Running Probabilistic Programs Backward

Neil Toronto Jay McCarthy

PLT @ Brigham Young University

ntoronto@racket-lang.org jay@cs.byu.edu

Abstract

Problem: doesn't exist: Turing-equivalent, probabilistic programming language with a semantics and implementation that supports any kind of probability measure and—critically—can do probabilistic conditioning

Why problem:

Solution:

Why solution:

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

General Terms XXX, XXX

Keywords XXX, XXX

1. Introduction

There is currently no efficient probabilistic language implementation that simultaneously

1. Places no extraneous restrictions on legal programs.
2. Allows **conditioning**, or restricting the output in a way that preserves relative probabilities.
3. Has a formal semantics. (XXX: implies Coq; change terminology—possibly "mathematical specification")

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

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

These common restrictions arise from reasoning about probability using **densities**, which are functions from random values to *changes* in probability. While simple and convenient, densities have many limitations. Densities for random values with different dimension are incomparable. Densities cannot be defined on infinite products. Densities can only be used to reason about conditioning in limited cases.

Densities cannot define distributions of discontinuous functions of random variables. For example, suppose we want to model a thermometer that reports in the range $[0, 100]$, and that the temperature it would report (if it could) is distributed according to a bell curve. We might encode the process like this:

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

While t's distribution has a density (a standard bell curve at mean μ), the distribution of t' does not. Using densities, this program cannot have a meaning.

The restrictions placed on legal programs are indeed onerous, if uses of such benign functions as min and max are suspect. (XXX: that sentence bugs Jay for some reason) 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**

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

defines the distribution over subsets B of Y. In the thermometer example (1), f would be an interpretation of the program as a function, X would be the set of all random sources, and Y would be the set of the program's possible outputs. For any $\mathsf{B} \subseteq \mathsf{Y}$, preimage f B would be well-defined, regardless of discontinuities.

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

1.2 Measure-Theoretic Semantics

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

It seems this approach should make it easy to interpret probabilistic programs measure-theoretically. For a probabilistic program $\mathsf{f} : \mathsf{X} \to \mathsf{Y}$, the probability measure on output sets $\mathsf{B} \subseteq \mathsf{Y}$ should be defined by preimages of B under f and the probability measure on X. Unfortunately, it is difficult to turn this simple-sounding idea into a compositional semantics, for the following reasons.

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

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

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

We address both 1 and 4 by developing our semantics in λ_{ZFC} [2], a λ-calculus with infinite sets, and both extensional and intensional functions. We address 5 by deriving and implementing a *conservative approximation* of the semantics.

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

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

1.3 Arrow Solution Overview

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

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

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

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

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. XXX: try to put a lift from $X \underset{\mathsf{pre}^*}{\rightsquigarrow} Y$ to $X \underset{\mathsf{pre}^*}{\rightsquigarrow}' Y$ on the diagram

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

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

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

2. Operational Metalanguage

We write all of the programs in this paper in λ_{ZFC} [2], an untyped, call-by-value lambda calculus designed for deriving implementable programs from contemporary mathematics.

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

In λ_{ZFC}, essentially every set is a value, as well as every lambda and every set of lambdas. All operations, including operations on infinite sets, are assumed to complete instantly if they converge.[1]

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

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

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

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

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

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

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

[2] Assuming the existence of an inaccessible cardinal.

The type $\mathsf{Set\ X}$ denotes the same values as the powerset $\mathcal{P}\ \mathsf{X}$, or *subsets* of X. Similarly, the type $\langle \mathsf{X}, \mathsf{Y} \rangle$ denotes the same values as the product set $\mathsf{X} \times \mathsf{Y}$. There is no subtyping.

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

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

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 $\mathcal{A}$, $\mathsf{image\ f\ A}$ applies f to each member of A and reduces to the set of results, and $|\mathsf{A}|$ reduces to the cardinality of A.

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

We import applicable ZFC theorems as lemmas.

2.1 Internal and External Equality

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

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

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

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

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

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

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

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

2.2 Additional Functions and Forms

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

A less typical set operation we use is disjoint union:

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

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

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

Syntax for defining unnamed mappings is defined by

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

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

Figure 1 defines more 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. Projections are produced by

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

which is particularly useful when f is unnamed.

3. Arrows and First-Order Semantics

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

3.1 Alternative Arrow Definitions

We do not give typical minimal arrow definitions. For each arrow a, instead of $\mathsf{first}_{\mathsf{a}}$, we define $(\&\&\&_{\mathsf{a}})$—typically called **fanout**, but its use will be clearer if we call it **pairing**—which applies two functions to an input and returns the pair of their outputs. Though $\mathsf{first}_{\mathsf{a}}$ may be defined in terms of $(\&\&\&_{\mathsf{a}})$ and vice-versa [1], we give $(\&\&\&_{\mathsf{a}})$ definitions because the applicable measure-theoretic theorems are in terms of pairing functions.

One way to strengthen an arrow a is to define an additional combinator $\mathsf{left}_{\mathsf{a}}$, which can be used to choose an arrow computation based on the result of another. Again, we define a different combinator, $\mathsf{ifte}_{\mathsf{a}}$, to make it easier to apply measure-theoretic theorems.

In a nonstrict λ-calculus, simply defining a choice combinator allows writing recursive functions using nothing but

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

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

$$\begin{aligned}
&\langle \cdot, \cdot \rangle_{\mathsf{map}} : (\mathsf{X} \rightharpoonup \mathsf{Y}_1) \Rightarrow (\mathsf{X} \rightharpoonup \mathsf{Y}_2) \Rightarrow (\mathsf{X} \rightharpoonup \mathsf{Y}_1 \times \mathsf{Y}_2)\\
&\langle \mathsf{g}_1, \mathsf{g}_2 \rangle_{\mathsf{map}}\ :=\ \mathsf{let}\ \ \mathsf{A} := (\mathsf{domain\ g}_1) \cap (\mathsf{domain\ g}_2)\\
&\qquad\qquad\qquad \mathsf{in}\ \ \lambda \mathsf{a} \in \mathsf{A}.\, \langle \mathsf{g}_1\ \mathsf{a}, \mathsf{g}_2\ \mathsf{a} \rangle\\[1ex]
&(\circ_{\mathsf{map}}) : (\mathsf{Y} \rightharpoonup \mathsf{Z}) \Rightarrow (\mathsf{X} \rightharpoonup \mathsf{Y}) \Rightarrow (\mathsf{X} \rightharpoonup \mathsf{Z})\\
&\mathsf{g}_2 \circ_{\mathsf{map}} \mathsf{g}_1\ :=\ \mathsf{let}\ \ \mathsf{A} := \mathsf{preimage\ g}_1\ (\mathsf{domain\ g}_2)\\
&\qquad\qquad\qquad \mathsf{in}\ \ \lambda \mathsf{a} \in \mathsf{A}.\, \mathsf{g}_2\ (\mathsf{g}_1\ \mathsf{a})\\[1ex]
&(\uplus_{\mathsf{map}}) : (\mathsf{X} \rightharpoonup \mathsf{Y}) \Rightarrow (\mathsf{X} \rightharpoonup \mathsf{Y}) \Rightarrow (\mathsf{X} \rightharpoonup \mathsf{Y})\\
&\mathsf{g}_1 \uplus_{\mathsf{map}} \mathsf{g}_2\ :=\ \mathsf{let}\ \ \mathsf{A} := (\mathsf{domain\ g}_1) \uplus (\mathsf{domain\ g}_2)\\
&\qquad\qquad\qquad \mathsf{in}\ \ \lambda \mathsf{a} \in \mathsf{A}.\, \mathsf{if}\ (\mathsf{a} \in \mathsf{domain\ g}_1)\ (\mathsf{g}_1\ \mathsf{a})\ (\mathsf{g}_2\ \mathsf{a})
\end{aligned}$$

Figure 1: Operations on mappings.

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

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

and try to define the following recursive function:

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

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

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

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

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

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

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

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

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

define an ***arrow with choice*** *if certain additional homomorphism and structural laws hold.*

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

The necessary homomorphism laws ensure that $\mathsf{arr}_{\mathsf{a}}$ distributes over function arrow combinators. These laws can be put in terms of more general homomorphism properties that deal with distributing an arrow-to-arrow lift, which we use extensively to prove correctness.

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

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

The homomorphism laws state that $\mathsf{arr}_{\mathsf{a}}$ must be a homomorphism from the function arrow to arrow a.

