Running Probabilistic Programs Backward

Neil Toronto and Jay McCarthy
neil.toronto@gmail.com and jay@cs.byu.edu

PLT @ Brigham Young University, Provo, Utah, USA

Abstract. To be useful in Bayesian practice, a probabilistic language must support conditioning: imposing constraints in a way that preserves the relative probabilities of program outputs. Every language to date that supports probabilistic conditioning also places seemingly artificial restrictions on legal programs, such as disallowing recursion and restricting conditions to simple equality constraints such as $x = 2$.
We develop a semantics for a first-order language with recursion, probabilistic choice and conditioning. Distributions over program outputs are defined by the probabilities of their preimages, a measure-theoretic approach that ensures the language is not artificially limited.
Preimages are generally uncomputable, so we derive an approximating semantics for computing rectangular covers of preimages. We implement the approximating semantics in Haskell and Typed Racket, and demonstrate its expressive power using stochastic ray tracing.

Keywords: Probability, Semantics, Domain-Specific Languages

1 Introduction

It is primarily Bayesian practice that drives probabilistic language development. To be useful, a probabilistic language must support **conditioning**, or imposing constraints in a way that preserves the relative probabilities of outputs.

Unfortunately, there is currently no efficient probabilistic language implementation that supports conditioning and does not restrict legal programs. Most commonly, languages that support conditioning disallow recursion, allow only discrete or continuous distributions, and restrict conditions to the form $x = c$.

These common language restrictions arise from reasoning about probability using **densities**, which are functions from random values to *changes* in probability. While simple and convenient, densities have many limitations. For example, densities for random values with different dimension are incomparable, and they cannot be defined on infinite products. Either limitation rules out recursion.

Densities generally cannot define distributions for the outputs of discontinuous functions. For example, suppose we want to model a thermometer that reports in the range $[0, 100]$, and that the temperature it would report (if it could) is distributed according to a bell curve. We might encode the process as

$$\begin{aligned} \mathsf{t}' \;:=\; & \mathsf{let}\;\; \mathsf{t} := \mathsf{normal}\ \mu\ 1 \\ & \mathsf{in}\;\; \mathsf{max}\ 0\ (\mathsf{min}\ 100\ \mathsf{t}) \end{aligned} \tag{1}$$

While t's distribution has a density, the distribution of t' does not.

Densities disallow all but the simplest conditions. **Bayes' law for densities** gives the density of x given an observed y in terms of other densities:

$$p_x(x \,|\, y) \;=\; \frac{p_y(y \,|\, x) \cdot \pi_x(x)}{\int p_y(y \,|\, x) \cdot \pi_x(x)\; dx} \tag{2}$$

Bayesians interpret probabilistic processes as defining p_y and π_x, and use (2) to find the distribution of "x given $y = c$." Even though "x given $x + y = 0$" has perfectly sensible distribution, Bayes' law for densities cannot express it.

Measure-theoretic probability [13] is widely believed to be able to define every reasonable distribution that densities cannot. It mainly does this by *assigning probabilities to sets* instead of *assigning changes in probability to values*. Functions that do so are called **probability measures**.

If a probability measure P assigns probabilities to subsets of X and $f : X \to Y$, then **preimage measure** defines the distribution over subsets of Y:

$$\Pr[B] \;=\; P(f^{-1}(B)) \tag{3}$$

The preimage $f^{-1}(B) = \{a \in X \mid f(a) \in B\}$ is the subset of X for which f yields a value in B. Preimages under f are well-defined regardless of discontinuities.

Measure-theoretic probability supports any kind of condition. If $\Pr[B] > 0$, the probability of $B' \subseteq Y$ given $B \subseteq Y$ is $\Pr[B' \,|\, B] = \Pr[B' \cap B] \;/\; \Pr[B]$. If $\Pr[B] = 0$, conditional probabilities can be calculated as the limit of $\Pr[B' \,|\, B_n]$ for positive-probability $B_1 \supseteq B_2 \supseteq B_3 \supseteq \cdots$ whose intersection is B. For example, if $Y = \mathbb{R} \times \mathbb{R}$, the distribution of "$\langle x, y\rangle \in Y$ given $x + y = 0$" can be calculated using the descending sequence $B_n = \{\langle x, y\rangle \in Y \mid |x + y| < 2^{-n}\}$.

Only special families of **measurable** sets can be assigned probabilities. Proving measurability, taking limits, and other complications tend to make measure-theoretic probability less attractive, even though it is strictly more powerful.

1.1 Measure-Theoretic Semantics

Most purely functional languages allow only nontermination as a side effect, and not probabilistic choice. Programmers therefore encode probabilistic programs as functions from random sources to outputs. Monads and other categorical classes such as idioms (i.e. applicative functors) can make doing so easier [10,26].

It seems this approach should make it easy to interpret probabilistic programs measure-theoretically. For a probabilistic program $f : X \to Y$, the probability measure on output sets $B \subseteq Y$ should be defined by preimages of B under f and the probability measure on X. Unfortunately, it is difficult to turn this simple-sounding idea into a compositional semantics, for the following reasons.

1. Preimages can be defined only for functions with observable domains, which excludes lambdas.
2. If subsets of X and Y must be measurable, taking preimages under f must preserve measurability (we say f itself is measurable). Proving the conditions under which this is true is difficult, especially if f may not terminate.

3. It is very difficult to define probability measures for arbitrary spaces of measurable functions [3].

Implementing a language based on such a semantics is complicated because

4. Contemporary mathematics is unlike any implementation's host language.
5. It requires running Turing-equivalent programs backward, efficiently, on possibly uncountable sets of outputs.

We address 1 and 4 by developing our semantics in λ_{ZFC} [27], a λ-calculus with infinite sets, and both extensional and intensional functions. We address 5 by deriving and implementing a *conservative approximation* of the semantics.

We have addressed difficulty 2 by proving that all programs' interpretations as functions are measurable if language primitives are measurable, including uncomputable primitives such as limits and real equality, regardless of nontermination. The proof interprets programs as extensional functions and applies well-known theorems from measure theory. The required machinery does not fit in this report; see the full version [25] for the entire development.

For difficulty 3, we have discovered that the "first-orderness" of arrows [9] is a perfect fit for the "first-orderness" of measure theory.

1.2 Arrow Solution Overview

We define *exact* and *approximating* semantics. The exact semantics includes

- A semantic function which, like the arrow calculus semantic function [16], transforms first-order programs into the computations of an arbitrary arrow.
- Arrows for evaluating expressions in different ways.

This commutative diagram describes the relationships among the exact arrows:

$$\begin{tikzcd}
\mathsf{X} \rightsquigarrow_{\bot} \mathsf{Y} \arrow[r, "\mathsf{lift}_{\mathsf{pre}}"] \arrow[d, "\eta_{\bot^*}"'] & \mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y} \arrow[d, "\eta_{\mathsf{pre}^*}"] \\
\mathsf{X} \rightsquigarrow_{\bot^*} \mathsf{Y} \arrow[r, "\mathsf{lift}_{\mathsf{pre}^*}"'] & \mathsf{X} \underset{\mathsf{pre}^*}{\rightsquigarrow} \mathsf{Y}
\end{tikzcd} \tag{4}$$

In the top row, $\mathsf{X} \rightsquigarrow_{\bot} \mathsf{Y}$ arrow computations are functions that may raise errors and $\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}$ instances compute preimages. The computations of the arrows in the bottom row are like those in the top, except they thread an infinite store of random values, and can be constructed to always terminate. Most of our correctness theorems rely on proofs that every morphism in (4) is a homomorphism.

The approximating semantics has the same semantic function, but its arrows $\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow}' \mathsf{Y}$ and $\mathsf{X} \underset{\mathsf{pre}^*}{\rightsquigarrow}' \mathsf{Y}$ compute conservative approximations. Given a library for representing and operating on rectangular sets, it is directly implementable.

1.3 Operational Metalanguage

We write programs in λ_{ZFC} [27], an untyped, call-by-value, operational λ-calculus designed for deriving implementable programs from contemporary mathematics.

Many mathematical areas are agnostic to their foundations, but measure theory is developed explicitly in **ZFC**: Zermelo-Fraenkel set theory with Choice. ZFC's intensional functions are first-order and it has no general recursion, which makes implementing a language defined by a transformation into ZFC difficult. Targeting λ_{ZFC} instead allows creating an exact semantics and deriving an approximating semantics without changing languages.

In λ_{ZFC}, essentially every set is a value, as well as every lambda and every set of lambdas. All operations, including operations on infinite sets, are assumed to complete instantly if they terminate. Almost everything definable in ZFC can be defined by a finite λ_{ZFC} program, and essentially every ZFC theorem applies to λ_{ZFC}'s set values without alteration. Proofs about λ_{ZFC}'s set values apply directly to ZFC sets, assuming the existence of an inaccessible cardinal.[1]

In λ_{ZFC}, algebraic data structures are encoded as sets; e.g. the pair $\langle \mathsf{x}, \mathsf{y} \rangle$ can be encoded as $\{\{\mathsf{x}\}, \{\mathsf{x}, \mathsf{y}\}\}$. Only the *existence* of encodings into sets is important, as it means data structures inherit a defining characteristic of sets: strictness. More precisely, the lengths of paths to data structure leaves is unbounded, but each path must be finite. Less precisely, data may be "infinitely wide" (such as $\mathbb{R}$) but not "infinitely tall" (such as infinite trees and lists).

Though λ_{ZFC} is untyped, it helps in this work to define an auxiliary type system. It is manually checked, polymorphic, and characterized by these rules:

- A free type variable is universally quantified; if uppercase, it denotes a set.
- A set denotes a member of that set.
- $\mathsf{x} \Rightarrow \mathsf{y}$ denotes a partial function.
- $\langle \mathsf{x}, \mathsf{y} \rangle$ denotes a pair of values with types x and y.
- $\mathsf{Set}\ \mathsf{x}$ denotes a set with members of type x.

Because the type $\mathsf{Set}\ \mathsf{X}$ denotes the same values as the set $\mathcal{P}\ \mathsf{X}$ (i.e. subsets of the set X) we regard them as equivalent. Similarly, $\langle \mathsf{X}, \mathsf{Y} \rangle$ is equivalent to $\mathsf{X} \times \mathsf{Y}$.

Examples of types are those of the λ_{ZFC} primitives membership $(\in) : \mathsf{x} \Rightarrow \mathsf{Set}\ \mathsf{x} \Rightarrow \mathsf{Bool}$, powerset $\mathcal{P} : \mathsf{Set}\ \mathsf{x} \Rightarrow \mathsf{Set}\ (\mathsf{Set}\ \mathsf{x})$, big union $\bigcup : \mathsf{Set}\ (\mathsf{Set}\ \mathsf{x}) \Rightarrow \mathsf{Set}\ \mathsf{x}$, and the `map`-like $\mathsf{image} : (\mathsf{x} \Rightarrow \mathsf{y}) \Rightarrow \mathsf{Set}\ \mathsf{x} \Rightarrow \mathsf{Set}\ \mathsf{y}$.

We use heavily sugared syntax, with automatic currying, binding forms such as indexed unions $\bigcup_{x \in e_A} e$, destructuring binds as in $\mathsf{swap}\ \langle \mathsf{x}, \mathsf{y} \rangle := \langle \mathsf{y}, \mathsf{x} \rangle$, and comprehensions like $\{\mathsf{x} \in \mathsf{A} \mid \mathsf{x} \in \mathsf{B}\}$. We assume logical operators, bounded quantifiers, and typical set operations are defined.

In set theory, extensional functions are encoded as sets of input-output pairs; e.g. the increment function for the natural numbers is $\{\langle 0, 1 \rangle, \langle 1, 2 \rangle, \langle 2, 3 \rangle, ...\}$. We call these **mappings** and intensional functions **lambdas**, and use **function** to mean either. As with lambdas, we use adjacency (e.g. $(\mathsf{f}\ \mathsf{x})$) to apply mappings.

The set $\mathsf{J} \to \mathsf{X}$ contains all the total mappings from J to X; equivalently, all the vectors of X indexed by J (which may be infinite). The function

$$\begin{aligned} &\pi : \mathsf{J} \Rightarrow (\mathsf{J} \to \mathsf{X}) \Rightarrow \mathsf{X} \\ &\pi\ \mathsf{j}\ \mathsf{f}\ := \ \mathsf{f}\ \mathsf{j} \end{aligned} \tag{5}$$

produces projections. This is particularly useful when f is unnamed.

[1] A modest assumption, as ZFC+κ is a smaller theory than Coq's [4].

