words, γ takes in a path condition $\Gamma \vdash \varphi$, some fresh variables $\Delta = (\alpha : \mathsf{Ber}\,p, \dots)$, and a function $f : \mathsf{Mod}(\Gamma + \Delta, \varphi) \to A$, and produces the probabilistic computation $\mathsf{Mod}(\Gamma, \varphi) \to \mathsf{Dist}\,A$ that runs

f after sampling one random Boolean per entry in Δ. Our lifting $\overline{\mathsf{Gen}}$ of $(\mathsf{Gen}, \mathsf{Dist})$ makes use of the following lemma about concretization:

Lemma E.1. The pair (Fn, γ) forms a strong monad morphism $\mathsf{Gen} \to \mathsf{Dist}$ in the sense that the following diagrams commute for all A and B:

(1)
$$\begin{array}{ccc} \mathsf{Fn}\,A & & \\ {\scriptstyle\eta}\downarrow & \searrow^{\mathsf{Fn}\,\eta} & \\ \mathsf{Gen}(\mathsf{Fn}\,A) & \xrightarrow[\gamma]{} & \mathsf{Fn}(\mathsf{Dist}\,A) \end{array}$$

(2)
$$\begin{array}{ccccc} \mathsf{Gen}(\mathsf{Gen}(\mathsf{Fn}\,A)) & \xrightarrow{\mathsf{Gen}\,\gamma} & \mathsf{Gen}(\mathsf{Fn}(\mathsf{Dist}\,A)) & \xrightarrow{\gamma} & \mathsf{Fn}(\mathsf{Dist}(\mathsf{Dist}\,A)) \\ {\scriptstyle\mu}\downarrow & & & & \downarrow{\scriptstyle\mathsf{Fn}\,\mu} \\ \mathsf{Gen}(\mathsf{Fn}\,A) & & \xrightarrow[\gamma]{} & & \mathsf{Fn}(\mathsf{Dist}\,A) \end{array}$$

(3)
$$\begin{array}{ccccc} \mathsf{Fn}\,A \times \mathsf{Gen}(\mathsf{Fn}\,B) & \xrightarrow{\sigma} & \mathsf{Gen}(\mathsf{Fn}\,A \times \mathsf{Fn}\,B) & \longrightarrow & \mathsf{Gen}(\mathsf{Fn}(A \times B)) \\ {\scriptstyle\mathsf{Fn}\,A \times \gamma}\downarrow & & & & \downarrow{\scriptstyle\gamma} \\ \mathsf{Fn}\,A \times \mathsf{Fn}(\mathsf{Dist}\,B) & \longrightarrow & \mathsf{Fn}(A \times \mathsf{Dist}\,B) & \xrightarrow[\mathsf{Fn}\,\sigma]{} & \mathsf{Fn}(\mathsf{Dist}(A \times B)) \end{array}$$

Proof. We establish that each type of diagram commutes by chasing elements through them.

(1) For any f in $(\mathsf{Fn}\,A)_{(\Gamma,\varphi)}$, the two routes through the diagram are $f \mapsto \nu\,\varnothing.\, f \mapsto \eta \circ f$ and $f \mapsto \eta \circ f$, where $\eta : A \to \mathsf{Dist}\,A$ is the unit of Dist.

(2) For any $\nu\,(\alpha : \mathsf{Ber}\,p, \dots).\, \nu\,(\beta : \mathsf{Ber}\,q).\, f$ in $(\mathsf{Gen}(\mathsf{Gen}(\mathsf{Fn}\,A)))_{(\Gamma,\varphi)}$, the two routes through the diagram are

$$\begin{aligned} & \nu\,(\alpha : \mathsf{Ber}\,p, \dots).\, \nu\,(\beta : \mathsf{Ber}\,q).\, f \\ \overset{\mathsf{Gen}\,\gamma}{\mapsto}\ & \nu\,(\alpha : \mathsf{Ber}\,p, \dots).\, \lambda\, m.\, \big(v \leftarrow \mathsf{Ber}\,p;\ \dots;\ \mathsf{pure}(f(m[\alpha \mapsto v, \dots])\big) \\ \overset{\gamma}{\mapsto}\ & \lambda\, m.\, \big(w \leftarrow \mathsf{Ber}\,q;\ \dots;\ v \leftarrow \mathsf{Ber}\,p;\ \dots;\ \mathsf{pure}(\mathsf{pure}(f(m[\beta \mapsto w, \dots][\alpha \mapsto v, \dots])))\big) \\ \overset{\mathsf{Fn}\,\mu}{\mapsto}\ & \lambda\, m.\, \big(w \leftarrow \mathsf{Ber}\,q;\ \dots;\ v \leftarrow \mathsf{Ber}\,p;\ \dots;\ \mathsf{pure}(f(m[\beta \mapsto w, \dots][\alpha \mapsto v, \dots]))\big) \end{aligned}$$

and

$$\begin{aligned} & \nu\,(\alpha : \mathsf{Ber}\,p, \dots).\, \nu\,(\beta : \mathsf{Ber}\,q).\, f \\ \overset{\mu}{\mapsto}\ & \nu\,(\alpha : \mathsf{Ber}\,p, \dots, \beta : \mathsf{Ber}\,q).\, f \\ \overset{\gamma}{\mapsto}\ & \lambda\, m.\, \big(v \leftarrow \mathsf{Ber}\,p;\ \dots;\ w \leftarrow \mathsf{Ber}\,q;\ \dots;\ \mathsf{pure}(f(m[\alpha \mapsto v, \dots, \beta \mapsto w, \dots]))\big) \end{aligned}$$

which are equal because Dist is a commutative monad (so the sampling commands can be reordered).

