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 probabilitic 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:

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

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

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

1.1 Measure-Theoretic Probability

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

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

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

defines the distribution over subsets B of Y, where preimage returns the subset of f's domain X for which f yields a value in B. In the thermometer example (1), f would be an interpretation of the program as a function, X would be the set of all random sources, and Y would be the set of the

[Copyright notice will appear here once 'preprint' option is removed.]

program's possible outputs. For any $B \subseteq Y$, $\mathsf{preimage}\ f\ B$ 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 $f : X \to Y$, the probability measure on output sets $B \subseteq Y$ should be defined by preimages of B under f and the probability measure on X. Unfortunately, it is difficult to turn this simple-sounding idea into a compositional semantics, for the following reasons.

1. Preimages can be defined only for functions with observable domains, which excludes lambdas.
2. 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 (XXX: cite Aumann).

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} [2], a λ-calculus with infinite sets, and both extensional and intensional functions. We address 5 by deriving and implementing a *conservative approximation* of the semantics.

There seems to be no way to simplify difficulty 2, so we work through it in Section 8. The outcome is worth it: we prove that all probabilistic programs are measurable, regardless of which inputs they diverge on. This includes uncomputable programs; for example, those that contain real equality tests and limits. We believe this result is the first of its kind, and is general enough to apply to almost all past and future work on probabilistic programming languages.

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

1.3 Arrow Solution Overview

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

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

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

$$\begin{tikzcd}
X \rightsquigarrow_{\bot} Y \arrow[r, "\mathsf{lift}_{\mathsf{map}}"] \arrow[d, "\eta_{\bot *}"'] & X \underset{\mathsf{map}}{\rightsquigarrow} Y \arrow[r, "\mathsf{lift}_{\mathsf{pre}}"] \arrow[d, "\eta_{\mathsf{map}*}"] & X \underset{\mathsf{pre}}{\rightsquigarrow} Y \arrow[d, "\eta_{\mathsf{pre}*}"] \\
X \rightsquigarrow_{\bot *} Y \arrow[r, "\mathsf{lift}_{\mathsf{map}*}"'] & X \underset{\mathsf{map}*}{\rightsquigarrow} Y \arrow[r, "\mathsf{lift}_{\mathsf{pre}*}"'] & X \underset{\mathsf{pre}*}{\rightsquigarrow} Y
\end{tikzcd} \tag{3}$$

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

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

An implementation implements (XXX: badness) the semantic function, and the $X \rightsquigarrow_{\bot} Y$ and $X \underset{\mathsf{pre}*}{\rightsquigarrow}' Y$ arrows' combinators.

Most of our correctness theorems rely on proofs that every lift and η in (3) is a homomorphism. We use lift and η to define the correctness of one arrow in terms of another arrow. Homomorphism properties allow lift and η to distribute over the other arrow's computations.

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

2. Operational Metalanguage

We write all of the programs in this paper in λ_{ZFC} [2], an untyped, call-by-value 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.[1]

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

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

In λ_{ZFC}, algebraic data structures are encoded as sets; e.g. a ***primitive ordered pair*** of x and y is $\{\{x\}, \{x, y\}\}$. Only the *existence* of encodings into sets is important, as

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

[2] Assuming the existence of an inaccessible cardinal.

it means data structures inherit a defining characteristic of sets: strictness. More precisely, the lengths of paths to data structure leaves is unbounded, but each path must be finite. Less precisely, data may be "infinitely wide" (such as $\mathbb{R}$) but not "infinitely tall" (such as infinite trees and lists).

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

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

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

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

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

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

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

We import applicable ZFC theorems as lemmas.

2.1 Internal and External Equality

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

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

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

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

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

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

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

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

2.2 Additional Functions and Forms

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

A less typical set operation we use is disjoint union:

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

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

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

Syntax for unnamed mappings is defined by

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

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

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

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

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

which is particularly useful when f is unnamed.

3. Arrows and First-Order Semantics

Like monads and idioms (XXX: cite Wadler, McBride), arrows (XXX: cite Hughes) are used in functional programming to thread effects through computations in a way that

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

$$\begin{aligned}
&\mathsf{domain} : (X \rightharpoonup Y) \Rightarrow \mathsf{Set}\ X\\
&\mathsf{domain}\ :=\ \mathsf{image\ fst}\\[1ex]
&\mathsf{range} : (X \rightharpoonup Y) \Rightarrow \mathsf{Set}\ Y\\
&\mathsf{range}\ :=\ \mathsf{image\ snd}\\[1ex]
&\mathsf{preimage} : (X \rightharpoonup Y) \Rightarrow \mathsf{Set}\ Y \Rightarrow \mathsf{Set}\ X\\
&\mathsf{preimage}\ f\ B\ :=\ \{a \in \mathsf{domain}\ f \mid f\ a \in B\}\\[1ex]
&\mathsf{restrict} : (X \rightharpoonup Y) \Rightarrow \mathsf{Set}\ X \Rightarrow (X \rightharpoonup Y)\\
&\mathsf{restrict}\ f\ A\ :=\ \lambda a \in (A \cap \mathsf{domain}\ f).\, f\ a
\end{aligned}$$

$$\begin{aligned}
&\langle\cdot,\cdot\rangle_{\mathsf{map}} : (X \rightharpoonup Y_1) \Rightarrow (X \rightharpoonup Y_2) \Rightarrow (X \rightharpoonup Y_1 \times Y_2)\\
&\langle g_1, g_2\rangle_{\mathsf{map}}\ :=\ \mathsf{let}\ \ A := (\mathsf{domain}\ g_1) \cap (\mathsf{domain}\ g_2)\\
&\qquad\qquad\qquad\ \mathsf{in}\ \ \lambda a \in A.\, \langle g_1\ a, g_2\ a\rangle\\[1ex]
&(\circ_{\mathsf{map}}) : (Y \rightharpoonup Z) \Rightarrow (X \rightharpoonup Y) \Rightarrow (X \rightharpoonup Z)\\
&g_2 \circ_{\mathsf{map}} g_1\ :=\ \mathsf{let}\ \ A := \mathsf{preimage}\ g_1\ (\mathsf{domain}\ g_2)\\
&\qquad\qquad\qquad\ \mathsf{in}\ \ \lambda a \in A.\, g_2\ (g_1\ a)\\[1ex]
&(\uplus_{\mathsf{map}}) : (X \rightharpoonup Y) \Rightarrow (X \rightharpoonup Y) \Rightarrow (X \rightharpoonup Y)\\
&g_1 \uplus_{\mathsf{map}} g_2\ :=\ \mathsf{let}\ \ A := (\mathsf{domain}\ g_1) \uplus (\mathsf{domain}\ g_2)\\
&\qquad\qquad\qquad\ \mathsf{in}\ \ \lambda a \in A.\, \mathsf{if}\ (a \in \mathsf{domain}\ g_1)\ (g_1\ a)\ (g_2\ a)
\end{aligned}$$

Figure 1: Operations on mappings.

imposes structure on the computations. Unlike monad and idiom computations, arrow computations are always

- 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 formalize shortly).
- First-order: There is no way to derive a computation $\mathsf{app} : \langle x \rightsquigarrow y, x\rangle \rightsquigarrow y$ 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 (XXX: cite Aumann). 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 [1], 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:

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

and try to define the following recursive function:

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

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

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

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

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

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

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

define an ***arrow*** *if certain monoid, homomorphism, and structural laws hold. (XXX: need to require* arr_a*'s argument to be total) The additional combinators*

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

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 ensure that arr_a distributes over function arrow combinators. These laws can be put in terms of more general homomorphism properties that deal with distributing an arrow-to-arrow lift, which we use extensively to prove correctness.

Definition 3.2 (arrow homomorphism)**.** *A function* $\mathsf{lift}_b : (x \rightsquigarrow_a y) \Rightarrow (x \rightsquigarrow_b y)$ *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*:*

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

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

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

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

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

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

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

$$(\mathsf{f_1} \ggg_{\mathsf{a}} \mathsf{f_2}) \ggg_{\mathsf{a}} \mathsf{f_3} \;\equiv\; \mathsf{f_1} \ggg_{\mathsf{a}} (\mathsf{f_2} \ggg_{\mathsf{a}} \mathsf{f_3}) \tag{20}$$

a pair extraction law:

$$(\mathsf{arr_a}\ \mathsf{f_1} \mathbin{\&\&\&_{\mathsf{a}}} \mathsf{f_2}) \ggg_{\mathsf{a}} \mathsf{arr_a}\ \mathsf{snd} \;\equiv\; \mathsf{f_2} \tag{21}$$

and distribution of pure computations over effectful:

$$\begin{aligned}
\mathsf{arr_a}\ \mathsf{f_1} \ggg_{\mathsf{a}} (\mathsf{f_2} \mathbin{\&\&\&_{\mathsf{a}}} \mathsf{f_3}) \;&\equiv\; (\mathsf{arr_a}\ \mathsf{f_1} \ggg_{\mathsf{a}} \mathsf{f_2}) \mathbin{\&\&\&_{\mathsf{a}}} \\
&\;\; (\mathsf{arr_a}\ \mathsf{f_1} \ggg_{\mathsf{a}} \mathsf{f_3}) && (22)\\
\mathsf{arr_a}\ \mathsf{f_1} \ggg_{\mathsf{a}} \mathsf{ifte_a}\ \mathsf{f_2}\ \mathsf{f_3}\ \mathsf{f_4} \;&\equiv\; \mathsf{ifte_a}\ (\mathsf{arr_a}\ \mathsf{f_1} \ggg_{\mathsf{a}} \mathsf{f_2}) && (23)\\
&\phantom{\equiv\;\mathsf{ifte_a}}\;\; (\mathsf{arr_a}\ \mathsf{f_1} \ggg_{\mathsf{a}} \mathsf{f_3}) \\
&\phantom{\equiv\;\mathsf{ifte_a}}\;\; (\mathsf{arr_a}\ \mathsf{f_1} \ggg_{\mathsf{a}} \mathsf{f_4}) \\
\mathsf{arr_a}\ \mathsf{f_1} \ggg_{\mathsf{a}} \mathsf{lazy_a}\ \mathsf{f_2} \;&\equiv\; \mathsf{lazy_a}\ \lambda 0.\, \mathsf{arr_a}\ \mathsf{f_1} \ggg_{\mathsf{a}} \mathsf{f_2}\ 0 && (24)
\end{aligned}$$

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

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

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

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

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

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

The proofs for every other law are similar. □

3.2 First-Order Let-Calculus Semantics

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

A program is a sequence of definition statements followed by a final expression. $[\![\cdot]\!]_{\mathsf{a}}$ compositionally transforms each defining expression and the final expression into arrow computations. Functions are named, but local variables and arguments are not. Instead, variables are referred to by De Bruijn indexes, with $\mathsf{0}$ referring to the innermost binding.

