Running Probabilistic Programs Backward

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

Abstract

Problem: doesn't exist: probabilistic programming language that doesn't restrict legal programs, allows conditioning, and has a mathematical specification
Why problem:
Solution:
Why solution:

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

General Terms XXX, XXX

Keywords XXX, XXX

1. Introduction

There is currently no efficient probabilistic language implementation that simultaneously

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

In the field of programming languages, there are a few examples of languages that do not restrict legal programs and have a semantics (XXX: cite all). Unfortunately, most of the demand for probabilistic languages comes from Bayesian practice, which requires conditioning.

Bayesian practitioners have implemented many probabilistic languages with conditioning (XXX: cite all). Almost all lack a semantics, so it is impossible to distinguish between an implementation error and an opportunity to learn. Almost all place restrictions on programs, most commonly disallowing recursion, allowing only continuous distributions, and allowing only very limited forms of conditioning.

These common 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. Densities for random values with different dimension are incomparable. Densities cannot be defined on infinite products. Densities can only be used to reason about conditioning in limited cases.

Densities cannot define distributions of discontinuous functions of random variables. 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:

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

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

The restrictions placed on programs are indeed onerous, if such benign uses of min and max must be disallowed. We cannot even model measuring devices correctly.

1.1 Measure-Theoretic Probability

Measure-theoretic probability (XXX: cite) is widely believed to be able to define everything reasonable that densities cannot. It mainly does this by *assigning probabilities to sets* instead of *assigning changes in probability to values*.

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

$$\mathsf{P}\ (\mathsf{preimage}\ \mathsf{f}\ \mathsf{B}) \tag{2}$$

defines the distribution over subsets B of Y, where preimage returns 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 the set of the program's possible outputs. For any $\mathsf{B} \subseteq \mathsf{Y}$, $\mathsf{preimage}\ \mathsf{f}\ \mathsf{B}$ would be well-defined, regardless of discontinuities.

Unfortunately, there is a complicated technical restriction: only *measurable* subsets of X and Y can be assigned probabilities. Conditioning on zero-probability sets can also be quite difficult. These complexities tend to drive practitioners to densities, even though they are so limited.

1.2 Measure-Theoretic Semantics

Because purely functional languages do not allow effects (except usually divergence), programmers must write probabilistic programs as functions from a random source to outputs. Monads and other categorical classes such as idioms and arrows can make doing so easier. (XXX: cite me, bunch of others)

It seems this approach should make it easy to interpret probabilistic programs measure-theoretically. For a probabilistic program $\mathsf{f} : \mathsf{X} \to \mathsf{Y}$, the probability measure on output sets $\mathsf{B} \subseteq \mathsf{Y}$ 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 are defined for extensional functions.

2. Requiring subsets of X and Y to be measurable constrains f: preimages of measurable subsets of Y must be measurable subsets of X. Proving the conditions under which this is true is difficult, especially when f may diverge.
3. Probability measures cannot be defined for arbitrary function spaces (XXX: cite).

Implementing a language based on such a semantics is complicated by these facts:

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} [2], 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 which inputs they diverge on. 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 is a perfect fit for the "first-orderness" of measure theory.

1.3 Arrow Solution Overview

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

- A semantic function which, like the semantic function for the arrow calculus (XXX: cite Hughes or Lindley), transforms first-order programs into the computations of an arbitrary arrow.
- Arrows for running programs in different ways.

This commutative diagram describes the relationships among the arrows used to define the exact semantics:

$$\begin{array}{ccccc} \mathsf{X} \rightsquigarrow_{\perp} \mathsf{Y} & \xrightarrow{\mathsf{lift}_{\mathsf{map}}} & \mathsf{X} \overset{\rightsquigarrow}{\underset{\mathsf{map}}{}} \mathsf{Y} & \xrightarrow{\mathsf{lift}_{\mathsf{pre}}} & \mathsf{X} \overset{\rightsquigarrow}{\underset{\mathsf{pre}}{}} \mathsf{Y} \\ {\scriptstyle \eta_{\perp^*}}\downarrow & & \downarrow{\scriptstyle \eta_{\mathsf{map}^*}} & & \downarrow{\scriptstyle \eta_{\mathsf{pre}^*}} \\ \mathsf{X} \rightsquigarrow_{\perp^*} \mathsf{Y} & \xrightarrow[\mathsf{lift}_{\mathsf{map}^*}]{} & \mathsf{X} \overset{\rightsquigarrow}{\underset{\mathsf{map}^*}{}} \mathsf{Y} & \xrightarrow[\mathsf{lift}_{\mathsf{pre}^*}]{} & \mathsf{X} \overset{\rightsquigarrow}{\underset{\mathsf{pre}^*}{}} \mathsf{Y} \end{array} \tag{3}$$

From top-left to top-right, $\mathsf{X} \rightsquigarrow_{\perp} \mathsf{Y}$ computations are intensional functions that may raise errors, $\mathsf{X} \overset{\rightsquigarrow}{\underset{\mathsf{map}}{}} \mathsf{Y}$ computations produce extensional functions, and $\mathsf{X} \overset{\rightsquigarrow}{\underset{\mathsf{pre}}{}} \mathsf{Y}$ computations compute preimages. The computations of the arrows in the bottom row are equivalent to those in the top, except they always converge. We can do this because in λ_{ZFC}, Turing-uncomputable programs are definable.

Our approximating semantics consists of the same semantic function and an arrow $\mathsf{X} \overset{\rightsquigarrow}{\underset{\mathsf{pre}^*}{}}{}' \mathsf{Y}$, derived from $\mathsf{X} \overset{\rightsquigarrow}{\underset{\mathsf{pre}^*}{}} \mathsf{Y}$, for computing conservative approximations of preimages.

An implementation implements the semantic function, and the $\mathsf{X} \rightsquigarrow_{\perp} \mathsf{Y}$ and $\mathsf{X} \overset{\rightsquigarrow}{\underset{\mathsf{pre}^*}{}}{}' \mathsf{Y}$ arrows' combinators.

Most of our correctness theorems rely on proofs that every lift and η in (**??**) is a homomorphism. We use lift and η to define the correctness of one arrow in terms of another arrow. Homomorphism properties allow lift and η to distribute over the other arrow's computations.

From here on, significant terms are introduced in **bold**, with those we invent introduced in ***bold italics***.

2. Operational Metalanguage

We write all of the programs in this paper in λ_{ZFC} [2], an untyped, call-by-value lambda 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 a precise mathematical specification and deriving an approximating specification 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 converge.[1]

Almost everything definable in ZFC can be formally defined by a finite λ_{ZFC} program, except objects that most mathematicians would agree are nonconstructive. More precisely, any set that *must* be defined by a statement of existence and uniqueness without giving a bounding set is not definable by a *finite* λ_{ZFC} program.

Because λ_{ZFC} includes an inner model of ZFC, essentially every ZFC theorem applies to λ_{ZFC}'s set values without alteration. Further, proofs about λ_{ZFC}'s set values apply to ZFC sets.[2]

In λ_{ZFC}, algebraic data structures are encoded as sets; e.g. a ***primitive ordered pair*** of x and y is $\{\{\mathsf{x}\}, \{\mathsf{x}, \mathsf{y}\}\}$. 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 lowercase type variable is universally quantified.
- A free uppercase type variable is 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.

The type $\mathsf{Set}\ \mathsf{X}$ denotes the same values as the powerset $\mathcal{P}\ \mathsf{X}$, or *subsets* of X. Similarly, the type $\langle \mathsf{X}, \mathsf{Y} \rangle$ denotes the same values as the product set $\mathsf{X} \times \mathsf{Y}$.

[1] An example of a diverging λ_{ZFC} function is one that attempts to decide whether arbitrary λ_{ZFC} expressions converge.

[2] Assuming the existence of an inaccessible cardinal.

We write λ_{ZFC} programs in heavily sugared λ-calculus syntax, with an if expression and these additional primitives:

$$\begin{aligned} \mathsf{true} &: \mathsf{Bool} & (\in) &: \mathsf{x} \Rightarrow \mathsf{Set\ x} \Rightarrow \mathsf{Bool} \\ \mathsf{false} &: \mathsf{Bool} & \mathcal{P} &: \mathsf{Set\ x} \Rightarrow \mathsf{Set\ (Set\ x)} \\ \varnothing &: \mathsf{Set\ x} & \bigcup &: \mathsf{Set\ (Set\ x)} \Rightarrow \mathsf{Set\ x} \\ \omega &: \mathsf{Ord} & \mathsf{image} &: (\mathsf{x} \Rightarrow \mathsf{y}) \Rightarrow \mathsf{Set\ x} \Rightarrow \mathsf{Set\ y} \\ \mathsf{take} &: \mathsf{Set\ x} \Rightarrow \mathsf{x} & |\cdot| &: \mathsf{Set\ x} \Rightarrow \mathsf{Ord} \end{aligned} \tag{4}$$

Shortly, $\varnothing$ is the empty set, ω is the cardinality of the natural numbers, $\mathsf{take}\ \{\mathsf{x}\}$ reduces to x and diverges for non-singleton sets, $\mathsf{x} \in \mathsf{A}$ decides membership, $\mathcal{P}\ \mathsf{A}$ reduces to the set of subsets of A, $\bigcup \mathcal{A}$ reduces to the union of the sets in $\mathcal{A}$, $\mathsf{image\ f\ A}$ applies f to each member of A and reduces to the set of results, and $|\mathsf{A}|$ reduces to the cardinality of A.

We assume literal set notation such as $\{0, 1, 2\}$ is already defined in terms of the set primitives.

We import applicable ZFC theorems as lemmas.

2.1 Internal and External Equality

Set theory extends first-order logic with an axiom that defines equality to be extensional, and with axioms that ensure the existence of sets in the domain of discourse. λ_{ZFC} is defined the same way as any other operational λ-calculus: by (conservatively) extending the domain of discourse with expressions and defining a reduction relation.

While λ_{ZFC} does not have an equality primitive, set theory's extensional equality can be recovered internally using $(\in)$. *Internal* extensional equality is defined by either of the following equivalent statements:

$$\begin{aligned} \mathsf{x} = \mathsf{y} &:= \mathsf{x} \in \{\mathsf{y}\} \\ (=) &:= \lambda \mathsf{x}.\, \lambda \mathsf{y}.\, \mathsf{x} \in \{\mathsf{y}\} \end{aligned} \tag{5}$$

Thus, $1 = 1$ reduces to $1 \in \{1\}$, which reduces to true.[3] Because of the particular way λ_{ZFC}'s lambda terms are defined, for two lambda terms f and g, $\mathsf{f} = \mathsf{g}$ reduces to true when f and g are structurally identical modulo renaming. For example, $(\lambda \mathsf{x}.\, \mathsf{x}) = (\lambda \mathsf{y}.\, \mathsf{y})$ reduces to true, but $(\lambda \mathsf{x}.\, 2) = (\lambda \mathsf{x}.\, 1 + 1)$ reduces to false.

We understand any λ_{ZFC} term e used as a truth statement as shorthand for "e reduces to true." Therefore, while the terms $(\lambda \mathsf{x}.\, \mathsf{x})\ 1$ and 1 are (externally, extensionally) unequal, we can say that $(\lambda \mathsf{x}.\, \mathsf{x})\ 1 = 1$.

Any truth statement e implies that e converges. In particular, the truth statement $e_1 = e_2$ implies that both e_1 and e_2 converge. However, we often want to say that e_1 and e_1 are equivalent when they both diverge. In these cases, we use a slightly weaker equivalence.

Definition 2.1 (observational equivalence)**.** *Two λ_{ZFC} terms e_1 and e_2 are **observationally equivalent**, written $e_1 \equiv e_2$, when $e_1 = e_2$ or both e_1 and e_2 diverge.*

It might seem helpful to introduce even coarser notions of equivalence, such as applicative or logical bisimilarity. 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 (including for the two-argument primitives $(\in)$ and image), 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:

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

$\mathsf{A} \uplus \mathsf{B}$ diverges when A and B overlap.

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 extensional functions ***mappings*** and intensional functions **lambdas**, and use the word **function** to mean either. For convenience, as with lambdas, we use adjacency (e.g. $(\mathsf{f\ x})$) to apply mappings.

Syntax for unnamed mappings is defined by

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

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

Figure 1 defines other common operations on mappings: domain, range, $\mathsf{preimage}$, $\mathsf{restrict}$, pairing, composition, and disjoint union. The latter three are particularly important in the preimage arrow's derivation, and $\mathsf{preimage}$ is critical in measure theory's account of probability. For symmetry with partial functions $\mathsf{x} \Rightarrow \mathsf{y}$, they are defined on $\mathsf{X} \rightharpoonup \mathsf{Y}$, where $\mathsf{X} \rightharpoonup \mathsf{Y}$ is the set of all partial mappings from any domain set X to any codomain set Y.

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

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

which is particularly useful when f is unnamed.

3. Arrows and First-Order Semantics

XXX: really short arrow intro (XXX: cite Hughes, Lindley et al)

3.1 Alternative Arrow Definitions and Laws

We do not give typical minimal arrow definitions. For each arrow a, instead of $\mathsf{first_a}$, 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 $\mathsf{first_a}$ may be defined in terms of $(\mathsf{\&\&\&_a})$ and vice-versa [1], 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 $\mathsf{left_a}$, which can be used to choose an arrow computation based on the result of another. Again, we define a different combinator, $\mathsf{ifte_a}$, to make it easier to apply measure-theoretic theorems.

