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.

Measurability is a basic property similar to continuity that is often neglected, but critical. We prove that all probabilistic programs are measurable regardless of nontermination, if the language's primitives are measurable—which includes real arithmetic, inequalities and limits.

Preimages are generally uncomputable, so we derive an approximating semantics for computing rectangular covers of preimages. We implement the approximating semantics in Haskell and Typed Racket, and demonstrate its expressive power using stochastic ray tracing.

Keywords: Probability, Semantics, Domain-Specific Languages

1 Introduction

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

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

1.1 Probability Densities

These common language restrictions arise from reasoning about probability using **densities**, which are functions from random values to *changes* in probability. While simple and convenient, densities have many limitations. For example,

densities for random values with different dimension are incomparable, and they cannot be defined on infinite products. Either limitation rules out recursion.

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

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

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

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

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

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

1.2 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 **preimage measure** defines the distribution over subsets of Y:

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

The preimage $f^{-1}(B) = \{a \in X \mid f(a) \in B\}$ is the subset of X for which f yields a value in B, and is well-defined for any f. 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 $\mathbb{R}$.

Measure-theoretic probability supports any kind of condition. If $\Pr[B] > 0$, the probability of $B' \subseteq Y$ given $B \subseteq Y$ is

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

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

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

1.3 Measure-Theoretic Semantics

Most purely functional languages allow only nontermination as a side effect, and not probabilistic choice. Programmers therefore encode probabilistic programs as functions from random sources to outputs. Monads and other categorical classes such as idioms (i.e. applicative functors) can make doing so easier [15,31].

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} [32], 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.

There seems to be no way to simplify difficulty 2, so we work through it in Section 9. The outcome is worth it: we prove all probabilistic programs are measurable, regardless of the inputs on which they do not terminate. This includes uncomputable programs; for example, those that contain real equality tests and limits. We believe this result is the first of its kind, and is general enough to apply to almost all past and future work on probabilistic programming languages.

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

1.4 Arrow Solution Overview

Using arrows, we define an *exact* semantics and an *approximating* semantics. The exact semantics includes

- A semantic function which, like the arrow calculus semantic function [23], 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:

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

From top-left to top-right, $\mathsf{X} \rightsquigarrow_{\perp} \mathsf{Y}$ arrow computations are intensional functions that may raise errors, $\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}$ instances produce extensional functions, and $\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}$ instances compute preimages. Instances of arrows in the bottom row are like those in the top, except they thread an infinite store of random values, and can be constructed to always terminate. Most of our correctness theorems rely on proofs that every morphism in (5) is a homomorphism.

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

2 Operational Metalanguage

We write programs in λ_{ZFC} [32], an untyped, call-by-value, operational λ-calculus designed for deriving implementable programs from contemporary mathematics.

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

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

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

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

– A free type variable is universally quantified; if uppercase, it denotes a set.

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

– A set denotes a member of that set.
– $\mathsf{x} \Rightarrow \mathsf{y}$ denotes a partial function.
– $\langle \mathsf{x}, \mathsf{y} \rangle$ denotes a pair of values with types x and y.
– $\mathsf{Set}\ \mathsf{x}$ denotes a set with members of type x.

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

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

We import ZFC theorems as lemmas; for example:

Lemma 1 (extensionality). *For all* $\mathsf{A} : \mathsf{Set}\ \mathsf{x}$ *and* $\mathsf{B} : \mathsf{Set}\ \mathsf{x}$, $\mathsf{A} = \mathsf{B}$ *if and only if* $\mathsf{A} \subseteq \mathsf{B}$ *and* $\mathsf{B} \subseteq \mathsf{A}$.

Or, $\mathsf{A} = \mathsf{B}$ if and only if they contain the same members.

2.1 Internal Equality and External Equivalence

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

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

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

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

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

2.2 Additional Functions and Syntactic Forms

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

A less typical set operation we use is disjoint union:

$$\begin{aligned} &(\uplus) : \mathsf{Set}\ \mathsf{x} \Rightarrow \mathsf{Set}\ \mathsf{x} \Rightarrow \mathsf{Set}\ \mathsf{x} \\ &\mathsf{A} \uplus \mathsf{B} \ :=\ \mathsf{if}\ (\mathsf{A} \cap \mathsf{B} = \varnothing)\ (\mathsf{A} \cup \mathsf{B})\ (\mathsf{take}\ \varnothing) \end{aligned} \tag{6}$$

The primitive $\mathsf{take} : \mathsf{Set}\ \mathsf{x} \Rightarrow \mathsf{x}$ returns the element in a singleton set, and does not terminate when applied to a non-singleton set. Thus, $\mathsf{A} \uplus \mathsf{B}$ terminates only when A and B are disjoint.

```
domain : (X ⇀ Y) ⇒ Set X
domain := image fst

range : (X ⇀ Y) ⇒ Set Y
range := image snd

preimage : (X ⇀ Y) ⇒ Set Y ⇒ Set X
preimage g B := {a ∈ domain g | g a ∈ B}

restrict : (X ⇀ Y) ⇒ Set X ⇒ (X ⇀ Y)
restrict g A := λa ∈ (A ∩ domain g). g a

⟨·,·⟩map : (X ⇀ Y1) ⇒ (X ⇀ Y2) ⇒ (X ⇀ Y1 × Y2)
⟨g1, g2⟩map := let A := (domain g1) ∩ (domain g2)
               in λa ∈ A. ⟨g1 a, g2 a⟩

(∘map) : (Y ⇀ Z) ⇒ (X ⇀ Y) ⇒ (X ⇀ Z)
g2 ∘map g1 := let A := preimage g1 (domain g2)
              in λa ∈ A. g2 (g1 a)

(⊎map) : (X ⇀ Y) ⇒ (X ⇀ Y) ⇒ (X ⇀ Y)
g1 ⊎map g2 :=
  let A := (domain g1) ⊎ (domain g2)
  in λa ∈ A. if (a ∈ domain g1) (g1 a) (g2 a)
```

Fig. 1: Operations on mappings.

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

Syntax for unnamed mappings is defined by

$$\lambda x_a \in e_A.\, e_b \;:\equiv\; \mathsf{mapping}\ (\lambda x_a.\, e_b)\ e_A \tag{7}$$

$$\begin{aligned} &\mathsf{mapping} : (\mathsf{X} \Rightarrow \mathsf{Y}) \Rightarrow \mathsf{Set\ X} \Rightarrow (\mathsf{X} \rightharpoonup \mathsf{Y}) \\ &\mathsf{mapping\ f\ A} := \mathsf{image}\ (\lambda \mathsf{a}.\, \langle \mathsf{a}, \mathsf{f\ a}\rangle)\ \mathsf{A} \end{aligned} \tag{8}$$

For symmetry with partial functions $\mathsf{x} \Rightarrow \mathsf{y}$, mapping returns a member of the set $\mathsf{X} \rightharpoonup \mathsf{Y}$ of all partial mappings from X to Y. Fig. 1 defines other mapping operations: domain, range, preimage, restrict, pairing, composition, and disjoint union. The latter three are particularly important in the preimage arrow's derivation.

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

$$\begin{aligned} &\pi : \mathsf{J} \Rightarrow (\mathsf{J} \to \mathsf{X}) \Rightarrow \mathsf{X} \\ &\pi\ \mathsf{j\ f} := \mathsf{f\ j} \end{aligned} \tag{9}$$

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

3 Arrows and First-Order Semantics

Like monads [34] and idioms [25], arrows [14] thread effects through computations in a way that imposes structure. But arrow computations are always

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

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

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 $\mathsf{first_a}$, we define $(\&\&\&_\mathsf{a})$. This combinator is typically called **fanout**, but its use will be clearer if we call it **pairing**. One way to strengthen an arrow a is to define an additional combinator $\mathsf{left_a}$, which can be used to choose an arrow computation based on the result of another. Again, we define a different combinator, $\mathsf{ifte_a}$ ("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 lazy for deferring conditional branches. For example, define the **function arrow** with choice:

$$\begin{aligned}
\mathsf{arr\ f} &:= \mathsf{f} \\
(\mathsf{f_1} \ggg \mathsf{f_2})\ \mathsf{a} &:= \mathsf{f_2\ (f_1\ a)} \\
(\mathsf{f_1} \mathbin{\&\&\&} \mathsf{f_2})\ \mathsf{a} &:= \langle \mathsf{f_1\ a}, \mathsf{f_2}, \mathsf{a} \rangle \\
\mathsf{ifte\ f_1\ f_2\ f_3\ a} &:= \mathsf{if\ (f_1\ a)\ (f_2\ a)\ (f_3\ a)} \\
\mathsf{lazy\ f\ a} &:= \mathsf{f\ 0\ a}
\end{aligned} \tag{10}$$

and try to define the following recursive function:

$$\mathsf{halt\text{-}on\text{-}true} := \mathsf{ifte\ (arr\ id)\ (arr\ id)\ halt\text{-}on\text{-}true} \tag{11}$$

In a strict λ-calculus, the defining expression does not terminate. But the following is well-defined in λ_{ZFC}, and loops only when applied to false:

$$\mathsf{halt\text{-}on\text{-}true} := \mathsf{ifte\ (arr\ id)\ (arr\ id)\ (lazy}\ \lambda 0.\,\mathsf{halt\text{-}on\text{-}true}\mathsf{)} \tag{12}$$

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

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

$$\begin{aligned}
\mathsf{arr_a} &: (\mathsf{x} \Rightarrow \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{y}) \\
(\ggg_\mathsf{a}) &: (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{y}) \Rightarrow (\mathsf{y} \rightsquigarrow_\mathsf{a} \mathsf{z}) \Rightarrow (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{z}) \\
(\&\&\&_\mathsf{a}) &: (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{z}) \Rightarrow (\mathsf{x} \rightsquigarrow_\mathsf{a} \langle \mathsf{y}, \mathsf{z} \rangle) \\
\mathsf{ifte_a} &: (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{Bool}) \Rightarrow (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{y}) \\
\mathsf{lazy_a} &: (1 \Rightarrow (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{y})) \Rightarrow (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{y})
\end{aligned} \tag{13}$$

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

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

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

$$\mathsf{lift_b}\ (\mathsf{arr_a}\ \mathsf{f}) \equiv \mathsf{arr_b}\ \mathsf{f} \tag{14}$$

$$\mathsf{lift_b}\ (\mathsf{f}_1 \ggg_\mathsf{a} \mathsf{f}_2) \equiv (\mathsf{lift_b}\ \mathsf{f}_1) \ggg_\mathsf{b} (\mathsf{lift_b}\ \mathsf{f}_2) \tag{15}$$

$$\mathsf{lift_b}\ (\mathsf{f}_1 \mathbin{\&\&\&_\mathsf{a}} \mathsf{f}_2) \equiv (\mathsf{lift_b}\ \mathsf{f}_1) \mathbin{\&\&\&_\mathsf{b}} (\mathsf{lift_b}\ \mathsf{f}_2) \tag{16}$$

$$\mathsf{lift_b}\ (\mathsf{ifte_a}\ \mathsf{f}_1\ \mathsf{f}_2\ \mathsf{f}_3) \equiv \mathsf{ifte_b}\ (\mathsf{lift_b}\ \mathsf{f}_1)\ (\mathsf{lift_b}\ \mathsf{f}_2)\ (\mathsf{lift_b}\ \mathsf{f}_3) \tag{17}$$

$$\mathsf{lift_b}\ (\mathsf{lazy_a}\ \mathsf{f}) \equiv \mathsf{lazy_b}\ \lambda 0.\, \mathsf{lift_b}\ (\mathsf{f}\ 0) \tag{18}$$

The arrow homomorphism laws state that $\mathsf{arr_a} : (\mathsf{x} \Rightarrow \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{y})$ must be a homomorphism from the function arrow (10) to arrow a. Roughly, arrow computations that do not use additional combinators can be transformed into $\mathsf{arr_a}$ applied to a pure computation. They must be *function-like*.

Only a few of the other arrow laws play a role in our semantics and its correctness. We need associativity of $(\ggg_\mathsf{a})$, a pair extraction law, and distribution of pure computations over effectful:

$$(\mathsf{f}_1 \ggg_\mathsf{a} \mathsf{f}_2) \ggg_\mathsf{a} \mathsf{f}_3 \equiv \mathsf{f}_1 \ggg_\mathsf{a} (\mathsf{f}_2 \ggg_\mathsf{a} \mathsf{f}_3) \tag{19}$$

$$(\mathsf{arr_a}\ \mathsf{f}_1 \mathbin{\&\&\&_\mathsf{a}} \mathsf{f}_2) \ggg_\mathsf{a} \mathsf{arr_a}\ \mathsf{snd} \equiv \mathsf{f}_2 \tag{20}$$

$$\mathsf{arr_a}\ \mathsf{f}_1 \ggg_\mathsf{a} (\mathsf{f}_2 \mathbin{\&\&\&_\mathsf{a}} \mathsf{f}_3) \equiv (\mathsf{arr_a}\ \mathsf{f}_1 \ggg_\mathsf{a} \mathsf{f}_2) \mathbin{\&\&\&_\mathsf{a}} (\mathsf{arr_a}\ \mathsf{f}_1 \ggg_\mathsf{a} \mathsf{f}_3) \tag{21}$$

$$\mathsf{arr_a}\ \mathsf{f}_1 \ggg_\mathsf{a} \mathsf{ifte_a}\ \mathsf{f}_2\ \mathsf{f}_3\ \mathsf{f}_4 \equiv \mathsf{ifte_a}\ (\mathsf{arr_a}\ \mathsf{f}_1 \ggg_\mathsf{a} \mathsf{f}_2)\ (\mathsf{arr_a}\ \mathsf{f}_1 \ggg_\mathsf{a} \mathsf{f}_3)\ (\mathsf{arr_a}\ \mathsf{f}_1 \ggg_\mathsf{a} \mathsf{f}_4) \tag{22}$$

$$\mathsf{arr_a}\ \mathsf{f}_1 \ggg_\mathsf{a} \mathsf{lazy_a}\ \mathsf{f}_2 \equiv \mathsf{lazy_a}\ \lambda 0.\, \mathsf{arr_a}\ \mathsf{f}_1 \ggg_\mathsf{a} \mathsf{f}_2\ 0 \tag{23}$$

Equivalence between different arrow representations is usually proved in a strongly normalizing λ-calculus [22,23], in which every function is free of effects, including nontermination. Such a λ-calculus has no need for $\mathsf{lazy_a}$, so we could not derive (23) from existing arrow laws. We follow Hughes's reasoning [14] for the original arrow laws: it is a function-like property (i.e. it holds for the function arrow), and it cannot not lose, reorder or duplicate effects.

The pair extraction law (20), which *can* be derived from existing arrow laws, is a more problematic, in nonstrict λ-calculii as well as λ_{ZFC}. If f_1 can loop, using (20) to transform a computation can turn a nonterminating expression into a terminating one, or vice-versa. We could condition the pair extraction law on f_1's termination. Instead, we require every argument to $\mathsf{arr_a}$ to terminate, which simplifies more proofs.

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

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

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

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

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

Fig. 2: Interpretation of a let-calculus with first-order definitions and De-Bruijn-indexed bindings as arrow a computations.

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 (epimorphism implies arrow laws). *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 the pair extraction law (20), rewrite in terms of $\mathsf{lift}_{\mathsf{b}}$, apply homomorphism laws, and apply the pair extraction law for arrow a:

$$
\begin{aligned}
&(\mathsf{arr}_{\mathsf{b}}\ \mathsf{f}_1\ \&\&\&_{\mathsf{b}}\ \mathsf{f}_2) \ggg_{\mathsf{b}} \mathsf{arr}_{\mathsf{b}}\ \mathsf{snd} \qquad (24) \\
&\quad\equiv (\mathsf{lift}_{\mathsf{b}}\ (\mathsf{arr}_{\mathsf{a}}\ \mathsf{f}_1)\ \&\&\&_{\mathsf{b}}\ (\mathsf{lift}_{\mathsf{b}}\ (\mathsf{lift}_{\mathsf{b}}^{-1}\ \mathsf{f}_2))) \ggg_{\mathsf{b}} \mathsf{arr}_{\mathsf{b}}\ \mathsf{snd} \\
&\quad\equiv \mathsf{lift}_{\mathsf{b}}\ (\mathsf{arr}_{\mathsf{a}}\ \mathsf{f}_1\ \&\&\&_{\mathsf{a}}\ \mathsf{lift}_{\mathsf{b}}^{-1}\ \mathsf{f}_2) \ggg_{\mathsf{b}} \mathsf{lift}_{\mathsf{b}}\ (\mathsf{arr}_{\mathsf{a}}\ \mathsf{snd}) \\
&\quad\equiv \mathsf{lift}_{\mathsf{b}}\ ((\mathsf{arr}_{\mathsf{a}}\ \mathsf{f}_1\ \&\&\&_{\mathsf{a}}\ \mathsf{lift}_{\mathsf{b}}^{-1}\ \mathsf{f}_2) \ggg_{\mathsf{b}} \mathsf{arr}_{\mathsf{a}}\ \mathsf{snd}) \\
&\quad\equiv \mathsf{lift}_{\mathsf{b}}\ (\mathsf{lift}_{\mathsf{b}}^{-1}\ \mathsf{f}_2) \\
&\quad\equiv \mathsf{f}_2
\end{aligned}
$$

The proofs for every other law are similar. □

3.2 First-Order Let-Calculus Semantics

Fig. 2 defines a transformation from a first-order let-calculus to arrow computations for any arrow a. A program is a sequence of definition statements followed by a final expression. The semantic function $[\![\cdot]\!]_{\mathsf{a}}$ transforms each defining expression and the final expression into arrow computations. Functions are named, but local variables and arguments are not. Instead, variables are referred to by De Bruijn indexes, with 0 referring to the innermost binding.

Perhaps unsurprisingly, interpretations act like stack machines. A final expression has type $\langle\rangle \rightsquigarrow_{\mathsf{a}} \mathsf{y}$, where y is the type of the program's value, and $\langle\rangle$

denotes an empty list, or stack. A let expression pushes a value onto the stack. First-order functions have type $\langle \mathsf{x}, \langle\rangle \rangle \rightsquigarrow_{\mathsf{a}} \mathsf{y}$ where x is the argument type and y is the return type. Application sends a stack containing just an x.

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

Definition 5 (well-defined expression). *An expression e is **well-defined** under arrow a if $[\![e]\!]_{\mathsf{a}} : \mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}$ 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.) Well-definedness does not guarantee that *running* an interpretation terminates. It just simplifies statements about expressions, such as the following theorem, on which most of our semantic correctness results rely.

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

Proof. By structural induction. Bases cases proceed by expansion and using $\mathsf{arr_b} \equiv \mathsf{lift_b} \circ \mathsf{arr_a}$ (14). For example, for constants:

$$\begin{aligned} [\![v]\!]_{\mathsf{b}} &\equiv \mathsf{arr_b}\ (\mathsf{const}\ v) \\ &\equiv \mathsf{lift_b}\ (\mathsf{arr_a}\ (\mathsf{const}\ v)) \\ &\equiv \mathsf{lift_b}\ [\![v]\!]_{\mathsf{a}} \end{aligned} \tag{25}$$

Inductive cases proceed by expansion, applying the inductive hypothesis on subterms, and applying distributive laws (15)–(18). For example, for pairing:

$$\begin{aligned} [\![\langle e_1, e_2 \rangle]\!]_{\mathsf{b}} &\equiv [\![e_1]\!]_{\mathsf{b}}\ \mathbin{\&\&\&_{\mathsf{b}}}\ [\![e_2]\!]_{\mathsf{b}} \\ &\equiv (\mathsf{lift_b}\ [\![e_1]\!]_{\mathsf{a}})\ \mathbin{\&\&\&_{\mathsf{b}}}\ (\mathsf{lift_b}\ [\![e_2]\!]_{\mathsf{a}}) \\ &\equiv \mathsf{lift_b}\ ([\![e_1]\!]_{\mathsf{a}}\ \mathbin{\&\&\&_{\mathsf{a}}}\ [\![e_2]\!]_{\mathsf{a}}) \\ &\equiv \mathsf{lift_b}\ [\![\langle e_1, e_2 \rangle]\!]_{\mathsf{a}} \end{aligned} \tag{26}$$

It is not hard to check the remaining cases. □

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

4 The Bottom Arrow

Using the diagram in (5) as a sort of map, we start in the upper-left corner:

$$\begin{array}{ccccc} \mathsf{X} \rightsquigarrow_{\perp} \mathsf{Y} & \xrightarrow{\mathsf{lift_{map}}} & \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y} & \xrightarrow{\mathsf{lift_{pre}}} & \mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y} \\ \downarrow{\scriptstyle \eta_{\perp^*}} & & \downarrow{\scriptstyle \eta_{\mathsf{map}^*}} & & \downarrow{\scriptstyle \eta_{\mathsf{pre}^*}} \\ \mathsf{X} \rightsquigarrow_{\perp^*} \mathsf{Y} & \xrightarrow[\mathsf{lift_{map^*}}]{} & \mathsf{X} \underset{\mathsf{map}^*}{\rightsquigarrow} \mathsf{Y} & \xrightarrow[\mathsf{lift_{pre^*}}]{} & \mathsf{X} \underset{\mathsf{pre}^*}{\rightsquigarrow} \mathsf{Y} \end{array} \tag{27}$$

$$\mathsf{X} \rightsquigarrow_\perp \mathsf{Y} \;::=\; \mathsf{X} \Rightarrow \mathsf{Y}_\perp$$

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

$$(\ggg_\perp) : (\mathsf{X} \rightsquigarrow_\perp \mathsf{Y}) \Rightarrow (\mathsf{Y} \rightsquigarrow_\perp \mathsf{Z}) \Rightarrow (\mathsf{X} \rightsquigarrow_\perp \mathsf{Z})$$
$$(\mathsf{f}_1 \ggg_\perp \mathsf{f}_2)\ \mathsf{a} \;:=\; \mathsf{if}\ (\mathsf{f}_1\ \mathsf{a} = \perp)\ \perp\ (\mathsf{f}_2\ (\mathsf{f}_1\ \mathsf{a}))$$

$$(\&\&\&_\perp) : (\mathsf{X} \rightsquigarrow_\perp \mathsf{Y}_1) \Rightarrow (\mathsf{X} \rightsquigarrow_\perp \mathsf{Y}_2) \Rightarrow (\mathsf{X} \rightsquigarrow_\perp \langle \mathsf{Y}_1, \mathsf{Y}_2 \rangle)$$
$$(\mathsf{f}_1 \&\&\&_\perp \mathsf{f}_2)\ \mathsf{a} \;:=\; \mathsf{if}\ (\mathsf{f}_1\ \mathsf{a} = \perp\ \mathsf{or}\ \mathsf{f}_2\ \mathsf{a} = \perp)\ \perp\ \langle \mathsf{f}_1\ \mathsf{a}, \mathsf{f}_2\ \mathsf{a} \rangle$$

$$\mathsf{ifte}_\perp : (\mathsf{X} \rightsquigarrow_\perp \mathsf{Bool}) \Rightarrow (\mathsf{X} \rightsquigarrow_\perp \mathsf{Y}) \Rightarrow (\mathsf{X} \rightsquigarrow_\perp \mathsf{Y}) \Rightarrow (\mathsf{X} \rightsquigarrow_\perp \mathsf{Y})$$
$$\begin{array}{l}\mathsf{ifte}_\perp\ \mathsf{f}_1\ \mathsf{f}_2\ \mathsf{f}_3\ \mathsf{a} :=\\ \quad \mathsf{case}\ \mathsf{f}_1\ \mathsf{a}\\ \quad\quad \mathsf{true} \longrightarrow \mathsf{f}_2\ \mathsf{a}\\ \quad\quad \mathsf{false} \longrightarrow \mathsf{f}_3\ \mathsf{a}\\ \quad\quad \perp \longrightarrow \perp\end{array}$$

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

Fig. 3: Bottom arrow definitions.