Perhaps unsurprisingly, the interpretation acts like a stack machine. The final expression has type $\langle\rangle \rightsquigarrow_{\mathsf{a}} \mathsf{y}$, where y is the type of the program's value, and $\langle\rangle$ denotes an empty list. Let-bindings push values onto the stack. First-order functions have type $\langle \mathsf{x}, \langle\rangle\rangle \rightsquigarrow_{\mathsf{a}} \mathsf{y}$ where x is the argument type and y is the return type. Application sends a stack containing just an x.

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

Definition 3.5 (well-defined expression)**.** *An expression e is* ***well-defined*** *under arrow* a *if* $[\![e]\!]_{\mathsf{a}} : \mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}$ *for some* x *and* y*, and* $[\![e]\!]_{\mathsf{a}}$ *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* $\mathsf{lift_b} : (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{b}} \mathsf{y})$ *be an arrow homomorphism. For all expressions e,* $[\![e]\!]_{\mathsf{b}} \equiv \mathsf{lift_b}\ [\![e]\!]_{\mathsf{a}}$.

Proof. By structural induction.

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

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

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

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

It is not hard to check the remaining cases. □

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

4. The Bottom and Mapping Arrows

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

$$\begin{tikzcd}
\mathsf{X} \rightsquigarrow_{\perp} \mathsf{Y} \arrow[r, "\mathsf{lift_{map}}"] \arrow[d, "\eta_{\perp *}"'] & \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y} \arrow[r, "\mathsf{lift_{pre}}"] \arrow[d, "\eta_{\mathsf{map}*}"] & \mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y} \arrow[d, "\eta_{\mathsf{pre}*}"] \\
\mathsf{X} \rightsquigarrow_{\perp *} \mathsf{Y} \arrow[r, "\mathsf{lift_{map*}}"'] & \mathsf{X} \underset{\mathsf{map*}}{\rightsquigarrow} \mathsf{Y} \arrow[r, "\mathsf{lift_{pre*}}"'] & \mathsf{X} \underset{\mathsf{pre*}}{\rightsquigarrow} \mathsf{Y}
\end{tikzcd} \tag{25}$$

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

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

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

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

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

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

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

$$\begin{aligned}
&\mathsf{x} \rightsquigarrow_{\bot} \mathsf{y}\ ::= \mathsf{x} \Rightarrow \mathsf{y}_{\bot} \\
\\
&\mathsf{arr}_{\bot} : (\mathsf{x} \Rightarrow \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\bot} \mathsf{y}) \\
&\mathsf{arr}_{\bot}\ \mathsf{f} := \mathsf{f} \\
\\
&(\ggg_{\bot}) : (\mathsf{x} \rightsquigarrow_{\bot} \mathsf{y}) \Rightarrow (\mathsf{y} \rightsquigarrow_{\bot} \mathsf{z}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\bot} \mathsf{z}) \\
&(\mathsf{f_1} \ggg_{\bot} \mathsf{f_2})\ \mathsf{a} := \mathsf{if}\ (\mathsf{f_1\ a} = \bot)\ \bot\ (\mathsf{f_2}\ (\mathsf{f_1\ a})) \\
\\
&(\&\&\&_{\bot}) : (\mathsf{x} \rightsquigarrow_{\bot} \mathsf{y_1}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\bot} \mathsf{y_2}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\bot} \langle \mathsf{y_1}, \mathsf{y_2} \rangle) \\
&(\mathsf{f_1}\ \&\&\&_{\bot}\ \mathsf{f_2})\ \mathsf{a} := \mathsf{if}\ (\mathsf{f_1\ a} = \bot\ \mathsf{or}\ \mathsf{f_2\ a} = \bot)\ \bot\ \langle \mathsf{f_1\ a}, \mathsf{f_2\ a} \rangle
\end{aligned}
\qquad
\begin{aligned}
&\mathsf{ifte}_{\bot} : (\mathsf{x} \rightsquigarrow_{\bot} \mathsf{Bool}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\bot} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\bot} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\bot} \mathsf{y}) \\
&\mathsf{ifte}_{\bot}\ \mathsf{f_1\ f_2\ f_3\ a} := \mathsf{case}\ \ \mathsf{f_1\ a} \\
&\qquad\qquad\qquad\qquad\ \ \mathsf{true} \longrightarrow \mathsf{f_2\ a} \\
&\qquad\qquad\qquad\qquad\ \ \mathsf{false} \longrightarrow \mathsf{f_3\ a} \\
&\qquad\qquad\qquad\qquad\ \ \bot \longrightarrow \bot \\
\\
&\mathsf{lazy}_{\bot} : (1 \Rightarrow (\mathsf{x} \rightsquigarrow_{\bot} \mathsf{y})) \Rightarrow (\mathsf{x} \rightsquigarrow_{\bot} \mathsf{y}) \\
&\mathsf{lazy}_{\bot}\ \mathsf{f\ a} := \mathsf{f}\ 0\ \mathsf{a}
\end{aligned}$$

Figure 3: Bottom arrow definitions.

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

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

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

4.1 Deriving the Mapping Arrow

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

It is tempting to try to make the mapping arrow's computations mapping-valued; i.e. define it using $\mathsf{X} \rightsquigarrow_{\mathsf{map}} \mathsf{Y} ::= \mathsf{X} \rightharpoonup \mathsf{Y}$, with $\mathsf{f_1} \ggg_{\mathsf{map}} \mathsf{f_2} := \mathsf{f_2} \circ_{\mathsf{map}} \mathsf{f_1}$ and $\mathsf{f_1}\ \&\&\&_{\mathsf{map}}\ \mathsf{f_2} := \langle \mathsf{f_1}, \mathsf{f_2} \rangle_{\mathsf{map}}$. Unfortunately, we could not define $\mathsf{arr_{map}} : (\mathsf{X} \Rightarrow \mathsf{Y}) \Rightarrow (\mathsf{X} \rightharpoonup \mathsf{Y})$: to define a mapping, we need a domain, but lambdas' domains are unobservable.

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

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

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

To use Theorem 3.6 to prove that expressions interpreted using $\llbracket\cdot\rrbracket_{\mathsf{map}}$ behave correctly, we need to define correctness using a lift from the bottom arrow to the mapping arrow. It is helpful to have a standalone function $\mathsf{domain}_{\bot}$ that computes the subset of A on which f does not return $\bot$. We define that first, and then define $\mathsf{lift_{map}}$ in terms of it:

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

$$\begin{aligned}
&\mathsf{lift_{map}} : (\mathsf{X} \rightsquigarrow_{\bot} \mathsf{Y}) \Rightarrow (\mathsf{X} \rightsquigarrow_{\mathsf{map}} \mathsf{Y}) \\
&\mathsf{lift_{map}}\ \mathsf{f\ A} := \mathsf{mapping\ f}\ (\mathsf{domain}_{\bot}\ \mathsf{f\ A})
\end{aligned} \tag{28}$$

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

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

Definition 4.3 (mapping arrow equivalence). *Two mapping arrow computations* $\mathsf{g_1} : \mathsf{X} \rightsquigarrow_{\mathsf{map}} \mathsf{Y}$ *and* $\mathsf{g_2} : \mathsf{X} \rightsquigarrow_{\mathsf{map}} \mathsf{Y}$ *are equivalent, or* $\mathsf{g_1} \equiv \mathsf{g_2}$*, when* $\mathsf{g_1\ A} \equiv \mathsf{g_2\ A}$ *for all* $\mathsf{A} \subseteq \mathsf{X}$.

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

$$\begin{aligned}
&\mathsf{arr_{map}} : (\mathsf{X} \Rightarrow \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y})\\
&\mathsf{arr_{map}} \;:=\; \mathsf{lift_{map}} \circ \mathsf{arr_{\bot}}
\end{aligned}$$

$$\begin{aligned}
&(\ggg_{\mathsf{map}}) : (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}) \Rightarrow (\mathsf{Y} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Z}) \Rightarrow (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Z})\\
&(\mathsf{g_1} \ggg_{\mathsf{map}} \mathsf{g_2})\ \mathsf{A} \;:=\; \begin{array}[t]{l}\mathsf{let}\ \ \mathsf{g_1'} := \mathsf{g_1\ A}\\ \qquad\ \mathsf{g_2'} := \mathsf{g_2\ (range\ g_1')}\\ \mathsf{in}\ \ \mathsf{g_2'} \circ_{\mathsf{map}} \mathsf{g_1'}\end{array}
\end{aligned}$$

$$\begin{aligned}
&(\mathbin{\&\&\&}_{\mathsf{map}}) : (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y_1}) \Rightarrow (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y_2}) \Rightarrow (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \langle \mathsf{Y_1}, \mathsf{Y_2} \rangle)\\
&(\mathsf{g_1} \mathbin{\&\&\&}_{\mathsf{map}} \mathsf{g_2})\ \mathsf{A} \;:=\; \langle \mathsf{g_1\ A}, \mathsf{g_2\ A} \rangle_{\mathsf{map}}
\end{aligned}$$

$$\begin{aligned}
&\mathsf{ifte_{map}} : (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Bool}) \Rightarrow (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y})\\
&\mathsf{ifte_{map}\ g_1\ g_2\ g_3\ A} \;:=\; \begin{array}[t]{l}\mathsf{let}\ \ \mathsf{g_1'} := \mathsf{g_1\ A}\\ \qquad\ \mathsf{g_2'} := \mathsf{g_2\ (preimage\ g_1'\ \{true\})}\\ \qquad\ \mathsf{g_3'} := \mathsf{g_3\ (preimage\ g_1'\ \{false\})}\\ \mathsf{in}\ \ \mathsf{g_2'} \uplus_{\mathsf{map}} \mathsf{g_3'}\end{array}
\end{aligned}$$

$$\begin{aligned}
&\mathsf{lazy_{map}} : (1 \Rightarrow (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y})) \Rightarrow (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y})\\
&\mathsf{lazy_{map}\ g\ A} \;:=\; \mathsf{if}\ (\mathsf{A} = \varnothing)\ \varnothing\ (\mathsf{g\ 0\ A})
\end{aligned}$$

$$\begin{aligned}
&\mathsf{lift_{map}} : (\mathsf{X} \rightsquigarrow_{\bot} \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y})\\
&\mathsf{lift_{map}\ f\ A} \;:=\; \{\langle \mathsf{a}, \mathsf{b} \rangle \in \mathsf{mapping\ f\ A} \mid \mathsf{b} \neq \bot\}
\end{aligned}$$

Figure 4: Mapping arrow definitions.

