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 directly in Typed Racket and Haskell.

Keywords: Probability, Semantics (XXX: more?)

1 Introduction

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

Unfortunately, there is currently no efficient probabilistic language implementation that supports conditioning and places no extraneous restrictions on 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.

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 $\mathtt{t}$'s distribution has a density (a standard bell curve at mean μ), the distribution of $\mathtt{t}'$ does not.

Densities do not allow reasoning about arbitrary conditions. If x and y are primitive random variables—loosely, untransformed probabilistic values, such as $\mathtt{t}$ in (1)—then **Bayes' law for densities** gives the density of x given y:

$$f_x(x \mid y) \;=\; \frac{f_y(y \mid x) \cdot \pi_x(x)}{\int f_y(y \mid x) \cdot \pi_x(x)\; dx} \tag{2}$$

Bayesians interpret probabilistic processes as defining densities π_x and f_y, and use (2) to discover the density of x given $y = c$ for some constant c. While x given $\sin(y) = -1$ and x given $x + y = 0$ are perfectly sensible to reason about, Bayes' law for densities cannot express them. Thus, reasoning with densities disallows all but the simplest conditions.

1.1 Probability Measures

Measure-theoretic probability [19] 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 probability **measures**. In contrast to densities, probabilities of sets of values with different dimension *are* comparable, and probability measures *can* be defined on infinite products.

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

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

defines the distribution over Y, where $f^{-1}(B)$ is the subset of f's domain X for which f yields a value in B. In the thermometer example (1), f would be an interpretation of the program as a function, X would be the set of all random sources, and Y would be $\mathbb{R}$. For any $B \subseteq Y$, $f^{-1}(B)$ is well-defined, regardless of discontinuities.

Measure-theoretic probability supports any kind of condition. The probability of $B' \subseteq Y$ given $B \subseteq Y$ is

$$\Pr[B' \mid B] \;=\; \Pr[B' \cap B] \;/\; \Pr[B] \tag{4}$$

if $\Pr[B] > 0$. If $\Pr[B] = 0$, conditional probabilities can be calculated by applying (4) to decending sequences $B_1 \supseteq B_2 \supseteq B_3 \supseteq \cdots$ of positive-probability sets whose intersection is B, and taking a limit. If $Y = \mathbb{R} \times \mathbb{R}$, for example, the distribution over $\langle x, y\rangle \in Y$ given that $x + y = 0$ can be calculated using a descending sequence of sets defined by $B_n = \{\langle x, y\rangle \in Y \mid |x + y| < 2^{-n}\}$.

Unfortunately, there is a complicated technical restriction: only *measurable* subsets of X and Y can be assigned probabilities. This and having to take limits tend to drive practitioners to densities, even though they are so limited.

1.2 Measure-Theoretic Semantics

Because purely functional languages do not allow side effects (except usually nontermination), programmers must write probabilistic programs as functions from a random source to outputs. Monads and other categorical classes such as idioms (i.e. applicative functors) can make doing so easier [15, 30].

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} [31], 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.

XXX: something about difficulty 2

For difficulty 3, we have discovered that the "first-orderness" of arrows [14] is a perfect fit for the "first-orderness" of measure theory.

1.3 Arrow Solution Overview

Using arrows, we define an *exact* semantics and an *approximating* semantics. Our exact semantics consists of

- A semantic function which, like the arrow calculus [22] semantic function, 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 arrows used to define the exact semantics:

$$\begin{array}{ccc} \mathsf{X} \rightsquigarrow_{\perp} \mathsf{Y} & \xrightarrow{\mathsf{lift}_{\mathsf{pre}}} & \mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y} \\ {\scriptstyle \eta_{\perp^*}}\downarrow & & \downarrow{\scriptstyle \eta_{\mathsf{pre}^*}} \\ \mathsf{X} \rightsquigarrow_{\perp^*} \mathsf{Y} & \xrightarrow[\mathsf{lift}_{\mathsf{pre}^*}]{} & \mathsf{X} \underset{\mathsf{pre}^*}{\rightsquigarrow} \mathsf{Y} \end{array} \tag{5}$$

In the top row, $\mathsf{X} \rightsquigarrow_{\perp} \mathsf{Y}$ computations are functions that may raise errors and $\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}$ computations 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 always terminate. (We can do this because in λ_{ZFC}, Turing-uncomputable programs are definable.) Most of our correctness theorems rely on proofs that every morphism in (5) is a homomorphism.

Our approximating semantics consists of the same semantic function and an arrow $\mathsf{X} \underset{\mathsf{pre}^*}{\rightsquigarrow}' \mathsf{Y}$, derived from $\mathsf{X} \underset{\mathsf{pre}^*}{\rightsquigarrow} \mathsf{Y}$, for computing conservative approximations of preimages. An implementation is comprised of the semantic function, and the $\mathsf{X} \rightsquigarrow_{\perp} \mathsf{Y}$ and $\mathsf{X} \underset{\mathsf{pre}^*}{\rightsquigarrow}' \mathsf{Y}$ arrows' combinators.

2 Operational Metalanguage

We write programs in λ_{ZFC} [31], an untyped, call-by-value λ-calculus designed for deriving implementable programs from contemporary mathematics.

Contemporary mathematics—measure theory in particular—is usually done in **ZFC**: **Zermelo-Fraenkel** set theory with the axiom of **Choice**. ZFC has only first-order functions and no general recursion, which makes implementing a language defined by a transformation into ZFC quite difficult. The problem is exacerbated if implementing the language requires approximation. 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. Essentially every ZFC theorem applies to λ_{ZFC}'s set values without alteration. Further, 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).

λ_{ZFC} is untyped so its users can define an auxiliary type system that best suits their application area. For this work, we use a manually checked, polymorphic type system 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.

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

Because the type $\mathsf{Set\ X}$ 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 $\mathsf{X} \times \mathsf{Y}$.

We write λ_{ZFC} programs in heavily sugared λ-calculus syntax, with an if expression and additional primitives such as membership $(\in) : \mathsf{x} \Rightarrow \mathsf{Set\ x} \Rightarrow \mathsf{Bool}$, powerset $\mathcal{P} : \mathsf{Set\ x} \Rightarrow \mathsf{Set\ (Set\ x)}$ and big union $\bigcup : \mathsf{Set\ (Set\ x)} \Rightarrow \mathsf{Set\ x}$. We use 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 we have logical operators, bounded quantifiers, and typical set operations.

In set theory, because functions are encoded as sets of input-output pairs, they inherit the extensionality of sets. The increment function for the natural numbers, for example, 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. For convenience, as with lambdas, we use adjacency (e.g. $(\mathsf{f\ x})$) to apply mappings.

The set $\mathsf{X} \to \mathsf{Y}$ contains all the *total* mappings from X to Y. We use total mappings as possibly infinite vectors, with application for indexing. Indexing functions are produced by

$$\begin{aligned} &\pi : \mathsf{J} \Rightarrow (\mathsf{J} \to \mathsf{X}) \Rightarrow \mathsf{X} \\ &\pi\ \mathsf{j\ f}\ :=\ \mathsf{f\ j} \end{aligned} \tag{6}$$

which is particularly useful when f is unnamed.

Because of the way λ_{ZFC}'s lambda terms are defined, for two lambdas e_1 and e_2, $e_1 = e_2$ reduces to true when e_1 and e_2 are alpha-equivalent. For example, $(\lambda \mathsf{a.\ a}) = (\lambda \mathsf{b.\ b})$ reduces to true, but $(\lambda \mathsf{a.}\ 2) = (\lambda \mathsf{a.}\ 1 + 1)$ reduces to false.

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

Any truth statement e implies e terminates. In particular, $e_1 = e_2$ implies e_1 and e_2 both terminate. However, we often want to say that e_1 and e_2 are equivalent when they both loop.

Definition 1 (observational equivalence). *Two λ_{ZFC} terms e_1 and e_2 are **observationally equivalent**, written $e_1 \equiv e_2$, when $e_1 = e_2$ or both e_1 and e_2 do not terminate.*

It might seem helpful to introduce even coarser notions of equivalence, such as applicative bisimilarity [2]. However, we do not want internal equality and external equivalence to differ too much, and we want the flexibility of extending "$\equiv$" with type-specific rules.

3 Arrows and First-Order Semantics

Like monads and idioms [32, 24], arrows [14] are used to thread effects through computations in a way that imposes structure on the computations. Unlike monad and idiom computations, 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 perfect fit for a compositional translation from expressions to functions, or to computations that compute preimages under those same functions. The second property makes arrows a perfect fit for a measure-theoretic semantics in particular, as app in the function arrow is generally not measurable [3]. Targeting arrows in the semantics therefore gives some assurance that we can meet measure theory's requirement that preimage measure be defined only for measurable functions.

3.1 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 $(\&\&\&_\mathsf{a})$—typically called **fanout**, but its use will be clearer if we call it **pairing**—which applies two functions to an input and returns the pair of their outputs. One way to strengthen an arrow a is to define an additional combinator left_a, which can be used to choose an arrow computation based on the result of another. Again, we define a different combinator, ifte_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 to defer computations in conditional branches. For example, define the **function arrow** with choice:

$$\begin{aligned} \mathsf{arr\ f} &:= \mathsf{f} \\ (\mathsf{f}_1 \ggg \mathsf{f}_2)\ \mathsf{a} &:= \mathsf{f}_2\ (\mathsf{f}_1\ \mathsf{a}) \\ (\mathsf{f}_1 \,\&\&\&\, \mathsf{f}_2)\ \mathsf{a} &:= \langle \mathsf{f}_1\ \mathsf{a}, \mathsf{f}_2, \mathsf{a} \rangle \\ \mathsf{ifte\ f}_1\ \mathsf{f}_2\ \mathsf{f}_3\ \mathsf{a} &:= \mathsf{if}\ (\mathsf{f}_1\ \mathsf{a})\ (\mathsf{f}_2\ \mathsf{a})\ (\mathsf{f}_3\ \mathsf{a}) \end{aligned} \tag{7}$$

and try to define the following recursive function:

$$\mathsf{halt\text{-}on\text{-}true} := \mathsf{ifte}\ (\mathsf{arr\ id})\ (\mathsf{arr\ id})\ \mathsf{halt\text{-}on\text{-}true} \tag{8}$$

The defining expression loops in a strict λ-calculus. In a nonstrict λ-calculus, it loops only when applied to false. Using $\mathsf{lazy\ f\ a} := \mathsf{f}\ 0\ \mathsf{a}$, which receives thunks and returns arrow computations, we can write halt-on-true using $\mathsf{lazy}\ \lambda 0.\ \mathsf{halt\text{-}on\text{-}true}$ for the else branch, so that it loops only when applied to false in any λ-calculus.

Definition 2 (arrow with choice). *A binary type constructor* $(\rightsquigarrow_\mathsf{a})$ *and*

$$\begin{aligned} \mathsf{arr}_\mathsf{a} &: (\mathsf{x} \Rightarrow \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{y}) \\ (\ggg_\mathsf{a}) &: (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{y}) \Rightarrow (\mathsf{y} \rightsquigarrow_\mathsf{a} \mathsf{z}) \Rightarrow (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{z}) \\ (\&\&\&_\mathsf{a}) &: (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{z}) \Rightarrow (\mathsf{x} \rightsquigarrow_\mathsf{a} \langle \mathsf{y}, \mathsf{z} \rangle) \end{aligned} \tag{9}$$

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

$$\begin{aligned} \mathsf{ifte}_\mathsf{a} &: (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{Bool}) \Rightarrow (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{y}) \\ \mathsf{lazy}_\mathsf{a} &: (1 \Rightarrow (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{y})) \Rightarrow (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{y}) \end{aligned} \tag{10}$$

where $1 = \{0\}$*, define an* ***arrow with choice*** *if certain additional homomorphism and structural laws hold.*

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

The necessary homomorphism laws can be put in terms of more general homomorphism properties that deal with distributing an arrow-to-arrow lift, which we use extensively to prove correctness.