Through Section 7, we move across the top to $\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}$.

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 with easily understood behavior. The function arrow (10) is an obvious candidate. However, we will need to explicitly handle nontermination as an error value, so we need a slightly more complicated arrow.

Fig. 3 defines the ***bottom arrow***. Its computations have type $\mathsf{X} \rightsquigarrow_\perp \mathsf{Y} ::= \mathsf{X} \Rightarrow \mathsf{Y}_\perp$, where $\mathsf{Y}_\perp ::= \mathsf{Y} \cup \{\perp\}$ and $\perp$ is a distinguished error value. The type $\mathsf{Bool}_\perp$, for example, denotes the members of $\mathsf{Bool} \cup \{\perp\} = \{\mathsf{true}, \mathsf{false}, \perp\}$.

To prove the arrow laws, we need a coarser notion of 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* $\mathsf{a} \in \mathsf{X}$*.*

Theorem 3. $\mathsf{arr}_\perp$*,* $(\&\&\&_\perp)$*,* $(\ggg_\perp)$*,* $\mathsf{ifte}_\perp$ *and* $\mathsf{lazy}_\perp$ *define an arrow.*

Proof. The bottom arrow is epimorphic to (in fact, isomorphic to) the Maybe monad's Kleisli arrow. □

5 Deriving the Mapping Arrow

Theorems about functions in set theory tend to be about mappings, not about lambdas that may raise errors. As in intermediate step, then, we need an arrow whose computations produce mappings or are mappings themselves.

It is tempting to try to make the mapping arrow's computations mapping-valued; i.e. $\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y} ::= \mathsf{X} \rightharpoonup \mathsf{Y}$, with $\mathsf{f}_1 \ggg_{\mathsf{map}} \mathsf{f}_2 := \mathsf{f}_2 \circ_{\mathsf{map}} \mathsf{f}_1$ and $\mathsf{f}_1 \&\&\&_{\mathsf{map}} \mathsf{f}_2 := \langle \mathsf{f}_1, \mathsf{f}_2 \rangle_{\mathsf{map}}$. Unfortunately, we could not define $\mathsf{arr}_{\mathsf{map}} : (\mathsf{X} \Rightarrow \mathsf{Y}) \Rightarrow (\mathsf{X} \rightharpoonup \mathsf{Y})$: to define a mapping, we need a domain, but lambdas' domains are unobservable.

To parameterize mapping arrow computations on a domain, we define the ***mapping arrow*** computation type as

$$\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y} \;::=\; \mathsf{Set}\ \mathsf{X} \Rightarrow (\mathsf{X} \rightharpoonup \mathsf{Y}) \tag{28}$$

The absence of $\perp$ in $\mathsf{Set}\ \mathsf{X} \Rightarrow (\mathsf{X} \rightharpoonup \mathsf{Y})$, and the fact that type parameters X and Y denote sets, will make it easier to apply well-known theorems from measure theory, which know nothing of lambda types and propagating error values.

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

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

$$(\ggg_{\mathsf{map}}) : (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}) \Rightarrow (\mathsf{Y} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Z}) \Rightarrow (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Z})$$
$$(\mathsf{g}_1 \ggg_{\mathsf{map}} \mathsf{g}_2)\ \mathsf{A} \;:=\; \mathsf{let}\ \ \mathsf{g}_1' := \mathsf{g}_1\ \mathsf{A}\ \ \mathsf{g}_2' := \mathsf{g}_2\ (\mathsf{range}\ \mathsf{g}_1')\ \ \mathsf{in}\ \ \mathsf{g}_2' \circ_{\mathsf{map}} \mathsf{g}_1'$$

$$(\&\&\&_{\mathsf{map}}) : (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}_1) \Rightarrow (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}_2) \Rightarrow (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \langle \mathsf{Y}_1, \mathsf{Y}_2 \rangle)$$
$$(\mathsf{g}_1 \;\&\&\&_{\mathsf{map}}\; \mathsf{g}_2)\ \mathsf{A} \;:=\; \langle \mathsf{g}_1\ \mathsf{A}, \mathsf{g}_2\ \mathsf{A} \rangle_{\mathsf{map}}$$

$$\mathsf{ifte}_{\mathsf{map}} : (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Bool}) \Rightarrow (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y})$$
$$\mathsf{ifte}_{\mathsf{map}}\ \mathsf{g}_1\ \mathsf{g}_2\ \mathsf{g}_3\ \mathsf{A} \;:=\; \mathsf{let}\ \ \mathsf{g}_1' := \mathsf{g}_1\ \mathsf{A}\ \ \mathsf{g}_2' := \mathsf{g}_2\ (\mathsf{preimage}\ \mathsf{g}_1'\ \{\mathsf{true}\})\ \ \mathsf{g}_3' := \mathsf{g}_3\ (\mathsf{preimage}\ \mathsf{g}_1'\ \{\mathsf{false}\})\ \ \mathsf{in}\ \ \mathsf{g}_2' \uplus_{\mathsf{map}} \mathsf{g}_3'$$

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

$$\mathsf{lift}_{\mathsf{map}} : (\mathsf{X} \rightsquigarrow_{\perp} \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y})$$
$$\mathsf{lift}_{\mathsf{map}}\ \mathsf{f}\ \mathsf{A} := \{\langle \mathsf{a}, \mathsf{b} \rangle \in \mathsf{mapping}\ \mathsf{f}\ \mathsf{A} \mid \mathsf{b} \neq \perp\}$$

Fig. 4: Mapping arrow definitions.

To use Theorem 2 to prove that expressions interpreted using $[\![\cdot]\!]_{\mathsf{map}}$ behave correctly with respect to $[\![\cdot]\!]_{\perp}$, we need to define correctness using a lift from the bottom arrow to the mapping arrow. It is helpful to have a standalone function $\mathsf{domain}_{\perp}$ that computes the subset of A on which f does not return $\perp$. We define that first, and then define $\mathsf{lift}_{\mathsf{map}}$ in terms of it:

$$\begin{aligned} &\mathsf{domain}_{\perp} : (\mathsf{X} \rightsquigarrow_{\perp} \mathsf{Y}) \Rightarrow \mathsf{Set}\ \mathsf{X} \Rightarrow \mathsf{Set}\ \mathsf{X} \\ &\mathsf{domain}_{\perp}\ \mathsf{f}\ \mathsf{A} \;:=\; \{\mathsf{a} \in \mathsf{A} \mid \mathsf{f}\ \mathsf{a} \neq \perp\} \end{aligned} \tag{29}$$

$$\begin{aligned} &\mathsf{lift}_{\mathsf{map}} : (\mathsf{X} \rightsquigarrow_{\perp} \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}) \\ &\mathsf{lift}_{\mathsf{map}}\ \mathsf{f}\ \mathsf{A} \;:=\; \mathsf{mapping}\ \mathsf{f}\ (\mathsf{domain}_{\perp}\ \mathsf{f}\ \mathsf{A}) \end{aligned} \tag{30}$$

So $\mathsf{lift}_{\mathsf{map}}\ \mathsf{f}\ \mathsf{A}$ is like $\mathsf{mapping}\ \mathsf{f}\ \mathsf{A}$, except the domain does not contain inputs that produce errors or nontermination—a good notion of correctness.

If $\mathsf{lift}_{\mathsf{map}}$ is to be a homomorphism, mapping arrow computation equivalence needs to be more extensional.

Definition 7 (mapping arrow equivalence). *Two computations* $\mathsf{g}_1 : \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}$ *and* $\mathsf{g}_2 : \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}$ *are equivalent, or* $\mathsf{g}_1 \equiv \mathsf{g}_2$*, when* $\mathsf{g}_1\ \mathsf{A} \equiv \mathsf{g}_2\ \mathsf{A}$ *for all* $\mathsf{A} \subseteq \mathsf{X}$*.*

Clearly $\mathsf{arr}_{\mathsf{map}} := \mathsf{lift}_{\mathsf{map}} \circ \mathsf{arr}_{\perp}$ meets the first homomorphism law (14). The remainder of this section derives $(\&\&\&_{\mathsf{map}})$, $(\ggg_{\mathsf{map}})$, $\mathsf{ifte}_{\mathsf{map}}$ and $\mathsf{lazy}_{\mathsf{map}}$ from bottom arrow combinators, in a way that ensures $\mathsf{lift}_{\mathsf{map}}$ is an arrow homomorphism. Fig. 4 contains the resulting definitions.

5.1 Composition

Starting with the left side of (15), we expand definitions, simplify f by restricting it to a set for which $\mathsf{f}_1\ \mathsf{a} \neq \bot$, and substitute f's definition:

$$\begin{aligned}
\mathsf{lift}_{\mathsf{map}}\ (\mathsf{f}_1 \ggg \mathsf{f}_2)\ \mathsf{A} \equiv\ & \mathsf{let}\quad \mathsf{f} := \lambda \mathsf{a}.\, \mathsf{if}\ (\mathsf{f}_1\ \mathsf{a} = \bot)\ \bot\ (\mathsf{f}_2\ (\mathsf{f}_1\ \mathsf{a})) \\
& \qquad\ \mathsf{A}' := \mathsf{domain}_\bot\ \mathsf{f}\ \mathsf{A} \\
& \mathsf{in}\ \ \mathsf{mapping}\ \mathsf{f}\ \mathsf{A}' \\
\equiv\ & \mathsf{let}\quad \mathsf{f} := \lambda \mathsf{a}.\, \mathsf{f}_2\ (\mathsf{f}_1\ \mathsf{a}) \\
& \qquad\ \mathsf{A}' := \mathsf{domain}_\bot\ \mathsf{f}\ (\mathsf{domain}_\bot\ \mathsf{f}_1\ \mathsf{A}) \\
& \mathsf{in}\ \ \mathsf{mapping}\ \mathsf{f}\ \mathsf{A}' \\
\equiv\ & \mathsf{let}\ \ \mathsf{A}' := \{\mathsf{a} \in \mathsf{domain}_\bot\ \mathsf{f}_1\ \mathsf{A} \mid \mathsf{f}_2\ (\mathsf{f}_1\ \mathsf{a}) \neq \bot\} \\
& \mathsf{in}\ \ \lambda \mathsf{a} \in \mathsf{A}'.\, \mathsf{f}_2\ (\mathsf{f}_1\ \mathsf{a})
\end{aligned} \tag{31}$$

We finish by converting bottom arrow computations to the mapping arrow and rewriting in terms of $(\circ_{\mathsf{map}})$:

$$\begin{aligned}
\mathsf{lift}_{\mathsf{map}}\ (\mathsf{f}_1 \ggg \mathsf{f}_2)\ \mathsf{A} \equiv\ & \mathsf{let}\ \ \mathsf{g}_1 := \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_1\ \mathsf{A} \\
& \qquad\ \mathsf{A}' := \mathsf{preimage}\ \mathsf{g}_1\ (\mathsf{domain}_\bot\ \mathsf{f}_2\ (\mathsf{range}\ \mathsf{g}_1)) \\
& \mathsf{in}\ \ \lambda \mathsf{a} \in \mathsf{A}'.\, \mathsf{f}_2\ (\mathsf{g}_1\ \mathsf{a}) \\
\equiv\ & \mathsf{let}\ \ \mathsf{g}_1 := \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_1\ \mathsf{A} \\
& \qquad\ \mathsf{g}_2 := \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_2\ (\mathsf{range}\ \mathsf{g}_1) \\
& \qquad\ \mathsf{A}' := \mathsf{preimage}\ \mathsf{g}_1\ (\mathsf{domain}\ \mathsf{g}_2) \\
& \mathsf{in}\ \ \lambda \mathsf{a} \in \mathsf{A}'.\, \mathsf{g}_2\ (\mathsf{g}_1\ \mathsf{a}) \\
\equiv\ & \mathsf{let}\ \ \mathsf{g}_1 := \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_1\ \mathsf{A} \\
& \qquad\ \mathsf{g}_2 := \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_2\ (\mathsf{range}\ \mathsf{g}_1) \\
& \mathsf{in}\ \ \mathsf{g}_2 \circ_{\mathsf{map}} \mathsf{g}_1
\end{aligned} \tag{32}$$

Substituting g_1 for $\mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_1$ and g_2 for $\mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_2$ gives a definition for $(\ggg_{\mathsf{map}})$ (Fig. 4) for which (15) holds.

5.2 Pairing

Starting with the left side of (16), we expand definitions and replace the definition of A' with one that does not depend on f:

$$\begin{aligned}
\mathsf{lift}_{\mathsf{map}}\ (\mathsf{f}_1 \mathbin{\&\&\&}_\bot \mathsf{f}_2)\ \mathsf{A} \equiv\ & \mathsf{let}\quad \mathsf{f} := \lambda \mathsf{a}.\, \mathsf{if}\ (\mathsf{f}_1\ \mathsf{a} = \bot\ \mathsf{or}\ \mathsf{f}_2\ \mathsf{a} = \bot)\ \bot\ \langle \mathsf{f}_1\ \mathsf{a}, \mathsf{f}_2\ \mathsf{a} \rangle \\
& \qquad\ \mathsf{A}' := \mathsf{domain}_\bot\ \mathsf{f}\ \mathsf{A} \\
& \mathsf{in}\ \ \mathsf{mapping}\ \mathsf{f}\ \mathsf{A}' \\
\equiv\ & \mathsf{let}\ \ \mathsf{A}' := \mathsf{domain}_\bot\ \mathsf{f}_1\ \mathsf{A} \cap \mathsf{domain}_\bot\ \mathsf{f}_2\ \mathsf{A} \\
& \mathsf{in}\ \ \lambda \mathsf{a} \in \mathsf{A}'.\, \langle \mathsf{f}_1\ \mathsf{a}, \mathsf{f}_2\ \mathsf{a} \rangle
\end{aligned} \tag{33}$$

We finish by converting bottom arrow computations to the mapping arrow and rewriting in terms of $\langle\cdot,\cdot\rangle_{\mathsf{map}}$:

$$\begin{aligned}\mathsf{lift}_{\mathsf{map}}\ (\mathsf{f}_1\ \&\&\&_{\perp}\ \mathsf{f}_2)\ \mathsf{A} \ \equiv\ & \mathsf{let}\ \ \mathsf{g}_1 := \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_1\ \mathsf{A} \\ & \qquad \mathsf{g}_2 := \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_2\ \mathsf{A} \\ & \qquad \mathsf{A}' := \mathsf{domain}\ \mathsf{g}_1 \cap \mathsf{domain}\ \mathsf{g}_2 \\ & \quad \mathsf{in}\ \ \lambda\,\mathsf{a} \in \mathsf{A}'.\,\langle \mathsf{g}_1\ \mathsf{a}, \mathsf{g}_2\ \mathsf{a}\rangle \\ \equiv\ & \langle \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_1\ \mathsf{A}, \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_2\ \mathsf{A}\rangle_{\mathsf{map}}\end{aligned} \tag{34}$$

Substituting g_1 for $\mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_1$ and g_2 for $\mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_2$ gives a definition for $(\&\&\&_{\mathsf{map}})$ (Fig. 4) for which (16) holds.

5.3 Conditional

Starting with the left side of (17), we expand definitions, and simplify f by restricting it to a domain for which $\mathsf{f}_1\ \mathsf{a} \neq \perp$:

$$\begin{aligned}\mathsf{lift}_{\mathsf{map}}\ (\mathsf{ifte}_{\perp}\ \mathsf{f}_1\ \mathsf{f}_2\ \mathsf{f}_3)\ \mathsf{A} \ \equiv\ & \mathsf{let}\ \ \mathsf{f} := \lambda\,\mathsf{a}.\,\mathsf{case}\ \ \mathsf{f}_1\ \mathsf{a} \\ & \qquad\qquad\qquad\qquad\qquad\quad \mathsf{true}\ \ \longrightarrow\ \ \mathsf{f}_2\ \mathsf{a} \\ & \qquad\qquad\qquad\qquad\qquad\quad \mathsf{false}\ \ \longrightarrow\ \ \mathsf{f}_3\ \mathsf{a} \\ & \qquad\qquad\qquad\qquad\qquad\quad \perp\ \ \longrightarrow\ \ \perp \\ & \quad \mathsf{in}\ \ \mathsf{mapping}\ \mathsf{f}\ (\mathsf{domain}_{\perp}\ \mathsf{f}\ \mathsf{A}) \\ \equiv\ & \mathsf{let}\ \ \mathsf{g}_1 := \mathsf{mapping}\ \mathsf{f}\ \mathsf{A} \\ & \qquad \mathsf{A}_2 := \mathsf{preimage}\ \mathsf{g}_1\ \{\mathsf{true}\} \\ & \qquad \mathsf{A}_3 := \mathsf{preimage}\ \mathsf{g}_1\ \{\mathsf{false}\} \\ & \qquad\ \ \mathsf{f} := \lambda\,\mathsf{a}.\,\mathsf{if}\ (\mathsf{f}_1\ \mathsf{a})\ (\mathsf{f}_2\ \mathsf{a})\ (\mathsf{f}_3\ \mathsf{a}) \\ & \quad \mathsf{in}\ \ \mathsf{mapping}\ \mathsf{f}\ (\mathsf{domain}_{\perp}\ \mathsf{f}\ (\mathsf{A}_2 \uplus \mathsf{A}_3))\end{aligned} \tag{35}$$

We finish by converting bottom arrow computations to the mapping arrow and rewriting in terms of $(\uplus_{\mathsf{map}})$:

$$\begin{aligned}\mathsf{lift}_{\mathsf{map}}\ (\mathsf{ifte}_{\perp}\ \mathsf{f}_1\ \mathsf{f}_2\ \mathsf{f}_3)\ \mathsf{A} \ \equiv\ & \mathsf{let}\ \ \mathsf{g}_1 := \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_1\ \mathsf{A} \\ & \qquad \mathsf{g}_2 := \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_2\ (\mathsf{preimage}\ \mathsf{g}_1\ \{\mathsf{true}\}) \\ & \qquad \mathsf{g}_3 := \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_3\ (\mathsf{preimage}\ \mathsf{g}_1\ \{\mathsf{false}\}) \\ & \qquad \mathsf{A}' := \mathsf{domain}\ \mathsf{g}_2 \uplus \mathsf{domain}\ \mathsf{g}_3 \\ & \quad \mathsf{in}\ \ \lambda\,\mathsf{a} \in \mathsf{A}'.\,\mathsf{if}\ (\mathsf{a} \in \mathsf{domain}\ \mathsf{g}_2)\ (\mathsf{g}_2\ \mathsf{a})\ (\mathsf{g}_3\ \mathsf{a}) \\ \equiv\ & \mathsf{let}\ \ \mathsf{g}_1 := \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_1\ \mathsf{A} \\ & \qquad \mathsf{g}_2 := \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_2\ (\mathsf{preimage}\ \mathsf{g}_1\ \{\mathsf{true}\}) \\ & \qquad \mathsf{g}_3 := \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_3\ (\mathsf{preimage}\ \mathsf{g}_1\ \{\mathsf{false}\}) \\ & \quad \mathsf{in}\ \ \mathsf{g}_2 \uplus_{\mathsf{map}} \mathsf{g}_3\end{aligned} \tag{36}$$

Substituting g_1 for $\mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_1$, g_2 for $\mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_2$, and g_3 for $\mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_3$ gives a definition for $\mathsf{ifte}_{\mathsf{map}}$ (Fig. 4) for which (17) holds.

