Running Probabilistic Programs Backward

Neil Toronto Jay McCarthy
PLT @ Brigham Young University
ntoronto@racket-lang.org jay@cs.byu.edu

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, extended with 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. As part of proving our semantics correct, we prove that all probabilistic programs are measurable regardless of nontermination, if the language's primitives are measurable. Such primitives include real arithmetic, inequalities and limits.

Because preimages are generally uncomputable, we develop an additional approximating semantics for computing rectangular covers of preimages. We implement the approximating semantics directly in Typed Racket and Haskell.

Categories and Subject Descriptors XXX-CR-number [*XXX-subcategory*]: XXX-third-level

General Terms XXX, XXX

Keywords XXX, XXX

> *This branch of mathematics [Probability] is the only one, I believe, in which good writers frequently get results which are entirely erroneous.*
>
> Charles S. Peirce

1. Introduction

Probability is notorious for being stubbornly counterintuitive. Any automation of probabilistic calculations or reasoning is therefore helpful. In Bayesian statistics, automation is taking the form of probabilistic languages for specifying random processes, which compute answers to questions about the processes under constraints.

We believe that any such language should be made to meet a mathematical specification. The reason is simple: if a probabilistic language implementation is made to always meet its maker's expectations, it is almost certainly wrong.

Unfortunately, there is currently no efficient probabilistic language implementation that simultaneously

1. Has a mathematical specification, or a **semantics**.
2. Allows **conditioning**, or imposing constraints in a way that preserves the relative probabilities of outputs.
3. Places no extraneous restrictions on legal programs.

The semantics defined in early work in probabilistic languages give meaning to all legal programs, but do not address probabilistic conditioning [15, 16, 21, 29]. There are many probabilistic languages defined by implementations rather than semantics, which support probabilistic conditioning in some form [6, 11, 18, 20, 24, 26, 34, 35]. Probabilitic conditioning has been incorporated in semantics only recently [5, 7, 28, 31]. So far, every language that supports probabilistic conditioning places extraneous restrictions on programs, most commonly disallowing recursion, allowing only discrete or continuous distributions, and restricting 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.

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 like this:

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

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

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

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

Bayesians interpret probabilistic processes as defining densities π_x and f_y, and use (2) to discover the density of x given $y = c$ for some constant c. While x given $\sin(y) = -1$ and x given $x + y = 0$ are perfectly sensible to reason about, Bayes'

law for densities cannot express them. Thus, reasoning with densities disallows all but the simplest conditions.

1.2 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 the **preimage measure**

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

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

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

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

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

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

1.3 Measure-Theoretic Semantics

Because purely functional languages do not allow side effects (except usually nontermination), programmers must write probabilistic programs as functions from a random source to outputs. Monads and other categorical classes such as idioms (i.e. applicative functors) can make doing so easier [15, 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, then 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 both 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 8. The outcome is worth it: we prove that 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. Our exact semantics consists of

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

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

From top-left to top-right, $\mathsf{X} \rightsquigarrow_{\perp} \mathsf{Y}$ computations are intensional functions that may raise errors, $\mathsf{X} \overset{\rightarrow}{\underset{\mathsf{map}}{\rightsquigarrow}} \mathsf{Y}$ computations produce extensional functions, and $\mathsf{X} \overset{\rightarrow}{\underset{\mathsf{pre}}{\rightsquigarrow}} \mathsf{Y}$ computations compute preimages. The computations of the arrows in the bottom row are like those in the top, except they thread an infinite store of random values, and always terminate. (We can do this because in λ_{ZFC}, Turing-uncomputable programs are definable.) Most of our correctness theorems rely on proofs that every lift and η in (5) is a homomorphism.

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

2. Operational Metalanguage

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

Generally, contemporary mathematics—measure theory in particular—is done in **ZFC**: **Zermelo-Fraenkel** set theory extended with the axiom of **Choice** (equivalently unique **Cardinality**). ZFC has only first-order functions and no general recursion, which makes implementing a language defined by a transformation into ZFC quite difficult. The problem is exacerbated if implementing the language requires approximation. Targeting λ_{ZFC} instead allows creating an

exact semantics and deriving an approximating semantics without changing languages.

In λ_{ZFC}, essentially every set is a value, as well as every lambda and every set of lambdas. All operations, including operations on infinite sets, are assumed to complete instantly if they terminate.

Almost everything definable in ZFC can be defined by a finite λ_{ZFC} program. Essentially every ZFC theorem applies to λ_{ZFC}'s set values without alteration. Further, proofs about λ_{ZFC}'s set values apply directly to ZFC sets, assuming the existence of an inaccessible cardinal.[1]

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

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

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

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 types. Similarly, the type $\langle \mathsf{X}, \mathsf{Y} \rangle$ is equivalent to $\mathsf{X} \times \mathsf{Y}$.

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

We import ZFC theorems as lemmas; for example:

Imported Lemma 2.1 (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 and External Equality

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

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

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

Definition 2.2 (observational equivalence)**.** *Two* λ_{ZFC} *terms* e_1 *and* e_2 *are* ***observationally equivalent****, written* $e_1 \equiv e_2$*, when* $e_1 = e_2$ *or both* e_1 *and* e_2 *do not terminate.*

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

2.2 Additional Functions and Forms

We assume a desugaring pass over λ_{ZFC} expressions, which automatically curries, and interprets special binding forms such as indexed unions $\bigcup_{x \in e_A} e$, destructuring binds as in $\mathsf{swap}\ \langle \mathsf{x}, \mathsf{y} \rangle := \langle \mathsf{y}, \mathsf{x} \rangle$, and comprehensions like $\{\mathsf{x} \in \mathsf{A} \mid \mathsf{x} \in \mathsf{B}\}$. We assume we have logical operators, bounded quantifiers, and typical set operations.

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 loops for any non-singleton argument. Thus, $\mathsf{A} \uplus \mathsf{B}$ terminates only when A and B are disjoint.

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

Syntax for unnamed mappings is defined by

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

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

where the primitive $\mathsf{image} : (\mathsf{x} \Rightarrow \mathsf{y}) \Rightarrow \mathsf{Set}\ \mathsf{x} \Rightarrow \mathsf{Set}\ \mathsf{y}$ is like map, but for sets. For symmetry with partial functions $\mathsf{x} \Rightarrow \mathsf{y}$, $\mathsf{mapping}$ returns a member of the set $\mathsf{X} \rightharpoonup \mathsf{Y}$ of all partial mappings from X to Y. Figure 1 defines other common mapping operations: domain, range, $\mathsf{preimage}$, $\mathsf{restrict}$, pairing, composition, and disjoint union. The latter three are particularly important in the preimage arrow's derivation.

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

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

which is particularly useful when f is unnamed.

3. Arrows and First-Order Semantics

Like monads and idioms [25, 33], arrows [14] are used to thread effects through computations in a way that imposes structure on the computations. Unlike monad and idiom computations, arrow computations are always

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

The first property makes arrows a perfect fit for a compositional translation from expressions to intensional or extensional functions—or, as we will see, to computations that compute preimages. The second property makes them a perfect fit for a measure-theoretic semantics in particular, as app in the function arrow is generally not measurable [3].

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

$$
\begin{aligned}
&\mathsf{domain} : (\mathsf{X} \rightharpoonup \mathsf{Y}) \Rightarrow \mathsf{Set}\ \mathsf{X}\\
&\mathsf{domain}\ :=\ \mathsf{image}\ \mathsf{fst}\\
&\\
&\mathsf{range} : (\mathsf{X} \rightharpoonup \mathsf{Y}) \Rightarrow \mathsf{Set}\ \mathsf{Y}\\
&\mathsf{range}\ :=\ \mathsf{image}\ \mathsf{snd}\\
&\\
&\mathsf{preimage} : (\mathsf{X} \rightharpoonup \mathsf{Y}) \Rightarrow \mathsf{Set}\ \mathsf{Y} \Rightarrow \mathsf{Set}\ \mathsf{X}\\
&\mathsf{preimage}\ \mathsf{f}\ \mathsf{B}\ :=\ \{\mathsf{a} \in \mathsf{domain}\ \mathsf{f} \mid \mathsf{f}\ \mathsf{a} \in \mathsf{B}\}\\
&\\
&\mathsf{restrict} : (\mathsf{X} \rightharpoonup \mathsf{Y}) \Rightarrow \mathsf{Set}\ \mathsf{X} \Rightarrow (\mathsf{X} \rightharpoonup \mathsf{Y})\\
&\mathsf{restrict}\ \mathsf{f}\ \mathsf{A}\ :=\ \lambda \mathsf{a} \in (\mathsf{A} \cap \mathsf{domain}\ \mathsf{f}).\, \mathsf{f}\ \mathsf{a}
\end{aligned}
$$

$$
\begin{aligned}
&\langle \cdot, \cdot \rangle_{\mathsf{map}} : (\mathsf{X} \rightharpoonup \mathsf{Y}_1) \Rightarrow (\mathsf{X} \rightharpoonup \mathsf{Y}_2) \Rightarrow (\mathsf{X} \rightharpoonup \mathsf{Y}_1 \times \mathsf{Y}_2)\\
&\langle \mathsf{g}_1, \mathsf{g}_2 \rangle_{\mathsf{map}}\ :=\ \mathsf{let}\ \ \mathsf{A} := (\mathsf{domain}\ \mathsf{g}_1) \cap (\mathsf{domain}\ \mathsf{g}_2)\\
&\qquad\qquad\qquad\qquad \mathsf{in}\ \ \lambda \mathsf{a} \in \mathsf{A}.\, \langle \mathsf{g}_1\ \mathsf{a}, \mathsf{g}_2\ \mathsf{a} \rangle\\
&\\
&(\circ_{\mathsf{map}}) : (\mathsf{Y} \rightharpoonup \mathsf{Z}) \Rightarrow (\mathsf{X} \rightharpoonup \mathsf{Y}) \Rightarrow (\mathsf{X} \rightharpoonup \mathsf{Z})\\
&\mathsf{g}_2 \circ_{\mathsf{map}} \mathsf{g}_1\ :=\ \mathsf{let}\ \ \mathsf{A} := \mathsf{preimage}\ \mathsf{g}_1\ (\mathsf{domain}\ \mathsf{g}_2)\\
&\qquad\qquad\qquad\qquad \mathsf{in}\ \ \lambda \mathsf{a} \in \mathsf{A}.\, \mathsf{g}_2\ (\mathsf{g}_1\ \mathsf{a})\\
&\\
&(\uplus_{\mathsf{map}}) : (\mathsf{X} \rightharpoonup \mathsf{Y}) \Rightarrow (\mathsf{X} \rightharpoonup \mathsf{Y}) \Rightarrow (\mathsf{X} \rightharpoonup \mathsf{Y})\\
&\mathsf{g}_1 \uplus_{\mathsf{map}} \mathsf{g}_2\ :=\ \mathsf{let}\ \ \mathsf{A} := (\mathsf{domain}\ \mathsf{g}_1) \uplus (\mathsf{domain}\ \mathsf{g}_2)\\
&\qquad\qquad\qquad\qquad \mathsf{in}\ \ \lambda \mathsf{a} \in \mathsf{A}.\, \mathsf{if}\ (\mathsf{a} \in \mathsf{domain}\ \mathsf{g}_1)\ (\mathsf{g}_1\ \mathsf{a})\ (\mathsf{g}_2\ \mathsf{a})
\end{aligned}
$$

Figure 1: Operations on mappings.

Targeting arrows in the semantics therefore gives some assurance that we can meet measure theory's requirement that preimage measure be defined only for measurable functions. We prove in Section 8 that it is sufficient.

3.1 Alternative Arrow Definitions and Laws

We do not give typical minimal arrow definitions. For each arrow a, instead of $\mathsf{first}_\mathsf{a}$, we define $(\&\&\&_\mathsf{a})$—typically called **fanout**, but its use will be clearer if we call it **pairing**—which applies two functions to an input and returns the pair of their outputs. Though $\mathsf{first}_\mathsf{a}$ may be defined in terms of $(\&\&\&_\mathsf{a})$ and vice-versa [14], we give $(\&\&\&_\mathsf{a})$ definitions because the applicable measure-theoretic theorems are in terms of pairing functions.

One way to strengthen an arrow a is to define an additional combinator left_a, which can be used to choose an arrow computation based on the result of another. Again, we define a different combinator, ifte_a ("if-then-else"), to make applying measure-theoretic theorems easier.

In a nonstrict λ-calculus, simply defining a choice combinator allows writing recursive functions using nothing but arrow combinators and lifted, pure functions. However, any strict λ-calculus (such as λ_{ZFC}) requires an extra combinator to defer computations in conditional branches. For example, define the **function arrow** with choice:

$$
\begin{aligned}
\mathsf{arr}\ \mathsf{f}\ &:=\ \mathsf{f}\\
(\mathsf{f}_1 \ggg \mathsf{f}_2)\ \mathsf{a}\ &:=\ \mathsf{f}_2\ (\mathsf{f}_1\ \mathsf{a})\\
(\mathsf{f}_1\ \&\&\&\ \mathsf{f}_2)\ \mathsf{a}\ &:=\ \langle \mathsf{f}_1\ \mathsf{a}, \mathsf{f}_2, \mathsf{a} \rangle\\
\mathsf{ifte}\ \mathsf{f}_1\ \mathsf{f}_2\ \mathsf{f}_3\ \mathsf{a}\ &:=\ \mathsf{if}\ (\mathsf{f}_1\ \mathsf{a})\ (\mathsf{f}_2\ \mathsf{a})\ (\mathsf{f}_3\ \mathsf{a})
\end{aligned}
\tag{10}
$$

and try to define the following recursive function:

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

The defining expression loops in a strict λ-calculus. In a nonstrict λ-calculus, it loops only when applied to false.

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

Definition 3.1 (arrow with choice)**.** *A binary type constructor* $(\rightsquigarrow_\mathsf{a})$ *and the combinators*

$$
\begin{aligned}
\mathsf{arr}_\mathsf{a} &: (\mathsf{x} \Rightarrow \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{y})\\
(\ggg_\mathsf{a}) &: (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{y}) \Rightarrow (\mathsf{y} \rightsquigarrow_\mathsf{a} \mathsf{z}) \Rightarrow (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{z})\\
(\&\&\&_\mathsf{a}) &: (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{z}) \Rightarrow (\mathsf{x} \rightsquigarrow_\mathsf{a} \langle \mathsf{y}, \mathsf{z} \rangle)
\end{aligned}
\tag{12}
$$

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

$$
\begin{aligned}
\mathsf{ifte}_\mathsf{a} &: (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{Bool}) \Rightarrow (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{y})\\
\mathsf{lazy}_\mathsf{a} &: (1 \Rightarrow (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{y})) \Rightarrow (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{y})
\end{aligned}
\tag{13}
$$

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

From here on, as all of our arrows are arrows with choice, we simply call them arrows.

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

Definition 3.2 (arrow homomorphism)**.** *A function* $\mathsf{lift}_\mathsf{b} : (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_\mathsf{b} \mathsf{y})$ *is an* ***arrow homomorphism*** *from arrow* a *to arrow* b *if the following distributive laws hold for appropriately typed* f*,* f_1*,* f_2 *and* f_3*:*

$$
\begin{aligned}
\mathsf{lift}_\mathsf{b}\ (\mathsf{arr}_\mathsf{a}\ \mathsf{f}) &\equiv \mathsf{arr}_\mathsf{b}\ \mathsf{f} && (14)\\
\mathsf{lift}_\mathsf{b}\ (\mathsf{f}_1 \ggg_\mathsf{a} \mathsf{f}_2) &\equiv (\mathsf{lift}_\mathsf{b}\ \mathsf{f}_1) \ggg_\mathsf{b} (\mathsf{lift}_\mathsf{b}\ \mathsf{f}_2) && (15)\\
\mathsf{lift}_\mathsf{b}\ (\mathsf{f}_1\ \&\&\&_\mathsf{a}\ \mathsf{f}_2) &\equiv (\mathsf{lift}_\mathsf{b}\ \mathsf{f}_1)\ \&\&\&_\mathsf{b}\ (\mathsf{lift}_\mathsf{b}\ \mathsf{f}_2) && (16)\\
\mathsf{lift}_\mathsf{b}\ (\mathsf{ifte}_\mathsf{a}\ \mathsf{f}_1\ \mathsf{f}_2\ \mathsf{f}_3) &\equiv \mathsf{ifte}_\mathsf{b}\ (\mathsf{lift}_\mathsf{b}\ \mathsf{f}_1)\ (\mathsf{lift}_\mathsf{b}\ \mathsf{f}_2)\ (\mathsf{lift}_\mathsf{b}\ \mathsf{f}_3) && (17)\\
\mathsf{lift}_\mathsf{b}\ (\mathsf{lazy}_\mathsf{a}\ \mathsf{f}) &\equiv \mathsf{lazy}_\mathsf{b}\ \lambda 0.\, \mathsf{lift}_\mathsf{b}\ (\mathsf{f}\ 0) && (18)
\end{aligned}
$$

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

Only a few of the other arrow laws play a role in our semantics and its correctness. We need associativity of $(\ggg_\mathsf{a})$:

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

a pair extraction law:

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

and distribution of pure computations over effectful:

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

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

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

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

Definition 3.3 (arrow epimorphism)**.** *An arrow homomorphism* $\mathsf{lift_b} : (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{b}} \mathsf{y})$ *that has a right inverse (equiv. surjective) is an* ***arrow epimorphism*** *from* a *to* b.