Clearly $\mathsf{arr_{map}} := \mathsf{lift_{map}} \circ \mathsf{arr_{\bot}}$ meets the first homomorphism law (15). The following subsections derive $(\mathbin{\&\&\&}_{\mathsf{map}})$, $(\ggg_{\mathsf{map}})$, $\mathsf{ifte_{map}}$ and $\mathsf{lazy_{map}}$ from bottom arrow combinators, in a way that ensures $\mathsf{lift_{map}}$ 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 $\mathsf{f_1} : \mathsf{X} \rightsquigarrow_{\bot} \mathsf{Y}$, $\mathsf{f_2} : \mathsf{X} \rightsquigarrow_{\bot} \mathsf{Z}$, and $\mathsf{A} \subseteq \mathsf{X}$,

$$\begin{aligned}
&\mathsf{lift_{map}}\ (\mathsf{f_1} \mathbin{\&\&\&}_{\bot} \mathsf{f_2})\ \mathsf{A}\\
&\quad\equiv\ \mathsf{lift_{map}}\ (\lambda \mathsf{a}.\, \mathsf{if}\ (\mathsf{f_1\ a} = \bot\ \mathsf{or}\ \mathsf{f_2\ a} = \bot)\ \bot\ \langle \mathsf{f_1\ a}, \mathsf{f_2\ a} \rangle)\ \mathsf{A}\\
&\quad\equiv\ \mathsf{let}\ \ \mathsf{f} := \lambda \mathsf{a}.\, \mathsf{if}\ (\mathsf{f_1\ a} = \bot\ \mathsf{or}\ \mathsf{f_2\ a} = \bot)\ \bot\ \langle \mathsf{f_1\ a}, \mathsf{f_2\ a} \rangle\\
&\qquad\quad \mathsf{in}\ \ \mathsf{mapping\ f\ (domain_{\bot}\ f\ A)}
\end{aligned} \tag{29}$$

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

$$\begin{aligned}
&\mathsf{lift_{map}}\ (\mathsf{f_1} \mathbin{\&\&\&}_{\bot} \mathsf{f_2})\ \mathsf{A}\\
&\quad\equiv\ \begin{array}[t]{l}\mathsf{let}\ \ \mathsf{A_1} := (\mathsf{domain_{\bot}\ f_1\ A})\\ \qquad\ \mathsf{A_2} := (\mathsf{domain_{\bot}\ f_2\ A})\\ \qquad\ \mathsf{A'} := \mathsf{A_1} \cap \mathsf{A_2}\\ \mathsf{in}\ \ \lambda \mathsf{a} \in \mathsf{A'}.\, \langle \mathsf{f_1\ a}, \mathsf{f_2\ a} \rangle\end{array}\\
&\quad\equiv\ \begin{array}[t]{l}\mathsf{let}\ \ \mathsf{g_1} := \mathsf{lift_{map}\ f_1\ A}\\ \qquad\ \mathsf{g_2} := \mathsf{lift_{map}\ f_2\ A}\\ \qquad\ \mathsf{A'} := (\mathsf{domain\ g_1}) \cap (\mathsf{domain\ g_2})\\ \mathsf{in}\ \ \lambda \mathsf{a} \in \mathsf{A'}.\, \langle \mathsf{g_1\ a}, \mathsf{g_2\ a} \rangle\end{array}\\
&\quad\equiv\ \langle \mathsf{lift_{map}\ f_1\ A}, \mathsf{lift_{map}\ f_2\ A} \rangle_{\mathsf{map}}
\end{aligned} \tag{30}$$

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

4.1.2 Case: Composition

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

XXX: add this, maybe cut later

4.1.3 Case: Conditional

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

$$\begin{aligned}
&\mathsf{lift_{map}}\ (\mathsf{ifte_{\bot}\ f_1\ f_2\ f_3})\ \mathsf{A}\\
&\quad\equiv\ \begin{array}[t]{l}\mathsf{let}\ \ \mathsf{f} := \lambda \mathsf{a}.\, \mathsf{case}\ \ \mathsf{f_1\ a}\\ \qquad\qquad\qquad \mathsf{true} \longrightarrow \mathsf{f_2\ a}\\ \qquad\qquad\qquad \mathsf{false} \longrightarrow \mathsf{f_3\ a}\\ \qquad\qquad\qquad \bot \longrightarrow \bot\\ \mathsf{in}\ \ \mathsf{mapping\ f\ (domain_{\bot}\ f\ A)}\end{array}\\
&\quad\equiv\ \begin{array}[t]{l}\mathsf{let}\ \ \mathsf{g_1} := \mathsf{mapping\ f\ A}\\ \qquad\ \mathsf{A_2} := \mathsf{preimage\ g_1\ \{true\}}\\ \qquad\ \mathsf{A_3} := \mathsf{preimage\ g_1\ \{false\}}\\ \qquad\ \mathsf{f} := \lambda \mathsf{a}.\, \mathsf{if}\ (\mathsf{f_1\ a})\ (\mathsf{f_2\ a})\ (\mathsf{f_3\ a})\\ \mathsf{in}\ \ \mathsf{mapping\ f\ (domain_{\bot}\ f\ (A_2} \uplus \mathsf{A_3))}\end{array}
\end{aligned} \tag{31}$$

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

$$\begin{aligned}
&\mathsf{lift_{map}}\ (\mathsf{ifte_{\bot}\ f_1\ f_2\ f_3})\ \mathsf{A}\\
&\quad\equiv\ \begin{array}[t]{l}\mathsf{let}\ \ \mathsf{g_1} := \mathsf{lift_{map}\ f_1\ A}\\ \qquad\ \mathsf{g_2} := \mathsf{lift_{map}\ f_2\ (preimage\ g_1\ \{true\})}\\ \qquad\ \mathsf{g_3} := \mathsf{lift_{map}\ f_3\ (preimage\ g_1\ \{false\})}\\ \qquad\ \mathsf{A'} := (\mathsf{domain\ g_2}) \uplus (\mathsf{domain\ g_3})\\ \mathsf{in}\ \ \lambda \mathsf{a} \in \mathsf{A'}.\, \mathsf{if}\ (\mathsf{a} \in \mathsf{domain\ g_2})\ (\mathsf{g_2\ a})\ (\mathsf{g_3\ a})\end{array}\\
&\quad\equiv\ \begin{array}[t]{l}\mathsf{let}\ \ \mathsf{g_1} := \mathsf{lift_{map}\ f_1\ A}\\ \qquad\ \mathsf{g_2} := \mathsf{lift_{map}\ f_2\ (preimage\ g_1\ \{true\})}\\ \qquad\ \mathsf{g_3} := \mathsf{lift_{map}\ f_3\ (preimage\ g_1\ \{false\})}\\ \mathsf{in}\ \ \mathsf{g_2} \uplus_{\mathsf{map}} \mathsf{g_3}\end{array}
\end{aligned} \tag{32}$$

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

4.1.4 Case: Laziness

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

$$\begin{aligned}
&\mathsf{lift_{map}}\ (\mathsf{lazy_{\bot}\ f})\ \mathsf{A}\\
&\quad\equiv\ \begin{array}[t]{l}\mathsf{let}\ \ \mathsf{A'} := \mathsf{domain_{\bot}}\ (\lambda \mathsf{a}.\, \mathsf{f\ 0\ a})\ \mathsf{A}\\ \mathsf{in}\ \ \mathsf{mapping}\ (\lambda \mathsf{a}.\, \mathsf{f\ 0\ a})\ \mathsf{A'}\end{array}
\end{aligned}$$

λ_{ZFC} does not have an η rule (i.e. $\lambda x.\, e\ x \not\equiv e$ because e may diverge), but we can use weaker facts. If $\mathsf{A} \neq \varnothing$, then $\mathsf{domain_{\bot}}\ (\lambda \mathsf{a}.\, \mathsf{f\ 0\ a})\ \mathsf{A} \equiv \mathsf{domain_{\bot}\ (f\ 0)\ A}$. Further, it diverges if and only if $\mathsf{mapping\ (f\ 0)\ A'}$ diverges. Therefore, if $\mathsf{A} \neq \varnothing$, we can replace $\lambda \mathsf{a}.\, \mathsf{f\ 0\ a}$ with $\mathsf{f\ 0}$. If $\mathsf{A} = \varnothing$, then

$\mathsf{lift_{map}}\ (\mathsf{lazy}_\perp\ \mathsf{f})\ \mathsf{A} = \varnothing$ (the empty mapping), so

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

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

4.1.5 Correctness

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

Proof. By construction. □

Corollary 4.5 (semantic correctness)**.** *For all expressions* e*,* $[\![e]\!]_{\mathsf{map}} \equiv \mathsf{lift_{map}}\ [\![e]\!]_\perp$*.*

4.1.6 Arrow Laws

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

$$\begin{aligned}
&\mathsf{nonmonotone} : \mathsf{Bool} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Bool}\\
&\mathsf{nonmonotone}\ \mathsf{A}\ :=\ \mathsf{if}\ (\mathsf{A} = \varnothing)\ (\mathsf{mapping}\ \mathsf{id}\ \mathsf{Bool})\ \varnothing
\end{aligned} \tag{33}$$

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

Definition 4.6 (mapping arrow law)**.** *Let* $\mathsf{g} : \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}$*. If there exists an* $\mathsf{f} : \mathsf{X} \rightsquigarrow_\perp \mathsf{Y}$ *such that* $\mathsf{g} \equiv \mathsf{lift_{map}}\ \mathsf{f}$*, 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 $\mathsf{g} : \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}$. By homomorphism of $\mathsf{lift_{map}}$, mapping arrow combinators return computations that obey this law.

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

Proof. Follows from Theorem 4.4 and Definition 4.6. □

Corollary 4.8. $\mathsf{arr_{map}}$*,* $(\&\&\&_{\mathsf{map}})$*,* $(\ggg_{\mathsf{map}})$*,* $\mathsf{ifte_{map}}$ *and* $\mathsf{lazy_{map}}$ *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

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

Like a mapping, an $\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y}$ has an observable domain—but computing the table of input-output pairs is delayed. We therefore call these ***lazy preimage mappings***.