5.4 Laziness

Starting with the left side of (18), we expand definitions:

$$\begin{aligned}\mathsf{lift}_{\mathsf{map}}\ (\mathsf{lazy}_\perp\ \mathsf{f})\ \mathsf{A} \ \equiv\ & \mathsf{let}\ \ \mathsf{A}' := \mathsf{domain}_\perp\ (\lambda \mathsf{a}.\, \mathsf{f}\ 0\ \mathsf{a})\ \mathsf{A} \\ & \mathsf{in}\ \ \mathsf{mapping}\ (\lambda \mathsf{a}.\, \mathsf{f}\ 0\ \mathsf{a})\ \mathsf{A}'\end{aligned} \tag{37}$$

It appears we need an η rule to continue, which λ_{ZFC} does not have (i.e. $\lambda x.\ e\ x \not\equiv e$ because e may not terminate). Fortunately, we can use weaker facts. If $\mathsf{A} \neq \varnothing$, then $\mathsf{domain}_\perp\ (\lambda \mathsf{a}.\, \mathsf{f}\ 0\ \mathsf{a})\ \mathsf{A} \equiv \mathsf{domain}_\perp\ (\mathsf{f}\ 0)\ \mathsf{A}$. Further, it terminates if and only if $\mathsf{mapping}\ (\mathsf{f}\ 0)\ \mathsf{A}'$ terminates. Therefore, if $\mathsf{A} \neq \varnothing$, we can replace $\lambda \mathsf{a}.\, \mathsf{f}\ 0\ \mathsf{a}$ with $\mathsf{f}\ 0$. If $\mathsf{A} = \varnothing$, then $\mathsf{lift}_{\mathsf{map}}\ (\mathsf{lazy}_\perp\ \mathsf{f})\ \mathsf{A} = \varnothing$ (the empty mapping), so

$$\begin{aligned}\mathsf{lift}_{\mathsf{map}}\ (\mathsf{lazy}_\perp\ \mathsf{f})\ \mathsf{A} \ &\equiv\ \mathsf{if}\ (\mathsf{A} = \varnothing)\ \varnothing\ (\mathsf{mapping}\ (\mathsf{f}\ 0)\ (\mathsf{domain}_\perp\ (\mathsf{f}\ 0)\ \mathsf{A})) \\ &\equiv\ \mathsf{if}\ (\mathsf{A} = \varnothing)\ \varnothing\ (\mathsf{lift}_{\mathsf{map}}\ (\mathsf{f}\ 0)\ \mathsf{A})\end{aligned} \tag{38}$$

Substituting $\mathsf{g}\ 0$ for $\mathsf{lift}_{\mathsf{map}}\ (\mathsf{f}\ 0)$ gives a $\mathsf{lazy}_{\mathsf{map}}$ (Fig. 4) for which (18) holds.

5.5 Correctness

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

Proof. By construction. □

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

Without restrictions, mapping arrow computations can be quite unruly. For example, the following computation is well-typed, but returns the identity mapping on Bool when applied to an empty domain, and the empty mapping when applied to any other domain:

$$\begin{aligned}&\mathsf{nonmonotone} : \mathsf{Bool} \mathrel{\underset{\mathsf{map}}{\rightsquigarrow}} \mathsf{Bool} \\ &\mathsf{nonmonotone}\ \mathsf{A}\ :=\ \mathsf{if}\ (\mathsf{A} = \varnothing)\ (\mathsf{mapping}\ \mathsf{id}\ \mathsf{Bool})\ \varnothing\end{aligned} \tag{39}$$

It would be nice if we could be sure that every $\mathsf{X} \mathrel{\underset{\mathsf{map}}{\rightsquigarrow}} \mathsf{Y}$ is not only monotone, but acts as if it returned restricted mappings. The following equivalent property is easier to state, and makes proving the arrow laws simple.

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

By homomorphism of $\mathsf{lift}_{\mathsf{map}}$, mapping arrow combinators preserve this law. It is therefore safe to assume that the mapping arrow law holds for all $\mathsf{g} : \mathsf{X} \mathrel{\underset{\mathsf{map}}{\rightsquigarrow}} \mathsf{Y}$.

Theorem 5. $\mathsf{lift}_{\mathsf{map}}$ *is an arrow epimorphism.*

Proof. Follows from Theorem 4 and restriction of $\mathsf{X} \mathrel{\underset{\mathsf{map}}{\rightsquigarrow}} \mathsf{Y}$ to instances for which the mapping arrow law (Definition 8) holds. □

Corollary 2. $\mathsf{arr}_{\mathsf{map}}$, $(\mathbin{\&\!\&\!\&}_{\mathsf{map}})$, $(\ggg_{\mathsf{map}})$, $\mathsf{ifte}_{\mathsf{map}}$ *and* $\mathsf{lazy}_{\mathsf{map}}$ *define an arrow.*

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

$$\mathsf{pre} : (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y})$$
$$\mathsf{pre\ g} \ := \ \langle \mathsf{range\ g}, \lambda \mathsf{B}.\ \mathsf{preimage\ g\ B} \rangle$$

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

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

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

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

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

$$(\uplus_{\mathsf{pre}}) : (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y})$$
$$\mathsf{h}_1 \uplus_{\mathsf{pre}} \mathsf{h}_2 \ := \ \mathsf{let}\ \ \mathsf{Y}' := (\mathsf{range}_{\mathsf{pre}}\ \mathsf{h}_1) \cup (\mathsf{range}_{\mathsf{pre}}\ \mathsf{h}_2)$$
$$\qquad \mathsf{p} := \lambda \mathsf{B}.\ (\mathsf{ap}_{\mathsf{pre}}\ \mathsf{h}_1\ \mathsf{B}) \uplus (\mathsf{ap}_{\mathsf{pre}}\ \mathsf{h}_2\ \mathsf{B})$$
$$\quad \mathsf{in}\ \ \langle \mathsf{Y}', \mathsf{p} \rangle$$

Fig. 5: Lazy preimage mappings and operations.

6 Lazy Preimage Mappings

On a computer, we do not often have the luxury of testing each function input to see whether it belongs to a preimage set. Even for finite domains, doing so is often intractable.

If we wish to compute with infinite sets in the language implementation, we will need an abstraction that makes it easy to replace computation on points with computation on sets whose representations allow efficient operations. Therefore, in the preimage arrow, we confine computation on points to instances of

$$\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y} \ ::= \ \langle \mathsf{Set\ Y}, \mathsf{Set\ Y} \Rightarrow \mathsf{Set\ X} \rangle \tag{40}$$

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***.

Converting a mapping to a lazy preimage mapping requires constructing a delayed application of preimage:

$$\begin{aligned} &\mathsf{pre} : (\mathsf{X} \rightharpoonup \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y}) \\ &\mathsf{pre\ g} \ := \ \langle \mathsf{range\ g}, \lambda \mathsf{B}.\ \mathsf{preimage\ g\ B} \rangle \end{aligned} \tag{41}$$

To apply a preimage mapping to some B, we intersect B with its range and apply the preimage function:

$$\begin{aligned} &\mathsf{ap}_{\mathsf{pre}} : (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y}) \Rightarrow \mathsf{Set\ Y} \Rightarrow \mathsf{Set\ X} \\ &\mathsf{ap}_{\mathsf{pre}}\ \langle \mathsf{Y}', \mathsf{p} \rangle\ \mathsf{B} \ := \ \mathsf{p}\ (\mathsf{B} \cap \mathsf{Y}') \end{aligned} \tag{42}$$

Preimage arrow correctness depends on this fact: that using $\mathsf{ap}_{\mathsf{pre}}$ to compute preimages is the same as computing them from a mapping using preimage.

Lemma 2. *Let* $\mathsf{g} \in \mathsf{X} \rightharpoonup \mathsf{Y}$. *For all* $\mathsf{B} \subseteq \mathsf{Y}$ *and* Y' *such that* $\mathsf{range\ g} \subseteq \mathsf{Y}' \subseteq \mathsf{Y}$, $\mathsf{preimage\ g}\ (\mathsf{B} \cap \mathsf{Y}') = \mathsf{preimage\ g\ B}$.

Theorem 6 ($\mathsf{ap_{pre}}$ computes preimages). *Let* $\mathsf{g} \in \mathsf{X} \rightharpoonup \mathsf{Y}$. *For all* $\mathsf{B} \subseteq \mathsf{Y}$, $\mathsf{ap_{pre}}\ (\mathsf{pre}\ \mathsf{g})\ \mathsf{B} = \mathsf{preimage}\ \mathsf{g}\ \mathsf{B}$.

Proof. Expand definitions and apply Lemma 2 with $\mathsf{Y'} = \mathsf{range}\ \mathsf{g}$. □

Fig. 5 defines more operations on preimage mappings, including pairing, composition, and disjoint union operations corresponding to the mapping operations in Fig. 1. To prove them correct, we need preimage mappings to be equivalent when they compute the same preimages.

Definition 9 (preimage mapping equivalence). $\mathsf{h_1} : \mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y}$ *and* $\mathsf{h_2} : \mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y}$ *are equivalent, or* $\mathsf{h_1} \equiv \mathsf{h_2}$, *when* $\mathsf{ap_{pre}}\ \mathsf{h_1}\ \mathsf{B} \equiv \mathsf{ap_{pre}}\ \mathsf{h_2}\ \mathsf{B}$ *for all* $\mathsf{B} \subseteq \mathsf{Y}$.

Similarly to proving arrows correct, we prove the operations in Fig. 5 are correct by proving that pre is a homomorphism (though not an arrow homomorphism): it distributes over mapping operations to yield preimage mapping operations. The remainder of this section states these distributive properties as theorems and proves them. We will use these theorems to derive the preimage arrow from the mapping arrow.

6.1 Composition

To prove pre distributes over mapping composition, we can make more or less direct use of the fact that preimage distributes over mapping composition.

Lemma 3 (preimage distributes over ($\circ_{\mathsf{map}}$)). *Let* $\mathsf{g_1} \in \mathsf{X} \rightharpoonup \mathsf{Y}$ *and* $\mathsf{g_2} \in \mathsf{Y} \rightharpoonup \mathsf{Z}$. *For all* $\mathsf{C} \subseteq \mathsf{Z}$, $\mathsf{preimage}\ (\mathsf{g_2} \circ_{\mathsf{map}} \mathsf{g_1})\ \mathsf{C} = \mathsf{preimage}\ \mathsf{g_1}\ (\mathsf{preimage}\ \mathsf{g_2}\ \mathsf{C})$.

Theorem 7 (pre distributes over ($\circ_{\mathsf{map}}$)). *Let* $\mathsf{g_1} \in \mathsf{X} \rightharpoonup \mathsf{Y}$ *and* $\mathsf{g_2} \in \mathsf{Y} \rightharpoonup \mathsf{Z}$. *Then* $\mathsf{pre}\ (\mathsf{g_2} \circ_{\mathsf{map}} \mathsf{g_1}) \equiv (\mathsf{pre}\ \mathsf{g_2}) \circ_{\mathsf{pre}} (\mathsf{pre}\ \mathsf{g_1})$.

Proof. Let $\langle \mathsf{Z'}, \mathsf{p_2} \rangle := \mathsf{pre}\ \mathsf{g_2}$ and $\mathsf{C} \subseteq \mathsf{Z}$. Starting from the right side, expand definitions, apply Theorem 6, apply Lemma 3, and apply Theorem 6 again:

$$
\begin{aligned}
&\mathsf{ap_{pre}}\ ((\mathsf{pre}\ \mathsf{g_2}) \circ_{\mathsf{pre}} (\mathsf{pre}\ \mathsf{g_1}))\ \mathsf{C} \\
&\quad \equiv\ \mathsf{let}\ \ \mathsf{h} := \lambda \mathsf{C}.\, \mathsf{ap_{pre}}\ (\mathsf{pre}\ \mathsf{g_1})\ (\mathsf{p_2}\ \mathsf{C}) \\
&\qquad\quad \mathsf{in}\ \ \mathsf{h}\ (\mathsf{C} \cap \mathsf{Z'}) \\
&\quad \equiv\ \mathsf{ap_{pre}}\ (\mathsf{pre}\ \mathsf{g_1})\ (\mathsf{p_2}\ (\mathsf{C} \cap \mathsf{Z'})) \\
&\quad \equiv\ \mathsf{ap_{pre}}\ (\mathsf{pre}\ \mathsf{g_1})\ (\mathsf{ap_{pre}}\ (\mathsf{pre}\ \mathsf{g_2})\ \mathsf{C}) \\
&\quad \equiv\ \mathsf{preimage}\ \mathsf{g_1}\ (\mathsf{preimage}\ \mathsf{g_2}\ \mathsf{C}) \\
&\quad \equiv\ \mathsf{preimage}\ (\mathsf{g_2} \circ_{\mathsf{map}} \mathsf{g_1})\ \mathsf{C} \\
&\quad \equiv\ \mathsf{ap_{pre}}\ (\mathsf{pre}\ (\mathsf{g_2} \circ_{\mathsf{map}} \mathsf{g_1}))\ \mathsf{C}
\end{aligned}
\tag{43}
$$

□

6.2 Pairing

We have less luck with pairing than with composition, because preimage does not distribute over pairing. Fortunately, it distributes over pairing and cartesian product together.

Lemma 4 (preimage **distributes over** $\langle\cdot,\cdot\rangle_{\mathsf{map}}$ **and** $(\times)$)**.** *Let* $\mathsf{g}_1 \in \mathsf{X} \rightharpoonup \mathsf{Y}_1$ *and* $\mathsf{g}_2 \in \mathsf{X} \rightharpoonup \mathsf{Y}_2$*. For all* $\mathsf{B}_1 \subseteq \mathsf{Y}_1$ *and* $\mathsf{B}_2 \subseteq \mathsf{Y}_2$*,* $\mathsf{preimage}\ \langle \mathsf{g}_1, \mathsf{g}_2\rangle_{\mathsf{map}}\ (\mathsf{B}_1 \times \mathsf{B}_2) = (\mathsf{preimage}\ \mathsf{g}_1\ \mathsf{B}_1) \cap (\mathsf{preimage}\ \mathsf{g}_2\ \mathsf{B}_2)$.

Theorem 8 (pre **distributes over** $\langle\cdot,\cdot\rangle_{\mathsf{map}}$)**.** *Let* $\mathsf{g}_1 \in \mathsf{X} \rightharpoonup \mathsf{Y}_1$ *and* $\mathsf{g}_2 \in \mathsf{X} \rightharpoonup \mathsf{Y}_2$*. Then* $\mathsf{pre}\ \langle \mathsf{g}_1, \mathsf{g}_2\rangle_{\mathsf{map}} \equiv \langle \mathsf{pre}\ \mathsf{g}_1, \mathsf{pre}\ \mathsf{g}_2\rangle_{\mathsf{pre}}$.

Proof. Let $\langle \mathsf{Y}_1', \mathsf{p}_1\rangle := \mathsf{pre}\ \mathsf{g}_1$, $\langle \mathsf{Y}_2', \mathsf{p}_2\rangle := \mathsf{pre}\ \mathsf{g}_2$ and $\mathsf{B} \in \mathsf{Y}_1 \times \mathsf{Y}_2$. Starting from the right side, expand definitions, apply Theorem 6, apply Lemma 4, note that a product of singletons is a singleton pair, distribute preimage over the union, apply Lemma 2, and apply Theorem 6 again:

$$
\begin{aligned}
&\mathsf{ap}_{\mathsf{pre}}\ \langle \mathsf{pre}\ \mathsf{g}_1, \mathsf{pre}\ \mathsf{g}_2\rangle_{\mathsf{pre}}\ \mathsf{B} && (44)\\
&\quad\equiv\ \mathsf{let}\ \ \mathsf{Y}' := \mathsf{Y}_1' \times \mathsf{Y}_2'\\
&\qquad\qquad \mathsf{p} := \lambda \mathsf{B}.\ \bigcup_{\langle \mathsf{y}_1, \mathsf{y}_2\rangle \in \mathsf{B}} (\mathsf{p}_1\ \{\mathsf{y}_1\}) \cap (\mathsf{p}_2\ \{\mathsf{y}_2\})\\
&\qquad\quad \mathsf{in}\ \ \mathsf{p}\ (\mathsf{B} \cap \mathsf{Y}')\\
&\quad\equiv \bigcup_{\langle \mathsf{y}_1, \mathsf{y}_2\rangle \in \mathsf{B} \cap (\mathsf{Y}_1' \times \mathsf{Y}_2')} (\mathsf{p}_1\ \{\mathsf{y}_1\}) \cap (\mathsf{p}_2\ \{\mathsf{y}_2\})\\
&\quad\equiv \bigcup_{\langle \mathsf{y}_1, \mathsf{y}_2\rangle \in \mathsf{B} \cap (\mathsf{Y}_1' \times \mathsf{Y}_2')} (\mathsf{preimage}\ \mathsf{g}_1\ \{\mathsf{y}_1\}) \cap (\mathsf{preimage}\ \mathsf{g}_2\ \{\mathsf{y}_2\})\\
&\quad\equiv \bigcup_{\langle \mathsf{y}_1, \mathsf{y}_2\rangle \in \mathsf{B} \cap (\mathsf{Y}_1' \times \mathsf{Y}_2')} (\mathsf{preimage}\ \langle \mathsf{g}_1, \mathsf{g}_2\rangle_{\mathsf{map}}\ (\{\mathsf{y}_1\} \times \{\mathsf{y}_2\}))\\
&\quad\equiv \bigcup_{\langle \mathsf{y}_1, \mathsf{y}_2\rangle \in \mathsf{B} \cap (\mathsf{Y}_1' \times \mathsf{Y}_2')} (\mathsf{preimage}\ \langle \mathsf{g}_1, \mathsf{g}_2\rangle_{\mathsf{map}}\ \{\langle \mathsf{y}_1, \mathsf{y}_2\rangle\})\\
&\quad\equiv\ \mathsf{preimage}\ \langle \mathsf{g}_1, \mathsf{g}_2\rangle_{\mathsf{map}}\ (\mathsf{B} \cap (\mathsf{Y}_1' \times \mathsf{Y}_2'))\\
&\quad\equiv\ \mathsf{preimage}\ \langle \mathsf{g}_1, \mathsf{g}_2\rangle_{\mathsf{map}}\ \mathsf{B}\\
&\quad\equiv\ \mathsf{ap}_{\mathsf{pre}}\ (\mathsf{pre}\ \langle \mathsf{g}_1, \mathsf{g}_2\rangle_{\mathsf{map}})\ \mathsf{B}
\end{aligned}
$$

□

6.3 Disjoint Union

Like proving pre distributes over composition, the proof that it distributes over dijoint union simply lifts a lemma about preimage to lazy preimage mappings.

Lemma 5 (preimage **distributes over** $(\uplus_{\mathsf{map}})$)**.** *Let* $\mathsf{g}_1 \in \mathsf{X} \rightharpoonup \mathsf{Y}$ *and* $\mathsf{g}_2 \in \mathsf{X} \rightharpoonup \mathsf{Y}$ *have disjoint domains. For all* $\mathsf{B} \subseteq \mathsf{Y}$*,* $\mathsf{preimage}\ (\mathsf{g}_1 \uplus_{\mathsf{map}} \mathsf{g}_2)\ \mathsf{B} = (\mathsf{preimage}\ \mathsf{g}_1\ \mathsf{B}) \uplus (\mathsf{preimage}\ \mathsf{g}_2\ \mathsf{B})$.

Theorem 9 (pre **distributes over** $(\uplus_{\mathsf{map}})$)**.** *Let* $\mathsf{g}_1 \in \mathsf{X} \rightharpoonup \mathsf{Y}$ *and* $\mathsf{g}_2 \in \mathsf{X} \rightharpoonup \mathsf{Y}$ *have disjoint domains. Then* $\mathsf{pre}\ (\mathsf{g}_1 \uplus_{\mathsf{map}} \mathsf{g}_2) \equiv (\mathsf{pre}\ \mathsf{g}_1) \uplus_{\mathsf{pre}} (\mathsf{pre}\ \mathsf{g}_2)$.