Theorem 3.4. *If* $\mathsf{lift_b} : (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{b}} \mathsf{y})$ *is an arrow epimorphism and the combinators of* a *define an arrow, then the combinators of* b *define an arrow.*

Proof. For the pair extraction law (20), rewrite in terms of $\mathsf{lift_b}$, apply homomorphism laws, and apply the pair extraction law for arrow a:

$$\begin{aligned}
&(\mathsf{arr_b}\ \mathsf{f}_1 \mathbin{\&\&\&_{\mathsf{b}}} \mathsf{f}_2) \ggg_{\mathsf{b}} \mathsf{arr_b}\ \mathsf{snd} \\
&\quad\equiv (\mathsf{lift_b}\ (\mathsf{arr_a}\ \mathsf{f}_1) \mathbin{\&\&\&_{\mathsf{b}}} (\mathsf{lift_b}\ (\mathsf{lift_b}^{-1}\ \mathsf{f}_2))) \ggg_{\mathsf{b}} \mathsf{arr_b}\ \mathsf{snd} \\
&\quad\equiv \mathsf{lift_b}\ (\mathsf{arr_a}\ \mathsf{f}_1 \mathbin{\&\&\&_{\mathsf{a}}} \mathsf{lift_b}^{-1}\ \mathsf{f}_2) \ggg_{\mathsf{b}} \mathsf{lift_b}\ (\mathsf{arr_a}\ \mathsf{snd}) \\
&\quad\equiv \mathsf{lift_b}\ ((\mathsf{arr_a}\ \mathsf{f}_1 \mathbin{\&\&\&_{\mathsf{a}}} \mathsf{lift_b}^{-1}\ \mathsf{f}_2) \ggg_{\mathsf{b}} \mathsf{arr_a}\ \mathsf{snd}) \\
&\quad\equiv \mathsf{lift_b}\ (\mathsf{lift_b}^{-1}\ \mathsf{f}_2) \equiv \mathsf{f}_2
\end{aligned}$$

The proofs for every other law are similar. □

3.2 First-Order Let-Calculus Semantics

Figure 2 defines a transformation $[\![\cdot]\!]_{\mathsf{a}}$ from a first-order let-calculus to arrow computations for any arrow a.

A program is a sequence of definition statements followed by a final expression. $[\![\cdot]\!]_{\mathsf{a}}$ compositionally 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, the interpretation acts like a stack machine. The final expression has type $\langle\rangle \rightsquigarrow_{\mathsf{a}} \mathsf{y}$, where y is the type of the program's value, and $\langle\rangle$ denotes an empty list. Let-bindings push values onto the stack. First-order functions have type $\langle \mathsf{x}, \langle\rangle\rangle \rightsquigarrow_{\mathsf{a}} \mathsf{y}$ where x is the argument type and y is the return type. Application sends a stack containing just an x.

Unless there is a reason to distinguish programs and expressions, we regard programs as if they were their final expressions. Thus, the following definition applies to both.

Definition 3.5 (well-defined expression)**.** *An expression e is* ***well-defined*** *under arrow* a *if* $[\![e]\!]_{\mathsf{a}} : \mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}$ *for some* x *and* y*, and* $[\![e]\!]_{\mathsf{a}}$ *terminates.*

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

An example is the following theorem, on which most of our semantic correctness theorems rely.

Theorem 3.6 (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 expressions 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:

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

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

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

It is not hard to check the remaining cases. □

If we assume that $\mathsf{lift_b}$ defines correct behavior for arrow b in terms of arrow a, and prove that $\mathsf{lift_b}$ is a homomorphism, then by Theorem 3.6, $[\![\cdot]\!]_{\mathsf{b}}$ is correct.

4. The Bottom and Mapping Arrows

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

$$\begin{array}{ccccc}
\mathsf{X} \rightsquigarrow_{\perp} \mathsf{Y} & \xrightarrow{\mathsf{lift_{map}}} & \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y} & \xrightarrow{\mathsf{lift_{pre}}} & \mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y} \\
\downarrow{\scriptstyle \eta_{\perp^*}} & & \downarrow{\scriptstyle \eta_{\mathsf{map}^*}} & & \downarrow{\scriptstyle \eta_{\mathsf{pre}^*}} \\
\mathsf{X} \rightsquigarrow_{\perp^*} \mathsf{Y} & \xrightarrow[\mathsf{lift_{map^*}}]{} & \mathsf{X} \underset{\mathsf{map}^*}{\rightsquigarrow} \mathsf{Y} & \xrightarrow[\mathsf{lift_{pre^*}}]{} & \mathsf{X} \underset{\mathsf{pre}^*}{\rightsquigarrow} \mathsf{Y}
\end{array} \quad (24)$$

Through Section 6, we move across the top to $\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}$.

To use Theorem 3.6 to prove correct the interpretations of expressions as preimage arrow computations, we need the preimage arrow to be homomorphic to a simpler arrow whose behavior is well-understood. One obvious candidate is the function arrow (10). However, we will need to explicitly handle nontermination as an error value, so we need a slightly more complicated arrow for which running computations may raise an error.

Figure 3 defines the ***bottom arrow***. Its computations are of type $\mathsf{x} \rightsquigarrow_{\perp} \mathsf{y} ::= \mathsf{x} \Rightarrow \mathsf{y}_{\perp}$, where the inhabitants of $\mathsf{y}_{\perp}$ are the error value $\perp$ as well as the inhabitants of y. The type $\mathsf{Bool}_{\perp}$, for example, denotes the members of $\mathsf{Bool} \uplus \{\perp\}$.

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

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

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

Figure 2: Transformation from a let-calculus with first-order definitions and De-Bruijn-indexed bindings to computations in arrow a.

$$
\begin{aligned}
&\mathsf{x} \rightsquigarrow_\perp \mathsf{y} \ ::= \ \mathsf{x} \Rightarrow \mathsf{y}_\perp \\
&\mathsf{arr}_\perp : (\mathsf{x} \Rightarrow \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_\perp \mathsf{y}) \\
&\mathsf{arr}_\perp\ \mathsf{f} \ := \ \mathsf{f} \\
&(\ggg_\perp) : (\mathsf{x} \rightsquigarrow_\perp \mathsf{y}) \Rightarrow (\mathsf{y} \rightsquigarrow_\perp \mathsf{z}) \Rightarrow (\mathsf{x} \rightsquigarrow_\perp \mathsf{z}) \\
&(\mathsf{f}_1 \ggg_\perp \mathsf{f}_2)\ \mathsf{a} \ := \ \mathsf{if}\ (\mathsf{f}_1\ \mathsf{a} = \perp)\ \perp\ (\mathsf{f}_2\ (\mathsf{f}_1\ \mathsf{a})) \\
&(\mathbin{\&\&\&_\perp}) : (\mathsf{x} \rightsquigarrow_\perp \mathsf{y}_1) \Rightarrow (\mathsf{x} \rightsquigarrow_\perp \mathsf{y}_2) \Rightarrow (\mathsf{x} \rightsquigarrow_\perp \langle \mathsf{y}_1, \mathsf{y}_2\rangle) \\
&(\mathsf{f}_1 \mathbin{\&\&\&_\perp} \mathsf{f}_2)\ \mathsf{a} \ := \ \mathsf{if}\ (\mathsf{f}_1\ \mathsf{a} = \perp \text{ or } \mathsf{f}_2\ \mathsf{a} = \perp)\ \perp\ \langle \mathsf{f}_1\ \mathsf{a}, \mathsf{f}_2\ \mathsf{a}\rangle
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{ifte}_\perp : (\mathsf{x} \rightsquigarrow_\perp \mathsf{Bool}) \Rightarrow (\mathsf{x} \rightsquigarrow_\perp \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_\perp \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_\perp \mathsf{y}) \\
&\mathsf{ifte}_\perp\ \mathsf{f}_1\ \mathsf{f}_2\ \mathsf{f}_3\ \mathsf{a} \ := \ \mathsf{case}\ \mathsf{f}_1\ \mathsf{a} \\
&\qquad\qquad\qquad\qquad \mathsf{true} \ \longrightarrow\ \mathsf{f}_2\ \mathsf{a} \\
&\qquad\qquad\qquad\qquad \mathsf{false} \ \longrightarrow\ \mathsf{f}_3\ \mathsf{a} \\
&\qquad\qquad\qquad\qquad \perp \ \longrightarrow\ \perp \\
&\mathsf{lazy}_\perp : (1 \Rightarrow (\mathsf{x} \rightsquigarrow_\perp \mathsf{y})) \Rightarrow (\mathsf{x} \rightsquigarrow_\perp \mathsf{y}) \\
&\mathsf{lazy}_\perp\ \mathsf{f}\ \mathsf{a} \ := \ \mathsf{f}\ 0\ \mathsf{a}
\end{aligned}
$$

Figure 3: Bottom arrow definitions.

Definition 4.1 (bottom arrow equivalence)**.** *Two bottom arrow computations* $\mathsf{f}_1 : \mathsf{x} \rightsquigarrow_\perp \mathsf{y}$ *and* $\mathsf{f}_2 : \mathsf{x} \rightsquigarrow_\perp \mathsf{y}$ *are equivalent, or* $\mathsf{f}_1 \equiv \mathsf{f}_2$*, when* $\mathsf{f}_1\ \mathsf{a} \equiv \mathsf{f}_2\ \mathsf{a}$ *for all* $\mathsf{a} : \mathsf{x}$*.*

Theorem 4.2. $\mathsf{arr}_\perp$*,* $(\mathbin{\&\&\&_\perp})$*,* $(\ggg_\perp)$*,* $\mathsf{ifte}_\perp$ *and* $\mathsf{lazy}_\perp$ *define an arrow.*

Proof. The bottom arrow is isomorphic (and thus epimorphic) to the Maybe monad's Kleisli arrow. □

4.1 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. define it using $\mathsf{X} \overset{\rightsquigarrow}{\underset{\mathsf{map}}{}} \mathsf{Y} ::= \mathsf{X} \rightharpoonup \mathsf{Y}$, with $\mathsf{f}_1 \ggg_{\mathsf{map}} \mathsf{f}_2 := \mathsf{f}_2 \circ_{\mathsf{map}} \mathsf{f}_1$ and $\mathsf{f}_1 \mathbin{\&\&\&_{\mathsf{map}}} \mathsf{f}_2 := \langle \mathsf{f}_1, \mathsf{f}_2\rangle_{\mathsf{map}}$. 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

$$\mathsf{X} \overset{\rightsquigarrow}{\underset{\mathsf{map}}{}} \mathsf{Y} \ ::= \ \mathsf{Set}\ \mathsf{X} \Rightarrow (\mathsf{X} \rightharpoonup \mathsf{Y}) \tag{25}$$

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.

To use Theorem 3.6 to prove that expressions interpreted using $[\![\cdot]\!]_{\mathsf{map}}$ behave correctly, 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:

$$
\begin{aligned}
&\mathsf{domain}_\perp : (\mathsf{X} \rightsquigarrow_\perp \mathsf{Y}) \Rightarrow \mathsf{Set}\ \mathsf{X} \Rightarrow \mathsf{Set}\ \mathsf{X} \\
&\mathsf{domain}_\perp\ \mathsf{f}\ \mathsf{A} \ := \ \{\mathsf{a} \in \mathsf{A} \mid \mathsf{f}\ \mathsf{a} \neq \perp\}
\end{aligned} \tag{26}
$$

$$
\begin{aligned}
&\mathsf{lift}_{\mathsf{map}} : (\mathsf{X} \rightsquigarrow_\perp \mathsf{Y}) \Rightarrow (\mathsf{X} \overset{\rightsquigarrow}{\underset{\mathsf{map}}{}} \mathsf{Y}) \\
&\mathsf{lift}_{\mathsf{map}}\ \mathsf{f}\ \mathsf{A} \ := \ \mathsf{mapping}\ \mathsf{f}\ (\mathsf{domain}_\perp\ \mathsf{f}\ \mathsf{A})
\end{aligned} \tag{27}
$$

So $\mathsf{lift}_{\mathsf{map}}\ \mathsf{f}\ \mathsf{A}$ is like $\mathsf{mapping}\ \mathsf{f}\ \mathsf{A}$, but without inputs that produce errors—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 4.3 (mapping arrow equivalence)**.** *Two mapping arrow computations* $\mathsf{g}_1 : \mathsf{X} \overset{\rightsquigarrow}{\underset{\mathsf{map}}{}} \mathsf{Y}$ *and* $\mathsf{g}_2 : \mathsf{X} \overset{\rightsquigarrow}{\underset{\mathsf{map}}{}} \mathsf{Y}$ *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 following subsections derive $(\mathbin{\&\&\&_{\mathsf{map}}})$, $(\ggg_{\mathsf{map}})$, $\mathsf{ifte}_{\mathsf{map}}$ and $\mathsf{lazy}_{\mathsf{map}}$ from bottom arrow combina-

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

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

$$(\ggg_{map}) : (X \overset{\rightsquigarrow}{\scriptstyle map} Y) \Rightarrow (Y \overset{\rightsquigarrow}{\scriptstyle map} Z) \Rightarrow (X \overset{\rightsquigarrow}{\scriptstyle map} Z)$$
$$(g_1 \ggg_{map} g_2)\ A \ := \ \mathsf{let}\ \ g_1' := g_1\ A$$
$$\qquad\qquad\qquad\qquad\qquad\qquad g_2' := g_2\ (\mathsf{range}\ g_1')$$
$$\qquad\qquad\qquad\qquad\qquad \mathsf{in}\ \ g_2' \circ_{map} g_1'$$

$$(\&\&\&_{map}) : (X \overset{\rightsquigarrow}{\scriptstyle map} Y_1) \Rightarrow (X \overset{\rightsquigarrow}{\scriptstyle map} Y_2) \Rightarrow (X \overset{\rightsquigarrow}{\scriptstyle map} \langle Y_1, Y_2\rangle)$$
$$(g_1 \ \&\&\&_{map}\ g_2)\ A \ := \ \langle g_1\ A, g_2\ A\rangle_{map}$$

$$\mathsf{ifte}_{map} : (X \overset{\rightsquigarrow}{\scriptstyle map} \mathsf{Bool}) \Rightarrow (X \overset{\rightsquigarrow}{\scriptstyle map} Y) \Rightarrow (X \overset{\rightsquigarrow}{\scriptstyle map} Y) \Rightarrow (X \overset{\rightsquigarrow}{\scriptstyle map} Y)$$
$$\mathsf{ifte}_{map}\ g_1\ g_2\ g_3\ A \ := \ \mathsf{let}\ \ g_1' := g_1\ A$$
$$\qquad\qquad\qquad\qquad\qquad\qquad g_2' := g_2\ (\mathsf{preimage}\ g_1'\ \{\mathsf{true}\})$$
$$\qquad\qquad\qquad\qquad\qquad\qquad g_3' := g_3\ (\mathsf{preimage}\ g_1'\ \{\mathsf{false}\})$$
$$\qquad\qquad\qquad\qquad\qquad \mathsf{in}\ \ g_2' \uplus_{map} g_3'$$

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

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

Figure 4: Mapping arrow definitions.

tors, in a way that ensures lift_{map} is an arrow homomorphism. Figure 4 contains the resulting definitions.