Definition 3 (arrow homomorphism). *A function* $\mathsf{lift}_\mathsf{b} : (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_\mathsf{b} \mathsf{y})$ *is an* ***arrow homomorphism*** *from arrow* a *to arrow* b *if the following distributive laws hold for appropriately typed* f*,* f_1*,* f_2 *and* f_3*:*

$$\mathsf{lift}_\mathsf{b}\ (\mathsf{arr}_\mathsf{a}\ \mathsf{f}) \equiv \mathsf{arr}_\mathsf{b}\ \mathsf{f} \tag{11}$$

$$\mathsf{lift}_\mathsf{b}\ (\mathsf{f}_1 \ggg_\mathsf{a} \mathsf{f}_2) \equiv (\mathsf{lift}_\mathsf{b}\ \mathsf{f}_1) \ggg_\mathsf{b} (\mathsf{lift}_\mathsf{b}\ \mathsf{f}_2) \tag{12}$$

$$\mathsf{lift}_\mathsf{b}\ (\mathsf{f}_1 \mathbin{\&\&\&_\mathsf{a}} \mathsf{f}_2) \equiv (\mathsf{lift}_\mathsf{b}\ \mathsf{f}_1) \mathbin{\&\&\&_\mathsf{b}} (\mathsf{lift}_\mathsf{b}\ \mathsf{f}_2) \tag{13}$$

$$\mathsf{lift}_\mathsf{b}\ (\mathsf{ifte}_\mathsf{a}\ \mathsf{f}_1\ \mathsf{f}_2\ \mathsf{f}_3) \equiv \mathsf{ifte}_\mathsf{b}\ (\mathsf{lift}_\mathsf{b}\ \mathsf{f}_1)\ (\mathsf{lift}_\mathsf{b}\ \mathsf{f}_2)\ (\mathsf{lift}_\mathsf{b}\ \mathsf{f}_3) \tag{14}$$

$$\mathsf{lift}_\mathsf{b}\ (\mathsf{lazy}_\mathsf{a}\ \mathsf{f}) \equiv \mathsf{lazy}_\mathsf{b}\ \lambda 0.\, \mathsf{lift}_\mathsf{b}\ (\mathsf{f}\ 0) \tag{15}$$

The arrow homomorphism laws state that $\mathsf{arr}_\mathsf{a} : (\mathsf{x} \Rightarrow \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{y})$ must be a homomorphism from the function arrow (7) to arrow a. Roughly, arrow computations that do not use additional combinators can be transformed into arr_a applied to a pure computation. They must be *function-like*.

Rather than prove each necessary arrow law, we prove arrows are *epimorphic* (not necessarily *isomorphic*) to arrows for which the laws are known to hold.

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

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

Proof. For each law, substitute right inverses, factor out lift_b, apply law for arrow a, distribute lift_b, and cancel right inverses. □

3.2 First-Order Let-Calculus Semantics

Fig. 1 defines a transformation $[\![\cdot]\!]_\mathsf{a}$ from a first-order let-calculus to arrow computations for any arrow a.

A program is a sequence of definition statements followed by a final expression. The semantic function $[\![\cdot]\!]_\mathsf{a}$ transforms each defining expression and the

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

$$
[\![x := e;\ \ldots\ ; e_{body}]\!]_{\mathsf{a}} :\equiv x := [\![e]\!]_{\mathsf{a}};\ \ldots\ ; [\![e_{body}]\!]_{\mathsf{a}}
$$

$$
\begin{aligned}
[\![x\ e]\!]_{\mathsf{a}} &:\equiv [\![\langle e, \langle\rangle\rangle]\!]_{\mathsf{a}} \ggg_{\mathsf{a}} x \\
[\![\langle e_1, e_2\rangle]\!]_{\mathsf{a}} &:\equiv [\![e_1]\!]_{\mathsf{a}} \mathbin{\&\&\&_{\mathsf{a}}} [\![e_2]\!]_{\mathsf{a}} \\
[\![\mathsf{fst}\ e]\!]_{\mathsf{a}} &:\equiv [\![e]\!]_{\mathsf{a}} \ggg_{\mathsf{a}} \mathsf{arr}_{\mathsf{a}}\ \mathsf{fst} \\
[\![\mathsf{snd}\ e]\!]_{\mathsf{a}} &:\equiv [\![e]\!]_{\mathsf{a}} \ggg_{\mathsf{a}} \mathsf{arr}_{\mathsf{a}}\ \mathsf{snd} \\
[\![v]\!]_{\mathsf{a}} &:\equiv \mathsf{arr}_{\mathsf{a}}\ (\mathsf{const}\ v) \\
\\
\mathsf{id} &:= \lambda \mathsf{a}.\, \mathsf{a} \\
\mathsf{const}\ \mathsf{b} &:= \lambda \mathsf{a}.\, \mathsf{b}
\end{aligned}
\qquad
\begin{aligned}
[\![\mathsf{let}\ e\ e_{body}]\!]_{\mathsf{a}} &:\equiv ([\![e]\!]_{\mathsf{a}} \mathbin{\&\&\&_{\mathsf{a}}} \mathsf{arr}_{\mathsf{a}}\ \mathsf{id}) \ggg_{\mathsf{a}} [\![e_{body}]\!]_{\mathsf{a}} \\
[\![\mathsf{env}\ 0]\!]_{\mathsf{a}} &:\equiv \mathsf{arr}_{\mathsf{a}}\ \mathsf{fst} \\
[\![\mathsf{env}\ (n+1)]\!]_{\mathsf{a}} &:\equiv \mathsf{arr}_{\mathsf{a}}\ \mathsf{snd} \ggg_{\mathsf{a}} [\![\mathsf{env}\ n]\!]_{\mathsf{a}} \\
[\![\mathsf{if}\ e_c\ e_t\ e_f]\!]_{\mathsf{a}} &:\equiv \mathsf{ifte}_{\mathsf{a}}\ [\![e_c]\!]_{\mathsf{a}}\ [\![\mathsf{lazy}\ e_t]\!]_{\mathsf{a}}\ [\![\mathsf{lazy}\ e_f]\!]_{\mathsf{a}} \\
[\![\mathsf{lazy}\ e]\!]_{\mathsf{a}} &:\equiv \mathsf{lazy}_{\mathsf{a}}\ \lambda 0.\, [\![e]\!]_{\mathsf{a}} \\
\\
&\text{subject to } [\![p]\!]_{\mathsf{a}} : \langle\rangle \rightsquigarrow_{\mathsf{a}} \mathsf{y} \text{ for some } \mathsf{y}
\end{aligned}
$$

Fig. 1: Transformation from a let-calculus with first-order definitions and De-Bruijn-indexed bindings to computations in arrow a.

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, the interpretation acts like a stack machine. The 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. Let-bindings push values 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 $[\![e]\!]_{\mathsf{a}} : \mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}$ for some x and y, and $[\![e]\!]_{\mathsf{a}}$ terminates.*

From here on, we assume all expressions are well-defined. (The arrow a will be clear from context.) This does not guarantee that *running* any given interpretation terminates; it just simplifies unqualified statements about expressions.

Most of our semantic correctness results rely on the following theorem.

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, $[\![e]\!]_{\mathsf{b}} \equiv \mathsf{lift}_{\mathsf{b}}\ [\![e]\!]_{\mathsf{a}}$.*

Proof. By structural induction and homomorphism properties (11)–(15). □

If we assume that $\mathsf{lift}_{\mathsf{b}}$ 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, $[\![\cdot]\!]_{\mathsf{b}}$ is correct.

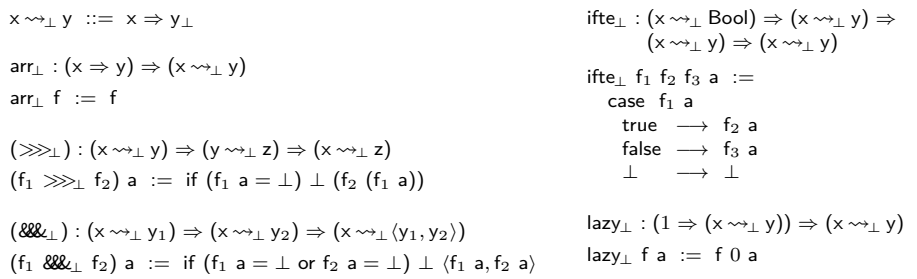

Fig. 2: Bottom arrow definitions.

4 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 arrow whose behavior is well-understood. One obvious candidate is the function arrow (7). However, we will need to explicitly handle nontermination as an error value, so we need a slightly more complicated arrow for which running computations may raise an error.

Fig. 2 defines the ***bottom arrow***. Its computations are of type $\mathsf{x} \rightsquigarrow_\perp \mathsf{y} ::= \mathsf{x} \Rightarrow \mathsf{y}_\perp$, where the inhabitants of $\mathsf{y}_\perp$ are the error value $\perp$ as well as the inhabitants of y. The type $\mathsf{Bool}_\perp$, for example, denotes the members of $\mathsf{Bool} \cup \{\perp\}$.

If we wish to claim that $\mathsf{x} \rightsquigarrow_\perp \mathsf{y}$ computations obey the arrow laws, we need a notion of equivalence that is slightly coarser than observational equivalence.

Definition 6 (bottom arrow equivalence). *Two computations* $\mathsf{f}_1 : \mathsf{x} \rightsquigarrow_\perp \mathsf{y}$ *and* $\mathsf{f}_2 : \mathsf{x} \rightsquigarrow_\perp \mathsf{y}$ *are equivalent, or* $\mathsf{f}_1 \equiv \mathsf{f}_2$*, when* $\mathsf{f}_1\ \mathsf{a} \equiv \mathsf{f}_2\ \mathsf{a}$ *for all* $\mathsf{a} : \mathsf{x}$*.*

It is not hard to show that the bottom arrow is epimorphic to the Maybe monad's Kleisli arrow; by Theorem 1, the arrow laws hold.

4.1 Lazy Preimage Mappings

To compute with infinite sets in the language implementation, we need an abstraction that makes it easy to replace computation on concrete sets with computation on abstract sets. Therefore, in the preimage arrow, we confine set computations to instances of

$$\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y} ::= \langle \mathsf{Set\ Y}, \mathsf{Set\ Y} \Rightarrow \mathsf{Set\ X} \rangle \tag{16}$$

Like a mapping, an $\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y}$ has an observable domain—but computing the input-output pairs is delayed. We therefore call these ***lazy preimage mappings***. The lack of $\perp$ in the type makes ignore nonterminating inputs easier further on.

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

$$\mathsf{pre} : (\mathsf{X} \rightsquigarrow_\perp \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y})$$
$$\mathsf{pre\ f\ A} := \langle \mathsf{image}_\perp\ \mathsf{f\ A}, \lambda \mathsf{B}.\, \mathsf{preimage}_\perp\ \mathsf{f\ A\ B} \rangle$$

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

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

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

$$\langle \cdot, \cdot \rangle_{\mathsf{pre}} : (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y}_1) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y}_2) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y}_1 \times \mathsf{Y}_2)$$
$$\langle \langle \mathsf{Y}'_1, \mathsf{p}_1 \rangle, \langle \mathsf{Y}'_2, \mathsf{p}_2 \rangle \rangle_{\mathsf{pre}} :=$$
$$\quad \mathsf{let}\ \ \mathsf{Y}' := \mathsf{Y}'_1 \times \mathsf{Y}'_2$$
$$\qquad \mathsf{p} := \lambda \mathsf{B}.\ \bigcup_{\langle \mathsf{b}_1, \mathsf{b}_2 \rangle \in \mathsf{B}} (\mathsf{p}_1\ \{\mathsf{b}_1\}) \cap (\mathsf{p}_2\ \{\mathsf{b}_2\})$$
$$\quad \mathsf{in}\ \ \langle \mathsf{Y}', \mathsf{p} \rangle$$

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

$$(\uplus_{\mathsf{pre}}) : (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y})$$
$$\mathsf{h}_1 \uplus_{\mathsf{pre}} \mathsf{h}_2 := \mathsf{let}\ \ \mathsf{Y}' := (\mathsf{range_{pre}}\ \mathsf{h}_1) \cup (\mathsf{range_{pre}}\ \mathsf{h}_2)$$
$$\mathsf{p} := \lambda \mathsf{B}.\, (\mathsf{ap_{pre}}\ \mathsf{h}_1\ \mathsf{B}) \cup (\mathsf{ap_{pre}}\ \mathsf{h}_2\ \mathsf{B})$$
$$\mathsf{in}\ \ \langle \mathsf{Y}', \mathsf{p} \rangle$$

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

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

Fig. 3: Lazy preimage mappings and operations.

