

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: Turing-equivalent, probabilistic programming language with a semantics and implementation that supports any kind of probability measure and—critically—can do probabilistic conditioning

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 formal semantics.

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:

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

While t 's distribution has a density (a standard bell curve at mean μ), the distribution of t' does not. Using densities, this program cannot have a meaning. Placing a condition on the value of t' to discover how restricting its value would affect μ 's distribution—an activity that should be central to Bayesian practice—is unfathomable.

The restrictions placed on legal programs are indeed onerous, if uses of such benign functions as **min** and **max** are suspect. 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 $f : X \rightarrow Y$, then the **preimage measure**

$$P (\text{preimage } f \ B) \quad (2)$$

defines the distribution over subsets B of Y . 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 $B \subseteq Y$, **preimage** $f \ B$ 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.

It seems this approach should make it easy to interpret probabilistic programs measure-theoretically. For a probabilistic program $f : X \rightarrow Y$, the probability measure on output sets $B \subseteq Y$ should be defined by preimages of B under f and the probability measure on X . Unfortunately, it is diffi-

cult 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:

$$\begin{array}{ccccc}
 X \rightsquigarrow_{\perp} Y & \xrightarrow{\text{lift}_{\text{map}}} & X \rightsquigarrow_{\text{map}} Y & \xrightarrow{\text{lift}_{\text{pre}}} & X \rightsquigarrow_{\text{pre}} Y \\
 \eta_{\perp}^* \downarrow & & \downarrow \eta_{\text{map}}^* & & \downarrow \eta_{\text{pre}}^* \\
 X \rightsquigarrow_{\perp}^* Y & \xrightarrow{\text{lift}_{\text{map}}^*} & X \rightsquigarrow_{\text{map}}^* Y & \xrightarrow{\text{lift}_{\text{pre}}^*} & X \rightsquigarrow_{\text{pre}}^* Y
 \end{array} \quad (3)$$

From top-left to top-right, $X \rightsquigarrow_{\perp} Y$ computations are intensional functions that may raise errors, $X \rightsquigarrow_{\text{map}} Y$ computations produce extensional functions, and $X \rightsquigarrow_{\text{pre}} Y$ 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 $X \rightsquigarrow_{\text{pre}}^* Y$, derived from $X \rightsquigarrow_{\text{pre}} Y$, for computing conservative approximations of preimages.

An implementation implements the semantic function, and $X \rightsquigarrow_{\perp} Y$ and $X \rightsquigarrow_{\text{pre}}^* Y$ arrows.

Most of our correctness theorems rely on proofs that every lift and η in (3) 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 implementation 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 $\{\{x\}, \{x, y\}\}$. 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).

We assume data structures, including pairs, are encoded as ***primitive*** ordered pairs with the first element a unique tag, so they can be distinguished by checking tags. Accessors such as **fst** and **snd** are trivial to define.

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.
- $x \Rightarrow y$ denotes a partial function.
- $\langle x, y \rangle$ denotes a pair of values with types x and y .
- **Set** x denotes a set with members of type x .

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.

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

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

$$\begin{array}{ll} \text{true} : \text{Bool} & (\epsilon) : x \Rightarrow \text{Set } x \Rightarrow \text{Bool} \\ \text{false} : \text{Bool} & \mathcal{P} : \text{Set } x \Rightarrow \text{Set } (\text{Set } x) \\ \emptyset : \text{Set } x & \bigcup : \text{Set } (\text{Set } x) \Rightarrow \text{Set } x \\ \omega : \text{Ord} & \text{image} : (x \Rightarrow y) \Rightarrow \text{Set } x \Rightarrow \text{Set } y \\ \text{take} : \text{Set } x \Rightarrow x & |\cdot| : \text{Set } x \Rightarrow \text{Ord} \end{array} \quad (4)$$

Shortly, \emptyset is the empty set, ω is the cardinality of the natural numbers, $\text{take } \{x\}$ reduces to x and diverges for non-singleton sets, $x \in A$ decides membership, $\mathcal{P} A$ reduces to the set of subsets of A , $\bigcup \mathcal{A}$ reduces to the union of the sets in \mathcal{A} , $\text{image } f A$ applies f to each member of A and reduces to the set of results, and $|A|$ 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 (ϵ) . *Internal* extensional equality is defined by either of the following equivalent statements:

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

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 , $f = g$ reduces to **true** when f and g are structurally identical modulo renaming. For example, $(\lambda x. x) = (\lambda y. y)$ reduces to **true**, but $(\lambda x. 2) = (\lambda x. 1 + 1)$ 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 $(\lambda x. x) 1$ and 1 are (externally, extensionally) unequal, we can say that $(\lambda x. x) 1 = 1$.

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_2 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.

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

2.2 Additional Functions and Forms

We assume a desugaring pass over λ_{ZFC} expressions, which automatically curries (including for the two-argument primitives (ϵ) and **image**), and interprets special binding forms such as indexed unions $\bigcup_{x \in e_A} e$, destructuring binds as in $\text{swap } \langle x, y \rangle := \langle y, x \rangle$, and comprehensions like $\{x \in A \mid x \in B\}$. We assume we have logical operators, bounded quantifiers, and typical set operations.

A less typical set operation we use is disjoint union:

$$\begin{aligned} (\uplus) &: \text{Set } x \Rightarrow \text{Set } x \Rightarrow \text{Set } x \\ A \uplus B &:= \text{if } (A \cap B = \emptyset) (A \cup B) (\text{take } \emptyset) \end{aligned} \quad (6)$$

$A \uplus B$ 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 $\{\langle 0, 1 \rangle, \langle 1, 2 \rangle, \langle 2, 3 \rangle, \dots\}$. 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. $(f \ x)$) to apply mappings.

Syntax for defining unnamed mappings is defined by

$$\lambda x_a \in e_A. e_b \equiv \text{mapping } (\lambda x_a. e_b) e_A \quad (7)$$

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

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

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

$$\begin{aligned} \pi &: J \Rightarrow (J \rightarrow X) \Rightarrow X \\ \pi j f &:= f j \end{aligned} \quad (9)$$

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

We do not give typical minimal arrow definitions. For each arrow a , instead of first_a , we define $(\&\&\&_a)$ —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 first_a may be defined in terms of $(\&\&\&_a)$ and vice-versa [1], we give $(\&\&\&_a)$ definitions because the applicable theorems are in terms of pairing functions.

One way to strengthen an arrow a is to define an additional combinator left_a , which can be used to choose an arrow computation based on the result of another. Again, we define a different combinator, ifte_a , to make it easier to apply contemporary 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

$\text{domain} : (X \multimap Y) \Rightarrow \text{Set } X$	$\langle \cdot, \cdot \rangle_{\text{map}} : (X \multimap Y_1) \Rightarrow (X \multimap Y_2) \Rightarrow (X \multimap Y_1 \times Y_2)$
$\text{domain} := \text{image fst}$	$\langle g_1, g_2 \rangle_{\text{map}} := \text{let } A := (\text{domain } g_1) \cap (\text{domain } g_2) \\ \text{in } \lambda a \in A. \langle g_1 a, g_2 a \rangle$
$\text{range} : (X \multimap Y) \Rightarrow \text{Set } Y$	$(\circ_{\text{map}}) : (Y \multimap Z) \Rightarrow (X \multimap Y) \Rightarrow (X \multimap Z)$
$\text{range} := \text{image snd}$	$g_2 \circ_{\text{map}} g_1 := \text{let } A := \text{preimage } g_1 (\text{domain } g_2) \\ \text{in } \lambda a \in A. g_2 (g_1 a)$
$\text{preimage} : (X \multimap Y) \Rightarrow \text{Set } Y \Rightarrow \text{Set } X$	$(\uplus_{\text{map}}) : (X \multimap Y) \Rightarrow (X \multimap Y) \Rightarrow (X \multimap Y)$
$\text{preimage } f B := \{a \in \text{domain } f \mid f a \in B\}$	$g_1 \uplus_{\text{map}} g_2 := \text{let } A := (\text{domain } g_1) \uplus (\text{domain } g_2) \\ \text{in } \lambda a \in A. \text{if } (a \in \text{domain } g_1) (g_1 a) (g_2 a)$
$\text{restrict} : (X \multimap Y) \Rightarrow \text{Set } X \Rightarrow (X \multimap Y)$	
$\text{restrict } f A := \lambda a \in (A \cap \text{domain } f). f a$	

Figure 1: Operations on mappings.

strict λ -calculus (such as λ_{ZFC}) requires an extra combinator to defer computations in conditional branches.

For example, suppose we define the **function arrow** with choice, by defining

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

and try to define the following recursive function:

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

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

Using **lazy** $f a := f 0 a$, which receives thunks and returns arrow computations, we can write **halt-on-true** as

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

which diverges only when applied to **false** in any λ -calculus.

Definition 3.1 (arrow with choice). *A binary type constructor (\rightsquigarrow_a) and the combinators*

$$\begin{aligned}
\text{arr}_a : (x \Rightarrow y) &\Rightarrow (x \rightsquigarrow_a y) \\
(\ggg_a) : (x \rightsquigarrow_a y) &\Rightarrow (y \rightsquigarrow_a z) \Rightarrow (x \rightsquigarrow_a z) \\
(\&\&_a) : (x \rightsquigarrow_a y) &\Rightarrow (x \rightsquigarrow_a z) \Rightarrow (x \rightsquigarrow_a \langle y, z \rangle)
\end{aligned} \tag{13}$$

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

$$\begin{aligned}
\text{ifte}_a : (x \rightsquigarrow_a \text{Bool}) &\Rightarrow (x \rightsquigarrow_a y) \Rightarrow (x \rightsquigarrow_a y) \Rightarrow (x \rightsquigarrow_a y) \\
\text{lazy}_a : (1 \Rightarrow (x \rightsquigarrow_a y)) &\Rightarrow (x \rightsquigarrow_a y)
\end{aligned} \tag{14}$$

*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 arr_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 $\text{lift}_b : (x \rightsquigarrow_a y) \Rightarrow (x \rightsquigarrow_b y)$ 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 :

$$\text{lift}_b (\text{arr}_a f) \equiv \text{arr}_b f \tag{15}$$

$$\text{lift}_b (f_1 \ggg_a f_2) \equiv (\text{lift}_b f_1) \ggg_b (\text{lift}_b f_2) \tag{16}$$

$$\text{lift}_b (f_1 \&\&_a f_2) \equiv (\text{lift}_b f_1) \&\&_b (\text{lift}_b f_2) \tag{17}$$

$$\text{lift}_b (\text{ifte}_a f_1 f_2 f_3) \equiv \text{ifte}_b (\text{lift}_b f_1) (\text{lift}_b f_2) (\text{lift}_b f_3) \tag{18}$$

$$\text{lift}_b (\text{lazy}_a f) \equiv \text{lazy}_b \lambda 0. \text{lift}_b (f 0) \tag{19}$$

The homomorphism laws state that arr_a must be a homomorphism from the function arrow to arrow a .

The monoid and structural arrow laws play little role in our semantics or its correctness. For the arrows we define, then, we elide the proofs of these arrow laws, and concentrate on homomorphisms.

XXX: actually, need to prove some of them, to prove that the natural transformation for the applicative store-passing arrow transformer is a homomorphism

3.2 First-Order Let-Calculus Semantics

Figure 2 shows a transformation $\llbracket \cdot \rrbracket_a$ 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. $\llbracket \cdot \rrbracket_a$ 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 accessed by their De Bruijn index, where index 0 refers to the innermost binding.

Perhaps unsurprisingly, the interpretation acts like a stack machine. The final expression has type $\langle \rangle \rightsquigarrow_a y$, 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 $\langle x, \langle \rangle \rangle \rightsquigarrow_a y$ 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 $\llbracket \cdot \rrbracket_a$ would complicate the simple proof of the following theorem, which underlies most of our semantic correctness claims.

