

Running Probabilistic Programs Backward

Neil Toronto Jay McCarthy

PLT @ Brigham Young University
ntoronto@racket-lang.org jay@cs.byu.edu

Abstract

It is primarily Bayesian statistical practice that drives probabilistic language development. Unfortunately, languages developed by programming language researchers most often lack probabilistic conditioning, making them nearly useless in Bayesian practice. Languages developed by Bayesian practitioners can be useful, but they lack specifications and are limited in seemingly arbitrary ways.

We develop a mathematical specification for a first-order language with recursion, extended with probabilistic choice and conditioning. Our semantics interprets programs as functions that compute preimages of sets of outputs. Distributions over outputs are defined by the probabilities of their preimages, a measure-theoretic approach that ensures the language is not artificially limited.

Measurability is a basic property similar to continuity that is critical, but often neglected. As part of proving correctness, we prove that all programs, including partial programs with real arithmetic, are measurable.

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

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

General Terms XXX, XXX

Keywords XXX, XXX

XXX: consider using paragraph headers instead of subsections and subsubsections

1. Introduction

There is currently no efficient probabilistic language implementation that simultaneously

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

In the field of programming language research, 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 implementation errors and errors in users' understanding. 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. (XXX: make the last sentence more specific, or remove it because it communicates nothing extra)

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:

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

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

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

1.1 Measure-Theoretic Probability

Measure-theoretic probability [13] 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 , where **preimage** returns the subset of f 's domain X for which f yields a value in B . In the thermometer example (1), f would be an interpretation of the program as a function, X would be the set of all random sources, and Y would be the set of the

program’s possible outputs. For any $B \subseteq Y$, $\text{preimage } f B$ 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 side-effects (except usually divergence), programmers must write probabilistic programs as functions from a random source to outputs. Monads and other categorical classes such as idioms and arrows can make doing so easier. (XXX: cite me, bunch of others)

It seems this approach should make it easy to interpret probabilistic programs measure-theoretically. For a probabilistic program $f : X \rightarrow Y$, 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 difficult to turn this simple-sounding idea into a compositional semantics, for the following reasons.

1. Preimages can be defined only for functions with observable domains, which excludes lambdas.
2. 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. It is very difficult to define probability measures for arbitrary spaces of measurable functions [3].

Implementing a language based on such a semantics is complicated because

4. Contemporary mathematics is unlike any implementation’s host language.
5. It requires running Turing-equivalent programs backward, efficiently, on possibly uncountable sets of outputs.

We address both 1 and 4 by developing our semantics in λ_{ZFC} [25], 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 [9] 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 [17], transforms first-order programs into the computations of an arbitrary arrow.
- Arrows for evaluating expressions in different ways.

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

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

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 like those in the top, except they thread an infinite store of random values, and always converge. (We can do this because in λ_{ZFC} , Turing-uncomputable programs are definable.) Most of our correctness theorems rely on proofs that every lift and η in (3) is a homomorphism.

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 (XXX: badness) the semantic function, and the $X \rightsquigarrow_{\perp} Y$ and $X \rightsquigarrow_{\text{pre}*} Y$ arrows’ combinators.

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} [25], an untyped, call-by-value λ -calculus designed for deriving implementable programs from contemporary mathematics.

Generally, contemporary mathematics—measure theory in particular—is done in **ZFC**: **Zermelo-Fraenkel** set theory extended with the axiom of **Choice** (equivalently unique **Cardinality**). ZFC has only first-order functions and no general recursion, which makes implementing a language defined by a transformation into ZFC quite difficult. The problem is exacerbated if implementing the language requires approximation. Targeting λ_{ZFC} instead allows creating a precise mathematical specification and deriving an approximating specification without changing languages.

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

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 directly to ZFC sets, assuming the existence of a single inaccessible cardinal.¹

In λ_{ZFC} , algebraic data structures are encoded as sets; e.g. (x, y) can be encoded as $\{\{x\}, \{x, y\}\}$. 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,

¹ A mild assumption, as ZFC+ κ is a smaller theory than Coq’s [4], and algebraic geometers assume a proper class of inaccessible.

data may be “infinitely wide” (such as \mathbb{R}) but not “infinitely tall” (such as infinite trees and lists).

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

- A free lowercase type variable is universally quantified.
- A free uppercase type variable is a set.
- A set denotes a member of that set.
- $x \Rightarrow y$ 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 .

Because the type $\text{Set } X$ denotes the same values as the set $\mathcal{P } X$ (i.e. subsets of X) we regard them as equivalent. Similarly, the type $\langle X, Y \rangle$ is equivalent to the 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 , $\bigcup A$ reduces to the union of the sets in 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:

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

Thus, $1 = 1$ reduces to $1 \in \{1\}$, which reduces to **true**.² 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 $(\lambda x. x) 1 = 1$.

²Technically, λ_{ZFC} has a big-step semantics, and the derivation tree for $1 = 1$ contains the derivation tree for $1 \in \{1\}$.

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 something slightly weaker.

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 bisimilarity [2]. However, we do not want internal equality and external equivalence to differ too much, and we want the flexibility of extending “ \equiv ” with type-specific rules.

2.2 Additional Functions and Forms

We assume a desugaring pass over λ_{ZFC} expressions, which automatically curries (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 **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:

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

$A \uplus B$ 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 unnamed mappings is defined by

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

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

Figure 1 defines other 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. Indexing functions are produced by

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

which is particularly useful when f is unnamed.

3. Arrows and First-Order Semantics

Like monads and idioms [19, 26], arrows [9] are used in functional programming to thread effects through computations in a way that imposes structure on the computations. Unlike monad and idiom computations, arrow computations are always

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

Figure 1: Operations on mappings.

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

The first property makes arrows a perfect fit for a compositional translation from expressions to intensional or extensional functions—or, as we will see, to computations that compute preimages. The second property makes them a perfect fit for a measure-theoretic semantics in particular, as app in the function arrow is generally not measurable [3]. Targeting arrows in the semantics therefore gives some assurance that we can meet measure theory's requirement that preimage measure be defined only for measurable functions. We prove in Section 8 that it is sufficient.

3.1 Alternative Arrow Definitions and Laws

We do not give typical minimal arrow definitions. For each arrow a , instead of 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 [9], we give $(\&\&\&_a)$ definitions because the applicable measure-theoretic theorems are in terms of pairing functions.

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

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

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

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

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*

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

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

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

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

Definition 3.2 (arrow homomorphism). *A function $\text{lift}_b : (x \rightsquigarrow_a y) \Rightarrow (x \rightsquigarrow_b y)$ 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 arrow homomorphism laws state that $\text{arr}_a : (x \Rightarrow y) \Rightarrow (x \rightsquigarrow_a y)$ must be a homomorphism from the function arrow (10) to arrow a . Roughly, arrow computations that do not use additional combinators can be transformed into arr_a applied to a pure computation. They must be *function-like*.

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

$$(f_1 \ggg_a f_2) \ggg_a f_3 \equiv f_1 \ggg_a (f_2 \ggg_a f_3) \quad (20)$$

a pair extraction law:

$$(\text{arr}_a f_1 \&\&_a f_2) \ggg_a \text{arr}_a \text{snd} \equiv f_2 \quad (21)$$

and distribution of pure computations over effectful:

$$\text{arr}_a f_1 \ggg_a (f_2 \&\&_a f_3) \equiv (\text{arr}_a f_1 \ggg_a f_2) \&\&_a (\text{arr}_a f_1 \ggg_a f_3) \quad (22)$$

$$\begin{aligned} \text{arr}_a f_1 \ggg_a \text{ifte}_a f_2 f_3 f_4 &\equiv \text{ifte}_a (\text{arr}_a f_1 \ggg_a f_2) \\ &\quad (\text{arr}_a f_1 \ggg_a f_3) \\ &\quad (\text{arr}_a f_1 \ggg_a f_4) \end{aligned} \quad (23)$$

$$\text{arr}_a f_1 \ggg_a \text{lazy}_a f_2 \equiv \text{lazy}_a \lambda 0. \text{arr}_a f_1 \ggg_a f_2 0 \quad (24)$$

Equivalence between different arrow representations is usually proved in a strongly normalizing λ -calculus [16, 17], in which every function is free of effects, including divergence. Such a λ -calculus has no need for lazy_a , so we could not derive (24) from existing arrow laws. But (XXX: justify).

The pair extraction law (21), which *can* be derived from existing arrow laws, is a little more problematic, in nonstrict λ -calculi as well as λ_{ZFC} . If f_1 can diverge, using (21) to transform a computation can turn a diverging expression into a converging one, or vice-versa. We could condition the pair extraction law on f_1 's totality. Instead, we require every argument to arr_a to be total, which simplifies more proofs.

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

Definition 3.3 (arrow epimorphism). *An arrow homomorphism $\text{lift}_b : (x \rightsquigarrow_a y) \Rightarrow (x \rightsquigarrow_b y)$ that has a right inverse (equiv. surjective) is an **arrow epimorphism** from a to b .*

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

Proof. For the pair extraction law (21), rewrite in terms of lift_b , apply homomorphism laws, and apply the pair extraction law for arrow a :

$$\begin{aligned} (\text{arr}_b f_1 \&\&_b f_2) \ggg_b \text{arr}_b \text{snd} \\ &\equiv (\text{lift}_b (\text{arr}_a f_1) \&\&_b (\text{lift}_b (\text{lift}_b^{-1} f_2))) \ggg_b \text{arr}_b \text{snd} \\ &\equiv \text{lift}_b (\text{arr}_a f_1 \&\&_a \text{lift}_b^{-1} f_2) \ggg_b \text{lift}_b (\text{arr}_a \text{snd}) \\ &\equiv \text{lift}_b ((\text{arr}_a f_1 \&\&_a \text{lift}_b^{-1} f_2) \ggg_a \text{arr}_a \text{snd}) \\ &\equiv \text{lift}_b (\text{lift}_b^{-1} f_2) \equiv f_2 \end{aligned}$$

The proofs for every other law are similar. \square

3.2 First-Order Let-Calculus Semantics

Figure 2 defines 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 referred to by De Bruijn indexes, with 0 referring to the innermost binding.

Perhaps unsurprisingly, the interpretation acts like a stack machine. The final expression has type $\langle \rangle \rightsquigarrow_a y$, 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 and y is the return type. Application sends a stack containing just an x .

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

Definition 3.5 (well-defined expression). *An expression e is **well-defined** under arrow a if $\llbracket e \rrbracket_a : x \rightsquigarrow_a y$ for some x and y , and $\llbracket e \rrbracket_a$ converges.*

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

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

Theorem 3.6 (homomorphisms distribute over expressions). *Let $\text{lift}_b : (x \rightsquigarrow_a y) \Rightarrow (x \rightsquigarrow_b y)$ be an arrow homomorphism. For all expressions e , $\llbracket e \rrbracket_b \equiv \text{lift}_b \llbracket e \rrbracket_a$.*

Proof. By structural induction.

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

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

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

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

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 3.6, $\llbracket \cdot \rrbracket_b$ is correct.