Any λ_{ZFC} term e used as a truth statement means "e reduces to true." Therefore, the terms $(\lambda\,\mathsf{a}.\,\mathsf{a})\ 1$ and 1 are (externally) unequal, but $(\lambda\,\mathsf{a}.\,\mathsf{a})\ 1 = 1$.

Because of the way λ_{ZFC}'s lambda terms are defined, lambda equality is alpha equivalence. For example, $(\lambda\,\mathsf{a}.\,\mathsf{a}) = (\lambda\,\mathsf{b}.\,\mathsf{b})$, but not $(\lambda\,\mathsf{a}.\,2) = (\lambda\,\mathsf{a}.\,1+1)$.

If $e_1 = e_2$, then e_1 and e_2 both terminate, and substituting one for the other in an expression does not change its value. Substitution is also safe if both e_1 and e_2 do not terminate, leading to a coarser notion of equivalence.

Definition 1 (observational equivalence). *For terms e_1 and e_2, $e_1 \equiv e_2$ when $e_1 = e_2$, or both e_1 and e_2 do not terminate.*

It might seem helpful to define basic equivalence even more coarsely. However, we want internal equality and external equivalence to be similar, and we want to be able to extend "$\equiv$" with type-specific rules.

We elide proofs to save space. They are in the full version of this paper [25].

2 Arrows and First-Order Semantics

Like monads [29] and idioms [18], arrows [9] thread effects through computations in a way that imposes structure. But arrow computations are always

- Function-like: An arrow computation of type $\mathsf{x} \rightsquigarrow \mathsf{y}$ must behave like a corresponding function of type $\mathsf{x} \Rightarrow \mathsf{y}$ (in a sense we explain shortly).
- First-order: There is no way to derive a computation $\mathsf{app} : \langle \mathsf{x} \rightsquigarrow \mathsf{y}, \mathsf{x}\rangle \rightsquigarrow \mathsf{y}$ from an arrow's minimal definition.

The first property makes arrows a good fit for a compositional translation from expressions to pure functions that operate on random sources. The second property makes arrows a good fit for a measure-theoretic semantics in particular, as app's corresponding function is generally not measurable [3].

2.1 Alternative Arrow Definitions and Laws

To make applying measure-theoretic theorems easier, and to simplify interpreting let-calculus expressions as arrow computations, we do not give typical minimal arrow definitions. For each arrow a, instead of $\mathsf{first}_{\mathsf{a}}$, we define $(\&\&\&_{\mathsf{a}})$. This combinator is typically called **fanout**, but its use will be clearer if we call it **pairing**. One way to strengthen an arrow a is to define an additional combinator $\mathsf{left}_{\mathsf{a}}$, which can be used to choose an arrow computation based on the result of another. Again, we define a different combinator, $\mathsf{ifte}_{\mathsf{a}}$ ("if-then-else").

In a nonstrict λ-calculus, defining a choice combinator allows writing recursive functions using nothing but arrow combinators and lifted, pure functions. However, a strict λ-calculus needs an extra combinator lazy for deferring conditional branches. For example, define the **function arrow** with choice:

$$\begin{aligned} \mathsf{arr}\ \mathsf{f} \ &:=\ \mathsf{f} & \mathsf{ifte}\ \mathsf{f}_1\ \mathsf{f}_2\ \mathsf{f}_3\ \mathsf{a} \ &:=\ \mathsf{if}\ (\mathsf{f}_1\ \mathsf{a})\ (\mathsf{f}_2\ \mathsf{a})\ (\mathsf{f}_3\ \mathsf{a}) \\ (\mathsf{f}_1 \ggg \mathsf{f}_2)\ \mathsf{a} \ &:=\ \mathsf{f}_2\ (\mathsf{f}_1\ \mathsf{a}) & \mathsf{lazy}\ \mathsf{f}\ \mathsf{a} \ &:=\ \mathsf{f}\ 0\ \mathsf{a} \\ (\mathsf{f}_1 \ \&\&\&\ \mathsf{f}_2)\ \mathsf{a} \ &:=\ \langle \mathsf{f}_1\ \mathsf{a}, \mathsf{f}_2, \mathsf{a}\rangle \end{aligned} \tag{6}$$

and try to define the following recursive function:

$$\mathsf{halt\text{-}on\text{-}true} \;:=\; \mathsf{ifte}\ (\mathsf{arr\ id})\ (\mathsf{arr\ id})\ \mathsf{halt\text{-}on\text{-}true} \tag{7}$$

In a strict λ-calculus, the defining expression does not terminate. But if the "else" branch is $\mathsf{lazy}\ \lambda 0.\,\mathsf{halt\text{-}on\text{-}true}$, it loops only when applied to false.

All of our arrows are arrows with choice, so we simply call them arrows.

Definition 2 (arrow). *Let* $1 := \{0\}$. *A binary type constructor* $(\rightsquigarrow_{\mathsf{a}})$ *and*

$$\begin{aligned}
\mathsf{arr_a} &: (\mathsf{x} \Rightarrow \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \\
(\ggg_{\mathsf{a}}) &: (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \Rightarrow (\mathsf{y} \rightsquigarrow_{\mathsf{a}} \mathsf{z}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{z}) \\
(\&\&\&_{\mathsf{a}}) &: (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{z}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \langle \mathsf{y},\mathsf{z}\rangle) \\
\mathsf{ifte_a} &: (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{Bool}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \\
\mathsf{lazy_a} &: (1 \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y})) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y})
\end{aligned} \tag{8}$$

*define an **arrow** if certain monoid, homomorphism, and structural laws hold.*

The arrow homomorphism laws can be put in terms of more general homomorphism properties that deal with distributing an arrow-to-arrow lift.

Definition 3 (arrow homomorphism). $\mathsf{lift_b} : (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{b}} \mathsf{y})$ *is an **arrow homomorphism** from arrow* a *to arrow* b *if these distributive laws hold:*

$$\mathsf{lift_b}\ (\mathsf{arr_a\ f}) \equiv \mathsf{arr_b\ f} \tag{9}$$

$$\mathsf{lift_b}\ (\mathsf{f_1} \ggg_{\mathsf{a}} \mathsf{f_2}) \equiv (\mathsf{lift_b\ f_1}) \ggg_{\mathsf{b}} (\mathsf{lift_b\ f_2}) \tag{10}$$

$$\mathsf{lift_b}\ (\mathsf{f_1}\ \&\&\&_{\mathsf{a}}\ \mathsf{f_2}) \equiv (\mathsf{lift_b\ f_1})\ \&\&\&_{\mathsf{b}}\ (\mathsf{lift_b\ f_2}) \tag{11}$$

$$\mathsf{lift_b}\ (\mathsf{ifte_a\ f_1\ f_2\ f_3}) \equiv \mathsf{ifte_b}\ (\mathsf{lift_b\ f_1})\ (\mathsf{lift_b\ f_2})\ (\mathsf{lift_b\ f_3}) \tag{12}$$

$$\mathsf{lift_b}\ (\mathsf{lazy_a\ f}) \equiv \mathsf{lazy_b}\ \lambda 0.\,\mathsf{lift_b}\ (\mathsf{f}\ 0) \tag{13}$$

The arrow homomorphism laws state that $\mathsf{arr_a} : (\mathsf{x} \Rightarrow \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y})$ must be a homomorphism from the function arrow (6) to arrow a. Roughly, arrow computations that do not use additional combinators can be transformed into $\mathsf{arr_a}$ applied to a pure computation. They must be *function-like*.

To prove arrow laws, we prove arrows are *epimorphic* to arrows for which the laws are known to hold. (Isomorphism is sufficient but not necessary.)

Definition 4 (arrow epimorphism). *An arrow homomorphism* $\mathsf{lift_b} : (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{b}} \mathsf{y})$ *that has a right inverse is an **arrow epimorphism** from* a *to* b.

Theorem 1. *If* $\mathsf{lift_b} : (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{b}} \mathsf{y})$ *is an arrow epimorphism and the combinators of* a *define an arrow, then the combinators of* b *define an arrow.*

2.2 First-Order Let-Calculus Semantics

Fig. 1 defines a transformation from a first-order let-calculus to arrow computations for any arrow a. A program is a sequence of definition statements followed

$$\begin{aligned}
p &::\equiv x := e;\ \ldots\ ; e \\
e &::\equiv x\ e \mid \mathsf{let}\ e\ e \mid \mathsf{env}\ n \mid \langle e, e\rangle \mid \mathsf{fst}\ e \mid \mathsf{snd}\ e \mid \mathsf{if}\ e\ e\ e \mid v \\
v &::\equiv \text{[first-order constants]}
\end{aligned}$$

$$\llbracket x := e;\ \ldots\ ; e_b \rrbracket_{\mathsf{a}} :\equiv x := \llbracket e \rrbracket_{\mathsf{a}}\ ;\ \ldots\ ; \llbracket e_b \rrbracket_{\mathsf{a}}$$

$$\begin{aligned}
\llbracket x\ e \rrbracket_{\mathsf{a}} &:\equiv \llbracket \langle e, \langle\rangle \rangle \rrbracket_{\mathsf{a}} \ggg_{\mathsf{a}} x \\
\llbracket \langle e_1, e_2 \rangle \rrbracket_{\mathsf{a}} &:\equiv \llbracket e_1 \rrbracket_{\mathsf{a}}\ \&\&\&_{\mathsf{a}}\ \llbracket e_2 \rrbracket_{\mathsf{a}} \\
\llbracket \mathsf{fst}\ e \rrbracket_{\mathsf{a}} &:\equiv \llbracket e \rrbracket_{\mathsf{a}} \ggg_{\mathsf{a}} \mathsf{arr}_{\mathsf{a}}\ \mathsf{fst} \\
\llbracket \mathsf{snd}\ e \rrbracket_{\mathsf{a}} &:\equiv \llbracket e \rrbracket_{\mathsf{a}} \ggg_{\mathsf{a}} \mathsf{arr}_{\mathsf{a}}\ \mathsf{snd} \\
\llbracket v \rrbracket_{\mathsf{a}} &:\equiv \mathsf{arr}_{\mathsf{a}}\ (\mathsf{const}\ v)
\end{aligned}
\qquad
\begin{aligned}
\llbracket \mathsf{let}\ e\ e_b \rrbracket_{\mathsf{a}} &:\equiv (\llbracket e \rrbracket_{\mathsf{a}}\ \&\&\&_{\mathsf{a}}\ \mathsf{arr}_{\mathsf{a}}\ \mathsf{id}) \ggg_{\mathsf{a}} \llbracket e_b \rrbracket_{\mathsf{a}} \\
\llbracket \mathsf{env}\ 0 \rrbracket_{\mathsf{a}} &:\equiv \mathsf{arr}_{\mathsf{a}}\ \mathsf{fst} \\
\llbracket \mathsf{env}\ (n+1) \rrbracket_{\mathsf{a}} &:\equiv \mathsf{arr}_{\mathsf{a}}\ \mathsf{snd} \ggg_{\mathsf{a}} \llbracket \mathsf{env}\ n \rrbracket_{\mathsf{a}} \\
\llbracket \mathsf{if}\ e_c\ e_t\ e_f \rrbracket_{\mathsf{a}} &:\equiv \mathsf{ifte}_{\mathsf{a}}\ \llbracket e_c \rrbracket_{\mathsf{a}}\ \llbracket \mathsf{lazy}\ e_t \rrbracket_{\mathsf{a}}\ \llbracket \mathsf{lazy}\ e_f \rrbracket_{\mathsf{a}} \\
\llbracket \mathsf{lazy}\ e \rrbracket_{\mathsf{a}} &:\equiv \mathsf{lazy}_{\mathsf{a}}\ \lambda 0.\, \llbracket e \rrbracket_{\mathsf{a}}
\end{aligned}$$

$$\begin{aligned}
\mathsf{id} &:= \lambda \mathsf{a}.\, \mathsf{a} \\
\mathsf{const}\ \mathsf{b} &:= \lambda \mathsf{a}.\, \mathsf{b}
\end{aligned}
\qquad
\text{subject to } \llbracket p \rrbracket_{\mathsf{a}} : \langle\rangle \rightsquigarrow_{\mathsf{a}} \mathsf{y} \text{ for some } \mathsf{y}$$

Fig. 1: Interpretation of a let-calculus with first-order definitions and De-Bruijn-indexed bindings as arrow a computations.

by a final expression. The semantic function $\llbracket\cdot\rrbracket_{\mathsf{a}}$ transforms each defining expression and the final expression into arrow computations. Functions are named, but local variables and arguments are not. Instead, variables are referred to by De Bruijn indexes, with 0 referring to the innermost binding.