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

$$\begin{aligned} &\mathsf{pre} : (\mathsf{X} \rightsquigarrow_\perp \mathsf{Y}) \Rightarrow \mathsf{Set\ X} \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y}) \\ &\mathsf{pre\ f\ A} := \langle \mathsf{image}_\perp\ \mathsf{f\ A}, \lambda \mathsf{B}.\, \mathsf{preimage}_\perp\ \mathsf{f\ A\ B} \rangle \end{aligned} \tag{17}$$

Fig. 3 defines $\mathsf{image}_\perp$, $\mathsf{preimage}_\perp$, and further operations on preimage mappings. One is $\mathsf{ap_{pre}}$, which applies a preimage mapping to any subset of its codomain, in a way that ensures using pre and $\mathsf{ap_{pre}}$ to compute preimages is the same as computing them using $\mathsf{preimage}_\perp$.

Theorem 3 ($\mathsf{ap_{pre}}$ computes preimages). *Let* $\mathsf{f} \in \mathsf{X} \Rightarrow \mathsf{Y}$. *For all* $\mathsf{A} \subseteq \mathsf{X}$ *and* $\mathsf{B} \subseteq \mathsf{Y}$, $\mathsf{ap_{pre}}\ (\mathsf{pre\ f\ A})\ \mathsf{B} \equiv \mathsf{preimage}_\perp\ \mathsf{f\ A\ B}$.

Proof. Expand definitions; use basic facts about $(\backslash)$, $(\cap)$ and image. □

Other operations are $\langle \cdot, \cdot \rangle_{\mathsf{pre}}$, which returns preimage mappings for computing preimages under pairing functions, and $(\circ_{\mathsf{pre}})$ and $(\uplus_{\mathsf{pre}})$, which do the same for compositions and disjoint unions.

4.2 The Preimage Arrow

Now we can define an arrow that computes preimages of output sets.

Its computations should produce preimage mappings or be preimage mappings. However, we cannot have the latter (i.e. $\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y} ::= \mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y}$): we run into

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

$$\mathsf{arr}_{\mathsf{pre}} : (\mathsf{X} \Rightarrow \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y})$$
$$\mathsf{arr}_{\mathsf{pre}} \ := \ \mathsf{lift}_{\mathsf{pre}} \circ \mathsf{arr}_{\perp}$$

$$(\ggg_{\mathsf{pre}}) : (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}) \Rightarrow (\mathsf{Y} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Z}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Z})$$
$$(\mathsf{h}_1 \ggg_{\mathsf{pre}} \mathsf{h}_2)\ \mathsf{A} \ := \ \mathsf{let}\ \ \mathsf{h}'_1 := \mathsf{h}_1\ \mathsf{A} \quad \mathsf{h}'_2 := \mathsf{h}_2\ (\mathsf{range}_{\mathsf{pre}}\ \mathsf{h}'_1) \quad \mathsf{in}\ \ \mathsf{h}'_2 \circ_{\mathsf{pre}} \mathsf{h}'_1$$

$$(\&\&\&_{\mathsf{pre}}) : (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Z}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y} \times \mathsf{Z})$$
$$(\mathsf{h}_1 \ \&\&\&_{\mathsf{pre}}\ \mathsf{h}_2)\ \mathsf{A} \ := \ \langle \mathsf{h}_1\ \mathsf{A}, \mathsf{h}_2\ \mathsf{A} \rangle_{\mathsf{pre}}$$

$$\mathsf{ifte}_{\mathsf{pre}} : (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Bool}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y})$$
$$\mathsf{ifte}_{\mathsf{pre}}\ \mathsf{h}_1\ \mathsf{h}_2\ \mathsf{h}_3\ \mathsf{A} \ := \ \mathsf{let}\ \ \mathsf{h}'_1 := \mathsf{h}_1\ \mathsf{A} \quad \mathsf{h}'_2 := \mathsf{h}_2\ (\mathsf{ap}_{\mathsf{pre}}\ \mathsf{h}'_1\ \{\mathsf{true}\}) \quad \mathsf{h}'_3 := \mathsf{h}_3\ (\mathsf{ap}_{\mathsf{pre}}\ \mathsf{h}'_1\ \{\mathsf{false}\}) \quad \mathsf{in}\ \ \mathsf{h}'_2 \uplus_{\mathsf{pre}} \mathsf{h}'_3$$

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

$$\mathsf{lift}_{\mathsf{pre}} \ := \ \mathsf{pre}$$

Fig. 4: Preimage arrow definitions.

trouble trying to define $\mathsf{arr}_{\mathsf{pre}}$ because a preimage mapping needs an observable range. Fortunately, if we define the ***preimage arrow*** type constructor as

$$\mathsf{X} \underset{\mathsf{pre}}{\rightrightarrows} \mathsf{Y} \ ::= \ \mathsf{Set}\ \mathsf{X} \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightarrow} \mathsf{Y}) \tag{18}$$

then we already have a lift $\mathsf{lift}_{\mathsf{pre}} : (\mathsf{X} \rightsquigarrow_{\perp} \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y})$ 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. If the arrow combinator definitions make $\mathsf{lift}_{\mathsf{pre}}$ a homomorphism, then $[\![\cdot]\!]_{\mathsf{pre}}$ is correct. For this to be true, we need preimage arrow computations to be equivalent when they compute 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* $\mathsf{ap}_{\mathsf{pre}}\ (\mathsf{h}_1\ \mathsf{A})\ \mathsf{B} \equiv \mathsf{ap}_{\mathsf{pre}}\ (\mathsf{h}_2\ \mathsf{A})\ \mathsf{B}$ *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, $[\![e]\!]_{\mathsf{pre}} \equiv \mathsf{lift}_{\mathsf{pre}}\ [\![e]\!]_{\perp}$.

While lifted bottom arrow computations behave intuitively, preimage arrow computations in general can be unruly. For example:

$$\begin{aligned} &\mathsf{unruly} : \mathsf{Bool} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Bool} \\ &\mathsf{unruly}\ \mathsf{A} \ := \ \langle \mathsf{Bool} \backslash \mathsf{A}, \lambda \mathsf{B}.\, \mathsf{B} \rangle \end{aligned} \tag{19}$$

With this, $\mathsf{ap}_{\mathsf{pre}}\ (\mathsf{unruly}\ \{\mathsf{true}\})\ \{\mathsf{false}\} = \{\mathsf{false}\} \cap (\mathsf{Bool} \backslash \{\mathsf{true}\}) = \{\mathsf{false}\}$—a "preimage" that does not even intersect the given domain $\{\mathsf{true}\}$. We would like to be sure each $\mathsf{h} : \mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}$ always acts as if it computes preimages under some bottom arrow computation.

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

We assume from here on that the preimage arrow law holds for all $\mathsf{h} : \mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}$. By homomorphism of $\mathsf{lift}_{\mathsf{pre}}$, preimage arrow combinators return computations that obey this law. The preimage arrow law implies $\mathsf{lift}_{\mathsf{pre}}$ is an epimorphism; by Theorem 1, the arrow laws hold.

5 Preimages Under Partial Functions

We have defined the top of our roadmap:

$$\begin{array}{ccc} \mathsf{X} \rightsquigarrow_{\perp} \mathsf{Y} & \xrightarrow{\mathsf{lift}_{\mathsf{pre}}} & \mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y} \\ {\scriptstyle \eta_{\perp^*}}\downarrow & & \downarrow{\scriptstyle \eta_{\mathsf{pre}^*}} \\ \mathsf{X} \rightsquigarrow_{\perp^*} \mathsf{Y} & \xrightarrow[\mathsf{lift}_{\mathsf{pre}^*}]{} & \mathsf{X} \underset{\mathsf{pre}^*}{\rightsquigarrow} \mathsf{Y} \end{array} \tag{20}$$

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.

5.1 Motivation

Probabilistic functions that may not terminate, but terminate with probability 1, are common. They come up not only when practitioners want to build data with random size or structure, but in simpler circumstances as well.

Suppose random retrieves a number $\mathsf{r}\ \mathsf{j} \in [0, 1]$ at index j in an implicit random source r. The following function, which defines the well-known **geometric distribution** with parameter p, counts the number of times $\mathsf{random} < \mathsf{p}$ is false:

$$\mathsf{geometric}\ \mathsf{p}\ :=\ \mathsf{if}\ (\mathsf{random} < \mathsf{p})\ 0\ (1 + \mathsf{geometric}\ \mathsf{p}) \tag{21}$$

For any $\mathsf{p} > 0$, $\mathsf{geometric}\ \mathsf{p}$ may not terminate, but the probability of always taking the false branch is $(1 - \mathsf{p}) \times (1 - \mathsf{p}) \times (1 - \mathsf{p}) \times \cdots = 0$. Therefore, for $\mathsf{p} > 0$, $\mathsf{geometric}\ \mathsf{p}$ terminates with probability 1.

Suppose we interpret (21) as $\mathsf{h} : \mathsf{R} \underset{\mathsf{pre}}{\rightsquigarrow} \mathbb{N}$, a preimage arrow computation from random sources in R to natural numbers, and that we have a probability measure $\mathsf{P} \in \mathcal{P}\ \mathsf{R} \rightharpoonup [0, 1]$. We could compute the probability of any output set $\mathsf{N} \subseteq \mathbb{N}$ using $\mathsf{P}\ (\mathsf{h}\ \mathsf{R}'\ \mathsf{N})$, where $\mathsf{R}' \subseteq \mathsf{R}$ and $\mathsf{P}\ \mathsf{R}' = 1$. We have three hurdles to overcome:

1. Ensuring $\mathsf{h}\ \mathsf{R}'$ terminates.
2. Ensuring each $\mathsf{r} \in \mathsf{R}$ contains enough random numbers.
3. Determining how random indexes numbers in r.

Ensuring $\mathsf{h}\ \mathsf{R}'$ terminates is the most difficult, but doing the other two will provide structure that makes it much easier.

5.2 Threading and Indexing

We clearly need a new arrow that threads a random source through its computations. To ensure it contains enough random numbers, it 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 new combinator is defined that removes the head of the random source and passes the tail along. This is likely preferred because pseudorandom number generators are almost universally monadic.

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 defined using pairing, the resulting definitions are large, conceptually difficult, and hard to manipulate. Fortunately, assigning each subcomputation a unique index into a tree-shaped random source, and passing the random source unchanged, is relatively easy.

To do this, we need a set of computation indexes.

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 $\mathsf{right}\ \mathsf{j} := \langle 1, \mathsf{j}\rangle$. Alternatively, let J be the set of dyadic rationals in $(0,1)$ (i.e. those with power-of-two denominators), $\mathsf{j}_0 := \frac{1}{2}$ and

$$\begin{aligned} \mathsf{left}\ (\mathsf{p}/\,\mathsf{q}) &:= (\mathsf{p} - \tfrac{1}{2})/\,\mathsf{q} \\ \mathsf{right}\ (\mathsf{p}/\,\mathsf{q}) &:= (\mathsf{p} + \tfrac{1}{2})/\,\mathsf{q} \end{aligned} \tag{22}$$

With this alternative, left-to-right evaluation order can be made to correspond with the natural order $(<)$ over J. 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).