Case: Composition Starting with the left side of (15), we expand definitions, simplify f by restricting it to a domain for which $f_1\ a \neq \perp$, and then substitute f's definition:

$$\begin{aligned}
&\mathsf{lift}_{map}\ (f_1 \ggg f_2)\ A \\
&\quad\equiv \ \mathsf{let}\quad f := \lambda a.\, \mathsf{if}\ (f_1\ a = \perp)\ \perp\ (f_2\ (f_1\ a)) \\
&\qquad\qquad A' := \mathsf{domain}_{\perp}\ f\ A \\
&\qquad\ \mathsf{in}\ \ \mathsf{mapping}\ f\ A' \\
&\quad\equiv \ \mathsf{let}\quad f := \lambda a.\, f_2\ (f_1\ a) \\
&\qquad\qquad A' := \mathsf{domain}_{\perp}\ f\ (\mathsf{domain}_{\perp}\ f_1\ A) \\
&\qquad\ \mathsf{in}\ \ \mathsf{mapping}\ f\ A' \\
&\quad\equiv \ \mathsf{let}\ \ A' := \{a \in \mathsf{domain}_{\perp}\ f_1\ A \mid f_2\ (f_1\ a) \neq \perp\} \\
&\qquad\ \mathsf{in}\ \ \lambda a \in A'.\, f_2\ (f_1\ a)
\end{aligned}$$

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

$$\begin{aligned}
&\mathsf{lift}_{map}\ (f_1 \ggg f_2)\ A \\
&\quad\equiv \ \mathsf{let}\ \ g_1 := \mathsf{lift}_{map}\ f_1\ A \\
&\qquad\qquad A' := \mathsf{preimage}\ g_1\ (\mathsf{domain}_{\perp}\ f_2\ (\mathsf{range}\ g_1)) \\
&\qquad\ \mathsf{in}\ \ \lambda a \in A'.\, f_2\ (g_1\ a) \\
&\quad\equiv \ \mathsf{let}\ \ g_1 := \mathsf{lift}_{map}\ f_1\ A \\
&\qquad\qquad g_2 := \mathsf{lift}_{map}\ f_2\ (\mathsf{range}\ g_1) \\
&\qquad\qquad A' := \mathsf{preimage}\ g_1\ (\mathsf{domain}\ g_2) \\
&\qquad\ \mathsf{in}\ \ \lambda a \in A'.\, g_2\ (g_1\ a) \\
&\quad\equiv \ \mathsf{let}\ \ g_1 := \mathsf{lift}_{map}\ f_1\ A \\
&\qquad\qquad g_2 := \mathsf{lift}_{map}\ f_2\ (\mathsf{range}\ g_1) \\
&\qquad\ \mathsf{in}\ \ g_2 \circ_{map} g_1
\end{aligned}$$

Substituting g_1 for $\mathsf{lift}_{map}\ f_1$ and g_2 for $\mathsf{lift}_{map}\ f_2$ gives a definition for $(\ggg_{map})$ (Figure 4) for which (15) holds.

Case: 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:

$$\begin{aligned}
&\mathsf{lift}_{map}\ (f_1\ \&\&\&_{\perp}\ f_2)\ A \\
&\quad\equiv \ \mathsf{let}\quad f := \lambda a.\, \mathsf{if}\ (f_1\ a = \perp \text{ or } f_2\ a = \perp)\ \perp\ \langle f_1\ a, f_2\ a\rangle \\
&\qquad\qquad A' := \mathsf{domain}_{\perp}\ f\ A \\
&\qquad\ \mathsf{in}\ \ \mathsf{mapping}\ f\ A' \\
&\quad\equiv \ \mathsf{let}\ \ A' := \mathsf{domain}_{\perp}\ f_1\ A \cap \mathsf{domain}_{\perp}\ f_2\ A \\
&\qquad\ \mathsf{in}\ \ \lambda a \in A'.\, \langle f_1\ a, f_2\ a\rangle
\end{aligned}$$

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

$$\begin{aligned}
&\mathsf{lift}_{map}\ (f_1\ \&\&\&_{\perp}\ f_2)\ A \\
&\quad\equiv \ \mathsf{let}\ \ g_1 := \mathsf{lift}_{map}\ f_1\ A \\
&\qquad\qquad g_2 := \mathsf{lift}_{map}\ f_2\ A \\
&\qquad\qquad A' := \mathsf{domain}\ g_1 \cap \mathsf{domain}\ g_2 \\
&\qquad\ \mathsf{in}\ \ \lambda a \in A'.\, \langle g_1\ a, g_2\ a\rangle \\
&\quad\equiv \ \langle \mathsf{lift}_{map}\ f_1\ A, \mathsf{lift}_{map}\ f_2\ A\rangle_{map} \qquad (28)
\end{aligned}$$

Substituting g_1 for $\mathsf{lift}_{map}\ f_1$ and g_2 for $\mathsf{lift}_{map}\ f_2$ gives a definition for $(\&\&\&_{map})$ (Figure 4) for which (16) holds.

Case: Conditional Starting with the left side of (17), we expand definitions, and simplify f by restricting it to a domain for which $f_1\ a \neq \perp$:

$$\begin{aligned}
&\mathsf{lift}_{map}\ (\mathsf{ifte}_{\perp}\ f_1\ f_2\ f_3)\ A \\
&\quad\equiv \ \mathsf{let}\ \ f := \lambda a.\, \mathsf{case}\ \ f_1\ a \\
&\qquad\qquad\qquad\qquad\quad \mathsf{true} \ \longrightarrow\ f_2\ a \\
&\qquad\qquad\qquad\qquad\quad \mathsf{false} \ \longrightarrow\ f_3\ a \\
&\qquad\qquad\qquad\qquad\quad \perp \ \longrightarrow\ \perp \\
&\qquad\ \mathsf{in}\ \ \mathsf{mapping}\ f\ (\mathsf{domain}_{\perp}\ f\ A) \\
&\quad\equiv \ \mathsf{let}\ \ g_1 := \mathsf{mapping}\ f\ A \qquad (29) \\
&\qquad\qquad A_2 := \mathsf{preimage}\ g_1\ \{\mathsf{true}\} \\
&\qquad\qquad A_3 := \mathsf{preimage}\ g_1\ \{\mathsf{false}\} \\
&\qquad\qquad\ f := \lambda a.\, \mathsf{if}\ (f_1\ a)\ (f_2\ a)\ (f_3\ a) \\
&\qquad\ \mathsf{in}\ \ \mathsf{mapping}\ f\ (\mathsf{domain}_{\perp}\ f\ (A_2 \uplus A_3))
\end{aligned}$$

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

$$\begin{aligned}
&\mathsf{lift}_{map}\ (\mathsf{ifte}_{\perp}\ f_1\ f_2\ f_3)\ A \qquad (30) \\
&\quad\equiv \ \mathsf{let}\ \ g_1 := \mathsf{lift}_{map}\ f_1\ A \\
&\qquad\qquad g_2 := \mathsf{lift}_{map}\ f_2\ (\mathsf{preimage}\ g_1\ \{\mathsf{true}\}) \\
&\qquad\qquad g_3 := \mathsf{lift}_{map}\ f_3\ (\mathsf{preimage}\ g_1\ \{\mathsf{false}\}) \\
&\qquad\qquad A' := \mathsf{domain}\ g_2 \uplus \mathsf{domain}\ g_3 \\
&\qquad\ \mathsf{in}\ \ \lambda a \in A'.\, \mathsf{if}\ (a \in \mathsf{domain}\ g_2)\ (g_2\ a)\ (g_3\ a) \\
&\quad\equiv \ \mathsf{let}\ \ g_1 := \mathsf{lift}_{map}\ f_1\ A \\
&\qquad\qquad g_2 := \mathsf{lift}_{map}\ f_2\ (\mathsf{preimage}\ g_1\ \{\mathsf{true}\}) \\
&\qquad\qquad g_3 := \mathsf{lift}_{map}\ f_3\ (\mathsf{preimage}\ g_1\ \{\mathsf{false}\}) \\
&\qquad\ \mathsf{in}\ \ g_2 \uplus_{map} g_3
\end{aligned}$$

Substituting g_1 for $\mathsf{lift}_{map}\ f_1$, g_2 for $\mathsf{lift}_{map}\ f_2$, and g_3 for $\mathsf{lift}_{map}\ f_3$ gives a definition for ifte_{map} (Figure 4) for which (17) holds.

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

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

λ_{ZFC} does not have an η rule (i.e. $\lambda x.\, e\ x \not\equiv e$ because e may loop), but 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 loops if and only if $\mathsf{mapping}\ (\mathsf{f}\ 0)\ \mathsf{A}'$ loops. 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

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

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

4.2 Correctness

Theorem 4.4 (mapping arrow correctness)**.** $\mathsf{lift}_{\mathsf{map}}$ *is an arrow homomorphism.*

Proof. By construction. □

Corollary 4.5 (semantic correctness)**.** *For all expressions* 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:

$$\begin{aligned}&\mathsf{nonmonotone} : \mathsf{Bool} \mathrel{\tilde{\to}}_{\mathsf{map}} \mathsf{Bool}\\ &\mathsf{nonmonotone}\ \mathsf{A}\ :=\ \mathsf{if}\ (\mathsf{A} = \varnothing)\ (\mathsf{mapping}\ \mathsf{id}\ \mathsf{Bool})\ \varnothing\end{aligned} \tag{31}$$

It would be nice if we could be sure that every $\mathsf{X} \mathrel{\tilde{\to}}_{\mathsf{map}} \mathsf{Y}$ 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 4.6 (mapping arrow law)**.** *Let* $\mathsf{g} : \mathsf{X} \mathrel{\tilde{\to}}_{\mathsf{map}} \mathsf{Y}$*. 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****.*

We assume from here on that the mapping arrow law holds for all $\mathsf{g} : \mathsf{X} \mathrel{\tilde{\to}}_{\mathsf{map}} \mathsf{Y}$. By homomorphism of $\mathsf{lift}_{\mathsf{map}}$, mapping arrow combinators return computations that obey this law.

Theorem 4.7. $\mathsf{lift}_{\mathsf{map}}$ *is an arrow epimorphism.*

Proof. Follows from Theorem 4.4 and Definition 4.6. □

Corollary 4.8. $\mathsf{arr}_{\mathsf{map}}$, $(\&\&\&_{\mathsf{map}})$, $(\ggg_{\mathsf{map}})$, $\mathsf{ifte}_{\mathsf{map}}$ *and* $\mathsf{lazy}_{\mathsf{map}}$ *define an arrow.*

5. 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 will confine computation on points to instances of

$$\mathsf{X} \mathrel{\overrightarrow{\mathsf{pre}}} \mathsf{Y}\ ::=\ \langle \mathsf{Set}\ \mathsf{Y}, \mathsf{Set}\ \mathsf{Y} \Rightarrow \mathsf{Set}\ \mathsf{X} \rangle \tag{32}$$

Like a mapping, an $\mathsf{X} \mathrel{\overrightarrow{\mathsf{pre}}} \mathsf{Y}$ has an observable domain—but computing the table of 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:

$$\begin{aligned}&\mathsf{pre} : (\mathsf{X} \rightharpoonup \mathsf{Y}) \Rightarrow (\mathsf{X} \mathrel{\overrightarrow{\mathsf{pre}}} \mathsf{Y})\\ &\mathsf{pre}\ \mathsf{g}\ :=\ \langle \mathsf{range}\ \mathsf{g}, \lambda\mathsf{B}.\,\mathsf{preimage}\ \mathsf{g}\ \mathsf{B} \rangle\end{aligned} \tag{33}$$

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

$$\begin{aligned}&\mathsf{ap}_{\mathsf{pre}} : (\mathsf{X} \mathrel{\overrightarrow{\mathsf{pre}}} \mathsf{Y}) \Rightarrow \mathsf{Set}\ \mathsf{Y} \Rightarrow \mathsf{Set}\ \mathsf{X}\\ &\mathsf{ap}_{\mathsf{pre}}\ \langle \mathsf{Y}', \mathsf{p} \rangle\ \mathsf{B}\ :=\ \mathsf{p}\ (\mathsf{B} \cap \mathsf{Y}')\end{aligned} \tag{34}$$

The necessary property here is that using $\mathsf{ap}_{\mathsf{pre}}$ to compute preimages is the same as computing them from a mapping using preimage.

Imported Lemma 5.1. *Let* $\mathsf{g} \in \mathsf{X} \rightharpoonup \mathsf{Y}$*. For all* $\mathsf{B} \subseteq \mathsf{Y}$ *and* Y' *such that* $\mathsf{range}\ \mathsf{g} \subseteq \mathsf{Y}' \subseteq \mathsf{Y}$*,* $\mathsf{preimage}\ \mathsf{g}\ (\mathsf{B} \cap \mathsf{Y}') = \mathsf{preimage}\ \mathsf{g}\ \mathsf{B}$.

Theorem 5.2 ($\mathsf{ap}_{\mathsf{pre}}$ computes preimages)**.** *Let* $\mathsf{g} \in \mathsf{X} \rightharpoonup \mathsf{Y}$*. For all* $\mathsf{B} \subseteq \mathsf{Y}$*,* $\mathsf{ap}_{\mathsf{pre}}\ (\mathsf{pre}\ \mathsf{g})\ \mathsf{B} = \mathsf{preimage}\ \mathsf{g}\ \mathsf{B}$.

Proof. Expand definitions and apply Lemma 5.1 with $\mathsf{Y}' = \mathsf{range}\ \mathsf{g}$. □

Figure 5 defines more operations on preimage mappings, including pairing, composition, and disjoint union operations corresponding to the mapping operations in Figure 1. The next three theorems establish that pre is a homomorphism (though not an arrow homomorphism): it distributes over mapping operations to yield preimage mapping operations. We will use these facts to derive the preimage arrow from the mapping arrow.

First, we need preimage mappings to be equivalent when they compute the same preimages.

Definition 5.3 (preimage mapping equivalence)**.** *Two preimage mappings* $\mathsf{h}_1 : \mathsf{X} \mathrel{\overrightarrow{\mathsf{pre}}} \mathsf{Y}$ *and* $\mathsf{h}_2 : \mathsf{X} \mathrel{\overrightarrow{\mathsf{pre}}} \mathsf{Y}$ *are equivalent, or* $\mathsf{h}_1 \equiv \mathsf{h}_2$*, when* $\mathsf{ap}_{\mathsf{pre}}\ \mathsf{h}_1\ \mathsf{B} \equiv \mathsf{ap}_{\mathsf{pre}}\ \mathsf{h}_2\ \mathsf{B}$ *for all* $\mathsf{B} \subseteq \mathsf{Y}$.

The following subsections prove distributive laws for preimage mapping pairing, composition, and disjoint union.

XXX: moar text in following subsections

Pairing

Imported Lemma 5.4 (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 5.5 (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* $\mathsf{pre}\ \langle \mathsf{g}_1, \mathsf{g}_2 \rangle_{\mathsf{map}} \equiv \langle \mathsf{pre}\ \mathsf{g}_1, \mathsf{pre}\ \mathsf{g}_2 \rangle_{\mathsf{pre}}$.

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

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

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

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

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

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

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

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

Figure 5: Lazy preimage mappings and operations.

Proof. Let $\langle \mathsf{Y}_1', \mathsf{p}_1 \rangle := \mathsf{pre\ g}_1$ and $\langle \mathsf{Y}_2', \mathsf{p}_2 \rangle := \mathsf{pre\ g}_2$. Starting from the right side, for all $\mathsf{B} \in \mathsf{Y}_1 \times \mathsf{Y}_2$,

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

□

Composition

Imported Lemma 5.6 (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* $\mathsf{C} \subseteq \mathsf{Z}$*,* $\mathsf{preimage}\ (\mathsf{g}_2 \circ_{\mathsf{map}} \mathsf{g}_1)\ \mathsf{C} = \mathsf{preimage\ g}_1\ (\mathsf{preimage\ g}_2\ \mathsf{C})$.

Theorem 5.7 (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* $\mathsf{pre}\ (\mathsf{g}_2 \circ_{\mathsf{map}} \mathsf{g}_1) \equiv (\mathsf{pre\ g}_2) \circ_{\mathsf{pre}} (\mathsf{pre\ g}_1)$.

Proof. Let $\langle \mathsf{Z}', \mathsf{p}_2 \rangle := \mathsf{pre\ g}_2$. Starting from the right side, for all $\mathsf{C} \subseteq \mathsf{Z}$,

$$\begin{array}{ll}& \mathsf{ap}_{\mathsf{pre}}\ ((\mathsf{pre\ g}_2) \circ_{\mathsf{pre}} (\mathsf{pre\ g}_1))\ \mathsf{C} \\ \equiv & \mathsf{let}\ \ \mathsf{h} := \lambda \mathsf{C}.\, \mathsf{ap}_{\mathsf{pre}}\ (\mathsf{pre\ g}_1)\ (\mathsf{p}_2\ \mathsf{C}) \\ & \quad \mathsf{in}\ \ \mathsf{h}\ (\mathsf{C} \cap \mathsf{Z}') \\ \equiv & \mathsf{ap}_{\mathsf{pre}}\ (\mathsf{pre\ g}_1)\ (\mathsf{p}_2\ (\mathsf{C} \cap \mathsf{Z}')) \\ \equiv & \mathsf{ap}_{\mathsf{pre}}\ (\mathsf{pre\ g}_1)\ (\mathsf{ap}_{\mathsf{pre}}\ (\mathsf{pre\ g}_2)\ \mathsf{C}) \\ \equiv & \mathsf{preimage\ g}_1\ (\mathsf{preimage\ g}_2\ \mathsf{C}) \\ \equiv & \mathsf{preimage}\ (\mathsf{g}_2 \circ_{\mathsf{map}} \mathsf{g}_1)\ \mathsf{C} \\ \equiv & \mathsf{ap}_{\mathsf{pre}}\ (\mathsf{pre}\ (\mathsf{g}_2 \circ_{\mathsf{map}} \mathsf{g}_1))\ \mathsf{C}\end{array}$$

□

Disjoint Union

Imported Lemma 5.8 (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* $\mathsf{B} \subseteq \mathsf{Y}$*,* $\mathsf{preimage}\ (\mathsf{g}_1 \uplus_{\mathsf{map}} \mathsf{g}_2)\ \mathsf{B} = (\mathsf{preimage\ g}_1\ \mathsf{B}) \uplus (\mathsf{preimage\ g}_2\ \mathsf{B})$.