Perhaps unsurprisingly, interpretations act like stack machines. A final expression has type $\langle\rangle \rightsquigarrow_{\mathsf{a}} \mathsf{y}$, where y is the type of the program's value, and $\langle\rangle$ denotes an empty list, or stack. A let expression pushes a value onto the stack. First-order functions have type $\langle \mathsf{x}, \langle\rangle\rangle \rightsquigarrow_{\mathsf{a}} \mathsf{y}$ where x is the argument type and y is the return type. Application sends a stack containing just an x.

We generally regard programs as if they were their final expressions. Thus, the following definition applies to both programs and expressions.

Definition 5 (well-defined expression). *An expression e is **well-defined** under arrow a if $\llbracket e \rrbracket_{\mathsf{a}} : \mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}$ for some x and y, and $\llbracket e \rrbracket_{\mathsf{a}}$ terminates.*

From here on, we assume all expressions are well-defined. (The arrow a will be clear from context.) Well-definedness does not guarantee that *running* an interpretation terminates. It just simplifies statements about expressions, such as the following theorem, on which most of our semantic correctness results rely.

Theorem 2 (homomorphisms distribute over expressions). *Let $\mathsf{lift}_{\mathsf{b}} : (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{b}} \mathsf{y})$ be an arrow homomorphism. For all e, $\llbracket e \rrbracket_{\mathsf{b}} \equiv \mathsf{lift}_{\mathsf{b}}\ \llbracket e \rrbracket_{\mathsf{a}}$.*

If we assume $\mathsf{lift}_{\mathsf{b}}$ defines correct behavior for arrow b in terms of arrow a, and prove that $\mathsf{lift}_{\mathsf{b}}$ is a homomorphism, then by Theorem 2, $\llbracket\cdot\rrbracket_{\mathsf{b}}$ is correct.

3 The Bottom and Preimage Arrows

To use Theorem 2 to prove correct the interpretations of expressions as preimage arrow computations, we need the preimage arrow to be homomorphic to a simpler

$$X \rightsquigarrow_\bot Y \;::=\; X \Rightarrow Y_\bot$$

$$\mathsf{arr}_\bot : (X \Rightarrow Y) \Rightarrow (X \rightsquigarrow_\bot Y)$$
$$\mathsf{arr}_\bot\ f \;:=\; f$$

$$(\ggg_\bot) : (X \rightsquigarrow_\bot Y) \Rightarrow (Y \rightsquigarrow_\bot Z) \Rightarrow (X \rightsquigarrow_\bot Z)$$
$$(f_1 \ggg_\bot f_2)\ a \;:=\; \mathsf{if}\ (f_1\ a = \bot)\ \bot\ (f_2\ (f_1\ a))$$

$$(\&\&\&_\bot) : (X \rightsquigarrow_\bot Y_1) \Rightarrow (X \rightsquigarrow_\bot Y_2) \Rightarrow (X \rightsquigarrow_\bot \langle Y_1, Y_2\rangle)$$
$$(f_1\ \&\&\&_\bot\ f_2)\ a \;:=\; \mathsf{if}\ (f_1\ a = \bot\ \mathsf{or}\ f_2\ a = \bot)\ \bot\ \langle f_1\ a, f_2\ a\rangle$$

$$\begin{aligned}\mathsf{ifte}_\bot :\ & (X \rightsquigarrow_\bot \mathsf{Bool}) \Rightarrow (X \rightsquigarrow_\bot Y) \Rightarrow \\ & (X \rightsquigarrow_\bot Y) \Rightarrow (X \rightsquigarrow_\bot Y)\end{aligned}$$

$$\begin{aligned}&\mathsf{ifte}_\bot\ f_1\ f_2\ f_3\ a \;:=\\ &\quad \mathsf{case}\ \ f_1\ a\\ &\qquad \mathsf{true} \;\longrightarrow\; f_2\ a\\ &\qquad \mathsf{false} \;\longrightarrow\; f_3\ a\\ &\qquad \bot \;\;\;\;\longrightarrow\; \bot\end{aligned}$$

$$\mathsf{lazy}_\bot : (1 \Rightarrow (X \rightsquigarrow_\bot Y)) \Rightarrow (X \rightsquigarrow_\bot Y)$$
$$\mathsf{lazy}_\bot\ f\ a \;:=\; f\ 0\ a$$

Fig. 2: Bottom arrow definitions.

$$X \underset{\mathsf{pre}}{\rightharpoonup} Y ::= \langle \mathsf{Set}\ Y, \mathsf{Set}\ Y \Rightarrow \mathsf{Set}\ X\rangle$$

$$\mathsf{pre} : (X \rightsquigarrow_\bot Y) \Rightarrow (X \underset{\mathsf{pre}}{\rightharpoonup} Y)$$
$$\begin{aligned}&\mathsf{pre}\ f\ A \;:=\\ &\quad \langle \mathsf{image}_\bot\ f\ A, \lambda B.\, \mathsf{preimage}_\bot\ f\ A\ B\rangle\end{aligned}$$

$$\varnothing_{\mathsf{pre}} \;:=\; \langle \varnothing, \lambda B.\, \varnothing\rangle$$

$$\mathsf{ap}_{\mathsf{pre}} : (X \underset{\mathsf{pre}}{\rightharpoonup} Y) \Rightarrow \mathsf{Set}\ Y \Rightarrow \mathsf{Set}\ X$$
$$\mathsf{ap}_{\mathsf{pre}}\ \langle Y', p\rangle\ B \;:=\; p\ (B \cap Y')$$

$$\mathsf{range}_{\mathsf{pre}} : (X \underset{\mathsf{pre}}{\rightharpoonup} Y) \Rightarrow \mathsf{Set}\ Y$$
$$\mathsf{range}_{\mathsf{pre}}\ \langle Y', p\rangle \;:=\; Y'$$

$$\langle \cdot,\cdot\rangle_{\mathsf{pre}} : (X \underset{\mathsf{pre}}{\rightharpoonup} Y_1) \Rightarrow (X \underset{\mathsf{pre}}{\rightharpoonup} Y_2) \Rightarrow (X \underset{\mathsf{pre}}{\rightharpoonup} Y_1 \times Y_2)$$
$$\begin{aligned}&\langle\langle Y_1', p_1\rangle, \langle Y_2', p_2\rangle\rangle_{\mathsf{pre}} \;:=\\ &\quad \mathsf{let}\ \ Y' := Y_1' \times Y_2'\\ &\qquad\quad p := \lambda B.\ \bigcup_{\langle b_1, b_2\rangle \in B} (p_1\ \{b_1\}) \cap (p_2\ \{b_2\})\\ &\quad\ \ \mathsf{in}\ \ \langle Y', p\rangle\end{aligned}$$

$$(\circ_{\mathsf{pre}}) : (Y \underset{\mathsf{pre}}{\rightharpoonup} Z) \Rightarrow (X \underset{\mathsf{pre}}{\rightharpoonup} Y) \Rightarrow (X \underset{\mathsf{pre}}{\rightharpoonup} Z)$$
$$\langle Z', p_2\rangle \circ_{\mathsf{pre}} h_1 \;:=\; \langle Z', \lambda C.\, \mathsf{ap}_{\mathsf{pre}}\ h_1\ (p_2\ C)\rangle$$

$$(\uplus_{\mathsf{pre}}) : (X \underset{\mathsf{pre}}{\rightharpoonup} Y) \Rightarrow (X \underset{\mathsf{pre}}{\rightharpoonup} Y) \Rightarrow (X \underset{\mathsf{pre}}{\rightharpoonup} Y)$$
$$\begin{aligned}h_1 \uplus_{\mathsf{pre}} h_2 \;:=\; &\mathsf{let}\ \ Y' := (\mathsf{range}_{\mathsf{pre}}\ h_1) \cup (\mathsf{range}_{\mathsf{pre}}\ h_2)\\ &\qquad\ p := \lambda B.\, (\mathsf{ap}_{\mathsf{pre}}\ h_1\ B) \cup (\mathsf{ap}_{\mathsf{pre}}\ h_2\ B)\\ &\ \ \mathsf{in}\ \ \langle Y', p\rangle\end{aligned}$$

$$\mathsf{image}_\bot : (X \rightsquigarrow_\bot Y) \Rightarrow \mathsf{Set}\ X \Rightarrow \mathsf{Set}\ Y$$
$$\mathsf{image}_\bot\ f\ A \;:=\; (\mathsf{image}\ f\ A) \backslash \{\bot\}$$

$$\mathsf{preimage}_\bot : (X \rightsquigarrow_\bot Y) \Rightarrow \mathsf{Set}\ X \Rightarrow \mathsf{Set}\ Y \Rightarrow \mathsf{Set}\ X$$
$$\mathsf{preimage}_\bot\ f\ A\ B \;:=\; \{a \in A \mid f\ a \in B\}$$

Fig. 3: Lazy preimage mappings and operations.

arrow with easily understood behavior. The function arrow (6) is an obvious candidate. However, we will need to explicitly handle nontermination as an error value, so we need a slightly more complicated arrow.

Fig. 2 defines the ***bottom arrow***. Its computations have type $X \rightsquigarrow_\bot Y ::= X \Rightarrow Y_\bot$, where $Y_\bot ::= Y \cup \{\bot\}$ and $\bot$ is an error value.

To prove the arrow laws, we need a coarser notion of equivalence.

Definition 6 (bottom arrow equivalence). *Two computations* $f_1 : X \rightsquigarrow_\bot Y$ *and* $f_2 : X \rightsquigarrow_\bot Y$ *are equivalent, or* $f_1 \equiv f_2$*, when* $f_1\ a \equiv f_2\ a$ *for all* $a \in X$*.*

Using bottom arrow equivalence, it is easy to show that $(\rightsquigarrow_\bot)$ is epimorphic to the `Maybe` monad's Kleisli arrow. By Theorem 1, the arrow laws hold.

3.1 Lazy Preimage Mappings

Approximation is smoother if we have an abstraction that hides infinite computations. Therefore, we confine operations on sets to instances of

$$X \underset{\mathsf{pre}}{\rightharpoonup} Y \;::=\; \langle \mathsf{Set}\ Y, \mathsf{Set}\ Y \Rightarrow \mathsf{Set}\ X\rangle \tag{14}$$

$$\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y} ::= \mathsf{Set\ X} \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y})$$

$$\mathsf{arr_{pre}} : (\mathsf{X} \Rightarrow \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y})$$
$$\mathsf{arr_{pre}} \ := \ \mathsf{lift_{pre}} \circ \mathsf{arr_{\bot}}$$

$$(\ggg_{\mathsf{pre}}) : (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}) \Rightarrow (\mathsf{Y} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Z}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Z})$$
$$\begin{aligned}(\mathsf{h_1} \ggg_{\mathsf{pre}} \mathsf{h_2})\ \mathsf{A} \ := \ \mathsf{let}\ & \mathsf{h_1'} := \mathsf{h_1\ A} \\ & \mathsf{h_2'} := \mathsf{h_2}\ (\mathsf{range_{pre}\ h_1'}) \\ \mathsf{in}\ & \mathsf{h_2'} \circ_{\mathsf{pre}} \mathsf{h_1'}\end{aligned}$$

$$(\&\&\&_{\mathsf{pre}}) : (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Z}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y} \times \mathsf{Z})$$
$$(\mathsf{h_1} \ \&\&\&_{\mathsf{pre}}\ \mathsf{h_2})\ \mathsf{A} \ := \ \langle \mathsf{h_1\ A}, \mathsf{h_2\ A} \rangle_{\mathsf{pre}}$$

$$\begin{aligned}\mathsf{ifte_{pre}} : {} & (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Bool}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}) \Rightarrow \\ & (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y})\end{aligned}$$
$$\begin{aligned}& \mathsf{ifte_{pre}\ h_1\ h_2\ h_3\ A} \ := \\ & \quad \mathsf{let}\ \mathsf{h_1'} := \mathsf{h_1\ A} \\ & \qquad\ \ \mathsf{h_2'} := \mathsf{h_2}\ (\mathsf{ap_{pre}\ h_1'}\ \{\mathsf{true}\}) \\ & \qquad\ \ \mathsf{h_3'} := \mathsf{h_3}\ (\mathsf{ap_{pre}\ h_1'}\ \{\mathsf{false}\}) \\ & \quad\ \ \mathsf{in}\ \mathsf{h_2'} \uplus_{\mathsf{pre}} \mathsf{h_3'}\end{aligned}$$

$$\mathsf{lazy_{pre}} : (1 \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y})) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y})$$
$$\mathsf{lazy_{pre}\ h\ A} \ := \ \mathsf{if}\ (\mathsf{A} = \varnothing)\ \varnothing_{\mathsf{pre}}\ (\mathsf{h}\ 0\ \mathsf{A})$$

$$\mathsf{lift_{pre}} \ := \ \mathsf{pre}$$

Fig. 4: Preimage arrow definitions.