Converting a mapping to a lazy preimage mapping requires constructing a delayed application of $\mathsf{preimage}$:

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

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

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

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

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

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

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

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

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

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

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

XXX: moar text in following subsections

5.1 Preimage Mapping Pairing

Imported Lemma 5.4 ($\mathsf{preimage}$ distributes over $\langle \cdot, \cdot \rangle_{\mathsf{map}}$ and $(\times)$)**.** *Let* $\mathsf{g}_1 \in \mathsf{X} \rightharpoonup \mathsf{Y}_1$ *and* $\mathsf{g}_2 \in \mathsf{X} \rightharpoonup \mathsf{Y}_2$*. For all* $\mathsf{B}_1 \subseteq \mathsf{Y}_1$ *and* $\mathsf{B}_2 \subseteq \mathsf{Y}_2$*,* $\mathsf{preimage}\ \langle \mathsf{g}_1, \mathsf{g}_2 \rangle_{\mathsf{map}}\ (\mathsf{B}_1 \times \mathsf{B}_2) = (\mathsf{preimage}\ \mathsf{g}_1\ \mathsf{B}_1) \cap (\mathsf{preimage}\ \mathsf{g}_2\ \mathsf{B}_2)$*.*

Theorem 5.5 (pre distributes over $\langle \cdot, \cdot \rangle_{\mathsf{map}}$)**.** *Let* $\mathsf{g}_1 \in \mathsf{X} \rightharpoonup \mathsf{Y}_1$ *and* $\mathsf{g}_2 \in \mathsf{X} \rightharpoonup \mathsf{Y}_2$*. Then* $\mathsf{pre}\ \langle \mathsf{g}_1, \mathsf{g}_2 \rangle_{\mathsf{map}} \equiv \langle \mathsf{pre}\ \mathsf{g}_1, \mathsf{pre}\ \mathsf{g}_2 \rangle_{\mathsf{pre}}$*.*

$$X \underset{\text{pre}}{\rightharpoondown} Y ::= \langle \text{Set } Y, \text{Set } Y \Rightarrow \text{Set } X \rangle$$

$$\begin{aligned} &\text{pre} : (X \underset{\text{map}}{\rightsquigarrow} Y) \Rightarrow (X \underset{\text{pre}}{\rightharpoondown} Y) \\ &\text{pre } g \;:=\; \langle \text{range } g, \lambda B.\, \text{preimage } g \; B \rangle \end{aligned}$$

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

$$\begin{aligned} &\text{domain}_{\text{pre}} : (X \underset{\text{pre}}{\rightharpoondown} Y) \Rightarrow \text{Set } X \\ &\text{domain}_{\text{pre}} \; \langle Y', p \rangle \;:=\; p \; Y' \end{aligned}$$

$$\begin{aligned} &\text{range}_{\text{pre}} : (X \underset{\text{pre}}{\rightharpoondown} Y) \Rightarrow \text{Set } Y \\ &\text{range}_{\text{pre}} \; \langle Y', p \rangle \;:=\; Y' \end{aligned}$$

$$\begin{aligned} &\langle \cdot, \cdot \rangle_{\text{pre}} : (X \underset{\text{pre}}{\rightharpoondown} Y_1) \Rightarrow (X \underset{\text{pre}}{\rightharpoondown} Y_2) \Rightarrow (X \underset{\text{pre}}{\rightharpoondown} Y_1 \times Y_2) \\ &\langle \langle Y'_1, p_1 \rangle, \langle Y'_2, p_2 \rangle \rangle_{\text{pre}} \;:=\; \begin{array}{l} \text{let } \; Y' := Y'_1 \times Y'_2 \\ \quad\;\; p := \lambda B.\, \bigcup\limits_{\langle b_1, b_2 \rangle \in B} (p_1 \; \{b_1\}) \cap (p_2 \; \{b_2\}) \\ \text{in } \; \langle Y', p \rangle \end{array} \end{aligned}$$

$$\begin{aligned} &(\circ_{\text{pre}}) : (Y \underset{\text{pre}}{\rightharpoondown} Z) \Rightarrow (X \underset{\text{pre}}{\rightharpoondown} Y) \Rightarrow (X \underset{\text{pre}}{\rightharpoondown} Z) \\ &\langle Z', p_2 \rangle \circ_{\text{pre}} h_1 \;:=\; \langle Z', \lambda C.\, \text{ap}_{\text{pre}} \; h_1 \; (p_2 \; C) \rangle \end{aligned}$$

$$\begin{aligned} &(\uplus_{\text{pre}}) : (X \underset{\text{pre}}{\rightharpoondown} Y) \Rightarrow (X \underset{\text{pre}}{\rightharpoondown} Y) \Rightarrow (X \underset{\text{pre}}{\rightharpoondown} Y) \\ &h_1 \uplus_{\text{pre}} h_2 \;:=\; \begin{array}{l} \text{let } \; Y' := (\text{range}_{\text{pre}} \; h_1) \cup (\text{range}_{\text{pre}} \; h_2) \\ \quad\;\; p := \lambda B.\, (\text{ap}_{\text{pre}} \; h_1 \; B) \uplus (\text{ap}_{\text{pre}} \; h_2 \; B) \\ \text{in } \; \langle Y', p \rangle \end{array} \end{aligned}$$

Figure 5: Lazy preimage mappings and operations.

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

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

□

5.2 Preimage Mapping Composition

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

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

□

5.3 Preimage Mapping Disjoint Union

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

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

□

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 $X \underset{\text{pre}}{\rightsquigarrow} Y ::= X \underset{\text{pre}}{\rightharpoondown} Y$: 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

$$X \underset{\text{pre}}{\rightsquigarrow} Y \;::=\; \text{Set } X \Rightarrow (X \underset{\text{pre}}{\rightharpoondown} Y) \tag{37}$$

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:

$$\begin{aligned} &\mathsf{lift_{pre}} : (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}) \\ &\mathsf{lift_{pre}}\ \mathsf{g}\ \mathsf{A}\ :=\ \mathsf{pre}\ (\mathsf{g}\ \mathsf{A}) \end{aligned} \tag{38}$$

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

$$\mathsf{ap_{pre}}\ (\mathsf{lift_{pre}}\ \mathsf{g}\ \mathsf{A})\ \mathsf{B} \equiv \mathsf{preimage}\ (\mathsf{g}\ \mathsf{A})\ \mathsf{B} \tag{39}$$

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

Again, we need a coarser notion of equivalence.

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

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

$$\begin{aligned} &\mathsf{ap_{pre}}\ (\mathsf{lift_{pre}}\ (\mathsf{g_1} \mathbin{\&\&\&}_{\mathsf{map}} \mathsf{g_2})\ \mathsf{A})\ \mathsf{B} \\ &\quad\equiv\ \mathsf{ap_{pre}}\ (\mathsf{pre}\ \langle \mathsf{g_1}\ \mathsf{A}, \mathsf{g_2}\ \mathsf{A}\rangle_{\mathsf{map}})\ \mathsf{B} \\ &\quad\equiv\ \mathsf{ap_{pre}}\ \langle \mathsf{pre}\ (\mathsf{g_1}\ \mathsf{A}), \mathsf{pre}\ (\mathsf{g_2}\ \mathsf{A})\rangle_{\mathsf{pre}}\ \mathsf{B} \\ &\quad\equiv\ \mathsf{ap_{pre}}\ \langle \mathsf{lift_{pre}}\ \mathsf{g_1}\ \mathsf{A}, \mathsf{lift_{pre}}\ \mathsf{g_2}\ \mathsf{A}\rangle_{\mathsf{pre}}\ \mathsf{B} \end{aligned}$$

Substituting $\mathsf{h_1}$ for $\mathsf{lift_{pre}}\ \mathsf{g_1}$ and $\mathsf{h_2}$ for $\mathsf{lift_{pre}}\ \mathsf{g_2}$, and removing the application of $\mathsf{ap_{pre}}$ from both sides of the equivalence gives a definition of $(\mathbin{\&\&\&}_{\mathsf{pre}})$ (Figure 6) for which (17) holds.

6.2 Case: Composition

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

$$\begin{aligned} &\mathsf{ap_{pre}}\ (\mathsf{lift_{pre}}\ (\mathsf{g_1} \ggg_{\mathsf{map}} \mathsf{g_2})\ \mathsf{A})\ \mathsf{C} \\ &\quad\equiv\ \mathsf{let}\ \ \mathsf{g_1'} := \mathsf{g_1}\ \mathsf{A} \\ &\qquad\qquad\ \mathsf{g_2'} := \mathsf{g_2}\ (\mathsf{range}\ \mathsf{g_1'}) \\ &\qquad\quad\ \mathsf{in}\ \ \mathsf{ap_{pre}}\ (\mathsf{pre}\ (\mathsf{g_2'} \circ_{\mathsf{map}} \mathsf{g_1'}))\ \mathsf{C} \\ &\quad\equiv\ \mathsf{let}\ \ \mathsf{g_1'} := \mathsf{g_1}\ \mathsf{A} \\ &\qquad\qquad\ \mathsf{g_2'} := \mathsf{g_2}\ (\mathsf{range}\ \mathsf{g_1'}) \\ &\qquad\quad\ \mathsf{in}\ \ \mathsf{ap_{pre}}\ ((\mathsf{pre}\ \mathsf{g_1'}) \circ_{\mathsf{pre}} (\mathsf{pre}\ \mathsf{g_2'}))\ \mathsf{C} \\ &\quad\equiv\ \mathsf{let}\ \ \mathsf{h_1} := \mathsf{lift_{pre}}\ \mathsf{g_1}\ \mathsf{A} \\ &\qquad\qquad\ \mathsf{h_2} := \mathsf{lift_{pre}}\ \mathsf{g_2}\ (\mathsf{range_{pre}}\ \mathsf{h_1}) \\ &\qquad\quad\ \mathsf{in}\ \ \mathsf{ap_{pre}}\ (\mathsf{h_2} \circ_{\mathsf{pre}} \mathsf{h_1})\ \mathsf{C} \end{aligned} \tag{40}$$

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

6.3 Case: Conditional

Starting with the left side of (18), we expand terms, apply Theorem 5.9, rewrite in terms of $\mathsf{lift_{pre}}$, and apply Theorem 5.2 in the definitions of $\mathsf{h_2}$ and $\mathsf{h_3}$:

$$\begin{aligned} &\mathsf{ap_{pre}}\ (\mathsf{lift_{pre}}\ (\mathsf{ifte_{map}}\ \mathsf{g_1}\ \mathsf{g_2}\ \mathsf{g_3})\ \mathsf{A})\ \mathsf{B} \\ &\quad\equiv\ \mathsf{let}\ \ \mathsf{g_1'} := \mathsf{g_1}\ \mathsf{A} \\ &\qquad\qquad\ \mathsf{g_2'} := \mathsf{g_2}\ (\mathsf{preimage}\ \mathsf{g_1'}\ \{\mathsf{true}\}) \\ &\qquad\qquad\ \mathsf{g_3'} := \mathsf{g_3}\ (\mathsf{preimage}\ \mathsf{g_1'}\ \{\mathsf{false}\}) \\ &\qquad\quad\ \mathsf{in}\ \ \mathsf{ap_{pre}}\ (\mathsf{pre}\ (\mathsf{g_2'} \uplus_{\mathsf{map}} \mathsf{g_3'}))\ \mathsf{B} \\ &\quad\equiv\ \mathsf{let}\ \ \mathsf{g_1'} := \mathsf{g_1}\ \mathsf{A} \\ &\qquad\qquad\ \mathsf{g_2'} := \mathsf{g_2}\ (\mathsf{preimage}\ \mathsf{g_1'}\ \{\mathsf{true}\}) \\ &\qquad\qquad\ \mathsf{g_3'} := \mathsf{g_3}\ (\mathsf{preimage}\ \mathsf{g_1'}\ \{\mathsf{false}\}) \\ &\qquad\quad\ \mathsf{in}\ \ \mathsf{ap_{pre}}\ ((\mathsf{pre}\ \mathsf{g_2'}) \uplus_{\mathsf{pre}} (\mathsf{pre}\ \mathsf{g_3'}))\ \mathsf{B} \\ &\quad\equiv\ \mathsf{let}\ \ \mathsf{h_1} := \mathsf{lift_{pre}}\ \mathsf{g_1}\ \mathsf{A} \\ &\qquad\qquad\ \mathsf{h_2} := \mathsf{lift_{pre}}\ \mathsf{g_2}\ (\mathsf{ap_{pre}}\ \mathsf{h_1}\ \{\mathsf{true}\}) \\ &\qquad\qquad\ \mathsf{h_3} := \mathsf{lift_{pre}}\ \mathsf{g_3}\ (\mathsf{ap_{pre}}\ \mathsf{h_1}\ \{\mathsf{false}\}) \\ &\qquad\quad\ \mathsf{in}\ \ \mathsf{ap_{pre}}\ (\mathsf{h_2} \uplus_{\mathsf{pre}} \mathsf{h_3})\ \mathsf{B} \end{aligned}$$

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

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

Substituting $\mathsf{h}\ 0$ for $\mathsf{lift_{pre}}\ (\mathsf{g}\ 0)$ and removing the application of $\mathsf{ap_{pre}}$ from both sides of the equivalence gives a definition for $\mathsf{lazy_{pre}}$ (Figure 6) for which (19) holds.