Theorem 5.9 (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* $\mathsf{pre}\ (\mathsf{g}_1 \uplus_{\mathsf{map}} \mathsf{g}_2) \equiv (\mathsf{pre\ g}_1) \uplus_{\mathsf{pre}} (\mathsf{pre\ g}_2)$.

Proof. Let $\mathsf{Y}_1' := \mathsf{range\ g}_1$ and $\mathsf{Y}_2' := \mathsf{range\ g}_2$. Starting from the right side, for all $\mathsf{B} \subseteq \mathsf{Y}$,

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

□

6. Deriving the Preimage Arrow

We are ready to define an arrow that runs expressions backward on sets of outputs. Its computations should produce preimage mappings or be preimage mappings themselves.

As with the mapping arrow and mappings, we cannot have $\mathsf{X} \underset{\mathsf{pre}}{\rightrightarrows} \mathsf{Y} ::= \mathsf{X} \underset{\mathsf{pre}}{\rightrightarrows} \mathsf{Y}$: 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 $\mathsf{Set\ X}$; therefore the ***preimage arrow*** type constructor is

$$\mathsf{X} \underset{\mathsf{pre}}{\rightrightarrows} \mathsf{Y} \ ::= \ \mathsf{Set\ X} \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightrightarrows} \mathsf{Y}) \tag{35}$$

or $\mathsf{Set\ X} \Rightarrow \langle \mathsf{Set\ Y}, \mathsf{Set\ Y} \Rightarrow \mathsf{Set\ X} \rangle$. 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 3.6, we need to define correctness using a lift from the mapping arrow to the preimage arrow:

$$\begin{aligned} &\mathsf{lift}_{\mathsf{pre}} : (\mathsf{X} \rightsquigarrow_{\mathsf{map}} \mathsf{Y}) \Rightarrow (\mathsf{X} \rightsquigarrow_{\mathsf{pre}} \mathsf{Y}) \\ &\mathsf{lift}_{\mathsf{pre}}\ \mathsf{g}\ \mathsf{A}\ :=\ \mathsf{pre}\ (\mathsf{g}\ \mathsf{A}) \end{aligned} \tag{36}$$

By Theorem 5.2, for all $\mathsf{g} : \mathsf{X} \rightsquigarrow_{\mathsf{map}} \mathsf{Y}$, $\mathsf{A} \subseteq \mathsf{X}$ and $\mathsf{B} \subseteq \mathsf{Y}$,

$$\mathsf{ap}_{\mathsf{pre}}\ (\mathsf{lift}_{\mathsf{pre}}\ \mathsf{g}\ \mathsf{A})\ \mathsf{B}\ \equiv\ \mathsf{preimage}\ (\mathsf{g}\ \mathsf{A})\ \mathsf{B} \tag{37}$$

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

Again, we need a coarser notion of equivalence.

Definition 6.1 (Preimage arrow equivalence)**.** *Two preimage arrow computations* $\mathsf{h}_1 : \mathsf{X} \rightsquigarrow_{\mathsf{pre}} \mathsf{Y}$ *and* $\mathsf{h}_2 : \mathsf{X} \rightsquigarrow_{\mathsf{pre}} \mathsf{Y}$ *are equivalent, or* $\mathsf{h}_1 \equiv \mathsf{h}_2$*, when* $\mathsf{h}_1\ \mathsf{A} \equiv \mathsf{h}_2\ \mathsf{A}$ *for all* $\mathsf{A} \subseteq \mathsf{X}$*.*

As with $\mathsf{arr}_{\mathsf{map}}$, defining $\mathsf{arr}_{\mathsf{pre}}$ as a composition meets (14). The following subsections derive $(\&\&\&_{\mathsf{pre}})$, $(\ggg_{\mathsf{pre}})$, $\mathsf{ifte}_{\mathsf{pre}}$ and $\mathsf{lazy}_{\mathsf{pre}}$ from mapping arrow combinators, in a way that ensures $\mathsf{lift}_{\mathsf{pre}}$ is an arrow homomorphism from the mapping arrow to the preimage arrow. Figure 6 contains the resulting definitions.

Case: Pairing Starting with the left side of (16), we expand definitions, apply Theorem 5.5, and rewrite in terms of $\mathsf{lift}_{\mathsf{pre}}$:

$$\begin{aligned} &\mathsf{ap}_{\mathsf{pre}}\ (\mathsf{lift}_{\mathsf{pre}}\ (\mathsf{g}_1\ \&\&\&_{\mathsf{map}}\ \mathsf{g}_2)\ \mathsf{A})\ \mathsf{B} \\ &\quad\equiv\ \mathsf{ap}_{\mathsf{pre}}\ (\mathsf{pre}\ \langle \mathsf{g}_1\ \mathsf{A}, \mathsf{g}_2\ \mathsf{A}\rangle_{\mathsf{map}})\ \mathsf{B} \\ &\quad\equiv\ \mathsf{ap}_{\mathsf{pre}}\ \langle \mathsf{pre}\ (\mathsf{g}_1\ \mathsf{A}), \mathsf{pre}\ (\mathsf{g}_2\ \mathsf{A})\rangle_{\mathsf{pre}}\ \mathsf{B} \\ &\quad\equiv\ \mathsf{ap}_{\mathsf{pre}}\ \langle \mathsf{lift}_{\mathsf{pre}}\ \mathsf{g}_1\ \mathsf{A}, \mathsf{lift}_{\mathsf{pre}}\ \mathsf{g}_2\ \mathsf{A}\rangle_{\mathsf{pre}}\ \mathsf{B} \end{aligned}$$

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 $(\&\&\&_{\mathsf{pre}})$ (Figure 6) for which (16) holds.

Case: Composition Starting with the left side of (15), we expand definitions, apply Theorem 5.7 and rewrite in terms of $\mathsf{lift}_{\mathsf{pre}}$:

$$\begin{aligned} &\mathsf{ap}_{\mathsf{pre}}\ (\mathsf{lift}_{\mathsf{pre}}\ (\mathsf{g}_1 \ggg_{\mathsf{map}} \mathsf{g}_2)\ \mathsf{A})\ \mathsf{C} \\ &\quad\equiv\ \mathsf{let}\ \ \mathsf{g}_1' := \mathsf{g}_1\ \mathsf{A} \\ &\qquad\quad\ \ \mathsf{g}_2' := \mathsf{g}_2\ (\mathsf{range}\ \mathsf{g}_1') \\ &\qquad\ \mathsf{in}\ \ \mathsf{ap}_{\mathsf{pre}}\ (\mathsf{pre}\ (\mathsf{g}_2' \circ_{\mathsf{map}} \mathsf{g}_1'))\ \mathsf{C} \\ &\quad\equiv\ \mathsf{let}\ \ \mathsf{g}_1' := \mathsf{g}_1\ \mathsf{A} \\ &\qquad\quad\ \ \mathsf{g}_2' := \mathsf{g}_2\ (\mathsf{range}\ \mathsf{g}_1') \\ &\qquad\ \mathsf{in}\ \ \mathsf{ap}_{\mathsf{pre}}\ ((\mathsf{pre}\ \mathsf{g}_1') \circ_{\mathsf{pre}} (\mathsf{pre}\ \mathsf{g}_2'))\ \mathsf{C} \\ &\quad\equiv\ \mathsf{let}\ \ \mathsf{h}_1 := \mathsf{lift}_{\mathsf{pre}}\ \mathsf{g}_1\ \mathsf{A} \\ &\qquad\quad\ \ \mathsf{h}_2 := \mathsf{lift}_{\mathsf{pre}}\ \mathsf{g}_2\ (\mathsf{range}_{\mathsf{pre}}\ \mathsf{h}_1) \\ &\qquad\ \mathsf{in}\ \ \mathsf{ap}_{\mathsf{pre}}\ (\mathsf{h}_2 \circ_{\mathsf{pre}} \mathsf{h}_1)\ \mathsf{C} \end{aligned} \tag{38}$$

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 $(\ggg_{\mathsf{pre}})$ (Figure 6) for which (15) holds.

Case: Conditional Starting with the left side of (17), we expand terms, apply Theorem 5.9, rewrite in terms of $\mathsf{lift}_{\mathsf{pre}}$, and apply Theorem 5.2 in the definitions of h_2 and h_3:

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

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 $\mathsf{ifte}_{\mathsf{pre}}$ (Figure 6) for which (17) holds.

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

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

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 $\mathsf{lazy}_{\mathsf{pre}}$ (Figure 6) for which (18) holds.

6.1 Correctness

Theorem 6.2 (preimage arrow correctness)**.** $\mathsf{lift}_{\mathsf{pre}}$ *is an arrow homomorphism.*

Proof. By construction. □

Corollary 6.3 (semantic correctness)**.** *For all expressions* 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 $\mathsf{h} : \mathsf{X} \rightsquigarrow_{\mathsf{pre}} \mathsf{Y}$ acts as if it always computes preimages under restricted mappings. The following equivalent property is easier to state, and makes proving the arrow laws simple.

Definition 6.4 (preimage arrow law)**.** *Let* $\mathsf{h} : \mathsf{X} \rightsquigarrow_{\mathsf{pre}} \mathsf{Y}$*. If there exists a* $\mathsf{g} : \mathsf{X} \rightsquigarrow_{\mathsf{map}} \mathsf{Y}$ *such that* $\mathsf{h} \equiv \mathsf{lift}_{\mathsf{pre}}\ \mathsf{g}$*, then* h *obeys the* ***preimage arrow law****.*

We assume from here on that the preimage arrow law holds for all $\mathsf{h} : \mathsf{X} \rightsquigarrow_{\mathsf{pre}} \mathsf{Y}$. By homomorphism of $\mathsf{lift}_{\mathsf{pre}}$, preimage arrow combinators return computations that obey this law.

Theorem 6.5. $\mathsf{lift}_{\mathsf{pre}}$ *is an arrow epimorphism.*

Proof. Follows from Theorem 6.2 and Definition 6.4. □

Corollary 6.6. $\mathsf{arr}_{\mathsf{pre}}$*,* $(\&\&\&_{\mathsf{pre}})$*,* $(\ggg_{\mathsf{pre}})$*,* $\mathsf{ifte}_{\mathsf{pre}}$ *and* $\mathsf{lazy}_{\mathsf{pre}}$ *define an arrow.*

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

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

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

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

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

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

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

Figure 6: Preimage arrow definitions.

7. Preimages Under Partial Functions

We have defined everything on the top of our roadmap:

$$\begin{array}{ccccc} \mathsf{X} \rightsquigarrow_{\perp} \mathsf{Y} & \xrightarrow{\mathsf{lift}_{\mathsf{map}}} & \mathsf{X} \overset{\rightsquigarrow}{\scriptstyle\mathsf{map}} \mathsf{Y} & \xrightarrow{\mathsf{lift}_{\mathsf{pre}}} & \mathsf{X} \overset{\rightsquigarrow}{\scriptstyle\mathsf{pre}} \mathsf{Y} \\ {\scriptstyle\eta_{\perp^*}}\downarrow & & \downarrow{\scriptstyle\eta_{\mathsf{map}^*}} & & \downarrow{\scriptstyle\eta_{\mathsf{pre}^*}} \\ \mathsf{X} \rightsquigarrow_{\perp^*} \mathsf{Y} & \xrightarrow[\mathsf{lift}_{\mathsf{map}^*}]{} & \mathsf{X} \overset{\rightsquigarrow}{\scriptstyle\mathsf{map}^*} \mathsf{Y} & \xrightarrow[\mathsf{lift}_{\mathsf{pre}^*}]{} & \mathsf{X} \overset{\rightsquigarrow}{\scriptstyle\mathsf{pre}^*} \mathsf{Y} \end{array} \tag{39}$$

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 (39) is an arrow homomorphism.

7.1 Motivation

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

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

$$\mathsf{geometric}\ \mathsf{p}\ :=\ \mathsf{if}\ (\mathsf{random} < \mathsf{p})\ 0\ (1 + \mathsf{geometric}\ \mathsf{p}) \tag{40}$$

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

Suppose we interpret (40) as $\mathsf{h} : \mathsf{R} \overset{\rightsquigarrow}{\scriptstyle\mathsf{pre}} \mathbb{N}$, a preimage arrow computation from random sources in R to natural numbers, and that we have a probability measure $\mathsf{P} \in \mathcal{P}\ \mathsf{R} \rightharpoonup [0,1]$. We could compute the probability of any output set $\mathsf{N} \subseteq \mathbb{N}$ using $\mathsf{P}\ (\mathsf{h}\ \mathsf{R}'\ \mathsf{N})$, where $\mathsf{R}' \subseteq \mathsf{R}$ and $\mathsf{P}\ \mathsf{R}' = 1$. We have three hurdles to overcome:

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

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

7.2 Threading and Indexing

We clearly need a new arrow that threads a random source through its computations. To ensure it contains enough random numbers, the source should be infinite.

In a pure λ-calculus, random sources are typically infinite streams, threaded monadically: each computation receives and produces a random source. A new combinator is defined that removes the head of the random source and passes the tail along. This is likely preferred because pseudorandom number generators are almost universally monadic.

A little-used alternative is for the random source to be a tree, threaded applicatively: each computation receives, but does not produce, a random source. Multi-argument combinators split the tree and pass subtrees to subcomputations.

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

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

Definition 7.1 (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

$$\begin{aligned} \mathsf{left}\ (\mathsf{p}/\,\mathsf{q})\ &:=\ (\mathsf{p} - \tfrac{1}{2})/\,\mathsf{q} \\ \mathsf{right}\ (\mathsf{p}/\,\mathsf{q})\ &:=\ (\mathsf{p} + \tfrac{1}{2})/\,\mathsf{q} \end{aligned} \tag{41}$$

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

In any case, the index set J is always 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).

7.3 Applicative, Associative Store Transformer

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

$$\mathsf{AStore}\ \mathsf{s}\ (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y})\ ::=\ \mathsf{J} \Rightarrow (\langle \mathsf{s}, \mathsf{x} \rangle \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \tag{42}$$

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

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

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

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

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

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

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

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

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

$$\begin{aligned} &\eta_{\mathsf{a}^*} : (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \Rightarrow \mathsf{AStore\ s\ (x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \\ &\eta_{\mathsf{a}^*}\ \mathsf{f}\ \mathsf{j} \;:=\; \mathsf{arr}_{\mathsf{a}}\ \mathsf{snd} \ggg_{\mathsf{a}} \mathsf{f} \end{aligned} \tag{43}$$

Because f never accesses the store, j is ignored.

Figure 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.

7.4 Partial, Probabilistic Programs

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

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

Definition 7.2 (random source)**.** *Let* $\mathsf{R} := \mathsf{J} \to [0,1]$*. A* ***random source*** *is any infinite binary tree* $\mathsf{r} \in \mathsf{R}$*.*

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

$$\begin{aligned} &\mathsf{random}_{\mathsf{a}^*} : \mathsf{x} \rightsquigarrow_{\mathsf{a}^*} [0,1] \\ &\mathsf{random}_{\mathsf{a}^*}\ \mathsf{j} \;:=\; \mathsf{arr}_{\mathsf{a}}\ (\mathsf{fst} \ggg \pi\ \mathsf{j}) \\ &[\![\mathsf{random}]\!]_{\mathsf{a}^*} \;::\equiv\; \mathsf{random}_{\mathsf{a}^*} \end{aligned} \tag{44}$$

Partial Programs One utimately implementable way to avoid nontermination is to use the store to dictate which branch of each conditional, if any, is allowed to be taken.