(3) For any $\Delta = (\alpha : \mathsf{Ber}\,p, \dots)$ and $(f, \nu\,\Delta.\, g)$ in $(\mathsf{Fn}\,A \times \mathsf{Gen}(\mathsf{Fn}\,B))_{(\Gamma,\varphi)}$, the two routes through the diagram are

$$\begin{aligned} & (f, \nu\,(\alpha : \mathsf{Ber}\,p, \dots).\, g) \\ \overset{\sigma}{\mapsto}\ & \nu\,(\alpha : \mathsf{Ber}\,p, \dots).\, (f \circ \pi_\Gamma, g) \\ \mapsto\ & \nu\,(\alpha : \mathsf{Ber}\,p, \dots).\, \langle f \circ \pi_\Gamma, g \rangle \\ \overset{\gamma}{\mapsto}\ & \lambda\, m.\, \big(v \leftarrow \mathsf{Ber}\,p;\ \dots;\ \mathsf{pure}(f(m), g(m[\alpha \mapsto v, \dots]))\big), \end{aligned}$$

where $\pi_\Gamma : \mathsf{Mod}(\Gamma + \Delta, \varphi) \to \mathsf{Mod}(\Gamma, \varphi)$ is the canonical projection, and

$$\begin{aligned}
& (f, \nu\,(\alpha : \mathsf{Ber}\,p, \dots).\, g) \\
\overset{\mathsf{Fn}\,A\times\gamma}{\mapsto}\ & (f, \lambda\, m.\, (v \leftarrow \mathsf{Ber}\,p;\ \dots;\ \mathsf{pure}(g(m[\alpha \mapsto v, \dots])))) \\
\mapsto\ & \lambda\, m.\, (f(m), \lambda\, m.\, (v \leftarrow \mathsf{Ber}\,p;\ \dots;\ \mathsf{pure}(g(m[\alpha \mapsto v, \dots])))) \\
\mapsto\ & \lambda\, m.\, (v \leftarrow \mathsf{Ber}\,p;\ \dots;\ \mathsf{pure}(f(m), g(m[\alpha \mapsto v, \dots]))).
\end{aligned}$$

□

With Lemma E.1 in hand, the lifted monad $\overline{\mathsf{G}}\mathsf{en}$ is defined by sending each instantiation span $(\widehat{A} \xleftarrow{i} \widehat{R} \xrightarrow{j} \mathsf{Fn}\,A)$ to the span $\mathsf{Gen}\,\widehat{A} \xleftarrow{\mathsf{Gen}(i)} \mathsf{Gen}\,\widehat{R} \xrightarrow{\mathsf{Gen}(j)} \mathsf{Gen}(\mathsf{Fn}\,A) \xrightarrow{\gamma_A} \mathsf{Fn}(\mathsf{Dist}\,A)$. The monad structure on $\overline{\mathsf{G}}\mathsf{en}$ is defined analogously to Section E.1. Letting $G = \mathsf{Gen}$, the unit of $\overline{\mathsf{G}}\mathsf{en}$ is

$$\begin{array}{ccccccc}
\widehat{A} & \longleftarrow & \widehat{R} & & \longrightarrow & & \mathsf{Fn}\,A \\
{\scriptstyle\eta}\downarrow & & \downarrow{\scriptstyle\eta} & & & \overset{\eta}{\swarrow} & \downarrow{\scriptstyle\mathsf{Fn}\,\eta} \\
G\widehat{A} & \longleftarrow & G\widehat{R} & \longrightarrow & G(\mathsf{Fn}\,A) & \longrightarrow & \mathsf{Fn}(\mathsf{Dist}\,A)
\end{array}$$

where the right-most triangle commutes by point (1) of Lemma E.1. Letting $D = \mathsf{Dist}$, the multiplication of $\overline{\mathsf{G}}\mathsf{en}$ is

$$\begin{array}{ccccccccc}
GG\widehat{A} & \longleftarrow & GG\widehat{R} & \longrightarrow & GG(\mathsf{Fn}\,A) & \xrightarrow{G\gamma} & G(\mathsf{Fn}(DA)) & \xrightarrow{\gamma} & \mathsf{Fn}(DDA) \\
{\scriptstyle\mu}\downarrow & & \downarrow{\scriptstyle\mu} & & \downarrow{\scriptstyle\mu} & & & & \downarrow{\scriptstyle\mathsf{Fn}\,\mu} \\
G\widehat{A} & \longleftarrow & G\widehat{R} & \longrightarrow & G(\mathsf{Fn}\,A) & & \xrightarrow[\gamma]{} & & \mathsf{Fn}(DA)
\end{array}$$

where the right-most rectangle commutes by point (2) of Lemma E.1. Finally, the strength of $\overline{\mathsf{G}}\mathsf{en}$ is