6.5 Correctness

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

Proof. By construction. □

Corollary 6.3 (semantic correctness)**.** *For all expressions* e*,* $[\![e]\!]_{\mathsf{pre}} \equiv \mathsf{lift_{pre}}\ [\![e]\!]_{\mathsf{map}}$*.*

6.6 Arrow Laws

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

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

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

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

Proof. Follows from Theorem 6.2 and Definition 6.4. □

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

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

$$\begin{aligned}
&\mathsf{arr_{pre}} : (\mathsf{X} \Rightarrow \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}) \\
&\mathsf{arr_{pre}} \ := \ \mathsf{lift_{pre}} \circ \mathsf{arr_{map}}
\end{aligned}$$

$$\begin{aligned}
&(\ggg_{\mathsf{pre}}) : (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}) \Rightarrow (\mathsf{Y} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Z}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Z}) \\
&(\mathsf{h_1} \ggg_{\mathsf{pre}} \mathsf{h_2})\ \mathsf{A} \ := \ \mathsf{let}\ \ \mathsf{h'_1} := \mathsf{h_1\ A} \\
&\qquad\qquad\qquad\qquad\quad\ \ \mathsf{h'_2} := \mathsf{h_2}\ (\mathsf{range_{pre}\ h'_1}) \\
&\qquad\qquad\qquad\qquad\ \ \mathsf{in}\ \ \mathsf{h'_2} \circ_{\mathsf{pre}} \mathsf{h'_1}
\end{aligned}$$

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

$$\begin{aligned}
&\mathsf{ifte_{pre}} : (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Bool}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}) \\
&\mathsf{ifte_{pre}\ h_1\ h_2\ h_3\ A} \ := \ \mathsf{let}\ \ \mathsf{h'_1} := \mathsf{h_1\ A} \\
&\qquad\qquad\qquad\qquad\qquad\quad\ \ \mathsf{h'_2} := \mathsf{h_2}\ (\mathsf{ap_{pre}\ h'_1}\ \{\mathsf{true}\}) \\
&\qquad\qquad\qquad\qquad\qquad\quad\ \ \mathsf{h'_3} := \mathsf{h_3}\ (\mathsf{ap_{pre}\ h'_1}\ \{\mathsf{false}\}) \\
&\qquad\qquad\qquad\qquad\quad\ \ \mathsf{in}\ \ \mathsf{h'_2} \uplus_{\mathsf{pre}} \mathsf{h'_3}
\end{aligned}$$

$$\begin{aligned}
&\mathsf{lazy_{pre}} : (1 \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y})) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}) \\
&\mathsf{lazy_{pre}\ h\ A} \ := \ \mathsf{if}\ (\mathsf{A} = \varnothing)\ (\mathsf{pre}\ \varnothing)\ (\mathsf{h}\ 0\ \mathsf{A})
\end{aligned}$$

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

Figure 6: Preimage arrow definitions.

7. Preimages Under Partial Functions

We have defined everything on the top of our roadmap:

$$\begin{tikzcd}
\mathsf{X} \rightsquigarrow_{\perp} \mathsf{Y} \arrow[r, "\mathsf{lift_{map}}"] \arrow[d, "\eta_{\perp*}"'] & \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y} \arrow[r, "\mathsf{lift_{pre}}"] \arrow[d, "\eta_{\mathsf{map*}}"] & \mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y} \arrow[d, "\eta_{\mathsf{pre*}}"] \\
\mathsf{X} \rightsquigarrow_{\perp^*} \mathsf{Y} \arrow[r, "\mathsf{lift_{map*}}"'] & \mathsf{X} \underset{\mathsf{map^*}}{\rightsquigarrow} \mathsf{Y} \arrow[r, "\mathsf{lift_{pre*}}"'] & \mathsf{X} \underset{\mathsf{pre^*}}{\rightsquigarrow} \mathsf{Y}
\end{tikzcd} \tag{41}$$

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

7.1 Motivation

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

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

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

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

Suppose we interpret (42) as $\mathsf{h} : \mathsf{R} \underset{\mathsf{pre}}{\rightsquigarrow} \mathbb{N}$, a preimage arrow computation from random sources in R to natural numbers, and that we have a probability measure $\mathsf{P} \in \mathcal{P}\ \mathsf{R} \rightharpoonup [0, 1]$. We could compute the probability of any output set $\mathsf{N} \subseteq \mathbb{N}$ using $\mathsf{P}\ (\mathsf{h\ R'\ N})$, where $\mathsf{R'} \subseteq \mathsf{R}$ and $\mathsf{P\ R'} = 1$. We have three hurdles to overcome:

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

Ensuring $\mathsf{h\ R'}$ 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,* $\mathsf{j_0} \in \mathsf{J}$ *a distinguished element, and* $\mathsf{left} : \mathsf{J} \Rightarrow \mathsf{J}$ *and* $\mathsf{right} : \mathsf{J} \Rightarrow \mathsf{J}$ *be total, injective functions. If for all* $\mathsf{j} \in \mathsf{J}$, $\mathsf{j} = \mathsf{next\ j_0}$ *for some finite composition* next *of* left *and* right, *then* J, $\mathsf{j_0}$, left *and* right *define a* ***binary indexing scheme***.

For example, let J be the set of lists of $\{0, 1\}$, $\mathsf{j_0} := \langle\rangle$, and $\mathsf{left\ j} := \langle 0, \mathsf{j} \rangle$ and $\mathsf{right\ j} := \langle 1, \mathsf{j} \rangle$.

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

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

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

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

7.3 Applicative, Associative Store Transformer

We thread a random store through bottom, mapping, and preimage arrow computations by defining an **arrow transformer**: a type constructor that receives and produces an arrow type, and combinators for arrows of the produced

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

$$\begin{aligned}
&\text{arr}_{a^*} : (x \Rightarrow y) \Rightarrow (x \rightsquigarrow_{a^*} y)\\
&\text{arr}_{a^*} \ := \ \eta_{a^*} \circ \text{arr}_a
\end{aligned}$$

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

$$\begin{aligned}
&(\&\&\&_{a^*}) : (x \rightsquigarrow_{a^*} y_1) \Rightarrow (x \rightsquigarrow_{a^*} y_2) \Rightarrow (x \rightsquigarrow_{a^*} \langle y_1, y_2\rangle)\\
&(k_1\ \&\&\&_{a^*}\ k_2)\ j \ := \ k_1\ (\text{left}\ j)\ \&\&\&_a\ k_2\ (\text{right}\ j)
\end{aligned}$$

$$\begin{aligned}
&\text{ifte}_{a^*} : (x \rightsquigarrow_{a^*} \text{Bool}) \Rightarrow (x \rightsquigarrow_{a^*} y) \Rightarrow (x \rightsquigarrow_{a^*} y) \Rightarrow (x \rightsquigarrow_{a^*} y)\\
&\text{ifte}_{a^*}\ k_1\ k_2\ k_3\ j \ := \ \text{ifte}_a\ (k_1\ (\text{left}\ j))\\
&\qquad\qquad\qquad\qquad\quad (k_2\ (\text{left}\ (\text{right}\ j)))\\
&\qquad\qquad\qquad\qquad\quad (k_3\ (\text{right}\ (\text{right}\ j)))
\end{aligned}$$

$$\begin{aligned}
&\text{lazy}_{a^*} : (1 \Rightarrow (x \rightsquigarrow_{a^*} y)) \Rightarrow (x \rightsquigarrow_{a^*} y)\\
&\text{lazy}_{a^*}\ k\ j \ := \ \text{lazy}_a\ \lambda 0.\, k\ 0\ j
\end{aligned}$$

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

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

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 $x \rightsquigarrow_a y$:

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

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:

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

Because f never accesses the store, j is ignored.

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

7.4 Partial, Probabilistic Programs

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

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

Definition 7.2 (random source)**.** *Let* $R := J \to [0,1]$. *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:

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

We extend the let-calculus semantic function with

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

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

Partial Programs One utimately 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* $t \in J \to \text{Bool}_\perp$ *such that* $t\ j = \text{true}$ *or* $t\ j = \text{false}$ *for no more than finitely many* $j \in J$.

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

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

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

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

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

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 $\text{ifte}^{\Downarrow}_{a^*}$. We define a new semantic function $[\![\cdot]\!]^{\Downarrow}_{a^*}$ by

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

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

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:

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

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

The definitions of $\text{ifte}^{\Downarrow}_{a^*}$ and $[\![\cdot]\!]^{\Downarrow}_{a^*}$ 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* $\mathsf{k}_1 \equiv \mathsf{k}_2$, *when* $\mathsf{k}_1\ \mathsf{j} \equiv \mathsf{k}_2\ \mathsf{j}$ *for all* $\mathsf{j} \in \mathsf{J}$.

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

Theorem 7.5 (pure AStore arrow correctness)**.** η_{a^*} *is an arrow homomorphism.*

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

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

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

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

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

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

The relationships are more clearly expressed by

$$\begin{tikzcd} \mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y} \arrow[r, "\mathsf{lift}_{\mathsf{b}}"] \arrow[d, "\eta_{\mathsf{a}^*}"'] & \mathsf{x} \rightsquigarrow_{\mathsf{b}} \mathsf{y} \arrow[d, "\eta_{\mathsf{b}^*}"] \\ \mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y} \arrow[r, "\mathsf{lift}_{\mathsf{b}^*}"'] & \mathsf{x} \rightsquigarrow_{\mathsf{b}^*} \mathsf{y} \end{tikzcd} \tag{55}$$

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

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

Proof. Expand definitions and apply homomorphism laws (16) and (15) for $\mathsf{lift}_{\mathsf{b}}$:

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

□

Theorem 7.8 (effectful AStore arrow correctness)**.** *If* $\mathsf{lift}_{\mathsf{b}}$ *is an arrow homomorphism, then* $\mathsf{lift}_{\mathsf{b}^*}$ *is an arrow homomorphism.*

Proof. XXX: do this □

Corollary 7.9 (effectful semantic correctness)**.** *If* $\mathsf{lift}_{\mathsf{b}}$ *is an arrow homomorphism, then for all expressions e,* $[\![e]\!]_{\mathsf{b}^*} \equiv \mathsf{lift}_{\mathsf{b}^*}\ \ [\![e]\!]_{\mathsf{a}^*}$ *and* $[\![e]\!]^{\Downarrow}_{\mathsf{b}^*} \equiv \mathsf{lift}_{\mathsf{b}^*}\ \ [\![e]\!]^{\Downarrow}_{\mathsf{a}^*}$.