Theorem 3.3 (homomorphisms distribute over programs). *Let $\text{lift}_b : (x \rightsquigarrow_a y) \Rightarrow (x \rightsquigarrow_b y)$ be an arrow homomorphism. For all programs e , $\llbracket e \rrbracket_b \equiv \text{lift}_b \llbracket e \rrbracket_a$.*

Proof. By structural induction on program terms.

$$\begin{array}{ll}
\llbracket x := e; \dots; e_{body} \rrbracket_a & \equiv x := \llbracket e \rrbracket_a; \dots; \llbracket e_{body} \rrbracket_a \\
\llbracket x e \rrbracket_a & \equiv \llbracket \langle e, \rangle \rrbracket_a \ggg_a x \\
\llbracket \text{let } e \text{ } e_{body} \rrbracket_a & \equiv (\llbracket e \rrbracket_a \&\&\&_a \text{arr}_a \text{id}) \ggg_a \llbracket e_{body} \rrbracket_a \\
\llbracket \text{env } 0 \rrbracket_a & \equiv \text{arr}_a \text{fst} \\
\llbracket \text{env } (n+1) \rrbracket_a & \equiv \text{arr}_a \text{snd} \ggg_a \llbracket \text{env } n \rrbracket_a \\
\llbracket v \rrbracket_a & \equiv \text{arr}_a \lambda _ . v \\
\llbracket \langle e_1, e_2 \rangle \rrbracket_a & \equiv \llbracket e_1 \rrbracket_a \&\&\&_a \llbracket e_2 \rrbracket_a \\
\llbracket \text{fst } e \rrbracket_a & \equiv \llbracket e \rrbracket_a \ggg_a \text{arr}_a \text{fst} \\
\llbracket \text{snd } e \rrbracket_a & \equiv \llbracket e \rrbracket_a \ggg_a \text{arr}_a \text{snd} \\
\llbracket \text{if } e_c \text{ } e_t \text{ } e_f \rrbracket_a & \equiv \text{ifte}_a \llbracket e_c \rrbracket_a (\text{lazy}_a \lambda 0. \llbracket e_t \rrbracket_a) (\text{lazy}_a \lambda 0. \llbracket e_f \rrbracket_a)
\end{array}$$

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

Bases cases proceed by expansion and using $\text{lift}_b \circ \text{arr}_a \equiv \text{arr}_b$ (15). For example, for constants:

$$\begin{aligned}
\text{lift}_b \llbracket v \rrbracket_a & \equiv \text{lift}_b (\text{arr}_a \lambda _ . v) \\
& \equiv \text{arr}_b \lambda _ . v \\
& \equiv \llbracket v \rrbracket_b
\end{aligned}$$

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

$$\begin{aligned}
\text{lift}_b \llbracket \langle e_1, e_2 \rangle \rrbracket_a & \equiv \text{lift}_b (\llbracket e_1 \rrbracket_a \&\&\&_a \llbracket e_2 \rrbracket_a) \\
& \equiv (\text{lift}_b \llbracket e_1 \rrbracket_a) \&\&\&_b (\text{lift}_b \llbracket e_2 \rrbracket_a) \\
& \equiv \llbracket e_1 \rrbracket_b \&\&\&_b \llbracket e_2 \rrbracket_b \\
& \equiv \llbracket \langle e_1, e_2 \rangle \rrbracket_b
\end{aligned}$$

It is not hard to check the remaining cases. \square

If we assume that lift_b defines correct behavior for arrow b in terms of arrow a , and prove that lift_b is a homomorphism, then by Theorem 3.3, $\llbracket \cdot \rrbracket_b$ is correct.

4. The Bottom and Mapping Arrows

We are certain that the preimage arrow correctly computes preimages under some function f because we ultimately *derive* it from a simpler arrow used to construct f .

One obvious candidate for the simpler arrow is the function arrow, defined in (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 $x \rightsquigarrow_{\perp} y ::= x \Rightarrow y_{\perp}$, where the inhabitants of y_{\perp} are the error value \perp as well as the inhabitants of y . The type Bool_{\perp} , for example, denotes the members of $\text{Bool} \cup \{\perp\}$.

Theorem 4.1. arr_{\perp} , $(\&\&\&_{\perp})$, (\ggg_{\perp}) , ifte_{\perp} and lazy_{\perp} define an arrow.

Proof. The bottom arrow is the Maybe monad’s Kleisli arrow with $\text{Nothing} = \perp$. \square

4.1 Deriving the Mapping Arrow

The contemporary theorems we need are 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 $X \rightsquigarrow_{\text{map}} Y ::= X \rightarrow Y$, with $f_1 \ggg_{\text{map}} f_2 := f_2 \circ_{\text{map}} f_1$ and $f_1 \&\&\&_{\text{map}} f_2 := \langle f_1, f_2 \rangle_{\text{map}}$. Unfortunately, we couldn’t define $\text{arr}_{\text{map}} : (X \Rightarrow Y) \Rightarrow (X \rightarrow Y)$: to compute a mapping, we need a domain, but a lambda’s domain is unobservable.

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

$$X \rightsquigarrow_{\text{map}} Y ::= \text{Set } X \Rightarrow (X \rightarrow Y) \quad (20)$$

Notice that \perp is absent in $\text{Set } X \Rightarrow (X \rightarrow Y)$. This will make it easier to exclude diverging inputs further on. The absence of \perp and the fact that the type parameters are sets will make it easier to apply contemporary theorems, which know nothing of error values and lambda types.

Further on, we will need every computation $g : X \rightsquigarrow_{\text{map}} Y$ to meet the **mapping arrow restriction law**: for all $A \subseteq X$ and $A' \subseteq A$ for which $g \ A$ converges,

$$g \ A' = \text{restrict } (g \ A) \ A' \quad (21)$$

Roughly, g must act as if it returns restricted mappings.

To use Theorem 3.3 to prove that programs interpreted using $\llbracket \cdot \rrbracket_{\text{map}}$ 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 domain_{\perp} that computes the subset of A on which f does not return \perp . We define that first, and then define lift_{map} in terms of it:

$$\begin{aligned}
\text{domain}_{\perp} : (X \rightsquigarrow_{\perp} Y) & \Rightarrow \text{Set } X \Rightarrow \text{Set } X \\
\text{domain}_{\perp} f \ A & := \{a \in A \mid f \ a \neq \perp\}
\end{aligned} \quad (22)$$

$$\begin{aligned}
\text{lift}_{\text{map}} : (X \rightsquigarrow_{\perp} Y) & \Rightarrow (X \rightsquigarrow_{\text{map}} Y) \\
\text{lift}_{\text{map}} f \ A & := \text{mapping } f \ (\text{domain}_{\perp} f \ A)
\end{aligned} \quad (23)$$

So $\text{lift}_{\text{map}} f \ A$ is like $\text{mapping } f \ A$, but without inputs that produce errors—a good notion of correctness.

If lift_{map} is to be a homomorphism, mapping arrow computation equivalence needs to be more extensional than observational equivalence.

Definition 4.2 (mapping arrow equivalence). *Two mapping arrow computations $g_1 : X \rightsquigarrow_{\text{map}} Y$ and $g_2 : X \rightsquigarrow_{\text{map}} Y$ are equivalent, or $g_1 \equiv g_2$, when $g_1 \ A \equiv g_2 \ A$ for all $A \subseteq X$.*

Clearly $\text{arr}_b := \text{lift}_b \circ \text{arr}_a$ meets the first homomorphism identity (15), so we define arr_{map} as a composition. The following subsections derive $(\&\&\&_{\text{map}})$, (\ggg_{map}) , ifte_{map} and lazy_{map} from their corresponding bottom arrow combinators, in a way that ensures lift_{map} is an arrow homomorphism. Figure 4 contains the resulting definitions.

$$\begin{aligned}
x \rightsquigarrow_{\perp} y &::= x \Rightarrow y_{\perp} \\
\text{arr}_{\perp} : (X \Rightarrow Y) &\Rightarrow (X \rightsquigarrow_{\perp} Y) \\
\text{arr}_{\perp} f &:= f \\
(\ggg_{\perp}) : (X \rightsquigarrow_{\perp} Y) &\Rightarrow (Y \rightsquigarrow_{\perp} Z) \Rightarrow (X \rightsquigarrow_{\perp} Z) \\
(f_1 \ggg_{\perp} f_2) a &:= \text{if } (f_1 a = \perp) \perp (f_2 (f_1 a)) \\
(\lll_{\perp}) : (X \rightsquigarrow_{\perp} Y_1) &\Rightarrow (X \rightsquigarrow_{\perp} Y_2) \Rightarrow (X \rightsquigarrow_{\perp} \langle Y_1, Y_2 \rangle) \\
(f_1 \lll_{\perp} f_2) a &:= \text{if } (f_1 a = \perp \text{ or } f_2 a = \perp) \perp \langle f_1 a, f_2 a \rangle \\
\text{ifte}_{\perp} : (X \rightsquigarrow_{\perp} \text{Bool}) &\Rightarrow (X \rightsquigarrow_{\perp} Y) \Rightarrow (X \rightsquigarrow_{\perp} Y) \Rightarrow (X \rightsquigarrow_{\perp} Y) \\
\text{ifte}_{\perp} f_1 f_2 f_3 a &:= \text{case } f_1 a \\
&\quad \text{true} \Rightarrow f_2 a \\
&\quad \text{false} \Rightarrow f_3 a \\
&\quad \text{else} \Rightarrow \perp \\
\text{lazy}_{\perp} : (1 \Rightarrow (X \rightsquigarrow_{\perp} Y)) &\Rightarrow (X \rightsquigarrow_{\perp} Y) \\
\text{lazy}_{\perp} f a &:= f 0 a
\end{aligned}$$

Figure 3: Bottom arrow definitions.

$$\begin{aligned}
X \rightsquigarrow_{\text{map}} Y &::= \text{Set } X \Rightarrow (X \rightarrow Y) \\
\text{arr}_{\text{map}} : (X \Rightarrow Y) &\Rightarrow (X \rightsquigarrow_{\text{map}} Y) \\
\text{arr}_{\text{map}} &:= \text{lift}_{\text{map}} \circ \text{arr}_{\perp} \\
(\ggg_{\text{map}}) : (X \rightsquigarrow_{\text{map}} Y) &\Rightarrow (Y \rightsquigarrow_{\text{map}} Z) \Rightarrow (X \rightsquigarrow_{\text{map}} Z) \\
(g_1 \ggg_{\text{map}} g_2) A &:= \text{let } g'_1 := g_1 A \\
&\quad g'_2 := g_2 (\text{range } g'_1) \\
&\quad \text{in } g'_2 \circ_{\text{map}} g'_1 \\
(\lll_{\text{map}}) : (X \rightsquigarrow_{\text{map}} Y_1) &\Rightarrow (X \rightsquigarrow_{\text{map}} Y_2) \Rightarrow (X \rightsquigarrow_{\text{map}} \langle Y_1, Y_2 \rangle) \\
(g_1 \lll_{\text{map}} g_2) A &:= \langle g_1 A, g_2 A \rangle_{\text{map}} \\
\text{ifte}_{\text{map}} : (X \rightsquigarrow_{\text{map}} \text{Bool}) &\Rightarrow (X \rightsquigarrow_{\text{map}} Y) \Rightarrow (X \rightsquigarrow_{\text{map}} Y) \Rightarrow (X \rightsquigarrow_{\text{map}} Y) \\
\text{ifte}_{\text{map}} g_1 g_2 g_3 A &:= \text{let } g'_1 := g_1 A \\
&\quad g'_2 := g_2 (\text{preimage } g'_1 \{\text{true}\}) \\
&\quad g'_3 := g_3 (\text{preimage } g'_1 \{\text{false}\}) \\
&\quad \text{in } g'_2 \uplus_{\text{map}} g'_3 \\
\text{lazy}_{\text{map}} : (1 \Rightarrow (X \rightsquigarrow_{\text{map}} Y)) &\Rightarrow (X \rightsquigarrow_{\text{map}} Y) \\
\text{lazy}_{\text{map}} g A &:= \text{if } (A = \emptyset) \emptyset (g 0 A) \\
\text{lift}_{\text{map}} : (X \rightsquigarrow_{\perp} Y) &\Rightarrow (X \rightsquigarrow_{\text{map}} Y) \\
\text{lift}_{\text{map}} f A &:= \{ \langle a, b \rangle \in \text{mapping } f A \mid b \neq \perp \}
\end{aligned}$$

Figure 4: Mapping arrow definitions.