4. The Bottom and Mapping Arrows

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

$$\begin{array}{ccccc} X \rightsquigarrow_{\perp} Y & \xrightarrow{\text{lift}_{\text{map}}} & X \rightsquigarrow_{\text{map}} Y & \xrightarrow{\text{lift}_{\text{pre}}} & X \rightsquigarrow_{\text{pre}} Y \\ \eta_{\perp}^* \downarrow & & \downarrow \eta_{\text{map}}^* & & \downarrow \eta_{\text{pre}}^* \\ X \rightsquigarrow_{\perp}^* Y & \xrightarrow{\text{lift}_{\text{map}}^*} & X \rightsquigarrow_{\text{map}}^* Y & \xrightarrow{\text{lift}_{\text{pre}}^*} & X \rightsquigarrow_{\text{pre}}^* Y \end{array} \quad (25)$$

Through Section 6, we move across the top to $X \rightsquigarrow_{\text{pre}} Y$.

To use Theorem 3.6 to prove correct the interpretations of expressions as preimage arrow computations, we need the preimage arrow to be homomorphic to a simpler arrow whose behavior is well-understood. One obvious candidate is the function arrow (10). However, we will need to explicitly handle 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

$p ::= x := e; \dots; e$	
$e ::= x \mid e \mid \text{let } e \mid \langle e, e \rangle \mid \text{fst } e \mid \text{snd } e \mid \text{if } e \mid v \mid \dots$	
$v ::= [\text{first-order constants}]$	
$\llbracket x := e; \dots; e_{\text{body}} \rrbracket_a ::= x := \llbracket e \rrbracket_a; \dots; \llbracket e_{\text{body}} \rrbracket_a$	$\llbracket \langle e_1, e_2 \rangle \rrbracket_a ::= \llbracket e_1 \rrbracket_a \&\&\&_a \llbracket e_2 \rrbracket_a$
$\llbracket x \ e \rrbracket_a ::= \llbracket \langle e, \rangle \rrbracket_a \ggg_a x$	$\llbracket \text{fst } e \rrbracket_a ::= \llbracket e \rrbracket_a \ggg_a \text{arr}_a \text{fst}$
$\llbracket \text{let } e \ e_{\text{body}} \rrbracket_a ::= (\llbracket e \rrbracket_a \&\&\&_a \text{arr}_a \text{id}) \ggg_a \llbracket e_{\text{body}} \rrbracket_a$	$\llbracket \text{snd } e \rrbracket_a ::= \llbracket e \rrbracket_a \ggg_a \text{arr}_a \text{snd}$
$\llbracket \text{env } 0 \rrbracket_a ::= \text{arr}_a \text{fst}$	$\llbracket \text{if } e_c \ e_t \ e_f \rrbracket_a ::= \text{ifte}_a \llbracket e_c \rrbracket_a (\text{lazy}_a \lambda 0. \llbracket e_t \rrbracket_a) (\text{lazy}_a \lambda 0. \llbracket e_f \rrbracket_a)$
$\llbracket \text{env } (n+1) \rrbracket_a ::= \text{arr}_a \text{snd} \ggg_a \llbracket \text{env } n \rrbracket_a$	$\llbracket v \rrbracket_a ::= \text{arr}_a (\text{const } v)$
$\text{id} ::= \lambda a. a$	\dots
$\text{const } b ::= \lambda a. b$	subject to $\llbracket p \rrbracket_a : \langle \rangle \rightsquigarrow_a y$ for some y

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

$x \rightsquigarrow_{\perp} y ::= x \Rightarrow y_{\perp}$	$\text{ifte}_{\perp} : (x \rightsquigarrow_{\perp} \text{Bool}) \Rightarrow (x \rightsquigarrow_{\perp} y) \Rightarrow (x \rightsquigarrow_{\perp} y) \Rightarrow (x \rightsquigarrow_{\perp} y)$
$\text{arr}_{\perp} : (x \Rightarrow y) \Rightarrow (x \rightsquigarrow_{\perp} y)$	$\text{ifte}_{\perp} \ f_1 \ f_2 \ f_3 \ a ::= \text{case } f_1 \ a$
$\text{arr}_{\perp} \ f := f$	$\text{true} \rightarrow f_2 \ a$
	$\text{false} \rightarrow f_3 \ a$
	$\perp \rightarrow \perp$
$(\ggg_{\perp}) : (x \rightsquigarrow_{\perp} y) \Rightarrow (y \rightsquigarrow_{\perp} z) \Rightarrow (x \rightsquigarrow_{\perp} z)$	$\text{lazy}_{\perp} : (1 \Rightarrow (x \rightsquigarrow_{\perp} y)) \Rightarrow (x \rightsquigarrow_{\perp} y)$
$(f_1 \ggg_{\perp} f_2) \ a := \text{if } (f_1 \ a = \perp) \perp (f_2 \ (f_1 \ a))$	$\text{lazy}_{\perp} \ f \ a := f \ 0 \ a$
$(\&\&\&_{\perp}) : (x \rightsquigarrow_{\perp} y_1) \Rightarrow (x \rightsquigarrow_{\perp} y_2) \Rightarrow (x \rightsquigarrow_{\perp} \langle y_1, y_2 \rangle)$	
$(f_1 \&\&\&_{\perp} f_2) \ a := \text{if } (f_1 \ a = \perp \text{ or } f_2 \ a = \perp) \perp \langle f_1 \ a, f_2 \ a \rangle$	

Figure 3: Bottom arrow definitions.

the error value \perp as well as the inhabitants of y . The type Bool_{\perp} , for example, denotes the members of $\text{Bool} \uplus \{\perp\}$.

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

Definition 4.1 (bottom arrow equivalence). *Two bottom arrow computations $f_1 : x \rightsquigarrow_{\perp} y$ and $f_2 : x \rightsquigarrow_{\perp} y$ are equivalent, or $f_1 \equiv f_2$, when $f_1 \ a \equiv f_2 \ a$ for all $a : x$.*

Theorem 4.2. 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 $\text{Nothing} = \perp$. The proof of (24) (pure computations distribute over lazy_{\perp}) is trivial. \square

4.1 Deriving the Mapping Arrow

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

It is tempting to try to make the mapping arrow's computations mapping-valued; i.e. define it using $X \rightsquigarrow_{\text{map}} Y ::= X \rightarrow Y$, 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 could not define $\text{arr}_{\text{map}} : (X \Rightarrow Y) \Rightarrow (X \rightarrow Y)$: to define a mapping, we need a domain, but lambdas' domains are unobservable.

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

$$X \rightsquigarrow_{\text{map}} Y ::= \text{Set } X \Rightarrow (X \rightarrow Y) \quad (26)$$

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

To use Theorem 3.6 to prove that expressions 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:

$$\begin{aligned} \text{domain}_{\perp} &: (X \rightsquigarrow_{\perp} Y) \Rightarrow \text{Set } X \Rightarrow \text{Set } X \\ \text{domain}_{\perp} \ f \ A &:= \{a \in A \mid f \ a \neq \perp\} \end{aligned} \quad (27)$$

$$\begin{aligned} \text{lift}_{\text{map}} &: (X \rightsquigarrow_{\perp} Y) \Rightarrow (X \rightsquigarrow_{\text{map}} Y) \\ \text{lift}_{\text{map}} \ f \ A &:= \text{mapping } f \ (\text{domain}_{\perp} \ f \ A) \end{aligned} \quad (28)$$

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.

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

Figure 4: Mapping arrow definitions.

Definition 4.3 (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 $\text{arr}_{\text{map}} := \text{lift}_{\text{map}} \circ \text{arr}_{\perp}$ meets the first homomorphism law (15). The following subsections derive $(\&\&\&_{\text{map}})$, (\ggg_{map}) , ifte_{map} and lazy_{map} from bottom arrow combinators, in a way that ensures lift_{map} is an arrow homomorphism. Figure 4 contains the resulting definitions.

4.1.1 Case: Pairing

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

$$\begin{aligned}
\text{lift}_{\text{map}} (f_1 \&\&\&_{\perp} f_2) A \\
&\equiv \text{lift}_{\text{map}} (\lambda a. \text{if } (f_1 a = \perp \text{ or } f_2 a = \perp) \perp \langle f_1 a, f_2 a \rangle) A \\
&\equiv \text{let } f := \lambda a. \text{if } (f_1 a = \perp \text{ or } f_2 a = \perp) \perp \langle f_1 a, f_2 a \rangle \\
&\quad \text{in mapping } f (\text{domain}_{\perp} f A)
\end{aligned} \tag{29}$$

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

$$\begin{aligned}
\text{lift}_{\text{map}} (f_1 \&\&\&_{\perp} f_2) A \\
&\equiv \text{let } A_1 := (\text{domain}_{\perp} f_1 A) \\
&\quad A_2 := (\text{domain}_{\perp} f_2 A) \\
&\quad A' := A_1 \cap A_2 \\
&\quad \text{in } \lambda a \in A'. \langle f_1 a, f_2 a \rangle \\
&\equiv \text{let } g_1 := \text{lift}_{\text{map}} f_1 A \\
&\quad g_2 := \text{lift}_{\text{map}} f_2 A \\
&\quad A' := (\text{domain } g_1) \cap (\text{domain } g_2) \\
&\quad \text{in } \lambda a \in A'. \langle g_1 a, g_2 a \rangle \\
&\equiv \langle \text{lift}_{\text{map}} f_1 A, \text{lift}_{\text{map}} f_2 A \rangle_{\text{map}}
\end{aligned} \tag{30}$$

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

4.1.2 Case: Composition

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

XXX: add this, maybe cut later

4.1.3 Case: Conditional

Starting with the left side of (18), we expand definitions, and simplify f by restricting it to a domain for which $f_1 a \neq \perp$:

$$\begin{aligned}
\text{lift}_{\text{map}} (\text{ifte}_{\perp} f_1 f_2 f_3) A \\
&\equiv \text{let } f := \lambda a. \text{case } f_1 a \\
&\quad \text{true} \rightarrow f_2 a \\
&\quad \text{false} \rightarrow f_3 a \\
&\quad \perp \rightarrow \perp \\
&\quad \text{in mapping } f (\text{domain}_{\perp} f A) \\
&\equiv \text{let } g_1 := \text{mapping } f A \\
&\quad A_2 := \text{preimage } g_1 \{\text{true}\} \\
&\quad A_3 := \text{preimage } g_1 \{\text{false}\} \\
&\quad f := \lambda a. \text{if } (f_1 a) (f_2 a) (f_3 a) \\
&\quad \text{in mapping } f (\text{domain}_{\perp} f (A_2 \uplus A_3))
\end{aligned} \tag{31}$$

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

$$\begin{aligned}
\text{lift}_{\text{map}} (\text{ifte}_{\perp} f_1 f_2 f_3) A \\
&\equiv \text{let } g_1 := \text{lift}_{\text{map}} f_1 A \\
&\quad g_2 := \text{lift}_{\text{map}} f_2 (\text{preimage } g_1 \{\text{true}\}) \\
&\quad g_3 := \text{lift}_{\text{map}} f_3 (\text{preimage } g_1 \{\text{false}\}) \\
&\quad A' := (\text{domain } g_2) \uplus (\text{domain } g_3) \\
&\quad \text{in } \lambda a \in A'. \text{if } (a \in \text{domain } g_2) (g_2 a) (g_3 a) \\
&\equiv \text{let } g_1 := \text{lift}_{\text{map}} f_1 A \\
&\quad g_2 := \text{lift}_{\text{map}} f_2 (\text{preimage } g_1 \{\text{true}\}) \\
&\quad g_3 := \text{lift}_{\text{map}} f_3 (\text{preimage } g_1 \{\text{false}\}) \\
&\quad \text{in } g_2 \uplus_{\text{map}} g_3
\end{aligned} \tag{32}$$

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. $\lambda x. e \ x \neq e$ 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.