$$\begin{array}{ccccccccc}
\widehat{A} \times G\widehat{B} & \longleftarrow & \widehat{R} \times G\widehat{S} & \longrightarrow & \mathsf{Fn}\,A \times G(\mathsf{Fn}\,B) & \xrightarrow{\mathsf{Fn}\,A\times\gamma} & \mathsf{Fn}\,A \times \mathsf{Fn}(DB) & \longrightarrow & \mathsf{Fn}(A \times DB) \\
{\scriptstyle\sigma}\downarrow & & \downarrow{\scriptstyle\sigma} & & \downarrow{\scriptstyle\sigma} & & & & \downarrow{\scriptstyle\mathsf{Fn}\,\sigma} \\
G(\widehat{A} \times \widehat{B}) & \longleftarrow & G(\widehat{R} \times \widehat{S}) & \longrightarrow & G(\mathsf{Fn}\,A \times \mathsf{Fn}\,B) & \longrightarrow & G(\mathsf{Fn}(A \times B)) & \xrightarrow[\gamma]{} & \mathsf{Fn}(D(A \times B))
\end{array}$$

where the right-most rectangle commutes by point (3) of Lemma E.1.

E.3 Lifting the effectful operation $(\mathsf{gen}\,p, \mathsf{Ber}\,p)$

We show that the following diagram commutes:

$$\begin{array}{ccccc}
1 & \longleftarrow & 1 & \longrightarrow & \mathsf{Fn}\,1 \\
{\scriptstyle\mathsf{gen}\,p}\downarrow & & \downarrow{\scriptstyle\mathsf{gen}\,p} & & \downarrow{\scriptstyle\mathsf{Fn}(\mathsf{Ber}\,p)} \\
\mathsf{Gen}(\mathsf{Piecewise}(1+1)) & \overset{\mathsf{id}}{\longleftarrow} & \mathsf{Gen}(\mathsf{Piecewise}(1+1)) & \xrightarrow[\gamma]{} & \mathsf{Fn}(\mathsf{Dist}(1+1))
\end{array}$$

The square on the left clearly commutes. For commutativity of the square on the right, we have the following for all (Γ, φ):

$$\begin{aligned}
&\gamma_{(\Gamma,\varphi)}\left((\mathsf{gen}\, p)_{(\Gamma,\varphi)}(\star)\right) \\
&= \gamma_{(\Gamma,\varphi)}\left(\nu\,(\alpha : \mathsf{Ber}\, p).\, [\varphi \wedge \alpha : \mathsf{inl}(\star), \varphi \wedge \neg\alpha : \mathsf{inr}(\star)]\right) \\
&= \lambda\, m.\, \left(v \leftarrow \mathsf{Ber}\, p;\ \mathsf{pure}(\mathsf{if}\ m \models \varphi \wedge \alpha\ \mathsf{then}\ \mathsf{inl}(\star)\ \mathsf{else}\ \mathsf{inr}(\star))\right) \\
&= \lambda\, m.\, \mathsf{Ber}\, p \\
&= \mathsf{Fn}(\mathsf{Ber}\, p)
\end{aligned}$$

E.4 Lifting the distributive laws

Let $G = \mathsf{Gen}$, $P = \mathsf{Piecewise}$, $\overline{G} = \overline{\mathsf{G}}\mathsf{en}$, $\overline{P} = \overline{\mathsf{P}}\mathsf{iecewise}$, $D = \mathsf{Dist}$, and $F = \mathsf{Fn}$.

E.4.1 Lifting δ_1. We lift the distributive law $(\delta_1, \mathsf{id}) : (GP, D) \to (PG, D)$ to $\overline{\delta}_1 : \overline{GP} \to \overline{PG}$, defined by the following diagram:

$$\begin{array}{ccccccccc}
GP\widehat{A} & \longleftarrow & GP\widehat{R} & \longrightarrow & GPFA & \xrightarrow{G\alpha} & GFA & \xrightarrow{\gamma} & FDA \\
\downarrow{\scriptstyle \delta_1} & & \downarrow{\scriptstyle \delta_1} & & \downarrow{\scriptstyle \delta_1} & & & & \downarrow{\scriptstyle \mathsf{id}} \\
PG\widehat{A} & \longleftarrow & PG\widehat{R} & \longrightarrow & PGFA & \xrightarrow[P\gamma]{} & PFDA & \xrightarrow[\alpha]{} & FDA
\end{array}$$