5.3 Applicative, Associative Store Transformer

We thread a random store 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 $\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{y}$:

$$\mathsf{AStore}\ \mathsf{s}\ (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{y}) ::= \mathsf{J} \Rightarrow (\langle \mathsf{s}, \mathsf{x}\rangle \rightsquigarrow_\mathsf{a} \mathsf{y}) \tag{23}$$

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:

$$\begin{aligned} &\eta_{\mathsf{a}^*} : (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{y}) \Rightarrow \mathsf{AStore}\ \mathsf{s}\ (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{y}) \\ &\eta_{\mathsf{a}^*}\ \mathsf{f}\ \mathsf{j} := \mathsf{arr}_\mathsf{a}\ \mathsf{snd} \ggg_\mathsf{a} \mathsf{f} \end{aligned} \tag{24}$$

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

$$\mathsf{arr}_{a^*} : (x \Rightarrow y) \Rightarrow (x \rightsquigarrow_{a^*} y)$$
$$\mathsf{arr}_{a^*} := \eta_{a^*} \circ \mathsf{arr}_a$$

$$(\ggg_{a^*}) : (x \rightsquigarrow_{a^*} y) \Rightarrow (y \rightsquigarrow_{a^*} z) \Rightarrow (x \rightsquigarrow_{a^*} z)$$
$$(k_1 \ggg_{a^*} k_2)\ j :=$$
$$(\mathsf{arr}_a\ \mathsf{fst} \mathbin{\&\&\&_a} k_1\ (\mathsf{left}\ j)) \ggg_a k_2\ (\mathsf{right}\ j)$$

$$(\mathbin{\&\&\&_{a^*}}) : (x \rightsquigarrow_{a^*} y_1) \Rightarrow (x \rightsquigarrow_{a^*} y_2) \Rightarrow (x \rightsquigarrow_{a^*} \langle y_1, y_2\rangle)$$
$$(k_1 \mathbin{\&\&\&_{a^*}} k_2)\ j := k_1\ (\mathsf{left}\ j) \mathbin{\&\&\&_a} k_2\ (\mathsf{right}\ j)$$

$$\mathsf{ifte}_{a^*} : (x \rightsquigarrow_{a^*} \mathsf{Bool}) \Rightarrow (x \rightsquigarrow_{a^*} y) \Rightarrow (x \rightsquigarrow_{a^*} y) \Rightarrow (x \rightsquigarrow_{a^*} y)$$
$$\mathsf{ifte}_{a^*}\ k_1\ k_2\ k_3\ j :=$$
$$\mathsf{ifte}_a\ (k_1\ (\mathsf{left}\ j))\ (k_2\ (\mathsf{left}\ (\mathsf{right}\ j)))\ (k_3\ (\mathsf{right}\ (\mathsf{right}\ j)))$$

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

$$\eta_{a^*} : (x \rightsquigarrow_a y) \Rightarrow (x \rightsquigarrow_{a^*} y)$$
$$\eta_{a^*}\ f\ j := \mathsf{arr}_a\ \mathsf{snd} \ggg_a f$$

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

Because f never accesses the store, j is ignored.

Fig. 5 defines the remaining combinators. Each subcomputation receives left j, 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.

5.4 Partial, Probabilistic Programs

We interpret partial and probabilistic programs using combinators that read a store at an expression index.

Probabilitic Programs. To interpret probabilitic programs, we use a tree-shaped random source as the store.

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

Let $x \rightsquigarrow_{a^*} y ::= \mathsf{AStore}\ R\ (x \rightsquigarrow_a y)$. We define a combinator random_{a^*} that returns the number at its tree index in the random source, and extend the let-calculus for arrows a^* for which random_{a^*} is defined:

$$\begin{aligned} &\mathsf{random}_{a^*} : x \rightsquigarrow_{a^*} [0,1] \\ &\mathsf{random}_{a^*}\ j := \mathsf{arr}_a\ (\mathsf{fst} \ggg \pi\ j) \\ &[\![\mathsf{random}]\!]_{a^*} :\equiv \mathsf{random}_{a^*} \end{aligned} \tag{25}$$

Partial Programs. One utimately implementable way to ensure termination is to have the store dictate which branch of each conditional, if any, can be taken.

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

Let $\mathsf{T} \subset \mathsf{J} \to \mathsf{Bool}_\bot$ be the set of all branch traces, and $\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y} ::= \mathsf{AStore}\ \mathsf{T}\ (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y})$. The following combinator returns the branch $\mathsf{t}\ \mathsf{j}$:

$$\begin{aligned} &\mathsf{branch}_{\mathsf{a}^*} : \mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{Bool} \\ &\mathsf{branch}_{\mathsf{a}^*}\ \mathsf{j} \ :=\ \mathsf{arr}_{\mathsf{a}}\ (\mathsf{fst} \ggg \pi\ \mathsf{j}) \end{aligned} \tag{26}$$

Using $\mathsf{branch}_{\mathsf{a}^*}$, we define an if-then-else combinator that ensures its test expression agrees with the branch trace:

$$\begin{aligned} &\mathsf{agrees} : \langle \mathsf{Bool}, \mathsf{Bool} \rangle \Rightarrow \mathsf{Bool}_\bot \\ &\mathsf{agrees}\ \langle \mathsf{b}_1, \mathsf{b}_2 \rangle \ :=\ \mathsf{if}\ (\mathsf{b}_1 = \mathsf{b}_2)\ \mathsf{b}_1\ \bot \end{aligned} \tag{27}$$

$$\begin{aligned} &\mathsf{ifte}^{\Downarrow}_{\mathsf{a}^*} : (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{Bool}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y}) \\ &\mathsf{ifte}^{\Downarrow}_{\mathsf{a}^*}\ \mathsf{k}_1\ \mathsf{k}_2\ \mathsf{k}_3\ \mathsf{j} \ :=\ \mathsf{ifte}_{\mathsf{a}}\ ((\mathsf{k}_1\ (\mathsf{left}\ \mathsf{j})\ \&\&\&_{\mathsf{a}}\ \mathsf{branch}_{\mathsf{a}^*}\ \mathsf{j}) \ggg_{\mathsf{a}} \mathsf{arr}_{\mathsf{a}}\ \mathsf{agrees}) \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad (\mathsf{k}_2\ (\mathsf{left}\ (\mathsf{right}\ \mathsf{j}))) \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad (\mathsf{k}_3\ (\mathsf{right}\ (\mathsf{right}\ \mathsf{j}))) \end{aligned} \tag{28}$$

If the branch trace agrees with the test expression, it computes a branch; otherwise, it returns an error.

Because we assume every expression is well-defined (Definition 5), every expression must have its recurrences guarded by if. Thus, to ensure running their interpretations always terminates, we should only need to replace $\mathsf{ifte}_{\mathsf{a}^*}$ with $\mathsf{ifte}^{\Downarrow}_{\mathsf{a}^*}$. We define a new semantic function $[\![\cdot]\!]^{\Downarrow}_{\mathsf{a}^*}$ by

$$[\![\mathsf{if}\ e_c\ e_t\ e_f]\!]^{\Downarrow}_{\mathsf{a}^*} \ :\equiv\ \mathsf{ifte}^{\Downarrow}_{\mathsf{a}^*}\ [\![e_c]\!]^{\Downarrow}_{\mathsf{a}^*}\ [\![\mathsf{lazy}\ e_t]\!]^{\Downarrow}_{\mathsf{a}^*}\ [\![\mathsf{lazy}\ e_f]\!]^{\Downarrow}_{\mathsf{a}^*} \tag{29}$$

with the remaining rules similar to those of $[\![\cdot]\!]_{\mathsf{a}^*}$.

For an AStore computation k, we obviously must run k on every branch trace in T and filter out $\bot$, or somehow find pairs of $\langle \mathsf{t}, \mathsf{a} \rangle$ (with $\mathsf{a} : \mathsf{x}$) for which agrees never returns $\bot$. Mapping and preimage AStore arrow computations do both.

Partial, Probabilistic Programs. Let $\mathsf{S} ::= \mathsf{R} \times \mathsf{T}$ be the set of stores. Define $\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y} ::= \mathsf{AStore}\ \mathsf{S}\ (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y})$, and update the $\mathsf{random}_{\mathsf{a}^*}$ and $\mathsf{branch}_{\mathsf{a}^*}$ combinators to reflect that stores are now pairs:

$$\begin{aligned} \mathsf{random}_{\mathsf{a}^*}\ \mathsf{j} \ &:=\ \mathsf{arr}_{\mathsf{a}}\ (\mathsf{fst} \ggg \mathsf{fst} \ggg \pi\ \mathsf{j}) \\ \mathsf{branch}_{\mathsf{a}^*}\ \mathsf{j} \ &:=\ \mathsf{arr}_{\mathsf{a}}\ (\mathsf{fst} \ggg \mathsf{snd} \ggg \pi\ \mathsf{j}) \end{aligned} \tag{30}$$

The definitions of $\mathsf{ifte}^{\Downarrow}_{\mathsf{a}^*}$ and $[\![\cdot]\!]^{\Downarrow}_{\mathsf{a}^*}$ remain the same.

Definition 12 (terminating, probabilistic arrows). *Define the type constructors for the* ***bottom* arrow*** *and the* ***preimage* arrow*** *as*

$$\begin{aligned} \mathsf{X} \rightsquigarrow_{\bot^*} \mathsf{Y} \ &::=\ \mathsf{AStore}\ (\mathsf{R} \times \mathsf{T})\ (\mathsf{X} \rightsquigarrow_{\bot} \mathsf{Y}) \\ \mathsf{X} \underset{\mathsf{pre}^*}{\rightsquigarrow} \mathsf{Y} \ &::=\ \mathsf{AStore}\ (\mathsf{R} \times \mathsf{T})\ (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}) \end{aligned} \tag{31}$$

A bottom* arrow computation's domain is $(\mathsf{R} \times \mathsf{T}) \times \mathsf{X}$. We assume each $\mathsf{r} \in \mathsf{R}$ is randomly chosen, but not each $\mathsf{t} \in \mathsf{T}$ nor $\mathsf{a} \in \mathsf{X}$; therefore, neither T nor X should affect the probabilities of output sets.

5.5 Correctness

We have two arrow lifts to prove homomorphic: one from pure computations to effectful (i.e. from those that do not access the store to those that do), 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*,* $[\![e]\!]_{\mathsf{a}^*} \equiv \eta_{\mathsf{a}^*}\ [\![e]\!]_{\mathsf{a}}$*.*

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

$$\begin{aligned} &\mathsf{lift}_{\mathsf{b}^*} : (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{b}^*} \mathsf{y}) \\ &\mathsf{lift}_{\mathsf{b}^*}\ \mathsf{f}\ \mathsf{j}\ :=\ \mathsf{lift}_{\mathsf{b}}\ (\mathsf{f}\ \mathsf{j}) \end{aligned} \tag{32}$$

where $\mathsf{lift}_{\mathsf{b}} : (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{b}} \mathsf{y})$.

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*,* $[\![e]\!]_{\mathsf{pre}^*} \equiv \mathsf{lift}_{\mathsf{pre}^*}\ [\![e]\!]_{\perp^*}$ *and* $[\![e]\!]^{\Downarrow}_{\mathsf{pre}^*} \equiv \mathsf{lift}_{\mathsf{pre}^*}\ [\![e]\!]^{\Downarrow}_{\perp^*}$*.*

5.6 Termination

To relate $[\![e]\!]^{\Downarrow}_{\mathsf{a}^*}$ computations to $[\![e]\!]_{\mathsf{a}^*}$ computations, we need to find the largest domain on which they should agree.

$$\begin{aligned} &\mathsf{domain}_{\perp} : (\mathsf{X} \rightsquigarrow_{\perp} \mathsf{Y}) \Rightarrow \mathsf{Set\ X} \Rightarrow \mathsf{Set\ X} \\ &\mathsf{domain}_{\perp}\ \mathsf{f}\ \mathsf{A}\ :=\ \{\mathsf{a} \in \mathsf{A} \mid \mathsf{f\ a} \neq \perp\} \end{aligned} \tag{33}$$

Definition 14 (maximal domain). *A computation's* ***maximal domain*** *is the largest* A^* *for which*

- *For* $\mathsf{f} : \mathsf{X} \rightsquigarrow_{\perp} \mathsf{Y}$*,* $\mathsf{domain}_{\perp}\ \mathsf{f}\ \mathsf{A}^* = \mathsf{A}^*$*.*
- *For* $\mathsf{h} : \mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}$*,* $\mathsf{domain}_{\mathsf{pre}}\ (\mathsf{h}\ \mathsf{A}^*) = \mathsf{A}^*$*.*

The maximal domain of $\mathsf{k} : \mathsf{X} \rightsquigarrow_{\mathsf{a}^*} \mathsf{Y}$ *is that of* $\mathsf{k}\ \mathsf{j}_0$*.*

Because the above statements imply termination, A^* is a subset of the largest domain for which the computations terminate. Lifting computations preserves the maximal domain; e.g. the maximal domain of $\mathsf{lift}_{\mathsf{pre}^*}\ \mathsf{f}$ is the same as f's.