4.1.5 Correctness

Theorem 4.4 (mapping arrow correctness). *lift_{map} is an arrow homomorphism.*

Proof. By construction. \square

Corollary 4.5 (semantic correctness). *For all expressions e , $\llbracket e \rrbracket_{\text{map}} \equiv \text{lift}_{\text{map}} \llbracket e \rrbracket_{\perp}$.*

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

$$\begin{aligned} \text{nonmonotone} : \text{Bool} \xrightarrow{\text{map}} \text{Bool} \\ \text{nonmonotone } A := \text{if } (A = \emptyset) (\text{mapping id Bool}) \emptyset \end{aligned} \quad (33)$$

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

Definition 4.6 (mapping arrow law). *Let $g : X \xrightarrow{\text{map}} Y$. If there exists an $f : X \rightsquigarrow_{\perp} Y$ such that $g \equiv \text{lift}_{\text{map}} f$, then g obeys the *mapping arrow law*.*

XXX: mapping arrow restriction theorem

We assume from here on that the mapping arrow law holds for all $g : X \xrightarrow{\text{map}} Y$. By homomorphism of lift_{map} , mapping arrow combinators return computations that obey this law.

Theorem 4.7. *lift_{map} is an arrow epimorphism.*

Proof. Follows from Theorem 4.4 and Definition 4.6. \square

Corollary 4.8. *arr_{map} , $(\&\&\&_{\text{map}})$, $(\>\>\>_{\text{map}})$, ifte_{map} and lazy_{map} define an arrow.*

5. Lazy Preimage Mappings

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

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

$$X \xrightarrow{\text{pre}} Y ::= \langle \text{Set } Y, \text{Set } Y \Rightarrow \text{Set } X \rangle \quad (34)$$

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:

$$\begin{aligned} \text{pre} : (X \rightarrow Y) &\Rightarrow (X \xrightarrow{\text{pre}} Y) \\ \text{pre } g &:= \langle \text{range } g, \lambda B. \text{preimage } g \ B \rangle \end{aligned} \quad (35)$$

Applying a preimage mapping to any subset of its codomain: (XXX: unclear prose)

$$\begin{aligned} \text{ap}_{\text{pre}} : (X \xrightarrow{\text{pre}} Y) &\Rightarrow \text{Set } Y \Rightarrow \text{Set } X \\ \text{ap}_{\text{pre}} \langle Y', p \rangle B &:= p (B \cap Y') \end{aligned} \quad (36)$$

The necessary property here is that using ap_{pre} to compute preimages is the same as computing them from a mapping using preimage.

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

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 $\text{ap}_{\text{pre}} h_1 B \equiv \text{ap}_{\text{pre}} h_2 B$ for all $B \subseteq Y$.*

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

XXX: moar text in following subsections

5.1 Preimage Mapping Pairing

Imported 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 $\text{pre } \langle g_1, g_2 \rangle_{\text{map}} \equiv \langle \text{pre } g_1, \text{pre } g_2 \rangle_{\text{pre}}$.*

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 $B \in Y_1 \times Y_2$,

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

\square

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

Figure 5: Lazy preimage mappings and operations.

5.2 Preimage Mapping Composition

Imported 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 $C \subseteq Z$,

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

□

5.3 Preimage Mapping Disjoint Union

Imported Lemma 5.8 (preimage distributes over (\uplus_{map})). *Let $g_1 \in X \rightarrow Y$ and $g_2 \in X \rightarrow Y$ have disjoint domains. 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 $B \subseteq Y$,

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

□

6. Deriving the Preimage Arrow

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

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

$$X \xrightarrow{\text{pre}} Y ::= \text{Set } X \Rightarrow (X \xrightarrow{\text{pre}} Y) \quad (37)$$

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 3.6, we need to define correctness using a lift from the mapping arrow to the preimage arrow:

$$\begin{aligned}
\text{lift}_{\text{pre}} : (X \xrightarrow{\text{map}} Y) \Rightarrow (X \xrightarrow{\text{pre}} Y) \\
\text{lift}_{\text{pre}} g A := \text{pre } (g A)
\end{aligned} \quad (38)$$

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

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

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

Again, we need a coarser notion of equivalence.

Definition 6.1 (Preimage arrow equivalence). *Two preimage arrow computations $h_1 : X \rightsquigarrow_{\text{pre}} Y$ 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 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.

XXX: try paragraph headings instead of subsections

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} :

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

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 ap_{pre} 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} :

$$\begin{aligned} \text{ap}_{\text{pre}} (\text{lift}_{\text{pre}} (g_1 \>\>\>_{\text{map}} g_2) A) C \\ &\equiv \text{let } g'_1 := g_1 A \\ &\quad g'_2 := g_2 (\text{range } g'_1) \\ &\quad \text{in } \text{ap}_{\text{pre}} (\text{pre} (g'_2 \circ_{\text{map}} g'_1)) C \\ &\equiv \text{let } g'_1 := g_1 A \\ &\quad g'_2 := g_2 (\text{range } g'_1) \\ &\quad \text{in } \text{ap}_{\text{pre}} ((\text{pre } g'_1) \circ_{\text{pre}} (\text{pre } g'_2)) C \\ &\equiv \text{let } h_1 := \text{lift}_{\text{pre}} g_1 A \\ &\quad h_2 := \text{lift}_{\text{pre}} g_2 (\text{range}_{\text{pre}} h_1) \\ &\quad \text{in } \text{ap}_{\text{pre}} (h_2 \circ_{\text{pre}} h_1) C \end{aligned} \quad (40)$$

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 ap_{pre} 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 Theorem 5.2 in the definitions of h_2 and h_3 :

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

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 ap_{pre} 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 lift_{pre} ($g \ 0$):

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

Substituting $h \ 0$ for $\text{lift}_{\text{pre}} (g \ 0)$ and removing the application of ap_{pre} 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 expressions e , $\llbracket e \rrbracket_{\text{pre}} \equiv \text{lift}_{\text{pre}} \llbracket e \rrbracket_{\text{map}}$.*

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

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

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

Theorem 6.5. *lift_{pre} is an arrow epimorphism.*

Proof. Follows from Theorem 6.2 and Definition 6.4. \square

Corollary 6.6. *arr_{pre} , $(\&\&\&_{\text{pre}})$, $(\>\>\>_{\text{pre}})$, ifte_{pre} and lazy_{pre} define an arrow.*

7. Preimages Under Partial Functions

We have defined everything on the top of our roadmap:

$$\begin{array}{ccccc} X \rightsquigarrow_{\perp} Y & \xrightarrow{\text{lift}_{\text{map}}} & X \rightsquigarrow_{\text{map}} Y & \xrightarrow{\text{lift}_{\text{pre}}} & X \rightsquigarrow_{\text{pre}} Y \\ \eta_{\perp*} \downarrow & & \downarrow \eta_{\text{map}*} & & \downarrow \eta_{\text{pre}*} \\ X \rightsquigarrow_{\perp*} Y & \xrightarrow{\quad} & X \rightsquigarrow_{\text{map}*} Y & \xrightarrow{\quad} & X \rightsquigarrow_{\text{pre}*} Y \end{array} \quad (41)$$

and proved that lift_{map} and lift_{pre} are homomorphisms. Now we move down from all three top arrows simultaneously, and prove every morphism in (41) is an arrow homomorphism.

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

Figure 6: Preimage arrow definitions.

7.1 Motivation

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

Suppose `random` retrieves a number $r \in [0, 1]$ 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 `random` $< p$ is false:

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

For any $p > 0$, `geometric` p may diverge, but the probability of always taking the false branch is $(1 - p) \times (1 - p) \times (1 - p) \times \dots = 0$. Therefore, for $p > 0$, `geometric` p converges with probability 1.

Suppose we interpret (42) as $h : R \rightsquigarrow_{\text{pre}} \mathbb{N}$, a preimage arrow computation from random sources in R to natural numbers, and that we have a probability measure $P \in \mathcal{P} R \rightarrow [0, 1]$. 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.

7.2 Threading and Indexing

XXX: ‘Prior to defn 7.1, you need a transition that communicates “I’ve just given you the intuition, now let’s do the work”’

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

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

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

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

We need a way to assign unique indexes to expressions.

Definition 7.1 (binary indexing scheme). *Let J be an index set, $j_0 \in J$ 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 of left and right , then J, j_0, left and right define a **binary indexing scheme**.*

For example, let J be the set of lists of $\{0, 1\}$, $j_0 := \langle \rangle$, 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

$$\begin{aligned}
\text{left } (p/q) &:= (p - \frac{1}{2})/q \\
\text{right } (p/q) &:= (p + \frac{1}{2})/q
\end{aligned} \quad (43)$$

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

In any case, the index set J is always countable, and can be thought of as a set of indexes into an infinite binary tree. Values of type $J \rightarrow A$ encode an infinite binary tree of A values as an infinite vector (i.e. total mapping).

7.3 Applicative, Associative Store Transformer

We thread a random store through bottom, mapping, and preimage arrow computations by defining an **arrow transformer**: a type constructor that receives and produces an arrow type, and combinators for arrows of the produced type. (XXX: ask Dan B for a cite or Brent Yorgey (tell him I’m Jay’s student, who is friends with his advisor))

The applicative store arrow transformer’s type constructor takes a store type s and an arrow type $x \rightsquigarrow_a y$:

$$\text{AStore } s (x \rightsquigarrow_a y) ::= J \Rightarrow (\langle s, x \rangle \rightsquigarrow_a y) \quad (44)$$

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

$$\begin{aligned}
\eta_{a^*} : (x \rightsquigarrow_a y) &\Rightarrow \text{AStore } s (x \rightsquigarrow_a y) \\
\eta_{a^*} f j &:= \text{arr}_a \text{snd } >>>_a f
\end{aligned} \quad (45)$$

Because f never accesses the store, j is ignored.

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

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

Figure 7 defines the remaining combinators. Each sub-computation receives `left j`, `right j`, or some other unique binary index. We thus think of programs interpreted as AStore arrows as being completely unrolled into an infinite binary tree, with each expression labeled with its tree index.

7.4 Partial, Probabilistic Programs

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

Probabilistic Programs To interpret probabilistic programs, we use a random tree as the store.

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

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

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

We extend the let-calculus semantic function with

$$\llbracket \text{random} \rrbracket_{a^*} \equiv \text{random}_{a^*} \quad (47)$$

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

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

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

Let $T \subseteq J \rightarrow \text{Bool}_\perp$ 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 :

$$\begin{array}{ll}
\text{branch}_{a^*} : x \rightsquigarrow_{a^*} \text{Bool} & \\
\text{branch}_{a^*} \ j := \text{arr}_a \ (\text{fst } \ggg \ \pi \ j) & (48)
\end{array}$$

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

$$\begin{array}{ll}
\text{agrees} : (\text{Bool}, \text{Bool}) \Rightarrow \text{Bool}_\perp & \\
\text{agrees} \ \langle b_1, b_2 \rangle := \text{if } (b_1 = b_2) \ b_1 \ \perp & (49)
\end{array}$$

$$\begin{array}{ll}
\text{ifte}_{a^*}^\downarrow : (x \rightsquigarrow_{a^*} \text{Bool}) \Rightarrow (x \rightsquigarrow_{a^*} y) \Rightarrow (x \rightsquigarrow_{a^*} y) \Rightarrow (x \rightsquigarrow_{a^*} y) & \\
\text{ifte}_{a^*}^\downarrow \ k_1 \ k_2 \ k_3 \ j := & \\
\text{ifte}_a \ ((k_1 \ (\text{left } j) \&\&\&_a \ \text{branch}_{a^*} \ j) \ggg_a \text{arr}_a \ \text{agrees}) & \\
\quad (k_2 \ (\text{left } (\text{right } j))) & \\
\quad (k_3 \ (\text{right } (\text{right } j))) & (50)
\end{array}$$

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

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

$$\llbracket \text{if } e_c \ e_t \ e_f \rrbracket_{a^*}^\downarrow \equiv \text{ifte}_{a^*}^\downarrow \ \llbracket e_c \rrbracket_{a^*}^\downarrow \ (\text{lazy}_a \ \lambda 0. \llbracket e_t \rrbracket_{a^*}^\downarrow) \ (\text{lazy}_a \ \lambda 0. \llbracket e_f \rrbracket_{a^*}^\downarrow) \quad (51)$$

with the remaining rules similar to those of $\llbracket \cdot \rrbracket_{a^*}$.