4.1.1 Case: Pairing

Starting with the left side of (17), we first expand definitions. For any $f_1 : X \rightsquigarrow_{\perp} Y$, $f_2 : X \rightsquigarrow_{\perp} Z$, and $A \subseteq X$,

$$\begin{aligned}
&\text{lift}_{\text{map}} (f_1 \lll_{\perp} f_2) A \\
&\equiv \text{lift}_{\text{map}} (\lambda a. \text{if } (f_1 a = \perp \text{ or } f_2 a = \perp) \perp \langle f_1 a, f_2 a \rangle) A \\
&\equiv \text{let } f := \lambda a. \text{if } (f_1 a = \perp \text{ or } f_2 a = \perp) \perp \langle f_1 a, f_2 a \rangle \\
&\quad \text{in } \text{mapping } f (\text{domain}_{\perp} f A)
\end{aligned} \tag{24}$$

Next, we replace the definition of A' with one that does not depend on f , and rewrite in terms of $\text{lift}_{\text{map}} f_1$ and $\text{lift}_{\text{map}} f_2$:

$$\begin{aligned}
&\text{lift}_{\text{map}} (f_1 \lll_{\perp} f_2) A \\
&\equiv \text{let } A_1 := (\text{domain}_{\perp} f_1 A) \\
&\quad A_2 := (\text{domain}_{\perp} f_2 A) \\
&\quad A' := A_1 \cap A_2 \\
&\quad \text{in } \lambda a \in A'. \langle f_1 a, f_2 a \rangle \\
&\equiv \text{let } g_1 := \text{lift}_{\text{map}} f_1 A \\
&\quad g_2 := \text{lift}_{\text{map}} f_2 A \\
&\quad A' := (\text{domain } g_1) \cap (\text{domain } g_2) \\
&\quad \text{in } \lambda a \in A'. \langle g_1 a, g_2 a \rangle \\
&\equiv \langle \text{lift}_{\text{map}} f_1 A, \text{lift}_{\text{map}} f_2 A \rangle_{\text{map}}
\end{aligned} \tag{25}$$

Substituting g_1 for $\text{lift}_{\text{map}} f_1$ and g_2 for $\text{lift}_{\text{map}} f_2$ gives a definition for (\lll_{map}) (Figure 4) for which (17) holds.

4.1.2 Case: Composition

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

XXX: include it?

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 $f_1 a \neq \perp$:

$$\begin{aligned}
&\text{lift}_{\text{map}} (\text{ifte}_{\perp} f_1 f_2 f_3) A \\
&\equiv \text{let } f := \lambda a. \text{case } f_1 a \\
&\quad \text{true} \Rightarrow f_2 a \\
&\quad \text{false} \Rightarrow f_3 a \\
&\quad \text{else} \Rightarrow \perp \\
&\quad \text{in } \text{mapping } f (\text{domain}_{\perp} f A) \\
&\equiv \text{let } g_1 := \text{mapping } f A \\
&\quad A_2 := \text{preimage } g_1 \{\text{true}\} \\
&\quad A_3 := \text{preimage } g_1 \{\text{false}\} \\
&\quad f := \lambda a. \text{if } (f_1 a) (f_2 a) (f_3 a) \\
&\quad \text{in } \text{mapping } f (\text{domain}_{\perp} f (A_2 \uplus A_3))
\end{aligned} \tag{26}$$

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

$$\begin{aligned}
& \text{lift}_{\text{map}} (\text{ifte}_{\perp} f_1 f_2 f_3) A \\
& \equiv \text{let } g_1 := \text{lift}_{\text{map}} f_1 A \\
& \quad g_2 := \text{lift}_{\text{map}} f_2 (\text{preimage } g_1 \{ \text{true} \}) \\
& \quad g_3 := \text{lift}_{\text{map}} f_3 (\text{preimage } g_1 \{ \text{false} \}) \\
& \quad A' := (\text{domain } g_2) \uplus (\text{domain } g_3) \\
& \quad \text{in } \lambda a \in A'. \text{if } (a \in \text{domain } g_2) (g_2 a) (g_3 a) \\
& \equiv \text{let } g_1 := \text{lift}_{\text{map}} f_1 A \\
& \quad g_2 := \text{lift}_{\text{map}} f_2 (\text{preimage } g_1 \{ \text{true} \}) \\
& \quad g_3 := \text{lift}_{\text{map}} f_3 (\text{preimage } g_1 \{ \text{false} \}) \\
& \quad \text{in } g_2 \uplus_{\text{map}} g_3
\end{aligned} \tag{27}$$

Substituting g_1 for $\text{lift}_{\text{map}} f_1$, g_2 for $\text{lift}_{\text{map}} f_2$, and g_3 for $\text{lift}_{\text{map}} f_3$ gives a definition for ifte_{map} (Figure 4) for which (18) holds.

4.1.4 Case: Laziness

Starting with the left side of (19), we first expand definitions:

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

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 $A \neq \emptyset$, then $\text{domain}_{\perp} (\lambda a. f 0 a) A \equiv \text{domain}_{\perp} (f 0) A$. Further, it diverges if and only if $\text{mapping } (f 0) A'$ diverges. Therefore, if $A \neq \emptyset$, we can replace $\lambda a. f 0 a$ with $f 0$. If $A = \emptyset$, then $\text{lift}_{\text{map}} (\text{lazy}_{\perp} f) A = \emptyset$ (the empty mapping), so

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

Substituting $g 0$ for $\text{lift}_{\text{map}} (f 0)$ gives a definition for lazy_{map} (Figure 4) for which (19) holds.

4.2 Correctness

Theorem 4.3 (mapping arrow correctness). *lift_{map} is an arrow homomorphism.*

Proof. By construction. \square

Corollary 4.4 (semantic correctness). *For all programs e , $\llbracket e \rrbracket_{\text{map}} \equiv \text{lift}_{\text{map}} \llbracket e \rrbracket_{\perp}$.*

5. Lazy Preimage Mappings

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

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

$$X \xrightarrow{\text{pre}} Y ::= \langle \text{Set } Y, \text{Set } Y \Rightarrow \text{Set } X \rangle \tag{28}$$

Like a mapping, an $X \xrightarrow{\text{pre}} Y$ 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**:

$$\begin{aligned}
& \text{pre} : (X \rightarrow Y) \Rightarrow (X \xrightarrow{\text{pre}} Y) \\
& \text{pre } g := \text{let } Y' := \text{range } g \\
& \quad p := \lambda B. \text{preimage } g B \\
& \quad \text{in } \langle Y', p \rangle
\end{aligned} \tag{29}$$

Applying a preimage mapping to any subset of its codomain:

$$\begin{aligned}
& \text{pre-ap} : (X \xrightarrow{\text{pre}} Y) \Rightarrow \text{Set } Y \Rightarrow \text{Set } X \\
& \text{pre-ap } \langle Y', p \rangle B := p (B \cap Y')
\end{aligned} \tag{30}$$

The necessary property here is that using **pre-ap** to compute preimages is the same as computing them from a mapping using **preimage**.

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

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

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

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 $h_1 : X \xrightarrow{\text{pre}} Y$ and $h_2 : X \xrightarrow{\text{pre}} Y$ are equivalent, or $h_1 \equiv h_2$, when $\text{pre-ap } h_1 B \equiv \text{pre-ap } h_2 B$ for all $B \subseteq Y$.*

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

5.0.1 Preimage Mapping Pairing

Lemma 5.4 (preimage distributes over $\langle \cdot, \cdot \rangle_{\text{map}}$ and (\times)). *Let $g_1 \in X \rightarrow Y_1$ and $g_2 \in X \rightarrow Y_2$. For all $B_1 \subseteq Y_1$ and $B_2 \subseteq Y_2$, $\text{preimage } \langle g_1, g_2 \rangle_{\text{map}} (B_1 \times B_2) = (\text{preimage } g_1 B_1) \cap (\text{preimage } g_2 B_2)$.*

Theorem 5.5 (pre distributes over $\langle \cdot, \cdot \rangle_{\text{map}}$). *Let $g_1 \in X \rightarrow Y_1$ and $g_2 \in X \rightarrow Y_2$. Then $\text{pre } \langle g_1, g_2 \rangle_{\text{map}} \equiv \langle \text{pre } g_1, \text{pre } g_2 \rangle_{\text{pre}}$.*

$$\begin{array}{ll}
X \xrightarrow{\text{pre}} Y ::= \langle \text{Set } Y, \text{Set } Y \Rightarrow \text{Set } X \rangle & \langle \cdot, \cdot \rangle_{\text{pre}} : (X \xrightarrow{\text{pre}} Y_1) \Rightarrow (X \xrightarrow{\text{pre}} Y_2) \Rightarrow (X \xrightarrow{\text{pre}} Y_1 \times Y_2) \\
\text{pre} : (X \xrightarrow{\text{map}} Y) \Rightarrow (X \xrightarrow{\text{pre}} Y) & \langle \langle Y'_1, p_1 \rangle, \langle Y'_2, p_2 \rangle \rangle_{\text{pre}} := \text{let } Y' := Y'_1 \times Y'_2 \\
\text{pre } g := \langle \text{range } g, \lambda B. \text{preimage } g \ B \rangle & \quad p := \lambda B. \bigcup_{\langle b_1, b_2 \rangle \in B} (p_1 \{b_1\}) \cap (p_2 \{b_2\}) \\
& \quad \text{in } \langle Y', p \rangle \\
\text{pre-ap} : (X \xrightarrow{\text{pre}} Y) \Rightarrow \text{Set } Y \Rightarrow \text{Set } X & (\circ_{\text{pre}}) : (Y \xrightarrow{\text{pre}} Z) \Rightarrow (X \xrightarrow{\text{pre}} Y) \Rightarrow (X \xrightarrow{\text{pre}} Z) \\
\text{pre-ap } \langle Y', p \rangle \ B := p \ (B \cap Y') & \langle Z', p_2 \rangle \circ_{\text{pre}} h_1 := \langle Z', \lambda C. \text{pre-ap } h_1 \ (p_2 \ C) \rangle \\
\text{pre-range} : (X \xrightarrow{\text{pre}} Y) \Rightarrow \text{Set } Y & (\uplus_{\text{pre}}) : (X \xrightarrow{\text{pre}} Y) \Rightarrow (X \xrightarrow{\text{pre}} Y) \Rightarrow (X \xrightarrow{\text{pre}} Y) \\
\text{pre-range} := \text{fst} & h_1 \uplus_{\text{pre}} h_2 := \text{let } Y' := (\text{pre-range } h_1) \cup (\text{pre-range } h_2) \\
& \quad p := \lambda B. (\text{pre-ap } h_1 \ B) \uplus (\text{pre-ap } h_2 \ B) \\
& \quad \text{in } \langle Y', p \rangle
\end{array}$$

Figure 5: Lazy preimage mappings and operations.