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

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

3.2 First-Order Let-Calculus Semantics

Figure 2 shows a transformation $[\![\cdot]\!]_{\mathsf{a}}$ 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 accessed by their De Bruijn index, where index 0 refers to the innermost binding.

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

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

Theorem 3.3 (homomorphisms distribute over programs)**.** *Let* $\mathsf{lift}_{\mathsf{b}} : (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{b}} \mathsf{y})$ *be an arrow homomorphism. For all programs* e*,* $[\![e]\!]_{\mathsf{b}} \equiv \mathsf{lift}_{\mathsf{b}}\ [\![e]\!]_{\mathsf{a}}$*.*

Proof. By structural induction on program terms.

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

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

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

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

$$\begin{aligned}
\mathsf{lift_b}\ \llbracket v \rrbracket_{\mathsf{a}} &\equiv \mathsf{lift_b}\ (\mathsf{arr_a}\ (\mathsf{const\ v})) \\
&\equiv \mathsf{arr_b}\ (\mathsf{const\ v}) \\
&\equiv \llbracket v \rrbracket_{\mathsf{b}}
\end{aligned}$$

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

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

It is not hard to check the remaining cases. □

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

4. The Bottom and Mapping Arrows

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

One obvious candidate for the simpler arrow is the function arrow, defined in (10). However, we will need to explicitly handle divergence as an error value, so we need a slightly more complicated arrow for which running computations may raise an error.

Figure 3 defines the ***bottom arrow***. Its computations are of type $\mathsf{x} \rightsquigarrow_\perp \mathsf{y} ::= \mathsf{x} \Rightarrow \mathsf{y}_\perp$, 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 $\mathsf{Bool} \cup \{\perp\}$.

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

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

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

Proof. The bottom arrow is the Maybe monad's Kleisli arrow with $\mathsf{Nothing} = \perp$. □

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 $\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y} ::= \mathsf{X} \rightharpoonup \mathsf{Y}$, with $\mathsf{f_1} \ggg_{\mathsf{map}} \mathsf{f_2} := \mathsf{f_2} \circ_{\mathsf{map}} \mathsf{f_1}$ and $\mathsf{f_1} \mathbin{\&\&\&_{\mathsf{map}}} \mathsf{f_2} := \langle \mathsf{f_1}, \mathsf{f_2} \rangle_{\mathsf{map}}$. Unfortunately, we could not define $\mathsf{arr_{map}} : (\mathsf{X} \Rightarrow \mathsf{Y}) \Rightarrow (\mathsf{X} \rightharpoonup \mathsf{Y})$: to define a mapping, we need a domain, but lambdas' domains are unobservable.

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

$$\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}\ ::=\ \mathsf{Set\ X} \Rightarrow (\mathsf{X} \rightharpoonup \mathsf{Y}) \tag{20}$$

Notice that $\perp$ is absent in $\mathsf{Set\ X} \Rightarrow (\mathsf{X} \rightharpoonup \mathsf{Y})$. (XXX: reword to remove "Notice") This will make it easier to exclude diverging inputs further on. The absence of $\perp$ and the fact that the type parameters denote sets will make it easier to apply theorems from set theory, which know nothing of error values and lambda types.

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

$$\mathsf{g\ A'}\ =\ \mathsf{restrict}\ (\mathsf{g\ A})\ \mathsf{A'} \tag{21}$$

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

To use Theorem 3.3 to prove that programs interpreted using $\llbracket\cdot\rrbracket_{\mathsf{map}}$ behave correctly, we need to define correctness using a lift from the bottom arrow to the mapping arrow. It is helpful to have a standalone function $\mathsf{domain}_\perp$ that computes the subset of A on which f does not return $\perp$. We define that first, and then define $\mathsf{lift_{map}}$ in terms of it:

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

$$\begin{aligned}
&\mathsf{lift_{map}} : (\mathsf{X} \rightsquigarrow_\perp \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}) \\
&\mathsf{lift_{map}\ f\ A}\ :=\ \mathsf{mapping\ f}\ (\mathsf{domain}_\perp\ \mathsf{f\ A})
\end{aligned} \tag{23}$$

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* $\mathsf{g_1} : \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}$ *and* $\mathsf{g_2} : \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}$ *are equivalent, or* $\mathsf{g_1} \equiv \mathsf{g_2}$*, when* $\mathsf{g_1\ A} \equiv \mathsf{g_2\ A}$ *for all* $\mathsf{A} \subseteq \mathsf{X}$*.*

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

$$\mathsf{x} \rightsquigarrow_\bot \mathsf{y} \;::=\; \mathsf{x} \Rightarrow \mathsf{y}_\bot$$

$$\begin{aligned}
&\mathsf{arr}_\bot : (\mathsf{x} \Rightarrow \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_\bot \mathsf{y})\\
&\mathsf{arr}_\bot\ \mathsf{f} \;:=\; \mathsf{f}
\end{aligned}$$

$$\begin{aligned}
&(\ggg_\bot) : (\mathsf{x} \rightsquigarrow_\bot \mathsf{y}) \Rightarrow (\mathsf{y} \rightsquigarrow_\bot \mathsf{z}) \Rightarrow (\mathsf{x} \rightsquigarrow_\bot \mathsf{z})\\
&(\mathsf{f}_1 \ggg_\bot \mathsf{f}_2)\ \mathsf{a} \;:=\; \mathsf{if}\ (\mathsf{f}_1\ \mathsf{a} = \bot)\ \bot\ (\mathsf{f}_2\ (\mathsf{f}_1\ \mathsf{a}))
\end{aligned}$$

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

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

$$\begin{aligned}
&\mathsf{lazy}_\bot : (1 \Rightarrow (\mathsf{x} \rightsquigarrow_\bot \mathsf{y})) \Rightarrow (\mathsf{x} \rightsquigarrow_\bot \mathsf{y})\\
&\mathsf{lazy}_\bot\ \mathsf{f}\ \mathsf{a} \;:=\; \mathsf{f}\ 0\ \mathsf{a}
\end{aligned}$$

Figure 3: Bottom arrow definitions.

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

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

$$\begin{aligned}
&(\ggg_{\mathsf{map}}) : (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}) \Rightarrow (\mathsf{Y} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Z}) \Rightarrow (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Z})\\
&(\mathsf{g}_1 \ggg_{\mathsf{map}} \mathsf{g}_2)\ \mathsf{A} \;:=\; \mathsf{let}\ \ \mathsf{g}_1' := \mathsf{g}_1\ \mathsf{A}\\
&\qquad\qquad\qquad\qquad\quad \mathsf{g}_2' := \mathsf{g}_2\ (\mathsf{range}\ \mathsf{g}_1')\\
&\qquad\qquad\qquad\qquad \mathsf{in}\ \ \mathsf{g}_2' \circ_{\mathsf{map}} \mathsf{g}_1'
\end{aligned}$$

$$\begin{aligned}
&(\mathbin{\&\&\&}_{\mathsf{map}}) : (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}_1) \Rightarrow (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}_2) \Rightarrow (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \langle \mathsf{Y}_1, \mathsf{Y}_2 \rangle)\\
&(\mathsf{g}_1 \mathbin{\&\&\&}_{\mathsf{map}} \mathsf{g}_2)\ \mathsf{A} \;:=\; \langle \mathsf{g}_1\ \mathsf{A}, \mathsf{g}_2\ \mathsf{A} \rangle_{\mathsf{map}}
\end{aligned}$$

$$\begin{aligned}
&\mathsf{ifte}_{\mathsf{map}} : (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Bool}) \Rightarrow (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y})\\
&\mathsf{ifte}_{\mathsf{map}}\ \mathsf{g}_1\ \mathsf{g}_2\ \mathsf{g}_3\ \mathsf{A} \;:=\; \mathsf{let}\ \ \mathsf{g}_1' := \mathsf{g}_1\ \mathsf{A}\\
&\qquad\qquad\qquad\qquad\quad \mathsf{g}_2' := \mathsf{g}_2\ (\mathsf{preimage}\ \mathsf{g}_1'\ \{\mathsf{true}\})\\
&\qquad\qquad\qquad\qquad\quad \mathsf{g}_3' := \mathsf{g}_3\ (\mathsf{preimage}\ \mathsf{g}_1'\ \{\mathsf{false}\})\\
&\qquad\qquad\qquad\qquad \mathsf{in}\ \ \mathsf{g}_2' \uplus_{\mathsf{map}} \mathsf{g}_3'
\end{aligned}$$

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

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

Figure 4: Mapping arrow definitions.