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.

[3] Technically, λ_{ZFC} has a big-step semantics, and the derivation tree for $1 = 1$ contains the derivation tree for $1 \in \{1\}$.

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

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

Figure 1: Operations on mappings.

For example, define the **function arrow** with choice:

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

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 diverges in a strict λ-calculus. In a nonstrict λ-calculus, it diverges 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 $\mathsf{halt\text{-}on\text{-}true}$ as

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

which diverges 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*

$$
\begin{aligned}
\mathsf{arr}_{\mathsf{a}} &: (\mathsf{x} \Rightarrow \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y})\\
(\ggg_{\mathsf{a}}) &: (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \Rightarrow (\mathsf{y} \rightsquigarrow_{\mathsf{a}} \mathsf{z}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{z})\\
(\mathbin{\&\&\&}_{\mathsf{a}}) &: (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{z}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \langle \mathsf{y}, \mathsf{z} \rangle)
\end{aligned}
\tag{13}
$$

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

$$
\begin{aligned}
&\mathsf{ifte}_{\mathsf{a}} : (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{Bool}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y})\\
&\mathsf{lazy}_{\mathsf{a}} : (1 \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y})) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y})
\end{aligned}
\tag{14}
$$

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 ensure that $\mathsf{arr}_{\mathsf{a}}$ distributes over function arrow combinators. These 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*:*

$$
\begin{aligned}
\mathsf{lift}_{\mathsf{b}}\ (\mathsf{arr}_{\mathsf{a}}\ \mathsf{f}) &\equiv \mathsf{arr}_{\mathsf{b}}\ \mathsf{f} && (15)\\
\mathsf{lift}_{\mathsf{b}}\ (\mathsf{f}_1 \ggg_{\mathsf{a}} \mathsf{f}_2) &\equiv (\mathsf{lift}_{\mathsf{b}}\ \mathsf{f}_1) \ggg_{\mathsf{b}} (\mathsf{lift}_{\mathsf{b}}\ \mathsf{f}_2) && (16)\\
\mathsf{lift}_{\mathsf{b}}\ (\mathsf{f}_1 \mathbin{\&\&\&}_{\mathsf{a}} \mathsf{f}_2) &\equiv (\mathsf{lift}_{\mathsf{b}}\ \mathsf{f}_1) \mathbin{\&\&\&}_{\mathsf{b}} (\mathsf{lift}_{\mathsf{b}}\ \mathsf{f}_2) && (17)\\
\mathsf{lift}_{\mathsf{b}}\ (\mathsf{ifte}_{\mathsf{a}}\ \mathsf{f}_1\ \mathsf{f}_2\ \mathsf{f}_3) &\equiv \mathsf{ifte}_{\mathsf{b}}\ (\mathsf{lift}_{\mathsf{b}}\ \mathsf{f}_1)\ (\mathsf{lift}_{\mathsf{b}}\ \mathsf{f}_2)\ (\mathsf{lift}_{\mathsf{b}}\ \mathsf{f}_3) && (18)\\
\mathsf{lift}_{\mathsf{b}}\ (\mathsf{lazy}_{\mathsf{a}}\ \mathsf{f}) &\equiv \mathsf{lazy}_{\mathsf{b}}\ \lambda 0.\, \mathsf{lift}_{\mathsf{b}}\ (\mathsf{f}\ 0) && (19)
\end{aligned}
$$

The homomorphism laws state that $\mathsf{arr}_{\mathsf{a}}$ must be a homomorphism from the function arrow (10) to arrow a. Roughly, arrow computations that do not use additional combinators can be transformed into $\mathsf{arr}_{\mathsf{a}}$ applied to a pure computation.

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

$$
(\mathsf{f}_1 \ggg_{\mathsf{a}} \mathsf{f}_2) \ggg_{\mathsf{a}} \mathsf{f}_3 \ \equiv\ \mathsf{f}_1 \ggg_{\mathsf{a}} (\mathsf{f}_2 \ggg_{\mathsf{a}} \mathsf{f}_3) \tag{20}
$$

a pair extraction law:

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

and distribution of pure computations over effectful:

$$
\begin{aligned}
\mathsf{arr}_{\mathsf{a}}\ \mathsf{f}_1 \ggg_{\mathsf{a}} (\mathsf{f}_2 \mathbin{\&\&\&}_{\mathsf{a}} \mathsf{f}_3) &\equiv (\mathsf{arr}_{\mathsf{a}}\ \mathsf{f}_1 \ggg_{\mathsf{a}} \mathsf{f}_2) \mathbin{\&\&\&}_{\mathsf{a}} (\mathsf{arr}_{\mathsf{a}}\ \mathsf{f}_1 \ggg_{\mathsf{a}} \mathsf{f}_3) && (22)\\
\mathsf{arr}_{\mathsf{a}}\ \mathsf{f}_1 \ggg_{\mathsf{a}} \mathsf{ifte}_{\mathsf{a}}\ \mathsf{f}_2\ \mathsf{f}_3\ \mathsf{f}_4 &\equiv \mathsf{ifte}_{\mathsf{a}}\ (\mathsf{arr}_{\mathsf{a}}\ \mathsf{f}_1 \ggg_{\mathsf{a}} \mathsf{f}_2)\ (\mathsf{arr}_{\mathsf{a}}\ \mathsf{f}_1 \ggg_{\mathsf{a}} \mathsf{f}_3)\ (\mathsf{arr}_{\mathsf{a}}\ \mathsf{f}_1 \ggg_{\mathsf{a}} \mathsf{f}_4) && (23)\\
\mathsf{arr}_{\mathsf{a}}\ \mathsf{f}_1 \ggg_{\mathsf{a}} \mathsf{lazy}_{\mathsf{a}}\ \mathsf{f}_2 &\equiv \mathsf{lazy}_{\mathsf{a}}\ \lambda 0.\, \mathsf{arr}_{\mathsf{a}}\ \mathsf{f}_1 \ggg_{\mathsf{a}} \mathsf{f}_2\ 0 && (24)
\end{aligned}
$$

Because arrows have traditionally been defined in nonstrict and strongly normalizing λ-calculii, we could not derive (24) from existing arrow laws. We are sure it is reasonable.

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

Definition 3.3 (arrow epimorphism)**.** *An arrow homomorphism* $\mathsf{lift}_{\mathsf{b}} : (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{b}} \mathsf{y})$ *is an* ***arrow epimorphism*** *from arrow* a *to* b *if it has a right inverse.*

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

Proof. For the pair extraction law (21), rewrite in terms of $\mathsf{lift}_{\mathsf{b}}$, apply homomorphism laws, and apply the pair extrac-

tion law for arrow a:

$$\begin{aligned}
&(\mathsf{arr_b}\ \mathsf{f_1}\ \mathbin{\&\&\&_b}\ \mathsf{f_2}) \ggg_\mathsf{b} \mathsf{arr_b}\ \mathsf{snd} \\
&\quad\equiv (\mathsf{lift_b}\ (\mathsf{arr_a}\ \mathsf{f_1})\ \mathbin{\&\&\&_b}\ (\mathsf{lift_b}\ (\mathsf{lift_b^{-1}}\ \mathsf{f_2}))) \ggg_\mathsf{b} \mathsf{arr_b}\ \mathsf{snd} \\
&\quad\equiv \mathsf{lift_b}\ (\mathsf{arr_a}\ \mathsf{f_1}\ \mathbin{\&\&\&_a}\ \mathsf{lift_b^{-1}}\ \mathsf{f_2}) \ggg_\mathsf{b} \mathsf{lift_b}\ (\mathsf{arr_a}\ \mathsf{snd}) \\
&\quad\equiv \mathsf{lift_b}\ ((\mathsf{arr_a}\ \mathsf{f_1}\ \mathbin{\&\&\&_a}\ \mathsf{lift_b^{-1}}\ \mathsf{f_2}) \ggg_\mathsf{b} \mathsf{arr_a}\ \mathsf{snd}) \\
&\quad\equiv \mathsf{lift_b}\ (\mathsf{lift_b^{-1}}\ \mathsf{f_2}) \equiv \mathsf{f_2}
\end{aligned}$$

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 $\mathsf{0}$ 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. Application sends a stack containing just x.

It is not difficult to allow named bindings, but it is better to do so in a separate semantic function. Baking such support into $[\![\cdot]\!]_\mathsf{a}$ might complicate the simple proof of the following theorem, which underlies most of our semantic correctness claims.

Theorem 3.5 (homomorphisms distribute over programs)**.** *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 programs* e, $[\![e]\!]_\mathsf{b} \equiv \mathsf{lift_b}\ [\![e]\!]_\mathsf{a}$.

Proof. By structural induction on program terms.

Bases cases proceed by expansion and using $\mathsf{lift_b} \circ \mathsf{arr_a} \equiv \mathsf{arr_b}$ (15). For example, for constants:

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

Inductive cases proceed by expansion, applying one or more distributive laws (16–19), and applying the inductive hypothesis on subterms. For example, for pairing:

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

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 3.5, $[\![\cdot]\!]_\mathsf{b}$ is correct.

4. The Bottom and Mapping Arrows

Using the diagram in (3) as a sort of map, we are starting in the upper-left corner:

$$\begin{array}{ccccc}
\mathsf{X} \rightsquigarrow_\perp \mathsf{Y} & \xrightarrow{\mathsf{lift_{map}}} & \mathsf{X} \overset{\rightsquigarrow}{\underset{\mathsf{map}}{}} \mathsf{Y} & \xrightarrow{\mathsf{lift_{pre}}} & \mathsf{X} \overset{\rightsquigarrow}{\underset{\mathsf{pre}}{}} \mathsf{Y} \\
\Big\downarrow{\scriptstyle \eta_{\perp^*}} & & \Big\downarrow{\scriptstyle \eta_{\mathsf{map}^*}} & & \Big\downarrow{\scriptstyle \eta_{\mathsf{pre}^*}} \\
\mathsf{X} \rightsquigarrow_{\perp^*} \mathsf{Y} & \xrightarrow[\mathsf{lift_{map^*}}]{} & \mathsf{X} \overset{\rightsquigarrow}{\underset{\mathsf{map}^*}{}} \mathsf{Y} & \xrightarrow[\mathsf{lift_{pre^*}}]{} & \mathsf{X} \overset{\rightsquigarrow}{\underset{\mathsf{pre}^*}{}} \mathsf{Y}
\end{array} \tag{25}$$

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

To use Theorem 3.5 to prove correct the interpretations of programs as preimage arrow computations, we need to define a simpler arrow whose behavior is easily understood. One obvious candidate is the function arrow (10). However, we will need to explicitly handle divergence 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.

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 the Maybe monad's Kleisli arrow with $\mathsf{Nothing} = \perp$. The proof of (24) (pure computations distribute over $\mathsf{lazy}_\perp$) is trivial. □

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 $\mathsf{arr_{map}} : (\mathsf{X} \Rightarrow \mathsf{Y}) \Rightarrow (\mathsf{X} \rightharpoonup \mathsf{Y})$: 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

$$\mathsf{X} \overset{\rightsquigarrow}{\underset{\mathsf{map}}{}} \mathsf{Y} ::= \mathsf{Set}\ \mathsf{X} \Rightarrow (\mathsf{X} \rightharpoonup \mathsf{Y}) \tag{26}$$

The fact that $\perp$ is absent from Y in $\mathsf{Set}\ \mathsf{X} \Rightarrow (\mathsf{X} \rightharpoonup \mathsf{Y})$ will make it easier to exclude diverging inputs. Its absence and the fact that the type parameters 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 3.5 to prove that programs 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

$$
\begin{array}{rcl@{\qquad}rcl}
[\![x := e;\ \cdots\ ; e_{body}]\!]_{\mathsf{a}} & :\equiv & x := [\![e]\!]_{\mathsf{a}};\ \cdots\ ; [\![e_{body}]\!]_{\mathsf{a}} & [\![v]\!]_{\mathsf{a}} & :\equiv & \mathsf{arr_a}\ (\mathsf{const}\ v) \\
[\![x\ e]\!]_{\mathsf{a}} & :\equiv & [\![\langle e, \langle\rangle\rangle]\!]_{\mathsf{a}} \ggg_{\mathsf{a}} x & [\![\langle e_1, e_2\rangle]\!]_{\mathsf{a}} & :\equiv & [\![e_1]\!]_{\mathsf{a}} \mathbin{\&\&\&_{\mathsf{a}}} [\![e_2]\!]_{\mathsf{a}} \\
[\![\mathsf{let}\ e\ e_{body}]\!]_{\mathsf{a}} & :\equiv & ([\![e]\!]_{\mathsf{a}} \mathbin{\&\&\&_{\mathsf{a}}} \mathsf{arr_a}\ \mathsf{id}) \ggg_{\mathsf{a}} [\![e_{body}]\!]_{\mathsf{a}} & [\![\mathsf{fst}\ e]\!]_{\mathsf{a}} & :\equiv & [\![e]\!]_{\mathsf{a}} \ggg_{\mathsf{a}} \mathsf{arr_a}\ \mathsf{fst} \\
[\![\mathsf{env}\ 0]\!]_{\mathsf{a}} & :\equiv & \mathsf{arr_a}\ \mathsf{fst} & [\![\mathsf{snd}\ e]\!]_{\mathsf{a}} & :\equiv & [\![e]\!]_{\mathsf{a}} \ggg_{\mathsf{a}} \mathsf{arr_a}\ \mathsf{snd} \\
[\![\mathsf{env}\ (n+1)]\!]_{\mathsf{a}} & :\equiv & \mathsf{arr_a}\ \mathsf{snd} \ggg_{\mathsf{a}} [\![\mathsf{env}\ n]\!]_{\mathsf{a}} & [\![\mathsf{if}\ e_c\ e_t\ e_f]\!]_{\mathsf{a}} & :\equiv & \mathsf{ifte_a}\ [\![e_c]\!]_{\mathsf{a}}\ (\mathsf{lazy_a}\ \lambda 0.\, [\![e_t]\!]_{\mathsf{a}})\ (\mathsf{lazy_a}\ \lambda 0.\, [\![e_f]\!]_{\mathsf{a}})
\end{array}
$$

where $\mathsf{const\ b} := \lambda \mathsf{a}.\, \mathsf{b}$ and $\mathsf{id} := \lambda \mathsf{a}.\, \mathsf{a}$