Definition 7.3 (branch trace)**.** *A* ***branch trace*** *is any* $\mathsf{t} \in \mathsf{J} \to \mathsf{Bool}_\perp$ *such that* $\mathsf{t\ j} = \mathsf{true}$ *or* $\mathsf{t\ j} = \mathsf{false}$ *for no more than finitely many* $\mathsf{j} \in \mathsf{J}$*.*

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

$$\begin{aligned} &\mathsf{branch}_{\mathsf{a}^*} : \mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{Bool} \\ &\mathsf{branch}_{\mathsf{a}^*}\ \mathsf{j} \;:=\; \mathsf{arr}_{\mathsf{a}}\ (\mathsf{fst} \ggg \pi\ \mathsf{j}) \end{aligned} \tag{45}$$

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

$$\begin{aligned} &\mathsf{agrees} : \langle \mathsf{Bool}, \mathsf{Bool} \rangle \Rightarrow \mathsf{Bool}_\perp \\ &\mathsf{agrees}\ \langle \mathsf{b}_1, \mathsf{b}_2 \rangle \;:=\; \mathsf{if}\ (\mathsf{b}_1 = \mathsf{b}_2)\ \mathsf{b}_1\ \perp \end{aligned} \tag{46}$$

$$\begin{aligned} &\mathsf{ifte}^{\Downarrow}_{\mathsf{a}^*} : (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{Bool}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y}) \\ &\mathsf{ifte}^{\Downarrow}_{\mathsf{a}^*}\ \mathsf{k}_1\ \mathsf{k}_2\ \mathsf{k}_3\ \mathsf{j} \;:= \\ &\quad \mathsf{ifte}_{\mathsf{a}}\ ((\mathsf{k}_1\ (\mathsf{left\ j}) \mathbin{\&\&\&_{\mathsf{a}}} \mathsf{branch}_{\mathsf{a}^*}\ \mathsf{j}) \ggg_{\mathsf{a}} \mathsf{arr}_{\mathsf{a}}\ \mathsf{agrees}) \\ &\qquad (\mathsf{k}_2\ (\mathsf{left\ (right\ j)})) \\ &\qquad (\mathsf{k}_3\ (\mathsf{right\ (right\ j)})) \end{aligned} \tag{47}$$

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

We assume every expression is well-defined (Definition 3.5), so every expression must have its recurrences guarded by if. Thus, to ensure running their interpretations always terminates, we should only need to replace $\mathsf{ifte}_{\mathsf{a}^*}$ with $\mathsf{ifte}^{\Downarrow}_{\mathsf{a}^*}$. We define a new semantic function $[\![\cdot]\!]^{\Downarrow}_{\mathsf{a}^*}$ by

$$\begin{aligned} [\![\mathsf{if}\ e_c\ e_t\ e_f]\!]^{\Downarrow}_{\mathsf{a}^*} \;::\equiv\; &\mathsf{ifte}^{\Downarrow}_{\mathsf{a}^*}\ [\![e_c]\!]^{\Downarrow}_{\mathsf{a}^*} \\ &(\mathsf{lazy}_{\mathsf{a}}\ \lambda 0.\, [\![e_t]\!]^{\Downarrow}_{\mathsf{a}^*}) \\ &(\mathsf{lazy}_{\mathsf{a}}\ \lambda 0.\, [\![e_f]\!]^{\Downarrow}_{\mathsf{a}^*}) \end{aligned} \tag{48}$$

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

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

Partial, Probabilistic Programs Let $\mathsf{S} ::= \mathsf{R} \times \mathsf{T}$ and $\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y} ::= \mathsf{AStore\ S\ (x} \rightsquigarrow_{\mathsf{a}} \mathsf{y})$, and update the $\mathsf{random}_{\mathsf{a}^*}$ and $\mathsf{branch}_{\mathsf{a}^*}$ combinators to reflect that the store is now a pair:

$$\mathsf{random}_{\mathsf{a}^*}\ \mathsf{j} \;:=\; \mathsf{arr}_{\mathsf{a}}\ (\mathsf{fst} \ggg \mathsf{fst} \ggg \pi\ \mathsf{j}) \tag{49}$$

$$\mathsf{branch}_{\mathsf{a}^*}\ \mathsf{j} \;:=\; \mathsf{arr}_{\mathsf{a}}\ (\mathsf{fst} \ggg \mathsf{snd} \ggg \pi\ \mathsf{j}) \tag{50}$$

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

Definition 7.4 (terminating, probabilistic arrows)**.** *Let*

$$\begin{aligned} \mathsf{x} \rightsquigarrow_{\perp^*} \mathsf{y} \;&::=\; \mathsf{AStore\ (R \times T)\ (x} \rightsquigarrow_{\perp} \mathsf{y}) \\ \mathsf{X} \underset{\mathsf{map}^*}{\rightsquigarrow} \mathsf{Y} \;&::=\; \mathsf{AStore\ (R \times T)\ (X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}) \\ \mathsf{X} \underset{\mathsf{pre}^*}{\rightsquigarrow} \mathsf{Y} \;&::=\; \mathsf{AStore\ (R \times T)\ (X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}) \end{aligned} \tag{51}$$

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

7.5 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 7.5 (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 general properties. From here on, we assume every AStore arrow's base type's combinators obey the arrow laws listed in Section 3.1.

Theorem 7.6 (AStore arrow correctness)**.** η_{a^*} *is an arrow 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 7.7 (pure semantic correctness)**.** *For all pure expressions* e*,* $[\![e]\!]_{\mathsf{a}^*} \equiv \eta_{\mathsf{a}^*}\ [\![e]\!]_{\mathsf{a}}$ *and* $[\![e]\!]^{\Downarrow}_{\mathsf{a}^*} \equiv \eta_{\mathsf{a}^*}\ [\![e]\!]^{\Downarrow}_{\mathsf{a}}$*.*

Effectful Expressions To prove all interpretations of effectful expressions correct, we need a lift between AStore arrows. Let $\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y} ::= \mathsf{AStore}\ \mathsf{s}\ (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y})$ and $\mathsf{x} \rightsquigarrow_{\mathsf{b}^*} \mathsf{y} ::= \mathsf{AStore}\ \mathsf{s}\ (\mathsf{x} \rightsquigarrow_{\mathsf{b}^*} \mathsf{y})$. Define

$$\begin{aligned} &\mathsf{lift}_{\mathsf{b}^*} : (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{b}^*} \mathsf{y}) \\ &\mathsf{lift}_{\mathsf{b}^*}\ \mathsf{f}\ \mathsf{j} \ :=\ \mathsf{lift}_{\mathsf{b}}\ (\mathsf{f}\ \mathsf{j}) \end{aligned} \tag{52}$$

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

The relationships are more clearly expressed by

$$\begin{array}{ccc} \mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y} & \xrightarrow{\mathsf{lift}_{\mathsf{b}}} & \mathsf{x} \rightsquigarrow_{\mathsf{b}} \mathsf{y} \\ {\scriptstyle \eta_{\mathsf{a}^*}}\downarrow & & \downarrow{\scriptstyle \eta_{\mathsf{b}^*}} \\ \mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y} & \xrightarrow[\mathsf{lift}_{\mathsf{b}^*}]{} & \mathsf{x} \rightsquigarrow_{\mathsf{b}^*} \mathsf{y} \end{array} \tag{53}$$

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 7.8 (natural transformation)**.** *If* $\mathsf{lift}_{\mathsf{b}}$ *is an arrow homomorphism, then* (53) *commutes.*

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

$$\begin{aligned} \mathsf{lift}_{\mathsf{b}^*}\ (\eta_{\mathsf{a}^*}\ \mathsf{f}) &\equiv \lambda \mathsf{j}.\, \mathsf{lift}_{\mathsf{b}}\ (\mathsf{arr}_{\mathsf{a}}\ \mathsf{snd} \ggg_{\mathsf{a}} \mathsf{f}) \\ &\equiv \lambda \mathsf{j}.\, \mathsf{lift}_{\mathsf{b}}\ (\mathsf{arr}_{\mathsf{a}}\ \mathsf{snd}) \ggg_{\mathsf{b}} \mathsf{lift}_{\mathsf{b}}\ \mathsf{f} \\ &\equiv \lambda \mathsf{j}.\, \mathsf{arr}_{\mathsf{b}}\ \mathsf{snd} \ggg_{\mathsf{b}} \mathsf{lift}_{\mathsf{b}}\ \mathsf{f} \\ &\equiv \eta_{\mathsf{b}^*}\ (\mathsf{lift}_{\mathsf{b}}\ \mathsf{f}) \end{aligned}$$

□

Theorem 7.9 (AStore arrow correctness)**.** *If* $\mathsf{lift}_{\mathsf{b}}$ *is an arrow homomorphism from* a *to* b*, then* $\mathsf{lift}_{\mathsf{b}^*}$ *is an arrow homomorphism from* a^* *to* b^**.*

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

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

Distributing $\mathsf{lift}_{\mathsf{b}^*}$ over $\mathsf{lazy}_{\mathsf{a}^*}$ requires defining an extra thunk before the last step. □

Corollary 7.10 (effectful semantic correctness)**.** *If* $\mathsf{lift}_{\mathsf{b}}$ *is an arrow homomorphism, then for all expressions* e*,* $[\![e]\!]_{\mathsf{b}^*} \equiv \mathsf{lift}_{\mathsf{b}^*}\ [\![e]\!]_{\mathsf{a}^*}$ *and* $[\![e]\!]^{\Downarrow}_{\mathsf{b}^*} \equiv \mathsf{lift}_{\mathsf{b}^*}\ [\![e]\!]^{\Downarrow}_{\mathsf{a}^*}$*.*

Corollary 7.11 (mapping* and preimage* arrow correctness)**.** *The following diagram commutes:*

$$\begin{array}{ccccc} \mathsf{X} \rightsquigarrow_{\perp} \mathsf{Y} & \xrightarrow{\mathsf{lift}_{\mathsf{map}}} & \mathsf{X} \overset{\rightsquigarrow}{\underset{\mathsf{map}}{}} \mathsf{Y} & \xrightarrow{\mathsf{lift}_{\mathsf{pre}}} & \mathsf{X} \overset{\rightsquigarrow}{\underset{\mathsf{pre}}{}} \mathsf{Y} \\ {\scriptstyle \eta_{\perp^*}}\downarrow & & \downarrow{\scriptstyle \eta_{\mathsf{map}^*}} & & \downarrow{\scriptstyle \eta_{\mathsf{pre}^*}} \\ \mathsf{X} \rightsquigarrow_{\perp^*} \mathsf{Y} & \xrightarrow[\mathsf{lift}_{\mathsf{map}^*}]{} & \mathsf{X} \overset{\rightsquigarrow}{\underset{\mathsf{map}^*}{}} \mathsf{Y} & \xrightarrow[\mathsf{lift}_{\mathsf{pre}^*}]{} & \mathsf{X} \overset{\rightsquigarrow}{\underset{\mathsf{pre}^*}{}} \mathsf{Y} \end{array} \tag{54}$$

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

Corollary 7.12 (effectful semantic correctness)**.** *For all expressions* e*,*

$$\begin{aligned} [\![e]\!]_{\mathsf{pre}^*} &\equiv \mathsf{lift}_{\mathsf{pre}^*}\ (\mathsf{lift}_{\mathsf{map}^*}\ [\![e]\!]_{\perp^*}) \\ [\![e]\!]^{\Downarrow}_{\mathsf{pre}^*} &\equiv \mathsf{lift}_{\mathsf{pre}^*}\ (\mathsf{lift}_{\mathsf{map}^*}\ [\![e]\!]^{\Downarrow}_{\perp^*}) \end{aligned} \tag{55}$$

7.6 Termination

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

Definition 7.13 (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 first need to make the intuition that computations "act as if they return restricted mappings" more precise.

Theorem 7.14 (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 4.6) there exists an $\mathsf{f} : \mathsf{X} \rightsquigarrow_\perp \mathsf{Y}$ such that $\mathsf{g} \equiv \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}$.

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

□

Theorem 7.15 (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 7.16. *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 $[\![e]\!]_{\perp^*}$ loops 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 7.17 (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* $\mathsf{J}' = \mathsf{J}'' \uplus \{\mathsf{right}\ \mathsf{j}\}$.

$\mathsf{J}\backslash\mathsf{J}'$ *is the corresponding* ***index suffix****.*

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 7.18. *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 7.19. *For all expressions* e*, the maximal domain of* $[\![e]\!]^{\Downarrow}_{\perp^*}$ *is a subset of that of* $[\![e]\!]_{\perp^*}$*.*

Corollary 7.20. *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 7.21 (correct computation everywhere)**.** *Suppose* $[\![e]\!]^{\Downarrow}_{\perp^*} : \mathsf{X} \rightsquigarrow_{\perp^*} \mathsf{Y}$ *has maximal domain* A^**. Let* $\mathsf{X}' := (\mathsf{R} \times \mathsf{T}) \times \mathsf{X}$*. For all* $\mathsf{a} \in \mathsf{X}'$, $\mathsf{A} \subseteq \mathsf{X}'$ *and* $\mathsf{B} \subseteq \mathsf{Y}$*,*

$$\begin{aligned}
[\![e]\!]^{\Downarrow}_{\perp^*}\ \mathsf{j}_0\ \mathsf{a} &= \mathsf{if}\ (\mathsf{a} \in \mathsf{A}^*)\ ([\![e]\!]_{\perp^*}\ \mathsf{j}_0\ \mathsf{a})\ \perp \\
[\![e]\!]^{\Downarrow}_{\mathsf{map}^*}\ \mathsf{j}_0\ \mathsf{A} &= [\![e]\!]_{\mathsf{map}^*}\ \mathsf{j}_0\ (\mathsf{A} \cap \mathsf{A}^*) \\
\mathsf{ap}_{\mathsf{pre}}\ ([\![e]\!]^{\Downarrow}_{\mathsf{pre}^*}\ \mathsf{j}_0\ \mathsf{A})\ \mathsf{B} &= \mathsf{ap}_{\mathsf{pre}}\ ([\![e]\!]_{\mathsf{pre}^*}\ \mathsf{j}_0\ (\mathsf{A} \cap \mathsf{A}^*))\ \mathsf{B}
\end{aligned} \tag{56}$$

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.

8. Output Probabilities and Measurability

We have not assigned probabilities to any output sets yet.

Typically, for $\mathsf{g} \in \mathsf{X} \rightharpoonup \mathsf{Y}$, the probability of $\mathsf{B} \subseteq \mathsf{Y}$ is

$$\mathsf{P}\ (\mathsf{preimage}\ \mathsf{g}\ \mathsf{B}) \tag{57}$$

where $\mathsf{P} \in \mathcal{P}\ \mathsf{X} \rightharpoonup [0, 1]$.

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

$$\mathsf{P}\ (\mathsf{image}\ (\mathsf{fst} \ggg \mathsf{fst})\ \mathsf{A}) \tag{58}$$

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.

8.1 Measurability

We assume readers are familiar with topology. Readers unfamiliar with topology or measure theory may wish to skip to Section 9.

Many topological concepts have analogues in measure theory; e.g. the analogue of a topology is a σ-algebra.

Definition 8.1 (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 8.2 (measurable mapping)**.** *Let* $\mathcal{A}$ *and* $\mathcal{B}$ *be σ-algebras respectively on* X *and* Y*. A mapping* $\mathsf{g} : \mathsf{X} \rightharpoonup \mathsf{Y}$ *is* ***$\mathcal{A}$-$\mathcal{B}$-measurable*** *if for all* $\mathsf{B} \in \mathcal{B}$, $\mathsf{preimage}\ \mathsf{g}\ \mathsf{B} \in \mathcal{A}$*.*

Measurability is usually a weaker condition than continuity. For example, with respect to the σ-algebra generated from $\mathbb{R}$'s standard topology, measurable $\mathbb{R} \rightharpoonup \mathbb{R}$ functions may have countably many discontinuities. Likewise, real equality and inequality functions are measurable.

Product spaces are defined the same way as in topology.

Definition 8.3 (finite product σ-algebra)**.** *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be σ-algebras on X_1 and X_2, and $\mathsf{X} := \langle \mathsf{X}_1, \mathsf{X}_2 \rangle$. The **product σ-algebra** $\mathcal{A}_1 \otimes \mathcal{A}_2$ is the smallest σ-algebra for which* $\mathsf{mapping\ fst\ X}$ *and* $\mathsf{mapping\ snd\ X}$ *are measurable.*

Definition 8.4 (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.*

8.2 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 resulting theorems to prove that the interpretations of all partial, probabilistic expressions are measurable.

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

Definition 8.5 (measurable mapping arrow computation)**.** *Let $\mathcal{A}$ and $\mathcal{B}$ be σ-algebras on X and Y. A computation $\mathsf{g} : \mathsf{X} \rightsquigarrow_{\mathsf{map}} \mathsf{Y}$ is $\mathcal{A}$-$\mathcal{B}$-**measurable** if $\mathsf{g\ A^*}$ is an $\mathcal{A}$-$\mathcal{B}$-measurable mapping, where $\mathsf{A^*}$ is g's maximal domain.*

Theorem 8.6 (maximal domain measurability)**.** *Let $\mathsf{g} : \mathsf{X} \rightsquigarrow_{\mathsf{map}} \mathsf{Y}$ be an $\mathcal{A}$-$\mathcal{B}$-measurable mapping arrow computation. Its maximal domain $\mathsf{A^*}$ is in $\mathcal{A}$.*

Proof. By definition, $\mathsf{g\ A^*}$ is a measurable mapping. $\mathsf{Y} \in \mathcal{B}$, and $\mathsf{preimage\ (g\ A^*)\ Y = domain\ (g\ A^*) = A^*}$. □

Of course, 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 $\mathsf{A^*}$. Fortunately, that is true without any extra conditions.