Proof. Let $\langle Y'_1, p_1 \rangle := \text{pre } g_1$ and $\langle Y'_2, p_2 \rangle := \text{pre } g_2$. Starting from the right side, for all $B \in Y_1 \times Y_2$,

$$\begin{aligned}
& \text{pre-ap } \langle \text{pre } g_1, \text{pre } g_2 \rangle_{\text{pre}} \ B \\
& \equiv \text{let } Y' := Y'_1 \times Y'_2 \\
& \quad p := \lambda B. \bigcup_{\langle y_1, y_2 \rangle \in B} (p_1 \{y_1\}) \cap (p_2 \{y_2\}) \\
& \quad \text{in } p \ (B \cap Y') \\
& \equiv \bigcup_{\langle y_1, y_2 \rangle \in B \cap (Y'_1 \times Y'_2)} (p_1 \{y_1\}) \cap (p_2 \{y_2\}) \\
& \equiv \bigcup_{\langle y_1, y_2 \rangle \in B \cap (Y'_1 \times Y'_2)} (\text{preimage } g_1 \ \{y_1\}) \cap (\text{preimage } g_2 \ \{y_2\}) \\
& \equiv \bigcup_{y \in B \cap (Y'_1 \times Y'_2)} (\text{preimage } \langle g_1, g_2 \rangle_{\text{map}} \ \{y\}) \\
& \equiv \text{preimage } \langle g_1, g_2 \rangle_{\text{map}} \ (B \cap (Y'_1 \times Y'_2)) \\
& \equiv \text{preimage } \langle g_1, g_2 \rangle_{\text{map}} \ B \\
& \equiv \text{pre-ap } (\text{pre } \langle g_1, g_2 \rangle_{\text{map}}) \ B
\end{aligned}$$

□

5.0.2 Preimage Mapping Composition

Lemma 5.6 (preimage distributes over (\circ_{map})). *Let $g_1 \in X \rightarrow Y$ and $g_2 \in Y \rightarrow Z$. For all $C \subseteq Z$, $\text{preimage } (g_2 \circ_{\text{map}} g_1) \ C = \text{preimage } g_1 \ (\text{preimage } g_2 \ C)$.*

Theorem 5.7 (pre distributes over (\circ_{map})). *Let $g_1 \in X \rightarrow Y$ and $g_2 \in Y \rightarrow Z$. Then $\text{pre } (g_2 \circ_{\text{map}} g_1) \equiv (\text{pre } g_2) \circ_{\text{pre}} (\text{pre } g_1)$.*

Proof. Let $\langle Z', p_2 \rangle := \text{pre } g_2$. Starting from the right side, for all $C \subseteq Z$,

$$\begin{aligned}
& \text{pre-ap } ((\text{pre } g_2) \circ_{\text{pre}} (\text{pre } g_1)) \ C \\
& \equiv \text{let } h := \lambda C. \text{pre-ap } (\text{pre } g_1) \ (p_2 \ C) \\
& \quad \text{in } h \ (C \cap Z') \\
& \equiv \text{pre-ap } (\text{pre } g_1) \ (p_2 \ (C \cap Z')) \\
& \equiv \text{pre-ap } (\text{pre } g_1) \ (\text{pre-ap } (\text{pre } g_2) \ C) \\
& \equiv \text{preimage } g_1 \ (\text{preimage } g_2 \ C) \\
& \equiv \text{preimage } (g_2 \circ_{\text{map}} g_1) \ C \\
& \equiv \text{pre-ap } (\text{pre } (g_2 \circ_{\text{map}} g_1)) \ C
\end{aligned}$$

□

5.0.3 Preimage Mapping Disjoint Union

Lemma 5.8 (preimage distributes over (\uplus_{map})). *Let $g_1 \in X \rightarrow Y$ and $g_2 \in X \rightarrow Y$ be disjoint mappings. For all $B \subseteq Y$, $\text{preimage } (g_1 \uplus_{\text{map}} g_2) \ B = (\text{preimage } g_1 \ B) \uplus (\text{preimage } g_2 \ B)$.*

Theorem 5.9 (pre distributes over (\uplus_{map})). *Let $g_1 \in X \rightarrow Y$ and $g_2 \in X \rightarrow Y$ have disjoint domains. Then $\text{pre } (g_1 \uplus_{\text{map}} g_2) \equiv (\text{pre } g_1) \uplus_{\text{pre}} (\text{pre } g_2)$.*

Proof. Let $Y'_1 := \text{range } g_1$ and $Y'_2 := \text{range } g_2$. Starting from the right side, for all $B \subseteq Y$,

$$\begin{aligned}
& \text{pre-ap } ((\text{pre } g_1) \uplus_{\text{pre}} (\text{pre } g_2)) \ B \\
& \equiv \text{let } Y' := Y'_1 \cup Y'_2 \\
& \quad h := \lambda B. (\text{pre-ap } (\text{pre } g_1) \ B) \uplus (\text{pre-ap } (\text{pre } g_2) \ B) \\
& \quad \text{in } h \ (B \cap Y') \\
& \equiv (\text{pre-ap } (\text{pre } g_1) \ (B \cap (Y'_1 \cup Y'_2))) \uplus \\
& \quad (\text{pre-ap } (\text{pre } g_2) \ (B \cap (Y'_1 \cup Y'_2))) \\
& \equiv (\text{preimage } g_1 \ (B \cap (Y'_1 \cup Y'_2))) \uplus \\
& \quad (\text{preimage } g_2 \ (B \cap (Y'_1 \cup Y'_2))) \\
& \equiv \text{preimage } (g_1 \uplus_{\text{map}} g_2) \ (B \cap (Y'_1 \cup Y'_2)) \\
& \equiv \text{preimage } (g_1 \uplus_{\text{map}} g_2) \ B \\
& \equiv \text{pre-ap } (\text{pre } (g_1 \uplus_{\text{map}} g_2)) \ B
\end{aligned}$$

□

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 $X \xrightarrow{\text{pre}} Y ::= X \xrightarrow{\text{pre}} Y$: we run into trouble trying to define arr_{pre} because a preimage mapping needs an observable domain. While a preimage mapping's domain is the *range* of the mapping it computes preimages for, it is still easiest to parameterize preimage computations on a $\text{Set } X$:

$$X \xrightarrow{\text{pre}} Y ::= \text{Set } X \Rightarrow (X \xrightarrow{\text{pre}} Y) \quad (31)$$

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

To use Theorem 3.3, we need to define correctness using a lift from the mapping arrow to the preimage arrow:

$$\begin{aligned} \text{lift}_{\text{pre}} : (X \rightsquigarrow_{\text{map}} Y) &\Rightarrow (X \rightsquigarrow_{\text{pre}} Y) \\ \text{lift}_{\text{pre}} g A &:= \text{pre} (g A) \end{aligned} \quad (32)$$

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

$$\text{pre-ap} (\text{lift}_{\text{pre}} g A) B \equiv \text{preimage} (g A) B \quad (33)$$

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

We also need a coarser notion of equivalence.

Definition 6.1 (Preimage arrow equivalence). *Two preimage arrow computations $h_1 : X \rightsquigarrow_{\text{pre}} Y$ and $h_2 : X \rightsquigarrow_{\text{pre}} Y$ are equivalent, or $h_1 \equiv h_2$, when $h_1 A \equiv h_2 A$ for all $A \subseteq X$.*

As with arr_{map} , defining arr_{pre} as a composition meets (15). The following subsections derive $(\&\&\&_{\text{pre}})$, $(\>\>\>_{\text{pre}})$, ifte_{pre} and lazy_{pre} from their corresponding mapping arrow combinators, in a way that ensures lift_{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 lift_{pre} :

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

Substituting h_1 for $\text{lift}_{\text{pre}} g_1$ and h_2 for $\text{lift}_{\text{pre}} g_2$, and removing the application of pre-ap from both sides of the equivalence gives a definition of $(\&\&\&_{\text{pre}})$ (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 lift_{pre} :

$$\begin{aligned} \text{pre-ap} (\text{lift}_{\text{pre}} (g_1 \>\>\>_{\text{map}} g_2) A) C \\ &\equiv \text{let } g'_1 := g_1 A \\ &\quad g'_2 := g_2 (\text{range } g'_1) \\ &\quad \text{in } \text{pre-ap} (\text{pre} (g'_2 \circ_{\text{map}} g'_1)) C \\ &\equiv \text{let } g'_1 := g_1 A \\ &\quad g'_2 := g_2 (\text{range } g'_1) \\ &\quad \text{in } \text{pre-ap} ((\text{pre } g'_1) \circ_{\text{pre}} (\text{pre } g'_2)) C \\ &\equiv \text{let } h_1 := \text{lift}_{\text{pre}} g_1 A \\ &\quad h_2 := \text{lift}_{\text{pre}} g_2 (\text{pre-range } h_1) \\ &\quad \text{in } \text{pre-ap} (h_2 \circ_{\text{pre}} h_1) C \end{aligned} \quad (34)$$

Substituting h_1 for $\text{lift}_{\text{pre}} g_1$ and h_2 for $\text{lift}_{\text{pre}} g_2$, and removing the application of pre-ap from both sides of the equivalence gives a definition of $(\>\>\>_{\text{pre}})$ (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 lift_{pre} , and apply Theo-

rem 5.2 in the definitions of h_2 and h_3 :

$$\begin{aligned} \text{pre-ap} (\text{lift}_{\text{pre}} (\text{ifte}_{\text{map}} g_1 g_2 g_3) A) B \\ &\equiv \text{let } g'_1 := g_1 A \\ &\quad g'_2 := g_2 (\text{preimage } g'_1 \{ \text{true} \}) \\ &\quad g'_3 := g_3 (\text{preimage } g'_1 \{ \text{false} \}) \\ &\quad \text{in } \text{pre-ap} (\text{pre} (g'_2 \uplus_{\text{map}} g'_3)) B \\ &\equiv \text{let } g'_1 := g_1 A \\ &\quad g'_2 := g_2 (\text{preimage } g'_1 \{ \text{true} \}) \\ &\quad g'_3 := g_3 (\text{preimage } g'_1 \{ \text{false} \}) \\ &\quad \text{in } \text{pre-ap} ((\text{pre } g'_2) \uplus_{\text{pre}} (\text{pre } g'_3)) B \\ &\equiv \text{let } h_1 := \text{lift}_{\text{pre}} g_1 A \\ &\quad h_2 := \text{lift}_{\text{pre}} g_2 (\text{pre-ap } h_1 \{ \text{true} \}) \\ &\quad h_3 := \text{lift}_{\text{pre}} g_3 (\text{pre-ap } h_1 \{ \text{false} \}) \\ &\quad \text{in } \text{pre-ap} (h_2 \uplus_{\text{pre}} h_3) B \end{aligned}$$

Substituting h_1 for $\text{lift}_{\text{pre}} g_1$, h_2 for $\text{lift}_{\text{pre}} g_2$ and h_3 for $\text{lift}_{\text{pre}} g_3$, and removing the application of pre-ap from both sides of the equivalence gives a definition of ifte_{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 $\text{lift}_{\text{pre}} (g \ 0)$:

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

Substituting $h \ 0$ for $\text{lift}_{\text{pre}} (g \ 0)$ and removing the application of pre-ap from both sides of the equivalence gives a definition for lazy_{pre} (Figure 6) for which (19) holds.