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

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

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

Figure 3: Bottom arrow definitions.

$$
\begin{array}{l}
\mathsf{X} \mathbin{\rightsquigarrow_{\mathsf{map}}} \mathsf{Y} \ ::= \ \mathsf{Set\ X} \Rightarrow (\mathsf{X} \rightharpoonup \mathsf{Y}) \\
\\
\mathsf{arr}_{\mathsf{map}} : (\mathsf{X} \Rightarrow \mathsf{Y}) \Rightarrow (\mathsf{X} \mathbin{\rightsquigarrow_{\mathsf{map}}} \mathsf{Y}) \\
\mathsf{arr}_{\mathsf{map}} \ := \ \mathsf{lift}_{\mathsf{map}} \circ \mathsf{arr}_{\perp} \\
\\
(\ggg_{\mathsf{map}}) : (\mathsf{X} \mathbin{\rightsquigarrow_{\mathsf{map}}} \mathsf{Y}) \Rightarrow (\mathsf{Y} \mathbin{\rightsquigarrow_{\mathsf{map}}} \mathsf{Z}) \Rightarrow (\mathsf{X} \mathbin{\rightsquigarrow_{\mathsf{map}}} \mathsf{Z}) \\
(\mathsf{g}_1 \ggg_{\mathsf{map}} \mathsf{g}_2)\ \mathsf{A} \ := \ \mathsf{let}\ \ \mathsf{g}_1' := \mathsf{g}_1\ \mathsf{A} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{g}_2' := \mathsf{g}_2\ (\mathsf{range}\ \mathsf{g}_1') \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{in}\ \ \mathsf{g}_2' \circ_{\mathsf{map}} \mathsf{g}_1' \\
\\
(\mathbin{\&\&\&_{\mathsf{map}}}) : (\mathsf{X} \mathbin{\rightsquigarrow_{\mathsf{map}}} \mathsf{Y}_1) \Rightarrow (\mathsf{X} \mathbin{\rightsquigarrow_{\mathsf{map}}} \mathsf{Y}_2) \Rightarrow (\mathsf{X} \mathbin{\rightsquigarrow_{\mathsf{map}}} \langle \mathsf{Y}_1, \mathsf{Y}_2 \rangle) \\
(\mathsf{g}_1 \mathbin{\&\&\&_{\mathsf{map}}} \mathsf{g}_2)\ \mathsf{A} \ := \ \langle \mathsf{g}_1\ \mathsf{A}, \mathsf{g}_2\ \mathsf{A} \rangle_{\mathsf{map}}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{ifte}_{\mathsf{map}} : (\mathsf{X} \mathbin{\rightsquigarrow_{\mathsf{map}}} \mathsf{Bool}) \Rightarrow (\mathsf{X} \mathbin{\rightsquigarrow_{\mathsf{map}}} \mathsf{Y}) \Rightarrow (\mathsf{X} \mathbin{\rightsquigarrow_{\mathsf{map}}} \mathsf{Y}) \Rightarrow (\mathsf{X} \mathbin{\rightsquigarrow_{\mathsf{map}}} \mathsf{Y}) \\
\mathsf{ifte}_{\mathsf{map}}\ \mathsf{g}_1\ \mathsf{g}_2\ \mathsf{g}_3\ \mathsf{A} \ := \ \mathsf{let}\ \ \mathsf{g}_1' := \mathsf{g}_1\ \mathsf{A} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{g}_2' := \mathsf{g}_2\ (\mathsf{preimage}\ \mathsf{g}_1'\ \{\mathsf{true}\}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{g}_3' := \mathsf{g}_3\ (\mathsf{preimage}\ \mathsf{g}_1'\ \{\mathsf{false}\}) \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{in}\ \ \mathsf{g}_2' \uplus_{\mathsf{map}} \mathsf{g}_3' \\
\\
\mathsf{lazy}_{\mathsf{map}} : (1 \Rightarrow (\mathsf{X} \mathbin{\rightsquigarrow_{\mathsf{map}}} \mathsf{Y})) \Rightarrow (\mathsf{X} \mathbin{\rightsquigarrow_{\mathsf{map}}} \mathsf{Y}) \\
\mathsf{lazy}_{\mathsf{map}}\ \mathsf{g}\ \mathsf{A} \ := \ \mathsf{if}\ (\mathsf{A} = \varnothing)\ \varnothing\ (\mathsf{g}\ 0\ \mathsf{A}) \\
\\
\hline \\
\mathsf{lift}_{\mathsf{map}} : (\mathsf{X} \rightsquigarrow_{\perp} \mathsf{Y}) \Rightarrow (\mathsf{X} \mathbin{\rightsquigarrow_{\mathsf{map}}} \mathsf{Y}) \\
\mathsf{lift}_{\mathsf{map}}\ \mathsf{f}\ \mathsf{A} \ := \ \{\langle \mathsf{a}, \mathsf{b} \rangle \in \mathsf{mapping\ f\ A} \mid \mathsf{b} \neq \perp\}
\end{array}
$$

Figure 4: Mapping arrow definitions.

define that first, and then define $\mathsf{lift}_{\mathsf{map}}$ in terms of it:

$$
\begin{array}{l}
\mathsf{domain}_{\perp} : (\mathsf{X} \rightsquigarrow_{\perp} \mathsf{Y}) \Rightarrow \mathsf{Set\ X} \Rightarrow \mathsf{Set\ X} \\
\mathsf{domain}_{\perp}\ \mathsf{f}\ \mathsf{A} \ := \ \{\mathsf{a} \in \mathsf{A} \mid \mathsf{f\ a} \neq \perp\}
\end{array} \tag{27}
$$

$$
\begin{array}{l}
\mathsf{lift}_{\mathsf{map}} : (\mathsf{X} \rightsquigarrow_{\perp} \mathsf{Y}) \Rightarrow (\mathsf{X} \mathbin{\rightsquigarrow_{\mathsf{map}}} \mathsf{Y}) \\
\mathsf{lift}_{\mathsf{map}}\ \mathsf{f}\ \mathsf{A} \ := \ \mathsf{mapping\ f\ (domain}_{\perp}\ \mathsf{f\ A)}
\end{array} \tag{28}
$$

So $\mathsf{lift}_{\mathsf{map}}\ \mathsf{f}\ \mathsf{A}$ is like $\mathsf{mapping\ f\ A}$, 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* $\mathsf{g}_1 : \mathsf{X} \mathbin{\rightsquigarrow_{\mathsf{map}}} \mathsf{Y}$ *and* $\mathsf{g}_2 : \mathsf{X} \mathbin{\rightsquigarrow_{\mathsf{map}}} \mathsf{Y}$ *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 (15). The following subsections derive $(\mathbin{\&\&\&_{\mathsf{map}}})$, $(\ggg_{\mathsf{map}})$, $\mathsf{ifte}_{\mathsf{map}}$ and $\mathsf{lazy}_{\mathsf{map}}$ from bottom arrow combinators, in a way that ensures $\mathsf{lift}_{\mathsf{map}}$ is an arrow homomorphism. Figure 4 contains the resulting definitions.

4.1.1 Case: Pairing

Starting with the left side of (17), we first expand definitions. For any $\mathsf{f}_1 : \mathsf{X} \rightsquigarrow_\perp \mathsf{Y}$, $\mathsf{f}_2 : \mathsf{X} \rightsquigarrow_\perp \mathsf{Z}$, and $\mathsf{A} \subseteq \mathsf{X}$,

$$\begin{aligned}
&\mathsf{lift}_{\mathsf{map}}\ (\mathsf{f}_1\ \&\&\&_\perp\ \mathsf{f}_2)\ \mathsf{A} \\
&\quad\equiv\ \mathsf{lift}_{\mathsf{map}}\ (\lambda\mathsf{a}.\,\mathsf{if}\ (\mathsf{f}_1\ \mathsf{a} = \perp\ \mathsf{or}\ \mathsf{f}_2\ \mathsf{a} = \perp)\ \perp\ \langle \mathsf{f}_1\ \mathsf{a}, \mathsf{f}_2\ \mathsf{a}\rangle)\ \mathsf{A} \\
&\quad\equiv\ \mathsf{let}\ \ \mathsf{f} := \lambda\mathsf{a}.\,\mathsf{if}\ (\mathsf{f}_1\ \mathsf{a} = \perp\ \mathsf{or}\ \mathsf{f}_2\ \mathsf{a} = \perp)\ \perp\ \langle \mathsf{f}_1\ \mathsf{a}, \mathsf{f}_2\ \mathsf{a}\rangle \\
&\qquad\ \mathsf{in}\ \ \mathsf{mapping}\ \mathsf{f}\ (\mathsf{domain}_\perp\ \mathsf{f}\ \mathsf{A})
\end{aligned} \tag{29}$$

Next, we replace the definition of A' with one that does not depend on f, and rewrite in terms of $\mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_1$ and $\mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_2$:

$$\begin{aligned}
&\mathsf{lift}_{\mathsf{map}}\ (\mathsf{f}_1\ \&\&\&_\perp\ \mathsf{f}_2)\ \mathsf{A} \\
&\quad\equiv\ \mathsf{let}\ \ \mathsf{A}_1 := (\mathsf{domain}_\perp\ \mathsf{f}_1\ \mathsf{A}) \\
&\qquad\qquad\ \mathsf{A}_2 := (\mathsf{domain}_\perp\ \mathsf{f}_2\ \mathsf{A}) \\
&\qquad\qquad\ \mathsf{A}' := \mathsf{A}_1 \cap \mathsf{A}_2 \\
&\qquad\ \mathsf{in}\ \ \lambda\mathsf{a} \in \mathsf{A}'.\,\langle \mathsf{f}_1\ \mathsf{a}, \mathsf{f}_2\ \mathsf{a}\rangle \\
&\quad\equiv\ \mathsf{let}\ \ \mathsf{g}_1 := \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_1\ \mathsf{A} \\
&\qquad\qquad\ \mathsf{g}_2 := \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_2\ \mathsf{A} \\
&\qquad\qquad\ \mathsf{A}' := (\mathsf{domain}\ \mathsf{g}_1) \cap (\mathsf{domain}\ \mathsf{g}_2) \\
&\qquad\ \mathsf{in}\ \ \lambda\mathsf{a} \in \mathsf{A}'.\,\langle \mathsf{g}_1\ \mathsf{a}, \mathsf{g}_2\ \mathsf{a}\rangle \\
&\quad\equiv\ \langle \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_1\ \mathsf{A}, \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_2\ \mathsf{A}\rangle_{\mathsf{map}}
\end{aligned} \tag{30}$$

Substituting g_1 for $\mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_1$ and g_2 for $\mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_2$ gives a definition for $(\&\&\&_{\mathsf{map}})$ (Figure 4) for which (17) holds.

4.1.2 Case: Composition

The derivation of $(\ggg_{\mathsf{map}})$ is similar to that of $(\&\&\&_{\mathsf{map}})$ but a little more involved.