Like a mapping, an $\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y}$ has an observable domain—which preimage arrow composition will need further on—but computing the input-output pairs is delayed. We therefore call these ***lazy preimage mappings***.

Converting a bottom arrow computation to a lazy preimage mapping requires computing its range and constructing a delayed preimage computation:

$$\begin{aligned}&\mathsf{pre} : (\mathsf{X} \rightsquigarrow_{\bot} \mathsf{Y}) \Rightarrow \mathsf{Set\ X} \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y}) \\ &\mathsf{pre\ f\ A} \ := \ \langle \mathsf{image_{\bot}\ f\ A}, \lambda \mathsf{B}.\, \mathsf{preimage_{\bot}\ f\ A\ B} \rangle\end{aligned} \tag{15}$$

Fig. 3 defines $\mathsf{image_{\bot}}$, $\mathsf{preimage_{\bot}}$, and operations on preimage mappings: $\langle \cdot, \cdot \rangle_{\mathsf{pre}}$ returns preimage mappings that compute preimages under pairing, and $(\circ_{\mathsf{pre}})$ and $(\uplus_{\mathsf{pre}})$ do the same for compositions and for unions of functions with disjoint domains. The $\mathsf{ap_{pre}}$ function applies a preimage mapping to a $\mathsf{Set\ Y}$.

Preimage arrow correctness depends on $\mathsf{ap_{pre}}$ and pre behaving like $\mathsf{preimage_{\bot}}$.

Theorem 3 ($\mathsf{ap_{pre}}$ of pre computes preimages). *Let* $\mathsf{f} : \mathsf{X} \rightsquigarrow_{\bot} \mathsf{Y}$*. For all* $\mathsf{A} \subseteq \mathsf{X}$ *and* $\mathsf{B} \subseteq \mathsf{Y}$*,* $\mathsf{ap_{pre}}\ (\mathsf{pre\ f\ A})\ \mathsf{B} \equiv \mathsf{preimage_{\bot}\ f\ A\ B}$*.*

3.2 The Preimage Arrow

If we define the ***preimage arrow*** type constructor as

$$\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y} \ ::= \ \mathsf{Set\ X} \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y}) \tag{16}$$

then we already have a lift $\mathsf{lift_{pre}} : (\mathsf{X} \rightsquigarrow_{\bot} \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y})$ from the bottom arrow to the preimage arrow: pre. By Theorem 3, lifted bottom arrow computations compute correct preimages, exactly as we should expect them to.

Fig. 4 defines the preimage arrow in terms of preimage mapping operations (Fig. 3). For these definitions to make $\mathsf{lift_{pre}}$ a homomorphism, we need preimage arrow equivalence to mean "computes the same preimages."

Definition 7 (preimage arrow equivalence). *Two preimage arrow computations* $\mathsf{h_1} : \mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}$ *and* $\mathsf{h_2} : \mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}$ *are equivalent, or* $\mathsf{h_1} \equiv \mathsf{h_2}$*, when* $\mathsf{ap_{pre}}\ (\mathsf{h_1\ A})\ \mathsf{B} \equiv \mathsf{ap_{pre}}\ (\mathsf{h_2\ A})\ \mathsf{B}$ *for all* $\mathsf{A} \subseteq \mathsf{X}$ *and* $\mathsf{B} \subseteq \mathsf{Y}$*.*

Theorem 4 (preimage arrow correctness). $\mathsf{lift}_{\mathsf{pre}}$ *is a homomorphism.*

Corollary 1 (semantic correctness). *For all e,* $[\![e]\!]_{\mathsf{pre}} \equiv \mathsf{lift}_{\mathsf{pre}}\ [\![e]\!]_{\bot}$.

The type $\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}$ does not constrain its inhabitants to behave intuitively; e.g.

$$\begin{aligned} \mathsf{unruly} &: \mathsf{Bool} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Bool} \\ \mathsf{unruly}\ \mathsf{A}\ &:=\ \langle \mathsf{Bool}\backslash\mathsf{A}, \lambda\,\mathsf{B}.\,\mathsf{B}\rangle \end{aligned} \tag{17}$$

So $\mathsf{ap}_{\mathsf{pre}}\ (\mathsf{unruly}\ \{\mathsf{true}\})\ \{\mathsf{false}\} = \{\mathsf{false}\} \cap (\mathsf{Bool}\backslash\{\mathsf{true}\}) = \{\mathsf{false}\}$—a "preimage" that does not even intersect the given domain $\{\mathsf{true}\}$. Other examples show that preimage computations are not necessarily monotone, and lack other desirable properties. Those with desirable properties obey the following law.

Definition 8 (preimage arrow law). *Let* $\mathsf{h} : \mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}$*. If there exists an* $\mathsf{f} : \mathsf{X} \rightsquigarrow_{\bot} \mathsf{Y}$ *such that* $\mathsf{h} \equiv \mathsf{lift}_{\mathsf{pre}}\ \mathsf{f}$*, then* h *obeys the* ***preimage arrow law****.*

By homomorphism of $\mathsf{lift}_{\mathsf{pre}}$, preimage arrow combinators preserve the preimage arrow law. From here on, we assume all $\mathsf{h} : \mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}$ obey it. By Definition 8, $\mathsf{lift}_{\mathsf{pre}}$ is an epimorphism; by Theorem 1, the arrow laws hold.

4 The Bottom* and Preimage* Arrows

We have defined the top of our roadmap:

$$\begin{tikzcd} \mathsf{X} \rightsquigarrow_{\bot} \mathsf{Y} \arrow[r, "\mathsf{lift}_{\mathsf{pre}}"] \arrow[d, "\eta_{\bot^*}"'] & \mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y} \arrow[d, "\eta_{\mathsf{pre}^*}"] \\ \mathsf{X} \rightsquigarrow_{\bot^*} \mathsf{Y} \arrow[r, "\mathsf{lift}_{\mathsf{pre}^*}"'] & \mathsf{X} \underset{\mathsf{pre}^*}{\rightsquigarrow} \mathsf{Y} \end{tikzcd} \tag{18}$$

so that $\mathsf{lift}_{\mathsf{pre}}$ is a homomorphism. Now we move down each side and connect the bottom, in a way that makes every morphism a homomorphism.

Probabilistic functions that may not terminate, but do so with probability 1, are common. For example, suppose random retrieves numbers in $[0, 1]$ from an implicit random source. The following probabilistic function defines the well-known geometric distribution by counting the number of times $\mathsf{random} < \mathsf{p}$:

$$\mathsf{geometric}\ \mathsf{p}\ :=\ \mathsf{if}\ (\mathsf{random} < \mathsf{p})\ 0\ (1 + \mathsf{geometric}\ \mathsf{p}) \tag{19}$$

For any $\mathsf{p} > 0$, $\mathsf{geometric}\ \mathsf{p}$ may not terminate, but the probability of never taking the "then" branch is $1 - (1 - \mathsf{p}) \cdot (1 - \mathsf{p}) \cdot (1 - \mathsf{p}) \cdot \dots = 1$.

Suppose we interpret $\mathsf{geometric}\ \mathsf{p}$ as $\mathsf{h} : \mathsf{R} \underset{\mathsf{pre}}{\rightsquigarrow} \mathbb{N}$, a preimage arrow computation from random sources to naturals, and we have a probability measure $\mathsf{P} : \mathsf{Set}\ \mathsf{R} \Rightarrow [0, 1]$. The probability of $\mathsf{N} \subseteq \mathbb{N}$ is $\mathsf{P}\ (\mathsf{ap}_{\mathsf{pre}}\ (\mathsf{h}\ \mathsf{R})\ \mathsf{N})$. To compute this, we must

- Ensure $\mathsf{ap}_{\mathsf{pre}}\ (\mathsf{h}\ \mathsf{R})\ \mathsf{N}$ terminates.
- Ensure each $\mathsf{r} \in \mathsf{R}$ contains enough random numbers.
- Determine how random indexes numbers in r.

Ensuring $\mathsf{ap}_{\mathsf{pre}}\ (\mathsf{h}\ \mathsf{R})\ \mathsf{N}$ terminates is the most difficult, but doing the other two will provide structure that makes it much easier.

$$\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y} \;::=\; \mathsf{AStore}\ \mathsf{s}\ (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \;::=\; \mathsf{J} \Rightarrow (\langle \mathsf{s},\mathsf{x}\rangle \rightsquigarrow_{\mathsf{a}} \mathsf{y})$$

$$\begin{array}{l}
\mathsf{arr}_{\mathsf{a}^*} : (\mathsf{x} \Rightarrow \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y}) \\
\mathsf{arr}_{\mathsf{a}^*} \;:=\; \eta_{\mathsf{a}^*} \circ \mathsf{arr}_{\mathsf{a}}
\end{array}$$

$$\begin{array}{l}
(\ggg_{\mathsf{a}^*}) : (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y}) \Rightarrow (\mathsf{y} \rightsquigarrow_{\mathsf{a}^*} \mathsf{z}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{z}) \\
(\mathsf{k}_1 \ggg_{\mathsf{a}^*} \mathsf{k}_2)\ \mathsf{j} \;:= \\
\quad (\mathsf{arr}_{\mathsf{a}}\ \mathsf{fst} \mathbin{\&\&\&_{\mathsf{a}}} \mathsf{k}_1\ (\mathsf{left}\ \mathsf{j})) \ggg_{\mathsf{a}} \mathsf{k}_2\ (\mathsf{right}\ \mathsf{j})
\end{array}$$

$$\begin{array}{l}
(\&\&\&_{\mathsf{a}^*}) : (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y}_1) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y}_2) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \langle \mathsf{y}_1, \mathsf{y}_2\rangle) \\
(\mathsf{k}_1 \mathbin{\&\&\&_{\mathsf{a}^*}} \mathsf{k}_2)\ \mathsf{j} \;:=\; \mathsf{k}_1\ (\mathsf{left}\ \mathsf{j}) \mathbin{\&\&\&_{\mathsf{a}}} \mathsf{k}_2\ (\mathsf{right}\ \mathsf{j})
\end{array}$$

$$\begin{array}{l}
\mathsf{ifte}_{\mathsf{a}^*} : (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{Bool}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y}) \Rightarrow \\
\qquad (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y}) \\
\mathsf{ifte}_{\mathsf{a}^*}\ \mathsf{k}_1\ \mathsf{k}_2\ \mathsf{k}_3\ \mathsf{j} \;:= \\
\quad \mathsf{ifte}_{\mathsf{a}}\ (\mathsf{k}_1\ (\mathsf{left}\ \mathsf{j})) \\
\qquad\quad (\mathsf{k}_2\ (\mathsf{left}\ (\mathsf{right}\ \mathsf{j}))) \\
\qquad\quad (\mathsf{k}_3\ (\mathsf{right}\ (\mathsf{right}\ \mathsf{j})))
\end{array}$$

$$\begin{array}{l}
\mathsf{lazy}_{\mathsf{a}^*} : (1 \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y})) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y}) \\
\mathsf{lazy}_{\mathsf{a}^*}\ \mathsf{k}\ \mathsf{j} \;:=\; \mathsf{lazy}_{\mathsf{a}}\ \lambda 0.\, \mathsf{k}\ 0\ \mathsf{j}
\end{array}$$

$$\begin{array}{l}
\eta_{\mathsf{a}^*} : (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y}) \\
\eta_{\mathsf{a}^*}\ \mathsf{f}\ \mathsf{j} \;:=\; \mathsf{arr}_{\mathsf{a}}\ \mathsf{snd} \ggg_{\mathsf{a}} \mathsf{f}
\end{array}$$

Fig. 5: AStore (associative store) arrow transformer definitions.

4.1 Threading and Indexing

We clearly need bottom and preimage arrows that thread a random source. To ensure random sources contain enough numbers, they should be infinite.

In a pure λ-calculus, random sources are typically infinite streams, threaded monadically: each computation receives and produces a random source. A little-used alternative is for the random source to be a tree, threaded applicatively: each computation receives, but does not produce, a random source. Combinators split the tree and pass subtrees to subcomputations.

With either alternative, for arrows, the resulting definitions are large, conceptually difficult, and hard to manipulate. Fortunately, it is relatively easy to assign each subcomputation a unique index into a tree-shaped random source and pass the random source unchanged. To do this, we need an indexing scheme.