4.1.1 Case: Pairing

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

$$\begin{aligned}
&\mathsf{lift}_{\mathsf{map}}\ (\mathsf{f}_1 \mathbin{\&\&\&}_\bot \mathsf{f}_2)\ \mathsf{A}\\
&\quad \equiv\ \mathsf{lift}_{\mathsf{map}}\ (\lambda \mathsf{a}.\, \mathsf{if}\ (\mathsf{f}_1\ \mathsf{a} = \bot\ \mathsf{or}\ \mathsf{f}_2\ \mathsf{a} = \bot)\ \bot\ \langle \mathsf{f}_1\ \mathsf{a}, \mathsf{f}_2\ \mathsf{a} \rangle)\ \mathsf{A}\\
&\quad \equiv\ \mathsf{let}\ \ \mathsf{f} := \lambda \mathsf{a}.\, \mathsf{if}\ (\mathsf{f}_1\ \mathsf{a} = \bot\ \mathsf{or}\ \mathsf{f}_2\ \mathsf{a} = \bot)\ \bot\ \langle \mathsf{f}_1\ \mathsf{a}, \mathsf{f}_2\ \mathsf{a} \rangle\\
&\qquad\quad \mathsf{in}\ \ \mathsf{mapping}\ \mathsf{f}\ (\mathsf{domain}_\bot\ \mathsf{f}\ \mathsf{A})
\end{aligned} \tag{24}$$

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

$$\begin{aligned}
&\mathsf{lift}_{\mathsf{map}}\ (\mathsf{f}_1 \mathbin{\&\&\&}_\bot \mathsf{f}_2)\ \mathsf{A}\\
&\quad \equiv\ \mathsf{let}\ \ \mathsf{A}_1 := (\mathsf{domain}_\bot\ \mathsf{f}_1\ \mathsf{A})\\
&\qquad\qquad\ \mathsf{A}_2 := (\mathsf{domain}_\bot\ \mathsf{f}_2\ \mathsf{A})\\
&\qquad\qquad\ \mathsf{A}' := \mathsf{A}_1 \cap \mathsf{A}_2\\
&\qquad\quad \mathsf{in}\ \ \lambda \mathsf{a} \in \mathsf{A}'.\, \langle \mathsf{f}_1\ \mathsf{a}, \mathsf{f}_2\ \mathsf{a} \rangle\\
&\quad \equiv\ \mathsf{let}\ \ \mathsf{g}_1 := \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_1\ \mathsf{A}\\
&\qquad\qquad\ \mathsf{g}_2 := \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_2\ \mathsf{A}\\
&\qquad\qquad\ \mathsf{A}' := (\mathsf{domain}\ \mathsf{g}_1) \cap (\mathsf{domain}\ \mathsf{g}_2)\\
&\qquad\quad \mathsf{in}\ \ \lambda \mathsf{a} \in \mathsf{A}'.\, \langle \mathsf{g}_1\ \mathsf{a}, \mathsf{g}_2\ \mathsf{a} \rangle\\
&\quad \equiv\ \langle \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_1\ \mathsf{A}, \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_2\ \mathsf{A} \rangle_{\mathsf{map}}
\end{aligned} \tag{25}$$

Substituting g_1 for $\mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_1$ and g_2 for $\mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_2$ gives a definition for $(\mathbin{\&\&\&}_{\mathsf{map}})$ (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 $\mathsf{f}_1\ \mathsf{a} \neq \bot$:

$$\begin{aligned}
&\mathsf{lift}_{\mathsf{map}}\ (\mathsf{ifte}_\bot\ \mathsf{f}_1\ \mathsf{f}_2\ \mathsf{f}_3)\ \mathsf{A}\\
&\quad \equiv\ \mathsf{let}\ \ \mathsf{f} := \lambda \mathsf{a}.\, \mathsf{case}\ \ \mathsf{f}_1\ \mathsf{a}\\
&\qquad\qquad\qquad\qquad \mathsf{true} \longrightarrow \mathsf{f}_2\ \mathsf{a}\\
&\qquad\qquad\qquad\qquad \mathsf{false} \longrightarrow \mathsf{f}_3\ \mathsf{a}\\
&\qquad\qquad\qquad\qquad \mathsf{else} \longrightarrow \bot\\
&\qquad\quad \mathsf{in}\ \ \mathsf{mapping}\ \mathsf{f}\ (\mathsf{domain}_\bot\ \mathsf{f}\ \mathsf{A})\\
&\quad \equiv\ \mathsf{let}\ \ \mathsf{g}_1 := \mathsf{mapping}\ \mathsf{f}\ \mathsf{A}\\
&\qquad\qquad\ \mathsf{A}_2 := \mathsf{preimage}\ \mathsf{g}_1\ \{\mathsf{true}\}\\
&\qquad\qquad\ \mathsf{A}_3 := \mathsf{preimage}\ \mathsf{g}_1\ \{\mathsf{false}\}\\
&\qquad\qquad\quad \mathsf{f} := \lambda \mathsf{a}.\, \mathsf{if}\ (\mathsf{f}_1\ \mathsf{a})\ (\mathsf{f}_2\ \mathsf{a})\ (\mathsf{f}_3\ \mathsf{a})\\
&\qquad\quad \mathsf{in}\ \ \mathsf{mapping}\ \mathsf{f}\ (\mathsf{domain}_\bot\ \mathsf{f}\ (\mathsf{A}_2 \uplus \mathsf{A}_3))
\end{aligned} \tag{26}$$

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

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

Substituting $\mathsf{g_1}$ for $\mathsf{lift_{map}}\ \mathsf{f_1}$, $\mathsf{g_2}$ for $\mathsf{lift_{map}}\ \mathsf{f_2}$, and $\mathsf{g_3}$ for $\mathsf{lift_{map}}\ \mathsf{f_3}$ 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:

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

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

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

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

4.2 Correctness

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

Proof. By construction. □

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

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

$$\mathsf{X} \underset{\mathsf{pre}}{\rightleftharpoons} \mathsf{Y}\ ::=\ \langle \mathsf{Set\ Y}, \mathsf{Set\ Y} \Rightarrow \mathsf{Set\ X} \rangle \tag{28}$$

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

$$\begin{aligned}
&\mathsf{pre} : (\mathsf{X} \rightharpoonup \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightleftharpoons} \mathsf{Y}) \\
&\mathsf{pre\ g}\ \ :=\ \ \langle \mathsf{range\ g}, \lambda \mathsf{B}.\, \mathsf{preimage\ g\ B} \rangle
\end{aligned} \tag{29}$$

Applying a preimage mapping to any subset of its codomain:

$$\begin{aligned}
&\mathsf{ap_{pre}} : (\mathsf{X} \underset{\mathsf{pre}}{\rightleftharpoons} \mathsf{Y}) \Rightarrow \mathsf{Set\ Y} \Rightarrow \mathsf{Set\ X} \\
&\mathsf{ap_{pre}}\ \langle \mathsf{Y'}, \mathsf{p} \rangle\ \mathsf{B}\ :=\ \mathsf{p}\ (\mathsf{B} \cap \mathsf{Y'})
\end{aligned} \tag{30}$$

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

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

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

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

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

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

Definition 5.3 (preimage mapping equivalence)**.** *Two preimage mappings $\mathsf{h_1} : \mathsf{X} \underset{\mathsf{pre}}{\rightleftharpoons} \mathsf{Y}$ and $\mathsf{h_2} : \mathsf{X} \underset{\mathsf{pre}}{\rightleftharpoons} \mathsf{Y}$ are equivalent, or $\mathsf{h_1} \equiv \mathsf{h_2}$, when $\mathsf{ap_{pre}}\ \mathsf{h_1}\ \mathsf{B} \equiv \mathsf{ap_{pre}}\ \mathsf{h_2}\ \mathsf{B}$ for all $\mathsf{B} \subseteq \mathsf{Y}$.*

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

5.0.1 Preimage Mapping Pairing

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 $\mathsf{B_2} \subseteq \mathsf{Y_2}$, $\mathsf{preimage}\ \langle \mathsf{g_1}, \mathsf{g_2} \rangle_{\mathsf{map}}\ (\mathsf{B_1} \times \mathsf{B_2}) = (\mathsf{preimage\ g_1\ B_1}) \cap (\mathsf{preimage\ g_2\ B_2})$.*