Proof. Let $\mathsf{Y}'_1 := \mathsf{range}\ \mathsf{g}_1$, $\mathsf{Y}'_2 := \mathsf{range}\ \mathsf{g}_2$ and $\mathsf{B} \subseteq \mathsf{Y}$. Starting from the right side, expand definitions, apply Theorem 6, apply Lemma 5, appy Lemma 2, and apply Theorem 6 again:

$$
\begin{aligned}
&\mathsf{ap}_{\mathsf{pre}}\ ((\mathsf{pre}\ \mathsf{g}_1) \uplus_{\mathsf{pre}} (\mathsf{pre}\ \mathsf{g}_2))\ \mathsf{B} \qquad (45)\\
&\quad\equiv\ \mathsf{let}\ \ \mathsf{Y}' := \mathsf{Y}'_1 \cup \mathsf{Y}'_2\\
&\qquad\qquad\ \ \mathsf{h} := \lambda \mathsf{B}.\, (\mathsf{ap}_{\mathsf{pre}}\ (\mathsf{pre}\ \mathsf{g}_1)\ \mathsf{B}) \uplus (\mathsf{ap}_{\mathsf{pre}}\ (\mathsf{pre}\ \mathsf{g}_2)\ \mathsf{B})\\
&\qquad\quad\ \mathsf{in}\ \ \mathsf{h}\ (\mathsf{B} \cap \mathsf{Y}')\\
&\quad\equiv\ (\mathsf{ap}_{\mathsf{pre}}\ (\mathsf{pre}\ \mathsf{g}_1)\ (\mathsf{B} \cap (\mathsf{Y}'_1 \cup \mathsf{Y}'_2))) \uplus\\
&\qquad(\mathsf{ap}_{\mathsf{pre}}\ (\mathsf{pre}\ \mathsf{g}_2)\ (\mathsf{B} \cap (\mathsf{Y}'_1 \cup \mathsf{Y}'_2)))\\
&\quad\equiv\ (\mathsf{preimage}\ \mathsf{g}_1\ (\mathsf{B} \cap (\mathsf{Y}'_1 \cup \mathsf{Y}'_2))) \uplus\\
&\qquad(\mathsf{preimage}\ \mathsf{g}_2\ (\mathsf{B} \cap (\mathsf{Y}'_1 \cup \mathsf{Y}'_2)))\\
&\quad\equiv\ \mathsf{preimage}\ (\mathsf{g}_1 \uplus_{\mathsf{map}} \mathsf{g}_2)\ (\mathsf{B} \cap (\mathsf{Y}'_1 \cup \mathsf{Y}'_2))\\
&\quad\equiv\ \mathsf{preimage}\ (\mathsf{g}_1 \uplus_{\mathsf{map}} \mathsf{g}_2)\ \mathsf{B}\\
&\quad\equiv\ \mathsf{ap}_{\mathsf{pre}}\ (\mathsf{pre}\ (\mathsf{g}_1 \uplus_{\mathsf{map}} \mathsf{g}_2))\ \mathsf{B} \qquad \square
\end{aligned}
$$

7 Deriving the Preimage Arrow

Now we can define an arrow that runs expressions backward on sets of outputs. Its computations should produce preimage mappings or be preimage mappings.

As with the mapping arrow and mappings, we cannot have $\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y} ::= \mathsf{X} \underset{\mathsf{pre}}{\rightarrow} \mathsf{Y}$: we run into trouble trying to define $\mathsf{arr}_{\mathsf{pre}}$ because a preimage mapping needs an observable range. To get one, it is easiest to parameterize preimage computations on a $\mathsf{Set}\ \mathsf{X}$; therefore the ***preimage arrow*** type constructor is

$$\mathsf{X} \underset{\mathsf{pre}}{\rightrightarrows} \mathsf{Y}\ ::=\ \mathsf{Set}\ \mathsf{X} \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightarrow} \mathsf{Y}) \qquad (46)$$

or $\mathsf{Set}\ \mathsf{X} \Rightarrow \langle \mathsf{Set}\ \mathsf{Y}, \mathsf{Set}\ \mathsf{Y} \Rightarrow \mathsf{Set}\ \mathsf{X} \rangle$. To deconstruct the type, a preimage arrow computation computes a range first, and returns the range and a lambda that computes preimages.

To use Theorem 2, we need to define correctness using a lift from the mapping arrow to the preimage arrow. A simple candidate with the right type is

$$
\begin{aligned}
&\mathsf{lift}_{\mathsf{pre}} : (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y})\\
&\mathsf{lift}_{\mathsf{pre}}\ \mathsf{g}\ \mathsf{A}\ :=\ \mathsf{pre}\ (\mathsf{g}\ \mathsf{A})
\end{aligned} \qquad (47)
$$

By $\mathsf{lift}_{\mathsf{pre}}$'s definition and Theorem 6, for all $\mathsf{g} : \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}$, $\mathsf{A} \subseteq \mathsf{X}$ and $\mathsf{B} \subseteq \mathsf{Y}$,

$$
\begin{aligned}
\mathsf{ap}_{\mathsf{pre}}\ (\mathsf{lift}_{\mathsf{pre}}\ \mathsf{g}\ \mathsf{A})\ \mathsf{B}\ &\equiv\ \mathsf{ap}_{\mathsf{pre}}\ (\mathsf{pre}\ (\mathsf{g}\ \mathsf{A}))\ \mathsf{B}\\
&\equiv\ \mathsf{preimage}\ (\mathsf{g}\ \mathsf{A})\ \mathsf{B}
\end{aligned} \qquad (48)
$$

Thus, lifted mapping arrow computations compute correct preimages, exactly as we should expect them to.

To derive the preimage arrow's combinators in a way that makes $\mathsf{lift}_{\mathsf{pre}}$ a homomorphism, we need preimage arrow equivalence to mean "computes the same preimages."

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

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

$$(\ggg_{pre}) : (X \underset{pre}{\rightsquigarrow} Y) \Rightarrow (Y \underset{pre}{\rightsquigarrow} Z) \Rightarrow (X \underset{pre}{\rightsquigarrow} Z)$$
$$(h_1 \ggg_{pre} h_2)\ A\ :=\ \mathsf{let}\ \ h_1' := h_1\ A \quad h_2' := h_2\ (\mathsf{range}_{pre}\ h_1')\ \mathsf{in}\ \ h_2' \circ_{pre} h_1'$$

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

$$\mathsf{ifte}_{pre} : (X \underset{pre}{\rightsquigarrow} \mathsf{Bool}) \Rightarrow (X \underset{pre}{\rightsquigarrow} Y) \Rightarrow (X \underset{pre}{\rightsquigarrow} Y) \Rightarrow (X \underset{pre}{\rightsquigarrow} Y)$$
$$\mathsf{ifte}_{pre}\ h_1\ h_2\ h_3\ A\ :=\ \mathsf{let}\ \ h_1' := h_1\ A \quad h_2' := h_2\ (\mathsf{ap}_{pre}\ h_1'\ \{\mathsf{true}\}) \quad h_3' := h_3\ (\mathsf{ap}_{pre}\ h_1'\ \{\mathsf{false}\})\ \mathsf{in}\ \ h_2' \uplus_{pre} h_3'$$

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

$$\mathsf{lift}_{pre} : (X \underset{map}{\rightsquigarrow} Y) \Rightarrow (X \underset{pre}{\rightsquigarrow} Y)$$
$$\mathsf{lift}_{pre}\ g\ A\ :=\ \mathsf{pre}\ (g\ A)$$

Fig. 6: Preimage arrow definitions.

Definition 10 (preimage arrow equivalence). *Two computations* $h_1 : X \underset{pre}{\rightsquigarrow} Y$ *and* $h_2 : X \underset{pre}{\rightsquigarrow} Y$ *are equivalent, or* $h_1 \equiv h_2$*, when* $h_1\ A \equiv h_2\ A$ *for all* $A \subseteq X$.

As with arr_{map}, defining arr_{pre} as a composition meets (14). The remainder of this section derives $(\&\&\&_{pre})$, $(\ggg_{pre})$, ifte_{pre} and lazy_{pre} from mapping arrow combinators, in a way that ensures lift_{pre} is an arrow homomorphism from the mapping arrow to the preimage arrow. Fig. 6 contains the resulting definitions.

7.1 Pairing

Starting with the left side of (16), we expand definitions, apply Theorem 8, and rewrite in terms of lift_{pre}:

$$\begin{aligned} \mathsf{ap}_{pre}\ (\mathsf{lift}_{pre}\ (g_1 \&\&\&_{map} g_2)\ A)\ B &\equiv \mathsf{ap}_{pre}\ (\mathsf{pre}\ \langle g_1\ A, g_2\ A\rangle_{map})\ B \\ &\equiv \mathsf{ap}_{pre}\ \langle \mathsf{pre}\ (g_1\ A), \mathsf{pre}\ (g_2\ A)\rangle_{pre}\ B \\ &\equiv \mathsf{ap}_{pre}\ \langle \mathsf{lift}_{pre}\ g_1\ A, \mathsf{lift}_{pre}\ g_2\ A\rangle_{pre}\ B \end{aligned} \tag{49}$$

Substituting h_1 for $\mathsf{lift}_{pre}\ g_1$ and h_2 for $\mathsf{lift}_{pre}\ g_2$, and removing the application of ap_{pre} from both sides of the equivalence gives a definition of $(\&\&\&_{pre})$ (Fig. 6) for which (16) holds.

7.2 Composition

Starting with the left side of (15), we expand definitions, apply Theorem 7 and rewrite in terms of lift_{pre}:

$$\begin{aligned} \mathsf{ap}_{pre}\ (\mathsf{lift}_{pre}\ (g_1 \ggg_{map} g_2)\ A)\ C &\equiv \mathsf{let}\ \ g_1' := g_1\ A \quad g_2' := g_2\ (\mathsf{range}\ g_1')\ \mathsf{in}\ \ \mathsf{ap}_{pre}\ (\mathsf{pre}\ (g_2' \circ_{map} g_1'))\ C \\ &\equiv \mathsf{let}\ \ g_1' := g_1\ A \quad g_2' := g_2\ (\mathsf{range}\ g_1')\ \mathsf{in}\ \ \mathsf{ap}_{pre}\ ((\mathsf{pre}\ g_1') \circ_{pre} (\mathsf{pre}\ g_2'))\ C \end{aligned} \tag{50}$$

$$\equiv \ \mathsf{let}\ \ \mathsf{h}_1 := \mathsf{lift}_{\mathsf{pre}}\ \mathsf{g}_1\ \mathsf{A}$$
$$\mathsf{h}_2 := \mathsf{lift}_{\mathsf{pre}}\ \mathsf{g}_2\ (\mathsf{range}_{\mathsf{pre}}\ \mathsf{h}_1)$$
$$\mathsf{in}\ \ \mathsf{ap}_{\mathsf{pre}}\ (\mathsf{h}_2 \circ_{\mathsf{pre}} \mathsf{h}_1)\ \mathsf{C}$$

Substituting h_1 for $\mathsf{lift}_{\mathsf{pre}}\ \mathsf{g}_1$ and h_2 for $\mathsf{lift}_{\mathsf{pre}}\ \mathsf{g}_2$, and removing the application of $\mathsf{ap}_{\mathsf{pre}}$ from both sides of the equivalence gives a definition of $(\ggg_{\mathsf{pre}})$ (Fig. 6) for which (15) holds.

7.3 Conditional

Starting with the left side of (17), we expand terms, apply Theorem 9, rewrite in terms of $\mathsf{lift}_{\mathsf{pre}}$, and apply Theorem 6 in h_2 and h_3:

$$\begin{aligned}
\mathsf{ap}_{\mathsf{pre}}\ (\mathsf{lift}_{\mathsf{pre}}\ (\mathsf{ifte}_{\mathsf{map}}\ \mathsf{g}_1\ \mathsf{g}_2\ \mathsf{g}_3)\ \mathsf{A})\ \mathsf{B} \ &\equiv\ \mathsf{let}\ \ \mathsf{g}'_1 := \mathsf{g}_1\ \mathsf{A} \\
&\qquad\ \ \mathsf{g}'_2 := \mathsf{g}_2\ (\mathsf{preimage}\ \mathsf{g}'_1\ \{\mathsf{true}\}) \\
&\qquad\ \ \mathsf{g}'_3 := \mathsf{g}_3\ (\mathsf{preimage}\ \mathsf{g}'_1\ \{\mathsf{false}\}) \\
&\qquad \mathsf{in}\ \ \mathsf{ap}_{\mathsf{pre}}\ (\mathsf{pre}\ (\mathsf{g}'_2 \uplus_{\mathsf{map}} \mathsf{g}'_3))\ \mathsf{B} \\
&\equiv\ \mathsf{let}\ \ \mathsf{g}'_1 := \mathsf{g}_1\ \mathsf{A} \\
&\qquad\ \ \mathsf{g}'_2 := \mathsf{g}_2\ (\mathsf{preimage}\ \mathsf{g}'_1\ \{\mathsf{true}\}) \\
&\qquad\ \ \mathsf{g}'_3 := \mathsf{g}_3\ (\mathsf{preimage}\ \mathsf{g}'_1\ \{\mathsf{false}\}) \\
&\qquad \mathsf{in}\ \ \mathsf{ap}_{\mathsf{pre}}\ ((\mathsf{pre}\ \mathsf{g}'_2) \uplus_{\mathsf{pre}} (\mathsf{pre}\ \mathsf{g}'_3))\ \mathsf{B} \\
&\equiv\ \mathsf{let}\ \ \mathsf{h}_1 := \mathsf{lift}_{\mathsf{pre}}\ \mathsf{g}_1\ \mathsf{A} \\
&\qquad\ \ \mathsf{h}_2 := \mathsf{lift}_{\mathsf{pre}}\ \mathsf{g}_2\ (\mathsf{ap}_{\mathsf{pre}}\ \mathsf{h}_1\ \{\mathsf{true}\}) \\
&\qquad\ \ \mathsf{h}_3 := \mathsf{lift}_{\mathsf{pre}}\ \mathsf{g}_3\ (\mathsf{ap}_{\mathsf{pre}}\ \mathsf{h}_1\ \{\mathsf{false}\}) \\
&\qquad \mathsf{in}\ \ \mathsf{ap}_{\mathsf{pre}}\ (\mathsf{h}_2 \uplus_{\mathsf{pre}} \mathsf{h}_3)\ \mathsf{B}
\end{aligned} \tag{51}$$

Substituting h_1 for $\mathsf{lift}_{\mathsf{pre}}\ \mathsf{g}_1$, h_2 for $\mathsf{lift}_{\mathsf{pre}}\ \mathsf{g}_2$ and h_3 for $\mathsf{lift}_{\mathsf{pre}}\ \mathsf{g}_3$, and removing the application of $\mathsf{ap}_{\mathsf{pre}}$ from both sides of the equivalence gives a definition of $\mathsf{ifte}_{\mathsf{pre}}$ (Fig. 6) for which (17) holds.

7.4 Laziness

Starting with the left side of (18), expand definitions, distribute pre over the branches of if, and rewrite in terms of $\mathsf{lift}_{\mathsf{pre}}\ (\mathsf{g}\ 0)$:

$$\begin{aligned}
\mathsf{ap}_{\mathsf{pre}}\ (\mathsf{lift}_{\mathsf{pre}}\ (\mathsf{lazy}_{\mathsf{map}}\ \mathsf{g})\ \mathsf{A})\ \mathsf{B} \ &\equiv\ \mathsf{let}\ \ \mathsf{g}' := \mathsf{if}\ (\mathsf{A} = \varnothing)\ \varnothing\ (\mathsf{g}\ 0\ \mathsf{A}) \\
&\qquad \mathsf{in}\ \ \mathsf{ap}_{\mathsf{pre}}\ (\mathsf{pre}\ \mathsf{g}')\ \mathsf{B} \\
&\equiv\ \mathsf{let}\ \ \mathsf{h} := \mathsf{if}\ (\mathsf{A} = \varnothing)\ (\mathsf{pre}\ \varnothing)\ (\mathsf{pre}\ (\mathsf{g}\ 0\ \mathsf{A})) \\
&\qquad \mathsf{in}\ \ \mathsf{ap}_{\mathsf{pre}}\ \mathsf{h}\ \mathsf{B} \\
&\equiv\ \mathsf{let}\ \ \mathsf{h} := \mathsf{if}\ (\mathsf{A} = \varnothing)\ (\mathsf{pre}\ \varnothing)\ (\mathsf{lift}_{\mathsf{pre}}\ (\mathsf{g}\ 0)\ \mathsf{A}) \\
&\qquad \mathsf{in}\ \ \mathsf{ap}_{\mathsf{pre}}\ \mathsf{h}\ \mathsf{B}
\end{aligned} \tag{52}$$

Substituting $\mathsf{h}\ 0$ for $\mathsf{lift}_{\mathsf{pre}}\ (\mathsf{g}\ 0)$ and removing the application of $\mathsf{ap}_{\mathsf{pre}}$ from both sides of the equivalence gives a definition for $\mathsf{lazy}_{\mathsf{pre}}$ (Fig. 6) for which (18) holds.

7.5 Correctness

Theorem 10 (preimage arrow correctness). $\mathsf{lift}_{\mathsf{pre}}$ *is a homomorphism.*

Proof. By construction. □

Corollary 3 (semantic correctness). *For all* e, $[\![e]\!]_{\mathsf{pre}} \equiv \mathsf{lift}_{\mathsf{pre}}\ [\![e]\!]_{\mathsf{map}}$.

As with the mapping arrow, preimage arrow computations can be unruly. We would like to assume that each $\mathsf{h} : \mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}$ acts as if it computes preimages under restricted mappings. The following equivalent property is easier to state, and makes proving the arrow laws simple.

Definition 11 (preimage arrow law). *Let* $\mathsf{h} : \mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}$. *If there exists a* $\mathsf{g} : \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}$ *such that* $\mathsf{h} \equiv \mathsf{lift}_{\mathsf{pre}}\ \mathsf{g}$, *then* h *obeys the* ***preimage arrow law****.*

By homomorphism of $\mathsf{lift}_{\mathsf{pre}}$, preimage arrow combinators preserve this law. It is therefore safe to assume that the preimage arrow law holds for all $\mathsf{h} : \mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}$.

Theorem 11. $\mathsf{lift}_{\mathsf{pre}}$ *is an arrow epimorphism.*

Proof. Follows from Theorem 10 and restriction of $\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}$ to instances for which the preimage arrow law (Definition 11) holds. □

Corollary 4. $\mathsf{arr}_{\mathsf{pre}}$, $(\&\&\&_{\mathsf{pre}})$, $(\ggg_{\mathsf{pre}})$, $\mathsf{ifte}_{\mathsf{pre}}$ *and* $\mathsf{lazy}_{\mathsf{pre}}$ *define an arrow.*

8 Preimages Under Partial, Probabilistic Functions

We have defined everything on the top of our roadmap:

$$\begin{array}{ccccc} \mathsf{X} \rightsquigarrow_{\perp} \mathsf{Y} & \xrightarrow{\mathsf{lift}_{\mathsf{map}}} & \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y} & \xrightarrow{\mathsf{lift}_{\mathsf{pre}}} & \mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y} \\ {\scriptstyle \eta_{\perp^*}}\downarrow & & \downarrow{\scriptstyle \eta_{\mathsf{map}^*}} & & \downarrow{\scriptstyle \eta_{\mathsf{pre}^*}} \\ \mathsf{X} \rightsquigarrow_{\perp^*} \mathsf{Y} & \xrightarrow[\mathsf{lift}_{\mathsf{map}^*}]{} & \mathsf{X} \underset{\mathsf{map}^*}{\rightsquigarrow} \mathsf{Y} & \xrightarrow[\mathsf{lift}_{\mathsf{pre}^*}]{} & \mathsf{X} \underset{\mathsf{pre}^*}{\rightsquigarrow} \mathsf{Y} \end{array} \tag{53}$$

and proved that $\mathsf{lift}_{\mathsf{map}}$ and $\mathsf{lift}_{\mathsf{pre}}$ are homomorphisms. Now we move down from all three top arrows simultaneously, and prove every morphism in (53) is an arrow homomorphism.

8.1 Motivation

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

$$\mathsf{geometric\ p} \ := \ \mathsf{if\ (random} < \mathsf{p)\ 0\ (1 + geometric\ p)} \tag{54}$$

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

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

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

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

8.2 Threading and Indexing

We clearly need to transform bottom, mapping, and preimage arrows so that they thread random sources. To ensure random sources contain enough numbers, they should be infinite.

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

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

Definition 12 (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

$$\begin{aligned} \mathsf{left}\ (\mathsf{p}/\,\mathsf{q}) &:= (\mathsf{p} - \tfrac{1}{2})/\,\mathsf{q} \\ \mathsf{right}\ (\mathsf{p}/\,\mathsf{q}) &:= (\mathsf{p} + \tfrac{1}{2})/\,\mathsf{q} \end{aligned} \tag{55}$$

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).

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

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

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

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

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

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

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

Fig. 7: AStore (associative store) arrow transformer definitions.

8.3 Applicative, Associative Store Transformer

We thread infinite binary trees through bottom, mapping, 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 applicative store arrow transformer's type constructor takes a store type s and an arrow type $x \rightsquigarrow_a y$:

$$\mathsf{AStore}\ s\ (x \rightsquigarrow_a y) \;::=\; J \Rightarrow (\langle s, x\rangle \rightsquigarrow_a y) \tag{56}$$

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

$$\begin{aligned} &\eta_{a^*} : (x \rightsquigarrow_a y) \Rightarrow \mathsf{AStore}\ s\ (x \rightsquigarrow_a y) \\ &\eta_{a^*}\ f\ j \;:=\; \mathsf{arr}_a\ \mathsf{snd} \ggg_a f \end{aligned} \tag{57}$$

Fig. 7 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.

8.4 Partial, Probabilistic Programs

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

Definition 13 (random source). *Let* $R := J \to [0, 1]$. *A* ***random source*** *is any infinite binary tree* $r \in R$.

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

Definition 14 (branch trace). *A* ***branch trace*** *is any* $\mathsf{t} \in \mathsf{J} \to \mathsf{Bool}_\perp$ *such that* $\mathsf{t}\ \mathsf{j} = \mathsf{true}$ *or* $\mathsf{t}\ \mathsf{j} = \mathsf{false}$ *for no more than finitely many* $\mathsf{j} \in \mathsf{J}$.