Definition 9 (binary indexing scheme). *Let* J *be an index set,* $\mathsf{j}_0 \in \mathsf{J}$ *a distinguished element, and* $\mathsf{left} : \mathsf{J} \Rightarrow \mathsf{J}$ *and* $\mathsf{right} : \mathsf{J} \Rightarrow \mathsf{J}$ *be total, injective functions. If for all* $\mathsf{j} \in \mathsf{J}$, $\mathsf{j} = \mathsf{next}\ \mathsf{j}_0$ *for some finite composition* next *of* left *and* right*, then* J, j_0*,* left *and* right *define a* ***binary indexing scheme****.*

For example, let J be the set of lists of $\{0,1\}$, $\mathsf{j}_0 := \langle\rangle$, and $\mathsf{left}\ \mathsf{j} := \langle 0, \mathsf{j}\rangle$ and $\mathsf{right}\ \mathsf{j} := \langle 1, \mathsf{j}\rangle$. In any case, J is countable, and can be thought of as a set of indexes into an infinite binary tree. Values of type $\mathsf{J} \to \mathsf{A}$ encode an infinite binary tree of A values as an infinite vector (i.e. total mapping).

We thread infinite binary trees through bottom and preimage arrow computations by defining an **arrow transformer**: a type constructor that receives and produces an arrow type, and combinators for arrows of the produced type. The AStore arrow type constructor takes a store type s and an arrow $\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}$:

$$\mathsf{AStore}\ \mathsf{s}\ (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \;::=\; \mathsf{J} \Rightarrow (\langle \mathsf{s},\mathsf{x}\rangle \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \tag{20}$$

Reading the type, we see that computations receive an index $\mathsf{j} \in \mathsf{J}$ and produce a computation that receives a store as well as an x. Lifting extracts the x from the input pair and sends it on to the original computation, ignoring j:

$$\begin{array}{l}
\eta_{\mathsf{a}^*} : (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \Rightarrow \mathsf{AStore}\ \mathsf{s}\ (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \\
\eta_{\mathsf{a}^*}\ \mathsf{f}\ \mathsf{j} \;:=\; \mathsf{arr}_{\mathsf{a}}\ \mathsf{snd} \ggg_{\mathsf{a}} \mathsf{f}
\end{array} \tag{21}$$

Fig. 5 defines the remaining combinators. Each subcomputation receives $\mathsf{left\ j}$, $\mathsf{right\ j}$, or some other unique binary index. We thus think of programs interpreted as AStore arrows as being completely unrolled into an infinite binary tree, with each expression labeled with its tree index.

4.2 Partial, Probabilistic Programs

To interpret probabilitic programs, we put an infinite random tree in the store.

Definition 10 (random source). *Let* $\mathsf{R} := \mathsf{J} \to [0,1]$. *A* ***random source*** *is any infinite binary tree* $\mathsf{r} \in \mathsf{R}$.

To interpret partial programs, we need to ensure termination. An utimately implementable way is to have the store dictate which branch of each conditional, if any, is taken.

Definition 11 (branch trace). *A* ***branch trace*** *is any* $\mathsf{t} \in \mathsf{J} \to \mathsf{Bool}_\perp$ *such that* $\mathsf{t\ j} = \mathsf{true}$ *or* $\mathsf{t\ j} = \mathsf{false}$ *for no more than finitely many* $\mathsf{j} \in \mathsf{J}$.

Let $\mathsf{T} \subset \mathsf{J} \to \mathsf{Bool}_\perp$ *be the largest set of branch traces.*

Let $\mathsf{X} \rightsquigarrow_{\mathsf{a}^*} \mathsf{Y} ::= \mathsf{AStore}\ (\mathsf{R} \times \mathsf{T})\ (\mathsf{X} \rightsquigarrow_{\mathsf{a}} \mathsf{Y})$ be an AStore arrow type that threads both random stores and branch traces.

For probabilistic programs, we define a combinator $\mathsf{random}_{\mathsf{a}^*}$ that returns the number at its tree index in the random source, and extend $[\![\cdot]\!]_{\mathsf{a}^*}$ for arrows a^* for which $\mathsf{random}_{\mathsf{a}^*}$ is defined:

$$\begin{aligned} &\mathsf{random}_{\mathsf{a}^*} : \mathsf{X} \rightsquigarrow_{\mathsf{a}^*} [0,1] \\ &\mathsf{random}_{\mathsf{a}^*}\ \mathsf{j}\ :=\ \mathsf{arr}_{\mathsf{a}}\ (\mathsf{fst} \ggg \mathsf{fst} \ggg \pi\ \mathsf{j}) \end{aligned} \qquad [\![\mathsf{random}]\!]_{\mathsf{a}^*}\ :\equiv\ \mathsf{random}_{\mathsf{a}^*} \tag{22}$$

For partial programs, we define a combinator that reads branch traces, and an if-then-else combinator that ensures its test expression agrees with the trace:

$$\begin{aligned} &\mathsf{branch}_{\mathsf{a}^*} : \mathsf{X} \rightsquigarrow_{\mathsf{a}^*} \mathsf{Bool} \\ &\mathsf{branch}_{\mathsf{a}^*}\ \mathsf{j}\ :=\ \mathsf{arr}_{\mathsf{a}}\ (\mathsf{fst} \ggg \mathsf{snd} \ggg \pi\ \mathsf{j}) \end{aligned}$$

$$\begin{aligned} &\mathsf{ifte}^{\Downarrow}_{\mathsf{a}^*} : (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{Bool}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y}) \\ &\mathsf{ifte}^{\Downarrow}_{\mathsf{a}^*}\ \mathsf{k}_1\ \mathsf{k}_2\ \mathsf{k}_3\ \mathsf{j}\ :=\ \mathsf{ifte}_{\mathsf{a}}\ ((\mathsf{k}_1\ (\mathsf{left\ j})\ \mathbin{\&\&\&}_{\mathsf{a}}\ \mathsf{branch}_{\mathsf{a}^*}\ \mathsf{j}) \ggg_{\mathsf{a}} \mathsf{arr}_{\mathsf{a}}\ \mathsf{agrees}) \\ &\qquad\qquad\qquad\qquad\qquad (\mathsf{k}_2\ (\mathsf{left\ (right\ j)})) \\ &\qquad\qquad\qquad\qquad\qquad (\mathsf{k}_3\ (\mathsf{right\ (right\ j)})) \end{aligned} \tag{23}$$

where $\mathsf{agrees}\ \langle \mathsf{b}_1, \mathsf{b}_2 \rangle := \mathsf{if}\ (\mathsf{b}_1 = \mathsf{b}_2)\ \mathsf{b}_1\ \perp$. Thus, if the branch trace does not agree with the test expression, it returns an error. We define a new semantic function $[\![\cdot]\!]^{\Downarrow}_{\mathsf{a}^*}$ by replacing the if rule in $[\![\cdot]\!]_{\mathsf{a}^*}$:

$$[\![\mathsf{if}\ e_c\ e_t\ e_f]\!]^{\Downarrow}_{\mathsf{a}^*}\ :\equiv\ \mathsf{ifte}^{\Downarrow}_{\mathsf{a}^*}\ [\![e_c]\!]^{\Downarrow}_{\mathsf{a}^*}\ [\![\mathsf{lazy}\ e_t]\!]^{\Downarrow}_{\mathsf{a}^*}\ [\![\mathsf{lazy}\ e_f]\!]^{\Downarrow}_{\mathsf{a}^*} \tag{24}$$

For an AStore computation k, we obviously must run k on every branch trace in T and filter out $\perp$, or somehow find inputs $\langle\langle \mathsf{r}, \mathsf{t} \rangle, \mathsf{a}\rangle$ for which agrees never returns $\perp$. Preimage AStore arrows do the former by first computing an image, and the latter by computing preimages of sets that cannot contain $\perp$.

Definition 12 (terminating, probabilistic arrows). *Define*

$$\begin{aligned} \mathsf{X} \rightsquigarrow_{\bot^*} \mathsf{Y} \;&::=\; \mathsf{AStore}\ (\mathsf{R} \times \mathsf{T})\ (\mathsf{X} \rightsquigarrow_{\bot} \mathsf{Y}) \\ \mathsf{X} \underset{\mathsf{pre}^*}{\rightsquigarrow} \mathsf{Y} \;&::=\; \mathsf{AStore}\ (\mathsf{R} \times \mathsf{T})\ (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}) \end{aligned} \tag{25}$$

as the type constructors for the ***bottom**** *and* ***preimage* arrows****.*

Suppose $\mathsf{f} := \llbracket e \rrbracket^{\Downarrow}_{\bot^*} : \mathsf{X} \rightsquigarrow_{\bot^*} \mathsf{Y}$. Its domain is $\mathsf{X}' := (\mathsf{R} \times \mathsf{T}) \times \mathsf{X}$. We assume each $\mathsf{r} \in \mathsf{R}$ is random, but not $\mathsf{t} \in \mathsf{T}$ nor $\mathsf{a} \in \mathsf{X}$; therefore, neither T nor X should affect the probabilities of output sets. The probability of $\mathsf{B} \subseteq \mathsf{Y}$ is therefore

$$\begin{aligned} &\mathsf{P}\ (\mathsf{image}\ (\mathsf{fst} \ggg \mathsf{fst})\ (\mathsf{preimage}_{\bot}\ \mathsf{f}\ \mathsf{X}'\ \mathsf{B})) \\ &\quad = \ \mathsf{P}\ (\mathsf{image}\ (\mathsf{fst} \ggg \mathsf{fst})\ (\mathsf{ap}_{\mathsf{pre}}\ (\mathsf{h}\ \mathsf{X}')\ \mathsf{B})) \end{aligned} \tag{26}$$

where $\mathsf{h} := \llbracket e \rrbracket^{\Downarrow}_{\mathsf{pre}^*}$, if f and h always terminate and $\llbracket \cdot \rrbracket^{\Downarrow}_{\mathsf{pre}^*}$ is correct.

4.3 Correctness and Termination

For correctness, we have two arrow lifts to prove homomorphic: one from pure computations to effectful, and one from effectful computations to effectful. For both, we need AStore arrow equivalence to be more extensional.

Definition 13 (AStore arrow equivalence). *Two* AStore *arrow computations* k_1 *and* k_2 *are equivalent, or* $\mathsf{k}_1 \equiv \mathsf{k}_2$*, when* $\mathsf{k}_1\ \mathsf{j} \equiv \mathsf{k}_2\ \mathsf{j}$ *for all* $\mathsf{j} \in \mathsf{J}$*.*

Theorem 5 (pure AStore arrow correctness). η_{a^*} *is a homomorphism.*

Corollary 2 (pure semantic correctness). *For all pure e,* $\llbracket e \rrbracket_{\mathsf{a}^*} \equiv \eta_{\mathsf{a}^*}\ \llbracket e \rrbracket_{\mathsf{a}}$*.*

We need a lift between AStore arrows. Let $\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y} ::= \mathsf{AStore}\ \mathsf{s}\ (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y})$, $\mathsf{x} \rightsquigarrow_{\mathsf{b}^*} \mathsf{y} ::= \mathsf{AStore}\ \mathsf{s}\ (\mathsf{x} \rightsquigarrow_{\mathsf{b}} \mathsf{y})$, and $\mathsf{lift}_{\mathsf{b}} : (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{b}} \mathsf{y})$. Define

$$\begin{aligned} &\mathsf{lift}_{\mathsf{b}^*} : (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{b}^*} \mathsf{y}) \\ &\mathsf{lift}_{\mathsf{b}^*}\ \mathsf{f}\ \mathsf{j} \ := \ \mathsf{lift}_{\mathsf{b}}\ (\mathsf{f}\ \mathsf{j}) \end{aligned} \tag{27}$$

Theorem 6 (effectful AStore arrow correctness). *If* $\mathsf{lift}_{\mathsf{b}}$ *is an arrow homomorphism from* a *to* b*, then* $\mathsf{lift}_{\mathsf{b}^*}$ *is an arrow homomorphism from* a^* *to* b^**.*

Corollary 3 (preimage* arrow correctness). $\mathsf{lift}_{\mathsf{pre}^*}$ *is a homomorphism.*

Corollary 4 (effectful semantic correctness). *For all expressions e,* $\llbracket e \rrbracket_{\mathsf{pre}^*} \equiv \mathsf{lift}_{\mathsf{pre}^*}\ \llbracket e \rrbracket_{\bot^*}$ *and* $\llbracket e \rrbracket^{\Downarrow}_{\mathsf{pre}^*} \equiv \mathsf{lift}_{\mathsf{pre}^*}\ \llbracket e \rrbracket^{\Downarrow}_{\bot^*}$*.*

For termination, we need to define the largest domain on which $\llbracket e \rrbracket^{\Downarrow}_{\mathsf{a}^*}$ and $\llbracket e \rrbracket_{\mathsf{a}^*}$ computations should agree.