All the squares commute by naturality of δ_1. The remaining rectangle commutes because, for any $\nu\,(\alpha : \mathsf{Ber}\, p, \dots).\, [\varphi_i : f_i]_{i\in I}$ in $(GPFA)_{(\Gamma,\varphi)}$, the two routes through the rectangle are

$$\begin{aligned}
&\nu\,(\alpha : \mathsf{Ber}\, p, \dots).\, [\varphi_i : f_i]_{i\in I} \\
\overset{G\alpha}{\mapsto}\ &\nu\,(\alpha : \mathsf{Ber}\, p, \dots).\, \lambda\, m.\, \left\{ f_i(m) \text{ if } m \models \varphi_i \text{ for some } i \in I \right. \\
\overset{\gamma}{\mapsto}\ &\lambda\, m.\, v \leftarrow \mathsf{Ber}\, p;\ \dots; \left\{ \mathsf{pure}(f_i(m[\alpha \mapsto v, \dots])) \text{ if } m \models \varphi_i \text{ for some } i \in I \right.
\end{aligned}$$

and

$$\begin{aligned}
&\nu\,(\alpha : \mathsf{Ber}\, p, \dots).\, [\varphi_i : f_i]_{i\in I} \\
\overset{\delta_1}{\mapsto}\ &[\varphi_i : \nu\,(\alpha : \mathsf{Ber}\, p, \dots).\, f_i]_{i\in I} \\
\overset{P\gamma}{\mapsto}\ &[\varphi_i : \lambda\, m.\, (v \leftarrow \mathsf{Ber}\, p;\ \dots;\ \mathsf{pure}(f_i(m[\alpha \mapsto v, \dots])))]_{i\in I} \\
\overset{\alpha}{\mapsto}\ &\lambda\, m.\, \left\{ v \leftarrow \mathsf{Ber}\, p;\ \dots;\ \mathsf{pure}(f_i(m[\alpha \mapsto v, \dots])) \text{ if } m \models \varphi_i \text{ for some } i \in I \right.
\end{aligned}$$

These functions are piecewise equal under the partition $(\varphi_i)_{i\in I}$ of $\mathsf{Mod}(\Gamma, \varphi)$, hence equal. The distributive law axioms are inherited from δ_1.

E.4.2 Lifting δ_2. We lift the distributive law $(\delta_2, \mathsf{id}) : (PG, D) \to (GP, D)$ to $\overline{\delta}_2 : \overline{PG} \to \overline{GP}$, defined by the following diagram:

$$\begin{array}{ccccccccc}
PG\widehat{A} & \longleftarrow & PG\widehat{R} & \longrightarrow & PGFA & \xrightarrow{P\gamma} & PFDA & \xrightarrow{\alpha} & FDA \\
\downarrow{\scriptstyle \delta_2} & & \downarrow{\scriptstyle \delta_2} & & \downarrow{\scriptstyle \delta_2} & & & & \downarrow{\scriptstyle \mathsf{id}} \\
GP\widehat{A} & \longleftarrow & GP\widehat{R} & \longrightarrow & GPFA & \xrightarrow[G\alpha]{} & GFA & \xrightarrow[\gamma]{} & FDA
\end{array}$$

All the squares commute by naturality of δ_2. The remaining rectangle commutes because, for any $[\varphi_i : \nu\, \Delta_i.\, f_i]_{i \in I}$ in $(PGFA)_{(\Gamma,\varphi)}$, the two routes through the rectangle are

$$\begin{aligned}
&[\varphi_i : \nu\, (\alpha_i : \mathsf{Ber}\, p_i, \dots).\, f_i]_{i \in I} \\
&\overset{P\gamma}{\mapsto} [\varphi_i : \lambda\, m.\, (v_i \leftarrow \mathsf{Ber}\, p_i;\ \dots;\ \mathsf{pure}(f_i(m[\alpha_i \mapsto v_i, \dots])))]_{i \in I} \\
&\overset{\alpha}{\mapsto} \lambda\, m.\, \begin{cases} (v_i \leftarrow \mathsf{Ber}\, p_i;\ \dots;\ \mathsf{pure}(f_i(m[\alpha_i \mapsto v_i, \dots]))) \text{ if } m \models \varphi_i \text{ for some } i \in I \end{cases}
\end{aligned}$$

and

$$\begin{aligned}
&[\varphi_i : \nu\, (\alpha_i : \mathsf{Ber}\, p_i, \dots).\, f_i]_{i \in I} \\
&\overset{\delta_2}{\mapsto} \nu\, (\textstyle\sum_{i \in I} (\alpha_i : \mathsf{Ber}\, p_i, \dots)).\, [\varphi_i : f_i]_{i \in I} \\
&\overset{G\alpha}{\mapsto} \nu\, (\textstyle\sum_{i \in I} (\alpha_i : \mathsf{Ber}\, p_i, \dots)).\, \lambda\, m.\, \begin{cases} f_i(m) \text{ if } m \models \varphi_i \text{ for some } i \in I \end{cases} \\
&\overset{\gamma}{\mapsto} \lambda\, m.\, (v_i \leftarrow \mathsf{Ber}\, p_i;\ \dots)_{i \in I};\ \begin{cases} \mathsf{pure}(f_i(m[\alpha_i \mapsto v_i, \dots]_{i \in I})) \text{ if } m \models \varphi_i \text{ for some } i \in I \end{cases}
\end{aligned}$$

These functions are piecewise-equal under the partition $(\varphi_i)_{i \in I}$, hence equivalent. Note that, for these functions to be equal under each component φ_i of the partition, we use the fact that Dist is affine—this implies that sampling commands irrelevant to the result of each restriction to φ_i can be freely discarded. The distributive law axioms are inherited from δ_2.