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:

$$\begin{array}{ll}
\text{random}_{a^*} : x \rightsquigarrow_{a^*} [0, 1] & \\
\text{random}_{a^*} \ j := \text{arr}_a \ (\text{fst } \ggg \ \text{fst } \ggg \ \pi \ j) & (52)
\end{array}$$

$$\begin{array}{ll}
\text{branch}_{a^*} : x \rightsquigarrow_{a^*} \text{Bool} & \\
\text{branch}_{a^*} \ j := \text{arr}_a \ (\text{fst } \ggg \ \text{snd } \ggg \ \pi \ j) & (53)
\end{array}$$

The definitions of $\text{ifte}_{a^*}^\downarrow$ and $\llbracket \cdot \rrbracket_{a^*}^\downarrow$ remain the same.

7.5 Correctness

We have two arrow lifts to prove homomorphic: one from pure computations to effectful (i.e. from those that do not access the store to those that do), and one from effectful computations to effectful. For both, we need AStore arrow equivalence to be more extensional.

Definition 7.4 (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 $j \in J$.*

Pure Expressions Proving η_a^* is a homomorphism proves $\llbracket \cdot \rrbracket_a^*$ correctly interprets pure expressions. Because AStore accepts any arrow type $x \rightsquigarrow_a y$, we can do so using only general properties. From here on, we assume every AStore arrow's base type's combinators obey the arrow laws listed in Section 3.1.

Theorem 7.5 (pure AStore arrow correctness). *η_a^* is an arrow homomorphism.*

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

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

Corollary 7.6 (pure semantic correctness). *For all pure expressions e , $\llbracket e \rrbracket_a^* \equiv \eta_a^* \llbracket e \rrbracket_a$ and $\llbracket e \rrbracket_a^{\downarrow} \equiv \eta_a^* \llbracket e \rrbracket_a^{\downarrow}$.*

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

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

where $\text{lift}_b : (x \rightsquigarrow_a y) \Rightarrow (x \rightsquigarrow_b y)$.

The relationships are more clearly expressed by

$$\begin{array}{ccc} x \rightsquigarrow_a y & \xrightarrow{\text{lift}_b} & x \rightsquigarrow_b y \\ \eta_a^* \downarrow & & \downarrow \eta_b^* \\ x \rightsquigarrow_{a^*} y & \xrightarrow{\text{lift}_{b^*}} & x \rightsquigarrow_{b^*} y \end{array} \quad (55)$$

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

Theorem 7.7 (natural transformation). *If lift_b is an arrow homomorphism, then (55) commutes.*

Proof. Expand definitions and apply homomorphism laws (16) and (15) for lift_b :

$$\begin{aligned} \text{lift}_{b^*} (\eta_a^* f) &\equiv \lambda j. \text{lift}_b (\text{arr}_a \text{ snd} \ggg_a f) \\ &\equiv \lambda j. \text{lift}_b (\text{arr}_a \text{ snd}) \ggg_b \text{lift}_b f \\ &\equiv \lambda j. \text{arr}_b \text{ snd} \ggg_b \text{lift}_b f \\ &\equiv \eta_b^* (\text{lift}_b f) \end{aligned}$$

\square

Theorem 7.8 (effectful AStore arrow correctness). *If lift_b is an arrow homomorphism, then lift_{b^*} is an arrow homomorphism.*

Proof. XXX: do this \square

Corollary 7.9 (effectful semantic correctness). *If lift_b is an arrow homomorphism, then for all expressions e , $\llbracket e \rrbracket_{b^*} \equiv \text{lift}_{b^*} \llbracket e \rrbracket_a^*$ and $\llbracket e \rrbracket_{b^*}^{\downarrow} \equiv \text{lift}_{b^*} \llbracket e \rrbracket_a^{\downarrow}$.*

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

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

$$\begin{array}{ccccc} X \rightsquigarrow_{\perp} Y & \xrightarrow{\text{lift}_{\text{map}}} & X \rightsquigarrow_{\text{map}} Y & \xrightarrow{\text{lift}_{\text{pre}}} & X \rightsquigarrow_{\text{pre}} Y \\ \eta_{\perp}^* \downarrow & & \downarrow \eta_{\text{map}^*} & & \downarrow \eta_{\text{pre}^*} \\ X \rightsquigarrow_{\perp^*} Y & \xrightarrow{\text{lift}_{\text{map}^*}} & X \rightsquigarrow_{\text{map}^*} Y & \xrightarrow{\text{lift}_{\text{pre}^*}} & X \rightsquigarrow_{\text{pre}^*} Y \end{array} \quad (56)$$

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

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

$$\begin{aligned} \llbracket e \rrbracket_{\text{pre}^*} &\equiv \text{lift}_{\text{pre}^*} (\text{lift}_{\text{map}^*} \llbracket e \rrbracket_{\perp^*}) \\ \llbracket e \rrbracket_{\text{pre}^*}^{\downarrow} &\equiv \text{lift}_{\text{pre}^*} (\text{lift}_{\text{map}^*} \llbracket e \rrbracket_{\perp^*}^{\downarrow}) \end{aligned} \quad (57)$$

7.6 Convergence

To relate $\llbracket e \rrbracket_a^{\downarrow}$ computations to $\llbracket e \rrbracket_a^*$ computations, we need to find the largest domain on which they should agree.

Definition 7.12 (maximal domain). *A computation's maximal domain is the largest A^* for which*

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

The maximal domain of $k : X \rightsquigarrow_{a^} Y$ is that of $k j_0$.*

Because the above statements imply convergence, A^* is a subset of the largest domain for which the computations converge. It is not too hard to show (but is a bit tedious) that lifting computations preserves the maximal domain; e.g. the maximal domain of $\text{lift}_{\text{map}} f$ is the same as f 's, and the maximal domain of $\text{lift}_{\text{pre}} g$ is the same as g 's.

To ensure maximal domains exist, we need the domain operations above to have certain properties.

Theorem 7.13 (domain closure operators). *If $f : X \rightsquigarrow_{\perp} Y$, $g : X \rightsquigarrow_{\text{map}} Y$ and $h : X \rightsquigarrow_{\text{pre}} Y$, then $\text{domain}_{\perp} f$, $\text{domain} \circ g$, and $\text{domain}_{\text{pre}} \circ h$ are monotone, decreasing, and idempotent in the subdomains on which they converge.*

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

For any input for which $\llbracket e \rrbracket_{\perp^*}$ converges, there should be a branch trace for which $\llbracket e \rrbracket_{\perp^*}^{\downarrow}$ returns the correct output; it should otherwise return \perp .

Theorem 7.14. *Let $f := \llbracket e \rrbracket_{\perp^*} : X \rightsquigarrow_{\perp^*} Y$ with maximal domain A^* , and $f' := \llbracket e \rrbracket_{\perp^*}^{\downarrow}$. For all $\langle \langle r, t \rangle, a \rangle \in A^*$, there exists a $T' \subseteq T$ such that*

- If $t' \in T'$ then $f' j_0 \langle \langle r, t' \rangle, a \rangle = f j_0 \langle \langle r, t \rangle, a \rangle$.
- If $t' \in T \setminus T'$ then $f' j_0 \langle \langle r, t' \rangle, a \rangle = \perp$.

Proof. Define T' as the set of all $t' \in J \rightarrow \text{Bool}_{\perp}$ such that $t' j = z$ if the subcomputation with index j is an if whose test returns z . Because $f j_0 \langle \langle r, t \rangle, a \rangle$ converges, $t' j \neq \perp$ for at most finitely many j , so each $t' \in T$.

Let $t' \in T'$. Because the test of every if subcomputation at index j agrees with $t' j$ and f ignores branch traces, $f' j_0 \langle \langle r, t' \rangle, a \rangle = f j_0 \langle \langle r, t \rangle, a \rangle$.

Let $t' \in T \setminus T'$. There exists an if subexpression with a test that does not agree with t' ; therefore $f' j_0 \langle \langle r, t' \rangle, a \rangle = \perp$. \square

For any input for which $\llbracket e \rrbracket_{\perp^*}$ diverges or returns \perp , $\llbracket e \rrbracket_{\perp^*}^{\downarrow}$ should return \perp . Proving this is a little easier if we first identify subsets of J that correspond with finite prefixes of an infinite binary tree.

Definition 7.15 (index prefix/suffix). *A finite $J' \subset J$ is an **index prefix** if $J' = \{j_0\}$ or, for some index prefix J'' and $j \in J''$, $J' = J'' \uplus \{\text{left } j\}$ or $J' = J'' \uplus \{\text{right } j\}$.*

*$J \setminus J'$ is the corresponding **index suffix**.*

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

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

Theorem 7.16. *Let $f := \llbracket e \rrbracket_{\perp^*} : X \rightsquigarrow_{\perp^*} Y$ with maximal domain A^* , and $f' := \llbracket e \rrbracket_{\perp^*}^{\downarrow}$. For all $a \in ((R \times T) \times X) \setminus A^*$, $f' j_0 a = \perp$.*

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

Suppose $f j_0 a$ converges. If an if subcomputation's test does not agree with t , then $f' j_0 a = \perp$. If every if's test agrees, $f' j_0 a = f j_0 a = \perp$.

Suppose $f j_0 a$ diverges. The set of all indexes j for which $t j \neq \perp$, because it is finite, is contained within an index prefix J' . By hypothesis, there is an if subcomputation at some index j' such that $j' \in J \setminus J'$. Because $t j' = \perp$, $f' j_0 a = \perp$. \square

Corollary 7.17. *For all expressions e , the maximal domain of $\llbracket e \rrbracket_{\perp^*}^{\downarrow}$ is a subset of that of $\llbracket e \rrbracket_{\perp^*}$.*

Corollary 7.18. *Let $f' := \llbracket e \rrbracket_{\perp^*}^{\downarrow} : X \rightsquigarrow_{\perp^*} Y$ with maximal domain A^* , and $f := \llbracket e \rrbracket_{\perp^*}$. For all $a \in A^*$, $f' j_0 a = f j_0 a$.*

Corollary 7.19 (correct convergence everywhere). *Suppose $\llbracket e \rrbracket_{\perp^*}^{\downarrow} : X \rightsquigarrow_{\perp^*} Y$ has maximal domain A^* . Let $X' := (R \times T) \times X$. For all $a \in X'$, $A \subseteq X'$ and $B \subseteq Y$,*

$$\begin{aligned} \llbracket e \rrbracket_{\perp^*}^{\downarrow} j_0 a &= \text{if } (a \in A^*) (\llbracket e \rrbracket_{\perp^*} j_0 a) \perp \\ \llbracket e \rrbracket_{\text{map}^*}^{\downarrow} j_0 A &= \llbracket e \rrbracket_{\text{map}^*} j_0 (A \cap A^*) \\ \text{ap}_{\text{pre}} (\llbracket e \rrbracket_{\text{pre}^*}^{\downarrow} j_0 A) B &= \text{ap}_{\text{pre}} (\llbracket e \rrbracket_{\text{pre}^*} j_0 (A \cap A^*)) B \end{aligned} \quad (58)$$

In other words, preimages computed using $\llbracket \cdot \rrbracket_{\text{pre}^*}^{\downarrow}$ always converge, never include inputs that give rise to errors or divergence, and are correct.