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

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

$$\begin{tikzcd} \mathsf{X} \rightsquigarrow_{\perp} \mathsf{Y} \arrow[r, "\mathsf{lift}_{\mathsf{map}}"] \arrow[d, "\eta_{\perp^*}"'] & \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y} \arrow[r, "\mathsf{lift}_{\mathsf{pre}}"] \arrow[d, "\eta_{\mathsf{map}^*}"] & \mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y} \arrow[d, "\eta_{\mathsf{pre}^*}"] \\ \mathsf{X} \rightsquigarrow_{\perp^*} \mathsf{Y} \arrow[r, "\mathsf{lift}_{\mathsf{map}^*}"'] & \mathsf{X} \underset{\mathsf{map}^*}{\rightsquigarrow} \mathsf{Y} \arrow[r, "\mathsf{lift}_{\mathsf{pre}^*}"'] & \mathsf{X} \underset{\mathsf{pre}^*}{\rightsquigarrow} \mathsf{Y} \end{tikzcd} \tag{56}$$

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

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

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

7.6 Convergence

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

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

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

The maximal domain of $\mathsf{k} : \mathsf{X} \rightsquigarrow_{\mathsf{a}^*} \mathsf{Y}$ *is that of* $\mathsf{k}\ \mathsf{j}_0$.

Because the above statements imply 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 $\mathsf{lift}_{\mathsf{map}}\ \mathsf{f}$ is the same as f's, and the maximal domain of $\mathsf{lift}_{\mathsf{pre}^*}\ \mathsf{g}$ is the same as g's.

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

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

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

For any input for which $[\![e]\!]_{\perp^*}$ converges, there should be a branch trace for which $[\![e]\!]^{\Downarrow}_{\perp^*}$ returns the correct output; it should otherwise return $\perp$.

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

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

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

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

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

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

Definition 7.15 (index prefix)**.** *$\mathsf{J}' \subset \mathsf{J}$ is an* ***index prefix*** *of J if $\mathsf{J}' = \{\mathsf{j}_0\}$ or at least one of the following is true:*

- *For some finite prefix J'' and $\mathsf{j} \in \mathsf{J}''$, $\mathsf{J}' = \mathsf{J}'' \uplus \{\mathsf{left}\ \mathsf{j}\}$.*
- *For some finite prefix J'' and $\mathsf{j} \in \mathsf{J}''$, $\mathsf{J}' = \mathsf{J}'' \uplus \{\mathsf{right}\ \mathsf{j}\}$.*

The corresponding ***index suffix*** is $\mathsf{J} \backslash \mathsf{J}'$, which is closed under left and right.

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

Theorem 7.16. *Let $\mathsf{f} := \llbracket e \rrbracket_{\perp *} : \mathsf{X} \rightsquigarrow_{\perp *} \mathsf{Y}$ with maximal domain A^*, and $\mathsf{f}' := \llbracket e \rrbracket^{\Downarrow}_{\perp *}$. For all $\mathsf{a} \in ((\mathsf{R} \times \mathsf{T}) \times \mathsf{X}) \backslash \mathsf{A}^*$, $\mathsf{f}'\ \mathsf{j}_0\ \mathsf{a} = \perp$.*

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

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

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

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

Corollary 7.18. *Let $\mathsf{f}' := \llbracket e \rrbracket^{\Downarrow}_{\perp *} : \mathsf{X} \rightsquigarrow_{\perp *} \mathsf{Y}$ with maximal domain A^*, and $\mathsf{f} := \llbracket e \rrbracket_{\perp *}$. For all $\mathsf{a} \in \mathsf{A}^*$, $\mathsf{f}'\ \mathsf{j}_0\ \mathsf{a} = \mathsf{f}\ \mathsf{j}_0\ \mathsf{a}$.*

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

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

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

8. Probabilities

We have not assigned probabilities to any output sets yet.

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

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

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

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

$$\mathsf{P}\ (\mathsf{image}\ (\mathsf{fst} \ggg \mathsf{fst})\ \mathsf{A}) \tag{60}$$

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

P is partial, so not every preimage set has a sensible measure. Sets that do are called *measurable*. Computing preimages and projecting onto R must preserve measurability.

8.1 Measurability

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

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

Definition 8.1 (σ-algebra, measurable set)**.** *A collection of sets $\mathcal{A} \subseteq \mathcal{P}\ \mathsf{X}$ is called a* ***σ-algebra*** *on X if it contains X and is closed under complements and countable unions. The sets in $\mathcal{A}$ are called* ***measurable sets****.*

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

The analogue of continuity is measurability.

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

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

Product spaces are defined the same way as in topology.

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

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

8.2 Measurable Pure Computations

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

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

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

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

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

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

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

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

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

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

8.2.1 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* $\mathsf{g} : \mathsf{X} \rightharpoonup \mathsf{Y}$ *and* $\mathsf{B} \subseteq \mathsf{Y}$*. Then* $\mathsf{image\ g\ (preimage\ g\ B)} \subseteq \mathsf{B}$*.*

Imported Lemma 8.10 (expanded post-composition)**.** *Let* $\mathsf{g}_1 : \mathsf{X} \rightharpoonup \mathsf{Y}$ *and* $\mathsf{g}_2 : \mathsf{Y} \rightharpoonup \mathsf{Z}$ *such that* $\mathsf{range\ g}_1 \subseteq \mathsf{domain\ g}_2$*, and let* $\mathsf{g}_2' : \mathsf{Y} \rightharpoonup \mathsf{Z}$ *such that* $\mathsf{g}_2 \subseteq \mathsf{g}_2'$*. Then* $\mathsf{g}_2 \circ_{\mathsf{map}} \mathsf{g}_1 = \mathsf{g}_2' \circ_{\mathsf{map}} \mathsf{g}_1$*.*

Theorem 8.11 (mapping arrow monotonicity)**.** *Let* $\mathsf{g} : \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}$*. For any* $\mathsf{A}' \subseteq \mathsf{A} \subseteq \mathsf{A}^*$*,* $\mathsf{g\ A}' \subseteq \mathsf{g\ A}$*.*

Proof. Follows from Definition 4.6. □

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

Proof. Follows from Theorem 7.13. □

Now we can prove measurability.

Imported Lemma 8.13 (measurability under $\circ_{\mathsf{map}}$)**.** *If* $\mathsf{g}_1 : \mathsf{X} \rightharpoonup \mathsf{Y}$ *is* $\mathcal{A}$-$\mathcal{B}$*-measurable and* $\mathsf{g}_2 : \mathsf{Y} \rightharpoonup \mathsf{Z}$ *is* $\mathcal{B}$-$\mathcal{C}$*-measurable, then* $\mathsf{g}_2 \circ_{\mathsf{map}} \mathsf{g}_1$ *is* $\mathcal{A}$-$\mathcal{C}$*-measurable.*

Theorem 8.14 (measurability under $(\ggg_{\mathsf{map}})$)**.** *If* $\mathsf{g}_1 : \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}$ *is* $\mathcal{A}$-$\mathcal{B}$*-measurable and* $\mathsf{g}_2 : \mathsf{Y} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Z}$ *is* $\mathcal{B}$-$\mathcal{C}$*-measurable, then* $\mathsf{g}_1 \ggg_{\mathsf{map}} \mathsf{g}_2$ *is* $\mathcal{A}$-$\mathcal{C}$*-measurable.*

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

$$\begin{aligned}(\mathsf{g}_1 \ggg_{\mathsf{map}} \mathsf{g}_2)\ \mathsf{A}^{**} \;=\; & \mathsf{let}\ \ \mathsf{g}_1' := \mathsf{g}_1\ \mathsf{A}^{**} \\ & \qquad \mathsf{g}_2' := \mathsf{g}_2\ (\mathsf{range\ g}_1') \\ & \mathsf{in}\ \ \mathsf{g}_2' \circ_{\mathsf{map}} \mathsf{g}_1'\end{aligned} \tag{61}$$

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 $\mathsf{range\ g}_1' \notin \mathcal{B}$.

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

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

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

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

8.2.2 Case: Pairing

Imported Lemma 8.15 (measurability under $\langle\cdot,\cdot\rangle_{\mathsf{map}}$)**.** *If* $\mathsf{g}_1 : \mathsf{X} \rightharpoonup \mathsf{Y}_1$ *is* $\mathcal{A}$-$\mathcal{B}_1$*-measurable and* $\mathsf{g}_2 : \mathsf{X} \rightharpoonup \mathsf{Y}_2$ *is* $\mathcal{A}$-$\mathcal{B}_2$*-measurable, then* $\langle \mathsf{g}_1, \mathsf{g}_2 \rangle_{\mathsf{map}}$ *is* $\mathcal{A}$-$(\mathcal{B}_1 \otimes \mathcal{B}_2)$*-measurable.*

Theorem 8.16 (measurability under $(\&\&\&_{\mathsf{map}})$)**.** *If* $\mathsf{g}_1 : \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}_1$ *is* $\mathcal{A}$-$\mathcal{B}_1$*-measurable and* $\mathsf{g}_2 : \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}_2$ *is* $\mathcal{A}$-$\mathcal{B}_2$*-measurable, then* $\mathsf{g}_1\ \&\&\&_{\mathsf{map}}\ \mathsf{g}_2$ *is* $\mathcal{A}$-$(\mathcal{B}_1 \otimes \mathcal{B}_2)$*-measurable.*

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

8.2.3 Case: Conditional

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

Theorem 8.18 (measurability under $\mathsf{ifte}_{\mathsf{map}}$)**.** *If* $\mathsf{g}_1 : \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Bool}$ *is* $\mathcal{A}$-$(\mathcal{P}\ \mathsf{Bool})$*-measurable, and* $\mathsf{g}_2 : \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}$ *and* $\mathsf{g}_3 : \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}$ *are* $\mathcal{A}$-$\mathcal{B}$*-measurable, then* $\mathsf{ifte}_{\mathsf{map}}\ \mathsf{g}_1\ \mathsf{g}_2\ \mathsf{g}_3$ *is* $\mathcal{A}$-$\mathcal{B}$*-measurable.*

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

$$\begin{aligned}\mathsf{A}_2^{**} \;&:=\; \mathsf{A}_2^* \cap \mathsf{preimage\ (g_1}\ \mathcal{A}_1^*)\ \{\mathsf{true}\} \\ \mathsf{A}_3^{**} \;&:=\; \mathsf{A}_3^* \cap \mathsf{preimage\ (g_1}\ \mathcal{A}_1^*)\ \{\mathsf{false}\} \\ \mathsf{A}^{**} \;&:=\; \mathsf{A}_2^{**} \uplus \mathsf{A}_3^{**}\end{aligned} \tag{62}$$