6.5 Correctness

Theorem 6.2 (preimage arrow correctness). *lift_{pre} is an arrow homomorphism.*

Proof. By construction. \square

Corollary 6.3 (semantic correctness). *For all programs e , $\llbracket e \rrbracket_{\text{pre}} \equiv \text{lift}_{\text{pre}} \llbracket e \rrbracket_{\text{map}}$.*

7. Preimages Under Partial Functions

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 $r \in [0, 1]$ 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 $\text{random} < p$ is false:

$$\text{geometric } p := \text{if } (\text{random} < p) \ 0 \ (1 + \text{geometric } p) \quad (35)$$

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

Suppose we interpret (35) as $h : \mathbb{R} \rightsquigarrow_{\text{pre}} \mathbb{N}$, a preimage arrow computation from random sources in \mathbb{R} to natural numbers, and that we have a probability measure $P \in \mathcal{P} \mathbb{R} \rightarrow [0, 1]$.

$$\begin{aligned}
X \rightsquigarrow_{\text{pre}} Y &::= \text{Set } X \Rightarrow (X \rightrightarrows_{\text{pre}} Y) \\
\text{arr}_{\text{pre}} &: (X \Rightarrow Y) \Rightarrow (X \rightsquigarrow_{\text{pre}} Y) \\
\text{arr}_{\text{pre}} &:= \text{lift}_{\text{pre}} \circ \text{arr}_{\text{map}} \\
(\ggg_{\text{pre}}) &: (X \rightsquigarrow_{\text{pre}} Y) \Rightarrow (Y \rightsquigarrow_{\text{pre}} Z) \Rightarrow (X \rightsquigarrow_{\text{pre}} Z) \\
(h_1 \ggg_{\text{pre}} h_2) A &:= \text{let } h'_1 := h_1 A \\
&\quad h'_2 := h_2 (\text{pre-range } h'_1) \\
&\quad \text{in } h'_2 \circ_{\text{pre}} h'_1 \\
(\&\&\&_{\text{pre}}) &: (X \rightsquigarrow_{\text{pre}} Y) \Rightarrow (X \rightsquigarrow_{\text{pre}} Z) \Rightarrow (X \rightsquigarrow_{\text{pre}} Y \times Z) \\
(h_1 \&\&\&_{\text{pre}} h_2) A &:= \langle h_1 A, h_2 A \rangle_{\text{pre}} \\
\text{ifte}_{\text{pre}} &: (X \rightsquigarrow_{\text{pre}} \text{Bool}) \Rightarrow (X \rightsquigarrow_{\text{pre}} Y) \Rightarrow (X \rightsquigarrow_{\text{pre}} Y) \Rightarrow (X \rightsquigarrow_{\text{pre}} Y) \\
\text{ifte}_{\text{pre}} h_1 h_2 h_3 A &:= \text{let } h'_1 := h_1 A \\
&\quad h'_2 := h_2 (\text{pre-ap } h'_1 \{\text{true}\}) \\
&\quad h'_3 := h_3 (\text{pre-ap } h'_1 \{\text{false}\}) \\
&\quad \text{in } h'_2 \uplus_{\text{pre}} h'_3 \\
\text{lazy}_{\text{pre}} &: (1 \Rightarrow (X \rightsquigarrow_{\text{pre}} Y)) \Rightarrow (X \rightsquigarrow_{\text{pre}} Y) \\
\text{lazy}_{\text{pre}} h A &:= \text{if } (A = \emptyset) (\text{pre } \emptyset) (h \ 0 \ A) \\
\text{lift}_{\text{pre}} &: (X \rightsquigarrow_{\text{map}} Y) \Rightarrow (X \rightsquigarrow_{\text{pre}} Y) \\
\text{lift}_{\text{pre}} g A &:= \text{pre } (g \ A)
\end{aligned}$$

Figure 6: Preimage arrow definitions.

We could compute the probability of any output set $N \subseteq \mathbb{N}$ using $P(h \ R' \ N)$, where $R' \subseteq R$ and $P \ R' = 1$. We have three hurdles to overcome:

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

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

7.1 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 sub-trees to sub-computations.

With either alternative, for arrows defined using pairing, the resulting definitions are large, conceptually difficult, and hard to manipulate. Fortunately, assigning each sub-computation a unique index into a tree-shaped random source, and passing it 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, $j_0 \in J$ a distinguished element, and $\text{left} : J \Rightarrow J$ and $\text{right} : J \Rightarrow J$ be total, injective functions. If for all $j \in J$, $j = \text{next } j_0$ 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 $\{0, 1\}$, $j_0 := \langle \rangle$, and $\text{left } j := \langle 0, j \rangle$ and $\text{right } j := \langle 1, j \rangle$.

Alternatively, let J be the set of dyadic rationals in $(0, 1)$ (i.e. those with power-of-two denominators), $j_0 := \frac{1}{2}$ and

$$\begin{aligned}
\text{left } (p/q) &:= (p - \frac{1}{2})/q \\
\text{right } (p/q) &:= (p + \frac{1}{2})/q
\end{aligned} \tag{36}$$

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.

7.2 Applicative, Associative Store

XXX: arrow transformer: an arrow whose combinators are defined entirely in terms of another arrow

XXX: computations receive an index and return an arrow from a store of type s paired with x , to y :

$$\text{AStore } s \ (x \rightsquigarrow_a y) ::= J \Rightarrow (\langle s, x \rangle \rightsquigarrow_a y) \tag{37}$$

XXX: motivational words for definition of lift:

$$\begin{aligned}
\eta_{a^*} &: (x \rightsquigarrow_a y) \Rightarrow \text{AStore } s \ (x \rightsquigarrow_a y) \\
\eta_{a^*} f \ j &:= \text{arr}_a \ \text{snd} \ \ggg_a f
\end{aligned} \tag{38}$$

Figure 7 defines the **AStore** arrow transformer. As with the other arrows, proving that its lift is a homomorphism allows us to prove that programs interpreted as its computations are correct. Again, to do so, we need to extend equivalence to be more extensional for arrows $\text{AStore } s \ (x \rightsquigarrow_a y)$.

Definition 7.2 (**AStore** arrow equivalence). *Two **AStore** arrow computations k_1 and k_2 are equivalent, or $k_1 \equiv k_2$, when $k_1 \ j \equiv k_2 \ j$ for all $j \in J$.*

XXX: need to define equivalence for the bottom arrow for this to make sense

Theorem 7.3 (**AStore** arrow correctness). *Let $x \rightsquigarrow_{a^*} y ::= \text{AStore } s \ (x \rightsquigarrow_a y)$. Then η_{a^*} is an arrow homomorphism.*

Proof. Defining arr_{a^*} as a composition clearly meets the first homomorphism identity (15).

Composition. Starting with the right side of (16), expand definitions and use $(\text{arr}_a \ f \ \&\&\&_a \ f_1) \ggg_a \text{arr}_a \ \text{snd} \equiv f_1$:

$$\begin{aligned}
(\eta_{a^*} \ f_1 \ggg_{a^*} \ \eta_{a^*} \ f_2) \ j \\
&\equiv (\text{arr}_a \ \text{fst} \ \&\&\&_a \ (\text{arr}_a \ \text{snd} \ \ggg_a \ f_1)) \ggg_a \text{arr}_a \ \text{snd} \ \ggg_a \ f_2 \\
&\equiv \text{arr}_a \ \text{snd} \ \ggg_a \ f_1 \ \ggg_a \ f_2 \\
&\equiv \eta_{a^*} \ (f_1 \ggg_a \ f_2) \ j
\end{aligned}$$

Pairing. Starting with the right side of (17), expand definitions and use the arrow law $\text{arr}_a \ f \ \ggg_a \ (f_1 \ \&\&\&_a \ f_2) \equiv$

$$\begin{array}{ll}
x \rightsquigarrow_{a^*} y ::= \text{AStore } s \ (x \rightsquigarrow_a y) ::= J \Rightarrow (\langle s, x \rangle \rightsquigarrow_a y) & \text{ifte}_{a^*} : (x \rightsquigarrow_{a^*} \text{Bool}) \Rightarrow (x \rightsquigarrow_{a^*} y) \Rightarrow (x \rightsquigarrow_{a^*} y) \Rightarrow (x \rightsquigarrow_{a^*} y) \\
\text{arr}_{a^*} : (x \Rightarrow y) \Rightarrow (x \rightsquigarrow_{a^*} y) & \text{ifte}_{a^*} \ k_1 \ k_2 \ k_3 \ j := \text{ifte}_a \ (k_1 \ (\text{left } j)) \\
\text{arr}_{a^*} := \eta_{a^*} \circ \text{arr}_a & \quad (k_2 \ (\text{left } (\text{right } j))) \\
& \quad (k_3 \ (\text{right } (\text{right } j))) \\
(\ggg_{a^*}) : (x \rightsquigarrow_{a^*} y) \Rightarrow (y \rightsquigarrow_{a^*} z) \Rightarrow (x \rightsquigarrow_{a^*} z) & \text{lazy}_{a^*} : (1 \Rightarrow (x \rightsquigarrow_{a^*} y)) \Rightarrow (x \rightsquigarrow_{a^*} y) \\
(k_1 \ggg_{a^*} k_2) \ j := & \text{lazy}_{a^*} \ k \ j := \text{lazy}_a \ \lambda 0. k \ 0 \ j \\
(\text{arr}_a \text{ fst } \&\&\&_a \ k_1 \ (\text{left } j)) \ggg_{a^*} k_2 \ (\text{right } j) & \\
(\&\&\&_{a^*}) : (x \rightsquigarrow_{a^*} y_1) \Rightarrow (x \rightsquigarrow_{a^*} y_2) \Rightarrow (x \rightsquigarrow_{a^*} \langle y_1, y_2 \rangle) & \\
(k_1 \&\&\&_{a^*} k_2) \ j := k_1 \ (\text{left } j) \&\&\&_a k_2 \ (\text{right } j) & \\
\eta_{a^*} : (x \rightsquigarrow_a y) \Rightarrow (x \rightsquigarrow_{a^*} y) & \\
\eta_{a^*} \ f \ j := \text{arr}_a \ \text{snd } \ggg_{a^*} f &
\end{array}$$

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

$$\begin{aligned}
& (\text{arr}_a \ f \ \ggg_a \ f_1) \ \&\&\&_a \ (\text{arr}_a \ f \ \ggg_a \ f_2) : \\
& \quad (\eta_{a^*} \ f_1 \ \&\&\&_{a^*} \ \eta_{a^*} \ f_2) \ j \\
& \quad \equiv (\text{arr}_a \ \text{snd } \ggg_a \ f_1) \ \&\&\&_a \ (\text{arr}_a \ \text{snd } \ggg_a \ f_2) \\
& \quad \equiv \text{arr}_a \ \text{snd } \ggg_a \ (f_1 \ \&\&\&_a \ f_2) \\
& \quad \equiv \eta_{a^*} \ (f_1 \ \&\&\&_a \ f_2) \ j
\end{aligned}$$

Conditional. Starting with the right side of (18), expand definitions and use the arrow law $\text{arr}_a \ f \ \ggg_a \ \text{ifte}_a \ f_1 \ f_2 \ f_3 \equiv \text{ifte}_a \ (\text{arr}_a \ f \ \ggg_a \ f_1) \ (\text{arr}_a \ f \ \ggg_a \ f_2) \ (\text{arr}_a \ f \ \ggg_a \ f_3)$:

$$\begin{aligned}
& (\text{ifte}_{a^*} \ (\eta_{a^*} \ f_1) \ (\eta_{a^*} \ f_2) \ (\eta_{a^*} \ f_3)) \ j \\
& \equiv \text{ifte}_a \ (\text{arr}_a \ \text{snd } \ggg_a \ f_1) \\
& \quad (\text{arr}_a \ \text{snd } \ggg_a \ f_2) \\
& \quad (\text{arr}_a \ \text{snd } \ggg_a \ f_3) \\
& \equiv \text{arr}_a \ \text{snd } \ggg_a \ (\text{ifte}_a \ f_1 \ f_2 \ f_3) \\
& \equiv \eta_{a^*} \ (\text{ifte}_a \ f_1 \ f_2 \ f_3) \ j
\end{aligned}$$

Laziness. Starting with the right side of (19), expand definitions, β -expand within the outer thunk, and use the arrow law $\text{arr}_a \ f \ \ggg_a \ \text{lazy}_a \ f_1 \equiv \text{lazy}_a \ \lambda 0. \text{arr}_a \ f \ \ggg_a \ f_1 \ 0$:

$$\begin{aligned}
& (\text{lazy}_{a^*} \ \lambda 0. \eta_{a^*} \ (f \ 0)) \ j \\
& \equiv \text{lazy}_a \ \lambda 0. (\lambda 0. \lambda j. \text{arr}_a \ \text{snd } \ggg_a \ f \ 0) \ 0 \ j \\
& \equiv \text{lazy}_a \ \lambda 0. \text{arr}_a \ \text{snd } \ggg_a \ f \ 0 \\
& \equiv \text{arr}_a \ \text{snd } \ggg_a \ \text{lazy}_a \ f \\
& \equiv \eta_{a^*} \ (\text{lazy}_a \ f) \ j
\end{aligned}$$

XXX: all of these rely on arrow laws that aren't proved for the mapping and preimage arrows \square

Corollary 7.4 (semantic correctness). *Let $x \rightsquigarrow_{a^*} y ::= \text{AStore } s \ (x \rightsquigarrow_a y)$. If $\llbracket e \rrbracket_a : x \rightsquigarrow_a y$, then $\eta_{a^*} \llbracket e \rrbracket_a \equiv \llbracket e \rrbracket_{a^*}$ and $\llbracket e \rrbracket_{a^*} : x \rightsquigarrow_{a^*} y$.*

In particular, Corollary 7.4 implies that, if a pure let-calculus expression is interpreted as a computation $k : \text{AStore } S \ (X \rightsquigarrow_{\text{pre}} Y)$, then $k \ j_0$ correctly computes preimages. We still need to know that preimages under functions that access the store are computed correctly, which we will get to after defining stores and combinators that access them.