XXX: add this, maybe cut later

4.1.3 Case: Conditional

Starting with the left side of (18), we expand definitions, and simplify f by restricting it to a domain for which $\mathsf{f}_1\ \mathsf{a} \neq \perp$:

$$\begin{aligned}
&\mathsf{lift}_{\mathsf{map}}\ (\mathsf{ifte}_\perp\ \mathsf{f}_1\ \mathsf{f}_2\ \mathsf{f}_3)\ \mathsf{A} \\
&\quad\equiv\ \mathsf{let}\ \ \mathsf{f} := \lambda\mathsf{a}.\,\mathsf{case}\ \ \mathsf{f}_1\ \mathsf{a} \\
&\qquad\qquad\qquad\qquad \mathsf{true}\ \ \longrightarrow\ \ \mathsf{f}_2\ \mathsf{a} \\
&\qquad\qquad\qquad\qquad \mathsf{false}\ \ \longrightarrow\ \ \mathsf{f}_3\ \mathsf{a} \\
&\qquad\qquad\qquad\qquad \mathsf{else}\ \ \longrightarrow\ \ \perp \\
&\qquad\ \mathsf{in}\ \ \mathsf{mapping}\ \mathsf{f}\ (\mathsf{domain}_\perp\ \mathsf{f}\ \mathsf{A}) \\
&\quad\equiv\ \mathsf{let}\ \ \mathsf{g}_1 := \mathsf{mapping}\ \mathsf{f}\ \mathsf{A} \\
&\qquad\qquad\ \mathsf{A}_2 := \mathsf{preimage}\ \mathsf{g}_1\ \{\mathsf{true}\} \\
&\qquad\qquad\ \mathsf{A}_3 := \mathsf{preimage}\ \mathsf{g}_1\ \{\mathsf{false}\} \\
&\qquad\qquad\ \mathsf{f} := \lambda\mathsf{a}.\,\mathsf{if}\ (\mathsf{f}_1\ \mathsf{a})\ (\mathsf{f}_2\ \mathsf{a})\ (\mathsf{f}_3\ \mathsf{a}) \\
&\qquad\ \mathsf{in}\ \ \mathsf{mapping}\ \mathsf{f}\ (\mathsf{domain}_\perp\ \mathsf{f}\ (\mathsf{A}_2 \uplus \mathsf{A}_3))
\end{aligned} \tag{31}$$

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

$$\begin{aligned}
&\mathsf{lift}_{\mathsf{map}}\ (\mathsf{ifte}_\perp\ \mathsf{f}_1\ \mathsf{f}_2\ \mathsf{f}_3)\ \mathsf{A} \\
&\quad\equiv\ \mathsf{let}\ \ \mathsf{g}_1 := \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_1\ \mathsf{A} \\
&\qquad\qquad\ \mathsf{g}_2 := \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_2\ (\mathsf{preimage}\ \mathsf{g}_1\ \{\mathsf{true}\}) \\
&\qquad\qquad\ \mathsf{g}_3 := \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_3\ (\mathsf{preimage}\ \mathsf{g}_1\ \{\mathsf{false}\}) \\
&\qquad\qquad\ \mathsf{A}' := (\mathsf{domain}\ \mathsf{g}_2) \uplus (\mathsf{domain}\ \mathsf{g}_3) \\
&\qquad\ \mathsf{in}\ \ \lambda\mathsf{a} \in \mathsf{A}'.\,\mathsf{if}\ (\mathsf{a} \in \mathsf{domain}\ \mathsf{g}_2)\ (\mathsf{g}_2\ \mathsf{a})\ (\mathsf{g}_3\ \mathsf{a}) \\
&\quad\equiv\ \mathsf{let}\ \ \mathsf{g}_1 := \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_1\ \mathsf{A} \\
&\qquad\qquad\ \mathsf{g}_2 := \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_2\ (\mathsf{preimage}\ \mathsf{g}_1\ \{\mathsf{true}\}) \\
&\qquad\qquad\ \mathsf{g}_3 := \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_3\ (\mathsf{preimage}\ \mathsf{g}_1\ \{\mathsf{false}\}) \\
&\qquad\ \mathsf{in}\ \ \mathsf{g}_2 \uplus_{\mathsf{map}} \mathsf{g}_3
\end{aligned} \tag{32}$$

Substituting g_1 for $\mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_1$, g_2 for $\mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_2$, and g_3 for $\mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_3$ gives a definition for $\mathsf{ifte}_{\mathsf{map}}$ (Figure 4) for which (18) holds.

4.1.4 Case: Laziness

Starting with the left side of (19), 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 diverge), 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 diverges if and only if $\mathsf{mapping}\ (\mathsf{f}\ 0)\ \mathsf{A}'$ diverges. 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 (19) holds.

4.1.5 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 programs* e, $[\![e]\!]_{\mathsf{map}} \equiv \mathsf{lift}_{\mathsf{map}}\ [\![e]\!]_\perp$.

4.1.6 Arrow Laws

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:

$$\begin{aligned}
&\mathsf{g} :: \mathsf{Bool} \overset{\sim}{\underset{\mathsf{map}}{\rightarrow}} \mathsf{Bool} \\
&\mathsf{g}\ \mathsf{A}\ :=\ \mathsf{if}\ (\mathsf{A} = \varnothing)\ (\mathsf{mapping}\ \mathsf{id}\ \mathsf{Bool})\ \varnothing
\end{aligned} \tag{33}$$

It would be nice if we could be sure that every $\mathsf{g} : \mathsf{X} \overset{\sim}{\underset{\mathsf{map}}{\rightarrow}} \mathsf{Y}$ 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* $\mathsf{g} : \mathsf{X} \overset{\sim}{\underset{\mathsf{map}}{\rightarrow}} \mathsf{Y}$*. 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 $\mathsf{g} : \mathsf{X} \overset{\sim}{\underset{\mathsf{map}}{\rightarrow}} \mathsf{Y}$. 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

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

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

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

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

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

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

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

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

Figure 5: Lazy preimage mappings and operations.

sets whose representations allow efficient operations. Therefore, in the preimage arrow, we will confine computation on points to instances of

$$\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y} ::= \langle \mathsf{Set\ Y}, \mathsf{Set\ Y} \Rightarrow \mathsf{Set\ X} \rangle \tag{34}$$

Like a mapping, an $\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y}$ 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:

$$\begin{aligned} &\mathsf{pre} : (\mathsf{X} \rightharpoonup \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y}) \\ &\mathsf{pre\ g} := \langle \mathsf{range\ g}, \lambda \mathsf{B}.\ \mathsf{preimage\ g\ B} \rangle \end{aligned} \tag{35}$$

Applying a preimage mapping to any subset of its codomain:

$$\begin{aligned} &\mathsf{ap_{pre}} : (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y}) \Rightarrow \mathsf{Set\ Y} \Rightarrow \mathsf{Set\ X} \\ &\mathsf{ap_{pre}}\ \langle \mathsf{Y}', \mathsf{p} \rangle\ \mathsf{B} := \mathsf{p}\ (\mathsf{B} \cap \mathsf{Y}') \end{aligned} \tag{36}$$

The necessary property here is that using $\mathsf{ap_{pre}}$ 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* $\mathsf{range\ g} \subseteq \mathsf{Y}' \subseteq \mathsf{Y}$, $\mathsf{preimage\ g}\ (\mathsf{B} \cap \mathsf{Y}') = \mathsf{preimage\ g\ B}$.

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

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

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* $\mathsf{h}_1 : \mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y}$ *and* $\mathsf{h}_2 : \mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y}$ *are equivalent, or* $\mathsf{h}_1 \equiv \mathsf{h}_2$, *when* $\mathsf{ap_{pre}}\ \mathsf{h}_1\ \mathsf{B} \equiv \mathsf{ap_{pre}}\ \mathsf{h}_2\ \mathsf{B}$ *for all* $\mathsf{B} \subseteq \mathsf{Y}$.

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

5.1 Preimage Mapping 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* $\mathsf{B}_2 \subseteq \mathsf{Y}_2$, $\mathsf{preimage}\ \langle \mathsf{g}_1, \mathsf{g}_2 \rangle_{\mathsf{map}}\ (\mathsf{B}_1 \times \mathsf{B}_2) = (\mathsf{preimage\ g}_1\ \mathsf{B}_1) \cap (\mathsf{preimage\ g}_2\ \mathsf{B}_2)$.

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* $\mathsf{pre}\ \langle \mathsf{g}_1, \mathsf{g}_2 \rangle_{\mathsf{map}} \equiv \langle \mathsf{pre\ g}_1, \mathsf{pre\ g}_2 \rangle_{\mathsf{pre}}$.

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 $\mathsf{B} \in \mathsf{Y}_1 \times \mathsf{Y}_2$,

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

□

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

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

□

5.3 Preimage Mapping Disjoint Union

Imported Lemma 5.8 (preimage distributes over $(\uplus_{\mathsf{map}})$)**.** *Let* $\mathsf{g}_1 \in \mathsf{X} \rightharpoonup \mathsf{Y}$ *and* $\mathsf{g}_2 \in \mathsf{X} \rightharpoonup \mathsf{Y}$ *be disjoint mappings. For all* $\mathsf{B} \subseteq \mathsf{Y}$, $\mathsf{preimage}\ (\mathsf{g}_1 \uplus_{\mathsf{map}} \mathsf{g}_2)\ \mathsf{B} = (\mathsf{preimage}\ \mathsf{g}_1\ \mathsf{B}) \uplus (\mathsf{preimage}\ \mathsf{g}_2\ \mathsf{B})$.

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

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

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

□

6. Deriving the Preimage Arrow

We are ready to define an arrow that runs programs 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 $\mathsf{X} \overset{\rightsquigarrow}{\scriptscriptstyle\mathsf{pre}} \mathsf{Y} ::= \mathsf{X} \overset{\rightarrow}{\scriptscriptstyle\mathsf{pre}} \mathsf{Y}$: we run into trouble trying to define $\mathsf{arr}_{\mathsf{pre}}$ because a preimage mapping needs an observable range. To get one, it is easiest to parameterize preimage computations on a $\mathsf{Set}\ \mathsf{X}$:

$$\mathsf{X} \overset{\rightsquigarrow}{\scriptscriptstyle\mathsf{pre}} \mathsf{Y}\ \ ::=\ \ \mathsf{Set}\ \mathsf{X} \Rightarrow (\mathsf{X} \overset{\rightarrow}{\scriptscriptstyle\mathsf{pre}} \mathsf{Y}) \tag{37}$$

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

To use Theorem 3.5, we need to define correctness using a lift from the mapping arrow to the preimage arrow:

$$\begin{aligned}
&\mathsf{lift}_{\mathsf{pre}} : (\mathsf{X} \overset{\rightsquigarrow}{\scriptscriptstyle\mathsf{map}} \mathsf{Y}) \Rightarrow (\mathsf{X} \overset{\rightsquigarrow}{\scriptscriptstyle\mathsf{pre}} \mathsf{Y}) \\
&\mathsf{lift}_{\mathsf{pre}}\ \mathsf{g}\ \mathsf{A}\ :=\ \mathsf{pre}\ (\mathsf{g}\ \mathsf{A})
\end{aligned} \tag{38}$$

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

$$\mathsf{ap}_{\mathsf{pre}}\ (\mathsf{lift}_{\mathsf{pre}}\ \mathsf{g}\ \mathsf{A})\ \mathsf{B} \equiv \mathsf{preimage}\ (\mathsf{g}\ \mathsf{A})\ \mathsf{B} \tag{39}$$

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* $\mathsf{h}_1 : \mathsf{X} \overset{\rightsquigarrow}{\scriptscriptstyle\mathsf{pre}} \mathsf{Y}$ *and* $\mathsf{h}_2 : \mathsf{X} \overset{\rightsquigarrow}{\scriptscriptstyle\mathsf{pre}} \mathsf{Y}$ *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 (15). The following subsections derive $(\mathbin{\&\&\&}_{\mathsf{pre}})$, $(\ggg_{\mathsf{pre}})$, $\mathsf{ifte}_{\mathsf{pre}}$ 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.

6.1 Case: Pairing

Starting with the left side of (17), we expand definitions, apply Theorem 5.5, and rewrite in terms of $\mathsf{lift}_{\mathsf{pre}}$:

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

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

6.2 Case: Composition

Starting with the left side of (16), we expand definitions, apply Theorem 5.7 and rewrite in terms of $\mathsf{lift}_{\mathsf{pre}}$:

$$\begin{aligned}
&\mathsf{ap}_{\mathsf{pre}}\ (\mathsf{lift}_{\mathsf{pre}}\ (\mathsf{g}_1 \ggg_{\mathsf{map}} \mathsf{g}_2)\ \mathsf{A})\ \mathsf{C} \\
&\quad\equiv\ \mathsf{let}\ \ \mathsf{g}'_1 := \mathsf{g}_1\ \mathsf{A} \\
&\qquad\ \ \mathsf{g}'_2 := \mathsf{g}_2\ (\mathsf{range}\ \mathsf{g}'_1) \\
&\qquad \mathsf{in}\ \ \mathsf{ap}_{\mathsf{pre}}\ (\mathsf{pre}\ (\mathsf{g}'_2 \circ_{\mathsf{map}} \mathsf{g}'_1))\ \mathsf{C} \\
&\quad\equiv\ \mathsf{let}\ \ \mathsf{g}'_1 := \mathsf{g}_1\ \mathsf{A} \\
&\qquad\ \ \mathsf{g}'_2 := \mathsf{g}_2\ (\mathsf{range}\ \mathsf{g}'_1) \\
&\qquad \mathsf{in}\ \ \mathsf{ap}_{\mathsf{pre}}\ ((\mathsf{pre}\ \mathsf{g}'_1) \circ_{\mathsf{pre}} (\mathsf{pre}\ \mathsf{g}'_2))\ \mathsf{C} \\
&\quad\equiv\ \mathsf{let}\ \ \mathsf{h}_1 := \mathsf{lift}_{\mathsf{pre}}\ \mathsf{g}_1\ \mathsf{A} \\
&\qquad\ \ \mathsf{h}_2 := \mathsf{lift}_{\mathsf{pre}}\ \mathsf{g}_2\ (\mathsf{range}_{\mathsf{pre}}\ \mathsf{h}_1) \\
&\qquad \mathsf{in}\ \ \mathsf{ap}_{\mathsf{pre}}\ (\mathsf{h}_2 \circ_{\mathsf{pre}} \mathsf{h}_1)\ \mathsf{C}
\end{aligned} \tag{40}$$

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 (16) holds.