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

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

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

$$\begin{aligned} &\mathsf{random}_{\mathsf{a}^*} : \mathsf{X} \rightsquigarrow_{\mathsf{a}^*} [0,1] \\ &\mathsf{random}_{\mathsf{a}^*}\ \mathsf{j}\ :=\ \mathsf{arr}_{\mathsf{a}}\ (\mathsf{fst} \ggg \mathsf{fst} \ggg \pi\ \mathsf{j}) \\ &[\![\mathsf{random}]\!]_{\mathsf{a}^*}\ :\equiv\ \mathsf{random}_{\mathsf{a}^*} \end{aligned} \tag{58}$$

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

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

$$\begin{aligned} &\mathsf{ifte}^{\Downarrow}_{\mathsf{a}^*} : (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{Bool}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y}) \\ &\mathsf{ifte}^{\Downarrow}_{\mathsf{a}^*}\ \mathsf{k}_1\ \mathsf{k}_2\ \mathsf{k}_3\ \mathsf{j}\ :=\ \mathsf{ifte}_{\mathsf{a}}\ ((\mathsf{k}_1\ (\mathsf{left}\ \mathsf{j})\ \&\&\&_{\mathsf{a}}\ \mathsf{branch}_{\mathsf{a}^*}\ \mathsf{j}) \ggg_{\mathsf{a}} \mathsf{arr}_{\mathsf{a}}\ \mathsf{agrees}) \\ &\qquad\qquad (\mathsf{k}_2\ (\mathsf{left}\ (\mathsf{right}\ \mathsf{j}))) \\ &\qquad\qquad (\mathsf{k}_3\ (\mathsf{right}\ (\mathsf{right}\ \mathsf{j}))) \end{aligned} \tag{59}$$

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

$$[\![\mathsf{if}\ e_c\ e_t\ e_f]\!]^{\Downarrow}_{\mathsf{a}^*}\ :\equiv\ \mathsf{ifte}^{\Downarrow}_{\mathsf{a}^*}\ [\![e_c]\!]^{\Downarrow}_{\mathsf{a}^*}\ [\![\mathsf{lazy}\ e_t]\!]^{\Downarrow}_{\mathsf{a}^*}\ [\![\mathsf{lazy}\ e_f]\!]^{\Downarrow}_{\mathsf{a}^*} \tag{60}$$

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

Definition 15 (terminating, probabilistic arrows). *Define*

$$\begin{aligned} \mathsf{X} \rightsquigarrow_{\perp^*} \mathsf{Y}\ &::=\ \mathsf{AStore}\ (\mathsf{R} \times \mathsf{T})\ (\mathsf{X} \rightsquigarrow_{\perp} \mathsf{Y}) \\ \mathsf{X} \underset{\mathsf{map}^*}{\rightsquigarrow} \mathsf{Y}\ &::=\ \mathsf{AStore}\ (\mathsf{R} \times \mathsf{T})\ (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}) \\ \mathsf{X} \underset{\mathsf{pre}^*}{\rightsquigarrow} \mathsf{Y}\ &::=\ \mathsf{AStore}\ (\mathsf{R} \times \mathsf{T})\ (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}) \end{aligned} \tag{61}$$

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

8.5 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 16 (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* $\mathsf{j} \in \mathsf{J}$*.*

Pure Expressions. Proving η_{a^*} is a homomorphism proves $[\![\cdot]\!]_{\mathsf{a}^*}$ correctly interprets pure expressions. Because AStore accepts any arrow type $\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}$, we can do so using only the arrow laws. From here on, we assume every AStore arrow's base type's combinators obey the arrow laws listed in Section 3.1.

Theorem 12 (pure AStore arrow correctness). η_{a^*} *is a homomorphism.*

Proof. Defining $\mathsf{arr}_{\mathsf{a}^*}$ as a composition clearly meets the first homomorphism law (14). For homomorphism laws (15)–(17), start from the right side, expand definitions, and use arrow laws (20)–(22) to factor out $\mathsf{arr}_{\mathsf{a}}\ \mathsf{snd}$.

For (18), additionally β-expand within the outer thunk, then use the lazy distributive law (23) to extract $\mathsf{arr}_{\mathsf{a}}\ \mathsf{snd}$. □

Corollary 5 (pure semantic correctness). *For all pure* e*,* $[\![e]\!]_{\mathsf{a}^*} \equiv \eta_{\mathsf{a}^*}\ [\![e]\!]_{\mathsf{a}}$*.*

Effectful Expressions. To prove all interpretations of effectful expressions correct, 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

$$\begin{aligned} &\mathsf{lift}_{\mathsf{b}^*} : (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{b}^*} \mathsf{y}) \\ &\mathsf{lift}_{\mathsf{b}^*}\ \mathsf{f}\ \mathsf{j}\ :=\ \mathsf{lift}_{\mathsf{b}}\ (\mathsf{f}\ \mathsf{j}) \end{aligned} \tag{62}$$

where $\mathsf{lift}_{\mathsf{b}} : (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{b}} \mathsf{y})$. This shows the relationships more clearly:

$$\begin{array}{ccc} \mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y} & \xrightarrow{\mathsf{lift}_{\mathsf{b}}} & \mathsf{x} \rightsquigarrow_{\mathsf{b}} \mathsf{y} \\ {\scriptstyle \eta_{\mathsf{a}^*}}\downarrow & & \downarrow{\scriptstyle \eta_{\mathsf{b}^*}} \\ \mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y} & \xrightarrow[\mathsf{lift}_{\mathsf{b}^*}]{} & \mathsf{x} \rightsquigarrow_{\mathsf{b}^*} \mathsf{y} \end{array} \tag{63}$$

At minimum, we should expect to produce equivalent $\mathsf{x} \rightsquigarrow_{\mathsf{b}^*} \mathsf{y}$ computations from $\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}$ computations whether a lift or an η is done first.

Theorem 13 (natural transformation). *If* $\mathsf{lift}_{\mathsf{b}}$ *is an arrow homomorphism, then* (63) *commutes.*

Proof. Expand definitions and apply homomorphism laws (15) and (14) for $\mathsf{lift_b}$:

$$\begin{aligned}
\mathsf{lift_{b^*}}\ (\eta_{\mathsf{a}^*}\ \mathsf{f}) &\equiv \lambda \mathsf{j}.\, \mathsf{lift_b}\ (\mathsf{arr_a}\ \mathsf{snd} \ggg_{\mathsf{a}} \mathsf{f}) \\
&\equiv \lambda \mathsf{j}.\, \mathsf{lift_b}\ (\mathsf{arr_a}\ \mathsf{snd}) \ggg_{\mathsf{b}} \mathsf{lift_b}\ \mathsf{f} \\
&\equiv \lambda \mathsf{j}.\, \mathsf{arr_b}\ \mathsf{snd} \ggg_{\mathsf{b}} \mathsf{lift_b}\ \mathsf{f} \\
&\equiv \eta_{\mathsf{b}^*}\ (\mathsf{lift_b}\ \mathsf{f})
\end{aligned} \tag{64}$$

□

Theorem 14 (effectful AStore arrow correctness). *If* $\mathsf{lift_b}$ *is an arrow homomorphism from* a *to* b*, then* $\mathsf{lift_{b^*}}$ *is an arrow homomorphism from* a^* *to* b^**.*

Proof. For each homomorphism property (14)–(18), expand the definitions of $\mathsf{lift_{b^*}}$ and the combinator, distribute $\mathsf{lift_b}$, rewrite in terms of $\mathsf{lift_{b^*}}$, and rewrite using the definition of the combinator. For example, for distribution over pairing:

$$\begin{aligned}
\mathsf{lift_{b^*}}\ (\mathsf{k}_1 \mathbin{\&\&\&_{\mathsf{a}^*}} \mathsf{k}_2)\ \mathsf{j} &\equiv \mathsf{lift_b}\ ((\mathsf{k}_1 \mathbin{\&\&\&_{\mathsf{a}^*}} \mathsf{k}_2)\ \mathsf{j}) \\
&\equiv \mathsf{lift_b}\ (\mathsf{k}_1\ (\mathsf{left}\ \mathsf{j}) \mathbin{\&\&\&_{\mathsf{a}}} \mathsf{k}_2\ (\mathsf{right}\ \mathsf{j})) \\
&\equiv \mathsf{lift_b}\ (\mathsf{k}_1\ (\mathsf{left}\ \mathsf{j})) \mathbin{\&\&\&_{\mathsf{b}}} \mathsf{lift_b}\ (\mathsf{k}_2\ (\mathsf{right}\ \mathsf{j})) \\
&\equiv (\mathsf{lift_{b^*}}\ \mathsf{k}_1)\ (\mathsf{left}\ \mathsf{j}) \mathbin{\&\&\&_{\mathsf{b}}} (\mathsf{lift_{b^*}}\ \mathsf{k}_2)\ (\mathsf{right}\ \mathsf{j}) \\
&\equiv (\mathsf{lift_{b^*}}\ \mathsf{k}_1 \mathbin{\&\&\&_{\mathsf{b}^*}} \mathsf{lift_{b^*}}\ \mathsf{k}_2)\ \mathsf{j}
\end{aligned} \tag{65}$$

The remaining properties are similar, though distributing $\mathsf{lift_{b^*}}$ over $\mathsf{lazy_{a^*}}$ requires defining an extra thunk in the last step. □

Corollary 6 (effectful semantic correctness). *If* $\mathsf{lift_b}$ *is an arrow homomorphism, then for all expressions* e*,* $[\![e]\!]_{\mathsf{b}^*} \equiv \mathsf{lift_{b^*}}\ [\![e]\!]_{\mathsf{a}^*}$ *and* $[\![e]\!]^{\Downarrow}_{\mathsf{b}^*} \equiv \mathsf{lift_{b^*}}\ [\![e]\!]^{\Downarrow}_{\mathsf{a}^*}$*.*

Corollary 7 (mapping* and preimage* arrow correctness). *The following diagram commutes:*

$$\begin{array}{ccccc}
\mathsf{X} \rightsquigarrow_{\perp} \mathsf{Y} & \xrightarrow{\mathsf{lift_{map}}} & \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y} & \xrightarrow{\mathsf{lift_{pre}}} & \mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y} \\
{\scriptstyle \eta_{\perp^*}}\downarrow & & \downarrow{\scriptstyle \eta_{\mathsf{map}^*}} & & \downarrow{\scriptstyle \eta_{\mathsf{pre}^*}} \\
\mathsf{X} \rightsquigarrow_{\perp^*} \mathsf{Y} & \xrightarrow[\mathsf{lift_{map^*}}]{} & \mathsf{X} \underset{\mathsf{map}^*}{\rightsquigarrow} \mathsf{Y} & \xrightarrow[\mathsf{lift_{pre^*}}]{} & \mathsf{X} \underset{\mathsf{pre}^*}{\rightsquigarrow} \mathsf{Y}
\end{array} \tag{66}$$

Further, $\mathsf{lift_{map^*}}$ *and* $\mathsf{lift_{pre^*}}$ *are arrow homomorphisms.*

Corollary 8 (effectful semantic correctness). *For all expressions* e*,*

$$\begin{aligned}
[\![e]\!]_{\mathsf{pre}^*} &\equiv \mathsf{lift_{pre^*}}\ (\mathsf{lift_{map^*}}\ [\![e]\!]_{\perp^*}) \\
[\![e]\!]^{\Downarrow}_{\mathsf{pre}^*} &\equiv \mathsf{lift_{pre^*}}\ (\mathsf{lift_{map^*}}\ [\![e]\!]^{\Downarrow}_{\perp^*})
\end{aligned} \tag{67}$$

8.6 Termination

Here, we relate $[\![e]\!]^{\Downarrow}_{\mathsf{a}^*}$ computations, which are interpreted using $\mathsf{ifte}^{\Downarrow}_{\mathsf{a}^*}$ and should always terminate, with $[\![e]\!]_{\mathsf{a}^*}$ computations, which are interpreted using $\mathsf{ifte_{a^*}}$ and may not terminate. To do so, we need to find the largest domain on which $[\![e]\!]^{\Downarrow}_{\mathsf{a}^*}$ and $[\![e]\!]_{\mathsf{a}^*}$ should agree.

Definition 17 (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* $\mathsf{g} : \mathsf{X} \overset{\rightsquigarrow}{\underset{\mathsf{map}}{}} \mathsf{Y}$, $\mathsf{domain}\ (\mathsf{g}\ \mathsf{A}^*) = \mathsf{A}^*$.
- *For* $\mathsf{h} : \mathsf{X} \overset{\rightsquigarrow}{\underset{\mathsf{pre}}{}} \mathsf{Y}$, $\mathsf{domain}_{\mathsf{pre}}\ (\mathsf{h}\ \mathsf{A}^*) = \mathsf{A}^*$.

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. It is not too hard to show (but is a bit tedious) that lifting computations preserves the maximal domain; e.g. the maximal domain of $\mathsf{lift}_{\mathsf{map}}\ \mathsf{f}$ is the same as f's, and the maximal domain of $\mathsf{lift}_{\mathsf{pre}^*}\ \mathsf{g}$ is the same as g's.

To ensure maximal domains exist, we need the domain operations above to have certain properties. For the mapping arrow, we must first make the intuition that computations "act as if they return restricted mappings" more precise.

Theorem 15 (mapping arrow restriction). *Let* $\mathsf{g} : \mathsf{X} \overset{\rightsquigarrow}{\underset{\mathsf{map}}{}} \mathsf{Y}$, *and* $\mathsf{A}^{\Downarrow} \subseteq \mathsf{X}$ *be the largest for which* $\mathsf{g}\ \mathsf{A}^{\Downarrow}$ *terminates. For all* $\mathsf{A} \subseteq \mathsf{A}^{\Downarrow}$, $\mathsf{g}\ \mathsf{A} = \mathsf{restrict}\ (\mathsf{g}\ \mathsf{A}^{\Downarrow})\ \mathsf{A}$.

Proof. By the mapping arrow law (Definition 8) there is an $\mathsf{f} : \mathsf{X} \rightsquigarrow_\perp \mathsf{Y}$ such that $\mathsf{g} \equiv \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}$. Thus,

$$\begin{aligned}
\mathsf{restrict}\ (\mathsf{g}\ \mathsf{A}^{\Downarrow})\ \mathsf{A} &\equiv \mathsf{restrict}\ (\mathsf{lift}_{\mathsf{map}}\ \mathsf{f}\ \mathsf{A}^{\Downarrow})\ \mathsf{A} \\
&\equiv \mathsf{restrict}\ (\{\langle \mathsf{a}, \mathsf{b}\rangle \in \mathsf{mapping}\ \mathsf{f}\ \mathsf{A}^{\Downarrow} \mid \mathsf{b} \neq \perp\})\ \mathsf{A} \\
&\equiv \{\langle \mathsf{a}, \mathsf{b}\rangle \in \mathsf{mapping}\ \mathsf{f}\ \mathsf{A} \mid \mathsf{b} \neq \perp\} \\
&\equiv \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}\ \mathsf{A} \\
&\equiv \mathsf{g}\ \mathsf{A}
\end{aligned} \tag{68}$$

□

Theorem 16 (domain closure operators). *If* $\mathsf{f} : \mathsf{X} \rightsquigarrow_\perp \mathsf{Y}$, $\mathsf{g} : \mathsf{X} \overset{\rightsquigarrow}{\underset{\mathsf{map}}{}} \mathsf{Y}$ *and* $\mathsf{h} : \mathsf{X} \overset{\rightsquigarrow}{\underset{\mathsf{pre}}{}} \mathsf{Y}$, *then* $\mathsf{domain}_\perp\ \mathsf{f}$, $\mathsf{domain} \circ \mathsf{g}$, *and* $\mathsf{domain}_{\mathsf{pre}} \circ \mathsf{h}$ *are monotone, decreasing, and idempotent in the subdomains on which they terminate.*

Proof. These properties follow from the same properties of selection, restriction, and of preimages of images. □

Now we can relate $[\![e]\!]^{\Downarrow}_{\perp^*}$ computations to $[\![e]\!]_{\perp^*}$ computations. First, for any input for which $[\![e]\!]_{\perp^*}$ terminates, there should be a branch trace for which $[\![e]\!]^{\Downarrow}_{\perp^*}$ returns the correct output; it should otherwise return $\perp$.

Theorem 17. *Let* $\mathsf{f} := [\![e]\!]_{\perp^*} : \mathsf{X} \rightsquigarrow_{\perp^*} \mathsf{Y}$ *with maximal domain* A^*, *and* $\mathsf{f}' := [\![e]\!]^{\Downarrow}_{\perp^*}$. *For all* $\langle\langle \mathsf{r}, \mathsf{t}\rangle, \mathsf{a}\rangle \in \mathsf{A}^*$, *there exists a* $\mathsf{T}' \subseteq \mathsf{T}$ *such that*

- *If* $\mathsf{t}' \in \mathsf{T}'$ *then* $\mathsf{f}'\ \mathsf{j}_0\ \langle\langle \mathsf{r}, \mathsf{t}'\rangle, \mathsf{a}\rangle = \mathsf{f}\ \mathsf{j}_0\ \langle\langle \mathsf{r}, \mathsf{t}\rangle, \mathsf{a}\rangle$.
- *If* $\mathsf{t}' \in \mathsf{T} \backslash \mathsf{T}'$ *then* $\mathsf{f}'\ \mathsf{j}_0\ \langle\langle \mathsf{r}, \mathsf{t}'\rangle, \mathsf{a}\rangle = \perp$.

Proof. Define T' as the set of all $\mathsf{t}' \in \mathsf{J} \to \mathsf{Bool}_\perp$ such that $\mathsf{t}'\ \mathsf{j} = \mathsf{z}$ if the subcomputation with index j is an if whose test returns z. Because $\mathsf{f}\ \mathsf{j}_0\ \langle\langle \mathsf{r}, \mathsf{t}\rangle, \mathsf{a}\rangle$ terminates, $\mathsf{t}'\ \mathsf{j} \neq \perp$ for at most finitely many j, so each $\mathsf{t}' \in \mathsf{T}$.

Let $\mathsf{t}' \in \mathsf{T}'$. Because the test of every if subcomputation at index j agrees with $\mathsf{t}'\ \mathsf{j}$ and f ignores branch traces, $\mathsf{f}'\ \mathsf{j}_0\ \langle\langle \mathsf{r}, \mathsf{t}'\rangle, \mathsf{a}\rangle = \mathsf{f}\ \mathsf{j}_0\ \langle\langle \mathsf{r}, \mathsf{t}\rangle, \mathsf{a}\rangle$.

Let $\mathsf{t}' \in \mathsf{T} \backslash \mathsf{T}'$. There exists an if subexpression with a test that does not agree with t'; therefore $\mathsf{f}'\ \mathsf{j}_0\ \langle\langle \mathsf{r}, \mathsf{t}'\rangle, \mathsf{a}\rangle = \perp$. □

Next, for any input for which $[\![e]\!]_{\perp^*}$ does not terminate or returns $\perp$, $[\![e]\!]^{\Downarrow}_{\perp^*}$ should return $\perp$. Proving this is a little easier if we first identify subsets of J that correspond with finite prefixes of an infinite binary tree.

Definition 18 (index prefix/suffix). *A finite* $\mathsf{J}' \subset \mathsf{J}$ *is an* ***index prefix*** *if* $\mathsf{J}' = \{\mathsf{j}_0\}$ *or, for some index prefix* J'' *and* $\mathsf{j} \in \mathsf{J}''$, $\mathsf{J}' = \mathsf{J}'' \uplus \{\mathsf{left}\ \mathsf{j}\}$ *or* $\mathsf{J}' = \mathsf{J}'' \uplus \{\mathsf{right}\ \mathsf{j}\}$. *The corresponding* ***index suffix*** *is* $\mathsf{J} \backslash \mathsf{J}'$.

It is not hard to show that every index suffix is closed under left and right.

For a given $\mathsf{t} \in \mathsf{T}$, an index prefix J' serves as a convenient bounding set for the finitely many indexes j for which $\mathsf{t}\ \mathsf{j} \neq \perp$. Applying left and/or right repeatedly to any $\mathsf{j} \in \mathsf{J}'$ eventually yields a $\mathsf{j}' \in \mathsf{J} \backslash \mathsf{J}'$, for which $\mathsf{t}\ \mathsf{j}' = \perp$.

Theorem 18. *Let* $\mathsf{f} := [\![e]\!]_{\perp^*} : \mathsf{X} \rightsquigarrow_{\perp^*} \mathsf{Y}$ *with maximal domain* A^*, *and* $\mathsf{f}' := [\![e]\!]^{\Downarrow}_{\perp^*}$. *For all* $\mathsf{a} \in ((\mathsf{R} \times \mathsf{T}) \times \mathsf{X}) \backslash \mathsf{A}^*$, $\mathsf{f}'\ \mathsf{j}_0\ \mathsf{a} = \perp$.

Proof. Let $\mathsf{t} := \mathsf{snd}\ (\mathsf{fst}\ \mathsf{a})$ be the branch trace element of a.

Suppose $\mathsf{f}\ \mathsf{j}_0\ \mathsf{a}$ terminates. If an if subcomputation's test does not agree with t, then $\mathsf{f}'\ \mathsf{j}_0\ \mathsf{a} = \perp$. If every if's test agrees, $\mathsf{f}'\ \mathsf{j}_0\ \mathsf{a} = \mathsf{f}\ \mathsf{j}_0\ \mathsf{a} = \perp$.