Imported Lemma 8.7. *Let $\mathsf{g} : \mathsf{X} \rightharpoonup \mathsf{Y}$ be an $\mathcal{A}$-$\mathcal{B}$-measurable mapping. For any $\mathsf{A} \in \mathcal{A}$,* $\mathsf{restrict\ g\ A}$ *is $\mathcal{A}$-$\mathcal{B}$-measurable.*

Theorem 8.8. *Let $\mathsf{g} : \mathsf{X} \rightsquigarrow_{\mathsf{map}} \mathsf{Y}$ be an $\mathcal{A}$-$\mathcal{B}$-measurable mapping arrow computation with maximal domain $\mathsf{A^*}$. For all $\mathsf{A} \subseteq \mathsf{A^*}$ with $\mathsf{A} \in \mathcal{A}$, $\mathsf{g\ A}$ is an $\mathcal{A}$-$\mathcal{B}$-measurable mapping.*

Proof. By Theorem 7.14 (mapping arrow restriction) and Lemma 8.7. □

We do not need to prove that all interpretations using $[\![\cdot]\!]_{\mathsf{a}}$ are measurable. However, we do need to prove that all the 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.

Imported Lemma 8.9 (images of preimages)**.** *Let $\mathsf{g} : \mathsf{X} \rightharpoonup \mathsf{Y}$ and $\mathsf{B} \subseteq \mathsf{Y}$. Then* $\mathsf{image\ g\ (preimage\ g\ B)} \subseteq \mathsf{B}$.

Imported Lemma 8.10 (expanded post-composition)**.** *Let $\mathsf{g_1} : \mathsf{X} \rightharpoonup \mathsf{Y}$ and $\mathsf{g_2} : \mathsf{Y} \rightharpoonup \mathsf{Z}$ such that* $\mathsf{range\ g_1} \subseteq \mathsf{domain\ g_2}$*, 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 8.11 (mapping arrow monotonicity)**.** *Let $\mathsf{g} : \mathsf{X} \rightsquigarrow_{\mathsf{map}} \mathsf{Y}$. For any $\mathsf{A'} \subseteq \mathsf{A} \subseteq \mathsf{A^*}$, $\mathsf{g\ A'} \subseteq \mathsf{g\ A}$.*

Proof. By Theorem 7.14 (mapping arrow restriction). □

Theorem 8.12 (maximal domain subsets)**.** *Let $\mathsf{g} : \mathsf{X} \rightsquigarrow_{\mathsf{map}} \mathsf{Y}$. For any $\mathsf{A} \subseteq \mathsf{A^*}$,* $\mathsf{domain\ (g\ A)} = \mathsf{A}$.

Proof. Follows from Theorem 7.15. □

Now we can prove measurability.

Imported Lemma 8.13 (measurability under $\circ_{\mathsf{map}}$)**.** *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 8.14 (measurability under $(\ggg_{\mathsf{map}})$)**.** *If $\mathsf{g_1} : \mathsf{X} \rightsquigarrow_{\mathsf{map}} \mathsf{Y}$ is $\mathcal{A}$-$\mathcal{B}$-measurable and $\mathsf{g_2} : \mathsf{Y} \rightsquigarrow_{\mathsf{map}} \mathsf{Z}$ 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 $\mathsf{g_1}$'s and $\mathsf{g_2}$'s maximal domains. The maximal domain of $\mathsf{g_1} \ggg_{\mathsf{map}} \mathsf{g_2}$ is $\mathsf{A^{**}} := \mathsf{preimage\ (g_1\ A^*)\ B^*}$, which is in $\mathcal{A}$. By definition,

$$\begin{array}{rcl} \mathsf{(g_1 \ggg_{map} g_2)\ A^{**}} & = & \mathsf{let\ \ g_1' := g_1\ A^{**}} \\ & & \quad\ \mathsf{g_2' := g_2\ (range\ g_1')} \\ & & \mathsf{in\ \ g_2' \circ_{map} g_1'} \end{array} \tag{59}$$

By Theorem 8.8, $\mathsf{g_1'}$ is an $\mathcal{A}$-$\mathcal{B}$-measurable mapping. Unfortunately, $\mathsf{g_2'}$ may not be $\mathcal{B}$-$\mathcal{C}$-measurable when $\mathsf{range\ g_1'} \notin \mathcal{B}$.

Let $\mathsf{g_2''} := \mathsf{g_2\ B^*}$, which is a $\mathcal{B}$-$\mathcal{C}$-measurable mapping. By Lemma 8.13, $\mathsf{g_2'' \circ_{map} g_1'}$ is $\mathcal{A}$-$\mathcal{C}$-measurable. We need only show that $\mathsf{g_2' \circ_{map} g_1' = g_2'' \circ_{map} g_1'}$, which by Lemma 8.10 is true if $\mathsf{range\ g_1' \subseteq domain\ g_2'}$ and $\mathsf{g_2' \subseteq g_2''}$.

By Theorem 8.12, $\mathsf{A^{**}} \subseteq \mathsf{A^*}$ implies $\mathsf{domain\ g_1' = A^{**}}$. By Theorem 8.11 and Lemma 8.9,

$$\begin{array}{rcl} \mathsf{range\ g_1'} & = & \mathsf{image\ (g_1\ A^{**})\ (preimage\ (g_1\ A^*)\ B^*)} \\ & = & \mathsf{image\ (g_1\ A^*)\ (preimage\ (g_1\ A^*)\ B^*)} \\ & \subseteq & \mathsf{B^*} \end{array}$$

$\mathsf{range\ g_1'} \subseteq \mathsf{B^*}$ implies (by Theorem 8.12) that $\mathsf{domain\ g_2'} = \mathsf{range\ g_1'}$, and (by Theorem 8.11) that $\mathsf{g_2' \subseteq g_2''}$. □

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

Imported Lemma 8.15 (measurability under $\langle \cdot, \cdot \rangle_{\mathsf{map}}$)**.** *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 8.16 (measurability under $(\&\&\&_{\mathsf{map}})$)**.** *If $\mathsf{g_1} : \mathsf{X} \rightsquigarrow_{\mathsf{map}} \mathsf{Y_1}$ is $\mathcal{A}$-$\mathcal{B}_1$-measurable and $\mathsf{g_2} : \mathsf{X} \rightsquigarrow_{\mathsf{map}} \mathsf{Y_2}$ 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 $\mathsf{A_1^*}$ and $\mathsf{A_2^*}$ be respectively $\mathsf{g_1}$'s and $\mathsf{g_2}$'s maximal domains. The maximal domain of $\mathsf{g_1} \&\&\&_{\mathsf{map}} \mathsf{g_2}$ is $\mathsf{A^{**}} := \mathsf{A_1^* \cap A_2^*}$, which is in $\mathcal{A}$. By definition, $\mathsf{(g_1 \&\&\&_{map} g_2)\ A^{**}} = \langle \mathsf{g_1\ A^{**}}, \mathsf{g_2\ A^{**}} \rangle_{\mathsf{map}}$, which by Lemma 8.15 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.

Imported Lemma 8.17 (union of disjoint, 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 8.18 (measurability under $\mathsf{ifte}_{\mathsf{map}}$)**.** *If $\mathsf{g_1} : \mathsf{X} \rightsquigarrow_{\mathsf{map}} \mathsf{Bool}$ is $\mathcal{A}$-$(\mathcal{P}\ \mathsf{Bool})$-measurable, and $\mathsf{g_2} : \mathsf{X} \rightsquigarrow_{\mathsf{map}} \mathsf{Y}$ and $\mathsf{g_3} : \mathsf{X} \rightsquigarrow_{\mathsf{map}} \mathsf{Y}$ are $\mathcal{A}$-$\mathcal{B}$-measurable, then $\mathsf{ifte_{map}\ g_1\ g_2\ g_3}$ 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

$$\begin{aligned} \mathsf{A}_2^{**} &:= \mathsf{A}_2^* \cap \mathsf{preimage}\ (\mathsf{g}_1\ \mathcal{A}_1^*)\ \{\mathsf{true}\} \\ \mathsf{A}_3^{**} &:= \mathsf{A}_3^* \cap \mathsf{preimage}\ (\mathsf{g}_1\ \mathcal{A}_1^*)\ \{\mathsf{false}\} \\ \mathsf{A}^{**} &:= \mathsf{A}_2^{**} \uplus \mathsf{A}_3^{**} \end{aligned} \tag{60}$$

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,

$$\begin{aligned} \mathsf{ifte}_{\mathsf{map}}\ \mathsf{g}_1\ \mathsf{g}_2\ \mathsf{g}_3\ \mathsf{A}^{**} \ =\ \mathsf{let}\ & \mathsf{g}_1' := \mathsf{g}_1\ \mathsf{A}^{**} \\ & \mathsf{g}_2' := \mathsf{g}_2\ (\mathsf{preimage}\ \mathsf{g}_1'\ \{\mathsf{true}\}) \\ & \mathsf{g}_3' := \mathsf{g}_3\ (\mathsf{preimage}\ \mathsf{g}_1'\ \{\mathsf{false}\}) \\ \mathsf{in}\ & \mathsf{g}_2' \uplus_{\mathsf{map}} \mathsf{g}_3' \end{aligned} \tag{61}$$

By hypothesis, g_1', g_2' and g_3' are measurable mappings. By Theorem 7.14 (mapping arrow restriction), g_2' and g_3' have disjoint domains. Apply Lemma 8.17. □

Laziness

Theorem 8.19 (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 8.20 (measurability under $\mathsf{lazy}_{\mathsf{map}}$)**.** *Let* $\mathsf{g} : 1 \Rightarrow (\mathsf{X} \overset{\rightsquigarrow}{\mathsf{map}} \mathsf{Y})$. *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 the same as that of $\mathsf{g}\ 0$. By definition,

$$\mathsf{lazy}_{\mathsf{map}}\ \mathsf{g}\ \mathsf{A}^{**} \ =\ \mathsf{if}\ (\mathsf{A}^{**} = \varnothing)\ \varnothing\ (\mathsf{g}\ 0\ \mathsf{A}^{**}) \tag{62}$$

If $\mathsf{A}^{**} = \varnothing$, then $\mathsf{lazy}_{\mathsf{map}}\ \mathsf{g}\ \mathsf{A}^{**} = \varnothing$; apply Theorem 8.19. If $\mathsf{A}^{**} \neq \varnothing$, then $\mathsf{lazy}_{\mathsf{map}}\ \mathsf{g} = \mathsf{g}\ 0$, which is $\mathcal{A}$-$\mathcal{B}$-measurable. □

8.3 Measurable Probabilistic Computations

As before, we first need to define what it means for a computation to be measurable.

Definition 8.21 (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 $\mathsf{g} : \mathsf{X} \overset{\rightsquigarrow}{\mathsf{map}^*} \mathsf{Y}$ is $\mathcal{A}$-$\mathcal{B}$-**measurable** if $\mathsf{g}\ \mathsf{j}_0$ is an $\mathcal{A}$-$\mathcal{B}$-measurable mapping arrow computation.*

Clearly, if any $\mathsf{g}\ \mathsf{j}$ is measurable, so are $\mathsf{g}\ (\mathsf{left}\ \mathsf{j})$ and $\mathsf{g}\ (\mathsf{right}\ \mathsf{j})$. By induction, if g is a measurable mapping* arrow computation, then for any $\mathsf{j} \in \mathsf{J}$, $\mathsf{g}\ \mathsf{j}$ is an $\mathcal{A}$-$\mathcal{B}$-measurable mapping arrow computation.

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 8.22 (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:*

$$\Sigma\ \langle \mathsf{X}_1, \mathsf{X}_2 \rangle = \Sigma\ \mathsf{X}_1 \otimes \Sigma\ \mathsf{X}_2 \tag{63}$$

$$\Sigma\ (\mathsf{J} \to \mathsf{X}) = (\Sigma\ \mathsf{X})^{\otimes \mathsf{J}} \tag{64}$$

The predicate "is measurable" means "is measurable with respect to standard σ-algebras."

So that we can measure boolean singletons and any set of branch traces, we define

$$\Sigma\ \mathsf{Bool} \ ::=\ \mathcal{P}\ \mathsf{Bool} \tag{65}$$

$$\Sigma\ \mathsf{T} \ ::=\ \mathcal{P}\ \mathsf{T} \tag{66}$$

Imported Lemma 8.23 (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 8.24. $\mathsf{arr}_{\mathsf{map}^*}\ \mathsf{id}$, $\mathsf{arr}_{\mathsf{map}^*}\ \mathsf{fst}$ *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 8.25 (AStore measurability transfer)**.** *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 8.26. $\mathsf{ifte}^{\Downarrow}_{\mathsf{map}^*}$ *is measurable.*

Proof. $\mathsf{branch}_{\mathsf{map}^*}$ is measurable, and $\mathsf{arr}_{\mathsf{map}}\ \mathsf{agrees}$ is measurable by (65). □

Proving all nonrecursive programs measurable now requires little more than a definition.

Definition 8.27 (finite expression)**.** *A **finite expression** is any expression for which no subexpression is a first-order application.*

Theorem 8.28 (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 8.29 (approximation with finite expressions)**.** *Let* $\mathsf{g} := [\![e]\!]^{\Downarrow}_{\mathsf{map}^*} : \mathsf{X} \overset{\rightsquigarrow}{\mathsf{map}^*} \mathsf{Y}$. *For all* $\mathsf{t} \in \mathsf{T}$, *there is a finite expression e' for which* $[\![e']\!]_{\mathsf{map}^*}\ \mathsf{j}_0\ \mathsf{A} = \mathsf{g}\ \mathsf{j}_0\ \mathsf{A}$, *where* $\mathsf{A} := (\mathsf{R} \times \{\mathsf{t}\}) \times \mathsf{X}$.

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 applications. □

Theorem 8.30 (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 7.21, $\mathsf{g}' = \mathsf{g}\ \mathsf{j}_0\ \mathsf{A}^*$ where A^* is g's maximal domain; thus we need only show that g' is a measurable mapping.

By Theorem 7.14 (mapping arrow restriction),

$$\mathsf{g}' \ =\ \bigcup_{\mathsf{t} \in \mathsf{T}} \mathsf{g}\ \mathsf{j}_0\ ((\mathsf{R} \times \{\mathsf{t}\}) \times \mathsf{X}) \tag{67}$$

By Theorem 8.29, for every $\mathsf{t} \in \mathsf{T}$, there is an expression that computes $\mathsf{g}\ \mathsf{j}_0\ ((\mathsf{R} \times \{\mathsf{t}\}) \times \mathsf{X})$. By (66) and Theorem 8.28, each is measurable. By Theorem 7.14, they have disjoint domains. By Lemma 8.17, their union is measurable. □

Theorem 8.30 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.

8.4 Measurable Projections

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

$$\mathsf{P}\ (\mathsf{image}\ (\mathsf{fst} \ggg \mathsf{fst})\ (\mathsf{preimage}\ (\mathsf{g}\ \mathsf{j}_0\ \mathsf{A}^*)\ \mathsf{B})) \tag{68}$$

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 8.31 (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 8.32. *Let* $\mathsf{g} : \mathsf{X} \overset{\rightsquigarrow}{\underset{\mathsf{map}^*}{}} \mathsf{Y}$ *be measurable. If* X *is at most countable and* $\Sigma\ \mathsf{X} = \mathcal{P}\ \mathsf{X}$, *then for all* $\mathsf{B} \in \Sigma\ \mathsf{Y}$,

$$\mathsf{image}\ (\mathsf{fst} \ggg \mathsf{fst})\ (\mathsf{preimage}\ (\mathsf{g}\ \mathsf{j}_0\ \mathsf{A}^*)\ \mathsf{B}) \ \in\ \Sigma\ \mathsf{R} \tag{69}$$

Proof. T is countable and $\Sigma\ \mathsf{T} = \mathcal{P}\ \mathsf{T}$ by definition (66); apply Theorem 8.31 twice. □

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

9. 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.

Trying to generalize all useful approximation methods would result in a specification that cannot be directly implemented. Instead, we focus on a specific method: approximating product sets with covering rectangles. We recover some generality by stating correctness theorems in terms of general properties such as monotonicity.

9.1 Implementable Lifts

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

Fortunately, we need only certain lifts. Figure 2 (which defines $[\![\cdot]\!]_{\mathsf{a}}$) lifts id, $\mathsf{const}\ \mathsf{b}$, fst and snd. Section 7.4, 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.