Definition 14 (maximal domain). *Let* $\mathsf{f} : \mathsf{X} \rightsquigarrow_{\bot^*} \mathsf{Y}$*. Its* ***maximal domain*** *is the largest* $\mathsf{A}^* \subseteq (\mathsf{R} \times \mathsf{T}) \times \mathsf{X}$ *for which* $\mathsf{A}^* = \{\mathsf{a} \in \mathsf{A}^* \mid \mathsf{f}\ \mathsf{j}_0\ \mathsf{a} \neq \bot\}$*.*

Because $\mathsf{f}\ \mathsf{j}_0\ \mathsf{a} \neq \bot$ implies termination, all inputs in A^* are terminating.

$$\begin{array}{rcl} \mathsf{id_{pre}\ A} &:=& \langle \mathsf{A}, \lambda \mathsf{B}.\, \mathsf{B} \rangle \\ \mathsf{fst_{pre}\ A} &:=& \langle \mathsf{proj_1\ A}, \mathsf{unproj_1\ A} \rangle \\ \mathsf{snd_{pre}\ A} &:=& \langle \mathsf{proj_2\ A}, \mathsf{unproj_2\ A} \rangle \end{array}$$

$$\begin{array}{l} \mathsf{proj_1} := \mathsf{image\ fst} \\ \mathsf{proj_2} := \mathsf{image\ snd} \\ \mathsf{unproj_1\ A\ B} \ := \ \mathsf{A} \cap (\mathsf{B} \times \mathsf{proj_2\ A}) \\ \mathsf{unproj_2\ A\ B} \ := \ \mathsf{A} \cap (\mathsf{proj_1\ A} \times \mathsf{B}) \end{array}$$

$$\begin{array}{rcl} \mathsf{const_{pre}\ b\ A} &:=& \langle \{\mathsf{b}\}, \lambda \mathsf{B}.\, \mathsf{if}\ (\mathsf{B} = \varnothing)\ \varnothing\ \mathsf{A} \rangle \\ \pi_{\mathsf{pre}}\ \mathsf{j\ A} &:=& \langle \mathsf{proj\ j\ A}, \mathsf{unproj\ j\ A} \rangle \end{array}$$

$$\begin{array}{l} \mathsf{proj} : \mathsf{J} \Rightarrow \mathsf{Set}\ (\mathsf{J} \to \mathsf{X}) \Rightarrow \mathsf{Set\ X} \\ \mathsf{proj\ j\ A} \ := \ \mathsf{image}\ (\pi\ \mathsf{j})\ \mathsf{A} \\ \\ \mathsf{unproj} : \mathsf{J} \Rightarrow \mathsf{Set}\ (\mathsf{J} \to \mathsf{X}) \Rightarrow \mathsf{Set\ X} \Rightarrow \mathsf{Set}\ (\mathsf{J} \to \mathsf{X}) \\ \mathsf{unproj\ j\ A\ B} \ := \ \mathsf{A} \cap \prod_{\mathsf{i} \in \mathsf{J}} \mathsf{if}\ (\mathsf{j} = \mathsf{i})\ \mathsf{B}\ (\mathsf{proj\ j\ A}) \end{array}$$

Fig. 6: Preimage arrow lifts needed to interpret probabilistic programs.

Theorem 7 (correct termination everywhere). *Let* $[\![e]\!]^{\Downarrow}_{\bot^*} : \mathsf{X} \rightsquigarrow_{\bot^*} \mathsf{Y}$ *have maximal domain* A^**, and* $\mathsf{X}' := (\mathsf{R} \times \mathsf{T}) \times \mathsf{X}$*. For all* $\mathsf{a} \in \mathsf{X}'$*,* $\mathsf{A} \subseteq \mathsf{X}'$ *and* $\mathsf{B} \subseteq \mathsf{Y}$*,*

$$\begin{array}{rcl} [\![e]\!]^{\Downarrow}_{\bot^*}\ \mathsf{j_0\ a} &=& \mathsf{if}\ (\mathsf{a} \in \mathsf{A}^*)\ ([\![e]\!]_{\bot^*}\ \mathsf{j_0\ a})\ \bot \\ \mathsf{ap_{pre}}\ ([\![e]\!]^{\Downarrow}_{\mathsf{pre}^*}\ \mathsf{j_0\ A})\ \mathsf{B} &=& \mathsf{ap_{pre}}\ ([\![e]\!]_{\mathsf{pre}^*}\ \mathsf{j_0}\ (\mathsf{A} \cap \mathsf{A}^*))\ \mathsf{B} \end{array} \tag{28}$$

In other words, preimages computed using $[\![\cdot]\!]^{\Downarrow}_{\mathsf{pre}^*}$ always terminate, never include inputs that give rise to errors or nontermination, and are correct.

5 Approximating Semantics

We would like to compute preimages of uncountable sets, such as real intervals—but $\mathsf{preimage_{\bot}\ f\ A\ B}$ is uncomputable for most uncountable sets A and B no matter how cleverly they are represented. Further, because pre, $\mathsf{lift_{pre}}$ and $\mathsf{arr_{pre}}$ are defined in terms of $\mathsf{preimage_{\bot}}$, we cannot implement them.

Fortunately, we need only certain lifts. Fig. 6 gives explicit definitions for $\mathsf{arr_{pre}\ id}$, $\mathsf{arr_{pre}\ fst}$, $\mathsf{arr_{pre}\ snd}$, $\mathsf{arr_{pre}}\ (\mathsf{const\ b})$ and $\mathsf{arr_{pre}}\ (\pi\ \mathsf{j})$. To implement them, we must model sets in a way that makes $\mathsf{A} = \varnothing$ is decidable, and the following representable and finitely computable:

- $\mathsf{A} \cap \mathsf{B}$, $\varnothing$, $\{\mathsf{true}\}$, $\{\mathsf{false}\}$ and $\{\mathsf{b}\}$ for every $\mathsf{const\ b}$
- $\mathsf{A_1} \times \mathsf{A_2}$, $\mathsf{proj_1\ A}$ and $\mathsf{proj_2\ A}$ (29)
- $\mathsf{J} \to \mathsf{X}$, $\mathsf{proj\ j\ A}$ and $\mathsf{unproj\ j\ A\ B}$

We first need to define families of sets under which these operations are closed.

Definition 15 (rectangular family). $\mathsf{Rect\ X}$ *denotes the* ***rectangular family*** *of subsets of* X*.* $\mathsf{Rect\ X}$ *must contain* $\varnothing$ *and* X*, and be closed under finite intersections. Products must satisfy the following rules:*

$$\mathsf{Rect}\ \langle \mathsf{X_1}, \mathsf{X_2} \rangle \ = \ (\mathsf{Rect\ X_1}) \boxtimes (\mathsf{Rect\ X_2}) \tag{30}$$

$$\mathsf{Rect}\ (\mathsf{J} \to \mathsf{X}) \ = \ (\mathsf{Rect\ X})^{\boxtimes \mathsf{J}} \tag{31}$$

where the following operations lift cartesian products to sets of sets:

$$\mathcal{A}_1 \boxtimes \mathcal{A}_2 \ := \ \{\mathsf{A_1} \times \mathsf{A_2} \mid \mathsf{A_1} \in \mathcal{A}_1, \mathsf{A_2} \in \mathcal{A}_2\} \tag{32}$$

$$\mathcal{A}^{\boxtimes \mathsf{J}} \ := \ \bigcup_{\mathsf{J}' \subset \mathsf{J}\ \textit{finite}} \left\{ \textstyle\prod_{\mathsf{j} \in \mathsf{J}} \mathsf{A_j} \,\middle|\, \mathsf{A_j} \in \mathcal{A}, \mathsf{j} \in \mathsf{J}' \iff \mathsf{A_j} \subset \bigcup \mathcal{A} \right\} \tag{33}$$

We additionally define $\mathsf{Rect\ Bool} ::= \mathcal{P}\ \mathsf{Bool}$. It is easy to show the collection of all rectangular families is closed under products, projections, and unproj.

Further, all of the operations in (29) can be exactly implemented if finite sets are modeled directly, sets in an ordered space (such as $\mathbb{R}$) are modeled by intervals, and sets in $\mathsf{Rect}\ \langle \mathsf{X}_1, \mathsf{X}_2 \rangle$ are modeled by pairs of type $\langle \mathsf{Rect\ X}_1, \mathsf{Rect\ X}_2 \rangle$. By (33), sets in $\mathsf{Rect}\ (\mathsf{J} \to \mathsf{X})$ have no more than finitely many projections that are proper subsets of X. They can be modeled by *finite* binary trees, where unrepresented projections are implicitly X.

The set of branch traces T is nonrectangular, but we can model T subsets by $\mathsf{J} \to \mathsf{Bool}_\perp$ rectangles, implicitly intersected with T.

Theorem 8 (T model). *If* $\mathsf{T}' \in \mathsf{Rect}\ (\mathsf{J} \to \mathsf{Bool}_\perp)$ *and* $\mathsf{j} \in \mathsf{J}$*, then* $\mathsf{proj\ j}\ (\mathsf{T}' \cap \mathsf{T}) = \mathsf{proj\ j\ T}'$*. If* $\mathsf{B} \subseteq \mathsf{Bool}_\perp$*, then* $\mathsf{unproj\ j}\ (\mathsf{T}' \cap \mathsf{T})\ \mathsf{B} = \mathsf{unproj\ j\ T}'\ \mathsf{B} \cap \mathsf{T}$*.*

Rectangular families are not closed under $(\cup)$. For conditionals, then, we need a lattice join $(\vee)$ with respect to $(\subseteq)$ with the following additional properties:

$$\begin{aligned} (\mathsf{A}_1 \times \mathsf{A}_2) \vee (\mathsf{B}_1 \times \mathsf{B}_2) &= (\mathsf{A}_1 \vee \mathsf{B}_1) \times (\mathsf{A}_2 \vee \mathsf{B}_2) \\ (\textstyle\prod_{\mathsf{j} \in \mathsf{J}} \mathsf{A}_\mathsf{j}) \vee (\textstyle\prod_{\mathsf{j} \in \mathsf{J}} \mathsf{B}_\mathsf{j}) &= \textstyle\prod_{\mathsf{j} \in \mathsf{J}} \mathsf{A}_\mathsf{j} \vee \mathsf{B}_\mathsf{j} \end{aligned} \tag{34}$$

If for every nonproduct type X, $\mathsf{Rect\ X}$ is closed under $(\vee)$, then rectangular families are clearly closed under $(\vee)$. Further, for any A and B, $\mathsf{A} \cup \mathsf{B} \subseteq \mathsf{A} \vee \mathsf{B}$.

Fig. 7 defines approximating preimage arrows. Approximating preimage mapping operations (Fig. 7a) are defined in terms of lattice operations on rectangular families. Every approximating preimage arrow combinator (Fig. 7b) is defined the same way as its corresponding exact preimage arrow combinator, but using approximating preimage mapping operations instead of exact. Fig. 7c defines $\mathsf{random}'_{\mathsf{pre}^*}$ and $\mathsf{branch}'_{\mathsf{pre}^*}$ without using the uncomputable $\mathsf{arr}'_{\mathsf{pre}^*}$, and $\mathsf{ifte}^{\Downarrow\prime}_{\mathsf{pre}^*}$, for interpreting expressions using $[\![\cdot]\!]^{\Downarrow\prime}_{\mathsf{pre}^*}$ for guaranteed termination.

Let $\mathsf{h} := [\![e]\!]^{\Downarrow}_{\mathsf{pre}^*} : \mathsf{X} \underset{\mathsf{pre}^*}{\rightsquigarrow} \mathsf{Y}$ and $\mathsf{h}' := [\![e]\!]^{\Downarrow\prime}_{\mathsf{pre}^*} : \mathsf{X} \underset{\mathsf{pre}^*}{\rightsquigarrow}{}' \mathsf{Y}$ for some expression e.

Theorem 9 (sound, terminating, decreasing). *For all* $\mathsf{A} \in \mathsf{Rect}\ \langle\langle \mathsf{R}, \mathsf{T} \rangle, \mathsf{X} \rangle$ *and* $\mathsf{B} \in \mathsf{Rect\ Y}$*,* $\mathsf{ap}_{\mathsf{pre}}\ (\mathsf{h\ j}_0\ \mathsf{A})\ \mathsf{B} \subseteq \mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}'\ \mathsf{j}_0\ \mathsf{A})\ \mathsf{B} \subseteq \mathsf{A}$*.*

Theorem 10 (monotone). $\lambda \mathsf{A}\,\mathsf{B}.\, \mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}'\ \mathsf{j}_0\ \mathsf{A})\ \mathsf{B}$ *is monotone in* A *and* B*.*

Given these properties, we might try to compute preimages of B by computing preimages with respect to increasingly fine discretizations of A.