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

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

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

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

$$\begin{aligned}
&\mathsf{range}_{\mathsf{pre}} : (X \underset{\mathsf{pre}}{\rightharpoonup} Y) \Rightarrow \mathsf{Set}\ Y \\
&\mathsf{range}_{\mathsf{pre}} \ :=\ \mathsf{fst}
\end{aligned}$$

$$\begin{aligned}
&\langle \cdot, \cdot \rangle_{\mathsf{pre}} : (X \underset{\mathsf{pre}}{\rightharpoonup} Y_1) \Rightarrow (X \underset{\mathsf{pre}}{\rightharpoonup} Y_2) \Rightarrow (X \underset{\mathsf{pre}}{\rightharpoonup} Y_1 \times Y_2) \\
&\langle \langle Y'_1, p_1 \rangle, \langle Y'_2, p_2 \rangle \rangle_{\mathsf{pre}} \ :=\ \mathsf{let}\ \ Y' := Y'_1 \times Y'_2 \\
&\qquad\qquad\qquad\qquad\qquad\quad p := \lambda B. \bigcup_{\langle b_1, b_2 \rangle \in B} (p_1\ \{b_1\}) \cap (p_2\ \{b_2\}) \\
&\qquad\qquad\qquad\qquad\quad \mathsf{in}\ \ \langle Y', p \rangle
\end{aligned}$$

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

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

Figure 5: Lazy preimage mappings and operations.

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

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

□

5.0.2 Preimage Mapping Composition

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

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

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

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

□

5.0.3 Preimage Mapping Disjoint Union

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

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

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

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

□

6. Deriving the Preimage Arrow

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

As with the mapping arrow and mappings, we cannot have $X \underset{\mathsf{pre}}{\rightsquigarrow} Y ::= X \underset{\mathsf{pre}}{\rightharpoonup} Y$: we run into trouble trying to define $\mathsf{arr}_{\mathsf{pre}}$ because a preimage mapping needs an observable domain. While a preimage mapping's domain is the *range* of the mapping it computes preimages for, it is still easiest to parameterize preimage computations on a $\mathsf{Set}\ X$:

$$X \underset{\mathsf{pre}}{\rightsquigarrow} Y \ ::=\ \mathsf{Set}\ X \Rightarrow (X \underset{\mathsf{pre}}{\rightharpoonup} Y) \tag{31}$$

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

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

$$\begin{aligned} &\mathsf{lift_{pre}} : (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}) \\ &\mathsf{lift_{pre}}\ \mathsf{g}\ \mathsf{A} \ := \ \mathsf{pre}\ (\mathsf{g}\ \mathsf{A}) \end{aligned} \tag{32}$$

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

$$\mathsf{ap_{pre}}\ (\mathsf{lift_{pre}}\ \mathsf{g}\ \mathsf{A})\ \mathsf{B} \equiv \mathsf{preimage}\ (\mathsf{g}\ \mathsf{A})\ \mathsf{B} \tag{33}$$

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

We also need a coarser notion of equivalence.

Definition 6.1 (Preimage arrow equivalence)**.** *Two preimage arrow computations* $\mathsf{h}_1 : \mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}$ *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 $(\&\&\&_{\mathsf{pre}})$, $(\ggg_{\mathsf{pre}})$, $\mathsf{ifte_{pre}}$ and $\mathsf{lazy_{pre}}$ from their corresponding 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.

6.1 Case: Pairing

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

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

Substituting h_1 for $\mathsf{lift_{pre}}\ \mathsf{g}_1$ and 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 $(\&\&\&_{\mathsf{pre}})$ (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 $\mathsf{lift_{pre}}$:

$$\begin{aligned} &\mathsf{ap_{pre}}\ (\mathsf{lift_{pre}}\ (\mathsf{g}_1 \ggg_{\mathsf{map}} \mathsf{g}_2)\ \mathsf{A})\ \mathsf{C} \\ &\quad \equiv\ \mathsf{let}\ \ \mathsf{g}_1' := \mathsf{g}_1\ \mathsf{A} \\ &\qquad\qquad\ \ \mathsf{g}_2' := \mathsf{g}_2\ (\mathsf{range}\ \mathsf{g}_1') \\ &\qquad\quad\ \ \mathsf{in}\ \ \mathsf{ap_{pre}}\ (\mathsf{pre}\ (\mathsf{g}_2' \circ_{\mathsf{map}} \mathsf{g}_1'))\ \mathsf{C} \\ &\quad \equiv\ \mathsf{let}\ \ \mathsf{g}_1' := \mathsf{g}_1\ \mathsf{A} \\ &\qquad\qquad\ \ \mathsf{g}_2' := \mathsf{g}_2\ (\mathsf{range}\ \mathsf{g}_1') \\ &\qquad\quad\ \ \mathsf{in}\ \ \mathsf{ap_{pre}}\ ((\mathsf{pre}\ \mathsf{g}_1') \circ_{\mathsf{pre}} (\mathsf{pre}\ \mathsf{g}_2'))\ \mathsf{C} \\ &\quad \equiv\ \mathsf{let}\ \ \mathsf{h}_1 := \mathsf{lift_{pre}}\ \mathsf{g}_1\ \mathsf{A} \\ &\qquad\qquad\ \ \mathsf{h}_2 := \mathsf{lift_{pre}}\ \mathsf{g}_2\ (\mathsf{range_{pre}}\ \mathsf{h}_1) \\ &\qquad\quad\ \ \mathsf{in}\ \ \mathsf{ap_{pre}}\ (\mathsf{h}_2 \circ_{\mathsf{pre}} \mathsf{h}_1)\ \mathsf{C} \end{aligned} \tag{34}$$

Substituting h_1 for $\mathsf{lift_{pre}}\ \mathsf{g}_1$ and 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 h_2 and 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 h_1 for $\mathsf{lift_{pre}}\ \mathsf{g}_1$, h_2 for $\mathsf{lift_{pre}}\ \mathsf{g}_2$ and 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 programs e,* $[\![e]\!]_{\mathsf{pre}} \equiv \mathsf{lift_{pre}}\ [\![e]\!]_{\mathsf{map}}$.

7. Preimages Under Partial Functions

XXX: diagram; YOU ARE HERE discussion

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

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

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

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

Suppose we interpret (35) as $\mathsf{h} : \mathsf{R} \underset{\mathsf{pre}}{\rightsquigarrow} \mathbb{N}$, a preimage arrow computation from random sources in R to natural numbers,

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

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

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

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

$$\begin{aligned}
&\mathsf{ifte}_{\mathsf{pre}} : (X \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Bool}) \Rightarrow (X \underset{\mathsf{pre}}{\rightsquigarrow} Y) \Rightarrow (X \underset{\mathsf{pre}}{\rightsquigarrow} Y) \Rightarrow (X \underset{\mathsf{pre}}{\rightsquigarrow} Y)\\
&\mathsf{ifte}_{\mathsf{pre}}\ h_1\ h_2\ h_3\ A \ := \ \mathsf{let}\ \ h_1' := h_1\ A\\
&\qquad\qquad\qquad\qquad\qquad\quad h_2' := h_2\ (\mathsf{ap}_{\mathsf{pre}}\ h_1'\ \{\mathsf{true}\})\\
&\qquad\qquad\qquad\qquad\qquad\quad h_3' := h_3\ (\mathsf{ap}_{\mathsf{pre}}\ h_1'\ \{\mathsf{false}\})\\
&\qquad\qquad\qquad\qquad\quad\ \ \mathsf{in}\ \ h_2' \uplus_{\mathsf{pre}} h_3'
\end{aligned}$$

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

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

Figure 6: Preimage arrow definitions.

and that we have a probability measure $\mathsf{P} \in \mathcal{P}\ \mathsf{R} \rightharpoonup [0,1]$. We could compute the probability of any output set $\mathsf{N} \subseteq \mathbb{N}$ using $\mathsf{P}\ (\mathsf{h}\ \mathsf{R}'\ \mathsf{N})$, where $\mathsf{R}' \subseteq \mathsf{R}$ and $\mathsf{P}\ \mathsf{R}' = 1$. We have three hurdles to overcome:

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

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

7.1 Threading and Indexing

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

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

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

We need a way to assign unique indexes to expressions.