7.3 Probabilistic Programs

Definition 7.5 (random source). *Let $R := J \rightarrow [0, 1]$. A **random source** is a total mapping $r \in R$; equivalently, an infinite vector of random numbers indexed by J .*

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

$$\begin{aligned}
& \text{random}_{a^*} : x \rightsquigarrow_{a^*} [0, 1] \\
& \text{random}_{a^*} \ j := \text{arr}_a \ (\text{fst } \ggg \ \pi \ j)
\end{aligned} \tag{39}$$

We extend the let-calculus semantic function with

$$\llbracket \text{random} \rrbracket_{a^*} := \text{random}_{a^*} \tag{40}$$

for arrows a^* for which random_{a^*} is defined.

7.4 Partial Programs

The most effective and ultimately implementable way we have found to avoid divergence in computing preimages is to use the store to dictate which branch of each conditional, if any, is allowed to be taken.

Definition 7.6 (branch trace). *A **branch trace** is a total mapping (i.e. vector) $t \in J \rightarrow \text{Bool}_\perp$ such that $t \ j = \text{true}$ or $t \ j = \text{false}$ for no more than finitely many $j \in J$.*

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

$$\begin{aligned}
& \text{branch}_{a^*} : x \rightsquigarrow_{a^*} \text{Bool} \\
& \text{branch}_{a^*} \ j := \text{arr}_a \ (\text{fst } \ggg \ \pi \ j)
\end{aligned} \tag{41}$$

Using branch_{a^*} , we define an additional if-then-else combinator, which ensures its conditional expression agrees with the branch trace:

$$\begin{aligned}
& \text{agrees} : \langle \text{Bool}, \text{Bool} \rangle \Rightarrow \text{Bool}_\perp \\
& \text{agrees} \ \langle b_1, b_2 \rangle := \text{if } (b_1 = b_2) \ b_1 \ \perp
\end{aligned} \tag{42}$$

$$\begin{aligned}
& \text{ifte}'_{a^*} : (x \rightsquigarrow_{a^*} \text{Bool}) \Rightarrow (x \rightsquigarrow_{a^*} y) \Rightarrow (x \rightsquigarrow_{a^*} y) \Rightarrow (x \rightsquigarrow_{a^*} y) \\
& \text{ifte}'_{a^*} \ k_1 \ k_2 \ k_3 := \\
& \quad \text{ifte}_{a^*} \ ((k_1 \ \&\&\&_{a^*} \ \text{branch}_{a^*}) \ \ggg_{a^*} \ \text{arr}_{a^*} \ \text{agrees}) \ k_2 \ k_3
\end{aligned} \tag{43}$$

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

Every computation defined using the let-calculus semantic function $\llbracket \cdot \rrbracket_a$, 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 ifte_{a^*} with ifte'_{a^*} to ensure *computations* always converge. Therefore, we can define a semantic function $\llbracket \cdot \rrbracket'_a$ for let-

calculus programs whose computations always converge by overriding only the if rule:

$$\begin{aligned} \llbracket \text{if } e_c \ e_t \ e_f \rrbracket_{a^*}' &::= \text{ifte}_{a^*}' \left(\begin{array}{l} \llbracket e_c \rrbracket_{a^*} \\ (\text{lazy}_a \ \lambda 0. \llbracket e_t \rrbracket_{a^*}) \\ (\text{lazy}_a \ \lambda 0. \llbracket e_f \rrbracket_{a^*}) \end{array} \right) \\ \llbracket e \rrbracket_{a^*}' &::= \llbracket e \rrbracket_{a^*} \end{aligned} \quad (44)$$

7.5 Partial, Probabilistic Programs

Let $S ::= R \times T$ and $x \rightsquigarrow_{a^*} y ::= \text{AStore } S \ (x \rightsquigarrow_a y)$, and update the random_{a^*} and branch_{a^*} combinators to reflect that the store is now a pair:

$$\begin{aligned} \text{random}_{a^*} &: x \rightsquigarrow_{a^*} [0, 1] \\ \text{random}_{a^*} \ j &:= \text{arr}_a \ (\text{fst} \gg \text{fst} \gg \pi \ j) \end{aligned} \quad (45)$$

$$\begin{aligned} \text{branch}_{a^*} &: x \rightsquigarrow_{a^*} \text{Bool} \\ \text{branch}_{a^*} \ j &:= \text{arr}_a \ (\text{fst} \gg \text{snd} \gg \pi \ j) \end{aligned} \quad (46)$$

The ifte_{a^*}' combinator's definition remains the same.

7.6 Correctness

Theorem 7.7 (natural transformation). *Let $x \rightsquigarrow_{a^*} y ::= \text{AStore } s \ (x \rightsquigarrow_a y)$ and $x \rightsquigarrow_{b^*} y ::= \text{AStore } s \ (x \rightsquigarrow_b y)$. Let $\text{lift}_b : (x \rightsquigarrow_a y) \Rightarrow (x \rightsquigarrow_b y)$ be an arrow homomorphism, and*

$$\begin{aligned} \text{lift}_{b^*} &: (x \rightsquigarrow_{a^*} y) \Rightarrow (x \rightsquigarrow_{b^*} y) \\ \text{lift}_{b^*} \ f \ j &:= \text{lift}_b \ (f \ j) \end{aligned} \quad (47)$$

The following diagram commutes:

$$\begin{array}{ccc} x \rightsquigarrow_a y & \xrightarrow{\text{lift}_b} & x \rightsquigarrow_b y \\ \eta_{a^*} \downarrow & & \downarrow \eta_{b^*} \\ x \rightsquigarrow_{a^*} y & \xrightarrow{\text{lift}_{b^*}} & x \rightsquigarrow_{b^*} y \end{array} \quad (48)$$

i.e. for all $f : x \rightsquigarrow_a y$, $\eta_{b^*} (\text{lift}_b \ f) \equiv \text{lift}_{b^*} \ (\eta_{a^*} \ f)$. Further, lift_{b^*} is an arrow homomorphism.

Proof. Starting from the right side of the equivalence, expand definitions and apply homomorphism identities (16) and (15) for lift_b :

$$\begin{aligned} \text{lift}_{b^*} \ (\eta_{a^*} \ f) &\equiv \lambda j. \text{lift}_b \ (\text{arr}_a \ \text{snd} \gg \gg_a \ f) \\ &\equiv \lambda j. \text{lift}_b \ (\text{arr}_a \ \text{snd}) \gg \gg_b \ \text{lift}_b \ f \\ &\equiv \lambda j. \text{arr}_b \ \text{snd} \gg \gg_b \ \text{lift}_b \ f \\ &\equiv \eta_{b^*} \ (\text{lift}_b \ f) \end{aligned}$$

Further, because η_{a^*} , η_{b^*} , and lift_b are homomorphisms, lift_{b^*} is a homomorphism by composition.

XXX: not sure I'm allowed to invoke converse of composition of homomorphisms without extra conditions \square

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

Corollary 7.8 (mapping* and preimage* arrow correctness). *The following diagram commutes:*

$$\begin{array}{ccccc} X \rightsquigarrow_{\perp} Y & \xrightarrow{\text{lift}_{\text{map}}} & X \rightsquigarrow_{\text{map}} Y & \xrightarrow{\text{lift}_{\text{pre}}} & X \rightsquigarrow_{\text{pre}} Y \\ \eta_{\perp^*} \downarrow & & \downarrow \eta_{\text{map}^*} & & \downarrow \eta_{\text{pre}^*} \\ X \rightsquigarrow_{\perp^*} Y & \xrightarrow{\text{lift}_{\text{map}^*}} & X \rightsquigarrow_{\text{map}^*} Y & \xrightarrow{\text{lift}_{\text{pre}^*}} & X \rightsquigarrow_{\text{pre}^*} Y \end{array} \quad (49)$$

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

Corollary 7.9 (semantic correctness). *If $\llbracket e \rrbracket_{\perp^*} : X \rightsquigarrow_{\perp^*} Y$, then $\text{lift}_{\text{map}^*} \ \llbracket e \rrbracket_{\perp^*} \equiv \llbracket e \rrbracket_{\text{map}^*}$ and $\text{lift}_{\text{pre}^*} \ \llbracket e \rrbracket_{\text{map}^*} \equiv \llbracket e \rrbracket_{\text{pre}^*}$.*

Corollary 7.10 (semantic' correctness). *If $\llbracket e \rrbracket_{\perp^*}' : X \rightsquigarrow_{\perp^*} Y$, then $\text{lift}_{\text{map}^*} \ \llbracket e \rrbracket_{\perp^*}' \equiv \llbracket e \rrbracket_{\text{map}^*}'$ and $\text{lift}_{\text{pre}^*} \ \llbracket e \rrbracket_{\text{map}^*}' \equiv \llbracket e \rrbracket_{\text{pre}^*}'$.*

In particular, $\llbracket e \rrbracket_{\text{pre}^*}$ and $\llbracket e \rrbracket_{\text{pre}^*}'$ correctly compute preimages under the interpretation of e as a function from an implicit random source. We will make stronger statements about $\llbracket \cdot \rrbracket_{\text{pre}^*}$ after proving its computations always converge.

Theorem 7.11 (divergence implies error). *Let $f := \llbracket e \rrbracket_{\perp^*}$ and $f' := \llbracket e \rrbracket_{\perp^*}'$ converge, where $f : x \rightsquigarrow_{\perp^*} y$. For all $r \in R$, $t \in T$ and $a : X$,*

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

Proof. Let $m : J \Rightarrow J$ invertibly map sub-computation indexes in f to corresponding sub-computation indexes in f' . (Defining m formally is tedious and unilluminating.)

Case 1. Define $t' \in J \rightarrow \text{Bool}_{\perp}$ such that $t' \ j = z$ if the sub-computation with index $m^{-1} \ j$ in f is an if condition that returns z , otherwise $t' \ j = \perp$. Because $f \ \langle \langle r, t \rangle, a \rangle$ converges, $t' \ j \neq \perp$ for at most finitely many j , so $t' \in T$. Exists t' .

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

To compare preimages computed by arrow instances produced by $\llbracket \cdot \rrbracket_{\text{pre}^*}$ and $\llbracket \cdot \rrbracket_{\text{pre}^*}'$, we need a set of inputs on which they should obviously always agree.

Definition 7.12 (halting set). *A computation's halting set is the largest $A^* \subseteq (R \times T) \times X$ for which*

- *For $f : X \rightsquigarrow_{\perp^*} Y$, $f \ j_0 \ x \neq \perp$ for all $x \in A^*$.*
- *For $g : X \rightsquigarrow_{\text{map}^*} Y$, $\text{domain} \ (g \ j_0 \ A^*) = A^*$.*
- *For $h : X \rightsquigarrow_{\text{pre}^*} Y$, $\text{pre-ap} \ (h \ j_0 \ A^*) \ Y = A^*$.*

Recall truth statements like $f \ j_0 \ x \neq \perp$ imply convergence.

That $\text{lift}_{\text{map}^*}$ and $\text{lift}_{\text{pre}^*}$ are arrow homomorphisms allows transporting halting set definitions and theorems between arrow types.