$$\mathsf{id}_{\mathsf{pre}}\ \mathsf{A} := \langle \mathsf{A}, \lambda \mathsf{B}.\, \mathsf{B} \rangle$$
$$\mathsf{fst}_{\mathsf{pre}}\ \mathsf{A} := \langle \mathsf{proj}_1\ \mathsf{A}, \mathsf{unproj}_1\ \mathsf{A} \rangle$$
$$\mathsf{snd}_{\mathsf{pre}}\ \mathsf{A} := \langle \mathsf{proj}_2\ \mathsf{A}, \mathsf{unproj}_2\ \mathsf{A} \rangle$$

$$\mathsf{const}_{\mathsf{pre}}\ \mathsf{b}\ \mathsf{A} := \langle \{\mathsf{b}\}, \lambda \mathsf{B}.\, \mathsf{if}\ (\mathsf{B} = \varnothing)\ \varnothing\ \mathsf{A} \rangle$$
$$\pi_{\mathsf{pre}}\ \mathsf{j}\ \mathsf{A} := \langle \mathsf{proj}\ \mathsf{j}\ \mathsf{A}, \mathsf{unproj}\ \mathsf{j}\ \mathsf{A} \rangle$$

$$\mathsf{proj}_1 := \mathsf{image}\ \mathsf{fst}; \quad \mathsf{proj}_2 := \mathsf{image}\ \mathsf{snd}$$

$$\mathsf{unproj}_1 : \mathsf{Set}\ \langle \mathsf{X}_1, \mathsf{X}_2 \rangle \Rightarrow \mathsf{Set}\ \mathsf{X}_1 \Rightarrow \mathsf{Set}\ \langle \mathsf{X}_1, \mathsf{X}_2 \rangle$$
$$\mathsf{unproj}_1\ \mathsf{A}\ \mathsf{B} := \mathsf{preimage}\ (\mathsf{mapping}\ \mathsf{fst}\ \mathsf{A})\ \mathsf{B}$$
$$\equiv\ \mathsf{A} \cap (\mathsf{B} \times \mathsf{proj}_2\ \mathsf{A})$$

$$\mathsf{proj} : \mathsf{J} \Rightarrow \mathsf{Set}\ (\mathsf{J} \to \mathsf{X}) \Rightarrow \mathsf{Set}\ \mathsf{X}$$
$$\mathsf{proj}\ \mathsf{j}\ \mathsf{A} := \mathsf{image}\ (\pi\ \mathsf{j})\ \mathsf{A}$$

$$\mathsf{unproj} : \mathsf{J} \Rightarrow \mathsf{Set}\ (\mathsf{J} \to \mathsf{X}) \Rightarrow \mathsf{Set}\ \mathsf{X} \Rightarrow \mathsf{Set}\ (\mathsf{J} \to \mathsf{X})$$
$$\mathsf{unproj}\ \mathsf{j}\ \mathsf{A}\ \mathsf{B} := \mathsf{preimage}\ (\mathsf{mapping}\ (\pi\ \mathsf{j})\ \mathsf{A})\ \mathsf{B}$$
$$\equiv\ \mathsf{A} \cap \textstyle\prod_{\mathsf{i} \in \mathsf{J}} \mathsf{if}\ (\mathsf{j} = \mathsf{i})\ \mathsf{B}\ (\mathsf{proj}\ \mathsf{j}\ \mathsf{A})$$

Fig. 6: Preimage arrow lifts needed to interpret probabilistic programs. The definition of unproj_2 is like unproj_1's.

Theorem 7 (correct computation everywhere). *Let* $[\![e]\!]^{\Downarrow}_{\perp^*} : \mathsf{X} \rightsquigarrow_{\perp^*} \mathsf{Y}$ *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* $\mathsf{B} \subseteq \mathsf{Y}$*,*

$$\begin{aligned} [\![e]\!]^{\Downarrow}_{\perp^*}\ \mathsf{j}_0\ \mathsf{a} &= \mathsf{if}\ (\mathsf{a} \in \mathsf{A}^*)\ ([\![e]\!]_{\perp^*}\ \mathsf{j}_0\ \mathsf{a})\ \perp \\ \mathsf{ap}_{\mathsf{pre}}\ ([\![e]\!]^{\Downarrow}_{\mathsf{pre}^*}\ \mathsf{j}_0\ \mathsf{A})\ \mathsf{B} &= \mathsf{ap}_{\mathsf{pre}}\ ([\![e]\!]_{\mathsf{pre}^*}\ \mathsf{j}_0\ (\mathsf{A} \cap \mathsf{A}^*))\ \mathsf{B} \end{aligned} \tag{34}$$

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.

Let $\mathsf{f} : [\![e]\!]^{\Downarrow}_{\perp^*} : \mathsf{X} \rightsquigarrow_{\perp^*} \mathsf{Y}$

Assuming correctness, the probability of an output set $\mathsf{B} \subseteq \mathsf{Y}$ is

$$\mathsf{P}\ (\mathsf{image}\ (\mathsf{fst} \ggg \mathsf{fst})\ (\mathsf{preimage}_{\perp}\ \mathsf{f}\ \mathsf{A}\ \mathsf{B}) \tag{35}$$

6 Approximating Semantics

If we were to confine preimage computation to finite sets, we could implement the preimage arrow directly. But we would like something that works efficiently on infinite sets, even if it means approximating. We focus on a specific method: approximating product sets with covering rectangles.

6.1 Implementable Lifts

We would like to be able to compute preimages of uncountable sets, such as real intervals. This would seem to be a show-stopper: $\mathsf{preimage}\ \mathsf{g}\ \mathsf{B}$ is uncomputable for most uncountable sets B no matter how cleverly they are represented. Further, because pre, $\mathsf{lift}_{\mathsf{pre}}$ and $\mathsf{arr}_{\mathsf{pre}}$ are ultimately defined in terms of $\mathsf{preimage}$, we cannot implement them.

Fortunately, we need only certain lifts. Fig. 1 (which defines $[\![\cdot]\!]_{\mathsf{a}}$) lifts id, $\mathsf{const}\ \mathsf{b}$, fst and snd. Section 5.4, which defines the combinators used to interpret

partial, probabilistic programs, lifts π j and agrees. Measurable functions made available as language primitives of course must be lifted to the preimage arrow.

Fig. 6 gives explicit definitions for $\mathsf{arr}_{\mathsf{pre}}$ id, $\mathsf{arr}_{\mathsf{pre}}$ fst, $\mathsf{arr}_{\mathsf{pre}}$ snd, $\mathsf{arr}_{\mathsf{pre}}$ (const b) and $\mathsf{arr}_{\mathsf{pre}}$ (π j). (We will deal with agrees separately.) By inspecting these expressions, we see that we need to model sets in a way that the following are representable and can be computed in finite time:

$$\begin{aligned}
&\bullet\ \mathsf{A} \cap \mathsf{B},\ \varnothing,\ \{\mathsf{true}\},\ \{\mathsf{false}\} \text{ and } \{\mathsf{b}\} \text{ for every const b} \\
&\bullet\ \mathsf{A}_1 \times \mathsf{A}_2,\ \mathsf{proj}_1\ \mathsf{A} \text{ and } \mathsf{proj}_2\ \mathsf{A} \\
&\bullet\ \mathsf{J} \to \mathsf{X},\ \mathsf{proj\ j\ A} \text{ and } \mathsf{unproj\ j\ A\ B} \\
&\bullet\ \mathsf{A} = \varnothing
\end{aligned} \tag{36}$$

Before addressing computability, we need to define families of sets under which these operations are closed.

Definition 15 (rectangular family). *For a set* X *used as a type,* Rect X *denotes the* ***rectangular family*** *of subsets of* X*. For nonproduct* X*,* $\varnothing \in$ Rect X *and* X $\in$ Rect X*, and* Rect X *must be closed under finite intersections. Products must satisfy the following rules:*

$$\mathsf{Rect}\ \langle \mathsf{X}_1, \mathsf{X}_2 \rangle \ =\ (\mathsf{Rect}\ \mathsf{X}_1) \boxtimes (\mathsf{Rect}\ \mathsf{X}_2) \tag{37}$$

$$\mathsf{Rect}\ (\mathsf{J} \to \mathsf{X}) \ =\ (\mathsf{Rect}\ \mathsf{X})^{\boxtimes \mathsf{J}} \tag{38}$$

where

$$\mathcal{A}_1 \boxtimes \mathcal{A}_2 \ :=\ \{\mathsf{A}_1 \times \mathsf{A}_2 \mid \mathsf{A}_1 \in \mathcal{A}_1, \mathsf{A}_2 \in \mathcal{A}_2\} \tag{39}$$

$$\mathcal{A}^{\boxtimes \mathsf{J}} \ :=\ \bigcup_{\mathsf{J}' \subset \mathsf{J}\ \mathit{finite}} \left\{ \prod_{\mathsf{j} \in \mathsf{J}} \mathsf{A}_\mathsf{j} \;\middle|\; \mathsf{A}_\mathsf{j} \in \mathcal{A}, \mathsf{j} \in \mathsf{J}' \iff \mathsf{A}_\mathsf{j} \subset \bigcup \mathcal{A} \right\} \tag{40}$$

lift cartesian products to sets of sets.

For example, if Rect $\mathbb{R}$ contains all the closed real intervals, then by (37), $[0, 2] \times [1, \pi] \in$ Rect $\langle \mathbb{R}, \mathbb{R} \rangle$.

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

Further, all of the operations in (36) 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 Rect $\langle \mathsf{X}_1, \mathsf{X}_2 \rangle$ are modeled by pairs of type $\langle \mathsf{Rect}\ \mathsf{X}_1, \mathsf{Rect}\ \mathsf{X}_2 \rangle$. By (40), sets in Rect (J $\to$ X) have no more than finitely many projections that are proper subsets of X. They can be modeled by *finite* binary trees, whose nodes contain projections for an index prefix J′ $\subset$ J (Definition **??**). Projections with indexes in the suffix J$\backslash$J′ are implicitly X.

The set of branch traces T is nonrectangular, containing every t $\in$ J $\to$ Bool$_\perp$ for which t j $\neq \perp$ for no more than finitely many J. Fortunately, we can model T subsets by J $\to$ Bool$_\perp$ rectangles, implicitly intersected with T.

Theorem 8 (rectangular T projection). *If* $\mathsf{T}' \in \mathsf{Rect}\ (\mathsf{J} \to \mathsf{Bool}_\bot)$, *then* $\mathsf{proj\ j}\ (\mathsf{T}' \cap \mathsf{T}) = \mathsf{proj\ j}\ \mathsf{T}'$ *for all* $\mathsf{j} \in \mathsf{J}$. *Also, for all* $\mathsf{B} \subseteq \mathsf{Bool}$, $\mathsf{unproj\ j}\ (\mathsf{T}' \cap \mathsf{T})\ \mathsf{B} = \mathsf{unproj\ j}\ \mathsf{T}'\ \mathsf{B} \cap \mathsf{T}$.

6.2 Approximate Preimage Mapping Operations

Implementing $\mathsf{lazy}_{\mathsf{pre}}$ (defined in Fig. 4) requires computing pre, but only for the empty mapping, which is trivial: $\mathsf{pre}\ \varnothing \equiv \langle \varnothing, \lambda\,\mathsf{B}.\,\varnothing \rangle$. Implementing the other combinators requires $(\circ_{\mathsf{pre}})$, $\langle \cdot, \cdot \rangle_{\mathsf{pre}}$ and $(\uplus_{\mathsf{pre}})$.

From the preimage mapping definitions (Fig. 3), we see that $\mathsf{ap}_{\mathsf{pre}}$ is defined using $(\cap)$ and that $(\circ_{\mathsf{pre}})$ is defined using $\mathsf{ap}_{\mathsf{pre}}$, so $(\circ_{\mathsf{pre}})$ is directly implementable. Unfortunately, we hit a snag with $\langle \cdot, \cdot \rangle_{\mathsf{pre}}$: it loops over possibly uncountably many members of B in a big union. At this point, we need to approximate.

Theorem 9 (pair preimage approximation). *Let* $\mathsf{g}_1 \in \mathsf{X} \rightharpoonup \mathsf{Y}_1$ *and* $\mathsf{g}_2 \in \mathsf{X} \rightharpoonup \mathsf{Y}_2$. *For all* $\mathsf{B} \subseteq \mathsf{Y}_1 \times \mathsf{Y}_2$, $\mathsf{preimage}\ \langle \mathsf{g}_1, \mathsf{g}_2 \rangle_{\mathsf{map}}\ \mathsf{B} \subseteq \mathsf{preimage}\ \mathsf{g}_1\ (\mathsf{proj}_1\ \mathsf{B}) \cap \mathsf{preimage}\ \mathsf{g}_2\ (\mathsf{proj}_2\ \mathsf{B})$.