Definition 7.1 (binary indexing scheme)**.** *Let* J *be an index set,* $\mathsf{j}_0 \in \mathsf{J}$ *a distinguished element, and* $\mathsf{left} : \mathsf{J} \Rightarrow \mathsf{J}$ *and* $\mathsf{right} : \mathsf{J} \Rightarrow \mathsf{J}$ *be total, injective functions. If for all* $\mathsf{j} \in \mathsf{J}$, $\mathsf{j} = \mathsf{next}\ \mathsf{j}_0$ *for some finite composition* next*, then* J*,* j_0*,* left *and* right *define a* ***binary indexing scheme****.*

For example, let J be the set of lists of $\{0,1\}$, $\mathsf{j}_0 := \langle\rangle$, and $\mathsf{left}\ \mathsf{j} := \langle 0, \mathsf{j}\rangle$ and $\mathsf{right}\ \mathsf{j} := \langle 1, \mathsf{j}\rangle$.

Alternatively, let J be the set of dyadic rationals in $(0,1)$ (i.e. those with power-of-two denominators), $\mathsf{j}_0 := \frac{1}{2}$ and

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

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

In any case, J is always countable, and can be thought of as a set of indexes into an infinite binary tree.

7.2 Applicative, Associative Store

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

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

$$\mathsf{AStore}\ \mathsf{s}\ (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \ ::= \ \mathsf{J} \Rightarrow (\langle \mathsf{s}, \mathsf{x}\rangle \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \tag{37}$$

XXX: motivational wurds for definition of lift:

$$\begin{aligned}
&\eta_{\mathsf{a}^*} : (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \Rightarrow \mathsf{AStore}\ \mathsf{s}\ (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y})\\
&\eta_{\mathsf{a}^*}\ \mathsf{f}\ \mathsf{j} \ := \ \mathsf{arr}_{\mathsf{a}}\ \mathsf{snd} \ggg_{\mathsf{a}} \mathsf{f}
\end{aligned} \tag{38}$$

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

Definition 7.2 (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* $\mathsf{j} \in \mathsf{J}$*.*

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

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

Proof. Defining $\mathsf{arr}_{\mathsf{a}^*}$ as a composition clearly meets the first homomorphism identity (15).

Composition. Starting with the right side of (16), expand definitions and use $(\mathsf{arr}_{\mathsf{a}}\ \mathsf{f} \mathbin{\&\&\&}_{\mathsf{a}} \mathsf{f}_1) \ggg_{\mathsf{a}} \mathsf{arr}_{\mathsf{a}}\ \mathsf{snd} \equiv \mathsf{f}_1$:

$$\begin{aligned}
&(\eta_{\mathsf{a}^*}\ \mathsf{f}_1 \ggg_{\mathsf{a}^*} \eta_{\mathsf{a}^*}\ \mathsf{f}_2)\ \mathsf{j}\\
&\quad\equiv\ (\mathsf{arr}_{\mathsf{a}}\ \mathsf{fst} \mathbin{\&\&\&}_{\mathsf{a}} (\mathsf{arr}_{\mathsf{a}}\ \mathsf{snd} \ggg_{\mathsf{a}} \mathsf{f}_1)) \ggg_{\mathsf{a}} \mathsf{arr}_{\mathsf{a}}\ \mathsf{snd} \ggg_{\mathsf{a}} \mathsf{f}_2\\
&\quad\equiv\ \mathsf{arr}_{\mathsf{a}}\ \mathsf{snd} \ggg_{\mathsf{a}} \mathsf{f}_1 \ggg_{\mathsf{a}} \mathsf{f}_2\\
&\quad\equiv\ \eta_{\mathsf{a}^*}\ (\mathsf{f}_1 \ggg_{\mathsf{a}} \mathsf{f}_2)\ \mathsf{j}
\end{aligned}$$

Pairing. Starting with the right side of (17), expand definitions and use the arrow law $\mathsf{arr}_{\mathsf{a}}\ \mathsf{f} \ggg_{\mathsf{a}} (\mathsf{f}_1 \mathbin{\&\&\&}_{\mathsf{a}} \mathsf{f}_2) \equiv$

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

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

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

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

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

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

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

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

$(\mathsf{arr}_a\ f \ggg_a f_1) \mathbin{\&\&\&_a} (\mathsf{arr}_a\ f \ggg_a f_2)$:

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

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

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

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

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

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

Corollary 7.4 (semantic correctness)**.** *Let* $x \rightsquigarrow_{a^*} y ::= \mathsf{AStore}\ s\ (x \rightsquigarrow_a y)$. *If* $[\![e]\!]_a : x \rightsquigarrow_a y$, *then* $\eta_{a^*}\ [\![e]\!]_a \equiv [\![e]\!]_{a^*}$ *and* $[\![e]\!]_{a^*} : x \rightsquigarrow_{a^*} y$.

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

7.3 Probabilistic Programs

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

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

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

We extend the let-calculus semantic function with

$$[\![\mathsf{random}]\!]_{a^*} :\equiv \mathsf{random}_{a^*} \tag{40}$$

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

7.4 Partial Programs

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

XXX: update above; just say why it's GOOD instead of why it's BETTER (unless describing other approaches)

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

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

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

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

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

$$\begin{aligned}
&\mathsf{ifte}^{\Downarrow}_{a^*} : (x \rightsquigarrow_{a^*} \mathsf{Bool}) \Rightarrow (x \rightsquigarrow_{a^*} y) \Rightarrow (x \rightsquigarrow_{a^*} y) \Rightarrow (x \rightsquigarrow_{a^*} y)\\
&\mathsf{ifte}^{\Downarrow}_{a^*}\ k_1\ k_2\ k_3\ j \;:=\\
&\quad \mathsf{ifte}^{\Downarrow}_a\ ((k_1\ (\mathsf{left}\ j) \mathbin{\&\&\&_a} \mathsf{branch}_{a^*}\ j) \ggg_a \mathsf{arr}_a\ \mathsf{agrees})\\
&\qquad\quad (k_2\ (\mathsf{left}\ (\mathsf{right}\ j)))\\
&\qquad\quad (k_3\ (\mathsf{right}\ (\mathsf{right}\ j)))
\end{aligned} \tag{43}$$

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

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

$$\begin{aligned}
\llbracket \mathsf{if}\ e_c\ e_t\ e_f \rrbracket^{\Downarrow}_{\mathsf{a*}} \ &:\equiv\ \mathsf{ifte}^{\Downarrow}_{\mathsf{a*}}\ \llbracket e_c \rrbracket_{\mathsf{a*}} \\
&\qquad\qquad (\mathsf{lazy_a}\ \lambda 0.\, \llbracket e_t \rrbracket_{\mathsf{a*}}) \\
&\qquad\qquad (\mathsf{lazy_a}\ \lambda 0.\, \llbracket e_f \rrbracket_{\mathsf{a*}}) \\
\llbracket e \rrbracket^{\Downarrow}_{\mathsf{a*}} \ &:\equiv\ \llbracket e \rrbracket_{\mathsf{a*}}
\end{aligned} \tag{44}$$

7.5 Partial, Probabilistic Programs

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

$$\begin{aligned}
&\mathsf{random}_{\mathsf{a*}} : \mathsf{x} \rightsquigarrow_{\mathsf{a*}} [0,1] \\
&\mathsf{random}_{\mathsf{a*}}\ \mathsf{j}\ :=\ \mathsf{arr_a}\ (\mathsf{fst} \ggg \mathsf{fst} \ggg \pi\ \mathsf{j})
\end{aligned} \tag{45}$$

$$\begin{aligned}
&\mathsf{branch}_{\mathsf{a*}} : \mathsf{x} \rightsquigarrow_{\mathsf{a*}} \mathsf{Bool} \\
&\mathsf{branch}_{\mathsf{a*}}\ \mathsf{j}\ :=\ \mathsf{arr_a}\ (\mathsf{fst} \ggg \mathsf{snd} \ggg \pi\ \mathsf{j})
\end{aligned} \tag{46}$$

The $\mathsf{ifte}^{\Downarrow}_{\mathsf{a*}}$ combinator's definition remains the same.