Definition 16 (preimage refinement algorithm). *Let* $\mathsf{B} \in \mathsf{Rect\ Y}$ *and*

$$\begin{aligned} &\mathsf{refine} : \mathsf{Rect}\ \langle\langle \mathsf{R}, \mathsf{T} \rangle, \mathsf{X} \rangle \Rightarrow \mathsf{Rect}\ \langle\langle \mathsf{R}, \mathsf{T} \rangle, \mathsf{X} \rangle \\ &\mathsf{refine\ A} := \mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}'\ \mathsf{j}_0\ \mathsf{A})\ \mathsf{B} \end{aligned} \tag{35}$$

Define $\mathsf{partition} : \mathsf{Rect}\ \langle\langle \mathsf{R}, \mathsf{T} \rangle, \mathsf{X} \rangle \Rightarrow \mathsf{Set}\ (\mathsf{Rect}\ \langle\langle \mathsf{R}, \mathsf{T} \rangle, \mathsf{X} \rangle)$ *to produce positive-measure, disjoint rectangles, and define*

$$\begin{aligned} &\mathsf{refine}^* : \mathsf{Set}\ (\mathsf{Rect}\ \langle\langle \mathsf{R}, \mathsf{T} \rangle, \mathsf{X} \rangle) \Rightarrow \mathsf{Set}\ (\mathsf{Rect}\ \langle\langle \mathsf{R}, \mathsf{T} \rangle, \mathsf{X} \rangle) \\ &\mathsf{refine}^*\ \mathcal{A} := \mathsf{image\ refine}\ \left(\textstyle\bigcup_{\mathsf{A} \in \mathcal{A}} \mathsf{partition\ A}\right) \end{aligned} \tag{36}$$

For any $\mathsf{A} \in \mathsf{Rect}\ \langle\langle \mathsf{R}, \mathsf{T} \rangle, \mathsf{X} \rangle$*, iterate* refine^* *on* $\{\mathsf{A}\}$*.*

$$\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup}' \mathsf{Y} ::= \langle \mathsf{Rect\ Y}, \mathsf{Rect\ Y} \Rightarrow \mathsf{Rect\ X} \rangle$$

$$\varnothing'_{\mathsf{pre}} \;:=\; \langle \varnothing, \lambda \mathsf{B}.\, \varnothing \rangle$$

$$\mathsf{ap}'_{\mathsf{pre}} : (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup}' \mathsf{Y}) \Rightarrow \mathsf{Rect\ Y} \Rightarrow \mathsf{Rect\ X}$$
$$\mathsf{ap}'_{\mathsf{pre}}\ \langle \mathsf{Y}', \mathsf{p} \rangle\ \mathsf{B} := \mathsf{p}\ (\mathsf{B} \cap \mathsf{Y}')$$

$$(\circ'_{\mathsf{pre}}) : (\mathsf{Y} \underset{\mathsf{pre}}{\rightharpoonup}' \mathsf{Z}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup}' \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup}' \mathsf{Z})$$
$$\langle \mathsf{Z}', \mathsf{p}_2 \rangle \circ'_{\mathsf{pre}} \mathsf{h}_1 := \langle \mathsf{Z}', \lambda \mathsf{C}.\, \mathsf{ap}'_{\mathsf{pre}}\ \mathsf{h}_1\ (\mathsf{p}_2\ \mathsf{C}) \rangle$$

$$\langle \cdot, \cdot \rangle'_{\mathsf{pre}} : (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup}' \mathsf{Y}_1) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup}' \mathsf{Y}_2) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup}' \mathsf{Y}_1 \times \mathsf{Y}_2)$$
$$\begin{aligned} &\langle \langle \mathsf{Y}'_1, \mathsf{p}_1 \rangle, \langle \mathsf{Y}'_2, \mathsf{p}_2 \rangle \rangle'_{\mathsf{pre}} \;:= \\ &\quad \langle \mathsf{Y}'_1 \times \mathsf{Y}'_2, \lambda \mathsf{B}.\, \mathsf{p}_1\ (\mathsf{proj}_1\ \mathsf{B}) \cap \mathsf{p}_2\ (\mathsf{proj}_2\ \mathsf{B}) \rangle \end{aligned}$$

$$(\uplus'_{\mathsf{pre}}) : (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup}' \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup}' \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup}' \mathsf{Y})$$
$$\begin{aligned} &\langle \mathsf{Y}'_1, \mathsf{p}_1 \rangle \uplus'_{\mathsf{pre}} \langle \mathsf{Y}'_2, \mathsf{p}_2 \rangle \;:= \\ &\quad \langle \mathsf{Y}'_1 \vee \mathsf{Y}'_2, \lambda \mathsf{B}.\, \mathsf{ap}'_{\mathsf{pre}}\ \langle \mathsf{Y}'_1, \mathsf{p}_1 \rangle\ \mathsf{B} \vee \mathsf{ap}'_{\mathsf{pre}}\ \langle \mathsf{Y}'_2, \mathsf{p}_2 \rangle\ \mathsf{B} \rangle \end{aligned}$$

(a) Definitions for preimage mappings that compute rectangular covers.

$$\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow}' \mathsf{Y} ::= \mathsf{Rect\ X} \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup}' \mathsf{Y})$$

$$(\ggg'_{\mathsf{pre}}) : (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow}' \mathsf{Y}) \Rightarrow (\mathsf{Y} \underset{\mathsf{pre}}{\rightsquigarrow}' \mathsf{Z}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow}' \mathsf{Z})$$
$$\begin{aligned} (\mathsf{h}_1 \ggg'_{\mathsf{pre}} \mathsf{h}_2)\ \mathsf{A} \;:=\; \mathsf{let}\ \ &\mathsf{h}'_1 := \mathsf{h}_1\ \mathsf{A} \\ &\mathsf{h}'_2 := \mathsf{h}_2\ (\mathsf{range}'_{\mathsf{pre}}\ \mathsf{h}'_1) \\ \mathsf{in}\ \ &\mathsf{h}'_2 \circ'_{\mathsf{pre}} \mathsf{h}'_1 \end{aligned}$$

$$(\&\&\&'_{\mathsf{pre}}) : (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow}' \mathsf{Y}_1) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow}' \mathsf{Y}_2) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow}' \langle \mathsf{Y}_1, \mathsf{Y}_2 \rangle)$$
$$(\mathsf{h}_1\ \&\&\&'_{\mathsf{pre}}\ \mathsf{h}_2)\ \mathsf{A} \;:=\; \langle \mathsf{h}_1\ \mathsf{A}, \mathsf{h}_2\ \mathsf{A} \rangle'_{\mathsf{pre}}$$

$$\begin{aligned} \mathsf{ifte}'_{\mathsf{pre}} : {} & (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow}' \mathsf{Bool}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow}' \mathsf{Y}) \Rightarrow \\ & (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow}' \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow}' \mathsf{Y}) \end{aligned}$$
$$\begin{aligned} &\mathsf{ifte}'_{\mathsf{pre}}\ \mathsf{h}_1\ \mathsf{h}_2\ \mathsf{h}_3\ \mathsf{A} \;:= \\ &\quad \mathsf{let}\ \ \mathsf{h}'_1 := \mathsf{h}_1\ \mathsf{A} \\ &\qquad\quad \mathsf{h}'_2 := \mathsf{h}_2\ (\mathsf{ap}'_{\mathsf{pre}}\ \mathsf{h}'_1\ \{\mathsf{true}\}) \\ &\qquad\quad \mathsf{h}'_3 := \mathsf{h}_3\ (\mathsf{ap}'_{\mathsf{pre}}\ \mathsf{h}'_1\ \{\mathsf{false}\}) \\ &\qquad \mathsf{in}\ \ \mathsf{h}'_2 \uplus'_{\mathsf{pre}} \mathsf{h}'_3 \end{aligned}$$

$$\mathsf{lazy}'_{\mathsf{pre}} : (1 \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow}' \mathsf{Y})) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow}' \mathsf{Y})$$
$$\mathsf{lazy}'_{\mathsf{pre}}\ \mathsf{h}\ \mathsf{A} \;:=\; \mathsf{if}\ (\mathsf{A} = \varnothing)\ \varnothing'_{\mathsf{pre}}\ (\mathsf{h}\ 0\ \mathsf{A})$$

(b) Approximating preimage arrow, defined using approximating preimage mappings.

$$\mathsf{X} \underset{\mathsf{pre*}}{\rightsquigarrow}' \mathsf{Y} ::= \mathsf{AStore}\ (\mathsf{R} \times \mathsf{T})\ (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow}' \mathsf{Y})$$

$$\mathsf{random}'_{\mathsf{pre*}} : \mathsf{X} \underset{\mathsf{pre*}}{\rightsquigarrow}' [0,1]$$
$$\begin{aligned} &\mathsf{random}'_{\mathsf{pre*}}\ \mathsf{j} \;:= \\ &\quad \mathsf{fst}_{\mathsf{pre}} \ggg'_{\mathsf{pre}} \mathsf{fst}_{\mathsf{pre}} \ggg'_{\mathsf{pre}} \pi_{\mathsf{pre}}\ \mathsf{j} \end{aligned}$$

$$\mathsf{branch}'_{\mathsf{pre*}} : \mathsf{X} \underset{\mathsf{pre*}}{\rightsquigarrow}' \mathsf{Bool}$$
$$\begin{aligned} &\mathsf{branch}'_{\mathsf{pre*}}\ \mathsf{j} \;:= \\ &\quad \mathsf{fst}_{\mathsf{pre}} \ggg'_{\mathsf{pre}} \mathsf{snd}_{\mathsf{pre}} \ggg'_{\mathsf{pre}} \pi_{\mathsf{pre}}\ \mathsf{j} \end{aligned}$$

$$\mathsf{fst}'_{\mathsf{pre*}} := \eta'_{\mathsf{pre*}}\ \mathsf{fst}_{\mathsf{pre}};\ \cdots$$

$$\mathsf{ifte}^{\Downarrow\prime}_{\mathsf{pre*}} : (\mathsf{X} \underset{\mathsf{pre*}}{\rightsquigarrow}' \mathsf{Bool}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre*}}{\rightsquigarrow}' \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre*}}{\rightsquigarrow}' \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre*}}{\rightsquigarrow}' \mathsf{Y})$$
$$\begin{aligned} &\mathsf{ifte}^{\Downarrow\prime}_{\mathsf{pre*}}\ \mathsf{k}_1\ \mathsf{k}_2\ \mathsf{k}_3\ \mathsf{j} \;:= \\ &\quad \mathsf{let}\ \ \langle \mathsf{C_k}, \mathsf{p_k} \rangle := \mathsf{k}_1\ (\mathsf{left\ j})\ \mathsf{A} \\ &\qquad\quad \langle \mathsf{C_b}, \mathsf{p_b} \rangle := \mathsf{branch}_{\mathsf{pre*}}\ \mathsf{j}\ \mathsf{A} \\ &\qquad\qquad \mathsf{C}_2 := \mathsf{C_k} \cap \mathsf{C_b} \cap \{\mathsf{true}\} \\ &\qquad\qquad \mathsf{C}_3 := \mathsf{C_k} \cap \mathsf{C_b} \cap \{\mathsf{false}\} \\ &\qquad\qquad \mathsf{A}_2 := \mathsf{p_k}\ \mathsf{C}_2 \cap \mathsf{p_b}\ \mathsf{C}_2 \\ &\qquad\qquad \mathsf{A}_3 := \mathsf{p_k}\ \mathsf{C}_3 \cap \mathsf{p_b}\ \mathsf{C}_3 \\ &\qquad \mathsf{in}\ \ \mathsf{if}\ (\mathsf{C_b} = \{\mathsf{true}, \mathsf{false}\}) \\ &\qquad\qquad \langle \top, \lambda \mathsf{B}.\, \mathsf{A}_2 \vee \mathsf{A}_3 \rangle \\ &\qquad\qquad (\mathsf{k}_2\ (\mathsf{left\ (right\ j)})\ \mathsf{A}_2 \uplus'_{\mathsf{pre}} \mathsf{k}_3\ (\mathsf{right\ (right\ j)})\ \mathsf{A}_3) \end{aligned}$$

(c) Preimage* arrow combinators for probabilistic choice and guaranteed termination. Fig. 5 (AStore arrow transformer) defines $\eta'_{\mathsf{pre*}}$, $(\ggg'_{\mathsf{pre*}})$, $(\&\&\&'_{\mathsf{pre*}})$, $\mathsf{ifte}'_{\mathsf{pre*}}$ and $\mathsf{lazy}'_{\mathsf{pre*}}$.