Suppose $\mathsf{f}\ \mathsf{j}_0\ \mathsf{a}$ does not terminate. The set of all indexes j for which $\mathsf{t}\ \mathsf{j} \neq \perp$ is contained within an index prefix J'. By hypothesis, there is an if subcomputation at some index j' such that $\mathsf{j}' \in \mathsf{J} \backslash \mathsf{J}'$. Because $\mathsf{t}\ \mathsf{j}' = \perp$, $\mathsf{f}'\ \mathsf{j}_0\ \mathsf{a} = \perp$. □

Corollary 9. *For all* e, *the maximal domain of* $[\![e]\!]^{\Downarrow}_{\perp^*}$ *is a subset of that of* $[\![e]\!]_{\perp^*}$.

Corollary 10. *Let* $\mathsf{f}' := [\![e]\!]^{\Downarrow}_{\perp^*} : \mathsf{X} \rightsquigarrow_{\perp^*} \mathsf{Y}$ *with maximal domain* A^*, *and* $\mathsf{f} := [\![e]\!]_{\perp^*}$. *For all* $\mathsf{a} \in \mathsf{A}^*$, $\mathsf{f}'\ \mathsf{j}_0\ \mathsf{a} = \mathsf{f}\ \mathsf{j}_0\ \mathsf{a}$.

Corollary 11 (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* $\mathsf{B} \subseteq \mathsf{Y}$,

$$\begin{aligned}
[\![e]\!]^{\Downarrow}_{\perp^*}\ \ \mathsf{j}_0\ \mathsf{a} &= \mathsf{if}\ (\mathsf{a} \in \mathsf{A}^*)\ ([\![e]\!]_{\perp^*}\ \mathsf{j}_0\ \mathsf{a})\ \perp \\
[\![e]\!]^{\Downarrow}_{\mathsf{map}^*}\ \mathsf{j}_0\ \mathsf{A} &= [\![e]\!]_{\mathsf{map}^*}\ \mathsf{j}_0\ (\mathsf{A} \cap \mathsf{A}^*) \qquad (69) \\
\mathsf{ap}_{\mathsf{pre}}\ ([\![e]\!]^{\Downarrow}_{\mathsf{pre}^*}\ \mathsf{j}_0\ \mathsf{A})\ \mathsf{B} &= \mathsf{ap}_{\mathsf{pre}}\ ([\![e]\!]_{\mathsf{pre}^*}\ \mathsf{j}_0\ (\mathsf{A} \cap \mathsf{A}^*))\ \mathsf{B}
\end{aligned}$$

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.

9 Output Probabilities and Measurability

Typically, for $\mathsf{g} \in \mathsf{X} \rightharpoonup \mathsf{Y}$, the probability of $\mathsf{B} \subseteq \mathsf{Y}$ is $\mathsf{P}\ (\mathsf{preimage\ g\ B})$, where $\mathsf{P} \in \mathcal{P}\ \mathsf{X} \rightharpoonup [0,1]$ assigns probabilities to subsets of X.

However, a mapping* computation's domain is $(\mathsf{R} \times \mathsf{T}) \times \mathsf{X}$, not X. We assume each $\mathsf{r} \in \mathsf{R}$ is randomly chosen, but not each $\mathsf{t} \in \mathsf{T}$ nor each $\mathsf{x} \in \mathsf{X}$; therefore, neither T nor X should affect the probabilities of output sets. We clearly must measure *projections* of preimage sets, or $\mathsf{P}\ (\mathsf{image}\ (\mathsf{fst} \ggg \mathsf{fst})\ \mathsf{A})$ for preimage sets $\mathsf{A} \subseteq (\mathsf{R} \times \mathsf{T}) \times \mathsf{X}$.

Not all preimage sets have sensible measures. Sets that do are called **measurable**. Computing preimages and projecting them onto R must preserve measurability.

9.1 Measurability

From here on, we assume readers are familiar with topology or measure theory. While the remainder of this section is critical in that it establishes that the outputs of programs have sensible distributions, understanding it is not required to understand the rest of the paper. Therefore, readers unfamiliar with topology and meaure theory may skip to Section 10 without skipping prerequisite facts.

For readers familiar with topology but not measure theory, we review the necessary fundamentals by analogy to topology.

The analogue of a topology is a σ-algebra.

Definition 19 (σ-algebra, measurable set). *A collection of sets $\mathcal{A} \subseteq \mathcal{P}\ \mathsf{X}$ is called a **σ-algebra** on X if it contains X and is closed under complements and countable unions. The sets in $\mathcal{A}$ are called **measurable sets**.*

$\mathsf{X} \backslash \mathsf{X} = \varnothing$, so $\varnothing \in \mathcal{A}$. Additionally, it follows from De Morgan's law that $\mathcal{A}$ is closed under countable intersections.

The analogue of continuity is measurability.

Definition 20 (measurable mapping). *Let $\mathcal{A}$ and $\mathcal{B}$ be σ-algebras on X and Y. A mapping $\mathsf{g} : \mathsf{X} \rightharpoonup \mathsf{Y}$ is $\mathcal{A}$-$\mathcal{B}$-**measurable** if for all $\mathsf{B} \in \mathcal{B}$, $\mathsf{preimage\ g\ B} \in \mathcal{A}$.*

When the domain and codomain σ-algebras $\mathcal{A}$ and $\mathcal{B}$ are clear from context, we will simply say g is **measurable**.

Measurability is usually a weaker condition than continuity. For example, with respect to the σ-algebra generated from $\mathbb{R}$'s standard topology (i.e. using the standard topology as a sort of "base"), measurable $\mathbb{R} \rightharpoonup \mathbb{R}$ functions may have countably many discontinuities. Likewise, real comparison functions such as $(=)$, $(<)$, $(>)$ and their negations are measurable, but not continuous.

Product σ-algebras are defined analogously to product topologies.

Definition 21 (finite product σ-algebra). *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be σ-algebras on X_1 and X_2, and define $\mathsf{X} := \mathsf{X}_1 \times \mathsf{X}_2$. The **product σ-algebra** $\mathcal{A}_1 \otimes \mathcal{A}_2$ is the smallest (i.e. coarsest) σ-algebra for which $\mathsf{mapping\ fst\ X}$ and $\mathsf{mapping\ snd\ X}$ are measurable.*

Definition 22 (arbitrary product σ-algebra). *Let $\mathcal{A}$ be a σ-algebra on X. The* ***product σ-algebra*** *$\mathcal{A}^{\otimes \mathsf{J}}$ is the smallest σ-algebra for which, for all $\mathsf{j} \in \mathsf{J}$, $\mathsf{mapping}\ (\pi\ \mathsf{j})\ (\mathsf{J} \to \mathsf{X})$ is measurable.*

9.2 Measurable Pure Computations

It is easier to prove measurability of pure computations than to prove measurability of partial, probabilistic ones. Further, we can use the results to prove that the interpretations of all partial, probabilistic expressions are measurable.

We must first define what it means for a *computation* to be measurable.

Definition 23 (measurable mapping arrow computation). *Let $\mathcal{A}$ and $\mathcal{B}$ be σ-algebras on X and Y. A computation $\mathsf{g} : \mathsf{X} \overset{\rightsquigarrow}{\underset{\mathsf{map}}{\rightarrow}} \mathsf{Y}$ is $\mathcal{A}$-$\mathcal{B}$-**measurable** if $\mathsf{g}\ \mathsf{A}^*$ is an $\mathcal{A}$-$\mathcal{B}$-measurable mapping, where A^* is g's maximal domain.*

Theorem 19 (maximal domain measurability). *Let $\mathsf{g} : \mathsf{X} \overset{\rightsquigarrow}{\underset{\mathsf{map}}{\rightarrow}} \mathsf{Y}$ be an $\mathcal{A}$-$\mathcal{B}$-measurable mapping arrow computation. Its maximal domain A^* is in $\mathcal{A}$.*

Proof. Because $\mathsf{g}\ \mathsf{A}^*$ is measurable, $\mathsf{preimage}\ (\mathsf{g}\ \mathsf{A}^*)\ \mathsf{Y} = \mathsf{A}^*$ is in $\mathcal{A}$. □

Mapping arrow computations can be applied to sets other than their maximal domains. We need to ensure doing so yields a measurable mapping, at least for measurable subsets of A^*. Fortunately, that is true without any extra conditions.

Lemma 6. *Let $\mathsf{g} : \mathsf{X} \rightharpoonup \mathsf{Y}$ be an $\mathcal{A}$-$\mathcal{B}$-measurable mapping. For any $\mathsf{A} \in \mathcal{A}$, $\mathsf{restrict}\ \mathsf{g}\ \mathsf{A}$ is $\mathcal{A}$-$\mathcal{B}$-measurable.*

Theorem 20. *Let $\mathsf{g} : \mathsf{X} \overset{\rightsquigarrow}{\underset{\mathsf{map}}{\rightarrow}} \mathsf{Y}$ be an $\mathcal{A}$-$\mathcal{B}$-measurable mapping arrow computation with maximal domain A^*. For all $\mathsf{A} \subseteq \mathsf{A}^*$ with $\mathsf{A} \in \mathcal{A}$, $\mathsf{g}\ \mathsf{A}$ is $\mathcal{A}$-$\mathcal{B}$-measurable.*

Proof. By Theorem 15 (mapping arrow restriction) and Lemma 6. □

We do not need to prove all interpretations using $[\![\cdot]\!]_{\mathsf{a}}$ are measurable. However, we do need to prove mapping arrow combinators preserve measurability.

Composition. Proving compositions are measurable takes the most work. The main complication is that, under measurable mappings, while *preimages* of measurable sets are measurable, *images* of measurable sets may not be. We need the following four extra theorems to get around this.

Lemma 7 (images of preimages decrease). *Let $\mathsf{g} : \mathsf{X} \rightharpoonup \mathsf{Y}$ and $\mathsf{B} \subseteq \mathsf{Y}$. Then $\mathsf{image}\ \mathsf{g}\ (\mathsf{preimage}\ \mathsf{g}\ \mathsf{B}) \subseteq \mathsf{B}$.*

Lemma 8 (expanded post-composition). *Let $\mathsf{g}_1 : \mathsf{X} \rightharpoonup \mathsf{Y}$ and $\mathsf{g}_2 : \mathsf{Y} \rightharpoonup \mathsf{Z}$ such that $\mathsf{range}\ \mathsf{g}_1 \subseteq \mathsf{domain}\ \mathsf{g}_2$, and let $\mathsf{g}_2' : \mathsf{Y} \rightharpoonup \mathsf{Z}$ such that $\mathsf{g}_2 \subseteq \mathsf{g}_2'$. Then $\mathsf{g}_2 \circ_{\mathsf{map}} \mathsf{g}_1 = \mathsf{g}_2' \circ_{\mathsf{map}} \mathsf{g}_1$.*

Theorem 21 (mapping arrow monotonicity). *Let $\mathsf{g} : \mathsf{X} \overset{\rightsquigarrow}{\underset{\mathsf{map}}{\rightarrow}} \mathsf{Y}$. For any $\mathsf{A}' \subseteq \mathsf{A} \subseteq \mathsf{A}^*$, $\mathsf{g}\ \mathsf{A}' \subseteq \mathsf{g}\ \mathsf{A}$.*

Proof. By Theorem 15 (mapping arrow restriction). □

Theorem 22 (maximal domain subsets). *Let* $\mathsf{g} : \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}$. *For any* $\mathsf{A} \subseteq \mathsf{A}^*$, $\mathsf{domain}\ (\mathsf{g}\ \mathsf{A}) = \mathsf{A}$.

Proof. Follows from Theorem 16. □

Now we can prove measurability.

Lemma 9 ($(\circ_{\mathsf{map}})$ measurability). *If* $\mathsf{g}_1 : \mathsf{X} \rightharpoonup \mathsf{Y}$ *is* $\mathcal{A}$-$\mathcal{B}$*-measurable and* $\mathsf{g}_2 : \mathsf{Y} \rightharpoonup \mathsf{Z}$ *is* $\mathcal{B}$-$\mathcal{C}$*-measurable, then* $\mathsf{g}_2 \circ_{\mathsf{map}} \mathsf{g}_1$ *is* $\mathcal{A}$-$\mathcal{C}$*-measurable.*

Theorem 23 ($(\ggg_{\mathsf{map}})$ measurability). *If* $\mathsf{g}_1 : \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}$ *is* $\mathcal{A}$-$\mathcal{B}$*-measurable and* $\mathsf{g}_2 : \mathsf{Y} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Z}$ *is* $\mathcal{B}$-$\mathcal{C}$*-measurable, then* $\mathsf{g}_1 \ggg_{\mathsf{map}} \mathsf{g}_2$ *is* $\mathcal{A}$-$\mathcal{C}$*-measurable.*

Proof. Let $\mathsf{A}^* \in \mathcal{A}$ and $\mathsf{B}^* \in \mathcal{B}$ be respectively g_1's and g_2's maximal domains. The maximal domain of $\mathsf{g}_1 \ggg_{\mathsf{map}} \mathsf{g}_2$ is $\mathsf{A}^{**} := \mathsf{preimage}\ (\mathsf{g}_1\ \mathsf{A}^*)\ \mathsf{B}^*$, which is in $\mathcal{A}$. By definition,

$$
\begin{aligned}
(\mathsf{g}_1 \ggg_{\mathsf{map}} \mathsf{g}_2)\ \mathsf{A}^{**} \;=\; \mathsf{let}\ \ & \mathsf{g}_1' := \mathsf{g}_1\ \mathsf{A}^{**} \\
& \mathsf{g}_2' := \mathsf{g}_2\ (\mathsf{range}\ \mathsf{g}_1') \\
\mathsf{in}\ \ & \mathsf{g}_2' \circ_{\mathsf{map}} \mathsf{g}_1'
\end{aligned}
\tag{70}
$$

By Theorem 20, g_1' is an $\mathcal{A}$-$\mathcal{B}$-measurable mapping. Unfortunately, g_2' may not be $\mathcal{B}$-$\mathcal{C}$-measurable when $\mathsf{range}\ \mathsf{g}_1' \notin \mathcal{B}$.

Let $\mathsf{g}_2'' := \mathsf{g}_2\ \mathsf{B}^*$, which is a $\mathcal{B}$-$\mathcal{C}$-measurable mapping. By Lemma 9, $\mathsf{g}_2'' \circ_{\mathsf{map}} \mathsf{g}_1'$ is $\mathcal{A}$-$\mathcal{C}$-measurable. We need only show that $\mathsf{g}_2' \circ_{\mathsf{map}} \mathsf{g}_1' = \mathsf{g}_2'' \circ_{\mathsf{map}} \mathsf{g}_1'$, which by Lemma 8 is true if $\mathsf{range}\ \mathsf{g}_1' \subseteq \mathsf{domain}\ \mathsf{g}_2'$ and $\mathsf{g}_2' \subseteq \mathsf{g}_2''$.

By Theorem 22, $\mathsf{A}^{**} \subseteq \mathsf{A}^*$ implies $\mathsf{domain}\ \mathsf{g}_1' = \mathsf{A}^{**}$. By Theorem 21 and Lemma 7,

$$
\begin{aligned}
\mathsf{range}\ \mathsf{g}_1' \;&=\; \mathsf{image}\ (\mathsf{g}_1\ \mathsf{A}^{**})\ (\mathsf{preimage}\ (\mathsf{g}_1\ \mathsf{A}^*)\ \mathsf{B}^*) \\
&=\; \mathsf{image}\ (\mathsf{g}_1\ \mathsf{A}^*)\ (\mathsf{preimage}\ (\mathsf{g}_1\ \mathsf{A}^*)\ \mathsf{B}^*) \\
&\subseteq\; \mathsf{B}^*
\end{aligned}
\tag{71}
$$

$\mathsf{range}\ \mathsf{g}_1' \subseteq \mathsf{B}^*$ implies (by Theorem 22) that $\mathsf{domain}\ \mathsf{g}_2' = \mathsf{range}\ \mathsf{g}_1'$, and (by Theorem 21) that $\mathsf{g}_2' \subseteq \mathsf{g}_2''$. □

Pairing. Proving pairing preserves measurability is straightforward given a corresponding theorem about mappings.

Lemma 10 ($\langle\cdot,\cdot\rangle_{\mathsf{map}}$ measurability). *If* $\mathsf{g}_1 : \mathsf{X} \rightharpoonup \mathsf{Y}_1$ *is* $\mathcal{A}$-$\mathcal{B}_1$*-measurable and* $\mathsf{g}_2 : \mathsf{X} \rightharpoonup \mathsf{Y}_2$ *is* $\mathcal{A}$-$\mathcal{B}_2$*-measurable, then* $\langle \mathsf{g}_1, \mathsf{g}_2 \rangle_{\mathsf{map}}$ *is* $\mathcal{A}$-$(\mathcal{B}_1 \otimes \mathcal{B}_2)$*-measurable.*

Theorem 24 ($(\&\&\&_{\mathsf{map}})$ measurability). *If* $\mathsf{g}_1 : \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}_1$ *is* $\mathcal{A}$-$\mathcal{B}_1$*-measurable and* $\mathsf{g}_2 : \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}_2$ *is* $\mathcal{A}$-$\mathcal{B}_2$*-measurable, then* $\mathsf{g}_1 \&\&\&_{\mathsf{map}} \mathsf{g}_2$ *is* $\mathcal{A}$-$(\mathcal{B}_1 \otimes \mathcal{B}_2)$*-measurable.*

Proof. Let A_1^* and A_2^* be respectively g_1's and g_2's maximal domains. The maximal domain of $\mathsf{g}_1 \mathbin{\&\&\&_{\mathsf{map}}} \mathsf{g}_2$ is $\mathsf{A}^{**} := \mathsf{A}_1^* \cap \mathsf{A}_2^*$, which is in $\mathcal{A}$. By definition, $(\mathsf{g}_1 \mathbin{\&\&\&_{\mathsf{map}}} \mathsf{g}_2)\ \mathsf{A}^{**} = \langle \mathsf{g}_1\ \mathsf{A}^{**}, \mathsf{g}_2\ \mathsf{A}^{**} \rangle_{\mathsf{map}}$, which by Lemma 10 is $\mathcal{A}$-$(\mathcal{B}_1 \otimes \mathcal{B}_2)$-measurable. □

Conditional. Conditionals can be proved measurable given a theorem that ensures the measurability of *finite* unions of disjoint, measurable mappings. We will need the corresponding theorem for *countable* unions further on, however.

Lemma 11 (union of measurable mappings). *The union of a countable set of $\mathcal{A}$-$\mathcal{B}$-measurable mappings with disjoint domains is $\mathcal{A}$-$\mathcal{B}$-measurable.*

Theorem 25 ($\mathsf{ifte}_{\mathsf{map}}$ measurability). *If $\mathsf{g}_1 : \mathsf{X} \overset{\rightsquigarrow}{\scriptscriptstyle\mathsf{map}} \mathsf{Bool}$, and $\mathsf{g}_2 : \mathsf{X} \overset{\rightsquigarrow}{\scriptscriptstyle\mathsf{map}} \mathsf{Y}$ and $\mathsf{g}_3 : \mathsf{X} \overset{\rightsquigarrow}{\scriptscriptstyle\mathsf{map}} \mathsf{Y}$ are respectively $\mathcal{A}$-$(\mathcal{P}\ \mathsf{Bool})$-measurable and $\mathcal{A}$-$\mathcal{B}$-measurable, then $\mathsf{ifte}_{\mathsf{map}}\ \mathsf{g}_1\ \mathsf{g}_2\ \mathsf{g}_3$ is $\mathcal{A}$-$\mathcal{B}$-measurable.*

Proof. Let $\mathcal{A}_1^*$, $\mathcal{A}_2^*$ and $\mathcal{A}_3^*$ be g_1's, g_2's and g_3's maximal domains. The maximal domain of $\mathsf{ifte}_{\mathsf{map}}\ \mathsf{g}_1\ \mathsf{g}_2\ \mathsf{g}_3$ is A^{**}, defined by

$$\begin{aligned} \mathsf{A}_2^{**} &:= \mathsf{A}_2^* \cap \mathsf{preimage}\ (\mathsf{g}_1\ \mathcal{A}_1^*)\ \{\mathsf{true}\} \\ \mathsf{A}_3^{**} &:= \mathsf{A}_3^* \cap \mathsf{preimage}\ (\mathsf{g}_1\ \mathcal{A}_1^*)\ \{\mathsf{false}\} \\ \mathsf{A}^{**} &:= \mathsf{A}_2^{**} \uplus \mathsf{A}_3^{**} \end{aligned} \tag{72}$$

Because $\mathsf{preimage}\ (\mathsf{g}_1\ \mathcal{A}_1^*)\ \mathsf{B} \in \mathcal{A}$ for any $\mathsf{B} \subseteq \mathsf{Bool}$, $\mathsf{A}^{**} \in \mathcal{A}$. By definition,

$$\begin{aligned} \mathsf{ifte}_{\mathsf{map}}\ \mathsf{g}_1\ \mathsf{g}_2\ \mathsf{g}_3\ \mathsf{A}^{**} \;=\; \mathsf{let}\ \ & \mathsf{g}_1' := \mathsf{g}_1\ \mathsf{A}^{**} \\ & \mathsf{g}_2' := \mathsf{g}_2\ (\mathsf{preimage}\ \mathsf{g}_1'\ \{\mathsf{true}\}) \\ & \mathsf{g}_3' := \mathsf{g}_3\ (\mathsf{preimage}\ \mathsf{g}_1'\ \{\mathsf{false}\}) \\ \mathsf{in}\ \ & \mathsf{g}_2' \uplus_{\mathsf{map}} \mathsf{g}_3' \end{aligned} \tag{73}$$

By hypothesis, g_1', g_2' and g_3' are measurable mappings. By Theorem 15 (mapping arrow restriction), g_2' and g_3' have disjoint domains. Apply Lemma 11. □

Laziness. We must first prove measurability of an often-ignored corner case.