Figure 8 gives expressions equivalent to $\mathsf{arr}_{\mathsf{pre}}\ \mathsf{id}$, $\mathsf{arr}_{\mathsf{pre}}\ \mathsf{fst}$, $\mathsf{arr}_{\mathsf{pre}}\ \mathsf{snd}$, $\mathsf{arr}_{\mathsf{pre}}\ (\mathsf{const}\ \mathsf{b})$ and $\mathsf{arr}_{\mathsf{pre}}\ (\pi\ \mathsf{j})$. (We will deal with agrees separately.) By inspecting these expressions, we see that we need to model sets in a way that the following are representable and can be computed in finite time:

- $\mathsf{A} \cap \mathsf{B}$, $\varnothing$, $\{\mathsf{true}\}$, $\{\mathsf{false}\}$ and $\{\mathsf{b}\}$ for every $\mathsf{const}\ \mathsf{b}$
- $\mathsf{A}_1 \times \mathsf{A}_2$, $\mathsf{proj}_{\mathsf{fst}}\ \mathsf{A}$ and $\mathsf{proj}_{\mathsf{snd}}\ \mathsf{A}$
- $\mathsf{J} \to \mathsf{X}$, $\mathsf{project}\ \mathsf{j}\ \mathsf{A}$ and $\mathsf{unproject}\ \mathsf{j}\ \mathsf{A}\ \mathsf{B}$
- $\mathsf{A} = \varnothing$

(70)

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

Definition 9.1 (rectangular family)**.** *For a set* X *used as a type,* $\mathsf{Rect}\ \mathsf{X}$ *denotes the* ***rectangular family*** *of subsets of* X. *For nonproduct* X, $\varnothing \in \mathsf{Rect}\ \mathsf{X}$ *and* $\mathsf{X} \in \mathsf{Rect}\ \mathsf{X}$, *and* $\mathsf{Rect}\ \mathsf{X}$ *must be closed under finite intersections. Products must satisfy the following rules:*

$$\mathsf{Rect}\ \langle \mathsf{X}_1, \mathsf{X}_2 \rangle \ =\ (\mathsf{Rect}\ \mathsf{X}_1) \boxtimes (\mathsf{Rect}\ \mathsf{X}_2) \tag{71}$$

$$\mathsf{Rect}\ (\mathsf{J} \to \mathsf{X}) \ =\ (\mathsf{Rect}\ \mathsf{X})^{\boxtimes \mathsf{J}} \tag{72}$$

where

$$\mathcal{A}_1 \boxtimes \mathcal{A}_2 \ :=\ \{\mathsf{A}_1 \times \mathsf{A}_2 \mid \mathsf{A}_1 \in \mathcal{A}_1, \mathsf{A}_2 \in \mathcal{A}_2\} \tag{73}$$

$$\mathcal{A}^{\boxtimes \mathsf{J}} \ :=\ \bigcup_{\mathsf{J}' \subset \mathsf{J}\ \mathit{finite}} \left\{ \prod\nolimits_{\mathsf{j} \in \mathsf{J}} \mathsf{A}_{\mathsf{j}} \;\middle|\; \mathsf{A}_{\mathsf{j}} \in \mathcal{A}, \mathsf{j} \in \mathsf{J}' \iff \mathsf{A}_{\mathsf{j}} \subset \bigcup \mathcal{A} \right\} \tag{74}$$

lift cartesian products to sets of sets.

For example, if $\mathsf{Rect}\ \mathbb{R}$ contains all the closed real intervals, then by (71), $[0, 2] \times [1, \pi] \in \mathsf{Rect}\ \langle \mathbb{R}, \mathbb{R} \rangle$.

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

Further, all of the operations in (70) can be exactly implemented if finite sets are modeled directly, sets in an ordered space (such as $\mathbb{R}$) are modeled by intervals, and sets in $\mathsf{Rect}\ \langle \mathsf{X}_1, \mathsf{X}_2 \rangle$ are modeled by pairs of type $\langle \mathsf{Rect}\ \mathsf{X}_1, \mathsf{Rect}\ \mathsf{X}_2 \rangle$. By (72), sets in $\mathsf{Rect}\ (\mathsf{J} \to \mathsf{X})$ have no more than finitely many axes that are proper subsets of X. They can be modeled by *finite* binary trees, whose nodes contain axes for an index prefix $\mathsf{J}' \subset \mathsf{J}$ (Definition 7.17). Axes with indexes in the suffix $\mathsf{J} \backslash \mathsf{J}'$ are implicitly X.

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

Theorem 9.2 (rectangular T projection)**.** *If* $\mathsf{T}' \in \mathsf{Rect}\ (\mathsf{J} \to \mathsf{Bool}_\perp)$, *then* $\mathsf{project}\ \mathsf{j}\ (\mathsf{T}' \cap \mathsf{T}) = \mathsf{project}\ \mathsf{j}\ \mathsf{T}'$ *for all* $\mathsf{j} \in \mathsf{J}$.

Proof. The subset case is by projection monotonicity. For the superset, let $\mathsf{b} \in \mathsf{project}\ \mathsf{j}\ \mathsf{T}'$. Define t by $\mathsf{t}\ \mathsf{j}' = \mathsf{b}$ if $\mathsf{j}' = \mathsf{j}$, $\mathsf{t}\ \mathsf{j}' = \perp$ if $\perp \in \mathsf{project}\ \mathsf{j}'\ \mathsf{T}'$, and $\mathsf{t}\ \mathsf{j}' \in \mathsf{project}\ \mathsf{j}'\ \mathsf{T}'$ otherwise.

For no more than finitely many $\mathsf{j}' \in \mathsf{J}$, $\mathsf{t}\ \mathsf{j}' \neq \perp$, so $\mathsf{t} \in \mathsf{T}$; also $\mathsf{t} \in \mathsf{T}'$ by construction. Thus, there exists a $\mathsf{t} \in \mathsf{T}' \cap \mathsf{T}$ such that $\mathsf{t}\ \mathsf{j} = \mathsf{b}$, so $\mathsf{b} \in \mathsf{project}\ \mathsf{j}\ (\mathsf{T}' \cap \mathsf{T})$. □

Corollary 9.3. *Under the same conditions, for all* $\mathsf{B} \subseteq \mathsf{Bool}$, $\mathsf{unproject}\ \mathsf{j}\ (\mathsf{T}' \cap \mathsf{T})\ \mathsf{B} = \mathsf{unproject}\ \mathsf{j}\ \mathsf{T}'\ \mathsf{B} \cap \mathsf{T}$.

9.2 Approximate Preimage Mapping Operations

Implementing $\mathsf{lazy}_{\mathsf{pre}}$ (defined in Figure 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 implementing the preimage mapping operations $(\circ_{\mathsf{pre}})$, $\langle \cdot, \cdot \rangle_{\mathsf{pre}}$ and $(\uplus_{\mathsf{pre}})$.

$$\begin{aligned}
\mathsf{id}_{\mathsf{pre}}\ \mathsf{A} &:= \mathsf{arr}_{\mathsf{pre}}\ \mathsf{id}\ \mathsf{A} \equiv \langle \mathsf{A}, \lambda\mathsf{B}.\,\mathsf{B}\rangle \\
\mathsf{fst}_{\mathsf{pre}}\ \mathsf{A} &:= \mathsf{arr}_{\mathsf{pre}}\ \mathsf{fst}\ \mathsf{A} \equiv \langle \mathsf{proj}_{\mathsf{fst}}\ \mathsf{A}, \mathsf{unproj}_{\mathsf{fst}}\ \mathsf{A}\rangle \\
\mathsf{snd}_{\mathsf{pre}}\ \mathsf{A} &:= \mathsf{arr}_{\mathsf{pre}}\ \mathsf{snd}\ \mathsf{A} \equiv \langle \mathsf{proj}_{\mathsf{snd}}\ \mathsf{A}, \mathsf{unproj}_{\mathsf{snd}}\ \mathsf{A}\rangle
\end{aligned}$$

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

$$\begin{aligned}
&\mathsf{unproj}_{\mathsf{fst}} : \mathsf{Set}\ \langle \mathsf{X}_1, \mathsf{X}_2\rangle \Rightarrow \mathsf{Set}\ \mathsf{X}_1 \Rightarrow \mathsf{Set}\ \langle \mathsf{X}_1, \mathsf{X}_2\rangle \\
&\mathsf{unproj}_{\mathsf{fst}}\ \mathsf{A}\ \mathsf{B} := \mathsf{preimage}\ (\mathsf{mapping}\ \mathsf{fst}\ \mathsf{A})\ \mathsf{B} \\
&\qquad\qquad \equiv \mathsf{A} \cap (\mathsf{B} \times \mathsf{proj}_{\mathsf{snd}}\ \mathsf{A})
\end{aligned}$$

$$\begin{aligned}
\mathsf{const}_{\mathsf{pre}}\ \mathsf{b}\ \mathsf{A} &:= \mathsf{arr}_{\mathsf{pre}}\ (\mathsf{const}\ \mathsf{b})\ \mathsf{A} \equiv \langle \{\mathsf{b}\}, \lambda\mathsf{B}.\,\mathsf{if}\ (\mathsf{B} = \varnothing)\ \varnothing\ \mathsf{A}\rangle \\
\pi_{\mathsf{pre}}\ \mathsf{j}\ \mathsf{A} &:= \mathsf{arr}_{\mathsf{pre}}\ (\pi\ \mathsf{j})\ \mathsf{A} \equiv \langle \mathsf{project}\ \mathsf{j}\ \mathsf{A}, \mathsf{unproject}\ \mathsf{j}\ \mathsf{A}\rangle
\end{aligned}$$

$$\begin{aligned}
&\mathsf{project} : \mathsf{J} \Rightarrow \mathsf{Set}\ (\mathsf{J} \to \mathsf{X}) \Rightarrow \mathsf{Set}\ \mathsf{X} \\
&\mathsf{project}\ \mathsf{j}\ \mathsf{A} := \mathsf{image}\ (\pi\ \mathsf{j})\ \mathsf{A}
\end{aligned}$$

$$\begin{aligned}
&\mathsf{unproject} : \mathsf{J} \Rightarrow \mathsf{Set}\ (\mathsf{J} \to \mathsf{X}) \Rightarrow \mathsf{Set}\ \mathsf{X} \Rightarrow \mathsf{Set}\ (\mathsf{J} \to \mathsf{X}) \\
&\mathsf{unproject}\ \mathsf{j}\ \mathsf{A}\ \mathsf{B} := \mathsf{preimage}\ (\mathsf{mapping}\ (\pi\ \mathsf{j})\ \mathsf{A})\ \mathsf{B} \\
&\qquad\qquad \equiv \mathsf{A} \cap \textstyle\prod_{\mathsf{i}\in\mathsf{J}} \mathsf{if}\ (\mathsf{j} = \mathsf{i})\ \mathsf{B}\ (\mathsf{project}\ \mathsf{j}\ \mathsf{A})
\end{aligned}$$

Figure 8: Preimage arrow lifts needed to interpret probabilistic programs. The definition of $\mathsf{unproj}_{\mathsf{snd}}$ is like $\mathsf{unproj}_{\mathsf{fst}}$'s.

From the preimage mapping definitions (Figure 5), we see that $\mathsf{ap}_{\mathsf{pre}}$ is defined in terms of $(\cap)$ and that $(\circ_{\mathsf{pre}})$ is defined in terms of $\mathsf{ap}_{\mathsf{pre}}$, so $(\circ_{\mathsf{pre}})$ is directly implementable. Unfortunately, we hit a snag with $\langle\cdot,\cdot\rangle_{\mathsf{pre}}$: it loops over possibly uncountably many members of B in a big union. At this point, we need to approximate.

Theorem 9.4 (pair preimage overapproximation)**.** *Let* $\mathsf{g}_1 \in \mathsf{X} \rightharpoonup \mathsf{Y}_1$ *and* $\mathsf{g}_2 \in \mathsf{X} \rightharpoonup \mathsf{Y}_2$*. For all* $\mathsf{B} \subseteq \mathsf{Y}_1 \times \mathsf{Y}_2$*,* $\mathsf{preimage}\ \langle \mathsf{g}_1, \mathsf{g}_2\rangle_{\mathsf{map}}\ \mathsf{B} \subseteq \mathsf{preimage}\ \mathsf{g}_1\ (\mathsf{proj}_{\mathsf{fst}}\ \mathsf{B}) \cap \mathsf{preimage}\ \mathsf{g}_2\ (\mathsf{proj}_{\mathsf{snd}}\ \mathsf{B})$.

Proof. By monotonicity of preimages and projections, and by Lemma 5.4. □

It is not hard to use Theorem 9.4 to show that

$$\begin{aligned}
&\langle\cdot,\cdot\rangle'_{\mathsf{pre}} : (\mathsf{X} \overset{\to}{\underset{\mathsf{pre}}{}} \mathsf{Y}_1) \Rightarrow (\mathsf{X} \overset{\to}{\underset{\mathsf{pre}}{}} \mathsf{Y}_2) \Rightarrow (\mathsf{X} \overset{\to}{\underset{\mathsf{pre}}{}} \mathsf{Y}_1 \times \mathsf{Y}_2) \\
&\langle\langle \mathsf{Y}'_1, \mathsf{p}_1\rangle, \langle \mathsf{Y}'_2, \mathsf{p}_2\rangle\rangle'_{\mathsf{pre}} := \\
&\qquad \langle \mathsf{Y}'_1 \times \mathsf{Y}'_2, \lambda\mathsf{B}.\,\mathsf{p}_1\ (\mathsf{proj}_{\mathsf{fst}}\ \mathsf{B}) \cap \mathsf{p}_2\ (\mathsf{proj}_{\mathsf{snd}}\ \mathsf{B})\rangle
\end{aligned} \tag{75}$$

computes covering rectangles of preimages under pairing.

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

$$\begin{aligned}
(\mathsf{A}_1 \times \mathsf{A}_2) \vee (\mathsf{B}_1 \times \mathsf{B}_2) &= (\mathsf{A}_1 \vee \mathsf{B}_1) \times (\mathsf{A}_2 \vee \mathsf{B}_2) \\
(\textstyle\prod_{\mathsf{j}\in\mathsf{J}} \mathsf{A}_\mathsf{j}) \vee (\textstyle\prod_{\mathsf{j}\in\mathsf{J}} \mathsf{B}_\mathsf{j}) &= \textstyle\prod_{\mathsf{j}\in\mathsf{J}} \mathsf{A}_\mathsf{j} \vee \mathsf{B}_\mathsf{j}
\end{aligned} \tag{76}$$

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

Replacing each union in $(\uplus_{\mathsf{pre}})$ with a join results in

$$\begin{aligned}
&(\uplus'_{\mathsf{pre}}) : (\mathsf{X} \overset{\to}{\underset{\mathsf{pre}}{}} \mathsf{Y}) \Rightarrow (\mathsf{X} \overset{\to}{\underset{\mathsf{pre}}{}} \mathsf{Y}) \Rightarrow (\mathsf{X} \overset{\to}{\underset{\mathsf{pre}}{}} \mathsf{Y}) \\
&\mathsf{h}_1 \uplus'_{\mathsf{pre}} \mathsf{h}_2 := \mathsf{let}\ \ \mathsf{Y}' := (\mathsf{range}_{\mathsf{pre}}\ \mathsf{h}_1) \vee (\mathsf{range}_{\mathsf{pre}}\ \mathsf{h}_2) \\
&\qquad\qquad\qquad\quad \mathsf{p} := \lambda\mathsf{B}.\,(\mathsf{ap}_{\mathsf{pre}}\ \mathsf{h}_1\ \mathsf{B}) \vee (\mathsf{ap}_{\mathsf{pre}}\ \mathsf{h}_2\ \mathsf{B}) \\
&\qquad\qquad\qquad \mathsf{in}\ \ \langle \mathsf{Y}', \mathsf{p}\rangle
\end{aligned} \tag{77}$$

which overapproximates $(\uplus_{\mathsf{pre}})$.

To interpret programs that may not terminate, or that terminate with probability 1, we need to approximate $\mathsf{ifte}^{\Downarrow}_{\mathsf{pre}^*}$ (47), 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:

$$\begin{aligned}
\mathsf{ifte}^{\Downarrow}_{\mathsf{pre}^*}\ \mathsf{k}_1\ \mathsf{k}_2\ \mathsf{k}_3\ \mathsf{j}\ \mathsf{A} \equiv\ \mathsf{let}\ \ &\langle \mathsf{C}_\mathsf{k}, \mathsf{p}_\mathsf{k}\rangle := \mathsf{k}_1\ \mathsf{j}_1\ \mathsf{A} \\
&\langle \mathsf{C}_\mathsf{b}, \mathsf{p}_\mathsf{b}\rangle := \mathsf{branch}_{\mathsf{pre}^*}\ \mathsf{j}\ \mathsf{A} \\
&\mathsf{C}_2 := \mathsf{C}_\mathsf{k} \cap \mathsf{C}_\mathsf{b} \cap \{\mathsf{true}\} \\
&\mathsf{C}_3 := \mathsf{C}_\mathsf{k} \cap \mathsf{C}_\mathsf{b} \cap \{\mathsf{false}\} \\
&\mathsf{A}_2 := \mathsf{p}_\mathsf{k}\ \mathsf{C}_2 \cap \mathsf{p}_\mathsf{b}\ \mathsf{C}_2 \\
&\mathsf{A}_3 := \mathsf{p}_\mathsf{k}\ \mathsf{C}_3 \cap \mathsf{p}_\mathsf{b}\ \mathsf{C}_3 \\
\mathsf{in}\ \ &(\mathsf{k}_2\ \mathsf{j}_2\ \mathsf{A}_2) \uplus_{\mathsf{pre}} (\mathsf{k}_3\ \mathsf{j}_3\ \mathsf{A}_3)
\end{aligned} \tag{78}$$

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 $\varnothing$.