8. Output Probabilities and Measurability

We have not assigned probabilities to any output sets yet.

Typically, for $g \in X \rightarrow Y$, the probability of $B \subseteq Y$ is

$$P(\text{preimage } g B) \quad (59)$$

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

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

$$P(\text{image } (\text{fst} \ggg \text{fst}) A) \quad (60)$$

for preimage sets $A \subseteq (R \times T) \times X$.

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

8.1 Measurability

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

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

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

8.2 Measurable Pure Computations

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

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

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

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

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

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

Imported Lemma 8.7. Let $g : X \rightarrow Y$ be an \mathcal{A} - \mathcal{B} -measurable mapping. For any $A \in \mathcal{A}$, restrict $g \restriction A$ 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 with maximal domain A^* . For all $A \subseteq A^*$ with $A \in \mathcal{A}$, $g \restriction A$ is an \mathcal{A} - \mathcal{B} -measurable mapping.

Proof. Use the mapping arrow law (Definition 4.6) and Lemma 8.7. \square

We do not need to prove that all interpretations using $\llbracket \cdot \rrbracket_a$ are measurable. However, we do need to prove that all the mapping arrow combinators preserve measurability.

8.2.1 Case: Composition

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

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

Imported 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 $A' \subseteq A \subseteq A^*$, $g \restriction A' \subseteq g \restriction A$.

Proof. Follows from Definition 4.6. \square

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

Proof. Follows from Theorem 7.13. \square

Now we can prove measurability.

Imported 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 maximal domains. The maximal domain of $g_1 \ggg_{\text{map}} g_2$ is $A^{**} := \text{preimage } (g_1 \restriction A^*) B^*$, which is in \mathcal{A} . By definition,

$$(g_1 \ggg_{\text{map}} g_2) A^{**} = \text{let } g'_1 := g_1 \restriction A^{**} \quad (61) \\ g'_2 := g_2 \restriction (\text{range } g'_1) \\ \text{in } g'_2 \circ_{\text{map}} g'_1$$

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 $g''_2 := g_2 \restriction B^*$, 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,

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

$\text{range } g'_1 \subseteq B^*$ 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

8.2.2 Case: Pairing

Imported 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 maximal domains. The maximal domain of $g_1 \&\&_{\text{map}} g_2$ is $A^{**} := A_1^* \cap A_2^*$, which is in \mathcal{A} . By definition, $(g_1 \&\&_{\text{map}} g_2) A^{**} = \langle g_1 \restriction A^{**}, g_2 \restriction A^{**} \rangle_{\text{map}}$, which by Lemma 8.15 is \mathcal{A} - $(\mathcal{B}_1 \otimes \mathcal{B}_2)$ -measurable. \square

8.2.3 Case: Conditional

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

Theorem 8.18 (measurability under 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 $g_3 : X \xrightarrow{\text{map}} Z$ 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 \mathcal{A}_1^* , \mathcal{A}_2^* and \mathcal{A}_3^* be g_1 's, g_2 's and g_3 's maximal domains. The maximal domain of $\text{ifte}_{\text{map}} g_1 g_2 g_3$ is

$$\begin{aligned} A_2^{**} &:= A_2^* \cap \text{preimage } (g_1 \restriction \mathcal{A}_1^*) \{\text{true}\} \\ A_3^{**} &:= A_3^* \cap \text{preimage } (g_1 \restriction \mathcal{A}_1^*) \{\text{false}\} \\ A^{**} &:= A_2^{**} \uplus A_3^{**} \end{aligned} \quad (62)$$

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

$$\begin{aligned} \text{ifte}_{\text{map}} g_1 g_2 g_3 A^{**} &= \text{let } g'_1 := g_1 \restriction A^{**} \\ &\quad g'_2 := g_2 \restriction (\text{preimage } g'_1 \{\text{true}\}) \\ &\quad g'_3 := g_3 \restriction (\text{preimage } g'_1 \{\text{false}\}) \\ &\quad \text{in } g'_2 \uplus_{\text{map}} g'_3 \end{aligned} \quad (63)$$

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

8.2.4 Case: Laziness

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

Proof. For any $B \in \mathcal{B}$, $\text{preimage } \emptyset B = \emptyset$, and $\emptyset \in \mathcal{A}$. \square

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

$$\text{lazy}_{\text{map}} g A^{**} = \text{if } (A^{**} = \emptyset) \emptyset (g \ 0 A^{**}) \quad (64)$$

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

8.3 Measurable Probabilistic Computations

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

Definition 8.21 (measurable mapping* arrow computation). *Let \mathcal{A} and \mathcal{B} be σ -algebras on $(R \times T) \times X$ and Y . A computation $g : X \rightsquigarrow_{\text{map}^*} Y$ is **\mathcal{A} - \mathcal{B} -measurable** if $g \ j_0$ is an \mathcal{A} - \mathcal{B} -measurable mapping arrow computation.*

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

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

Definition 8.22 (standard σ -algebra). *For a set X used as a type, ΣX denotes its **standard σ -algebra**, which must be defined under the following constraints:*

$$\Sigma \langle X_1, X_2 \rangle = \Sigma X_1 \otimes \Sigma X_2 \quad (65)$$

$$\Sigma (J \rightarrow X) = (\Sigma X)^{\otimes J} \quad (66)$$

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

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

$$\Sigma \text{Bool} ::= \mathcal{P} \text{Bool} \quad (67)$$

$$\Sigma T ::= \mathcal{P} T \quad (68)$$

Imported Lemma 8.23 (measurable mapping arrow lifts). *$\text{arr}_{\text{map}} \text{id}$, $\text{arr}_{\text{map}} \text{fst}$ and $\text{arr}_{\text{map}} \text{snd}$ are measurable. $\text{arr}_{\text{map}} (\text{const } b)$ is measurable if $\{b\}$ is a measurable set. For all $j \in J$, $\text{arr}_{\text{map}} (\pi \ j)$ is measurable.*

Corollary 8.24. *$\text{arr}_{\text{map}^*} \text{id}$, $\text{arr}_{\text{map}^*} \text{fst}$ and $\text{arr}_{\text{map}^*} \text{snd}$ are measurable. $\text{arr}_{\text{map}^*} (\text{const } b)$ is measurable if $\{b\}$ is a measurable set. $\text{random}_{\text{map}^*}$ and $\text{branch}_{\text{map}^*}$ are measurable.*

Theorem 8.25 (AStore measurability transfer). *Every AStore arrow combinator produces measurable mapping* computations from measurable mapping* computations.*

Proof. AStore’s combinators are defined in terms of the base arrow’s combinators and $\text{arr}_{\text{map}} \text{fst}$ and $\text{arr}_{\text{map}} \text{snd}$. \square

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

Proof. $\text{branch}_{\text{map}^*}$ is measurable, and $\text{arr}_{\text{map}} \text{agrees}$ is measurable by (67). \square

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

Theorem 8.28 (all finite expressions are measurable). *For all finite expressions e , $\llbracket e \rrbracket_{\text{map}^*}$ is measurable.*

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

Theorem 8.29 (approximation with finite expressions). *Let $g := \llbracket e \rrbracket_{\text{map}^*}^\downarrow : X \rightsquigarrow_{\text{map}^*} Y$. For all $t \in T$, there is a finite expression e' for which $\llbracket e' \rrbracket_{\text{map}^*} \ j_0 A = g \ j_0 A$, where $A := (R \times \{t\}) \times X$.*

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

Theorem 8.30 (all probabilistic expressions are measurable). *For all expressions e , $\llbracket e \rrbracket_{\text{map}^*}^\downarrow$ is measurable.*

Proof. Let $g := \llbracket e \rrbracket_{\text{map}^*}^\downarrow$ and $g' := g \ j_0 ((R \times T) \times X)$. By Corollary 7.19, $g' = g \ j_0 A^*$ where A^* is g ’s maximal domain; thus we need only show that g' is a measurable mapping.

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

$$g' = \bigcup_{t \in T} g \ j_0 ((R \times \{t\}) \times X) \quad (69)$$

By Theorem 8.29, for every $t \in T$, there is an expression that computes $g ((R \times \{t\}) \times X)$. By (68) and Theorem 8.28, each is measurable. By the mapping arrow law (Definition 4.6), each is disjoint. By Lemma 8.17, their union is measurable. \square

Theorem 8.30 remains true when $\llbracket \cdot \rrbracket_a$ is extended with any rule whose right side is measurable, including rules for real arithmetic, equality, inequality and limits. More generally, any continuous or (countably) piecewise continuous function can be made available as a language primitive, as long as its domain’s and codomain’s standard σ -algebras are generated from their topologies.

It is not difficult to compose $\llbracket \cdot \rrbracket_a$ with another semantic function that lifts and defunctionalizes lambda expressions. Thus, the interpretations of all expressions in higher-order languages are measurable.

8.4 Measurable Projections

If $g := \llbracket e \rrbracket_{\text{map}^*}^\downarrow : X \rightsquigarrow_{\text{map}^*} Y$, then the probability of a measurable output set $B \in \Sigma Y$ is

$$P (\text{image } (\text{fst} \ggg \text{fst}) (\text{preimage } (g \ j_0 A^*) B)) \quad (70)$$

if domain $P = \Sigma R$. Unfortunately, projected sets are generally not measurable. Fortunately, for interpretations of programs $\llbracket p \rrbracket_{\text{map}^*}^\downarrow$, for which $X = \{\langle \rangle\}$, we have a special case.

Theorem 8.31 (measurable finite projections). *Let $A \in \Sigma \langle X_1, X_2 \rangle$. If X_2 is at most countable and $\Sigma X_2 = \mathcal{P} X_2$, then $\text{image } \text{fst } A \in \mathcal{A}_1$.*

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

$$\text{image } (\text{fst} \ggg \text{fst}) (\text{preimage } (g \ j_0 A^*) B) \in \Sigma R \quad (71)$$

Proof. T is countable and $\Sigma T = \mathcal{P} T$ by definition (68); apply Theorem 8.31 twice. \square

In particular, for any $\llbracket p \rrbracket_{\text{map}^*}^\downarrow : \{\langle \rangle\} \rightsquigarrow_{\text{map}^*} Y$, the probabilities of measurable output sets are well-defined.

9. Approximating Semantics

If we were to confine preimage computation to finite sets, we could implement the preimage arrow directly. But we would like something that works efficiently on infinite sets, even if it means approximating.

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

9.1 Implementable Lifts

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

Fortunately, we need only certain lifts. Figure 2 (which defines $\llbracket \cdot \rrbracket_a$) lifts id , $\text{const } b$, fst and snd . Section 7.4, which defines the combinators used to interpret partial, probabilistic programs, lifts πj and agrees . Measurable functions made available as language primitives of course must be lifted to the preimage arrow.