6.3 Case: Conditional

Starting with the left side of (18), 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:

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

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

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

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

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

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

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

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

Figure 6: Preimage arrow definitions.

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 (18) holds.

6.4 Case: Laziness

Starting with the left side of (19), 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 (19) holds.

6.5 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 programs* e, $[\![e]\!]_{\mathsf{pre}} \equiv \mathsf{lift}_{\mathsf{pre}}\ [\![e]\!]_{\mathsf{map}}$.

6.6 Arrow Laws

As with the mapping arrow, preimage arrow computations can be unruly. We would like to assume that each $\mathsf{h} : \mathsf{X} \overset{\rightsquigarrow}{\underset{\mathsf{pre}}{}} \mathsf{Y}$ 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* $\mathsf{h} : \mathsf{X} \overset{\rightsquigarrow}{\underset{\mathsf{pre}}{}} \mathsf{Y}$*. If there exists an* $\mathsf{g} : \mathsf{X} \overset{\rightsquigarrow}{\underset{\mathsf{map}}{}} \mathsf{Y}$ *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 $\mathsf{h} : \mathsf{X} \overset{\rightsquigarrow}{\underset{\mathsf{pre}}{}} \mathsf{Y}$. 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.*

7. Preimages Under Partial Functions

We have defined everything on the top of our roadmap:

$$\begin{array}{ccccc}
\mathsf{X} \rightsquigarrow_{\perp} \mathsf{Y} & \xrightarrow{\mathsf{lift}_{\mathsf{map}}} & \mathsf{X} \overset{\rightsquigarrow}{\underset{\mathsf{map}}{}} \mathsf{Y} & \xrightarrow{\mathsf{lift}_{\mathsf{pre}}} & \mathsf{X} \overset{\rightsquigarrow}{\underset{\mathsf{pre}}{}} \mathsf{Y} \\
\downarrow{\scriptstyle \eta_{\perp^*}} & & \downarrow{\scriptstyle \eta_{\mathsf{map}^*}} & & \downarrow{\scriptstyle \eta_{\mathsf{pre}^*}} \\
\mathsf{X} \rightsquigarrow_{\perp^*} \mathsf{Y} & \xrightarrow[\mathsf{lift}_{\mathsf{map}^*}]{} & \mathsf{X} \overset{\rightsquigarrow}{\underset{\mathsf{map}^*}{}} \mathsf{Y} & \xrightarrow[\mathsf{lift}_{\mathsf{pre}^*}]{} & \mathsf{X} \overset{\rightsquigarrow}{\underset{\mathsf{pre}^*}{}} \mathsf{Y}
\end{array} \tag{41}$$

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 (41) is an arrow homomorphism.

7.1 Motivation

Probabilistic functions that may diverge, but converge 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:

$$\mathsf{geometric}\ \mathsf{p}\ :=\ \mathsf{if}\ (\mathsf{random} < \mathsf{p})\ 0\ (1 + \mathsf{geometric}\ \mathsf{p}) \tag{42}$$

For any $\mathsf{p} > 0$, geometric p may diverge, but the probability of always taking the false branch is $(1 - \mathsf{p}) \times (1 - \mathsf{p}) \times (1 - \mathsf{p}) \times \cdots = 0$. Divergence with probability 0 simply does not happen in practice.

Suppose we interpret (42) as $\mathsf{h} : \mathsf{R} \overset{\rightsquigarrow}{\underset{\mathsf{pre}}{}} \mathbb{N}$, 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 $\mathsf{h}\ \mathsf{R}'$ converges.
2. Ensuring each $\mathsf{r} \in \mathsf{R}$ contains enough random numbers.
3. Determining how random indexes numbers in r.

Ensuring $\mathsf{h}\ \mathsf{R}'$ converges 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.

We need a way to assign unique indexes to expressions.

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*, 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

$$\begin{aligned} \mathsf{left}\ (\mathsf{p}/\,\mathsf{q}) &:= (\mathsf{p} - \tfrac{1}{2})/\,\mathsf{q} \\ \mathsf{right}\ (\mathsf{p}/\,\mathsf{q}) &:= (\mathsf{p} + \tfrac{1}{2})/\,\mathsf{q} \end{aligned} \tag{43}$$

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

In any case, J is always countable, and can be thought of as a set of indexes into an infinite binary tree. Values of type $\mathsf{J} \to \mathsf{R}$ encode an infinite binary tree 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. (XXX: find better cite than the experimental-looking Hackage package)

The applicative store arrow transformer's type constructor takes a store type s and an arrow type $\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}$:

$$\mathsf{AStore}\ \mathsf{s}\ (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \ ::= \ \mathsf{J} \Rightarrow (\langle \mathsf{s}, \mathsf{x}\rangle \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \tag{44}$$

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

$$\begin{aligned} &\eta_{\mathsf{a}^*} : (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \Rightarrow \mathsf{AStore}\ \mathsf{s}\ (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \\ &\eta_{\mathsf{a}^*}\ \mathsf{f}\ \mathsf{j} \ := \ \mathsf{arr}_{\mathsf{a}}\ \mathsf{snd} \ggg_{\mathsf{a}} \mathsf{f} \end{aligned} \tag{45}$$

Because f never accesses the store, j is ignored.

Figure 7 defines the remaining combinators. Each subcomputation receives $\mathsf{left}\ \mathsf{j}$, $\mathsf{right}\ \mathsf{j}$, 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. Probabilistic and partial programs are interpreted using combinators that read a store at their expression index.

7.4 Probabilistic Programs

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

Let $\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y} ::= \mathsf{AStore}\ \mathsf{R}\ (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y})$. The following combinator returns the number at its own index in the random source:

$$\begin{aligned} &\mathsf{random}_{\mathsf{a}^*} : \mathsf{x} \rightsquigarrow_{\mathsf{a}^*} [0, 1] \\ &\mathsf{random}_{\mathsf{a}^*}\ \mathsf{j} \ := \ \mathsf{arr}_{\mathsf{a}}\ (\mathsf{fst} \ggg \pi\ \mathsf{j}) \end{aligned} \tag{46}$$

We extend the let-calculus semantic function with

$$[\![\mathsf{random}]\!]_{\mathsf{a}^*} :\equiv \mathsf{random}_{\mathsf{a}^*} \tag{47}$$

for arrows a^* for which $\mathsf{random}_{\mathsf{a}^*}$ is defined.

7.5 Partial Programs

One utimately implementable way to avoid divergence in computing preimages is to use a 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* $\mathsf{t}\ \mathsf{j} = \mathsf{true}$ *or* $\mathsf{t}\ \mathsf{j} = \mathsf{false}$ *for no more than finitely many* $\mathsf{j} \in \mathsf{J}$*.*

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

$$\begin{aligned} &\mathsf{branch}_{\mathsf{a}^*} : \mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{Bool} \\ &\mathsf{branch}_{\mathsf{a}^*}\ \mathsf{j} \ := \ \mathsf{arr}_{\mathsf{a}}\ (\mathsf{fst} \ggg \pi\ \mathsf{j}) \end{aligned} \tag{48}$$

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

$$\begin{aligned} &\mathsf{agrees} : \langle \mathsf{Bool}, \mathsf{Bool}\rangle \Rightarrow \mathsf{Bool}_\perp \\ &\mathsf{agrees}\ \langle \mathsf{b}_1, \mathsf{b}_2\rangle \ := \ \mathsf{if}\ (\mathsf{b}_1 = \mathsf{b}_2)\ \mathsf{b}_1\ \perp \end{aligned} \tag{49}$$

$$\begin{aligned} &\mathsf{ifte}^{\Downarrow}_{\mathsf{a}^*} : (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{Bool}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y}) \\ &\mathsf{ifte}^{\Downarrow}_{\mathsf{a}^*}\ \mathsf{k}_1\ \mathsf{k}_2\ \mathsf{k}_3\ \mathsf{j} \ := \\ &\quad \mathsf{ifte}_{\mathsf{a}}\ ((\mathsf{k}_1\ (\mathsf{left}\ \mathsf{j})\ \&\&\&_{\mathsf{a}}\ \mathsf{branch}_{\mathsf{a}^*}\ \mathsf{j}) \ggg_{\mathsf{a}} \mathsf{arr}_{\mathsf{a}}\ \mathsf{agrees}) \\ &\qquad (\mathsf{k}_2\ (\mathsf{left}\ (\mathsf{right}\ \mathsf{j}))) \\ &\qquad (\mathsf{k}_3\ (\mathsf{right}\ (\mathsf{right}\ \mathsf{j}))) \end{aligned} \tag{50}$$

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

Every computation defined using the let-calculus semantic function $[\![\cdot]\!]_{\mathsf{a}}$, whose *defining expression* converges, must have its recurrences guarded by an if. Thus, if we stick to well-defined let-calculus programs, we need only replace $\mathsf{ifte}_{\mathsf{a}^*}$ with $\mathsf{ifte}^{\Downarrow}_{\mathsf{a}^*}$ to ensure *computations* always converge.

We extend $[\![\cdot]\!]_{\mathsf{a}^*}$ by replacing the if rule:

$$\begin{aligned} [\![\mathsf{if}\ e_c\ e_t\ e_f]\!]^{\Downarrow}_{\mathsf{a}^*} \ :\equiv \ &\mathsf{ifte}^{\Downarrow}_{\mathsf{a}^*}\ [\![e_c]\!]^{\Downarrow}_{\mathsf{a}^*} \\ &(\mathsf{lazy}_{\mathsf{a}}\ \lambda 0.\ [\![e_t]\!]^{\Downarrow}_{\mathsf{a}^*}) \\ &(\mathsf{lazy}_{\mathsf{a}}\ \lambda 0.\ [\![e_f]\!]^{\Downarrow}_{\mathsf{a}^*}) \\ [\![e]\!]^{\Downarrow}_{\mathsf{a}^*} \ :\equiv \ &[\![e]\!]_{\mathsf{a}^*} \end{aligned} \tag{51}$$

If $[\![e]\!]^{\Downarrow}_{\mathsf{a}^*}$ is well-defined, it always converges.

7.6 Partial, Probabilistic Programs

Let $\mathsf{S} ::= \mathsf{R} \times \mathsf{T}$ and $\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y} ::= \mathsf{AStore}\ \mathsf{S}\ (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y})$, and update the $\mathsf{random}_{\mathsf{a}^*}$ and $\mathsf{branch}_{\mathsf{a}^*}$ combinators to reflect

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

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

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

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

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

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

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

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

that the store is now a pair:

$$\begin{aligned} &\mathsf{random}_{\mathsf{a}^*} : \mathsf{x} \rightsquigarrow_{\mathsf{a}^*} [0,1] \\ &\mathsf{random}_{\mathsf{a}^*}\ \mathsf{j} \;:=\; \mathsf{arr}_{\mathsf{a}}\ (\mathsf{fst} \ggg \mathsf{fst} \ggg \pi\ \mathsf{j}) \end{aligned} \tag{52}$$

$$\begin{aligned} &\mathsf{branch}_{\mathsf{a}^*} : \mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{Bool} \\ &\mathsf{branch}_{\mathsf{a}^*}\ \mathsf{j} \;:=\; \mathsf{arr}_{\mathsf{a}}\ (\mathsf{fst} \ggg \mathsf{snd} \ggg \pi\ \mathsf{j}) \end{aligned} \tag{53}$$

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

7.7 Correctness

The combinator η_{a^*} lifts the interpretations of *pure* programs (i.e. those that do not access the store) to AStore arrow computations. To prove it is a homomorphism, and thus that *pure* programs interpreted using $[\![\cdot]\!]_{\mathsf{a}^*}$ are correct, we need to extend equivalence to be more extensional.

Definition 7.4 (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* $\mathsf{j} \in \mathsf{J}$*.*

Because AStore accepts any arrow type $\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}$, we can prove homomorphism of η_{a^*} 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 7.5 (pure AStore arrow correctness)**.** η_{a^*} *is an arrow homomorphism.*

Proof. Defining $\mathsf{arr}_{\mathsf{a}^*}$ as a composition clearly meets the first homomorphism law (15). For homomorphism laws (16–18), start from the right side, expand definitions, and use arrow laws (21–23) to factor out $\mathsf{arr}_{\mathsf{a}}\ \mathsf{snd}$.