It is not hard to use Theorem 9 to show that

$$\begin{aligned} &\langle \cdot, \cdot \rangle'_{\mathsf{pre}} : (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y}_1) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y}_2) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y}_1 \times \mathsf{Y}_2) \\ &\langle \langle \mathsf{Y}'_1, \mathsf{p}_1 \rangle, \langle \mathsf{Y}'_2, \mathsf{p}_2 \rangle \rangle'_{\mathsf{pre}} \ := \\ &\quad \langle \mathsf{Y}'_1 \times \mathsf{Y}'_2, \lambda\,\mathsf{B}.\,\mathsf{p}_1\ (\mathsf{proj}_1\ \mathsf{B}) \cap \mathsf{p}_2\ (\mathsf{proj}_2\ \mathsf{B}) \rangle \end{aligned} \tag{41}$$

computes covering rectangles of preimages under pairing.

For $(\uplus_{\mathsf{pre}})$, we need an approximating replacement for $(\cup)$ under which rectangular families are closed. In other words, we need a lattice join $(\vee)$ with respect to $(\subseteq)$, with the following additional properties:

$$\begin{aligned} (\mathsf{A}_1 \times \mathsf{A}_2) \vee (\mathsf{B}_1 \times \mathsf{B}_2) &= (\mathsf{A}_1 \vee \mathsf{B}_1) \times (\mathsf{A}_2 \vee \mathsf{B}_2) \\ (\textstyle\prod_{\mathsf{j} \in \mathsf{J}} \mathsf{A}_\mathsf{j}) \vee (\textstyle\prod_{\mathsf{j} \in \mathsf{J}} \mathsf{B}_\mathsf{j}) &= \textstyle\prod_{\mathsf{j} \in \mathsf{J}} \mathsf{A}_\mathsf{j} \vee \mathsf{B}_\mathsf{j} \end{aligned} \tag{42}$$

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

Replacing each union in $(\uplus_{\mathsf{pre}})$ with join yields the overapproximating $(\uplus'_{\mathsf{pre}})$:

$$\begin{aligned} &(\uplus'_{\mathsf{pre}}) : (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y}) \\ &\mathsf{h}_1 \uplus'_{\mathsf{pre}} \mathsf{h}_2 \ := \ \mathsf{let}\ \ \mathsf{Y}' := \mathsf{range}_{\mathsf{pre}}\ \mathsf{h}_1 \vee \mathsf{range}_{\mathsf{pre}}\ \mathsf{h}_2 \\ &\qquad\qquad\qquad\quad \mathsf{p} := \lambda\,\mathsf{B}.\,\mathsf{ap}_{\mathsf{pre}}\ \mathsf{h}_1\ \mathsf{B} \vee \mathsf{ap}_{\mathsf{pre}}\ \mathsf{h}_2\ \mathsf{B} \\ &\qquad\qquad\qquad\ \ \mathsf{in}\ \ \langle \mathsf{Y}', \mathsf{p} \rangle \end{aligned} \tag{43}$$

To interpret programs that may not terminate, or that terminate with probability 1, we need to approximate $\mathsf{ife}^{\Downarrow}_{\mathsf{pre}^*}$ (28), which is defined in terms of agrees. Defining its approximation in terms of an approximation of agrees would not allow us to preserve the fact that expressions interpreted using $\mathsf{ife}^{\Downarrow}_{\mathsf{pre}^*}$ always terminate. The best approximation of the preimage of Bool under agrees (as a

mapping) is $\mathsf{Bool} \times \mathsf{Bool}$, which contains $\langle \mathsf{true}, \mathsf{false} \rangle$ and $\langle \mathsf{false}, \mathsf{true} \rangle$, and thus would not constrain the test to agree with the branch trace.

A lengthy (elided) sequence of substitutions to the defining expression for $\mathsf{ifte}^{\Downarrow}_{\mathsf{pre}^*}$ results in an agrees-free equivalence:

$$\begin{aligned} \mathsf{ifte}^{\Downarrow}_{\mathsf{pre}^*}\ \mathsf{k}_1\ \mathsf{k}_2\ \mathsf{k}_3\ \mathsf{j}\ \mathsf{A} \ \equiv\ \mathsf{let}\ \ \langle \mathsf{C_k}, \mathsf{p_k} \rangle &:= \mathsf{k}_1\ \mathsf{j}_1\ \mathsf{A} \\ \langle \mathsf{C_b}, \mathsf{p_b} \rangle &:= \mathsf{branch}_{\mathsf{pre}^*}\ \mathsf{j}\ \mathsf{A} \\ \mathsf{C}_2 &:= \mathsf{C_k} \cap \mathsf{C_b} \cap \{\mathsf{true}\} \\ \mathsf{C}_3 &:= \mathsf{C_k} \cap \mathsf{C_b} \cap \{\mathsf{false}\} \\ \mathsf{A}_2 &:= \mathsf{p_k}\ \mathsf{C}_2 \cap \mathsf{p_b}\ \mathsf{C}_2 \\ \mathsf{A}_3 &:= \mathsf{p_k}\ \mathsf{C}_3 \cap \mathsf{p_b}\ \mathsf{C}_3 \\ \mathsf{in}\ \ \mathsf{k}_2\ \mathsf{j}_2\ \mathsf{A}_2 &\uplus_{\mathsf{pre}} \mathsf{k}_3\ \mathsf{j}_3\ \mathsf{A}_3 \end{aligned} \tag{44}$$

where $\mathsf{j}_1 = \mathsf{left}\ \mathsf{j}$ and so on. Unfortunately, a straightforward approximation of this would still take unnecessary branches, when A_2 or A_3 overapproximates $\varnothing$.

$\mathsf{C_b}$ is the branch trace projection at j (with $\bot$ removed). The set of indexes for which $\mathsf{C_b}$ is either $\{\mathsf{true}\}$ or $\{\mathsf{false}\}$ is finite, so it is bounded by an index prefix, outside of which branch trace projections are $\{\mathsf{true}, \mathsf{false}\}$. Therefore, if the approximating $\mathsf{ifte}^{\Downarrow}_{\mathsf{pre}^*}{}'$ takes *no branches* when $\mathsf{C_b} = \{\mathsf{true}, \mathsf{false}\}$, but approximates with a finite computation, expressions interpreted using $\mathsf{ifte}^{\Downarrow}_{\mathsf{pre}^*}{}'$ will always terminate.

We need an overapproximation for the non-branching case. In the exact semantics, the returned preimage mapping's range is a subset of Y, and it returns subsets of $\mathsf{A}_2 \uplus \mathsf{A}_3$. Therefore, $\mathsf{ifte}^{\Downarrow}_{\mathsf{pre}^*}{}'$ may return $\langle \mathsf{Y}, \lambda \mathsf{B}.\, \mathsf{A}_2 \vee \mathsf{A}_3 \rangle$ when $\mathsf{C_b} = \{\mathsf{true}, \mathsf{false}\}$. We cannot refer to the type Y in the function definition, so we represent it using $\top$ in the approximating semantics. Implementations can model it by a singleton "universe" instance for every $\mathsf{Rect}\ \mathsf{Y}$.

Fig. 7 defines the final approximating preimage arrow. This arrow, the lifts in Fig. 6, and the semantic function $[\![\cdot]\!]_{\mathsf{a}}$ in Fig. 1 define an approximating semantics for partial, probabilistic programs.

6.3 Correctness

From here on, $[\![\cdot]\!]^{\Downarrow}_{\mathsf{pre}^*}{}'$ interprets programs as approximating preimage* arrow computations using $\mathsf{ifte}^{\Downarrow}_{\mathsf{pre}^*}{}'$. The following theorems assume $\mathsf{h} := [\![e]\!]^{\Downarrow}_{\mathsf{pre}^*} : \mathsf{X} \rightsquigarrow_{\mathsf{pre}^*} \mathsf{Y}$ and $\mathsf{h}' := [\![e]\!]^{\Downarrow}_{\mathsf{pre}^*}{}' : \mathsf{X} \rightsquigarrow_{\mathsf{pre}^*}{}' \mathsf{Y}$ for some expression e.

Theorem 10 (soundness). *For all* $\mathsf{A} \in \mathsf{Rect}\ \langle\langle \mathsf{R}, \mathsf{T} \rangle, \mathsf{X} \rangle$ *and* $\mathsf{B} \in \mathsf{Rect}\ \mathsf{Y}$, $\mathsf{ap}_{\mathsf{pre}}\ (\mathsf{h}\ \mathsf{j}_0\ \mathsf{A})\ \mathsf{B} \subseteq \mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}'\ \mathsf{j}_0\ \mathsf{A})\ \mathsf{B}$.

Theorem 11 (termination). *For all* $\mathsf{A}' \in \mathsf{Rect}\ \langle\langle \mathsf{R}, \mathsf{T} \rangle, \mathsf{X} \rangle$ *and* $\mathsf{B} \in \mathsf{Rect}\ \mathsf{Y}$, $\mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}'\ \mathsf{j}_0\ \mathsf{A}')\ \mathsf{B}$ *terminates.*

Theorem 12 (monotonicity). $\mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}'\ \mathsf{j}_0\ \mathsf{A})\ \mathsf{B}$ *is monotone in both* A *and* B.

Theorem 13 (decreasing). *For all* $\mathsf{A} \in \mathsf{Rect}\ \langle\langle \mathsf{R}, \mathsf{T} \rangle, \mathsf{X} \rangle$ *and* $\mathsf{B} \in \mathsf{Rect}\ \mathsf{Y}$, $\mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}'\ \mathsf{j}_0\ \mathsf{A})\ \mathsf{B} \subseteq \mathsf{A}$.

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

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

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

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

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

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

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

$$(\ggg'_{\mathsf{pre}}) : (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow}' \mathsf{Y}) \Rightarrow (\mathsf{Y} \underset{\mathsf{pre}}{\rightsquigarrow}' \mathsf{Z}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow}' \mathsf{Z})$$
$$(\mathsf{h}_1 \ggg'_{\mathsf{pre}} \mathsf{h}_2)\ \mathsf{A} := \mathsf{let}\ \ \mathsf{h}'_1 := \mathsf{h}_1\ \mathsf{A}$$
$$\qquad\qquad\qquad\qquad\quad \mathsf{h}'_2 := \mathsf{h}_2\ (\mathsf{range}'_{\mathsf{pre}}\ \mathsf{h}'_1)$$
$$\qquad\qquad\qquad\quad \mathsf{in}\ \ \mathsf{h}'_2 \circ'_{\mathsf{pre}} \mathsf{h}'_1$$

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

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

$$\mathsf{lazy}'_{\mathsf{pre}} : (1 \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow}' \mathsf{Y})) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow}' \mathsf{Y})$$
$$\mathsf{lazy}'_{\mathsf{pre}}\ \mathsf{h}\ \mathsf{A} := \mathsf{if}\ (\mathsf{A} = \varnothing)\ \langle \varnothing, \lambda\, \mathsf{B}.\, \varnothing \rangle\ (\mathsf{h}\ 0\ \mathsf{A})$$