Fig. 7: Implementable arrows that approximate preimage arrows. Specific lifts such as $\mathsf{fst}_{\mathsf{pre}} := \mathsf{arr}_{\mathsf{pre}}\ \mathsf{fst}$ are computable (see Fig. 6), but $\mathsf{arr}'_{\mathsf{pre}}$ is not.

Theorem 10 guarantees refining a partition of A never does worse than refining A itself. Theorem 9 guarantees $\mathsf{refine\ A} \subseteq \mathsf{A}$ (which implies termination), and that the algorithm is **sound**: the preimage of B is always contained in the covering partition refine^* returns. Ideally, it would be complete in the sense that covering partitions converge to a set that overapproximates by a measure-zero subset. Unfortunately, convergence fails on some examples that terminate with probability less than one. We leave completeness conditions for future work, and for now, use algorithms that depend only on soundness.

6 Implementations

We have three implementations: two direct implementations of the approximating semantics, and a less direct but more efficient one called ***Dr. Bayes***.

Given a library for operating on rectangular sets, the approximating preimage arrows defined in Figs. 6 and 7 can be implemented with few changes in any practical λ-calculus. We have done so in Typed Racket [24] and Haskell [1]. Both implementations are almost line-for-line transliterations from the figures. They are at `https://github.com/ntoronto/writing/tree/master/2014esop-code`.

Dr. Bayes is written in Typed Racket. It includes $[\![\cdot]\!]_{\mathsf{a}^*}$ (Fig. 1), its extension $[\![\cdot]\!]^{\Downarrow}_{\mathsf{a}^*}$, the bottom* arrow (Figs. 2 and 5), the approximating preimage and preimage* arrows (Figs. 6 and 7), and extensions to compute approximate preimages under arithmetic. The preimage arrows operate on a monomorphic rectangular set data type, which includes tagged rectangles and disjoint unions for ad-hoc polymorphism, and floating-point intervals to overapproximate real intervals.

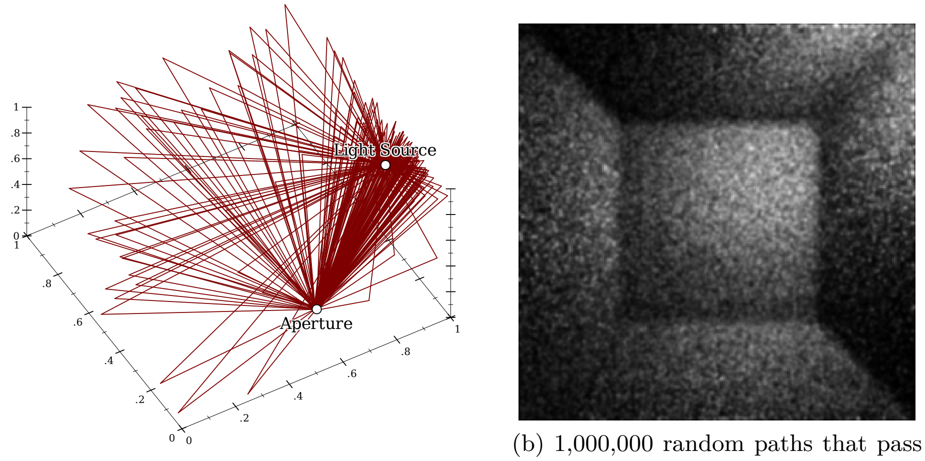
Definition 16 outlines preimage refinement, a discretization algorithm that repeatedly shrinks and repartitions a program's domain. Dr. Bayes does not use this algorithm directly because it is inefficient: good accuracy requires fine discretization, which is exponential in the number of discretized axes. Instead of enumerating partitions of the random source, Dr. Bayes samples from them with time complexity linear in the number of samples and discretized axes. It uses bottom* arrow computations to get output samples, rejecting those outside the requested output set. Thus, it relies only on preimage refinement's soundness.

Fig. 8 shows the result of using Dr. Bayes for stochastic ray tracing [28]. In this instance, photons are cast from a light source in a uniformly random direction and are reflected by the walls of a square room, generating paths. The objective is to sample, with the correct distribution, only those paths that pass through an aperture. The smaller the aperture, the smaller the probability a path passes through it, and the more focused the resulting image.

All efficient implementations of stochastic ray tracing to date use sophisticated, specialized sampling methods that bear little resemblance to the physical processes they simulate. The proof-of-concept ray tracer, written in Dr. Bayes, is little more than a simple physics simulation and a conditional query.

7 Related Work

Our approximating semantics can be regarded as an abstract interpretation [7]. It is typical in some ways: it is sound, the abstract domain is a lattice, and the concrete (i.e. exact) semantics performs infinite computations. It is atypical in other ways: it is used to run programs, there is no Kleene iteration, abstraction boundaries are `if` expressions in (conceptually) infinite programs, and infinite computations are confined to a combinator library. This last property makes our work similar to monadic abstract interpretation [23], but λ_{ZFC} allows concrete combinators to perform uncountably many operations.



(a) Random paths from a single light source, conditioned on passing through an aperture.

(b) 1,000,000 random paths that pass through the aperture, projected onto a plane and accumulated.

```
(struct/drbayes collision (time point normal))

(define/drbayes (ray-plane-intersect p0 v n d)
  (let ([denom  (- (vec-dot v n))])
    (if (positive? denom)
        (let ([t  (/ (+ d (vec-dot p0 n)) denom)])
          (if (positive? t) (collision t (vec+ p0 (vec-scale v t)) n) #f))
        #f)))
```

(c) Part of the ray tracer implementation. Sampling involves computing approximate preimages under functions like this.

Fig. 8: Stochastic ray tracing in Dr. Bayes is little more than physics simulation.

Probabilistic languages can be approximately placed into two groups: those defined by an implementation, and those defined by a semantics.

Some languages defined by an implementation are probabilistic Scheme [14], BUGS [17], BLOG [19], BLAISE [5], Church [8], and Kiselyov's embedded language for O'Caml [12]. The reports on these languages generally describe interpreters, compilers, and algorithms for sampling with probabilistic conditions. Recently, Wingate et al [30] have defined the semantics of nonstandard interpretations that enable efficient inference, but do not define the languages.

Early work in probabilistic language semantics is not motivated by Bayesian concerns, and thus does not address conditioning. Examples are Kozen [15], Hurd [10], Jones [11], Ramsey and Pfeffer [22], and Park [20]. Recent semantics work tackles conditioning, such as IBAL [21] and Fun [6]. While Fun's measure-theoretic semantics looks promising, its implementations are based on probability densities, so they cannot handle recursion or aribtrary conditions.

8 Conclusions and Future Work

To allow recursion and arbitrary conditions in probabilitic programs, we combined the power of measure theory with the unifying elegance of arrows. We

1. Defined a transformation from first-order programs to arbitrary arrows.
2. Defined the bottom arrow as a standard translation target.
3. Derived the uncomputable preimage arrow as an alternative target.
4. Derived a sound, computable approximation of the preimage arrow, and enough computable lifts to transform programs.

Critically, the preimage arrow's lift from the bottom arrow distributes over bottom arrow computations. Our semantics thus generalizes this process to all programs: 1) encode a program as a bottom arrow computation; 2) lift this computation to get an uncomputable function that computes preimages; 3) distribute the lift; and 4) replace uncomputable expressions with sound approximations.

Our semantics trades efficiency for simplicity by threading a constant, tree-shaped random source (Section 4.1). Passing subtrees instead would make random a constant-time primitive, and allow combinators to detect lack of change and return cached values. Other future optimization work includes creating new sampling algorithms, and using other easily measured but more expressive set representations, such as parallelotopes [2]. On the theory side, we intend to explore preimage computation's connection to type checking and type inference, investigate ways to integrate and leverage polymorphic type systems, and find the conditions under which preimage refinement is complete in the limit.

More broadly, we hope to advance Bayesian practice by providing a rich modeling language with an efficient, correct implementation, which allows general recursion and any computable, probabilistic condition.

References

1. Haskell 98 language and libraries, the revised report (December 2002), `http://www.haskell.org/onlinereport/`
2. Amato, G., Scozzari, F.: The abstract domain of parallelotopes. Electronic Notes in Theoretical Computer Science 287, 17–28 (November 2012)
3. Aumann, R.J.: Borel structures for function spaces. Illinois Journal of Mathematics 5, 614–630 (1961)
4. Barras, B.: Sets in Coq, Coq in sets. Journal of Formalized Reasoning 3(1) (2010)
5. Bonawitz, K.A.: Composable Probabilistic Inference with Blaise. Ph.D. thesis, Massachusetts Institute of Technology (2008)
6. Borgström, J., Gordon, A.D., Greenberg, M., Margetson, J., Gael, J.V.: Measure transformer semantics for Bayesian machine learning. In: European Symposium on Programming. pp. 77–96 (2011)
7. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Principles of Programming Languages. pp. 238–252 (1977)
8. Goodman, N., Mansinghka, V., Roy, D., Bonawitz, K., Tenenbaum, J.: Church: a language for generative models. In: Uncertainty in Artificial Intelligence (2008)

9. Hughes, J.: Generalizing monads to arrows. In: Science of Computer Programming. vol. 37, pp. 67–111 (2000)
10. Hurd, J.: Formal Verification of Probabilistic Algorithms. Ph.D. thesis, University of Cambridge (2002)
11. Jones, C.: Probabilistic Non-Determinism. Ph.D. thesis, University of Edinburgh (1990)
12. Kiselyov, O., Shan, C.: Monolingual probabilistic programming using generalized coroutines. In: Uncertainty in Artificial Intelligence (2008)
13. Klenke, A.: Probability Theory: A Comprehensive Course. Springer (2006)
14. Koller, D., McAllester, D., Pfeffer, A.: Effective Bayesian inference for stochastic programs. In: 14th National Conference on Artificial Intelligence (August 1997)
15. Kozen, D.: Semantics of probabilistic programs. In: Foundations of Computer Science (1979)
16. Lindley, S., Wadler, P., Yallop, J.: The arrow calculus. Journal of Functional Programming 20, 51–69 (2010)
17. Lunn, D.J., Thomas, A., Best, N., Spiegelhalter, D.: WinBUGS – a Bayesian modelling framework. Statistics and Computing 10(4) (2000)
18. McBride, C., Paterson, R.: Applicative programming with effects. Journal of Functional Programming 18(1) (2008)
19. Milch, B., Marthi, B., Russell, S., Sontag, D., Ong, D., Kolobov, A.: BLOG: Probabilistic models with unknown objects. In: International Joint Conference on Artificial Intelligence (2005)
20. Park, S., Pfenning, F., Thrun, S.: A probabilistic language based upon sampling functions. Transactions on Programming Languages and Systems 31(1) (2008)
21. Pfeffer, A.: The design and implementation of IBAL: A general-purpose probabilistic language. In: Statistical Relational Learning. MIT Press (2007)
22. Ramsey, N., Pfeffer, A.: Stochastic lambda calculus and monads of probability distributions. In: Principles of Programming Languages (2002)
23. Sergey, I., Devriese, D., Might, M., Midtgaard, J., Darais, D., Clarke, D., Piessens, F.: Monadic abstract interpreters. In: Programming Language Design and Implementation. pp. 399–410 (2013)
24. Tobin-Hochstadt, S., Felleisen, M.: The design and implementation of typed Scheme. In: Principles of Programming Languages. pp. 395–406 (2008)
25. Toronto, N., McCarthy, J.: Running probabilistic programs backward (full version). Tech. rep., `https://github.com/ntoronto/writing/tree/master/2014esop-long`
26. Toronto, N., McCarthy, J.: From Bayesian notation to pure Racket, via measure-theoretic probability in λ_{ZFC}. In: Implementation and Application of Functional Languages (2010)
27. Toronto, N., McCarthy, J.: Computing in Cantor's paradise with λ_{ZFC}. In: Functional and Logic Programming Symposium. pp. 290–306 (2012)
28. Veach, E., Guibas, L.J.: Metropolis light transport. In: ACM SIGGRAPH. pp. 65–76 (1997)
29. Wadler, P.: Monads for functional programming. In: Jeuring, J., Meijer, E. (eds.) Advanced Functional Programming (2001)
30. Wingate, D., Goodman, N.D., Stuhlmüller, A., Siskind, J.M.: Nonstandard interpretations of probabilistic programs for efficient inference. In: Neural Information Processing Systems. pp. 1152–1160 (2011)