F Proof of Proposition 5.4

Because Piecewise is sheafification (§C), a symbolic set is mergable if and only if it can be equipped with an algebra for Piecewise. The distributive law $\delta_2 : \mathsf{Piecewise}\, \mathsf{Gen} \to \mathsf{Gen}\, \mathsf{Piecewise}$ from §D can be used to show that if $\widehat{A}$ comes with an algebra map $\alpha : \mathsf{Piecewise}\, \widehat{A} \to \widehat{A}$ then so does $\mathsf{Piecewise}(\mathsf{Gen}\, \widehat{A})$, with algebra map $\mathsf{Piecewise}(\mathsf{Gen}\, \widehat{A}) \xrightarrow{\delta_{2,\widehat{A}}} \mathsf{Gen}(\mathsf{Piecewise}\, \widehat{A}) \xrightarrow{\mathsf{Gen}(\alpha)} \mathsf{Gen}\, \widehat{A}$.

G Fun

G.1 Syntax

$$\begin{aligned}
\sigma, \tau &::= \mathtt{Unit} \mid \mathtt{Bool} \mid \sigma + \tau \mid \sigma \times \tau \mid \sigma \to \tau \mid \mathtt{List}\, \tau \mid \mathtt{Prob}\, \tau \\
e &::= () \\
&\quad \mid \mathtt{true} \mid \mathtt{false} \mid \mathtt{if}(e_1, e_2, e_3) \\
&\quad \mid \mathtt{inl}\, e \mid \mathtt{inr}\, e \mid \mathtt{case}(e, x_1.e_1, x_2.e_2) \\
&\quad \mid (e_1, e_2) \mid \mathtt{fst}\, e \mid \mathtt{snd}\, e \\
&\quad \mid x \mid \lambda x.\, e \mid e_1\, e_2 \\
&\quad \mid \mathtt{nil} \mid \mathtt{cons}(e_1, e_2) \mid \mathtt{rec}(e_1, e_2, xyz.e_3) \\
&\quad \mid \mathtt{flip}\, p \mid \mathtt{pure}\, e \mid (x \leftarrow e_1; e_2)
\end{aligned}$$

G.2 Typing rules

$$\frac{}{\Gamma \vdash () : \mathtt{Unit}} \qquad \frac{}{\Gamma \vdash \mathtt{true} : \mathtt{Bool}} \qquad \frac{}{\Gamma \vdash \mathtt{false} : \mathtt{Bool}}$$

$$\frac{\Gamma \vdash e_1 : \mathtt{Bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \mathtt{if}(e_1, e_2, e_3) : \tau} \qquad \frac{\Gamma \vdash e : \sigma}{\Gamma \vdash \mathtt{inl}\, e : \sigma + \tau} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathtt{inr}\, e : \sigma + \tau}$$

$$\frac{\Gamma \vdash e : \sigma_1 + \sigma_2 \qquad \Gamma, x_1 : \sigma_1 \vdash e_1 : \tau \qquad \Gamma, x_2 : \sigma_2 \vdash e_2 : \tau}{\Gamma \vdash \mathtt{case}(e, x_1.e_1, x_2.e_2) : \tau} \qquad \frac{\Gamma \vdash e_1 : \sigma \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash (e_1, e_2) : \sigma \times \tau}$$

$$\frac{\Gamma \vdash e : \sigma \times \tau}{\Gamma \vdash \mathtt{fst}\, e : \sigma} \qquad \frac{\Gamma \vdash e : \sigma \times \tau}{\Gamma \vdash \mathtt{snd}\, e : \tau} \qquad \frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash \lambda x.\, e : \sigma \to \tau}$$

$$\frac{\Gamma \vdash e_1 : \sigma \to \tau \qquad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash e_1\, e_2 : \tau} \qquad \frac{}{\Gamma \vdash \mathtt{nil} : \mathtt{List}\, \tau} \qquad \frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \mathtt{List}\, \tau}{\Gamma \vdash \mathtt{cons}(e_1, e_2) : \mathtt{List}\, \tau}$$

$$\frac{\Gamma \vdash e_1 : \mathtt{List}\, \sigma \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma, x : \sigma, y : \mathtt{List}\, \sigma, z : \tau \vdash e_3 : \tau}{\Gamma \vdash \mathtt{rec}(e_1, e_2, xyz.e_3) : \tau} \qquad \frac{}{\Gamma \vdash \mathtt{flip}\, p : \mathtt{Prob}\, \mathtt{Bool}}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathtt{pure}\, e : \mathtt{Prob}\, \tau} \qquad \frac{\Gamma \vdash e_1 : \mathtt{Prob}\, \sigma \qquad \Gamma, x : \sigma \vdash e_2 : \mathtt{Prob}\, \tau}{\Gamma \vdash (x \leftarrow e_1;\ e_2) : \mathtt{Prob}\, \tau}$$

G.3 Interpretation in the symbolic metalanguage

G.3.1 *Interpretation of types.*

$$\begin{aligned}
[\![\mathtt{Unit}]\!] &= \mathbf{1} \\
[\![\mathtt{Bool}]\!] &= \mathbf{1} + \mathbf{1} \\
[\![\sigma + \tau]\!] &= [\![\sigma]\!] + [\![\tau]\!] \\
[\![\sigma \times \tau]\!] &= [\![\sigma]\!] \times [\![\tau]\!] \\
[\![\sigma \to \tau]\!] &= [\![\sigma]\!] \Rightarrow [\![\tau]\!] \\
[\![\mathtt{List}\, \tau]\!] &= \coprod_{n \in \mathbb{N}} [\![\tau]\!]^n \\
[\![\mathtt{Prob}\, \tau]\!] &= \mathrm{Gen}\, [\![\tau]\!]
\end{aligned}$$