7.6 Correctness

Theorem 7.7 (natural transformation)**.** *Let* $\mathsf{x} \rightsquigarrow_{\mathsf{a*}} \mathsf{y} ::= \mathsf{AStore\ s\ (x} \rightsquigarrow_{\mathsf{a}} \mathsf{y)}$ *and* $\mathsf{x} \rightsquigarrow_{\mathsf{b*}} \mathsf{y} ::= \mathsf{AStore\ s\ (x} \rightsquigarrow_{\mathsf{b*}} \mathsf{y)}$*. Let* $\mathsf{lift_b} : (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{b}} \mathsf{y})$ *be an arrow homomorphism, and*

$$\begin{aligned}
&\mathsf{lift}_{\mathsf{b*}} : (\mathsf{x} \rightsquigarrow_{\mathsf{a*}} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{b*}} \mathsf{y}) \\
&\mathsf{lift}_{\mathsf{b*}}\ \mathsf{f}\ \mathsf{j}\ :=\ \mathsf{lift_b}\ (\mathsf{f}\ \mathsf{j})
\end{aligned} \tag{47}$$

The following diagram commutes:

$$\begin{tikzcd}
\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y} \arrow[r, "\mathsf{lift_b}"] \arrow[d, "\eta_{\mathsf{a*}}"'] & \mathsf{x} \rightsquigarrow_{\mathsf{b}} \mathsf{y} \arrow[d, "\eta_{\mathsf{b*}}"] \\
\mathsf{x} \rightsquigarrow_{\mathsf{a*}} \mathsf{y} \arrow[r, "\mathsf{lift}_{\mathsf{b*}}"'] & \mathsf{x} \rightsquigarrow_{\mathsf{b*}} \mathsf{y}
\end{tikzcd} \tag{48}$$

i.e. for all $\mathsf{f} : \mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}$, $\eta_{\mathsf{b*}}\ (\mathsf{lift_b\ f}) \equiv \mathsf{lift}_{\mathsf{b*}}\ (\eta_{\mathsf{a*}}\ \mathsf{f})$*. Further,* $\mathsf{lift}_{\mathsf{b*}}$ *is an arrow homomorphism.*

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

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

Further, because $\eta_{\mathsf{a*}}$, $\eta_{\mathsf{b*}}$, and $\mathsf{lift_b}$ are homomorphisms, $\mathsf{lift}_{\mathsf{b*}}$ is a homomorphism by composition.

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

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

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

$$\begin{tikzcd}
\mathsf{X} \rightsquigarrow_{\perp} \mathsf{Y} \arrow[r, "\mathsf{lift_{map}}"] \arrow[d, "\eta_{\perp*}"'] & \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y} \arrow[r, "\mathsf{lift_{pre}}"] \arrow[d, "\eta_{\mathsf{map*}}"] & \mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y} \arrow[d, "\eta_{\mathsf{pre*}}"] \\
\mathsf{X} \rightsquigarrow_{\perp*} \mathsf{Y} \arrow[r, "\mathsf{lift_{map*}}"'] & \mathsf{X} \underset{\mathsf{map*}}{\rightsquigarrow} \mathsf{Y} \arrow[r, "\mathsf{lift_{pre*}}"'] & \mathsf{X} \underset{\mathsf{pre*}}{\rightsquigarrow} \mathsf{Y}
\end{tikzcd} \tag{49}$$

Further, $\mathsf{lift_{map*}}$ *and* $\mathsf{lift_{pre*}}$ *are arrow homomorphisms.*

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

Corollary 7.10 (semantic$^{\Downarrow}$ correctness)**.** *If* $\llbracket e \rrbracket^{\Downarrow}_{\perp*} : \mathsf{X} \rightsquigarrow_{\perp*} \mathsf{Y}$*, then* $\mathsf{lift_{map*}}\ \llbracket e \rrbracket^{\Downarrow}_{\perp*} \equiv \llbracket e \rrbracket^{\Downarrow}_{\mathsf{map*}}$ *and* $\mathsf{lift_{pre*}}\ \llbracket e \rrbracket^{\Downarrow}_{\mathsf{map*}} \equiv \llbracket e \rrbracket^{\Downarrow}_{\mathsf{pre*}}$*.*

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

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

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

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

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

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

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

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

Recall truth statements like $\mathsf{f}\ \mathsf{j_0}\ \mathsf{x} \neq \perp$ *imply convergence.*

That $\mathsf{lift_{map*}}$ and $\mathsf{lift_{pre*}}$ are arrow homomorphisms allows transporting halting set definitions and theorems between arrow types.

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

Proof. XXX: do this □

Corollary 7.14 (computed halting set)**.** *Let* $\llbracket e \rrbracket_{\perp*} : \mathsf{X} \rightsquigarrow_{\perp*} \mathsf{Y}$ *converge. Then* $\mathsf{A}^* = \mathsf{domain}\ (\llbracket e \rrbracket^{\Downarrow}_{\mathsf{map*}}\ \mathsf{j_0}\ ((\mathsf{R} \times \mathsf{T}) \times \mathsf{X}))$.

Corollary 7.15 (semantic correctness (final))**.** *Let* $\llbracket e \rrbracket_{\perp*} : \mathsf{X} \rightsquigarrow_{\perp*} \mathsf{Y}$ *converge and have halting set* A^**. For all* $\mathsf{A} \subseteq (\mathsf{R} \times \mathsf{T}) \times \mathsf{X}$ *and* $\mathsf{B} \subseteq \mathsf{Y}$, $\mathsf{ap_{pre}}\ (\llbracket e \rrbracket^{\Downarrow}_{\mathsf{pre*}}\ \mathsf{j_0}\ \mathsf{A})\ \mathsf{B} = \mathsf{preimage}\ (\llbracket e \rrbracket_{\mathsf{map*}}\ \mathsf{j_0}\ (\mathsf{A} \cap \mathsf{A}^*))\ \mathsf{B}$.

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

8. Measurability

We have not assigned probabilities to any sets yet.

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

$$\mathsf{P}\ (\mathsf{preimage\ g\ B}) \tag{50}$$

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

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

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

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

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

The analogue of continuity is measurability.

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

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

Product spaces are defined the same way as in topology.

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

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

XXX: remind readers that lemmas are imported theorems

8.1 Measurable Pure Computations

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

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

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

Definition 8.6 (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 $\mathsf{g\ A}^*$ is an $\mathcal{A}$-$\mathcal{B}$-measurable mapping, where A^* is g's halting set.*

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

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. Then for all $\mathsf{A} \in \mathcal{A}$, $\mathsf{g\ A}$ is an $\mathcal{A}$-$\mathcal{B}$-measurable mapping.*

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

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

That all programs interpreted by $[\![\cdot]\!]_{\mathsf{a}}$ are measurable will be proved by structural induction on terms. (XXX: not really—we only do that for probabilistic programs) We therefore need a case for each arrow combinator.

8.1.1 Case: Composition

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

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

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 $\mathsf{A} \subseteq \mathsf{X}$, $\mathsf{domain}\ (\mathsf{g\ A}) \subseteq \mathsf{A}$. For any $\mathsf{A}' \subseteq \mathsf{A}$, $\mathsf{g\ A}' \subseteq \mathsf{g\ A}$.*

Proof. Use the mapping arrow restriction law (21). □

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

Proof. Use the mapping arrow restriction law (21). □

Now we can prove measurability.

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 halting sets. The halting set of $\mathsf{g}_1 \ggg_{\mathsf{map}} \mathsf{g}_2$ is $\mathsf{A}^{**} := \mathsf{preimage}\ (\mathsf{g}_1\ \mathsf{A}^*)\ \mathsf{B}^*$, which is in $\mathcal{A}$. By definition,

$$\begin{aligned} (\mathsf{g}_1 \ggg_{\mathsf{map}} \mathsf{g}_2)\ \mathsf{A}^{**} \ = \ \ \mathsf{let}\ \ & \mathsf{g}_1' := \mathsf{g}_1\ \mathsf{A}^{**} \\ & \mathsf{g}_2' := \mathsf{g}_2\ (\mathsf{range\ g}_1') \\ \mathsf{in}\ \ & \mathsf{g}_2' \circ_{\mathsf{map}} \mathsf{g}_1' \end{aligned} \tag{51}$$