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

- $A \cap B$, \emptyset , $\{\text{true}\}$, $\{\text{false}\}$ and $\{b\}$ for every $\text{const } b$
- $A_1 \times A_2$, $\text{proj}_{\text{fst}} A$ and $\text{proj}_{\text{snd}} A$
- $J \rightarrow X$, $\text{project } j \ A$ and $\text{unproject } j \ A \ B$
- $A = \emptyset$

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

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

$$\text{Rect } \langle X_1, X_2 \rangle = (\text{Rect } X_1) \boxtimes (\text{Rect } X_2) \quad (73)$$

$$\text{Rect } (J \rightarrow X) = (\text{Rect } X)^{\boxtimes J} \quad (74)$$

where

$$\mathcal{A}_1 \boxtimes \mathcal{A}_2 := \{A_1 \times A_2 \mid A_1 \in \mathcal{A}_1, A_2 \in \mathcal{A}_2\} \quad (75)$$

$$\mathcal{A}^{\boxtimes J} := \bigcup_{J' \subset J \text{ finite}} \left\{ \prod_{j \in J'} A_j \mid A_j \in \mathcal{A}, j \in J' \iff A_j \subset \bigcup \mathcal{A} \right\} \quad (76)$$

lift cartesian products to sets of sets.

For example, if $\text{Rect } \mathbb{R}$ contains all the closed real intervals, then by (73), $[0, 2] \times [1, \pi] \in \text{Rect } \langle \mathbb{R}, \mathbb{R} \rangle$.

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

Further, all of the operations in (72) can be exactly implemented if finite sets are modeled directly, sets in an ordered space (such as \mathbb{R}) are modeled by intervals, and sets in $\text{Rect } \langle X_1, X_2 \rangle$ are modeled by pairs of type

$\langle \text{Rect } X_1, \text{Rect } X_2 \rangle$. By (74), sets in $\text{Rect } (J \rightarrow X)$ have no more than finitely many axes that are proper subsets of X . They can be modeled by *finite* binary trees, whose nodes contain axes for an index prefix $J' \subset J$ (Definition 7.15). Axes with indexes in the suffix $J \setminus J'$ are implicitly X .

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

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

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

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

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

9.2 Approximate Preimage Mapping Operations

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

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

Theorem 9.4 (pair preimage overapproximation). *Let $g_1 \in X \rightarrow Y_1$ and $g_2 \in X \rightarrow Y_2$. For all $B \subseteq Y_1 \times Y_2$, $\text{preimage } \langle g_1, g_2 \rangle_{\text{map}} \ B \subseteq \text{preimage } g_1 \ (\text{proj}_{\text{fst}} B) \cap \text{preimage } g_2 \ (\text{proj}_{\text{snd}} B)$.*

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

It is not hard to use Theorem 9.4 to show that

$$\begin{aligned} \langle \cdot, \cdot \rangle'_{\text{pre}} : (X \xrightarrow{\text{pre}} Y_1) &\Rightarrow (X \xrightarrow{\text{pre}} Y_2) \Rightarrow (X \xrightarrow{\text{pre}} Y_1 \times Y_2) \\ \langle \langle Y'_1, p_1 \rangle, \langle Y'_2, p_2 \rangle \rangle'_{\text{pre}} &:= \\ \langle Y'_1 \times Y'_2, \lambda B. p_1 \ (\text{proj}_{\text{fst}} B) \cap p_2 \ (\text{proj}_{\text{snd}} B) \rangle & \end{aligned} \quad (77)$$

computes covering rectangles of preimages under pairing.

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

$$\begin{aligned} (A_1 \times A_2) \vee (B_1 \times B_2) &= (A_1 \vee B_1) \times (A_2 \vee B_2) \\ (\prod_{j \in J} A_j) \vee (\prod_{j \in J} B_j) &= \prod_{j \in J} A_j \vee B_j \end{aligned} \quad (78)$$

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

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

$$\begin{aligned} (\uplus'_{\text{pre}}) : (X \xrightarrow{\text{pre}} Y) &\Rightarrow (X \xrightarrow{\text{pre}} Y) \Rightarrow (X \xrightarrow{\text{pre}} Y) \\ h_1 \uplus'_{\text{pre}} h_2 &:= \text{let } Y' := (\text{range}_{\text{pre}} h_1) \vee (\text{range}_{\text{pre}} h_2) \\ p &:= \lambda B. (\text{ap}_{\text{pre}} h_1 \ B) \vee (\text{ap}_{\text{pre}} h_2 \ B) \\ \text{in } \langle Y', p \rangle & \end{aligned} \quad (79)$$

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

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

$$\begin{aligned}
\text{unproj}_{\text{fst}} : \text{Set } \langle X_1, X_2 \rangle &\Rightarrow \text{Set } X_1 \Rightarrow \text{Set } \langle X_1, X_2 \rangle \\
\text{unproj}_{\text{fst}} A B &:= \text{preimage } (\text{mapping fst } A) B \\
&\equiv A \cap (B \times \text{proj}_{\text{snd}} A)
\end{aligned}$$

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

$$\text{project} : J \Rightarrow \text{Set } (J \rightarrow X) \Rightarrow \text{Set } X$$

$$\text{project } j A := \text{image } (\pi j) A$$

$$\text{unproject} : J \Rightarrow \text{Set } (J \rightarrow X) \Rightarrow \text{Set } X \Rightarrow \text{Set } (J \rightarrow X)$$

$$\begin{aligned}
\text{unproject } j A B &:= \text{preimage } (\text{mapping } (\pi j) A) B \\
&\equiv A \cap \prod_{i \in J} \text{if } (j = i) B (\text{project } j A)
\end{aligned}$$

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

which overapproximates (\uplus_{pre}).

To interpret programs that may diverge, or that converge with probability 1, we need to approximate $\text{ifte}_{\text{pre}^*}^{\downarrow}$ (50).

Directly implementing $\text{ifte}_{\text{pre}^*}^{\downarrow}$ in terms of agrees cannot work: the preimage of Bool under agrees (as a mapping) is $\{\langle \text{true}, \text{true} \rangle, \langle \text{false}, \text{false} \rangle\}$, which is not rectangular. A lengthy (elided) sequence of substitutions to the defining expression for $\text{ifte}_{\text{pre}^*}^{\downarrow}$ results in

$$\begin{aligned}
\text{ifte}_{\text{pre}^*}^{\downarrow} k_1 k_2 k_3 j A &\equiv \text{let } \langle C_k, p_k \rangle := k_1 j_1 A \\
&\quad \langle C_b, p_b \rangle := \text{branch}_{\text{pre}^*} j A \\
&\quad C_2 := C_k \cap C_b \cap \{\text{true}\} \\
&\quad C_3 := C_k \cap C_b \cap \{\text{false}\} \\
&\quad A_2 := p_k C_2 \cap p_b C_2 \\
&\quad A_3 := p_k C_3 \cap p_b C_3 \\
&\text{in } (k_2 j_2 A_2) \uplus_{\text{pre}} (k_3 j_3 A_3)
\end{aligned} \tag{50}$$

where $j_1 = \text{left } j$ and so on. This does not use agrees and preserves rectangularity if k_1 , k_2 and k_3 do. Unfortunately, if A_2 or A_3 overapproximates \emptyset , it takes unnecessary branches, which can cause divergence. In the exact semantics, expressions interpreted using $\text{ifte}_{\text{pre}^*}^{\downarrow}$ never diverge.

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

We need an overapproximation for the non-branching case. In the exact semantics, the returned preimage mapping's range is a subset of Y , and it returns subsets of $A_2 \uplus A_3$. Therefore, $\text{ifte}_{\text{pre}^*}^{\downarrow}$ may return $\langle Y, \lambda B. A_2 \vee A_3 \rangle$ when $C_b = \{\text{true}, \text{false}\}$. We cannot refer to the type Y in the function definition, so we represent it using \top in the approximating semantics. Implementations can model it by a singleton “universe” instance for every $\text{Rect } Y$.

Figure 9 defines the final approximating preimage arrow. This arrow, the lifts in Figure 8, and the semantic function $\llbracket \cdot \rrbracket_3$ in Figure 2 define an approximating semantics for partial, probabilistic programs.

9.3 Correctness

From here on, $\llbracket \cdot \rrbracket_{\text{pre}^*}^{\downarrow}$ interprets programs as approximating preimage* arrow computations using $\text{ifte}_{\text{pre}^*}^{\downarrow}$. The following

theorems assume $h := \llbracket e \rrbracket_{\text{pre}^*}^{\downarrow} : X \rightsquigarrow_{\text{pre}^*} Y$ and $h' := \llbracket e' \rrbracket_{\text{pre}^*}^{\downarrow} : X \rightsquigarrow_{\text{pre}^*} Y$ for some expression e .

Theorem 9.5 (approximation). *For all $A \in \text{Rect } \langle \langle R, T \rangle, X \rangle$ and $B \in \text{Rect } Y$, $\text{ap}_{\text{pre}} (h j_0 A) B \subseteq \text{ap}'_{\text{pre}} (h' j_0 A) B$.*

Proof. By construction. \square

To use structural induction on the interpretation of e , we need a theorem that allows representing it as a finite expression (Definition 8.27). Because $\text{ifte}_{\text{pre}^*}^{\downarrow}$ does not branch when either branch could be taken, an equivalent finite expression exists for each rectangular domain subset A .

Theorem 9.6 (equivalent finite expression). *Let $A \in \text{Rect } \langle \langle R, T \rangle, X \rangle$. There is a finite expression e' for which $\text{ap}'_{\text{pre}} (h'' j_0 A) B = \text{ap}_{\text{pre}} (h' j_0 A) B$ for all $B \in \text{Rect } Y$, where $h'' := \llbracket e' \rrbracket_{\text{pre}^*}^{\downarrow}$.*

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

Corollary 9.7 (convergence). *For all $A \in \text{Rect } \langle \langle R, T \rangle, X \rangle$ and $B \in \text{Rect } Y$, $\text{ap}'_{\text{pre}} (h' j_0 A) B$ converges.*

Theorem 9.8 (monotonicity). *$\text{ap}'_{\text{pre}} (h' j_0 A) B$ is monotone in both A and B .*

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

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

For monotonicity in A , let $A' \in \text{Rect } \langle \langle R, T \rangle, X \rangle$, and e' be a finite expression equivalent to e for A' (Theorem 9.3). Monotonicity for rectangular subsets of A' follows from structural induction on the interpretation of e' .

$$\begin{aligned}
X \xrightarrow{\text{pre}}' Y &::= \langle \text{Rect } Y, \text{Rect } Y \Rightarrow \text{Rect } X \rangle & \langle \cdot, \cdot \rangle_{\text{pre}}' : (X \xrightarrow{\text{pre}}' Y_1) \Rightarrow (X \xrightarrow{\text{pre}}' Y_2) \Rightarrow (X \xrightarrow{\text{pre}}' Y_1 \times Y_2) \\
\text{ap}'_{\text{pre}} : (X \xrightarrow{\text{pre}}' Y) \Rightarrow \text{Rect } Y \Rightarrow \text{Rect } X & & \langle \langle Y'_1, p_1 \rangle, \langle Y'_2, p_2 \rangle \rangle'_{\text{pre}} := \\
\text{ap}'_{\text{pre}} \langle Y', p \rangle B &:= p (B \cap Y') & \langle Y'_1 \times Y'_2, \lambda B. p_1 (\text{proj}_{\text{fst}} B) \cap p_2 (\text{proj}_{\text{snd}} B) \rangle \\
(\circ'_{\text{pre}}) : (Y \xrightarrow{\text{pre}}' Z) \Rightarrow (X \xrightarrow{\text{pre}}' Y) \Rightarrow (X \xrightarrow{\text{pre}}' Z) & & (\uplus'_{\text{pre}}) : (X \xrightarrow{\text{pre}}' Y) \Rightarrow (X \xrightarrow{\text{pre}}' Y) \Rightarrow (X \xrightarrow{\text{pre}}' Y) \\
\langle Z', p_2 \rangle \circ'_{\text{pre}} h_1 &:= \langle Z', \lambda C. \text{ap}'_{\text{pre}} h_1 (p_2 C) \rangle & \langle Y'_1, p_1 \rangle \uplus'_{\text{pre}} \langle Y'_2, p_2 \rangle := \\
& & \langle Y'_1 \vee Y'_2, \lambda B. (\text{ap}'_{\text{pre}} \langle Y'_1, p_1 \rangle B) \vee (\text{ap}'_{\text{pre}} \langle Y'_2, p_2 \rangle B) \rangle
\end{aligned}$$