For (19), additionally β-expand within the outer thunk, then use the lazy distributive law (24) to extract $\mathsf{arr}_{\mathsf{a}}\ \mathsf{snd}$. □

Corollary 7.6 (pure semantic correctness)**.** *For all pure programs* e*,* $[\![e]\!]_{\mathsf{a}^*} \equiv \eta_{\mathsf{a}^*}\ [\![e]\!]_{\mathsf{a}}$ *and* $[\![e]\!]^{\Downarrow}_{\mathsf{a}^*} \equiv \eta_{\mathsf{a}^*}\ [\![e]\!]^{\Downarrow}_{\mathsf{a}}$*.*

To prove correct the interpretations of effectful programs (i.e. those that access the store), we need a lift between AStore arrows. Let $\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y} ::= \mathsf{AStore\ s\ (x} \rightsquigarrow_{\mathsf{a}} \mathsf{y)}$ and $\mathsf{x} \rightsquigarrow_{\mathsf{b}^*} \mathsf{y} ::= \mathsf{AStore\ s\ (x} \rightsquigarrow_{\mathsf{b}^*} \mathsf{y)}$. Define

$$\begin{aligned} &\mathsf{lift}_{\mathsf{b}^*} : (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{b}^*} \mathsf{y}) \\ &\mathsf{lift}_{\mathsf{b}^*}\ \mathsf{f}\ \mathsf{j} \;:=\; \mathsf{lift}_{\mathsf{b}}\ (\mathsf{f\ j}) \end{aligned} \tag{54}$$

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

$$\begin{array}{ccc} \mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y} & \xrightarrow{\mathsf{lift}_{\mathsf{b}}} & \mathsf{x} \rightsquigarrow_{\mathsf{b}} \mathsf{y} \\ {\scriptstyle \eta_{\mathsf{a}^*}}\downarrow & & \downarrow{\scriptstyle \eta_{\mathsf{b}^*}} \\ \mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y} & \xrightarrow[\mathsf{lift}_{\mathsf{b}^*}]{} & \mathsf{x} \rightsquigarrow_{\mathsf{b}^*} \mathsf{y} \end{array} \tag{55}$$

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 7.7 (natural transformation)**.** *If* $\mathsf{lift}_{\mathsf{b}}$ *is an arrow homomorphism, then* (55) *commutes.*

Proof. Expand definitions and apply homomorphism laws (16) and (15) 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 7.8 (effectful AStore arrow correctness)**.** *If* $\mathsf{lift}_{\mathsf{b}}$ *is an arrow homomorphism, then* $\mathsf{lift}_{\mathsf{b}^*}$ *is an arrow homomorphism.*

Proof. XXX: do this □

Corollary 7.9 (effectful semantic correctness)**.** *For all programs* e*,* $[\![e]\!]_{\mathsf{b}^*} \equiv \mathsf{lift}_{\mathsf{b}^*}\ [\![e]\!]_{\mathsf{a}^*}$ *and* $[\![e]\!]^{\Downarrow}_{\mathsf{b}^*} \equiv \mathsf{lift}_{\mathsf{b}^*}\ [\![e]\!]^{\Downarrow}_{\mathsf{a}^*}$*.*

From here on, let $\mathsf{x} \rightsquigarrow_{\perp^*} \mathsf{y} ::= \mathsf{AStore\ (R \times T)\ (x} \rightsquigarrow_{\perp} \mathsf{y)}$; similarly for $\mathsf{X} \underset{\mathsf{map}^*}{\rightsquigarrow} \mathsf{Y}$ and $\mathsf{X} \underset{\mathsf{pre}^*}{\rightsquigarrow} \mathsf{Y}$.

Corollary 7.10 (mapping* and preimage* arrow correctness)**.** *The following diagram commutes:*

$$\begin{array}{ccccc} \mathsf{X} \rightsquigarrow_{\perp} \mathsf{Y} & \xrightarrow{\mathsf{lift}_{\mathsf{map}}} & \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y} & \xrightarrow{\mathsf{lift}_{\mathsf{pre}}} & \mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y} \\ {\scriptstyle \eta_{\perp^*}}\downarrow & & \downarrow{\scriptstyle \eta_{\mathsf{map}^*}} & & \downarrow{\scriptstyle \eta_{\mathsf{pre}^*}} \\ \mathsf{X} \rightsquigarrow_{\perp^*} \mathsf{Y} & \xrightarrow[\mathsf{lift}_{\mathsf{map}^*}]{} & \mathsf{X} \underset{\mathsf{map}^*}{\rightsquigarrow} \mathsf{Y} & \xrightarrow[\mathsf{lift}_{\mathsf{pre}^*}]{} & \mathsf{X} \underset{\mathsf{pre}^*}{\rightsquigarrow} \mathsf{Y} \end{array} \tag{56}$$

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

Corollary 7.11 (effectful semantic correctness)**.** *For all programs e,*

$$\begin{aligned} [\![e]\!]_{\mathsf{pre}^*} &\equiv \mathsf{lift}_{\mathsf{pre}^*}\ (\mathsf{lift}_{\mathsf{map}^*}\ [\![e]\!]_{\perp^*}) \\ [\![e]\!]^{\Downarrow}_{\mathsf{pre}^*} &\equiv \mathsf{lift}_{\mathsf{pre}^*}\ (\mathsf{lift}_{\mathsf{map}^*}\ [\![e]\!]^{\Downarrow}_{\perp^*}) \end{aligned} \tag{57}$$

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

The maximal domain of $\mathsf{k} : \mathsf{X} \rightsquigarrow_{\mathsf{a}^*} \mathsf{Y}$ *is that of* $\mathsf{k}\ \mathsf{j}_0$*, and is a subset of* $(\mathsf{R} \times \mathsf{T}) \times \mathsf{X}$.

Because the above statements imply convergence, A^* is a subset of the largest domain for which the computations converge. It is not too hard (but is a bit tedious) to show 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.

Theorem 7.13 (domain closure operators)**.** *If* $\mathsf{f} : \mathsf{X} \rightsquigarrow_{\perp} \mathsf{Y}$, $\mathsf{g} : \mathsf{X} \overset{\rightsquigarrow}{\mathsf{map}} \mathsf{Y}$ *and* $\mathsf{h} : \mathsf{X} \overset{\rightsquigarrow}{\mathsf{pre}} \mathsf{Y}$*, then* $\mathsf{domain}_{\perp}\ \mathsf{f}$, $\mathsf{domain} \circ \mathsf{g}$*, and* $\mathsf{domain}_{\mathsf{pre}} \circ \mathsf{h}$ *are monotone, nonincreasing, and idempotent.*

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

(XXX: idempotence isn't necessary for maximal domains to exist)

Theorem 7.14 (divergence as an error)**.** *Suppose* $[\![e]\!]^{\Downarrow}_{\perp^*} : \mathsf{X} \rightsquigarrow_{\perp^*} \mathsf{Y}$ *converges. Let* $\mathsf{f} := [\![e]\!]_{\perp^*}\ \mathsf{j}_0$ *and* $\mathsf{f}' := [\![e]\!]^{\Downarrow}_{\perp^*}\ \mathsf{j}_0$*. For all* $\mathsf{r} \in \mathsf{R}$, $\mathsf{t} \in \mathsf{T}$ *and* $\mathsf{a} \in \mathsf{X}$,

1. *If* $\mathsf{f}\ \langle\langle \mathsf{r}, \mathsf{t}\rangle, \mathsf{a}\rangle = \mathsf{b}$*, then* $\mathsf{f}'\ \langle\langle \mathsf{r}, \mathsf{t}'\rangle, \mathsf{a}\rangle = \mathsf{b}$ *for some* $\mathsf{t}' \in \mathsf{T}$.
2. *If* $\mathsf{f}\ \langle\langle \mathsf{r}, \mathsf{t}\rangle, \mathsf{a}\rangle$ *diverges,* $\mathsf{f}'\ \langle\langle \mathsf{r}, \mathsf{t}'\rangle, \mathsf{a}\rangle = \perp$ *for all* $\mathsf{t}' \in \mathsf{T}$.

Proof. Case 1. Define $\mathsf{t}' \in \mathsf{J} \to \mathsf{Bool}_{\perp}$ such that $\mathsf{t}'\ \mathsf{j} = \mathsf{z}$ if the subcomputation with index j is an if whose test returns z, otherwise $\mathsf{t}'\ \mathsf{j} = \perp$. Because $\mathsf{f}\ \langle\langle \mathsf{r}, \mathsf{t}\rangle, \mathsf{a}\rangle$ converges, $\mathsf{t}'\ \mathsf{j} \neq \perp$ for at most finitely many j, so $\mathsf{t}' \in \mathsf{T}$.

Case 2. Let $\mathsf{t}' \in \mathsf{T}$. There exists an infinite *suffix* $\mathsf{J}' \subset \mathsf{J}$ closed under left and right, such that for all $\mathsf{j} \in \mathsf{J}'$, $\mathsf{t}'\ \mathsf{j} = \perp$. Because $\mathsf{f}\ \langle\langle \mathsf{r}, \mathsf{t}\rangle, \mathsf{a}\rangle$ diverges, the indexes of its if computations are unbounded; there is therefore an if at some index j such that $\mathsf{j} \in \mathsf{J}'$. It returns $\perp$, so $\mathsf{f}'\ \langle\langle \mathsf{r}, \mathsf{t}'\rangle, \mathsf{a}\rangle = \perp$. □

Corollary 7.15 (correct convergence everywhere)**.** *Suppose* $[\![e]\!]^{\Downarrow}_{\perp^*} : \mathsf{X} \rightsquigarrow_{\perp^*} \mathsf{Y}$ *converges and has maximal domain* A^**. For any* $\mathsf{a} \in (\mathsf{R} \times \mathsf{T}) \times \mathsf{X}$, $\mathsf{A} \subseteq (\mathsf{R} \times \mathsf{T}) \times \mathsf{X}$, *and* $\mathsf{B} \subseteq \mathsf{Y}$,

$$\begin{aligned} [\![e]\!]^{\Downarrow}_{\perp^*}\ \mathsf{j}_0\ \mathsf{a} &= \mathsf{if}\ (\mathsf{a} \in \mathsf{A}^*)\ ([\![e]\!]_{\perp^*}\ \mathsf{j}_0\ \mathsf{a})\ \perp \\ [\![e]\!]^{\Downarrow}_{\mathsf{map}^*}\ \mathsf{j}_0\ \mathsf{A} &= [\![e]\!]_{\mathsf{map}^*}\ \mathsf{j}_0\ (\mathsf{A} \cap \mathsf{A}^*) \\ \mathsf{ap}_{\mathsf{pre}}\ ([\![e]\!]^{\Downarrow}_{\mathsf{pre}^*}\ \mathsf{j}_0\ \mathsf{A})\ \mathsf{B} &= \mathsf{ap}_{\mathsf{pre}}\ ([\![e]\!]_{\mathsf{pre}^*}\ \mathsf{j}_0\ (\mathsf{A} \cap \mathsf{A}^*))\ \mathsf{B} \end{aligned} \tag{58}$$

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

8. Measurability

We have not assigned probabilities to any sets yet.

For $\mathsf{g} : \mathsf{X} \rightharpoonup \mathsf{Y}$, the probability of an output set $\mathsf{B} \subseteq \mathsf{Y}$ is

$$\mathsf{P}\ (\mathsf{preimage}\ \mathsf{g}\ \mathsf{B}) \tag{59}$$

where $\mathsf{P} \in \mathcal{P}\ \mathsf{X} \rightharpoonup [0, 1]$ is a probability measure on X. This was the motivation for defining arrows to compute preimages in the first place. P is a *partial* function: domain P consists only of *measurable* subsets of X. Before we can compute probabilities, we need to ensure P is applied only to measurable sets, which requires preimages of measurable subsets under g to be measurable.

To save space, we assume readers are familiar with either topology or measure theory. Readers unfamiliar with both may wish to skip to the next section.

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* mapping fst X *and* 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.*

8.1 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 all partial, probabilistic programs 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* $\mathsf{g} : \mathsf{X} \overset{\rightsquigarrow}{\mathsf{map}} \mathsf{Y}$ *is* ***$\mathcal{A}$-$\mathcal{B}$-measurable*** *if* $\mathsf{g}\ \mathsf{A}^*$ *is an $\mathcal{A}$-$\mathcal{B}$-measurable mapping, where* A^* *is* g*'s maximal domain.*

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

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

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

Proof. Use the mapping arrow law (Definition 4.6) 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.

8.1.1 Case: 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* $\mathsf{g} : \mathsf{X} \mathrel{\tilde{\rightharpoonup}_{\mathsf{map}}} \mathsf{Y}$. *For any* $\mathsf{A'} \subseteq \mathsf{A} \subseteq \mathsf{A}^*$, $\mathsf{g\ A'} \subseteq \mathsf{g\ A}$.

Proof. Follows from Definition 4.6. □

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

Proof. Follows from Theorem 7.13. □

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* $\mathsf{g_1} : \mathsf{X} \mathrel{\tilde{\rightharpoonup}_{\mathsf{map}}} \mathsf{Y}$ *is* $\mathcal{A}$-$\mathcal{B}$-*measurable and* $\mathsf{g_2} : \mathsf{Y} \mathrel{\tilde{\rightharpoonup}_{\mathsf{map}}} \mathsf{Z}$ *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,

$$\begin{array}{rcl}(\mathsf{g_1} \ggg_{\mathsf{map}} \mathsf{g_2})\ \mathsf{A}^{**} & = & \mathsf{let}\ \ \mathsf{g_1'} := \mathsf{g_1\ A}^{**} \\ & & \qquad\ \mathsf{g_2'} := \mathsf{g_2\ (range\ g_1')} \\ & & \mathsf{in}\ \ \mathsf{g_2'} \circ_{\mathsf{map}} \mathsf{g_1'}\end{array} \tag{60}$$

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, $\mathsf{g_2''} \circ_{\mathsf{map}} \mathsf{g_1'}$ is $\mathcal{A}$-$\mathcal{C}$-measurable. We need only show that $\mathsf{g_2'} \circ_{\mathsf{map}} \mathsf{g_1'} = \mathsf{g_2''} \circ_{\mathsf{map}} \mathsf{g_1'}$, which by Lemma 8.10 is true if $\mathsf{range\ g_1'} \subseteq \mathsf{domain\ g_2'}$ and $\mathsf{g_2'} \subseteq \mathsf{g_2''}$.