G.3.2 Interpretation of terms.

$$
\begin{aligned}
[\![\Gamma \vdash e : \tau]\!] &: [\![\Gamma]\!] \to [\![\tau]\!] \\
[\![()]\!](\gamma) &= \star \\
[\![\texttt{true}]\!](\gamma) &= \mathsf{inl}(\star) \\
[\![\texttt{false}]\!](\gamma) &= \mathsf{inr}(\star) \\
[\![\texttt{if}(e_1, e_2, e_3)]\!](\gamma) &= \text{if } [\![e_1]\!](\gamma) \text{ then } [\![e_2]\!](\gamma) \text{ else } [\![e_3]\!](\gamma) \\
[\![\texttt{inl}(e)]\!](\gamma) &= \mathsf{inl}([\![e]\!](\gamma)) \\
[\![\texttt{inr}(e)]\!](\gamma) &= \mathsf{inr}([\![e]\!](\gamma)) \\
[\![\texttt{case}(e, x_1.e_1, x_2.e_2)]\!](\gamma) &= \text{case } [\![e]\!](\gamma) \text{ of } \{\mathsf{inl}(v_1) \Rightarrow [\![e_1]\!](\gamma, v_1);\ \mathsf{inl}(v_2) \Rightarrow [\![e_2]\!](\gamma, v_2)\} \\
[\![(e_1, e_2)]\!](\gamma) &= ([\![e_1]\!](\gamma), [\![e_2]\!](\gamma)) \\
[\![\texttt{fst}\, e]\!](\gamma) &= \pi_1([\![e]\!](\gamma)) \\
[\![\texttt{snd}\, e]\!](\gamma) &= \pi_2([\![e]\!](\gamma)) \\
[\![x]\!](\gamma) &= \gamma(x) \\
[\![\lambda x.\, e]\!](\gamma) &= \lambda v.\ [\![e]\!](\gamma, v) \\
[\![e_1\, e_2]\!](\gamma) &= [\![e_1]\!](\gamma)([\![e_2]\!](\gamma)) \\
[\![\texttt{nil}]\!](\gamma) &= \mathsf{inj}_0() \\
[\![\texttt{cons}(e_1, e_2)]\!](\gamma) &= \text{case } [\![e_2]\!](\gamma) \text{ of } \left\{\mathsf{inj}_n(\vec{v}) \Rightarrow \mathsf{inj}_{n+1}(v, \vec{v})\right\}_{n \in \mathbb{N}} \text{ where } [\![e_1]\!](\gamma) = v \\
[\![\texttt{rec}_{\texttt{List}\,\sigma}(e_1, e_2, xyz.e_3) : \tau]\!](\gamma) &= \text{case } [\![e_2]\!](\gamma) \text{ of } \left\{\mathsf{inj}_n(\vec{v}) \Rightarrow \mathsf{go}_n(\vec{v})\right\}_{n \in \mathbb{N}} \text{ where} \\
&\qquad \mathsf{go}_n : [\![\sigma]\!]^n \to [\![\tau]\!] \\
&\qquad \mathsf{go}_0() = [\![e_2]\!](\gamma) \\
&\qquad \mathsf{go}_{n+1}(v, \vec{v}) = [\![e_3]\!](\gamma[x \mapsto v, y \mapsto \vec{v}, z \mapsto \mathsf{go}_n(\vec{v})]) \\
[\![\texttt{flip}\, p]\!](\gamma) &= \mathsf{gen}\, p \\
[\![\texttt{pure}\, e]\!](\gamma) &= \mathsf{pure}([\![e]\!](\gamma)) \\
[\![x \leftarrow e_1;\ e_2]\!](\gamma) &= \left(v \leftarrow [\![e_1]\!](\gamma);\ [\![e_2]\!](\gamma, v)\right)
\end{aligned}
$$

G.4 Semantics in $\mathrm{SymSet}_{\mathsf{m}}$

G.4.1 Semantics of types.

$$
\begin{aligned}
\mathcal{S}_{\mathsf{m}}[\![\texttt{Unit}]\!] &= \widehat{\mathbf{1}} \\
\mathcal{S}_{\mathsf{m}}[\![\texttt{Bool}]\!] &= \mathbb{B} \\
\mathcal{S}_{\mathsf{m}}[\![\sigma + \tau]\!] &= \mathsf{Piecewise}(\mathcal{S}_{\mathsf{m}}[\![\sigma]\!] + \mathcal{S}_{\mathsf{m}}[\![\tau]\!]) \\
\mathcal{S}_{\mathsf{m}}[\![\sigma \times \tau]\!] &= \mathcal{S}_{\mathsf{m}}[\![\sigma]\!] \times \mathcal{S}_{\mathsf{m}}[\![\tau]\!] \\
\mathcal{S}_{\mathsf{m}}[\![\sigma \to \tau]\!] &= \mathcal{S}_{\mathsf{m}}[\![\sigma]\!] \Rightarrow \mathcal{S}_{\mathsf{m}}[\![\tau]\!] \\
\mathcal{S}_{\mathsf{m}}[\![\texttt{List}\,\tau]\!] &= \mathsf{Piecewise}(\coprod_{n \in \mathbb{N}} \mathcal{S}_{\mathsf{m}}[\![\tau]\!]^n) \\
\mathcal{S}_{\mathsf{m}}[\![\texttt{Prob}\,\tau]\!] &= \mathsf{Gen}\, \mathcal{S}_{\mathsf{m}}[\![\tau]\!]
\end{aligned}
$$