Theorem 26 (measurability of $\varnothing$). *For any σ-algebras $\mathcal{A}$ and $\mathcal{B}$, the empty mapping $\varnothing$ is $\mathcal{A}$-$\mathcal{B}$-measurable.*

Proof. For any $\mathsf{B} \in \mathcal{B}$, $\mathsf{preimage}\ \varnothing\ \mathsf{B} = \varnothing$, and $\varnothing \in \mathcal{A}$. □

Theorem 27 (measurability under $\mathsf{lazy}_{\mathsf{map}}$). *Let $\mathsf{g} : 1 \Rightarrow (\mathsf{X} \overset{\rightsquigarrow}{\scriptscriptstyle\mathsf{map}} \mathsf{Y})$. If $\mathsf{g}\ 0$ is $\mathcal{A}$-$\mathcal{B}$-measurable, then $\mathsf{lazy}_{\mathsf{map}}\ \mathsf{g}$ is $\mathcal{A}$-$\mathcal{B}$-measurable.*

Proof. The maximal domain A^{**} of $\mathsf{lazy}_{\mathsf{map}}\ \mathsf{g}$ is that of $\mathsf{g}\ 0$. By definition,

$$\mathsf{lazy}_{\mathsf{map}}\ \mathsf{g}\ \mathsf{A}^{**} \;=\; \mathsf{if}\ (\mathsf{A}^{**} = \varnothing)\ \varnothing\ (\mathsf{g}\ 0\ \mathsf{A}^{**}) \tag{74}$$

If $\mathsf{A}^{**} = \varnothing$, then $\mathsf{lazy}_{\mathsf{map}}\ \mathsf{g}\ \mathsf{A}^{**} = \varnothing$; apply Theorem 26. If $\mathsf{A}^{**} \neq \varnothing$, then $\mathsf{lazy}_{\mathsf{map}}\ \mathsf{g} = \mathsf{g}\ 0$, which is $\mathcal{A}$-$\mathcal{B}$-measurable. □

9.3 Measurable Probabilistic Computations

As with pure computations, we must first define what it means for an effectful computation to be measurable.

Definition 24 (measurable mapping* arrow computation). *Let $\mathcal{A}$ and $\mathcal{B}$ be σ-algebras on $\mathsf{R}\times\mathsf{T})\times\mathsf{X}$ and Y. A computation $\mathsf{g} : \mathsf{X} \overset{\rightsquigarrow}{\underset{\mathsf{map}^*}{}} \mathsf{Y}$ is $\mathcal{A}$-$\mathcal{B}$-**measurable** if $\mathsf{g}\ \mathsf{j}_0$ is an $\mathcal{A}$-$\mathcal{B}$-measurable mapping arrow computation.*

Theorem 28. *If $\mathsf{g} : \mathsf{X} \overset{\rightsquigarrow}{\underset{\mathsf{map}^*}{}} \mathsf{Y}$ is $\mathcal{A}$-$\mathcal{B}$-measurable, then for all $\mathsf{j} \in \mathsf{J}$, $\mathsf{g}\ \mathsf{j}$ is an $\mathcal{A}$-$\mathcal{B}$-measurable mapping arrow computation.*

Proof. By induction on J: if $\mathsf{g}\ \mathsf{j}$ is measurable, so are $\mathsf{g}\ (\mathsf{left}\ \mathsf{j})$ and $\mathsf{g}\ (\mathsf{right}\ \mathsf{j})$. □

To make general measurability statements about computations, whether they have flat or product types, it helps to have a notion of a standard σ-algebra.

Definition 25 (standard σ-algebra). *For a set X used as a type, $\Sigma\ \mathsf{X}$ denotes its **standard σ-algebra**, which must be defined under the following constraints:*

$$\Sigma\ \langle \mathsf{X}_1, \mathsf{X}_2 \rangle = \Sigma\ \mathsf{X}_1 \otimes \Sigma\ \mathsf{X}_2 \tag{75}$$

$$\Sigma\ (\mathsf{J} \to \mathsf{X}) = (\Sigma\ \mathsf{X})^{\otimes \mathsf{J}} \tag{76}$$

From here on, when no σ-algebras are given, "measurable" means "measurable with respect to standard σ-algebras."

The following definitions allow distinguishing the results of conditional expressions and any two branch traces:

$$\Sigma\ \mathsf{Bool} ::= \mathcal{P}\ \mathsf{Bool} \tag{77}$$

$$\Sigma\ \mathsf{T} ::= \mathcal{P}\ \mathsf{T} \tag{78}$$

Lemma 12 (measurable mapping arrow lifts). $\mathsf{arr}_{\mathsf{map}}\ \mathsf{id}$, $\mathsf{arr}_{\mathsf{map}}\ \mathsf{fst}$ *and* $\mathsf{arr}_{\mathsf{map}}\ \mathsf{snd}$ *are measurable.* $\mathsf{arr}_{\mathsf{map}}\ (\mathsf{const}\ \mathsf{b})$ *is measurable if* $\{\mathsf{b}\}$ *is a measurable set. For all* $\mathsf{j} \in \mathsf{J}$, $\mathsf{arr}_{\mathsf{map}}\ (\pi\ \mathsf{j})$ *is measurable.*

Corollary 12 (measurable mapping* arrow lifts). $\mathsf{arr}_{\mathsf{map}^*}\ \mathsf{id}$, $\mathsf{arr}_{\mathsf{map}^*}\ \mathsf{fst}$ *and* $\mathsf{arr}_{\mathsf{map}^*}\ \mathsf{snd}$ *are measurable.* $\mathsf{arr}_{\mathsf{map}^*}\ (\mathsf{const}\ \mathsf{b})$ *is measurable if* $\{\mathsf{b}\}$ *is a measurable set.* $\mathsf{random}_{\mathsf{map}^*}$ *and* $\mathsf{branch}_{\mathsf{map}^*}$ *are measurable.*

Theorem 29 (AStore combinators preserve measurability). *Every* AStore *arrow combinator produces measurable mapping* computations from measurable mapping* computations.*

Proof. AStore's combinators are defined in terms of the base arrow's combinators and $\mathsf{arr}_{\mathsf{map}}\ \mathsf{fst}$ and $\mathsf{arr}_{\mathsf{map}}\ \mathsf{snd}$. □

Theorem 30 ($\mathsf{ifte}^{\Downarrow}_{\mathsf{map}^*}$ measurability). $\mathsf{ifte}^{\Downarrow}_{\mathsf{map}^*}$ *is measurable.*

Proof. $\mathsf{branch}_{\mathsf{map}^*}$ is measurable, and $\mathsf{arr}_{\mathsf{map}}\ \mathsf{agrees}$ is measurable by (77). □

We can now prove all nonrecursive programs measurable by induction.

Definition 26 (finite expression). *A **finite expression** is any expression for which no subexpression is a first-order application.*

Theorem 31 (all finite expressions are measurable). *For all finite expressions e, $[\![e]\!]_{\mathsf{map}^*}$ is measurable.*

Proof. By structural induction and the above theorems. □

Now all we need to do is represent recursive programs as a net of finite expressions, and take a sort of limit.

Theorem 32 (approximation with finite expressions). *Let $\mathsf{g} := [\![e]\!]^{\Downarrow}_{\mathsf{map}^*} : \mathsf{X} \overset{\rightsquigarrow}{\underset{\mathsf{map}^*}{}} \mathsf{Y}$ and $\mathsf{t} \in \mathsf{T}$. Define $\mathsf{A} := (\mathsf{R} \times \{\mathsf{t}\}) \times \mathsf{X}$. There is a finite expression e' for which $[\![e']\!]_{\mathsf{map}^*}\ \mathsf{j}_0\ \mathsf{A} = \mathsf{g}\ \mathsf{j}_0\ \mathsf{A}$.*

Proof. Let the index prefix J' contain every j for which $\mathsf{t}\ \mathsf{j} \neq \bot$. To construct e', exhaustively apply first-order functions in e, but replace any $\mathsf{ifte}^{\Downarrow}_{\mathsf{map}^*}$ whose index j is not in J' with the equivalent expression $\bot$. Because e is well-defined, recurrences must be guarded by if, so this process terminates after finitely many first-order applications. □

Theorem 33 (all probabilistic expressions are measurable). *For all expressions e, $[\![e]\!]^{\Downarrow}_{\mathsf{map}^*}$ is measurable.*

Proof. Let $\mathsf{g} := [\![e]\!]^{\Downarrow}_{\mathsf{map}^*}$ and $\mathsf{g}' := \mathsf{g}\ \mathsf{j}_0\ ((\mathsf{R} \times \mathsf{T}) \times \mathsf{X})$. By Corollary 11 (correct computation everywhere), $\mathsf{g}' = \mathsf{g}\ \mathsf{j}_0\ \mathsf{A}^*$ where A^* is g's maximal domain; thus we need only show g' is a measurable mapping.

By Theorem 15 (mapping arrow restriction),

$$\mathsf{g}' \;=\; \bigcup_{\mathsf{t} \in \mathsf{T}} \mathsf{g}\ \mathsf{j}_0\ ((\mathsf{R} \times \{\mathsf{t}\}) \times \mathsf{X}) \tag{79}$$

By Theorem 32 (approximation with finite expressions), for every $\mathsf{t} \in \mathsf{T}$, there is a finite expression whose interpretation agrees with g on $(\mathsf{R} \times \{\mathsf{t}\}) \times \mathsf{X}$. Therefore, by Theorem 31 (all finite expressions are measurable), $\mathsf{g}\ \mathsf{j}_0\ ((\mathsf{R} \times \{\mathsf{t}\}) \times \mathsf{X})$ is a measurable mapping. By Theorem 15 (mapping arrow restriction), they have disjoint domains. By Lemma 11 (union of measurable mappings), their union is measurable. □

Theorem 33 remains true when $[\![\cdot]\!]_{\mathsf{a}}$ is extended with any rule whose right side is measurable, including rules for real arithmetic, equality, inequality and limits. More generally, any continuous or (countably) piecewise continuous function can be made available as a language primitive, as long as its domain's and codomain's standard σ-algebras are generated from their topologies.

It is not difficult to compose $[\![\cdot]\!]_{\mathsf{a}}$ with another semantic function that defunctionalizes lambda expressions. Thus, the interpretations of all expressions in higher-order languages are measurable.

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

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

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

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

$$\mathsf{unproj}_2\ \mathsf{A}\ \mathsf{B} := \mathsf{A} \cap (\mathsf{proj}_1\ \mathsf{A} \times \mathsf{B})$$

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

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

Fig. 8: Preimage arrow lifts needed to interpret probabilistic programs.

9.4 Measurable Projections

If $\mathsf{g} := [\![e]\!]^{\Downarrow}_{\mathsf{map}^*} : \mathsf{X} \overset{\rightsquigarrow}{\underset{\mathsf{map}^*}{}} \mathsf{Y}$, the probability of a measurable output set $\mathsf{B} \in \Sigma\ \mathsf{Y}$ is

$$\mathsf{P}\ (\mathsf{image}\ (\mathsf{fst} \ggg \mathsf{fst})\ (\mathsf{preimage}\ (\mathsf{g}\ \mathsf{j}_0\ \mathsf{A}^*)\ \mathsf{B})) \tag{80}$$

Unfortunately, projections are generally not measurable. Fortunately, for interpretations of programs $[\![p]\!]^{\Downarrow}_{\mathsf{map}^*}$, for which $\mathsf{X} = \{\langle\rangle\}$, we have a special case.

Theorem 34 (measurable finite projections). *Let* $\mathsf{A} \in \Sigma\ \langle \mathsf{X}_1, \mathsf{X}_2\rangle$. *If* X_2 *is at most countable and* $\Sigma\ \mathsf{X}_2 = \mathcal{P}\ \mathsf{X}_2$, *then* $\mathsf{image}\ \mathsf{fst}\ \mathsf{A} \in \mathcal{A}_1$.

Proof. Because $\Sigma\ \mathsf{X}_2 = \mathcal{P}\ \mathsf{X}_2$, A is a countable union of rectangles of the form $\mathsf{A}_1 \times \{\mathsf{a}_2\}$, where $\mathsf{A}_1 \in \Sigma\ \mathsf{X}_1$ and $\mathsf{a}_2 \in \mathsf{X}_2$. Because $\mathsf{image}\ \mathsf{fst}$ distributes over unions, $\mathsf{image}\ \mathsf{fst}\ \mathsf{A}$ is a countable union of sets in $\Sigma\ \mathsf{X}_1$. □

Theorem 35. *Let* $\mathsf{g} : \mathsf{X} \overset{\rightsquigarrow}{\underset{\mathsf{map}^*}{}} \mathsf{Y}$ *be measurable. If* X *is at most countable and* $\Sigma\ \mathsf{X} = \mathcal{P}\ \mathsf{X}$, *for all* $\mathsf{B} \in \Sigma\ \mathsf{Y}$, $\mathsf{image}\ (\mathsf{fst} \ggg \mathsf{fst})\ (\mathsf{preimage}\ (\mathsf{g}\ \mathsf{j}_0\ \mathsf{A}^*)\ \mathsf{B}) \in \Sigma\ \mathsf{R}$.

Proof. T is countable and $\Sigma\ \mathsf{T} = \mathcal{P}\ \mathsf{T}$ by (78); apply Theorem 34 twice. □

In particular, for $[\![p]\!]^{\Downarrow}_{\mathsf{map}^*} : \{\langle\rangle\} \overset{\rightsquigarrow}{\underset{\mathsf{map}^*}{}} \mathsf{Y}$, the probabilities of $\Sigma\ \mathsf{Y}$ are well-defined.

10 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.

10.1 Implementable Lifts

We would like to be able to compute preimages of uncountable sets, such as real intervals—but $\mathsf{preimage}\ (\mathsf{g}\ \mathsf{A})\ \mathsf{B}$ is uncomputable for most uncountable sets A and 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. 2 (which defines $[\![\cdot]\!]_{\mathsf{a}}$) lifts id, $\mathsf{const}\ \mathsf{b}$, fst and snd. Section 8.4, which defines the combinators used to interpret partial, probabilistic programs, lifts $\pi\ \mathsf{j}$ and agrees. Measurable functions made available as language primitives of course must be lifted to the preimage arrow.

Fig. 8 gives explicit definitions for $\mathsf{arr}_{\mathsf{pre}}\ \mathsf{id}$, $\mathsf{arr}_{\mathsf{pre}}\ \mathsf{fst}$, $\mathsf{arr}_{\mathsf{pre}}\ \mathsf{snd}$, $\mathsf{arr}_{\mathsf{pre}}\ (\mathsf{const}\ \mathsf{b})$ and $\mathsf{arr}_{\mathsf{pre}}\ (\pi\ \mathsf{j})$. (We will deal with agrees separately.) To implement them, we must model sets in a way that makes $\mathsf{A} = \varnothing$ is decidable, and the following representable and finitely computable:

$$\begin{aligned}
&- \ \mathsf{A} \cap \mathsf{B},\ \varnothing,\ \{\mathsf{true}\},\ \{\mathsf{false}\} \text{ and } \{\mathsf{b}\} \text{ for every } \mathsf{const}\ \mathsf{b} \\
&- \ \mathsf{A}_1 \times \mathsf{A}_2,\ \mathsf{proj}_1\ \mathsf{A} \text{ and } \mathsf{proj}_2\ \mathsf{A} \\
&- \ \mathsf{J} \to \mathsf{X},\ \mathsf{proj}\ \mathsf{j}\ \mathsf{A} \text{ and } \mathsf{unproj}\ \mathsf{j}\ \mathsf{A}\ \mathsf{B}
\end{aligned} \tag{81}$$

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

Definition 27 (rectangular family). $\mathsf{Rect}\ \mathsf{X}$ *denotes the* ***rectangular family*** *of subsets of* X. $\mathsf{Rect}\ \mathsf{X}$ *must contain* $\varnothing$ *and* X*, and be closed under finite intersections. Products must satisfy the following rules:*

$$\mathsf{Rect}\ \langle \mathsf{X}_1, \mathsf{X}_2 \rangle \ = \ (\mathsf{Rect}\ \mathsf{X}_1) \boxtimes (\mathsf{Rect}\ \mathsf{X}_2) \tag{82}$$

$$\mathsf{Rect}\ (\mathsf{J} \to \mathsf{X}) \ = \ (\mathsf{Rect}\ \mathsf{X})^{\boxtimes \mathsf{J}} \tag{83}$$

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

$$\mathcal{A}_1 \boxtimes \mathcal{A}_2 \ := \ \{\mathsf{A}_1 \times \mathsf{A}_2 \mid \mathsf{A}_1 \in \mathcal{A}_1, \mathsf{A}_2 \in \mathcal{A}_2\} \tag{84}$$

$$\mathcal{A}^{\boxtimes \mathsf{J}} \ := \ \bigcup_{\mathsf{J}' \subset \mathsf{J}\ \mathit{finite}} \left\{ \prod_{\mathsf{j} \in \mathsf{J}} \mathsf{A}_{\mathsf{j}} \;\middle|\; \mathsf{A}_{\mathsf{j}} \in \mathcal{A}, \mathsf{j} \in \mathsf{J}' \iff \mathsf{A}_{\mathsf{j}} \subset \bigcup \mathcal{A} \right\} \tag{85}$$

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

Further, all of the operations in (81) can be exactly implemented if finite sets are modeled directly, sets in ordered spaces (such as $\mathbb{R}$) are modeled by intervals, and sets in $\mathsf{Rect}\ \langle \mathsf{X}_1, \mathsf{X}_2 \rangle$ are modeled by pairs of type $\langle \mathsf{Rect}\ \mathsf{X}_1, \mathsf{Rect}\ \mathsf{X}_2 \rangle$. By (85), sets in $\mathsf{Rect}\ (\mathsf{J} \to \mathsf{X})$ have no more than finitely many projections that are proper subsets of X. They can be modeled by *finite* binary trees, where unrepresented projections are implicitly X.

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

Theorem 36 (T model). *If* $\mathsf{T}' \in \mathsf{Rect}\ (\mathsf{J} \to \mathsf{Bool}_\perp)$ *and* $\mathsf{j} \in \mathsf{J}$*, then* $\mathsf{proj}\ \mathsf{j}\ \mathsf{T}' = \mathsf{proj}\ \mathsf{j}\ (\mathsf{T}' \cap \mathsf{T})$*. If* $\mathsf{B} \subseteq \mathsf{Bool}_\perp$*, then* $\mathsf{unproj}\ \mathsf{j}\ (\mathsf{T}' \cap \mathsf{T})\ \mathsf{B} = \mathsf{unproj}\ \mathsf{j}\ \mathsf{T}'\ \mathsf{B} \cap \mathsf{T}$.

Proof. Subset case is by projection monotonicity. For superset, let $\mathsf{b} \in \mathsf{proj\ j\ T'}$. Define t by $\mathsf{t\ j'} = \mathsf{b}$ if $\mathsf{j'} = \mathsf{j}$; $\mathsf{t\ j'} = \bot$ if $\bot \in \mathsf{proj\ j'\ T'}$; otherwise $\mathsf{t\ j'} \in \mathsf{proj\ j'\ T'}$.

By construction, $\mathsf{t} \in \mathsf{T'}$. For no more than finitely many $\mathsf{j'} \in \mathsf{J}$, $\mathsf{t\ j'} \neq \bot$, so $\mathsf{t} \in \mathsf{T}$. Thus, there exists a $\mathsf{t} \in \mathsf{T'} \cap \mathsf{T}$ such that $\mathsf{t\ j} = \mathsf{b}$, so $\mathsf{b} \in \mathsf{proj\ j\ (T' \cap T)}$.

The statement about unproj is an easy corollary. □

10.2 Approximate Preimage Mapping Operations

Implementing $\mathsf{lazy_{pre}}$ (defined in Fig. 6) 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. 5), we see that $\mathsf{ap_{pre}}$ is defined using $(\cap)$ and that $(\circ_{\mathsf{pre}})$ is defined using $\mathsf{ap_{pre}}$, 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 37 (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* $\mathsf{B} \subseteq \mathsf{Y_1} \times \mathsf{Y_2}$, $\mathsf{preimage}\ \langle \mathsf{g_1}, \mathsf{g_2} \rangle_{\mathsf{map}}\ \mathsf{B} \subseteq \mathsf{preimage}\ \mathsf{g_1}\ (\mathsf{proj_1}\ \mathsf{B}) \cap \mathsf{preimage}\ \mathsf{g_2}\ (\mathsf{proj_2}\ \mathsf{B})$.