By Theorem 8.12, $\mathsf{A}^{**} \subseteq \mathsf{A}^*$ implies $\mathsf{domain\ g_1'} = \mathsf{A}^{**}$. 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 $\mathsf{g_2'} \subseteq \mathsf{g_2''}$. □

8.1.2 Case: Pairing

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* $\mathsf{g_1} : \mathsf{X} \mathrel{\tilde{\rightharpoonup}_{\mathsf{map}}} \mathsf{Y_1}$ *is* $\mathcal{A}$-$\mathcal{B}_1$-*measurable and* $\mathsf{g_2} : \mathsf{X} \mathrel{\tilde{\rightharpoonup}_{\mathsf{map}}} \mathsf{Y_2}$ *is* $\mathcal{A}$-$\mathcal{B}_2$-*measurable, then* $\mathsf{g_1} \mathbin{\&\&\&_{\mathsf{map}}} \mathsf{g_2}$ *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 $\mathsf{g_1} \mathbin{\&\&\&_{\mathsf{map}}} \mathsf{g_2}$ is $\mathsf{A}^{**} := \mathsf{A_1^*} \cap \mathsf{A_2^*}$, which is in $\mathcal{A}$. By definition, $(\mathsf{g_1} \mathbin{\&\&\&_{\mathsf{map}}} \mathsf{g_2})\ \mathsf{A}^{**} = \langle \mathsf{g_1\ A}^{**}, \mathsf{g_2\ A}^{**} \rangle_{\mathsf{map}}$, which by Lemma 8.15 is $\mathcal{A}$-$(\mathcal{B}_1 \otimes \mathcal{B}_2)$-measurable. □

8.1.3 Case: Conditional

Imported Lemma 8.17 (union of disjoint, measurable mappings)**.** *Let* $\mathsf{G} : \mathsf{Set\ (X} \rightharpoonup \mathsf{Y)}$ *be a countable set of* $\mathcal{A}$-$\mathcal{B}$-*measurable mappings with disjoint domains. Their union is* $\mathcal{A}$-$\mathcal{B}$-*measurable.*

Theorem 8.18 (measurability under $\mathsf{ifte}_{\mathsf{map}}$)**.** *If* $\mathsf{g_1} : \mathsf{X} \mathrel{\tilde{\rightharpoonup}_{\mathsf{map}}} \mathsf{Bool}$ *is* $\mathcal{A}$-$(\mathcal{P}\ \mathsf{Bool})$-*measurable, and* $\mathsf{g_2} : \mathsf{X} \mathrel{\tilde{\rightharpoonup}_{\mathsf{map}}} \mathsf{Y}$ *and* $\mathsf{g_3} : \mathsf{X} \mathrel{\tilde{\rightharpoonup}_{\mathsf{map}}} \mathsf{Y}$ *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 $\mathsf{g_1}$'s, $\mathsf{g_2}$'s and $\mathsf{g_3}$'s maximal domains. The maximal domain of $\mathsf{ifte_{map}\ g_1\ g_2\ g_3}$ is

$$\begin{array}{rcl}\mathsf{A_2^{**}} & := & \mathsf{A_2^*} \cap \mathsf{preimage\ (g_1\ \mathcal{A}_1^*)\ \{true\}} \\ \mathsf{A_3^{**}} & := & \mathsf{A_3^*} \cap \mathsf{preimage\ (g_1\ \mathcal{A}_1^*)\ \{false\}} \\ \mathsf{A}^{**} & := & \mathsf{A_2^{**}} \uplus \mathsf{A_3^{**}}\end{array} \tag{61}$$

Because $\mathsf{preimage\ (g_1\ \mathcal{A}_1^*)\ B} \in \mathcal{A}$ for any $\mathsf{B} \subseteq \mathsf{Bool}$, $\mathsf{A}^{**} \in \mathcal{A}$. By definition,

$$\begin{array}{rcl}\mathsf{ifte_{map}\ g_1\ g_2\ g_3\ A}^{**} & = & \mathsf{let}\ \ \mathsf{g_1'} := \mathsf{g_1\ A}^{**} \\ & & \qquad\ \mathsf{g_2'} := \mathsf{g_2\ (preimage\ g_1'\ \{true\})} \\ & & \qquad\ \mathsf{g_3'} := \mathsf{g_3\ (preimage\ g_1'\ \{false\})} \\ & & \mathsf{in}\ \ \mathsf{g_2'} \uplus_{\mathsf{map}} \mathsf{g_3'}\end{array} \tag{62}$$

By hypothesis, $\mathsf{g_1'}$, $\mathsf{g_2'}$ and $\mathsf{g_3'}$ are measurable mappings, and the mapping arrow law imples $\mathsf{g_2'}$ and $\mathsf{g_3'}$ have disjoint domains. Apply Lemma 8.17. □

8.1.4 Case: 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 $\mathsf{lazy_{map}}$)**.** *Let* $\mathsf{g} : 1 \Rightarrow (\mathsf{X} \mathrel{\tilde{\rightharpoonup}_{\mathsf{map}}} \mathsf{Y})$. *If* $\mathsf{g\ 0}$ *is* $\mathcal{A}$-$\mathcal{B}$-*measurable, then* $\mathsf{lazy_{map}\ g}$ *is* $\mathcal{A}$-$\mathcal{B}$-*measurable.*

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

$$\mathsf{lazy}_{\mathsf{map}}\ \mathsf{g}\ \mathsf{A}^{**} \ = \ \mathsf{if}\ (\mathsf{A}^{**} = \varnothing)\ \varnothing\ (\mathsf{g}\ \mathsf{0}\ \mathsf{A}^{**}) \tag{63}$$

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

8.2 Measurable Probabilistic Programs

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 $\mathsf{g} : \mathsf{X} \underset{\mathsf{map}^*}{\rightsquigarrow} \mathsf{Y}$ 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:*

$$\Sigma\ \langle \mathsf{X}_1, \mathsf{X}_2 \rangle = \Sigma\ \mathsf{X}_1 \otimes \Sigma\ \mathsf{X}_2 \tag{64}$$

$$\Sigma\ (\mathsf{J} \to \mathsf{X}) = (\Sigma\ \mathsf{X})^{\otimes \mathsf{J}} \tag{65}$$

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

For $\mathsf{ifte}_{\mathsf{map}^*}$ measurability and later proofs, we define

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

$$\Sigma\ \mathsf{T} \ ::= \ \mathcal{P}\ \mathsf{T} \tag{67}$$

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 (66). □

Theorem 8.27 (all expressions are measurable)**.** *For any program e lacking function definitions, $[\![e]\!]_{\mathsf{map}^*}$ is measurable.*

Proof. By structural induction and the above theorems. □

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

Proof. Let $\mathsf{j} \in \mathsf{J}$ be the largest 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 condition's index is greater than j with the equivalent expression $\bot$. Because g converges, recurrences must be guarded by if, so this process terminates after finitely many applications. □

Theorem 8.29 (all probabilistic programs are measurable)**.** *If $[\![e]\!]^{\Downarrow}_{\mathsf{map}^*}$ converges, it 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 7.15, $\mathsf{g}' = \mathsf{g}\ \mathsf{j}_0\ \mathsf{A}^*$ where A^* is g's maximal domain; thus we need only show that g' is a measurable mapping.

By mapping arrow monotonicity (XXX?) (Theorem 8.11),

$$\mathsf{g}' \ = \ \bigcup_{\mathsf{t} \in \mathsf{T}} \mathsf{g}\ \mathsf{j}_0\ ((\mathsf{R} \times \{\mathsf{t}\}) \times \mathsf{X}) \tag{68}$$

By Theorem 8.28, for every $\mathsf{t} \in \mathsf{T}$, there is an expression that computes $\mathsf{g}\ ((\mathsf{R} \times \{\mathsf{t}\}) \times \mathsf{X})$. By (67) and Theorem 8.27, each is measurable. By the mapping arrow law (Definition 4.6), each is disjoint. By Lemma 8.17, their union is measurable. □

Theorem 8.29 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 lifts and defunctionalizes lambda expressions. Thus, all higher-order programs are measurable.

8.3 Random Store Probabilities

Preimages under probabilistic programs are measurable subsets of $(\mathsf{R} \times \mathsf{T}) \times \mathsf{X}$. While it is possible to put probability measures on such domains, doing so would be surprising for end-users. For example, the probabilities of outputs of the $\mathsf{geometric}$ function defined in (42) would depend not only on the probability of $\mathsf{random} < \mathsf{p}$, but also on some arbitrary probability that each branch is taken. It would not define the geometric distribution.

We therefore have to measure *projections* of subsets of $(\mathsf{R} \times \mathsf{T}) \times \mathsf{X}$. Unfortunately, projected sets are generally not measurable. Fortunately, ours is a special case: the excluded dimensions are countable.

As previously, we start with measuring the maximal domain.

Definition 8.30 (standard probability measure)**.** *For a type X, a **standard probability measure** is a probability measure $\mathsf{P} \in \mathcal{P}\ \mathsf{X} \rightharpoonup [0, 1]$ where $\mathsf{domain}\ \mathsf{P} = \Sigma\ \mathsf{X}$.*

Definition 8.31 (maximal probability)**.** *Let $\mathsf{g} : \mathsf{X} \underset{\mathsf{map}^*}{\rightsquigarrow} \mathsf{Y}$ be measurable, with A^* its maximal domain. Let $\mathsf{P} \in \mathcal{P}\ \mathsf{R} \rightharpoonup [0, 1]$ be a standard probability measure over random stores. The **maximal probability** of g is $\mathsf{P}\ (\mathsf{image}\ (\mathsf{fst} \ggg \mathsf{fst})\ \mathsf{A}^*)$.*

XXX: not too happy with the proof below, or with focusing on maximal probabilities

Theorem 8.32 (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 $a_2 \in X_2$. Because image fst distributes over unions, image fst A is a countable union of sets in $\Sigma\ X_1$. □

Theorem 8.33. *Let* $g : X \rightsquigarrow_{map^*} Y$ *be measurable. If* X *is at most countable and* $\Sigma\ X = \mathcal{P}\ X$*, then* g*'s maximal probability is well-defined.*

Proof. T is countable; apply Theorem 8.32 twice. □

In particular, for programs interpreted using $[\![\cdot]\!]_{map^*}$, typically $X = \{\langle\rangle\}$ (the empty list/stack), so their maximal probabilities are well-defined.

Corollary 8.34. *Let* $g : X \rightsquigarrow_{map^*} Y$ *be measurable, with* X *at most countable and* $\Sigma\ X = \mathcal{P}\ X$*. Let* $A \subseteq A^*$ *be a measurable set.* P (image (fst ⋙ fst) A) *is well-defined.*

In particular, for any converging $g := [\![e]\!]^{\Downarrow}_{map^*}$, preimages A of measurable subsets B are measurable, and the random store component of A has a well-defined probability.

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: preimage g B is uncomputable for most uncountable sets B no matter how cleverly they are represented. Further, because pre, lift_{pre} and arr_{pre} are ultimately defined in terms of preimage, we cannot implement them.

Fortunately, we need only certain lifts. Figure 2 (which defines $[\![\cdot]\!]_a$) lifts id, const b, fst and snd. Sections 7.4 and 7.5, which define the combinators used to interpret partial, probabilistic programs, lift π 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 arr_{pre} id, arr_{pre} fst, arr_{pre} snd, arr_{pre} (const b) and arr_{pre} (π j). (We will deal with agrees separately.) By inspecting these expressions, we see that we need to model sets in a way that the following are representable and can be computed in finite time:

- $A \cap B$, $\varnothing$, {true}, {false} and {b} for every const b
- $A_1 \times A_2$, proj_{fst} A and proj_{snd} A
- $J \to X$, project j A and unproject j A B
- $A = \varnothing$

(69)

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

9.2 Rectangular Families

Definition 9.1 (rectangular family)**.** *For a set* X *used as a type,* Rect X *denotes the* ***rectangular family*** *of subsets of* X*, which must be satisfy the following rules:*

$$\mathsf{Rect}\ \langle X_1, X_2\rangle \;=\; (\mathsf{Rect}\ X_1) \boxtimes (\mathsf{Rect}\ X_2) \tag{70}$$

$$\mathsf{Rect}\ (J \to X) \;=\; (\mathsf{Rect}\ X)^{\boxtimes J} \tag{71}$$

where

$$\mathcal{A}_1 \boxtimes \mathcal{A}_2 \;:=\; \{A_1 \times A_2 \mid A_1 \in \mathcal{A}_1, A_2 \in \mathcal{A}_2\} \tag{72}$$

$$\mathcal{A}^{\boxtimes J} \;:=\; \{\textstyle\prod_{j\in J} A_j \mid \forall\, j \in J.\, A_j \in \mathcal{A}\} \tag{73}$$

lift cartesian products to sets of sets.

XXX: consider baking "no more than finitely many non-full axes" into the definition of rectangular family

For example, if Rect $\mathbb{R}$ contains all the closed real intervals, then by (70), $[0, 2] \times [1, \pi] \in \mathsf{Rect}\ \langle\mathbb{R}, \mathbb{R}\rangle$.