C_b is the branch trace projection at j (with $\bot$ removed). The set of indexes for which 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 $\mathsf{C}_\mathsf{b} = \{\mathsf{true},\mathsf{false}\}$, 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 $\mathsf{C}_\mathsf{b} = \{\mathsf{true},\mathsf{false}\}$. 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 $\mathsf{Rect}\ \mathsf{Y}$.

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

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

Theorem 9.5 (soundness)**.** *For all* $\mathsf{A} \in \mathsf{Rect}\ \langle(\mathsf{R},\mathsf{T}\rangle, \mathsf{X}\rangle$ *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. □

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

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

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

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

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

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

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

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

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

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

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

(b) An approximating preimage arrow, defined in terms of approximating preimage mappings.

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

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

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

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

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

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

(c) An approximating preimage* arrow, defined in terms of the approximating preimage arrow.

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

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

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

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

(d) Additional preimage* arrow combinators, for retrieving random numbers and branch traces, and guaranteed termination.

Figure 9: Implementable arrows that approximate preimage arrows. Because $\mathsf{arr}_{\mathsf{pre}}$ is generally uncomputable, there is no corresponding $\mathsf{arr}'_{\mathsf{pre}}$ combinator. However, specific lifts such as $\mathsf{fst}_{\mathsf{pre}} := \mathsf{arr}_{\mathsf{pre}}\ \mathsf{fst}$ are computable, and are defined in Figure 8.

To use structural induction on the interpretation of e, we need a theorem that allows representing it as a finite expression (Definition 8.27). 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 9.6 (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 $\mathsf{T}' := \mathsf{proj}_{\mathsf{snd}}\ (\mathsf{proj}_{\mathsf{fst}}\ \mathsf{A}')$, and the index prefix J' contain every j' for which $(\mathsf{project}\ \mathsf{j}'\ \mathsf{T}')\backslash\{\bot\}$ 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 $\mathsf{if}\ e_1 \perp \perp$. Because e is well-defined, recurrences must be guarded by if, so this process terminates after finitely many applications. □

Corollary 9.7 (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 9.8 (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 Figure 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 (Figure 9) 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 9.6 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 9.6 with $\mathsf{A}' := \mathsf{A}$, and structural induction. □

Theorem 9.9 (decreasing)**.** *For* $\mathsf{A} \in \mathsf{Rect}\ \langle\langle \mathsf{R}, \mathsf{T}\rangle, \mathsf{X}\rangle$ *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 Figure 8 are decreasing.

By definition and applying basic lattice properties,

$$\mathsf{ap}'_{\mathsf{pre}}\ ((\mathsf{h}_1 \ggg'_{\mathsf{pre}} \mathsf{h}_2)\ \mathsf{A})\ \mathsf{B} \equiv \mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}_1\ \mathsf{A})\ \mathsf{B}' \quad \text{for some } \mathsf{B}'$$

$$\mathsf{ap}'_{\mathsf{pre}}\ ((\mathsf{h}_1 \mathbin{\&\&\&}'_{\mathsf{pre}} \mathsf{h}_2)\ \mathsf{A})\ \mathsf{B} \equiv \mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}_1\ \mathsf{A})\ (\mathsf{proj}_{\mathsf{fst}}\ \mathsf{B}) \cap \mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}_2\ \mathsf{A})\ (\mathsf{proj}_{\mathsf{snd}}\ \mathsf{B})$$

$$\mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{ifte}'_{\mathsf{pre}}\ \mathsf{h}_1\ \mathsf{h}_2\ \mathsf{h}_3\ \mathsf{A})\ \mathsf{B} \equiv \mathsf{let}\ \mathsf{A}_2 := \mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}_1\ \mathsf{A})\ \{\mathsf{true}\}\ \mathsf{A}_3 := \mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}_1\ \mathsf{A})\ \{\mathsf{false}\}\ \mathsf{in}\ \mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}_2\ \mathsf{A}_2)\ \mathsf{B} \vee \mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}_3\ \mathsf{A}_3)\ \mathsf{B}$$

$$\mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{lazy}'_{\mathsf{pre}}\ \mathsf{h}\ \mathsf{A})\ \mathsf{B} \equiv \mathsf{if}\ (\mathsf{A} = \varnothing)\ \varnothing\ (\mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}\ 0\ \mathsf{A})\ \mathsf{B})$$

Thus, approximating preimage arrow combinators return decreasing computations when given decreasing computations. This property transfers trivially to approximating preimage* arrow combinators. Use Theorem 9.6 with $\mathsf{A}' := \mathsf{A}$, and structural induction. □

9.4 Preimage Refinement Algorithm

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

Definition 9.10 (preimage refinement algorithm)**.** *Let* $\mathsf{h}' := [\![e]\!]'^{\Downarrow}_{\mathsf{pre}^*} : \mathsf{X} \mathrel{\tilde{\to}}'_{\mathsf{pre}^*} \mathsf{Y}$, $\mathsf{B} \in \mathsf{Rect}\ \mathsf{Y}$, *and define*

$$\begin{aligned} &\mathsf{refine} : \mathsf{Rect}\ \langle\langle \mathsf{R}, \mathsf{T}\rangle, \mathsf{X}\rangle \Rightarrow \mathsf{Rect}\ \langle\langle \mathsf{R}, \mathsf{T}\rangle, \mathsf{X}\rangle \\ &\mathsf{refine}\ \mathsf{A} := \mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}'\ \mathsf{j}_0\ \mathsf{A})\ \mathsf{B} \end{aligned} \tag{79}$$

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*

$$\begin{aligned} &\mathsf{refine}^* : \mathsf{Set}\ (\mathsf{Rect}\ \langle\langle \mathsf{R}, \mathsf{T}\rangle, \mathsf{X}\rangle) \Rightarrow \mathsf{Set}\ (\mathsf{Rect}\ \langle\langle \mathsf{R}, \mathsf{T}\rangle, \mathsf{X}\rangle) \\ &\mathsf{refine}^*\ \mathcal{A} := \mathsf{image}\ \mathsf{refine}\ \left(\textstyle\bigcup_{\mathsf{A}\in\mathcal{A}} \mathsf{partition}\ \mathsf{A}\right) \end{aligned} \tag{80}$$

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

Theorem 9.9 (decreasing) guarantees that $\mathsf{refine}\ \mathsf{A}$ is never larger than A. Theorem 9.8 (monotonicity) guarantees that refining a *partition* of A never does worse than refining A itself. Theorem 9.5 (soundness) guarantees that the algorithm is **sound**: the true preimage of B is always contained in the covering partition refine^* returns.

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

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

For now, we use algorithms that depend only on preimage refinement's soundness.

10. 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***.

10.1 Direct Implementations

If sets are restricted to be finite, the arrows used as translation targets in the exact semantics, defined in Figures 1, 3, 4, 5, 6 and 7, can be implemented directly in any practical λ-calculus with a set data type. Computing exact preimages is very inefficient, even under the interpretations of very small programs. However, we have found our Typed Racket [30] implementation useful for finding theorem candidates.

Given a rectangular set library, the approximating preimage arrows defined in Figures 8 and 9 can be implemented with few changes in any practical λ-calculus. We have done so in Typed Racket and Haskell [1]. Both implementations' arrow combinator definitions are almost line-for-line transliterations from the figures.

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

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

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

10.2 Dr. Bayes

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

$$\begin{aligned}\mathsf{mul}_{\mathsf{pre}} := \mathsf{ifte}_{\mathsf{pre}}\ &(\mathsf{fst}_{\mathsf{pre}} \ggg_{\mathsf{pre}} \mathsf{pos?}_{\mathsf{pre}}) \\ &(\mathsf{ifte}_{\mathsf{pre}}\ (\mathsf{snd}_{\mathsf{pre}} \ggg_{\mathsf{pre}} \mathsf{pos?}_{\mathsf{pre}}) \\ &\qquad \mathsf{mul}^{++}_{\mathsf{pre}} \\ &\qquad (\mathsf{ifte}_{\mathsf{pre}}\ (\mathsf{snd}_{\mathsf{pre}} \ggg_{\mathsf{pre}} \mathsf{neg?}_{\mathsf{pre}}) \\ &\qquad\qquad \mathsf{mul}^{+-}_{\mathsf{pre}} \\ &\qquad\qquad (\mathsf{const}_{\mathsf{pre}}\ 0))) \\ &\ldots\end{aligned} \tag{81}$$

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

Section 9.4 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 our main implementation 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 much more efficient than exact methods based on enumeration.

Let $\mathsf{g} : \mathsf{X} \overset{\rightsquigarrow}{\underset{\mathsf{map}}{}} \mathsf{Y}$ be the interpretation of a program as a mapping arrow computation. A Bayesian is primarily interested in the probability of $\mathsf{B}' \subseteq \mathsf{Y}$ given some condition set $\mathsf{B} \subseteq \mathsf{Y}$. If $\mathsf{A} := \mathsf{preimage}\ (\mathsf{g}\ \mathsf{X})\ \mathsf{B}$ and $\mathsf{A}' := \mathsf{preimage}\ (\mathsf{g}\ \mathsf{X})\ \mathsf{B}'$, the probability of B' given B is

$$\mathsf{Pr}[\mathsf{B}'|\mathsf{B}] \ :=\ \mathsf{P}\ (\mathsf{A}' \cap \mathsf{A}) / \mathsf{P}\ \mathsf{A} \tag{82}$$

This can be approximated using **rejection sampling**. Given a nonempty list of samples xs from any superset of A, the conditional probability in (82) is approximately

$$\mathsf{Pr}[\mathsf{B}'|\mathsf{B}] \ \approx\ \frac{\mathsf{length}\ (\mathsf{filter}\ (\in \mathsf{A}' \cap \mathsf{A})\ \mathsf{xs})}{\mathsf{length}\ (\mathsf{filter}\ (\in \mathsf{A})\ \mathsf{xs})} \tag{83}$$

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 x 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 (83), but we can use the fact that A and A' are preimages, and use the interpretation of the program as a bottom arrow computation $\mathsf{f} : \mathsf{X} \rightsquigarrow_{\perp} \mathsf{Y}$:

$$\begin{aligned}\mathsf{filter}\ (\in \mathsf{A})\ \mathsf{xs} \ &=\ \mathsf{filter}\ (\in \mathsf{preimage}\ (\mathsf{g}\ \mathsf{X})\ \mathsf{B})\ \mathsf{xs} \\ &=\ \mathsf{filter}\ (\lambda \mathsf{a}.\, \mathsf{g}\ \mathsf{X}\ \mathsf{a} \in \mathsf{B})\ \mathsf{xs} \\ &=\ \mathsf{filter}\ (\lambda \mathsf{a}.\, \mathsf{f}\ \mathsf{a} \in \mathsf{B})\ \mathsf{xs}\end{aligned} \tag{84}$$

Substituting into (83) gives

$$\mathsf{Pr}[\mathsf{B}'|\mathsf{B}] \ \approx\ \frac{\mathsf{length}\ (\mathsf{filter}\ (\lambda \mathsf{a}.\, \mathsf{f}\ \mathsf{a} \in \mathsf{B}' \cap \mathsf{B})\ \mathsf{xs})}{\mathsf{length}\ (\mathsf{filter}\ (\lambda \mathsf{a}.\, \mathsf{f}\ \mathsf{a} \in \mathsf{B})\ \mathsf{xs})} \tag{85}$$

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

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

XXX: specific problems: thermometer, stochastic ray tracing

11. 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 abstrac-

tion 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 [34, 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 lambda calculus with probabilistic choice, defining it for a very 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 compositionally 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.

12. Conclusions and Future Work

XXX: todo

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

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

relation to type systems

constraints

sampling algorithms

different abstract domains

References

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

[2] S. Abramsky. The lazy lambda calculus. In *Research Topics in Functional Programming*, pages 65–116. Addison-Wesley, 1990.

[3] R. J. Aumann. Borel structures for function spaces. *Illinois Journal of Mathematics*, 5:614–630, 1961.

[4] B. Barras. Sets in Coq, Coq in sets. *Journal of Formalized Reasoning*, 3(1), 2010.

[5] S. Bhat, J. Borgström, A. D. Gordon, and C. Russo. Deriving probability density functions from probabilistic functional programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2013.

[6] K. A. Bonawitz. *Composable Probabilistic Inference with Blaise*. PhD thesis, Massachusetts Institute of Technology, 2008.

[7] J. Borgström, A. D. Gordon, M. Greenberg, J. Margetson, and J. V. Gael. Measure transformer semantics for Bayesian machine learning. In *European Symposium on Programming*, 2011.

[8] M. M. T. Chakravarty, G. Keller, S. P. Jones, and S. Marlow. Associated types with class. In *Principles of Programming Languages*, pages 1–13, 2005.

[9] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages*, pages 238–252, 1977.

[10] M. DeGroot and M. Schervish. *Probability and Statistics*. Addison Wesley Publishing Company, Inc., 2012. ISBN 9780321500465.

[11] N. Goodman, V. Mansinghka, D. Roy, K. Bonawitz, and J. Tenenbaum. Church: a language for generative models. In *Uncertainty in Artificial Intelligence*, 2008.

[12] M. Hanus. Multi-paradigm declarative languages. In *Logic Programming*, pages 45–75. 2007.

[13] M. Hofmann, B. C. Pierce, , and D. Wagner. Edit lenses. In *Principles of Programming Languages*, 2012.

[14] J. Hughes. Generalizing monads to arrows. In *Science of Computer Programming*, volume 37, pages 67–111, 2000.

[15] J. Hurd. *Formal Verification of Probabilistic Algorithms*. PhD thesis, University of Cambridge, 2002.

[16] C. Jones. *Probabilistic Non-Determinism*. PhD thesis, University of Edinburgh, 1990.

[17] R. B. Kearfott. Interval computations: Introduction, uses, and resources. *Euromath Bulletin*, 2:95–112, 1996.

[18] O. Kiselyov and C. Shan. Monolingual probabilistic programming using generalized coroutines. In *Uncertainty in Artificial Intelligence*, 2008.

[19] A. Klenke. *Probability Theory: A Comprehensive Course*. Springer, 2006. ISBN 978-1-84800-047-6.

[20] D. Koller, D. McAllester, and A. Pfeffer. Effective Bayesian inference for stochastic programs. In *14th National Conference on Artificial Intelligence*, August 1997.

[21] D. Kozen. Semantics of probabilistic programs. In *Foundations of Computer Science*, 1979.

[22] S. Lindley, P. Wadler, and J. Yallop. Idioms are oblivious, arrows are meticulous, monads are promiscuous. *Electronic Notes in Theoretical Computer Science*, 2008.

[23] S. Lindley, P. Wadler, and J. Yallop. The arrow calculus. *Journal of Functional Programming*, 20:51–69, 2010.

[24] D. J. Lunn, A. Thomas, N. Best, and D. Spiegelhalter. WinBUGS – a Bayesian modelling framework. *Statistics and Computing*, 10(4), 2000.

[25] C. McBride and R. Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(1), 2008.

[26] B. Milch, B. Marthi, S. Russell, D. Sontag, D. Ong, and A. Kolobov. BLOG: Probabilistic models with unknown objects. In *International Joint Conference on Artificial Intelligence*, 2005.

[27] S. Park, F. Pfenning, and S. Thrun. A probabilistic language based upon sampling functions. *Transactions on Programming Languages and Systems*, 31(1), 2008.

[28] A. Pfeffer. The design and implementation of IBAL: A general-purpose probabilistic language. In *Statistical Relational Learning*. MIT Press, 2007.

[29] N. Ramsey and A. Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *Principles of Programming Languages*, 2002.

[30] S. Tobin-Hochstadt and M. Felleisen. The design and implementation of typed Scheme. In *Principles of Programming Languages*, 2008.

[31] N. Toronto and J. McCarthy. From Bayesian notation to pure Racket, via measure-theoretic probability in λ_{ZFC}. In *Implementation and Application of Functional Languages*, 2010.

[32] N. Toronto and J. McCarthy. Computing in Cantor's paradise with λ_{ZFC}. In *Functional and Logic Programming Symposium (FLOPS)*, pages 290–306, 2012.

[33] P. Wadler. Monads for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*. 2001.

[34] D. Wingate, N. D. Goodman, A. Stuhlmüller, and J. M. Siskind. Nonstandard interpretations of probabilistic programs for efficient inference. In *Neural Information Processing Systems*, 2011.

[35] D. Wingate, A. Stuhlmüller, and N. D. Goodman. Lightweight implementations of probabilistic programming languages via transformational compilation. In *Artificial Intelligence and Statistics*, 2011.