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

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

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

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

8.1.2 Case: Pairing

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

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

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

8.1.3 Case: Conditional

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

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

Proof. Let $\mathcal{A}_1^*$, $\mathcal{A}_2^*$ and $\mathcal{A}_3^*$ be respectively g_1's, g_2's and g_3's halting sets. The halting set of $\mathsf{ifte}_{\mathsf{map}}\ g_1\ g_2\ g_3$ is defined by

$$\begin{aligned} A_2^{**} &:= A_2^* \cap \mathsf{preimage}\ (g_1\ \mathcal{A}_1^*)\ \{\mathsf{true}\} \\ A_3^{**} &:= A_3^* \cap \mathsf{preimage}\ (g_1\ \mathcal{A}_1^*)\ \{\mathsf{false}\} \\ A^{**} &:= A_2^{**} \uplus A_3^{**} \end{aligned} \tag{52}$$

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

$$\begin{aligned} \mathsf{ifte}_{\mathsf{map}}\ g_1\ g_2\ g_3\ A^{**} \ = \ \mathsf{let}\ \ & g_1' := g_1\ A^{**} \\ & g_2' := g_2\ (\mathsf{preimage}\ g_1'\ \{\mathsf{true}\}) \\ & g_3' := g_3\ (\mathsf{preimage}\ g_1'\ \{\mathsf{false}\}) \\ \mathsf{in}\ \ & g_2' \uplus_{\mathsf{map}} g_3' \end{aligned} \tag{53}$$

By hypothesis, g_1', g_2' and g_3' are measurable mappings, and the mapping arrow restriction law (21) imples g_2' and g_3' have disjoint domains. Apply Lemma 8.17. □

8.1.4 Case: Laziness

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

XXX: possibly make the lemma a theorem and prove

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

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

$$\mathsf{lazy}_{\mathsf{map}}\ g\ A^{**} \ = \ \mathsf{if}\ (A^{**} = \varnothing)\ \varnothing\ (g\ 0\ A^{**}) \tag{54}$$

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

8.2 Measurable Probabilistic Programs

We are now prepared to lift the previous theorems to probabilistic computations. XXX: after the following definitions, we're ready...

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

XXX: don't we need $g\ j\ A^*$ to be measurable for all $j \in J$?

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,* $\Sigma\ X$ *denotes its* ***standard σ-algebra**, which must be defined under the following constraints:*

$$\Sigma\ \langle X_1, X_2\rangle = \Sigma\ X_1 \otimes \Sigma\ X_2 \tag{55}$$

$$\Sigma\ (J \to X) = (\Sigma\ X)^{\otimes J} \tag{56}$$

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

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

$$\Sigma\ \mathsf{Bool} \ ::= \ \mathcal{P}\ \mathsf{Bool} \tag{57}$$

$$\Sigma\ T \ ::= \ \mathcal{P}\ T \tag{58}$$

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.* $\mathsf{arr}_{\mathsf{map}}\ (\mathsf{const}\ b)$ *is measurable if* $\{b\}$ *is a measurable set. For all* $j \in J$*,* $\mathsf{arr}_{\mathsf{map}}\ (\pi\ j)$ *is measurable.*

XXX: should that really be a lemma? $\mathsf{arr}_{\mathsf{map}}$ removes $\bot$ from the domain

Corollary 8.24. $\mathsf{arr}_{\mathsf{map^*}}\ \mathsf{id}$, $\mathsf{arr}_{\mathsf{map^*}}\ \mathsf{fst}$ *and* $\mathsf{arr}_{\mathsf{map^*}}\ \mathsf{snd}$ *are measurable.* $\mathsf{arr}_{\mathsf{map^*}}\ (\mathsf{const}\ b)$ *is measurable if* $\{b\}$ *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 (57). □

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

Proof. By structural induction and the above theorems. □

Theorem 8.28 (approximation with expressions)**.** *Let* $g := [\![e]\!]^{\Downarrow}_{\mathsf{map^*}}$ *converge, where* $g : X \underset{\mathsf{map^*}}{\leadsto} Y$*. For all* $t \in T$*, let* $A := (R \times \{t\}) \times X$*. There is an expression e' for which* $[\![e']\!]_{\mathsf{map^*}}\ j_0\ A = g\ j_0\ A$*.*

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

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

Proof. Let $\mathsf{g} := [\![e]\!]^{\Downarrow}_{\mathsf{map^*}}$ and $\mathsf{g}' := \mathsf{g\ j_0\ ((R \times T) \times X)}$. Because $\mathsf{g}' = \mathsf{g\ j_0\ A^*}$ where $\mathsf{A^*}$ is g's halting set, we need only show that g' is a measurable mapping.

By mapping arrow monotonicity (Theorem 8.11),

$$\mathsf{g}' \;=\; \bigcup_{\mathsf{t} \in \mathsf{T}} \mathsf{g\ j_0\ ((R \times \{t\}) \times X)} \qquad (59)$$

By Theorem 8.28, for every $\mathsf{t} \in \mathsf{T}$, there is an expression that computes $\mathsf{g\ ((R \times \{t\}) \times X)}$. By (58) and Theorem 8.27, each is measurable. By mapping arrow restriction (21), 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, if we do not mind measuring closures instead of function spaces, all higher-order programs are measurable. XXX: as written, this sounds like a cop-out

8.3 Random Store Probabilities

Preimages under probabilistic programs are measurable subsets of $\mathsf{(R \times T) \times X}$. While it is possible to put probability measures on such domains, doing so would be surprising for end-users. For example, the probabilities of outputs of the geometric function defined in (35) would depend not only on the probability of $\mathsf{random} < \mathsf{p}$, but also on some arbitrary probability that each branch is taken. It would not define the geometric distribution.

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

As previously, we start with measuring the halting set.

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

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

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

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 image fst distributes over unions, image fst A is a countable union of sets in $\Sigma\ \mathsf{X_1}$. □

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

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

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

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

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

9. Approximating Semantics

If we were to confine preimage computation to computably enumerable sets, we could model sets as streams and implement the preimage arrow almost directly. But we would like something more efficient, even if it means approximating.

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

9.1 Implementable Lifts

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

Fortunately, we need only certain lifts. Figure 2 (which defines $[\![\cdot]\!]_{\mathsf{a}}$) lifts id, const b, fst and snd. Sections 7.3 and 7.4, which define the combinators used to interpret partial, probabilistic programs, lift π j and agrees. Measurable functions made available as language primitives of course must be lifted to the preimage arrow.

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

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

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

9.2 Rectangular Families

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

$$\mathsf{Rect}\ \langle \mathsf{X_1}, \mathsf{X_2} \rangle \;=\; (\mathsf{Rect\ X_1}) \boxtimes (\mathsf{Rect\ X_2}) \qquad (60)$$

$$\mathsf{Rect}\ (\mathsf{J} \to \mathsf{X}) \;=\; (\mathsf{Rect\ X})^{\boxtimes \mathsf{J}} \qquad (61)$$

$$\begin{aligned}
&\mathsf{arr_{pre}\ id\ A} \;\equiv\; \langle \mathsf{A}, \lambda \mathsf{B}.\, \mathsf{B} \rangle \\[1ex]
&\mathsf{arr_{pre}\ fst\ A} \;\equiv\; \mathsf{let}\ \ \mathsf{A_1 := image\ fst\ A} \\
&\qquad\qquad\qquad\qquad \mathsf{A_2 := image\ snd\ A} \\
&\qquad\qquad\qquad\quad \mathsf{in}\ \ \langle \mathsf{A_1}, \lambda \mathsf{B}.\, \mathsf{A} \cap (\mathsf{B} \times \mathsf{A_2}) \rangle \\[1ex]
&\mathsf{arr_{pre}\ snd\ A} \;\equiv\; \mathsf{let}\ \ \mathsf{A_1 := image\ fst\ A} \\
&\qquad\qquad\qquad\qquad \mathsf{A_2 := image\ snd\ A} \\
&\qquad\qquad\qquad\quad \mathsf{in}\ \ \langle \mathsf{A_2}, \lambda \mathsf{B}.\, \mathsf{A} \cap (\mathsf{A_1} \times \mathsf{B}) \rangle
\end{aligned}$$

$$\begin{aligned}
&\mathsf{arr_{pre}\ (const\ b)\ A} \;\equiv\; \langle \{\mathsf{b}\}, \lambda \mathsf{B}.\, \mathsf{if}\ (\mathsf{B} = \varnothing)\ \varnothing\ \mathsf{A} \rangle \\[1ex]
&\mathsf{arr_{pre}}\ (\pi\ \mathsf{j})\ \mathsf{A} \;\equiv\; \langle \mathsf{image}\ (\pi\ \mathsf{j})\ \mathsf{A}, \mathsf{unproject\ j\ A} \rangle
\end{aligned}$$

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

Figure 8: Instances of $\mathsf{arr_{pre}\ f}$ necessary for interpreting probabilistic programs.

where

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

$$\mathcal{A}^{\boxtimes \mathsf{J}} \;:=\; \{\textstyle\prod_{\mathsf{j} \in \mathsf{J}} \mathsf{A_j} \mid \forall \mathsf{j} \in \mathsf{J}.\, \mathsf{A_j} \in \mathcal{A}\} \qquad (63)$$

lift cartesian products to sets of sets.