For every non-product type X, we require $\varnothing \in \mathsf{Rect}\ X$, a universal set $X \in \mathsf{Rect}\ X$, singletons $\{a\} \in \mathsf{Rect}\ X$ for all $a \in X$, and for Rect X to be closed under intersection. It is not hard to show that these properties extend to rectangular families, and that the collection of all rectangular families is closed under products, projections, and unproject.

Further, all of the operations in (69) 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 $\mathsf{Rect}\ \langle X_1, X_2\rangle$ are modeled by pairs of type $\langle \mathsf{Rect}\ X_1, \mathsf{Rect}\ X_2\rangle$. Though J is infinite, sets in $\mathsf{Rect}\ (J \to X)$ can be modeled by *finite* binary trees of sets in Rect X, because a converging preimage computation can apply unproject only finitely many times to $J \to X$.

We defined one nonrectangular domain: the set of branch traces T, which contains every $t \in J \to \mathsf{Bool}_\perp$ for which $t\ j \neq \perp$ for no more than finitely many J. Fortunately, under conditions that are always met while computing approximate preimages, we can represent T subsets as $J \to \mathsf{Bool}_\perp$ rectangles, implicitly intersected with T.

Theorem 9.2. *Let* $T' \in \mathsf{Rect}\ (J \to \mathsf{Bool}_\perp)$ *such that* $\perp \notin \mathsf{project}\ j\ T'$ *for no more than finitely many* $j \in J$*. Then* $\mathsf{project}\ j\ (T' \cap T) = \mathsf{project}\ j\ T'$.

Proof. The subset case is by monotonicity of projections. For the superset case, let $b \in \mathsf{project}\ j\ T'$. Define t by

$$t\ j' = \begin{cases} b & j' = j \\ \text{any member of } \mathsf{project}\ j'\ T' & \perp \notin \mathsf{project}\ j'\ T' \\ \perp & \perp \in \mathsf{project}\ j'\ T' \end{cases} \tag{74}$$

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

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

9.3 Approximate Preimage Mapping Operations

Implementing lazy_{pre} (defined in Figure 6) requires computing pre, but only for the empty mapping, which is trivial: $\mathsf{pre}\ \varnothing \equiv \langle\varnothing, \lambda B.\, \varnothing\rangle$. Implementing the other combinators requires implementing the preimage mapping operations $(\circ_{pre})$, $\langle\cdot,\cdot\rangle_{pre}$ and $(\uplus_{pre})$.

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

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

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

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

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

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

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

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

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 show that the following replacement:

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

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 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 non-product 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

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

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

9.4 Partial Programs

We have enough of the specification to implement the preimage arrow and the AStore arrow transformer. But we cannot yet deal with programs that may diverge, or that converge with probability 1. For that, we need to approximate $\mathsf{ifte}^{\Downarrow}_{\mathsf{pre}^*}$ (50).

Directly implementing $\mathsf{ifte}^{\Downarrow}_{\mathsf{pre}^*}$, and thus agrees, cannot work. Turning agrees into a mapping shows why:

$$\begin{aligned}
&\mathsf{arr}_{\mathsf{map}}\ \mathsf{agrees}\ (\mathsf{Bool} \times \mathsf{Bool}) = \\
&\qquad \{\langle \langle \mathsf{true}, \mathsf{true}\rangle, \mathsf{true}\rangle, \langle \langle \mathsf{false}, \mathsf{false}\rangle, \mathsf{false}\rangle\}
\end{aligned} \tag{78}$$

The preimage of Bool is $\{\langle \mathsf{true}, \mathsf{true}\rangle, \langle \mathsf{false}, \mathsf{false}\rangle\}$, which is not rectangular.

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

$$\begin{aligned}
&\mathsf{ifte}^{\Downarrow}_{\mathsf{pre}^*}\ \mathsf{k}_1\ \mathsf{k}_2\ \mathsf{k}_3\ \mathsf{j}\ \mathsf{A} \equiv \\
&\quad \mathsf{let}\ \ \langle \mathsf{C}_\mathsf{k}, \mathsf{p}_\mathsf{k}\rangle := \mathsf{k}_1\ \mathsf{j}_1\ \mathsf{A} \\
&\qquad \langle \mathsf{C}_\mathsf{b}, \mathsf{p}_\mathsf{b}\rangle := \mathsf{branch}_{\mathsf{pre}^*}\ \mathsf{j}\ \mathsf{A} \\
&\qquad\qquad \mathsf{C}_2 := \mathsf{C}_\mathsf{k} \cap \mathsf{C}_\mathsf{b} \cap \{\mathsf{true}\} \\
&\qquad\qquad \mathsf{C}_3 := \mathsf{C}_\mathsf{k} \cap \mathsf{C}_\mathsf{b} \cap \{\mathsf{false}\} \\
&\qquad\qquad \mathsf{A}_2 := \mathsf{p}_\mathsf{k}\ \mathsf{C}_2 \cap \mathsf{p}_\mathsf{b}\ \mathsf{C}_2 \\
&\qquad\qquad \mathsf{A}_3 := \mathsf{p}_\mathsf{k}\ \mathsf{C}_3 \cap \mathsf{p}_\mathsf{b}\ \mathsf{C}_3 \\
&\quad \mathsf{in}\ \ (\mathsf{k}_2\ \mathsf{j}_2\ \mathsf{A}_2) \uplus_{\mathsf{pre}} (\mathsf{k}_3\ \mathsf{j}_3\ \mathsf{A}_3)
\end{aligned} \tag{79}$$

where $\mathsf{j}_1 = \mathsf{left}\ \mathsf{j}$ and so on. This has no trace of agrees and clearly preserves rectangularity if k_1, k_2 and k_3 do. Yet it is still not good enough. When A_2 and A_3 overapproximate $\varnothing$, it takes unnecessary branches, which can lead to divergence. In the exact semantics, a well-defined program interpreted using $\mathsf{ifte}^{\Downarrow}_{\mathsf{pre}^*}$ never diverges.

Suppose we defined the approximating $\mathsf{ifte}^{\Downarrow}_{\mathsf{pre}^*}{}'$ to take *no branches* when $\mathsf{branch}_{\mathsf{pre}^*}\ \mathsf{j}\ \mathsf{A}$ contains both true and false. Because $\mathsf{branch}_{\mathsf{pre}^*}\ \mathsf{j}\ \mathsf{A}$ can return $\{\mathsf{true}\}$ or $\{\mathsf{false}\}$ for at most finitely many $\mathsf{j} \in \mathsf{J}$, there exists a suffix set $\mathsf{J}' \subseteq \mathsf{J}$ for which no branches will be taken.

The returned preimage mapping's range is a subset of Y, and the preimages it computes must be subsets of $\mathsf{A}_2 \vee \mathsf{A}_3$. Therefore, we might replace the let body in (79) with

$$\begin{aligned}
&\mathsf{if}\ (\mathsf{C}_\mathsf{b} = \{\mathsf{true}, \mathsf{false}\}) \\
&\quad \langle \mathsf{Y}, \lambda \mathsf{B}.\, \mathsf{A}_2 \vee \mathsf{A}_3\rangle \\
&\quad (\mathsf{k}_2\ \mathsf{j}_2\ \mathsf{A}_2 \uplus_{\mathsf{pre}} \mathsf{k}_3\ \mathsf{j}_3\ \mathsf{A}_3)
\end{aligned} \tag{80}$$

which computes the same preimages if $\mathsf{C}_\mathsf{b} \subset \{\mathsf{true}, \mathsf{false}\}$, and takes no branches and overapproximates otherwise. A well-defined program interpreted using a conditional defined this way should always converge.

Unfortunately, we cannot refer to Y in a function definition: it is only part of the type of $\mathsf{ifte}^{\Downarrow}_{\mathsf{pre}^*}$. We need a value $\top \in \mathsf{Rect}\ \mathsf{X}$ for every X to represent X itself. It should behave exactly as X; e.g. $\mathsf{x} \in \top$ for all $\mathsf{x} \in \mathsf{X}$ and $\top \cap \mathsf{X}' = \mathsf{X}'$ for all $\mathsf{X}' \subseteq \mathsf{X}$. Thus, the approximating $\mathsf{ifte}^{\Downarrow}_{\mathsf{pre}^*}{}'$ can be written in terms of $\top$ instead of in terms of its (erased) type 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.

9.5 Correctness

From here on, $[\![\cdot]\!]^{\Downarrow}_{\mathsf{pre}^*}{}'$ interprets programs as approximating preimage* arrow computations using $\mathsf{ifte}^{\Downarrow}_{\mathsf{pre}^*}{}'$.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

(d) Additional preimage* arrow combinators, for retrieving random numbers and branch traces, and computing without diverging.

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.

The following theorems assume $\mathsf{h} := [\![e]\!]^{\Downarrow}_{\mathsf{pre}^*} : \mathsf{X} \underset{\mathsf{pre}^*}{\rightsquigarrow}' \mathsf{Y}$ and $\mathsf{h}' := [\![e]\!]^{\Downarrow}{}'_{\mathsf{pre}^*} : \mathsf{X} \underset{\mathsf{pre}^*}{\rightsquigarrow}' \mathsf{Y}$ for some program e. Further, define

$$\begin{aligned} \mathsf{refine}\ \mathsf{A} &:= \mathsf{ap}_{\mathsf{pre}}\ (\mathsf{h}\ \mathsf{j}_0\ \mathsf{A})\ \mathsf{B} \\ \mathsf{refine}'\ \mathsf{A} &:= \mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}'\ \mathsf{j}_0\ \mathsf{A})\ \mathsf{B} \end{aligned} \tag{81}$$

Theorem 9.5 (approximation)**.** *For all* $\mathsf{A} \in \mathsf{Rect}\ \langle \mathsf{S}, \mathsf{X} \rangle$ *and* $\mathsf{B} \in \mathsf{Rect\ Y}$, $\mathsf{refine\ A} \subseteq \mathsf{refine}'\ \mathsf{A}$.

Proof. By construction. □

Theorem 9.6 (monotonicity)**.** $\mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}'\ \mathsf{j}_0\ \mathsf{A})\ \mathsf{B}$ *is monotone in both* A *and* B*.*

Proof. XXX: todo □

Theorem 9.7 (approximate preimages are nonincreasing)**.** *For all* $\mathsf{A} \in \mathsf{Rect}\ \langle \mathsf{S}, \mathsf{X} \rangle$ *and* $\mathsf{B} \in \mathsf{Rect}\ \mathsf{Y}$*,* $\mathsf{refine}'\ \mathsf{A} \subseteq \mathsf{A}$*.*

Proof. XXX: todo □

Corollary 9.8 (disjointness of approximate preimages)**.** *If* A_1 *and* A_2 *are disjoint, then* $\mathsf{refine}'\ \mathsf{A}_1$ *and* $\mathsf{refine}'\ \mathsf{A}_2$ *are disjoint.*

9.6 Limitations

With disjointness and monotonicity properties, 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. For example, starting with any partition $\mathcal{A} : \mathsf{Set}\ (\mathsf{Rect}\ \langle \mathsf{S}, \mathsf{X} \rangle)$, we might repeat the following:

1. Refine every rectangle in $\mathcal{A}$: let $\mathcal{A}' := \mathsf{image}\ \mathsf{refine}'\ \mathcal{A}$.
2. Partition the rectangles in $\mathcal{A}'$: let $\mathcal{A} := \bigcup_{\mathsf{A}' \in \mathcal{A}'} \mathsf{partition}\ \mathsf{A}'$.

If the second step yields finer partitions, it seems the sum of the measures of each rectangle in $\mathcal{A}$ should approach the probability of B.

We are not so fortunate: in the limit, we would be computing the **Jordan outer measure** (XXX: cite) of the preimage of B, which is not always equal to its measure.

For example, suppose we extend the language with real numbers—which is reasonable, given that Bayesian practice requires them. If it also has $(<)$, which is also a reasonable extension, it is possible to define $(\geq)$ and thus $(=)$ for real numbers, and thus a $\mathsf{rational?}$ function that semidecides rationality by enumerating the rationals and returning true when it finds its input. The preimage of $\{\mathsf{true}\}$ under $[\![\mathsf{rational?}\ \mathsf{random}]\!]^{\Downarrow}_{\mathsf{map}^*}$ has measure 0, but its outer Jordan measure is 1. Roughly, it is impossible to tightly cover a dense set like $\mathbb{Q}$ using finite partitions.

For programs that halt with probability 1, finite discretization tends to work out anyway. Unfortunately, the sparsity of work on Jordan measure, especially in general spaces, makes it difficult to characterize the conditions under which converging approximations converge. We leave this for future work.

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 (XXX: cite) 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 (XXX: cite). 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 GHC's multi-parameter typeclasses or indexed type families (XXX: cite) 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 and newtype wrappers for arrow 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. Roughly, it consists of a monomorphic rectangular set data type, 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.

Section 9.6 outlines a discretization algorithm that seems to converge for programs that halt with probability 1, consisting of repeatedly refining a program's domain and repartitioning it. We do not use this algorithm in our main implementation, not only because it may not converge: it is also terribly 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 $4^{10} = 1,048,576$.

To avoid this explosion in time and space in the main implementation, we sample from the discretization instead of enumerating it.

11. Unrelated Work

12. Related Work

13. Conclusions and Future Work

References

[1] J. Hughes. Programming with arrows. In *5th International Summer School on Advanced Functional Programming*, pages 73–129, 2005.

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