G.4.2 Merge operations.

$$
\begin{aligned}
\star \sqcup \star &= \star \\
[\psi] \sqcup_{\varphi,\varphi'} [\psi'] &= [(\psi \wedge \varphi) \vee (\psi' \vee \varphi')] \\
[\varphi : \mathsf{inl}(v), \psi : \mathsf{inr}(w)] \sqcup [\varphi' : \mathsf{inl}(v'), \psi' : \mathsf{inr}(w')] &= [(\varphi \vee \varphi') : \mathsf{inl}(v \sqcup v'), (\psi \vee \psi') : \mathsf{inr}(w \sqcup w')] \\
(v, w) \sqcup (v', w') &= (v \sqcup v', w \sqcup w') \\
(f \sqcup g) &= \big((r, v) \mapsto f_{(\Gamma,\varphi)}(r, v) \sqcup g_{(\Gamma,\varphi)}(r, v)\big)_{(\Gamma,\varphi)} \\
[\varphi_i : \mathsf{inj}_{n_i}(\vec{v}_i)]_{i \in I} \sqcup [\varphi'_i : \mathsf{inj}_{n_i}(\vec{v}'_i)] &= [(\varphi_i \vee \varphi'_i) : \mathsf{inj}_{n_i}(\vec{v}_i \sqcup \vec{v}'_i)] \\
(\nu\, \Gamma.\, v) \sqcup (\nu\, \Gamma'.\, v') &= \nu\, (\Gamma + \Gamma').\, (v \sqcup v')
\end{aligned}
$$

G.4.3 Semantics of terms.

$\mathcal{S}_{\mathsf{m}}[\![\Gamma \vdash e : \tau]\!] : \mathcal{S}_{\mathsf{m}}[\![\Gamma]\!] \to \mathcal{S}_{\mathsf{m}}[\![\tau]\!]$

$\mathcal{S}_{\mathsf{m}}[\![()]\!]_{(\Gamma,\varphi)}(\gamma) = \star$

$\mathcal{S}_{\mathsf{m}}[\![\texttt{true}]\!]_{(\Gamma,\varphi)}(\gamma) = [\top]$

$\mathcal{S}_{\mathsf{m}}[\![\texttt{false}]\!]_{(\Gamma,\varphi)}(\gamma) = [\bot]$

$\mathcal{S}_{\mathsf{m}}[\![\texttt{if}(e_1, e_2, e_3)]\!]_{(\Gamma,\varphi)}(\gamma) = \mathcal{S}_{\mathsf{m}}[\![e_2]\!]_{(\Gamma,\varphi\wedge\psi)}(\gamma|_{\varphi\wedge\psi}) \sqcup \mathcal{S}_{\mathsf{m}}[\![e_3]\!]_{(\Gamma,\varphi\wedge\neg\psi)}(\gamma|_{\varphi\wedge\neg\psi})$ where $\mathcal{S}_{\mathsf{m}}[\![e_1]\!](\gamma) = [\psi]$

$\mathcal{S}_{\mathsf{m}}[\![\texttt{inl}\ e]\!]_{(\Gamma,\varphi)}(\gamma) = [\varphi : \mathsf{inl}(\mathcal{S}_{\mathsf{m}}[\![e]\!]_{(\Gamma,\varphi)}(\gamma))]$

$\mathcal{S}_{\mathsf{m}}[\![\texttt{inr}\ e]\!]_{(\Gamma,\varphi)}(\gamma) = [\varphi : \mathsf{inr}(\mathcal{S}_{\mathsf{m}}[\![e]\!]_{(\Gamma,\varphi)}(\gamma))]$

$\mathcal{S}_{\mathsf{m}}[\![\texttt{case}(e, x_1.e_1, x_2.e_2)]\!]_{(\Gamma,\varphi)}(\gamma) = \mathcal{S}_{\mathsf{m}}[\![e_1]\!]_{(\Gamma,\varphi_1)}(\gamma|_{\varphi_1}[x_1 \mapsto v_1]) \sqcup \mathcal{S}_{\mathsf{m}}[\![e_2]\!]_{(\Gamma,\varphi_2)}(\gamma|_{\varphi_2}[x_2 \mapsto v_2])$ where

$\mathcal{S}_{\mathsf{m}}[\![e]\!]_{(\Gamma,\varphi)}(\gamma) = [\varphi_1 : \mathsf{inl}(v_1), \varphi_2 : \mathsf{inr}(v_2)]$