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

$$\begin{aligned}\mathsf{ifte}_{\mathsf{map}}\ \mathsf{g}_1\ \mathsf{g}_2\ \mathsf{g}_3\ \mathsf{A}^{**} \;=\; & \mathsf{let}\ \ \mathsf{g}_1' := \mathsf{g}_1\ \mathsf{A}^{**} \\ & \qquad \mathsf{g}_2' := \mathsf{g}_2\ (\mathsf{preimage\ g}_1'\ \{\mathsf{true}\}) \\ & \qquad \mathsf{g}_3' := \mathsf{g}_3\ (\mathsf{preimage\ g}_1'\ \{\mathsf{false}\}) \\ & \mathsf{in}\ \ \mathsf{g}_2' \uplus_{\mathsf{map}} \mathsf{g}_3'\end{aligned} \tag{63}$$

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

8.2.4 Case: Laziness

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

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

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

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

$$\mathsf{lazy}_{\mathsf{map}}\ \mathsf{g}\ \mathsf{A}^{**} \ = \ \mathsf{if}\ (\mathsf{A}^{**} = \varnothing)\ \varnothing\ (\mathsf{g}\ 0\ \mathsf{A}^{**}) \tag{64}$$

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

8.3 Measurable Probabilistic Computations

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

Definition 8.21 (measurable mapping* arrow computation)**.** *Let $\mathcal{A}$ and $\mathcal{B}$ be σ-algebras on $(\mathsf{R} \times \mathsf{T}) \times \mathsf{X}$ and Y. A computation $\mathsf{g} : \mathsf{X} \underset{\mathsf{map}^*}{\rightsquigarrow} \mathsf{Y}$ is **$\mathcal{A}$-$\mathcal{B}$-measurable** if $\mathsf{g}\ \mathsf{j}_0$ is an $\mathcal{A}$-$\mathcal{B}$-measurable mapping arrow computation.*

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

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

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

$$\Sigma\ \langle \mathsf{X}_1, \mathsf{X}_2 \rangle = \Sigma\ \mathsf{X}_1 \otimes \Sigma\ \mathsf{X}_2 \tag{65}$$

$$\Sigma\ (\mathsf{J} \to \mathsf{X}) = (\Sigma\ \mathsf{X})^{\otimes \mathsf{J}} \tag{66}$$

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

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

$$\Sigma\ \mathsf{T} \ ::= \ \mathcal{P}\ \mathsf{T} \tag{68}$$

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

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

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

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

Theorem 8.26. $\mathsf{ifte}^{\Downarrow}_{\mathsf{map}^*}$ *is measurable.*

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

Theorem 8.27 (finite expressions are measurable)**.** *For any expression e lacking first-order applications, $[\![e]\!]_{\mathsf{map}^*}$ is measurable.*

Proof. By structural induction and the above theorems. □

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

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

Theorem 8.29 (all probabilistic expressions are measurable)**.** *For all expressions e, $[\![e]\!]^{\Downarrow}_{\mathsf{map}^*}$ is measurable.*

Proof. Let $\mathsf{g} := [\![e]\!]^{\Downarrow}_{\mathsf{map}^*}$ and $\mathsf{g}' := \mathsf{g}\ \mathsf{j}_0\ ((\mathsf{R} \times \mathsf{T}) \times \mathsf{X})$. By Corollary 7.19, $\mathsf{g}' = \mathsf{g}\ \mathsf{j}_0\ \mathsf{A}^*$ where A^* is g's maximal domain; thus we need only show that g' is a measurable mapping.

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

$$\mathsf{g}' \ = \ \bigcup_{\mathsf{t} \in \mathsf{T}} \mathsf{g}\ \mathsf{j}_0\ ((\mathsf{R} \times \{\mathsf{t}\}) \times \mathsf{X}) \tag{69}$$

By Theorem 8.28, for every $\mathsf{t} \in \mathsf{T}$, there is an expression that computes $\mathsf{g}\ ((\mathsf{R} \times \{\mathsf{t}\}) \times \mathsf{X})$. By (68) and Theorem 8.27, each is measurable. By the mapping arrow law (Definition 4.6), each is disjoint. By Lemma 8.17, their union is measurable. □

Theorem 8.29 remains true when $[\![\cdot]\!]_{\mathsf{a}}$ is extended with any rule whose right side is measurable, including rules for real arithmetic, equality, inequality and limits. More generally, any continuous or (countably) piecewise continuous function can be made available as a language primitive, as long as its domain's and codomain's standard σ-algebras are generated from their topologies.

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

8.4 Measurable Projections

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

$$\mathsf{P}\ (\mathsf{image}\ (\mathsf{fst} \ggg \mathsf{fst})\ (\mathsf{preimage}\ (\mathsf{g}\ \mathsf{j}_0\ \mathsf{A}^*)\ \mathsf{B})) \tag{70}$$

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

Theorem 8.30 (measurable finite projections)**.** *Let $\mathsf{A} \in \Sigma\ \langle \mathsf{X}_1, \mathsf{X}_2 \rangle$. If X_2 is at most countable and $\Sigma\ \mathsf{X}_2 = \mathcal{P}\ \mathsf{X}_2$, then* $\mathsf{image}\ \mathsf{fst}\ \mathsf{A} \in \mathcal{A}_1$.

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

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

$$\mathsf{image}\ (\mathsf{fst} \ggg \mathsf{fst})\ (\mathsf{preimage}\ (\mathsf{g}\ \mathsf{j}_0\ \mathsf{A}^*)\ \mathsf{B}) \ \in \ \Sigma\ \mathsf{R} \tag{71}$$

Proof. T is countable and $\Sigma\ \mathsf{T} = \mathcal{P}\ \mathsf{T}$ by definition (68); apply Theorem 8.30 twice. □

In particular, for any well-defined $[\![p]\!]^{\Downarrow}_{\mathsf{map}^*} : \{\langle\rangle\} \underset{\mathsf{map}^*}{\rightsquigarrow} \mathsf{Y}$, 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: $\mathsf{preimage}\ g\ B$ is uncomputable for most uncountable sets B no matter how cleverly they are represented. Further, because pre, $\mathsf{lift}_{\mathsf{pre}}$ and $\mathsf{arr}_{\mathsf{pre}}$ are ultimately defined in terms of $\mathsf{preimage}$, we cannot implement them.

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

Figure 8 gives expressions equivalent to $\mathsf{arr}_{\mathsf{pre}}\ \mathsf{id}$, $\mathsf{arr}_{\mathsf{pre}}\ \mathsf{fst}$, $\mathsf{arr}_{\mathsf{pre}}\ \mathsf{snd}$, $\mathsf{arr}_{\mathsf{pre}}\ (\mathsf{const}\ b)$ and $\mathsf{arr}_{\mathsf{pre}}\ (\pi\ j)$. (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:

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

(72)

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

9.2 Rectangular Families

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

$$\mathsf{Rect}\ \langle X_1, X_2\rangle \ = \ (\mathsf{Rect}\ X_1) \boxtimes (\mathsf{Rect}\ X_2) \tag{73}$$

$$\mathsf{Rect}\ (J \to X) \ = \ (\mathsf{Rect}\ X)^{\boxtimes J} \tag{74}$$

where

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

$$\mathcal{A}^{\boxtimes J} \ := \ \{\textstyle\prod_{j \in J} A_j \mid \forall j \in J.\, A_j \in \mathcal{A}\} \tag{76}$$

lift cartesian products to sets of sets.

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

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

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

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

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

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

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

$$t\ j' = \begin{cases} b & j' = j \\ \text{any member of } \mathsf{project}\ j'\ T' & \perp \notin \mathsf{project}\ j'\ T' \\ \perp & \perp \in \mathsf{project}\ j'\ T' \end{cases} \tag{77}$$

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

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

9.3 Approximate Preimage Mapping Operations

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

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

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

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

It is not hard to show that the following replacement:

$$\begin{aligned} &\langle\cdot,\cdot\rangle'_{\mathsf{pre}} : (X \underset{\mathsf{pre}}{\rightharpoonup} Y_1) \Rightarrow (X \underset{\mathsf{pre}}{\rightharpoonup} Y_2) \Rightarrow (X \underset{\mathsf{pre}}{\rightharpoonup} Y_1 \times Y_2) \\ &\langle\langle Y'_1, p_1\rangle, \langle Y'_2, p_2\rangle\rangle'_{\mathsf{pre}} \ := \\ &\quad \langle Y'_1 \times Y'_2, \lambda B.\, p_1\ (\mathsf{proj}_{\mathsf{fst}}\ B) \cap p_2\ (\mathsf{proj}_{\mathsf{snd}}\ B)\rangle \end{aligned} \tag{78}$$

computes covering rectangles of preimages under pairing.

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

$$\begin{aligned} (A_1 \times A_2) \vee (B_1 \times B_2) \ &= \ (A_1 \vee B_1) \times (A_2 \vee B_2) \\ (\textstyle\prod_{j \in J} A_j) \vee (\prod_{j \in J} B_j) \ &= \ \textstyle\prod_{j \in J} A_j \vee B_j \end{aligned} \tag{79}$$

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

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

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

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

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

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

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

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

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

$$\begin{aligned}
&(\uplus'_{\mathsf{pre}}) : (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y}) \\
&\mathsf{h}_1 \uplus'_{\mathsf{pre}} \mathsf{h}_2 := \mathsf{let}\ \ \mathsf{Y}' := (\mathsf{range}_{\mathsf{pre}}\ \mathsf{h}_1) \vee (\mathsf{range}_{\mathsf{pre}}\ \mathsf{h}_2) \\
&\qquad\qquad\qquad\quad \mathsf{p} := \lambda \mathsf{B}.\, (\mathsf{ap}_{\mathsf{pre}}\ \mathsf{h}_1\ \mathsf{B}) \vee (\mathsf{ap}_{\mathsf{pre}}\ \mathsf{h}_2\ \mathsf{B}) \\
&\qquad\qquad\quad\ \mathsf{in}\ \ \langle \mathsf{Y}', \mathsf{p}\rangle
\end{aligned} \tag{80}$$

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

9.4 Partial Programs

XXX: too much yanking the user's chain here

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

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

$$\begin{aligned}
&\mathsf{arr}_{\mathsf{map}}\ \mathsf{agrees}\ (\mathsf{Bool} \times \mathsf{Bool}) = \\
&\quad \{\langle\langle \mathsf{true}, \mathsf{true}\rangle, \mathsf{true}\rangle, \langle\langle \mathsf{false}, \mathsf{false}\rangle, \mathsf{false}\rangle\}
\end{aligned} \tag{81}$$