Theorem 7.13 (halting set equality). *Let $f : X \rightsquigarrow_{\perp^*} Y$, and $g : X \rightsquigarrow_{\text{map}^*} Y$ and $h : X \rightsquigarrow_{\text{pre}^*} Y$ such that $g \equiv \text{lift}_{\text{map}^*} \ f$ and $h \equiv \text{lift}_{\text{pre}^*} \ g$. Then f , g and h have the same halting set.*

Proof. XXX: do this \square

Corollary 7.14 (computed halting set). *Let $\llbracket e \rrbracket_{\perp^*} : X \rightsquigarrow_{\perp^*} Y$ converge. Then $A^* = \text{domain} \ (\llbracket e \rrbracket_{\text{map}^*}' \ j_0 \ ((R \times T) \times X))$.*

Corollary 7.15 (semantic correctness (final)). *Let $\llbracket e \rrbracket_{\perp^*} : X \rightsquigarrow_{\perp^*} Y$ converge and have halting set A^* . For all $A \subseteq (R \times T) \times X$ and $B \subseteq Y$, $\text{pre-ap} \ (\llbracket e \rrbracket_{\text{pre}^*}' \ j_0 \ A) \ B = \text{preimage} \ (\llbracket e \rrbracket_{\text{map}^*}' \ j_0 \ (A \cap A^*)) \ B$.*

In other words, preimages computed using $\llbracket \cdot \rrbracket_{\text{pre}^*}'$ 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 $g : X \rightarrow Y$, the probability of an output set $B \subseteq Y$ is

$$P(\text{preimage } g B) \quad (50)$$

where $P \in \mathcal{P} X \rightarrow [0,1]$ 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: $\text{domain } P$ 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 concepts in measure theory can be understood by analogy to topology.

The analogue of a topology is a σ -algebra.

Definition 8.1 (σ -algebra, measurable set). *A collection of sets $\mathcal{A} \subseteq \mathcal{P} X$ 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**.*

$X \setminus X = \emptyset$, so $\emptyset \in \mathcal{A}$. 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 $g : X \rightarrow Y$ is **\mathcal{A} - \mathcal{B} -measurable** if for all $B \in \mathcal{B}$, $\text{preimage } g B \in \mathcal{A}$.*

Measurability is usually a weaker condition than continuity. For example, with respect to the σ -algebra generated from \mathbb{R} 's standard topology, measurable $\mathbb{R} \rightarrow \mathbb{R}$ 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 $X := \langle X_1, X_2 \rangle$. The **product σ -algebra** $\mathcal{A}_1 \otimes \mathcal{A}_2$ is the smallest σ -algebra for which mapping $\text{fst } X$ and mapping $\text{snd } X$ are measurable.*

Definition 8.4 (arbitrary product σ -algebra). *Let \mathcal{A} be a σ -algebra on X . The **product σ -algebra** $\mathcal{A}^{\otimes J}$ is the smallest σ -algebra for which, for all $j \in J$, mapping $(\pi j) (J \rightarrow X)$ is measurable.*

8.1 Measurable Pure Computations

It is easier to prove measurability of pure computations than to prove measurability of probabilistic ones. Further, we can use the resulting theorems to prove that all probabilistic programs are measurable.

A single mapping arrow computation can produce many mappings, which, it seems, could complicate proving measurability. Fortunately, we need only consider the mapping produced by applying a computation to its halting set.

Definition 8.5 (halting set). *Let $g : X \xrightarrow{\text{map}} Y$. Its **halting set** is the largest $A^* \subseteq X$ for which $\text{domain } (g A^*) = A^*$.*

Definition 8.6 (measurable mapping arrow computation). *Let \mathcal{A} and \mathcal{B} be σ -algebras on X and Y . A computation $g : X \xrightarrow{\text{map}} Y$ is **\mathcal{A} - \mathcal{B} -measurable** if $g A^*$ is an \mathcal{A} - \mathcal{B} -measurable mapping, where A^* is g 's halting set.*

The definition of halting set implies $\text{preimage } (g A^*) Y = A^*$. Because $Y \in \mathcal{B}$, $A^* \in \mathcal{A}$.

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

Theorem 8.8. *Let $g : X \xrightarrow{\text{map}} Y$ be an \mathcal{A} - \mathcal{B} -measurable mapping arrow computation. Then for all $A \in \mathcal{A}$, $g A$ is an \mathcal{A} - \mathcal{B} -measurable mapping.*

Proof. Use the mapping arrow restriction law (21) and Lemma 8.7. \square

Roughly, if the largest mapping that can be produced by a computation is measurable, any mapping it can produce that we care about is measurable.

That all programs interpreted by $\llbracket \cdot \rrbracket_a$ are measurable will be proved by structural induction on terms. We therefore need a case for each arrow combinator.

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.

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

Lemma 8.10 (expanded post-composition). *Let $g_1 : X \rightarrow Y$ and $g_2 : Y \rightarrow Z$ such that $\text{range } g_1 \subseteq \text{domain } g_2$, and let $g'_2 : Y \rightarrow Z$ such that $g_2 \subseteq g'_2$. Then $g_2 \circ_{\text{map}} g_1 = g'_2 \circ_{\text{map}} g_1$.*

Theorem 8.11 (mapping arrow monotonicity). *Let $g : X \xrightarrow{\text{map}} Y$. For any $A \subseteq X$, $\text{domain } (g A) \subseteq A$. For any $A' \subseteq A$, $g A' \subseteq g A$.*

Theorem 8.12 (halting subsets). *Let $g : X \xrightarrow{\text{map}} Y$ with halting set A^* . For any $A \subseteq A^*$, $\text{domain } (g A) = A$.*

Proof. Use the mapping arrow restriction law (21). \square

Now we can prove measurability.

Lemma 8.13 (measurability under \circ_{map}). *If $g_1 : X \rightarrow Y$ is \mathcal{A} - \mathcal{B} -measurable and $g_2 : Y \rightarrow Z$ is \mathcal{B} - \mathcal{C} -measurable, then $g_2 \circ_{\text{map}} g_1$ is \mathcal{A} - \mathcal{C} -measurable.*

Theorem 8.14 (measurability under \ggg_{map}). *If $g_1 : X \xrightarrow{\text{map}} Y$ is \mathcal{A} - \mathcal{B} -measurable and $g_2 : Y \xrightarrow{\text{map}} Z$ is \mathcal{B} - \mathcal{C} -measurable, then $g_1 \ggg_{\text{map}} g_2$ is \mathcal{A} - \mathcal{C} -measurable.*

Proof. Let $A^* \in \mathcal{A}$ and $B^* \in \mathcal{B}$ be respectively g_1 's and g_2 's halting sets. The halting set of $g_1 \ggg_{\text{map}} g_2$ is $A^{**} := \text{preimage } (g_1 A^*) B^*$, which is in \mathcal{A} . By definition,

$$(g_1 \ggg_{\text{map}} g_2) A^{**} = \text{let } g'_1 := g_1 A^{**} \quad (51) \\ g'_2 := g_2 (\text{range } g'_1) \\ \text{in } g'_2 \circ_{\text{map}} g'_1$$

By Theorem 8.8, g'_1 is an \mathcal{A} - \mathcal{B} -measurable mapping. Unfortunately, g'_2 may not be \mathcal{B} - \mathcal{C} -measurable when $\text{range } g'_1 \notin \mathcal{B}$.

Let $g''_2 := g_2 B^*$, which is a \mathcal{B} - \mathcal{C} -measurable mapping. By Lemma 8.13, $g''_2 \circ_{\text{map}} g'_1$ is \mathcal{A} - \mathcal{C} -measurable. We need only show that $g'_2 \circ_{\text{map}} g'_1 = g''_2 \circ_{\text{map}} g'_1$, which by Lemma 8.10 is true if $\text{range } g'_1 \subseteq \text{domain } g'_2$ and $g'_2 \subseteq g''_2$.

By Theorem 8.12, $A^{**} \subseteq A^*$ implies $\text{domain } g'_1 = A^{**}$. By Theorem 8.11 and Lemma 8.9,

$$\begin{aligned} \text{range } g'_1 &= \text{image } (g_1 A^{**}) (\text{preimage } (g_1 A^*) B^*) \\ &= \text{image } (g_1 A^*) (\text{preimage } (g_1 A^*) B^*) \\ &\subseteq B^* \end{aligned}$$

range $g'_1 \subseteq B^*$ implies (by Theorem 8.12) that $\text{domain } g'_2 = \text{range } g'_1$, and (by Theorem 8.11) that $g'_2 \subseteq g'_2$. \square

8.1.2 Case: Pairing

Lemma 8.15 (measurability under $\langle \cdot, \cdot \rangle_{\text{map}}$). *If $g_1 : X \rightarrow Y_1$ is \mathcal{A} - \mathcal{B}_1 -measurable and $g_2 : X \rightarrow Y_2$ is \mathcal{A} - \mathcal{B}_2 -measurable, then $\langle g_1, g_2 \rangle_{\text{map}}$ is \mathcal{A} -($\mathcal{B}_1 \otimes \mathcal{B}_2$)-measurable.*

Theorem 8.16 (measurability under $(\&\&\&_{\text{map}})$). *If $g_1 : X \xrightarrow{\text{map}} Y_1$ is \mathcal{A} - \mathcal{B}_1 -measurable and $g_2 : X \xrightarrow{\text{map}} Y_2$ is \mathcal{A} - \mathcal{B}_2 -measurable, then $g_1 \&\&\&_{\text{map}} g_2$ is \mathcal{A} -($\mathcal{B}_1 \otimes \mathcal{B}_2$)-measurable.*

Proof. Let A_1^* and A_2^* be respectively g_1 's and g_2 's halting sets. The halting set of $g_1 \&\&\&_{\text{map}} g_2$ is $A^{**} := A_1^* \cap A_2^*$, which is in \mathcal{A} . By definition, $(g_1 \&\&\&_{\text{map}} g_2) A^{**} = \langle g_1 A^{**}, g_2 A^{**} \rangle_{\text{map}}$, which by Lemma 8.15 is \mathcal{A} -($\mathcal{B}_1 \otimes \mathcal{B}_2$)-measurable. \square

8.1.3 Case: Conditional

Lemma 8.17 (union of disjoint, measurable mappings). *Let $g : \text{Set } (X \rightarrow Y)$ be a countable set of measurable mappings with disjoint domains. Then $\bigcup g$ is measurable.*

Theorem 8.18 (measurability under ifte_{map}). *If $g_1 : X \xrightarrow{\text{map}} \text{Bool}$ is \mathcal{A} -($\mathcal{P} \text{ Bool}$)-measurable, and $g_2 : X \xrightarrow{\text{map}} Y$ and $g_3 : X \xrightarrow{\text{map}} Y$ are \mathcal{A} - \mathcal{B} -measurable, then $\text{ifte}_{\text{map}} g_1 g_2 g_3$ is \mathcal{A} - \mathcal{B} -measurable.*

Proof. Let A_1^* , A_2^* and A_3^* be respectively g_1 's, g_2 's and g_3 's halting sets. The halting set of $\text{ifte}_{\text{map}} g_1 g_2 g_3$ is defined by

$$\begin{aligned} A_2^{**} &:= A_2^* \cap \text{preimage } (g_1 A_1^*) \{\text{true}\} \\ A_3^{**} &:= A_3^* \cap \text{preimage } (g_1 A_1^*) \{\text{false}\} \\ A^{**} &:= A_2^{**} \uplus A_3^{**} \end{aligned} \quad (52)$$

Because $\text{preimage } (g_1 A_1^*) B \in \mathcal{A}$ for any $B \subseteq \text{Bool}$, $A^{**} \in \mathcal{A}$. By definition,

$$\begin{aligned} \text{ifte}_{\text{map}} g_1 g_2 g_3 A^{**} &= \text{let } g'_1 := g_1 A^{**} \\ &\quad g'_2 := g_2 (\text{preimage } g'_1 \{\text{true}\}) \\ &\quad g'_3 := g_3 (\text{preimage } g'_1 \{\text{false}\}) \\ &\quad \text{in } g'_2 \uplus_{\text{map}} g'_3 \end{aligned} \quad (53)$$

By hypothesis, g'_1 , g'_2 and g'_3 are measurable mappings, and the mapping arrow restriction law (21) implies g'_2 and g'_3 have disjoint domains. Apply Lemma 8.17. \square

8.1.4 Case: Laziness

Lemma 8.19 (measurability of \emptyset). *For any σ -algebras \mathcal{A} and \mathcal{B} , the empty mapping \emptyset is \mathcal{A} - \mathcal{B} -measurable.*

Theorem 8.20 (measurability under lazy_{map}). *Let $g : 1 \Rightarrow (X \xrightarrow{\text{map}} Y)$. If $g \ 0$ is \mathcal{A} - \mathcal{B} -measurable, then $\text{lazy}_{\text{map}} g$ is \mathcal{A} - \mathcal{B} -measurable.*

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

$$\text{lazy}_{\text{map}} g A^{**} = \text{if } (A^{**} = \emptyset) \emptyset (g \ 0 A^{**}) \quad (54)$$

If $A^{**} = \emptyset$, then $\text{lazy}_{\text{map}} g A^{**} = \emptyset$; apply Lemma 8.19. If $A^{**} \neq \emptyset$, then $\text{lazy}_{\text{map}} g = g \ 0$, which is \mathcal{A} - \mathcal{B} -measurable. \square

8.2 Measurable Probabilistic Programs

We are now prepared to lift the previous theorems to probabilistic computations.