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

$$\mathsf{X} \underset{\mathsf{pre}*}{\rightsquigarrow}' \mathsf{Y} ::= \mathsf{J} \Rightarrow (\langle \mathsf{S}, \mathsf{X} \rangle \underset{\mathsf{pre}}{\rightsquigarrow}' \mathsf{Y})$$
$$\qquad \mathsf{S} ::= (\mathsf{J} \to [0,1]) \times (\mathsf{J} \to \mathsf{Bool}_\perp)$$

$$(\ggg'_{\mathsf{pre}*}) : (\mathsf{X} \underset{\mathsf{pre}*}{\rightsquigarrow}' \mathsf{Y}) \Rightarrow (\mathsf{Y} \underset{\mathsf{pre}*}{\rightsquigarrow}' \mathsf{Z}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}*}{\rightsquigarrow}' \mathsf{Z})$$
$$(\mathsf{k}_1 \ggg'_{\mathsf{pre}*} \mathsf{k}_2)\ \mathsf{j} :=$$
$$\quad (\mathsf{fst}_{\mathsf{pre}}\ \&\&\&'_{\mathsf{pre}}\ \mathsf{k}_1\ (\mathsf{left\ j})) \ggg'_{\mathsf{pre}}\ \mathsf{k}_2\ (\mathsf{right\ j})$$

$$(\&\&\&'_{\mathsf{pre}*}) : (\mathsf{X} \underset{\mathsf{pre}*}{\rightsquigarrow}' \mathsf{Y}_1) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}*}{\rightsquigarrow}' \mathsf{Y}_2) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}*}{\rightsquigarrow}' \langle \mathsf{Y}_1, \mathsf{Y}_2 \rangle)$$
$$(\mathsf{k}_1\ \&\&\&'_{\mathsf{pre}*}\ \mathsf{k}_2)\ \mathsf{j} := \mathsf{k}_1\ (\mathsf{left\ j})\ \&\&\&'_{\mathsf{pre}}\ \mathsf{k}_2\ (\mathsf{right\ j})$$

$$\mathsf{ifte}'_{\mathsf{pre}*} : (\mathsf{X} \underset{\mathsf{pre}*}{\rightsquigarrow}' \mathsf{Bool}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}*}{\rightsquigarrow}' \mathsf{Y}) \Rightarrow$$
$$\qquad (\mathsf{X} \underset{\mathsf{pre}*}{\rightsquigarrow}' \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}*}{\rightsquigarrow}' \mathsf{Y})$$
$$\mathsf{ifte}'_{\mathsf{pre}*}\ \mathsf{k}_1\ \mathsf{k}_2\ \mathsf{k}_3\ \mathsf{j} :=$$
$$\quad \mathsf{ifte}'_{\mathsf{pre}}\ (\mathsf{k}_1\ (\mathsf{left\ j}))$$
$$\qquad\qquad (\mathsf{k}_2\ (\mathsf{left}\ (\mathsf{right\ j})))$$
$$\qquad\qquad (\mathsf{k}_3\ (\mathsf{right}\ (\mathsf{right\ j})))$$

$$\mathsf{lazy}'_{\mathsf{pre}*} : (1 \Rightarrow (\mathsf{X} \underset{\mathsf{pre}*}{\rightsquigarrow}' \mathsf{Y})) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}*}{\rightsquigarrow}' \mathsf{Y})$$
$$\mathsf{lazy}'_{\mathsf{pre}*}\ \mathsf{k}\ \mathsf{j} := \mathsf{lazy}'_{\mathsf{pre}}\ \lambda 0.\, \mathsf{k}\ 0\ \mathsf{j}$$

$$\eta'_{\mathsf{pre}*} : (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow}' \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}*}{\rightsquigarrow}' \mathsf{Y})$$
$$\eta'_{\mathsf{pre}*}\ \mathsf{f}\ \mathsf{j} := \mathsf{snd}_{\mathsf{pre}} \ggg'_{\mathsf{pre}}\ \mathsf{f}$$

(c) An approximating preimage* arrow.

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

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

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

$$\mathsf{ifte}^{\Downarrow}{}'_{\mathsf{pre}*} : (\mathsf{X} \underset{\mathsf{pre}*}{\rightsquigarrow}' \mathsf{Bool}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}*}{\rightsquigarrow}' \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}*}{\rightsquigarrow}' \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}*}{\rightsquigarrow}' \mathsf{Y})$$
$$\mathsf{ifte}^{\Downarrow}{}'_{\mathsf{pre}*}\ \mathsf{k}_1\ \mathsf{k}_2\ \mathsf{k}_3\ \mathsf{j} :=$$
$$\quad \mathsf{let}\ \ \langle \mathsf{C}_\mathsf{k}, \mathsf{p}_\mathsf{k} \rangle := \mathsf{k}_1\ (\mathsf{left\ j})\ \mathsf{A}$$
$$\qquad\ \ \langle \mathsf{C}_\mathsf{b}, \mathsf{p}_\mathsf{b} \rangle := \mathsf{branch}_{\mathsf{pre}*}\ \mathsf{j}\ \mathsf{A}$$
$$\qquad\qquad\ \ \mathsf{C}_2 := \mathsf{C}_\mathsf{k} \cap \mathsf{C}_\mathsf{b} \cap \{\mathsf{true}\}$$
$$\qquad\qquad\ \ \mathsf{C}_3 := \mathsf{C}_\mathsf{k} \cap \mathsf{C}_\mathsf{b} \cap \{\mathsf{false}\}$$
$$\qquad\qquad\ \ \mathsf{A}_2 := \mathsf{p}_\mathsf{k}\ \mathsf{C}_2 \cap \mathsf{p}_\mathsf{b}\ \mathsf{C}_2$$
$$\qquad\qquad\ \ \mathsf{A}_3 := \mathsf{p}_\mathsf{k}\ \mathsf{C}_3 \cap \mathsf{p}_\mathsf{b}\ \mathsf{C}_3$$
$$\quad \mathsf{in}\ \ \mathsf{if}\ (\mathsf{C}_\mathsf{b} = \{\mathsf{true}, \mathsf{false}\})$$
$$\qquad\qquad \langle \top, \lambda\_.\, \mathsf{A}_2 \vee \mathsf{A}_3 \rangle$$
$$\qquad\qquad (\mathsf{k}_2\ (\mathsf{left}\ (\mathsf{right\ j}))\ \mathsf{A}_2 \uplus'_{\mathsf{pre}}\ \mathsf{k}_3\ (\mathsf{right}\ (\mathsf{right\ j}))\ \mathsf{A}_3)$$

(d) Preimage* arrow combinators for probabilistic choice and guaranteed termination.

Fig. 7: Implementable arrows that approximate preimage arrows.

6.4 Preimage Refinement Algorithm

It is natural to suppose that we can compute probabilities of preimages of B by computing preimages with respect to increasingly fine discretizations of A.

Definition 16 (preimage refinement algorithm). *Let* $\mathsf{h}' := [\![e]\!]^{\Downarrow}_{\mathsf{pre}^*}{}' : \mathsf{X} \rightsquigarrow_{\mathsf{pre}^*}{}' \mathsf{Y}$, $\mathsf{B} \in \mathsf{Rect}\ \mathsf{Y}$*, and define*

$$\begin{aligned} &\mathsf{refine} : \mathsf{Rect}\ \langle\langle \mathsf{R}, \mathsf{T}\rangle, \mathsf{X}\rangle \Rightarrow \mathsf{Rect}\ \langle\langle \mathsf{R}, \mathsf{T}\rangle, \mathsf{X}\rangle \\ &\mathsf{refine}\ \mathsf{A} := \mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}'\ \mathsf{j}_0\ \mathsf{A})\ \mathsf{B} \end{aligned} \tag{45}$$

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*

$$\begin{aligned} &\mathsf{refine}^* : \mathsf{Set}\ (\mathsf{Rect}\ \langle\langle \mathsf{R}, \mathsf{T}\rangle, \mathsf{X}\rangle) \Rightarrow \mathsf{Set}\ (\mathsf{Rect}\ \langle\langle \mathsf{R}, \mathsf{T}\rangle, \mathsf{X}\rangle) \\ &\mathsf{refine}^*\ \mathcal{A} := \mathsf{image}\ \mathsf{refine}\ \left(\textstyle\bigcup_{\mathsf{A}\in\mathcal{A}} \mathsf{partition}\ \mathsf{A}\right) \end{aligned} \tag{46}$$

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

Theorem 13 (decreasing) guarantees $\mathsf{refine}\ \mathsf{A}$ is never larger than A. Theorem 12 (monotonicity) guarantees refining a *partition* of A never does worse than refining A itself. Theorem 10 (soundness) guarantees the algorithm is **sound**: the preimage of B is always contained in the covering partition refine^* returns.

We would like it to be **complete** in the limit, up to null sets: covering partitions' measures should converge to the true preimage measure. Unfortunately, preimage refinement appears to compute the **Jordan outer measure** of a preimage, which is not always its measure. A counterexample is the expression $\mathsf{rational?}\ \mathsf{random}$, where $\mathsf{rational?}$ returns true when its argument is rational and loops otherwise. (This is definable using a $(\leq)$ primitive.) The preimage of $\{\mathsf{true}\}$ has measure 0, but its Jordan outer measure is 1.

We conjecture that a minimal requirement for preimage refinement's measures to converge is that a program must converge with probability 1. There are certainly other requirements. We leave these and proof of convergence of measures for future work.

For now, we use algorithms that depend only on soundness.

7 Implementations

We have four implementations: one of the exact semantics, two direct implementations of the approximating semantics, and a less direct but more efficient implementation of the approximating semantics, which we call ***Dr. Bayes***.

7.1 Direct Implementations

If sets are restricted to be finite, the arrows used as translation targets in the exact semantics, defined in Figs. **??**, 2, **??**, 3, 4 and 5, can be implemented directly in any practical λ-calculus. Computing exact preimages is very inefficient,

even under the interpretations of very small programs. Still, we have found our Typed Racket [29] implementation useful for finding theorem candidates.

Given a rectangular set library, 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 and Haskell [1]. Both implementations' arrow combinator definitions are almost line-for-line transliterations from the figures.

Making the rectangular set type polymorphic seems to require the equivalent of a typeclass system. In Haskell, it also requires multi-parameter typeclasses or indexed type families [8] to associate set types with the types of their members. Using indexed type families, the only significant differences between the Haskell implementation and the approximating semantics are type contexts, `newtype` wrappers for arrow types, and using `Maybe` types as bottom arrow return types.

Typed Racket has no typeclass system on top of its type system, so the rectangular set type is monomorphic; thus, so are the arrow types. The lack of type variables in the combinator types is the only significant difference between the implementation and the approximating semantics.

All three direct implementations can currently be found at XXX: URL.

7.2 Dr. Bayes

Our main implementation, ***Dr. Bayes***, is written in Typed Racket. It consists of the semantic function $[\![\cdot]\!]_{\mathsf{a}^*}$ from Fig. 1 and its extension $[\![\cdot]\!]^{\Downarrow}_{\mathsf{a}^*}$, the bottom* arrow as defined in Figs. 2 and 5, the approximating preimage and preimage* arrows as defined in Figs. 6 and 7, and algorithms to compute approximate probabilities. We use it to test the feasibility of solving real-world problems by computing approximate preimages.

Dr. Bayes's arrows operate on a monomorphic rectangular set data type. It includes floating-point intervals to overapproximate real intervals, with which we compute approximate preimages under arithmetic and inequalities. Finding the smallest covering rectangle for images and preimages under $\mathsf{add} : \langle \mathbb{R}, \mathbb{R} \rangle \Rightarrow \mathbb{R}$ and other monotone functions is fairly straightforward. For piecewise monotone functions, we distinguish cases using $\mathsf{ifte}_{\mathsf{pre}}$; e.g.

$$\begin{aligned}\mathsf{mul}_{\mathsf{pre}} \;:=\; & \mathsf{ifte}_{\mathsf{pre}}\ (\mathsf{fst}_{\mathsf{pre}} \ggg_{\mathsf{pre}} \mathsf{pos?}_{\mathsf{pre}}) \\ & \quad (\mathsf{ifte}_{\mathsf{pre}}\ (\mathsf{snd}_{\mathsf{pre}} \ggg_{\mathsf{pre}} \mathsf{pos?}_{\mathsf{pre}}) \\ & \qquad \mathsf{mul}^{++}_{\mathsf{pre}} \\ & \qquad (\mathsf{ifte}_{\mathsf{pre}}\ (\mathsf{snd}_{\mathsf{pre}} \ggg_{\mathsf{pre}} \mathsf{neg?}_{\mathsf{pre}})\ \mathsf{mul}^{+-}_{\mathsf{pre}}\ (\mathsf{const}_{\mathsf{pre}}\ 0))) \\ & \quad \ldots\end{aligned} \tag{47}$$

To support data types, the set type includes tagged rectangles; for ad-hoc polymorphism, it includes disjoint unions.

Section 6.4 outlines preimage refinement: a discretization algorithm that seems to converge for programs that halt with probability 1, consisting of repeatedly shrinking and repartitioning a program's domain. We do not use this algorithm directly in Dr. Bayes because it is inefficient. Good accuracy requires fine discretization, which is *exponential* in the number of discretized axes. For

example, a nonrecursive program that contains only 10 uses of random would need to partition 10 axes of R, the set of random sources. Splitting each axis into only 4 disjoint intervals yields a partition of R of size $4^{10} = 1,048,576$.