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

$$\begin{aligned}
X \xrightarrow{\text{pre}}' Y &::= \text{Rect } X \Rightarrow (X \xrightarrow{\text{pre}}' Y) & \text{ifte}'_{\text{pre}} : (X \xrightarrow{\text{pre}}' \text{Bool}) \Rightarrow (X \xrightarrow{\text{pre}}' Y) \Rightarrow (X \xrightarrow{\text{pre}}' Y) \Rightarrow (X \xrightarrow{\text{pre}}' Y) \\
(\ggg'_{\text{pre}}) : (X \xrightarrow{\text{pre}}' Y) \Rightarrow (Y \xrightarrow{\text{pre}}' Z) \Rightarrow (X \xrightarrow{\text{pre}}' Z) & & \text{ifte}'_{\text{pre}} h_1 h_2 h_3 A := \text{let } h'_1 := h_1 A \\
(h_1 \ggg'_{\text{pre}} h_2) A &:= \text{let } h'_1 := h_1 A & h'_2 := h_2 (\text{ap}'_{\text{pre}} h'_1 \{\text{true}\}) \\
& h'_2 := h_2 (\text{range}'_{\text{pre}} h'_1) & h'_3 := h_3 (\text{ap}'_{\text{pre}} h'_1 \{\text{false}\}) \\
& \text{in } h'_2 \circ'_{\text{pre}} h'_1 & \text{in } h'_2 \uplus'_{\text{pre}} h'_3 \\
(\lll'_{\text{pre}}) : (X \xrightarrow{\text{pre}}' Y_1) \Rightarrow (X \xrightarrow{\text{pre}}' Y_2) \Rightarrow (X \xrightarrow{\text{pre}}' \langle Y_1, Y_2 \rangle) & & \text{lazy}'_{\text{pre}} : (1 \Rightarrow (X \xrightarrow{\text{pre}}' Y)) \Rightarrow (X \xrightarrow{\text{pre}}' Y) \\
(h_1 \lll'_{\text{pre}} h_2) A &:= \langle h_1 A, h_2 A \rangle'_{\text{pre}} & \text{lazy}'_{\text{pre}} h A := \text{if } (A = \emptyset) \langle \emptyset, \lambda B. \emptyset \rangle (h 0 A)
\end{aligned}$$

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

$$\begin{aligned}
X \xrightarrow{\text{pre}^*}' Y &::= J \Rightarrow (\langle S, X \rangle \xrightarrow{\text{pre}^*}' Y) & \text{ifte}'_{\text{pre}^*} : (X \xrightarrow{\text{pre}^*}' \text{Bool}) \Rightarrow (X \xrightarrow{\text{pre}^*}' Y) \Rightarrow (X \xrightarrow{\text{pre}^*}' Y) \Rightarrow (X \xrightarrow{\text{pre}^*}' Y) \\
S &::= (J \rightarrow [0, 1]) \times (J \rightarrow \text{Bool}_{\perp}) & \text{ifte}'_{\text{pre}^*} k_1 k_2 k_3 j := \text{ifte}'_{\text{pre}} (k_1 (\text{left } j)) \\
(\ggg'_{\text{pre}^*}) : (X \xrightarrow{\text{pre}^*}' Y) \Rightarrow (Y \xrightarrow{\text{pre}^*}' Z) \Rightarrow (X \xrightarrow{\text{pre}^*}' Z) & & (k_2 (\text{left } (\text{right } j))) \\
(k_1 \ggg'_{\text{pre}^*} k_2) j &:= & (k_3 (\text{right } (\text{right } j))) \\
(\text{fst}_{\text{pre}} \lll'_{\text{pre}^*} k_1 (\text{left } j)) \ggg'_{\text{pre}} k_2 (\text{right } j) & & \text{lazy}'_{\text{pre}^*} : (1 \Rightarrow (X \xrightarrow{\text{pre}^*}' Y)) \Rightarrow (X \xrightarrow{\text{pre}^*}' Y) \\
(\lll'_{\text{pre}^*}) : (X \xrightarrow{\text{pre}^*}' Y_1) \Rightarrow (X \xrightarrow{\text{pre}^*}' Y_2) \Rightarrow (X \xrightarrow{\text{pre}^*}' \langle Y_1, Y_2 \rangle) & & \text{lazy}'_{\text{pre}^*} k j := \text{lazy}'_{\text{pre}} \lambda 0. k 0 j \\
(k_1 \lll'_{\text{pre}^*} k_2) j &:= k_1 (\text{left } j) \lll'_{\text{pre}} k_2 (\text{right } j) & \eta'_{\text{pre}^*} : (X \xrightarrow{\text{pre}^*}' Y) \Rightarrow (X \xrightarrow{\text{pre}^*}' Y) \\
& & \eta'_{\text{pre}^*} f j := \text{snd}_{\text{pre}} \ggg'_{\text{pre}} f
\end{aligned}$$

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

$$\begin{aligned}
\text{random}'_{\text{pre}^*} : X \xrightarrow{\text{pre}^*}' [0, 1] & & \text{ifte}^{\downarrow}_{\text{pre}^*}' : (X \xrightarrow{\text{pre}^*}' \text{Bool}) \Rightarrow (X \xrightarrow{\text{pre}^*}' Y) \Rightarrow (X \xrightarrow{\text{pre}^*}' Y) \Rightarrow (X \xrightarrow{\text{pre}^*}' Y) \\
\text{random}'_{\text{pre}^*} j &:= \text{fst}_{\text{pre}} \ggg'_{\text{pre}} \text{fst}_{\text{pre}} \ggg'_{\text{pre}} \pi_{\text{pre}} j & \text{ifte}^{\downarrow}_{\text{pre}^*}' k_1 k_2 k_3 j := \\
\text{branch}'_{\text{pre}^*} : X \xrightarrow{\text{pre}^*}' \text{Bool} & & \text{let } \langle C_k, p_k \rangle := k_1 (\text{left } j) A \\
\text{branch}'_{\text{pre}^*} j &:= \text{fst}_{\text{pre}} \ggg'_{\text{pre}} \text{snd}_{\text{pre}} \ggg'_{\text{pre}} \pi_{\text{pre}} j & \langle C_b, p_b \rangle := \text{branch}_{\text{pre}^*} j A \\
& & A_2 := p_k (C_k \cap C_b \cap \{\text{true}\}) \cap p_b (C_k \cap C_b \cap \{\text{true}\}) \\
& & A_3 := p_k (C_k \cap C_b \cap \{\text{false}\}) \cap p_b (C_k \cap C_b \cap \{\text{false}\}) \\
\text{fst}'_{\text{pre}^*} &:= \eta'_{\text{pre}^*} \text{fst}_{\text{pre}} & \text{in if } (C_b = \{\text{true}, \text{false}\}) \\
\text{snd}'_{\text{pre}^*} &:= \eta'_{\text{pre}^*} \text{snd}_{\text{pre}}; \dots & \langle T, \lambda _. A_2 \vee A_3 \rangle \\
& & (k_2 (\text{left } (\text{right } j)) A_2 \uplus'_{\text{pre}} k_3 (\text{right } (\text{right } j)) A_3)
\end{aligned}$$

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

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

For monotonicity in B , let e' be the finite expression equivalent to e for any fixed A . Monotonicity follows from structural induction on the interpretation of e' . \square

Theorem 9.9 (approximate preimages are decreasing). *For $A \in \text{Rect } \langle \langle R, T \rangle, X \rangle$ and $B \in \text{Rect } Y$, $\text{ap}'_{\text{pre}} (h' j_0 A) B \subseteq A$.*

Proof. Because they compute exact preimages of rectangular sets under restriction to rectangular domains, id_{pre} and the other lifts in Figure 8 are decreasing.

By definition and applying basic lattice properties,

$$\begin{aligned} \text{ap}'_{\text{pre}} ((h_1 \ggg'_{\text{pre}} h_2) A) B &\equiv \text{ap}'_{\text{pre}} (h_1 A) B' \text{ for some } B' \\ \text{ap}'_{\text{pre}} ((h_1 \&\&\&'_{\text{pre}} h_2) A) B &\equiv \text{ap}'_{\text{pre}} (h_1 A) (\text{proj}_{\text{fst}} B) \cap \\ &\quad \text{ap}'_{\text{pre}} (h_2 A) (\text{proj}_{\text{snd}} B) \\ \text{ap}'_{\text{pre}} (\text{ifte}'_{\text{pre}} h_1 h_2 h_3 A) B &\equiv \text{let } A_2 := \text{ap}'_{\text{pre}} (h_1 A) \{\text{true}\} \\ &\quad A_3 := \text{ap}'_{\text{pre}} (h_1 A) \{\text{false}\} \\ &\quad \text{in } \text{ap}'_{\text{pre}} (h_2 A_2) B \vee \\ &\quad \text{ap}'_{\text{pre}} (h_3 A_3) B \\ \text{ap}'_{\text{pre}} (\text{lazy}'_{\text{pre}} h A) B &\equiv \text{if } (A = \emptyset) \emptyset (\text{ap}'_{\text{pre}} (h \ 0 A) B) \end{aligned}$$

Thus, approximating preimage arrow combinators return decreasing computations when given decreasing computations. This property transfers trivially to approximating preimage* arrow combinators.

Let e' be the finite expression equivalent to e for A ; that approximate preimages are decreasing follows from structural induction on the interpretation of e' . \square

Corollary 9.10 (approximate preimages preserve disjointness). *If $A_1 \cap A_2 = \emptyset$, then $\text{ap}'_{\text{pre}} (h' \ j_0 \ A_1) B \cap \text{ap}'_{\text{pre}} (h' \ j_0 \ A_2) B = \emptyset$.*

9.4 Preimage Refinement Algorithm

With disjointness and monotonicity properties proved, it is natural to suppose that we can compute probabilities of preimages of B by computing preimages with respect to increasingly fine discretizations of A .