Definition 8.21 (measurable mapping* arrow computation). *Let \mathcal{A} and \mathcal{B} be σ -algebras on $(R \times T) \times X$ and Y . A computation $g : X \xrightarrow{\text{map}^*} Y$ is \mathcal{A} - \mathcal{B} -measurable if $g \ j_0 A^*$ is an \mathcal{A} - \mathcal{B} -measurable mapping, where A^* is g 's halting set.*

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, ΣX denotes its **standard σ -algebra**, which must be defined under the following constraints:*

$$\Sigma \langle X_1, X_2 \rangle = \Sigma X_1 \otimes \Sigma X_2 \quad (55)$$

$$\Sigma (J \rightarrow X) = (\Sigma X)^{\otimes J} \quad (56)$$

$$X' \in \Sigma X \implies \Sigma X' = \{X' \cap A \mid A \in \Sigma X\} \quad (57)$$

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

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

$$\Sigma \text{Bool} ::= \mathcal{P} \text{Bool} \quad (58)$$

$$\Sigma T ::= \mathcal{P} T \quad (59)$$

Theorem 8.23. $\text{arr}_{\text{map}^*} \text{fst}$ and $\text{arr}_{\text{map}^*} \text{snd}$ are measurable.

Proof. Follows from (55) and Definition 8.3. \square

Theorem 8.24 (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 $\text{arr}_{\text{map}^*} \text{fst}$ and $\text{arr}_{\text{map}^*} \text{snd}$. \square

Theorem 8.25. For all $j \in J$, $\text{arr}_{\text{map}^*} (\pi \ j)$ is measurable.

Proof. Follows from (56) and Definition 8.4. \square

In particular, $\text{random}_{\text{map}^*}$ and $\text{branch}_{\text{map}^*}$ are measurable.

Theorem 8.26. $\text{ifte}'_{\text{map}^*}$ is measurable.

Proof. $\text{branch}_{\text{map}^*}$ is measurable, and $\text{arr}_{\text{map}^*} \text{agrees}$ is measurable by (58). \square

Theorem 8.27 (all expressions are measurable). *For any program e lacking function definitions, $\llbracket e \rrbracket_{\text{map}^*}$ is measurable.*

Proof. By structural induction and the above theorems. \square

Theorem 8.28 (approximation with expressions). *Let $g := \llbracket e \rrbracket'_{\text{map}^*}$ converge, where $g : X \xrightarrow{\text{map}^*} Y$. For all $t \in T$, let $A := (R \times \{t\}) \times X$. There is an expression e' for which $\llbracket e' \rrbracket_{\text{map}^*} j_0 A = g j_0 A$.*

Proof. Let $j \in J$ be the largest for which $t \ j \neq \perp$. To construct e' , exhaustively apply first-order functions in e , but replace any $\text{ifte}'_{\text{map}^*}$ whose condition's index is greater than j with the equivalent expression \perp . Because g converges, recurrences must be guarded by if , so this process terminates after finitely many applications. \square

Theorem 8.29 (all probabilistic programs are measurable). *If $\llbracket e \rrbracket'_{\text{map}^*}$ converges, it is measurable.*

Proof. Let $g := \llbracket e \rrbracket'_{\text{map}^*}$ and $g' := g \text{ j}_0 ((R \times T) \times X)$. Because $g' = g \text{ j}_0 A^*$ where A^* is g 's halting set, we need only show that g' is a measurable mapping.

By mapping arrow monotonicity (Theorem 8.11),

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

By Theorem 8.28, for every $t \in T$, there is a nonrecursive program that computes $g((R \times \{t\}) \times X)$. By (59) and Theorem 8.27, each is measurable. By mapping arrow restriction (21), each is disjoint. By Lemma 8.17, their union is measurable. \square

Theorem 8.29 remains true when $\llbracket \cdot \rrbracket_a$ is extended with any rule whose right side is measurable. Examples include real operations (arithmetic, equality, inequality, limits) and lambda expressions transformed into closures.

8.3 Random Store Probabilities

Preimages under probabilistic programs are measurable subsets of $(R \times T) \times X$. 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 geometric function defined in (35) would depend not only on the probability of $\text{random} < p$, 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 $(R \times T) \times X$. 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 halting set.

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

Definition 8.31 (halting probability). *Let $g : X \rightsquigarrow_{\text{map}^*} Y$ be measurable, with A^* its halting set. Let $P \in \mathcal{P} R \rightarrow [0, 1]$ be a standard probability measure over random stores. The **halting probability** of g is $P(\text{image } \text{fst} \gg \text{fst}) A^*$.*

Theorem 8.32 (measurable finite projections). *Let $A \in \Sigma (X_1, X_2)$. If X_2 is at most countable, $\text{image } \text{fst } A \in \mathcal{A}_1$.*

Proof. Because $\Sigma X_2 = \mathcal{P} X_2$, A is a countable union of rectangles of the form $A_1 \times \{a_2\}$, where $A_1 \in \Sigma X_1$ and $a_2 \in X_2$. Because $\text{image } \text{fst}$ distributes over unions, $\text{image } \text{fst } A$ is a countable union of sets in ΣX_1 . \square

Theorem 8.33. *Let $g : X \rightsquigarrow_{\text{map}^*} Y$ be measurable. If X is at most countable, g 's halting probability is well-defined.*

Proof. T is countable; apply Theorem 8.32 twice. \square

In particular, for programs interpreted using $\llbracket \cdot \rrbracket_{\text{map}^*}$, $X = \{\langle \rangle\}$ (the empty list/stack), so their halting probabilities are well-defined.

Corollary 8.34. *Let $g : X \rightsquigarrow_{\text{map}^*} Y$ be measurable and $A \subseteq A^*$ a measurable set. $P(\text{image } (\text{fst} \gg \text{fst}) A)$, the probability of A , is well-defined.*

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

9. Implementable Approximation

If we were to confine preimage computation to countable sets, we could model them as lazy lists or streams and implement the preimage arrow almost directly. But we would like something more efficient, even if it means approximating, and we would like to be able to compute preimages of sets of real numbers.

If instead we confine preimage computation to arbitrary *rectangular* sets, only **preimage** and functions that use it would be uncomputable: **pre**, and thus **lift_{pre}**, and thus **arr_{pre}**. It might seem that not being able to lift pure functions to preimage arrow computations would be a major handicap, but it is not. For pure first-order programs, we need only lift **id**, $\lambda _ . y$, **fst** and **snd**. For probabilistic, partial programs, we need to lift $\pi \text{ j}$ to index the random store and branch traces, and **agrees** to implement $\text{ifte}'_{\text{pre}^*}$. We of course need lifts of any other primitives that we would like to include in the language.

Figure 8 defines preimage arrow computations equivalent to the necessary lifts. All except **arr_{pre}** **agrees** preserve rectangularity.

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

$$\text{Rect } \langle X_1, X_2 \rangle = (\text{Rect } X_1) \boxtimes (\text{Rect } X_2) \quad (60)$$

$$\text{Rect } (J \rightarrow X) = (\text{Rect } X)^{\boxtimes J} \quad (61)$$

where

$$\mathcal{A}_1 \boxtimes \mathcal{A}_2 := \{A_1 \times A_2 \mid A_1 \in \mathcal{A}_1, A_2 \in \mathcal{A}_2\} \quad (62)$$

$$\mathcal{A}^{\boxtimes J} := \{\prod_{j \in J} A_j \mid j \in J, A_j \in \mathcal{A}\} \quad (63)$$

lift cartesian products to families of sets.

It is not difficult to show that $(\cap) : \text{Rect } X \Rightarrow \text{Rect } X \Rightarrow \text{Rect } X$, that $\{a\} \in \text{Rect } X$ if $a \in X$

XXX: **arr_{pre}** is generally uncomputable, but we don't need that many lifts; Figure 8 has the rest of the non-arithmetic ones we'll need

XXX: figure out a good way to present the following info
Figure 4:

- **pre**: can't implement
- **pre-ap**: need \cap
- $\langle \cdot, \cdot \rangle_{\text{pre}}$: approximate; need \times and \cap
- \circ_{pre} : no change
- \uplus_{pre} : approximate; need join

Figure 6:

- **arr_{pre}** (and **lift_{pre}**): can't implement
- \gg_{pre} : no change
- $\&\&\&_{\text{pre}}$: use approximating $\langle \cdot, \cdot \rangle_{\text{pre}}$
- **ifte_{pre}**: need $\{\text{true}\}$ and $\{\text{false}\}$; use approximating \uplus_{pre}
- **lazy_{pre}**: need $(= \emptyset)$, $(\text{pre } \emptyset)$

Figure 8:

- **id_{pre}**: no change
- **const_{pre}**: need $\{y\}$, $(= \emptyset)$, \emptyset
- **fst_{pre}** and **snd_{pre}**: need projections, i , \times
- π_{pre} : need projections, \cap , arbitrary products

$$\begin{array}{ll}
\text{arr}_{\text{pre}} \text{ id } A \equiv \langle A, \lambda B. B \rangle & \text{arr}_{\text{pre}} (\lambda _ . y) A \equiv \langle \{y\}, \lambda B. \text{if } (B = \emptyset) \emptyset A \rangle \\
\text{arr}_{\text{pre}} \text{ fst } A \equiv \text{let } A_1 := \text{image fst } A & \text{arr}_{\text{pre}} (\pi j) A \equiv \text{let } B := \text{image } (\pi j) A \\
\quad A_2 := \text{image snd } A & \quad p := \lambda B. A \cap \prod_{i \in J} \text{if } (j = i) B (\text{image } (\pi j) A) \\
\quad \text{in } \langle A_1, \lambda B. A \cap (B \times A_2) \rangle & \quad \text{in } \langle B, p \rangle \\
\text{arr}_{\text{pre}} \text{ snd } A \equiv \text{let } A_1 := \text{image fst } A & \text{arr}_{\text{pre}} \text{ agrees } A \equiv \text{let } B := (\text{if } (\langle \text{true}, \text{true} \rangle \in A) \{ \text{true} \} \emptyset) \uplus \\
\quad A_2 := \text{image snd } A & \quad (\text{if } (\langle \text{false}, \text{false} \rangle \in A) \{ \text{false} \} \emptyset) \\
\quad \text{in } \langle A_2, \lambda B. A \cap (A_1 \times B) \rangle & \quad p := \lambda B. (\text{if } (\text{true} \in B) \{ \langle \text{true}, \text{true} \rangle \} \emptyset) \uplus \\
& \quad (\text{if } (\text{false} \in B) \{ \langle \text{false}, \text{false} \rangle \} \emptyset) \\
& \quad \text{in } \langle B, p \rangle
\end{array}$$

Figure 8: Specific instances of $\text{arr}_{\text{pre}} f$

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.