Fortunately, Bayesian practitioners tend to be satisified with sampling methods, which are usually more efficient than exact methods based on enumeration.

Let $\mathsf{g} : \mathsf{X} \overset{\rightsquigarrow}{\underset{\mathsf{map}}{}} \mathsf{Y}$ be the interpretation of a program as a mapping arrow computation. A Bayesian is primarily interested in the probability of $\mathsf{B}' \subseteq \mathsf{Y}$ given some condition set $\mathsf{B} \subseteq \mathsf{Y}$. This can be approximated using **rejection sampling**. If $\mathsf{A} := \mathsf{preimage}\ (\mathsf{g}\ \mathsf{X})\ \mathsf{B}$ and $\mathsf{A}' := \mathsf{preimage}\ (\mathsf{g}\ \mathsf{X})\ \mathsf{B}'$, and xs is a list of samples from any superset of A that has at least one element in A, then

$$\mathsf{Pr}[\mathsf{B}'|\mathsf{B}] \approx \frac{\mathsf{length}\ (\mathsf{filter}\ (\in \mathsf{A}' \cap \mathsf{A})\ \mathsf{xs})}{\mathsf{length}\ (\mathsf{filter}\ (\in \mathsf{A})\ \mathsf{xs})} \tag{48}$$

where "≈" (rather loosely) denotes convergence as the length of xs increases. The probability that any given element of xs is in A is often extremely small, so it would clearly be best to sample only within A. While we cannot do that, we can easily sample from a partition covering A.

For a fixed number d of uses of random, n samples, and m repartitions that split each rectangle in two, enumerating and sampling from a covering partition has time complexity $O(2^{md} + n)$. Fortunately, we do not have to enumerate the rectangles in the partition: we sample them instead, and sample one value from each rectangle, which is $O(mdn)$.

We cannot directly compute $\mathsf{a} \in \mathsf{A}$ or $\mathsf{a} \in \mathsf{A}' \cap \mathsf{A}$ in (48), but we can use the fact that A and A′ are preimages, and use the interpretation of the program as a bottom arrow computation $\mathsf{f} : \mathsf{X} \rightsquigarrow_{\perp} \mathsf{Y}$:

$$\begin{aligned} \mathsf{filter}\ (\in \mathsf{A})\ \mathsf{xs} &= \mathsf{filter}\ (\in \mathsf{preimage}\ (\mathsf{g}\ \mathsf{X})\ \mathsf{B})\ \mathsf{xs} \\ &= \mathsf{filter}\ (\lambda\,\mathsf{a}.\,\mathsf{g}\ \mathsf{X}\ \mathsf{a} \in \mathsf{B})\ \mathsf{xs} \\ &= \mathsf{filter}\ (\lambda\,\mathsf{a}.\,\mathsf{f}\ \mathsf{a} \in \mathsf{B})\ \mathsf{xs} \end{aligned} \tag{49}$$

Substituting into (48) gives

$$\mathsf{Pr}[\mathsf{B}'|\mathsf{B}] \approx \frac{\mathsf{length}\ (\mathsf{filter}\ (\lambda\,\mathsf{a}.\,\mathsf{f}\ \mathsf{a} \in \mathsf{B}' \cap \mathsf{B})\ \mathsf{xs})}{\mathsf{length}\ (\mathsf{filter}\ (\lambda\,\mathsf{a}.\,\mathsf{f}\ \mathsf{a} \in \mathsf{B})\ \mathsf{xs})} \tag{50}$$

which converges to the probability of B′ given B as the number of samples xs from the covering partition increases.

For simplicity, the preceeding discussion does not deal with projecting preimages from the domain of programs $(\mathsf{R} \times \mathsf{T}) \times \{\langle\rangle\}$ onto the set of random sources R. Shortly, Dr. Bayes samples rectangles from covering partitions of $(\mathsf{R} \times \mathsf{T}) \times \{\langle\rangle\}$ subsets, weights each rectangle by the inverse of the probability with which it is sampled, and projects onto R. This alogorithm is a variant of **importance sampling** [10, Section 12.4], where the candidate distribution is defined by the sampling algorithm's partitioning choices, and the target distribution is P.

XXX: specific problems: thermometer, stochastic ray tracing

8 Related Work

Any programming language research described by the words "bijective" or "reversible" might seem to have much in common with ours. Unfortunately, when we look more closely, we can usually draw only loose analogies and perhaps inspiration. An example is lenses [13], which are transformations from X to Y that can be run forward and backward, in a way that maintains some relationship between X and Y. Usually, a destructive, external process is assumed, so that, for example, a change from $\mathsf{y} \in \mathsf{Y}$ to $\mathsf{y}' \in \mathsf{Y}$ induces a corresponding change from $\mathsf{x} \in \mathsf{X}$ to some $\mathsf{x}' \in \mathsf{X}$. When transformations lose information, lenses must satisfy certain behavioral laws. In our work, no input or output is updated, and preimages are always definable regardless of non-injectivity.

Many multi-paradigm languages [12], especially constraint functional languages, bear a strong resemblance to our work. In fact, it is easy to add a fail expression to our semantics, or to transform constraints into boolean program outputs. The most obvious difference is evaluation strategy. The most important difference is that our interpretation of programs returns *distributions* of constrained outputs, rather than arbitrary single values that meet constraints.

The forward phase in computing preimages takes a subdomain and returns an overapproximation of the function's range for that subdomain. This clearly generalizes interval arithmetic [17] to all first-order algebraic types.

Our approximating semantics can be regarded as an abstract interpretation [9] where the concrete domain consists of measurable sets and the abstract domain consists of rectangular sets. In some ways, it is quite typical: it is sound, it labels expressions, the abstract domain is a lattice, and the exact semantics it approximates performs infinite computations. However, it is far from typical in other ways. It is used to run programs, not for static analysis. The abstraction boundaries are the if branches of completely unrolled, infinite programs, and are not fixed. There is no Kleene iteration. Infinite computations are done in a library of λ_{ZFC}-computable combinators, not by a semantic function. This cleanly separates the syntax from the semantics, and allows us to prove the exact semantics correct mostly by proving simple categorical properties.

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 a probabilistic Scheme by Koller and Pfeffer [20], BUGS [23], BLOG [25], BLAISE [6], Church [11], and Kiselyov's embedded language for O'Caml based on continuations [18]. The reports on these languages generally describe interpreters, compilers, and algorithms for sampling with probabilistic conditions. Recently, Wingate et al [34, 33] 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. Kozen [21] defines the meaning of bounded-space, imperative "while" programs as functions from probability measures to probability measures. Hurd [15] proves properties about programs with binary random choice by encoding programs and portions of measure

theory in HOL. Jones [16] develops a domain-theoretic variation of probability theory, and with it defines the probability monad, whose discrete version is a distribution-valued variation of the set or list monad. Ramsey and Pfeffer [28] define the probability monad measure-theoretically and implement a language for finite probability. Park [26] extends a λ-calculus with probabilistic choice from a general class of probability measures using inverse transform sampling.

Some recent work in probabilistic language semantics tackles conditioning. Pfeffer's IBAL [27] is the earliest lambda calculus with finite probabilistic choice that also defines conditional queries. Borgström et al [7] develop Fun, a first-order functional language without recursion, extended with probabilistic choice and conditioning. Its semantics interprets programs as *measure transformers* by transforming expressions into arrow-like combinators. The implementation generates a decomposition of the probability density represented by the program, if it exists. Bhat et al [5] replaces Fun's if with match, and interprets programs more directly as probability density functions by compositionally transforming expressions into an extension of the probability monad.

9 Conclusions and Future Work

XXX: todo

Understanding the exact semantics, and implementing the approximating semantics, requires little more than basic set theory and some experience using combinator libraries in a pure λ-calculus.

the conditions under which the approximating semantics is complete in the limit, up to null sets

relation to type systems

constraints

sampling algorithms

different abstract domains

References

1. Haskell 98 language and libraries, the revised report (December 2002), http://www.haskell.org/onlinereport/
2. Abramsky, S.: The lazy lambda calculus. In: Research Topics in Functional Programming. pp. 65–116. Addison-Wesley (1990)
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. Bhat, S., Borgström, J., Gordon, A.D., Russo, C.: Deriving probability density functions from probabilistic functional programs. In: Tools and Algorithms for the Construction and Analysis of Systems (2013)
6. Bonawitz, K.A.: Composable Probabilistic Inference with Blaise. Ph.D. thesis, Massachusetts Institute of Technology (2008)

7. Borgström, J., Gordon, A.D., Greenberg, M., Margetson, J., Gael, J.V.: Measure transformer semantics for Bayesian machine learning. In: European Symposium on Programming (2011)
8. Chakravarty, M.M.T., Keller, G., Jones, S.P., Marlow, S.: Associated types with class. In: Principles of Programming Languages. pp. 1–13 (2005)
9. 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)
10. DeGroot, M., Schervish, M.: Probability and Statistics. Addison Wesley Publishing Company, Inc. (2012)
11. Goodman, N., Mansinghka, V., Roy, D., Bonawitz, K., Tenenbaum, J.: Church: a language for generative models. In: Uncertainty in Artificial Intelligence (2008)
12. Hanus, M.: Multi-paradigm declarative languages. In: Logic Programming, pp. 45–75 (2007)
13. Hofmann, M., Pierce, B.C., , Wagner, D.: Edit lenses. In: Principles of Programming Languages (2012)
14. Hughes, J.: Generalizing monads to arrows. In: Science of Computer Programming. vol. 37, pp. 67–111 (2000)
15. Hurd, J.: Formal Verification of Probabilistic Algorithms. Ph.D. thesis, University of Cambridge (2002)
16. Jones, C.: Probabilistic Non-Determinism. Ph.D. thesis, University of Edinburgh (1990)
17. Kearfott, R.B.: Interval computations: Introduction, uses, and resources. Euromath Bulletin 2, 95–112 (1996)
18. Kiselyov, O., Shan, C.: Monolingual probabilistic programming using generalized coroutines. In: Uncertainty in Artificial Intelligence (2008)
19. Klenke, A.: Probability Theory: A Comprehensive Course. Springer (2006)
20. Koller, D., McAllester, D., Pfeffer, A.: Effective Bayesian inference for stochastic programs. In: 14th National Conference on Artificial Intelligence (August 1997)
21. Kozen, D.: Semantics of probabilistic programs. In: Foundations of Computer Science (1979)
22. Lindley, S., Wadler, P., Yallop, J.: The arrow calculus. Journal of Functional Programming 20, 51–69 (2010)
23. Lunn, D.J., Thomas, A., Best, N., Spiegelhalter, D.: WinBUGS – a Bayesian modelling framework. Statistics and Computing 10(4) (2000)
24. McBride, C., Paterson, R.: Applicative programming with effects. Journal of Functional Programming 18(1) (2008)
25. 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)
26. Park, S., Pfenning, F., Thrun, S.: A probabilistic language based upon sampling functions. Transactions on Programming Languages and Systems 31(1) (2008)
27. Pfeffer, A.: The design and implementation of IBAL: A general-purpose probabilistic language. In: Statistical Relational Learning. MIT Press (2007)
28. Ramsey, N., Pfeffer, A.: Stochastic lambda calculus and monads of probability distributions. In: Principles of Programming Languages (2002)
29. Tobin-Hochstadt, S., Felleisen, M.: The design and implementation of typed Scheme. In: Principles of Programming Languages (2008)
30. 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)

31. Toronto, N., McCarthy, J.: Computing in Cantor's paradise with λ_{ZFC}. In: Functional and Logic Programming Symposium (FLOPS). pp. 290–306 (2012)
32. Wadler, P.: Monads for functional programming. In: Jeuring, J., Meijer, E. (eds.) Advanced Functional Programming (2001)
33. Wingate, D., Goodman, N.D., Stuhlmüller, A., Siskind, J.M.: Nonstandard interpretations of probabilistic programs for efficient inference. In: Neural Information Processing Systems (2011)
34. Wingate, D., Stuhlmüller, A., Goodman, N.D.: Lightweight implementations of probabilistic programming languages via transformational compilation. In: Artificial Intelligence and Statistics (2011)