$\mathcal{S}_{\mathsf{m}}[\![(e_1, e_2)]\!]_{(\Gamma,\varphi)}(\gamma) = (\mathcal{S}_{\mathsf{m}}[\![e_1]\!]_{(\Gamma,\varphi)}(\gamma), \mathcal{S}_{\mathsf{m}}[\![e_2]\!]_{(\Gamma,\varphi)}(\gamma))$

$\mathcal{S}_{\mathsf{m}}[\![\texttt{fst}\ e]\!]_{(\Gamma,\varphi)}(\gamma) = \pi_1(\mathcal{S}_{\mathsf{m}}[\![e]\!]_{(\Gamma,\varphi)}(\gamma))$

$\mathcal{S}_{\mathsf{m}}[\![\texttt{snd}\ e]\!]_{(\Gamma,\varphi)}(\gamma) = \pi_2(\mathcal{S}_{\mathsf{m}}[\![e]\!]_{(\Gamma,\varphi)}(\gamma))$

$\mathcal{S}_{\mathsf{m}}[\![x]\!]_{(\Gamma,\varphi)}(\gamma) = \gamma(x)$

$\mathcal{S}_{\mathsf{m}}[\![\lambda x.\, e]\!]_{(\Gamma,\varphi)}(\gamma) = \Big((r, v) \mapsto \mathcal{S}_{\mathsf{m}}[\![e]\!]_{(\Gamma',\varphi')}(r(\gamma)|_{\varphi'}[x \mapsto v])\Big)_{(\Gamma',\varphi')}$

$\mathcal{S}_{\mathsf{m}}[\![e_1\ e_2]\!]_{(\Gamma,\varphi)}(\gamma) = \mathcal{S}_{\mathsf{m}}[\![e_1]\!]_{(\Gamma,\varphi)}(\mathsf{id}_{(\Gamma,\varphi)}, \mathcal{S}_{\mathsf{m}}[\![e_2]\!]_{(\Gamma,\varphi)}(\gamma))$

$\mathcal{S}_{\mathsf{m}}[\![\texttt{nil}]\!]_{(\Gamma,\varphi)}(\hat{\gamma}) = [\varphi : \mathsf{inj}_0()]$

$\mathcal{S}_{\mathsf{m}}[\![\texttt{cons}(e_1, e_2)]\!]_{(\Gamma,\varphi)}(\gamma) = [\varphi_i : \mathsf{inj}_{n_i+1}(v|_{\varphi_i}, \vec{v}_i)]_{i\in I}$ where

$\mathcal{S}_{\mathsf{m}}[\![e_1]\!]_{(\Gamma,\varphi)} = v$

$\mathcal{S}_{\mathsf{m}}[\![e_2]\!]_{(\Gamma,\varphi_i)} = [\varphi_i : \mathsf{inj}_{n_i}(\vec{v}_i)]_{i\in I}$

$\mathcal{S}_{\mathsf{m}}[\![\texttt{rec}(e_1, e_2, xyz.e_3)]\!](\gamma) = \bigsqcup_{i\in I} \mathrm{go}_{n_i,(\Gamma,\varphi_i)}(\vec{v}_i)$ where

$\mathcal{S}_{\mathsf{m}}[\![e_1]\!](\gamma) = [\varphi_i : \mathsf{inj}_{n_i}(\vec{v}_i)]_{i\in I}$

$\mathrm{go}_n : \mathcal{S}_{\mathsf{m}}[\![\sigma]\!]^n \to \mathcal{S}_{\mathsf{m}}[\![\tau]\!]$

$\mathrm{go}_{0,(\Gamma,\varphi)}() = \mathcal{S}_{\mathsf{m}}[\![e_2]\!]_{(\Gamma,\varphi)}(\gamma)$

$\mathrm{go}_{n+1,(\Gamma,\varphi)}(v, \vec{v}) = \mathcal{S}_{\mathsf{m}}[\![e_3]\!]_{(\Gamma,\varphi)}(\gamma[x \mapsto v, y \mapsto \vec{v}, z \mapsto \mathrm{go}_{n,(\Gamma,\varphi)}(\vec{v})])$

$\mathcal{S}_{\mathsf{m}}[\![\texttt{flip}\ p]\!]_{(\Gamma,\varphi)}(\gamma) = \nu\,(\alpha : \mathrm{Ber}\,p).\,[\alpha]$

$\mathcal{S}_{\mathsf{m}}[\![\texttt{pure}\ e]\!]_{(\Gamma,\varphi)}(\gamma) = \nu\,\varnothing.\ \mathcal{S}_{\mathsf{m}}[\![e]\!]_{(\Gamma,\varphi)}(\gamma)$

$\mathcal{S}_{\mathsf{m}}[\![x \leftarrow e_1;\ e_2]\!]_{(\Gamma,\varphi)}(\gamma) = \nu\,(\Gamma_1 + \Gamma_2).\,v_2$ where

$\mathcal{S}_{\mathsf{m}}[\![e_1]\!]_{(\Gamma,\varphi)}(\gamma) = \nu\,\Gamma_1.\,v_1$

$\mathcal{S}_{\mathsf{m}}[\![e_2]\!]_{(\Gamma,\varphi)}(\gamma[x \mapsto v_1]) = \nu\,\Gamma_2.\,v_2$

Received 2025-11-13; accepted 2026-04-03