Definition 9.11 (preimage refinement algorithm). *Let $h' := \llbracket e \rrbracket_{\text{pre}}^{\downarrow} : X \rightsquigarrow_{\text{pre}}' Y$, $B \in \text{Rect } Y$, and define*

$$\begin{aligned} \text{refine} : \text{Rect } \langle \langle R, T \rangle, X \rangle &\Rightarrow \text{Rect } \langle \langle R, T \rangle, X \rangle \\ \text{refine } A &:= \text{ap}'_{\text{pre}} (h' \ j_0 \ A) B \end{aligned} \quad (81)$$

Define $\text{partition} : \text{Rect } \langle \langle R, T \rangle, X \rangle \Rightarrow \text{Set } (\text{Rect } \langle \langle R, T \rangle, X \rangle)$ to produce positive-measure, disjoint rectangles, and define

$$\begin{aligned} \text{refine}^* : \text{Set } (\text{Rect } \langle \langle R, T \rangle, X \rangle) &\Rightarrow \text{Set } (\text{Rect } \langle \langle R, T \rangle, X \rangle) \\ \text{refine}^* \mathcal{A} &:= \text{image refine } \left(\bigcup_{A \in \mathcal{A}} \text{partition } A \right) \end{aligned} \quad (82)$$

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

Theorem 9.8 (monotonicity) guarantees that the union of preimage refinements of a *partition* of A is never larger than the refinement of A itself. Corollary 9.10 (disjointness) guarantees that the refinements of a partition of A are a partition; thus, its measure is the sum of the measures of the rectangles that comprise it. Theorem 9.5 guarantees that the algorithm is **sound**: the true preimage of B is always contained in the covering partition refine^* returns.

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

We conjecture that a minimal requirement for preimage refinement's measures to converge is that a program must

converge with probability 1. There are certainly other requirements. We leave these and proof of convergence of measures for future work.

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

10. Implementations

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

10.1 Direct Implementations

If sets are restricted to be finite, the arrows used as translation targets in the exact semantics, defined in Figures 1, 3, 4, 5, 6 and 7, can be implemented directly in any practical λ -calculus with a set data type. Computing exact preimages is very inefficient, even under the interpretations of very small programs. However, we have found our Typed Racket [24] implementation useful for finding theorem candidates.

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

Making the rectangular set type polymorphic seems to require the equivalent of a typeclass system. In Haskell, it also requires multi-parameter typeclasses or indexed type families [6] to associate set types with the types of their members. Using indexed type families, the only significant differences between the Haskell implementation and the approximating semantics are type contexts, `newtype` wrappers for arrow types, and using `Maybe` types as bottom arrow return types.

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

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

10.2 Dr. Bayes

Our main implementation, **Dr. Bayes**, is written in Typed Racket. It consists of the semantic function $\llbracket \cdot \rrbracket_{\text{a}}^{\downarrow}$ from Figure 2 and its extension $\llbracket \cdot \rrbracket_{\text{a}}^{\downarrow}$, the bottom* arrow as defined in Figures 3 and 7, the approximating preimage and preimage* arrows as defined in Figures 8 and 9, and algorithms to compute approximate probabilities. We use it to test the feasibility of solving real-world problems by computing approximate preimages.

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

tinguish cases using ifte_{pre} ; e.g.

$$\begin{aligned} \text{mul}_{\text{pre}} := & \text{ifte}_{\text{pre}} (\text{fst}_{\text{pre}} \ggg_{\text{pre}} \text{positive?}_{\text{pre}}) \\ & (\text{ifte}_{\text{pre}} (\text{snd}_{\text{pre}} \ggg_{\text{pre}} \text{positive?}_{\text{pre}}) \\ & \quad \text{mul}_{\text{pre}}^{++} \\ & \quad (\text{ifte}_{\text{pre}} (\text{snd}_{\text{pre}} \ggg_{\text{pre}} \text{negative?}_{\text{pre}}) \\ & \quad \quad \text{mul}_{\text{pre}}^{+-} \\ & \quad \quad (\text{const}_{\text{pre}} 0))) \\ & \dots \end{aligned} \quad (83)$$

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

Section 9.4 outlines preimage refinement: a discretization algorithm that seems to converge for programs that halt with probability 1, consisting of repeatedly shrinking and repartitioning a program's domain. We do not use this algorithm directly in our main implementation because it is inefficient. Good accuracy requires fine discretization, which is *exponential* in the number of discretized axes. For example, a nonrecursive program that contains only 10 uses of `random` would need to partition 10 axes of R , the set of random sources. Splitting each axis into only 4 disjoint intervals yields a partition of R of size $4^{10} = 1,048,576$.

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

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

$$\text{Pr}[B'|B] := P(A' \cap A) / P A \quad (84)$$

This can be approximated using **rejection sampling**. Given a nonempty list of samples xs from any superset of A , the conditional probability in (84) is approximately

$$\text{Pr}[B'|B] \approx \frac{\text{length}(\text{filter}(\in A' \cap A) xs)}{\text{length}(\text{filter}(\in A) xs)} \quad (85)$$

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

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

We cannot directly compute $a \in A$ or $a \in A' \cap A$ in (85), but we can use the fact that A and A' are preimages, and use the interpretation of the program as a bottom arrow computation $f : X \rightsquigarrow_{\perp} Y$:

$$\begin{aligned} \text{filter}(\in A) xs &= \text{filter}(\in \text{preimage}(g X) B) xs \\ &= \text{filter}(\lambda a. g X a \in B) xs \\ &= \text{filter}(\lambda a. f a \in B) xs \end{aligned} \quad (86)$$

Substituting into (85) gives

$$\text{Pr}[B'|B] \approx \frac{\text{length}(\text{filter}(\lambda a. f a \in B' \cap B) xs)}{\text{length}(\text{filter}(\lambda a. f a \in B) xs)} \quad (87)$$

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

For simplicity, the preceding discussion does not deal with projecting preimages from the domain of programs $(R \times T) \times \{\langle \rangle\}$ onto the set of random sources R . Shortly, Dr. Bayes samples rectangles from covering partitions of $(R \times T) \times \{\langle \rangle\}$ subsets, weights each rectangle by the inverse of the probability with which it is sampled, and projects onto R . This algorithm is a variant of **importance sampling** [7, Section 12.4], where the candidate distribution is defined by the sampling algorithm's partitioning choices, and the target distribution is P .

XXX: specific problems

11. Loosely Related Work

XXX: possible citations:

- labeling/numbering expressions (check CFA lit.)
- bananas and lenses
- isomorphic data structure transformations
- prolog stuff
- constraint-functional programming

12. Related Work

XXX: copy citations below to appropriate places in the introduction

XXX: moar citations

XXX: rework prose below to fit paper better

Koller and Pfeffer [14] efficiently compute exact distributions for the outputs of programs in a Scheme-like language. BUGS [18] focuses on efficient approximate computation for probabilistic theories with a finitely many statements, with distributions that practitioners typically use. BLOG [20] exists specifically to allow stating distributions over countably infinite vectors. BLAISE [5] allows stating both distribution and approximation method for each random variable. Church [8] is a Scheme-like probabilistic language with approximate inference, and focuses on expressiveness.

Kiselyov [12] embeds a probabilistic language in O'Caml for efficient computation. It uses continuations to enumerate or sample random variable values, and has a `fail` construct for the *complement* of conditioning. The sampler looks ahead for `fail` and can handle it efficiently. This may be justified by commutativity (Thm. ??), depending on interaction with other language features.

There is a fair amount of semantics work in probabilistic languages. Most of it is not motivated by Bayesian concerns, and thus does not define conditioning. Kozen [15] defines the meaning of bounded-space, imperative “while” programs as functions from probability measures to probability measures. Hurd [10] proves properties about programs with binary random choice by encoding programs and portions of measure theory in HOL.

Jones [11] develops a domain-theoretic variation of probability theory, and with it defines the probability monad, whose discrete version is a distribution-valued variation of the set or list monad. Ramsey and Pfeffer [23] define the probability monad measure-theoretically and implement a language for finite probability. We do not build on this work because the probability monad does not build a probability space, making it difficult to reason about conditioning.

Pfeffer also develops IBAL [22], apparently the only lambda calculus with finite probabilistic choice that also defines conditional queries. Park [21] extends a lambda calculus with probabilistic choice, defining it for a very general

class of probability measures using inverse transform sampling.

13. Conclusions and Future Work

XXX: todo

References

- [1] Haskell 98 language and libraries, the revised report, December 2002. URL <http://www.haskell.org/onlinereport/>.
- [2] S. Abramsky. The lazy lambda calculus. In *Research Topics in Functional Programming*, pages 65–116. Addison-Wesley, 1990.
- [3] R. J. Aumann. Borel structures for function spaces. *Illinois Journal of Mathematics*, 5:614–630, 1961.
- [4] B. Barras. Sets in Coq, Coq in sets. *Journal of Formalized Reasoning*, 3(1), 2010.
- [5] K. A. Bonawitz. *Composable Probabilistic Inference with Blaise*. PhD thesis, Massachusetts Institute of Technology, 2008.
- [6] M. M. T. Chakravarty, G. Keller, S. P. Jones, and S. Marlow. Associated types with class. In *Principles of Programming Languages*, pages 1–13, 2005.
- [7] M. DeGroot and M. Schervish. *Probability and Statistics*. Addison Wesley Publishing Company, Inc., 2012. ISBN 9780321500465.
- [8] N. Goodman, V. Mansinghka, D. Roy, K. Bonawitz, and J. Tenenbaum. Church: a language for generative models. In *Uncertainty in Artificial Intelligence*, 2008.
- [9] J. Hughes. Programming with arrows. In *5th International Summer School on Advanced Functional Programming*, pages 73–129, 2005.
- [10] J. Hurd. *Formal Verification of Probabilistic Algorithms*. PhD thesis, University of Cambridge, 2002.
- [11] C. Jones. *Probabilistic Non-Determinism*. PhD thesis, University of Edinburgh, 1990.
- [12] O. Kiselyov and C. Shan. Monolingual probabilistic programming using generalized coroutines. In *Uncertainty in Artificial Intelligence*, 2008.
- [13] A. Klenke. *Probability Theory: A Comprehensive Course*. Springer, 2006. ISBN 978-1-84800-047-6.
- [14] D. Koller, D. McAllester, and A. Pfeffer. Effective Bayesian inference for stochastic programs. In *14th National Conference on Artificial Intelligence*, August 1997.
- [15] D. Kozen. Semantics of probabilistic programs. In *Foundations of Computer Science*, 1979.
- [16] S. Lindley, P. Wadler, and J. Yallop. Idioms are oblivious, arrows are meticulous, monads are promiscuous. *Electronic Notes in Theoretical Computer Science*, 2008.
- [17] S. Lindley, P. Wadler, and J. Yallop. The arrow calculus. *Journal of Functional Programming*, 20:51–69, 2010.
- [18] D. J. Lunn, A. Thomas, N. Best, and D. Spiegelhalter. WinBUGS – a Bayesian modelling framework. *Statistics and Computing*, 10(4), 2000.
- [19] C. McBride and R. Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(1), 2008.
- [20] B. Milch, B. Marthi, S. Russell, D. Sontag, D. Ong, and A. Kolobov. BLOG: Probabilistic models with unknown objects. In *International Joint Conference on Artificial Intelligence*, 2005.
- [21] S. Park, F. Pfenning, and S. Thrun. A probabilistic language based upon sampling functions. *Transactions on Programming Languages and Systems*, 31(1), 2008.
- [22] A. Pfeffer. The design and implementation of IBAL: A general-purpose probabilistic language. In *Statistical Relational Learning*. MIT Press, 2007.
- [23] N. Ramsey and A. Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *Principles of Programming Languages*, 2002.
- [24] S. Tobin-Hochstadt and M. Felleisen. The design and implementation of typed Scheme. In *Principles of Programming Languages*, 2008.
- [25] N. Toronto and J. McCarthy. Computing in Cantor’s paradise with λ -ZFC. In *Functional and Logic Programming Symposium (FLOPS)*, pages 290–306, 2012.
- [26] P. Wadler. Monads for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*. 2001.