Proof. By monotonicity of preimages and projections, and by Lemma 4. □

It is not hard to use Theorem 37 to show that

$$\begin{aligned} &\langle \cdot, \cdot \rangle'_{\mathsf{pre}} : (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y_1}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y_2}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y_1} \times \mathsf{Y_2}) \\ &\langle \langle \mathsf{Y'_1}, \mathsf{p_1} \rangle, \langle \mathsf{Y'_2}, \mathsf{p_2} \rangle \rangle'_{\mathsf{pre}} \ := \\ &\quad \langle \mathsf{Y'_1} \times \mathsf{Y'_2}, \lambda \mathsf{B}.\, \mathsf{p_1}\ (\mathsf{proj_1}\ \mathsf{B}) \cap \mathsf{p_2}\ (\mathsf{proj_2}\ \mathsf{B}) \rangle \end{aligned} \tag{86}$$

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:

$$\begin{aligned} (\mathsf{A_1} \times \mathsf{A_2}) \vee (\mathsf{B_1} \times \mathsf{B_2}) &= (\mathsf{A_1} \vee \mathsf{B_1}) \times (\mathsf{A_2} \vee \mathsf{B_2}) \\ (\textstyle\prod_{\mathsf{j} \in \mathsf{J}} \mathsf{A_j}) \vee (\prod_{\mathsf{j} \in \mathsf{J}} \mathsf{B_j}) &= \textstyle\prod_{\mathsf{j} \in \mathsf{J}} \mathsf{A_j} \vee \mathsf{B_j} \end{aligned} \tag{87}$$

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

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

$$\begin{aligned} &(\uplus'_{\mathsf{pre}}) : (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y}) \\ &\mathsf{h_1} \uplus'_{\mathsf{pre}} \mathsf{h_2} \ := \ \mathsf{let}\ \ \mathsf{Y'} := \mathsf{range_{pre}}\ \mathsf{h_1} \vee \mathsf{range_{pre}}\ \mathsf{h_2} \\ &\qquad\qquad\qquad\quad \mathsf{p} := \lambda \mathsf{B}.\, \mathsf{ap_{pre}}\ \mathsf{h_1}\ \mathsf{B} \vee \mathsf{ap_{pre}}\ \mathsf{h_2}\ \mathsf{B} \\ &\qquad\qquad\qquad\ \ \mathsf{in}\ \ \langle \mathsf{Y'}, \mathsf{p} \rangle \end{aligned} \tag{88}$$

To interpret programs that may not terminate, or that terminate with probability 1, we need to approximate $\mathsf{ife}^{\Downarrow}_{\mathsf{pre}^*}$ (59), 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 $\mathsf{ifte}^{\Downarrow}_{\mathsf{pre}^*}$ 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:

$$\begin{aligned} \mathsf{ifte}^{\Downarrow}_{\mathsf{pre}^*}\ \mathsf{k}_1\ \mathsf{k}_2\ \mathsf{k}_3\ \mathsf{j}\ \mathsf{A} \ \equiv\ \mathsf{let}\ \ \langle \mathsf{C_k}, \mathsf{p_k} \rangle &:= \mathsf{k}_1\ \mathsf{j}_1\ \mathsf{A} \\ \langle \mathsf{C_b}, \mathsf{p_b} \rangle &:= \mathsf{branch}_{\mathsf{pre}^*}\ \mathsf{j}\ \mathsf{A} \\ \mathsf{C}_2 &:= \mathsf{C_k} \cap \mathsf{C_b} \cap \{\mathsf{true}\} \\ \mathsf{C}_3 &:= \mathsf{C_k} \cap \mathsf{C_b} \cap \{\mathsf{false}\} \\ \mathsf{A}_2 &:= \mathsf{p_k}\ \mathsf{C}_2 \cap \mathsf{p_b}\ \mathsf{C}_2 \\ \mathsf{A}_3 &:= \mathsf{p_k}\ \mathsf{C}_3 \cap \mathsf{p_b}\ \mathsf{C}_3 \\ \mathsf{in}\ \ \mathsf{k}_2\ \mathsf{j}_2\ \mathsf{A}_2 &\uplus_{\mathsf{pre}} \mathsf{k}_3\ \mathsf{j}_3\ \mathsf{A}_3 \end{aligned} \tag{89}$$

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. 9b defines the final approximating preimage arrow. This arrow, the lifts in Fig. 8, and the semantic function $[\![\cdot]\!]_{\mathsf{a}}$ in Fig. 2 define an approximating semantics for partial, probabilistic programs.

10.3 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 38 (soundness). *For all* $\mathsf{A} \in \mathsf{Rect}\ \langle \langle \mathsf{R}, \mathsf{T} \rangle, \mathsf{X} \rangle$ *and* $\mathsf{B} \in \mathsf{Rect}\ \mathsf{Y}$, $\mathsf{ap}_{\mathsf{pre}}\ (\mathsf{h}\ \mathsf{j}_0\ \mathsf{A})\ \mathsf{B} \subseteq \mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}'\ \mathsf{j}_0\ \mathsf{A})\ \mathsf{B}$.

Proof. By construction. □

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

(c) Preimage* arrow combinators for probabilistic choice and guaranteed termination. Fig. 7 (AStore arrow transformer) defines η'_{pre^*}, $(\ggg'_{\mathsf{pre}^*})$, $(\&\&\&'_{\mathsf{pre}^*})$, $\mathsf{ifte}'_{\mathsf{pre}^*}$ and $\mathsf{lazy}'_{\mathsf{pre}^*}$.

Fig. 9: Implementable arrows that approximate preimage arrows. Specific lifts such as $\mathsf{fst}_{\mathsf{pre}} := \mathsf{arr}_{\mathsf{pre}}\ \mathsf{fst}$ are computable (see Fig. 8), but $\mathsf{arr}'_{\mathsf{pre}}$ is not.

To use structural induction on the interpretation of e, we need a theorem that allows representing it as a finite expression (Definition 26). Because $\mathsf{ifte}^{\Downarrow'}{}_{\mathsf{pre}^*}$ does not branch when either branch could be taken, an equivalent finite expression exists for each rectangular domain subset A.

Theorem 39 (equivalent finite expression). *Let* $\mathsf{A}' \in \mathsf{Rect}\ \langle \langle \mathsf{R}, \mathsf{T} \rangle, \mathsf{X} \rangle$. *There is a finite expression* e' *for which* $\mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}''\ \mathsf{j}_0\ \mathsf{A}')\ \mathsf{B} = \mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}'\ \mathsf{j}_0\ \mathsf{A}')\ \mathsf{B}$ *for all* $\mathsf{B} \in \mathsf{Rect\ Y}$, *where* $\mathsf{h}'' := [\![e']\!]^{\Downarrow'}{}_{\mathsf{pre}^*}$.

Proof. Let $\mathsf{T}' := \mathsf{proj}_2\ (\mathsf{proj}_1\ \mathsf{A}')$, and the index prefix J' contain every j' for which $(\mathsf{proj}\ \mathsf{j}'\ \mathsf{T}')\backslash\{\bot\}$ is either $\{\mathsf{true}\}$ or $\{\mathsf{false}\}$. To construct e', exhaustively apply first-order functions in e, but replace any $\mathsf{if}\ e_1\ e_2\ e_3$ whose index is not in J' with the equivalent expression $\mathsf{if}\ e_1\ \bot\ \bot$. Because e is well-defined, recurrences must be guarded by if, so this process terminates after finitely many applications. □

Corollary 13 (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 40 (monotonicity). $\mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}'\ \mathsf{j}_0\ \mathsf{A})\ \mathsf{B}$ *is monotone in both* A *and* B.

Proof. Lattice operators $(\cap)$ and $(\vee)$ are monotone, as is $(\times)$. Therefore, $\mathsf{id}_{\mathsf{pre}}$ and the other lifts in Fig. 8 are monotone, and each approximating preimage arrow combinator preserves monotonicity. Approximating preimage* arrow combinators, which are defined in terms of approximating preimage arrow combinators (Fig. 9b) likewise preserve monotonicity, as does η'_{pre^*}; therefore $\mathsf{id}_{\mathsf{pre}^*}$ and other lifts are monotone.

The definition of $\mathsf{ifte}^{\Downarrow'}_{\mathsf{pre}^*}$ can be written in terms of lattice operators and approximating preimage arrow combinators for any A for which $\mathsf{C_b} \subset \{\mathsf{true}, \mathsf{false}\}$, and thus preserves monotonicity in that case. If $\mathsf{C_b} = \{\mathsf{true}, \mathsf{false}\}$, which is an upper bound for $\mathsf{C_b}$, the returned value is an upper bound.

For monotonicity in A, suppose $\mathsf{A}_1 \subseteq \mathsf{A}_2$. Apply Theorem 39 with $\mathsf{A}' := \mathsf{A}_1$ to yield e'; clearly, it is also an equivalent finite expression for A_2. Monotonicity follows from structural induction on the interpretation of e'.

For monotonicity in B, use Theorem 39 with fixed $\mathsf{A}' := \mathsf{A}$. □

Theorem 41 (decreasing). *For all* $\mathsf{A} \in \mathsf{Rect}\ \langle\langle \mathsf{R}, \mathsf{T}\rangle, \mathsf{X}\rangle$ *and* $\mathsf{B} \in \mathsf{Rect}\ \mathsf{Y}$, $\mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}'\ \mathsf{j}_0\ \mathsf{A})\ \mathsf{B} \subseteq \mathsf{A}$.

Proof. Because they compute exact preimages of rectangular sets under restriction to rectangular domains, $\mathsf{id}_{\mathsf{pre}}$ and the other lifts in Fig. 8 are decreasing.

By definition and applying basic lattice properties,

$$
\begin{aligned}
\mathsf{ap}'_{\mathsf{pre}}\ ((\mathsf{h}_1 \ggg'_{\mathsf{pre}} \mathsf{h}_2)\ \mathsf{A})\ \mathsf{B} &\equiv \mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}_1\ \mathsf{A})\ \mathsf{B}' \ \text{ for some } \mathsf{B}' \qquad (90)\\
\mathsf{ap}'_{\mathsf{pre}}\ ((\mathsf{h}_1 \mathbin{\&\&\&}'_{\mathsf{pre}} \mathsf{h}_2)\ \mathsf{A})\ \mathsf{B} &\equiv \mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}_1\ \mathsf{A})\ (\mathsf{proj}_1\ \mathsf{B}) \cap \\
&\qquad\ \mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}_2\ \mathsf{A})\ (\mathsf{proj}_2\ \mathsf{B})\\
\mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{ifte}'_{\mathsf{pre}}\ \mathsf{h}_1\ \mathsf{h}_2\ \mathsf{h}_3\ \mathsf{A})\ \mathsf{B} &\equiv \mathsf{let}\ \ \mathsf{A}_2 := \mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}_1\ \mathsf{A})\ \{\mathsf{true}\}\\
&\qquad\qquad\ \ \mathsf{A}_3 := \mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}_1\ \mathsf{A})\ \{\mathsf{false}\}\\
&\qquad\qquad \mathsf{in}\ \ \mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}_2\ \mathsf{A}_2)\ \mathsf{B} \vee\\
&\qquad\qquad\qquad\ \mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}_3\ \mathsf{A}_3)\ \mathsf{B}\\
\mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{lazy}'_{\mathsf{pre}}\ \mathsf{h}\ \mathsf{A})\ \mathsf{B} &\equiv \mathsf{if}\ (\mathsf{A} = \varnothing)\ \varnothing\ (\mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}\ 0\ \mathsf{A})\ \mathsf{B})
\end{aligned}
$$

Thus, approximating preimage arrow combinators return decreasing computations when given decreasing computations. This property transfers trivially to approximating preimage* arrow combinators. Use Theorem 39 with $\mathsf{A}' := \mathsf{A}$, and structural induction. □

10.4 Preimage Refinement Algorithm

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

Definition 28 (preimage refinement algorithm). *Let* $\mathsf{B} \in \mathsf{Rect}\ \mathsf{Y}$ *and*

$$\begin{aligned} &\mathsf{refine} : \mathsf{Rect}\ \langle\langle \mathsf{R}, \mathsf{T}\rangle, \mathsf{X}\rangle \Rightarrow \mathsf{Rect}\ \langle\langle \mathsf{R}, \mathsf{T}\rangle, \mathsf{X}\rangle \\ &\mathsf{refine}\ \mathsf{A} := \mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}'\ \mathsf{j}_0\ \mathsf{A})\ \mathsf{B} \end{aligned} \tag{91}$$

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*

$$\begin{aligned} &\mathsf{refine}^* : \mathsf{Set}\ (\mathsf{Rect}\ \langle\langle \mathsf{R}, \mathsf{T}\rangle, \mathsf{X}\rangle) \Rightarrow \mathsf{Set}\ (\mathsf{Rect}\ \langle\langle \mathsf{R}, \mathsf{T}\rangle, \mathsf{X}\rangle) \\ &\mathsf{refine}^*\ \mathcal{A} := \mathsf{image}\ \mathsf{refine}\ \left(\textstyle\bigcup_{\mathsf{A}\in\mathcal{A}} \mathsf{partition}\ \mathsf{A}\right) \end{aligned} \tag{92}$$

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

Theorem 41 (decreasing) guarantees $\mathsf{refine}\ \mathsf{A}$ is never larger than A. Theorem 40 (monotonicity) guarantees refining a *partition* of A never does worse than refining A itself. Theorem 38 (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 $\{\mathsf{true}\}$ (the rational numbers) 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.

11 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***.

11.1 Direct Implementations

If sets are restricted to be finite, the arrows used as translation targets in the exact semantics, defined in Figs. 1, 3, 4, 5, 6 and 7, 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 [30] implementation useful for finding theorem candidates.

Given a rectangular set library, the approximating preimage arrows defined in Figs. 8 and 9b 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. They are at `https://github.com/ntoronto/writing/tree/master/2014esop-code`.

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.

11.2 Dr. Bayes

Our main implementation, ***Dr. Bayes***, is written in Typed Racket. It consists of the semantic function $[\![\cdot]\!]_{\mathsf{a}^*}$ from Fig. 2 and its extension $[\![\cdot]\!]^{\Downarrow}_{\mathsf{a}^*}$, the bottom* arrow as defined in Figs. 3 and 7, the approximating preimage and preimage* arrows as defined in Figs. 8 and 9b, 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.

$$\begin{aligned}\mathsf{mul}_{\mathsf{pre}} \;:=\; & \mathsf{ifte}_{\mathsf{pre}}\ (\mathsf{fst}_{\mathsf{pre}} \ggg_{\mathsf{pre}} \mathsf{pos?}_{\mathsf{pre}}) \\ & \quad (\mathsf{ifte}_{\mathsf{pre}}\ (\mathsf{snd}_{\mathsf{pre}} \ggg_{\mathsf{pre}} \mathsf{pos?}_{\mathsf{pre}}) \\ & \qquad \mathsf{mul}^{++}_{\mathsf{pre}} \\ & \qquad (\mathsf{ifte}_{\mathsf{pre}}\ (\mathsf{snd}_{\mathsf{pre}} \ggg_{\mathsf{pre}} \mathsf{neg?}_{\mathsf{pre}})\ \mathsf{mul}^{+-}_{\mathsf{pre}}\ (\mathsf{const}_{\mathsf{pre}}\ 0))) \\ & \quad \ldots\end{aligned} \tag{93}$$

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

Section 10.4 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 $\mathsf{g} : \mathsf{X} \overset{\rightsquigarrow}{\scriptscriptstyle\mathsf{map}} \mathsf{Y}$ 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

$$\mathsf{Pr}[\mathsf{B}'|\mathsf{B}] \approx \frac{\mathsf{length}\ (\mathsf{filter}\ (\in \mathsf{A}' \cap \mathsf{A})\ \mathsf{xs})}{\mathsf{length}\ (\mathsf{filter}\ (\in \mathsf{A})\ \mathsf{xs})} \tag{94}$$

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 (94), but we can use the fact that A and A′ are preimages, and use the interpretation of the program as a bottom arrow computation $\mathsf{f} : \mathsf{X} \rightsquigarrow_{\perp} \mathsf{Y}$:

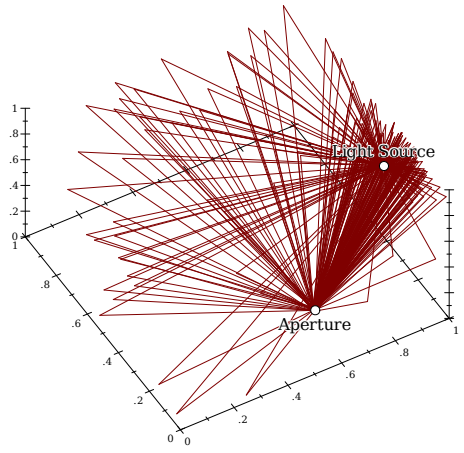
$$\begin{aligned} \mathsf{filter}\ (\in \mathsf{A})\ \mathsf{xs} &= \mathsf{filter}\ (\in \mathsf{preimage}\ (\mathsf{g}\ \mathsf{X})\ \mathsf{B})\ \mathsf{xs} \\ &= \mathsf{filter}\ (\lambda \mathsf{a}.\, \mathsf{g}\ \mathsf{X}\ \mathsf{a} \in \mathsf{B})\ \mathsf{xs} \\ &= \mathsf{filter}\ (\lambda \mathsf{a}.\, \mathsf{f}\ \mathsf{a} \in \mathsf{B})\ \mathsf{xs} \end{aligned} \tag{95}$$

Substituting into (94) gives

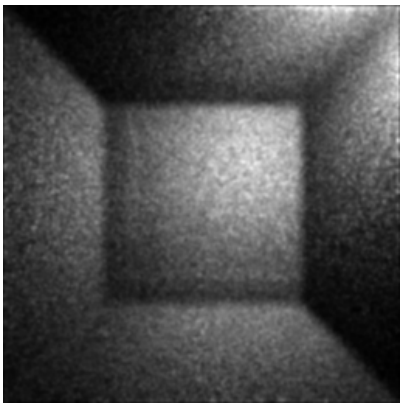
$$\mathsf{Pr}[\mathsf{B}'|\mathsf{B}] \approx \frac{\mathsf{length}\ (\mathsf{filter}\ (\lambda \mathsf{a}.\, \mathsf{f}\ \mathsf{a} \in \mathsf{B}' \cap \mathsf{B})\ \mathsf{xs})}{\mathsf{length}\ (\mathsf{filter}\ (\lambda \mathsf{a}.\, \mathsf{f}\ \mathsf{a} \in \mathsf{B})\ \mathsf{xs})} \tag{96}$$

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.



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



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

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

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

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

Fig. 10: Stochastic ray tracing in Dr. Bayes is little more than physics simulation.

Fig. 10 shows the result of using Dr. Bayes for stochastic ray tracing [33]. In this instance, photons are cast from a light source in a uniformly random direction and are reflected by the walls of a square room, generating paths. The objective is to sample, with the correct distribution, only those paths that pass through an aperture. The smaller the aperture, the smaller the probability a path passes through it, and the more focused the resulting image.

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

12 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 [24], BLOG [26], 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 [36,35] 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 [29] define the probability monad measure-theoretically and implement a language

for finite probability. Park [27] 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 [28] 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.

13 Conclusions and Future Work

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

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

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

Our semantics trades efficiency for simplicity by threading a constant, tree-shaped random source (Section 8.2). Passing subtrees instead would make random a constant-time primitive, and allow combinators to detect lack of change and return cached values. Other future optimization work includes creating new sampling algorithms, and using other easily measured but more expressive set representations, such as parallelotopes [2]. On the theory side, we intend to explore preimage computation's connection to type checking and type inference, investigate ways to integrate and leverage polymorphic type systems, and find the conditions under which preimage refinement is complete in the limit.

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

References

1. Haskell 98 language and libraries, the revised report (December 2002), `http://www.haskell.org/onlinereport/`

2. Amato, G., Scozzari, F.: The abstract domain of parallelotopes. Electronic Notes in Theoretical Computer Science 287, 17–28 (November 2012)
3. Aumann, R.J.: Borel structures for function spaces. Illinois Journal of Mathematics 5, 614–630 (1961)
4. Barras, B.: Sets in Coq, Coq in sets. Journal of Formalized Reasoning 3(1) (2010)
5. 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.: Idioms are oblivious, arrows are meticulous, monads are promiscuous. Electronic Notes in Theoretical Computer Science (2008)
23. Lindley, S., Wadler, P., Yallop, J.: The arrow calculus. Journal of Functional Programming 20, 51–69 (2010)
24. Lunn, D.J., Thomas, A., Best, N., Spiegelhalter, D.: WinBUGS – a Bayesian modelling framework. Statistics and Computing 10(4) (2000)
25. McBride, C., Paterson, R.: Applicative programming with effects. Journal of Functional Programming 18(1) (2008)

26. 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)
27. Park, S., Pfenning, F., Thrun, S.: A probabilistic language based upon sampling functions. Transactions on Programming Languages and Systems 31(1) (2008)
28. Pfeffer, A.: The design and implementation of IBAL: A general-purpose probabilistic language. In: Statistical Relational Learning. MIT Press (2007)
29. Ramsey, N., Pfeffer, A.: Stochastic lambda calculus and monads of probability distributions. In: Principles of Programming Languages (2002)
30. Tobin-Hochstadt, S., Felleisen, M.: The design and implementation of typed Scheme. In: Principles of Programming Languages (2008)
31. 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)
32. Toronto, N., McCarthy, J.: Computing in Cantor's paradise with λ_{ZFC}. In: Functional and Logic Programming Symposium (FLOPS). pp. 290–306 (2012)
33. Veach, E., Guibas, L.J.: Metropolis light transport. In: ACM SIGGRAPH. pp. 65–76 (1997)
34. Wadler, P.: Monads for functional programming. In: Jeuring, J., Meijer, E. (eds.) Advanced Functional Programming (2001)
35. Wingate, D., Goodman, N.D., Stuhlmüller, A., Siskind, J.M.: Nonstandard interpretations of probabilistic programs for efficient inference. In: Neural Information Processing Systems (2011)
36. Wingate, D., Stuhlmüller, A., Goodman, N.D.: Lightweight implementations of probabilistic programming languages via transformational compilation. In: Artificial Intelligence and Statistics (2011)