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

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

$$\begin{aligned}
&\mathsf{ifte}^{\Downarrow}_{\mathsf{pre*}}\ \mathsf{k}_1\ \mathsf{k}_2\ \mathsf{k}_3\ \mathsf{j}\ \mathsf{A} \equiv \\
&\quad \mathsf{let}\ \ \langle \mathsf{C}_{\mathsf{k}}, \mathsf{p}_{\mathsf{k}}\rangle := \mathsf{k}_1\ \mathsf{j}_1\ \mathsf{A} \\
&\qquad\quad\ \langle \mathsf{C}_{\mathsf{b}}, \mathsf{p}_{\mathsf{b}}\rangle := \mathsf{branch}_{\mathsf{pre*}}\ \mathsf{j}\ \mathsf{A} \\
&\qquad\qquad\quad \mathsf{C}_2 := \mathsf{C}_{\mathsf{k}} \cap \mathsf{C}_{\mathsf{b}} \cap \{\mathsf{true}\} \\
&\qquad\qquad\quad \mathsf{C}_3 := \mathsf{C}_{\mathsf{k}} \cap \mathsf{C}_{\mathsf{b}} \cap \{\mathsf{false}\} \\
&\qquad\qquad\quad \mathsf{A}_2 := \mathsf{p}_{\mathsf{k}}\ \mathsf{C}_2 \cap \mathsf{p}_{\mathsf{b}}\ \mathsf{C}_2 \\
&\qquad\qquad\quad \mathsf{A}_3 := \mathsf{p}_{\mathsf{k}}\ \mathsf{C}_3 \cap \mathsf{p}_{\mathsf{b}}\ \mathsf{C}_3 \\
&\quad\ \ \mathsf{in}\ \ (\mathsf{k}_2\ \mathsf{j}_2\ \mathsf{A}_2) \uplus_{\mathsf{pre}} (\mathsf{k}_3\ \mathsf{j}_3\ \mathsf{A}_3)
\end{aligned} \tag{82}$$

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

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

The returned preimage mapping's range is a subset of Y, and the preimages it computes must be subsets of $\mathsf{A}_2 \vee \mathsf{A}_3$. Therefore, we might replace the let body in (82) with

$$\begin{aligned}
&\mathsf{if}\ (\mathsf{C}_{\mathsf{b}} = \{\mathsf{true}, \mathsf{false}\}) \\
&\quad \langle \mathsf{Y}, \lambda \mathsf{B}.\, \mathsf{A}_2 \vee \mathsf{A}_3\rangle \\
&\quad (\mathsf{k}_2\ \mathsf{j}_2\ \mathsf{A}_2 \uplus_{\mathsf{pre}} \mathsf{k}_3\ \mathsf{j}_3\ \mathsf{A}_3)
\end{aligned} \tag{83}$$

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

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

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

9.5 Correctness

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

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

$$\begin{aligned}
\mathsf{refine}\ \mathsf{A} &:= \mathsf{ap}_{\mathsf{pre}}\ (\mathsf{h}\ \mathsf{j}_0\ \mathsf{A})\ \mathsf{B} \\
\mathsf{refine}'\ \mathsf{A} &:= \mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}'\ \mathsf{j}_0\ \mathsf{A})\ \mathsf{B}
\end{aligned} \tag{84}$$

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

Proof. By construction. □

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

Proof. XXX: todo □

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

Proof. XXX: todo □

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$$\begin{aligned}
\mathsf{fst}'_{\mathsf{pre*}}\ &:= \eta'_{\mathsf{pre*}}\ \mathsf{fst}_{\mathsf{pre}} \\
\mathsf{snd}'_{\mathsf{pre*}}\ &:= \eta'_{\mathsf{pre*}}\ \mathsf{snd}_{\mathsf{pre}};\ \cdots
\end{aligned}$$

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

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

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

9.6 Preimage Refinement

With disjointness and monotonicity properties, it is natural to suppose that we can compute probabilities of preimages of B by computing preimages with respect to increasingly fine discretizations of A. For example, starting with any partition $\mathcal{A} : \mathsf{Set}\ (\mathsf{Rect}\ \langle \mathsf{S}, \mathsf{X} \rangle)$, we might repeat the following:

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

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

In general, however, we are computing the **Jordan outer measure** (XXX: cite) of the preimage of B, which is not always equal to its measure. An example of a program for which preimage refinement diverges is $\mathsf{rational?\ random}$, where $\mathsf{rational?}$ returns true when its argument is rational and diverges otherise: the preimage of $\{\mathsf{true}\}$ has measure 0, but its Jordan outer measure is 1.

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

10. Implementations

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

10.1 Direct Implementations

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

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

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

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

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

10.2 Dr. Bayes

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

Dr. Bayes's preimage arrow implementation operates on a monomorphic rectangular set data type. It includes floating-point intervals to overapproximate real intervals, with which we compute approximate preimages under arithmetic and inequalities. Finding the smallest covering rectangle for images and preimages under $\mathsf{add} : \langle \mathbb{R}, \mathbb{R} \rangle \Rightarrow \mathbb{R}$ and other monotone functions is fairly straightforward. For piecewise monotone functions, we distinguish cases using $\mathsf{ifte}_{\mathsf{pre}}$; e.g.

$$\begin{array}{l}
\mathsf{mul}_{\mathsf{pre}} := \mathsf{ifte}_{\mathsf{pre}}\ (\mathsf{fst}_{\mathsf{pre}} \ggg_{\mathsf{pre}} \mathsf{positive?}_{\mathsf{pre}}) \\
\qquad\qquad\quad (\mathsf{ifte}_{\mathsf{pre}}\ (\mathsf{snd}_{\mathsf{pre}} \ggg_{\mathsf{pre}} \mathsf{positive?}_{\mathsf{pre}}) \\
\qquad\qquad\qquad\quad \mathsf{mul}^{++}_{\mathsf{pre}} \\
\qquad\qquad\qquad\quad (\mathsf{ifte}_{\mathsf{pre}}\ (\mathsf{snd}_{\mathsf{pre}} \ggg_{\mathsf{pre}} \mathsf{negative?}_{\mathsf{pre}}) \\
\qquad\qquad\qquad\qquad\quad \mathsf{mul}^{+-}_{\mathsf{pre}} \\
\qquad\qquad\qquad\qquad\quad (\mathsf{const}_{\mathsf{pre}}\ 0))) \\
\qquad\qquad\quad \ldots
\end{array} \tag{85}$$

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

Section 9.6 outlines preimage refinement: a discretization algorithm that seems to converge for programs that halt with probability 1, consisting of repeatedly refining a program's domain and repartitioning it. We do not use this algorithm 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 do not care much about measuring preimages for its own sake. They are concerned with the renormalized distribution of outputs given some condition. Further, they are perfectly satisfied with sampling methods, which are usually much more efficient than methods based on enumeration.

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

$$\mathsf{Pr}[\mathsf{A}^* | \mathsf{A}] \ := \ \mathsf{P}\ (\mathsf{A}^* \cap \mathsf{A}) / \mathsf{P}\ \mathsf{A} \tag{86}$$

One way to approximately compute probabilities is using **rejection sampling**. Given a list of samples xs of X,

$$\mathsf{P}\ \mathsf{A} \ \approx \ \frac{\mathsf{length}\ (\mathsf{filter}\ (\in \mathsf{A})\ \mathsf{xs})}{\mathsf{length}\ \mathsf{xs}} \tag{87}$$

Here, "$\approx$" roughly denotes probabilistic convergence as the length of xs increases. Combining (86) and (87) yields

$$\begin{aligned}
\mathsf{Pr}[\mathsf{A}^* | \mathsf{A}] \ &= \ \mathsf{P}\ (\mathsf{A}^* \cap \mathsf{A}) / \mathsf{P}\ \mathsf{A} \\
&\approx \ \frac{\mathsf{length}\ (\mathsf{filter}\ (\in \mathsf{A}^* \cap \mathsf{A})\ \mathsf{xs}) / \mathsf{length}\ \mathsf{xs}}{\mathsf{length}\ (\mathsf{filter}\ (\in \mathsf{A})\ \mathsf{xs}) / \mathsf{length}\ \mathsf{xs}} \\
&\approx \ \frac{\mathsf{length}\ (\mathsf{filter}\ (\in \mathsf{A}^* \cap \mathsf{A})\ \mathsf{xs})}{\mathsf{length}\ (\mathsf{filter}\ (\in \mathsf{A})\ \mathsf{xs})}
\end{aligned} \tag{88}$$

to compute approximate conditional probabilities.

The probability that any given element of xs is in A can be extremely small, so it would clearly be best to sample only within A. While we cannot do that, we can quite easily sample xs from a rectangular cover $\mathsf{A}' \supseteq \mathsf{A}$.

To compute these probabilities, we use the fact that A and A^* are preimages:

$$\begin{aligned} \mathsf{filter}\ (\in \mathsf{A})\ \mathsf{xs} &= \mathsf{filter}\ (\in \mathsf{preimage}\ (\mathsf{g}\ \mathsf{X})\ \mathsf{B})\ \mathsf{xs} \\ &= \mathsf{filter}\ (\lambda \mathsf{a}.\, \mathsf{g}\ \mathsf{X}\ \mathsf{a} \in \mathsf{B})\ \mathsf{xs} \\ &= \mathsf{filter}\ (\lambda \mathsf{a}.\, \mathsf{f}\ \mathsf{a} \in \mathsf{B})\ \mathsf{xs} \end{aligned} \tag{89}$$

Here, $\mathsf{f} : \mathsf{X} \rightsquigarrow_{\bot} \mathsf{Y}$ is the interpretation of the program as a bottom arrow computation. Thus,

$$\mathsf{Pr}[\mathsf{A}^*|\mathsf{A}] \approx \frac{\mathsf{filter}\ (\lambda \mathsf{a}.\, \mathsf{f}\ \mathsf{a} \in \mathsf{B}^* \cap \mathsf{B})\ \mathsf{xs}}{\mathsf{filter}\ (\lambda \mathsf{a}.\, \mathsf{f}\ \mathsf{a} \in \mathsf{B})\ \mathsf{xs}} \tag{90}$$

converges to the probability of B^* given B as the number of samples xs from the rectangular cover A' increases.

The rectangular cover A' does not have to be enumerated: the rectangles that comprise it can be sampled. Sampling from each sampled rectangle yields xs.

The preceeding discussion does not treat
$\mathsf{P}\ (\mathsf{image}\ (\mathsf{fst} \ggg \mathsf{fst})\ \mathsf{A})$
about computations of type $\mathsf{X} \rightsquigarrow_{\bot} \mathsf{Y}$, $\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}$ and $\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow}' \mathsf{Y}$. Sampling correctly from preimages computed by $\mathsf{X} \underset{\mathsf{pre}^*}{\rightsquigarrow}' \mathsf{Y}$

For programs interpreted using $[\![\cdot]\!]^{\Downarrow}_{\bot *}$ and $[\![\cdot]\!]^{\Downarrow}_{\mathsf{pre}*}$

11. Unrelated Work

XXX: todo

12. Related Work

XXX: todo

13. Conclusions and Future Work

XXX: todo

References

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

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