For example, if $\mathsf{Rect}\ \mathbb{R}$ contains all the closed real intervals, then by (60), $[0,2] \times [1,\pi] \in \mathsf{Rect}\ \langle \mathbb{R}, \mathbb{R} \rangle$.

For every non-product type X, we require $\varnothing \in \mathsf{Rect\ X}$, $\mathsf{X} \in \mathsf{Rect\ X}$, $\{\mathsf{a}\} \in \mathsf{Rect\ X}$ for all $\mathsf{a} \in \mathsf{X}$, 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 necessary operations can be exactly implemented if finite sets are modeled directly, sets in an ordered space (such as $\mathbb{R}$) are modeled by intervals, and sets in $\mathsf{Rect}\ \langle \mathsf{X_1}, \mathsf{X_2} \rangle$ are modeled by pairs of type $\langle \mathsf{Rect\ X_1}, \mathsf{Rect\ X_2} \rangle$. Though J is infinite, sets in $\mathsf{Rect}\ (\mathsf{J} \to \mathsf{X})$ 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 $\mathsf{J} \to \mathsf{X}$.

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

9.3 Approximate Preimage Mapping Operations

From the preimage mapping definitions (Figure 5), we see that $\mathsf{ap_{pre}}$, which $(\circ_{\mathsf{pre}})$ and $(\uplus_{\mathsf{pre}})$ depend on, can be implemented directly if $\mathsf{A} \cap \mathsf{B}$ is implemented. Given an $\mathsf{ap_{pre}}$ implementation, $(\circ_{\mathsf{pre}})$ 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 9.2 (pair preimage overapproximation)**.** *Let* $\mathsf{g_1} \in \mathsf{X} \rightharpoonup \mathsf{Y_1}$ *and* $\mathsf{g_2} \in \mathsf{X} \rightharpoonup \mathsf{Y_2}$*. For all* $\mathsf{B} \subseteq \mathsf{Y_1} \times \mathsf{Y_2}$, $\mathsf{preimage}\ \langle \mathsf{g_1}, \mathsf{g_2} \rangle_{\mathsf{map}}\ \mathsf{B} \subseteq \mathsf{preimage\ g_1\ (image\ fst\ B)} \cap \mathsf{preimage\ g_2\ (image\ snd\ B)}$*.*

Proof. By monotonicity of preimages and Lemma 5.4. □

Thus, the following replacement:

$$\begin{aligned}
&\langle \cdot, \cdot \rangle'_{\mathsf{pre}} : (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y_1}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y_2}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y_1} \times \mathsf{Y_2}) \\
&\langle \langle \mathsf{Y'_1}, \mathsf{p_1} \rangle, \langle \mathsf{Y'_2}, \mathsf{p_2} \rangle \rangle'_{\mathsf{pre}} \ := \qquad\qquad\qquad\qquad\qquad (64) \\
&\quad \langle \mathsf{Y'_1} \times \mathsf{Y'_2}, \lambda \mathsf{B}.\, \mathsf{p_1\ (image\ fst\ B)} \cap \mathsf{p_2\ (image\ snd\ B)} \rangle
\end{aligned}$$

computes covering rectangles of preimages under pairing.

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

$$\begin{aligned}
(\mathsf{A_1} \times \mathsf{A_2}) \vee (\mathsf{B_1} \times \mathsf{B_2}) \;&=\; (\mathsf{A_1} \vee \mathsf{B_1}) \times (\mathsf{A_2} \vee \mathsf{B_2}) \\
(\textstyle\prod_{\mathsf{j} \in \mathsf{J}} \mathsf{A_j}) \vee (\textstyle\prod_{\mathsf{j} \in \mathsf{J}} \mathsf{B_j}) \;&=\; \textstyle\prod_{\mathsf{j} \in \mathsf{J}} \mathsf{A_j} \vee \mathsf{B_j}
\end{aligned} \qquad (65)$$

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

For example, if $\mathsf{Rect}\ \mathbb{R}$ contains all the closed real intervals, then the following join is allowable:

$$[\mathsf{a_1}, \mathsf{b_1}] \vee [\mathsf{a_2}, \mathsf{b_2}] \;=\; [\mathsf{min\ a_1\ a_2}, \mathsf{max\ b_1\ b_2}] \qquad (66)$$

With this join, $([0,1] \times [0,1]) \vee ([2,3] \times [2,3]) = [0,3] \times [0,3]$.

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

$$\begin{aligned}
&(\uplus'_{\mathsf{pre}}) : (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y}) \\
&\mathsf{h_1} \uplus'_{\mathsf{pre}} \mathsf{h_2} \ := \ \mathsf{let}\ \ \mathsf{Y'} := (\mathsf{range_{pre}\ h_1}) \vee (\mathsf{range_{pre}\ h_2}) \\
&\qquad\qquad\qquad\qquad\quad \mathsf{p} := \lambda \mathsf{B}.\, (\mathsf{ap_{pre}\ h_1\ B}) \vee (\mathsf{ap_{pre}\ h_2\ B}) \\
&\qquad\qquad\qquad\ \ \mathsf{in}\ \ \langle \mathsf{Y'}, \mathsf{p} \rangle
\end{aligned} \qquad (67)$$

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

9.4 Partial Programs

We have enough of the specification to implement the preimage arrow and the AStore arrow transformer. But we cannot yet deal with programs that may not halt, or that halt with probability 1. For that, we need to approximate $\mathsf{ifte}^{\Downarrow}_{\mathsf{pre^*}}$ (43).

$$\begin{aligned}
&\mathsf{branch_{a^*}} : \mathsf{x} \rightsquigarrow_{\mathsf{a^*}} \mathsf{Bool} \\
&\mathsf{branch_{a^*}\ j} \ := \ \mathsf{arr_a}\ (\mathsf{fst} \ggg \mathsf{snd} \ggg \pi\ \mathsf{j})
\end{aligned} \qquad (68)$$

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

$$\begin{aligned}
&\mathsf{arr_{map}\ agrees\ (Bool \times Bool)} = \\
&\quad \{\langle \langle \mathsf{true}, \mathsf{true} \rangle, \mathsf{true} \rangle, \langle \langle \mathsf{false}, \mathsf{false} \rangle, \mathsf{false} \rangle\}
\end{aligned} \qquad (69)$$

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

$$
\begin{aligned}
&\mathsf{ifte}^{\Downarrow}_{\mathsf{pre*}}\ \mathsf{k_1\ k_2\ k_3\ j\ A}\ \equiv\\
&\quad \mathsf{let}\ \ \langle \mathsf{C_k}, \mathsf{p_k}\rangle := \mathsf{k_1\ j_1\ A}\\
&\quad\quad\quad \langle \mathsf{C_b}, \mathsf{p_b}\rangle := \mathsf{branch_{pre*}\ j\ A}\\
&\quad\quad\quad\quad\quad \mathsf{C_2} := \mathsf{C_k} \cap \mathsf{C_b} \cap \{\mathsf{true}\}\\
&\quad\quad\quad\quad\quad \mathsf{C_3} := \mathsf{C_k} \cap \mathsf{C_b} \cap \{\mathsf{false}\}\\
&\quad\quad\quad\quad\quad \mathsf{A_2} := \mathsf{p_k\ C_2} \cap \mathsf{p_b\ C_2}\\
&\quad\quad\quad\quad\quad \mathsf{A_3} := \mathsf{p_k\ C_3} \cap \mathsf{p_b\ C_3}\\
&\quad\quad \mathsf{in}\ \ (\mathsf{k_2\ j_2\ A_2}) \uplus_{\mathsf{pre}} (\mathsf{k_3\ j_3\ A_3})
\end{aligned}
\tag{70}
$$

where $\mathsf{j_1} = \mathsf{left\ j}$ and so on. This has no trace of agrees and clearly preserves rectangularity if $\mathsf{k_1}$, $\mathsf{k_2}$ and $\mathsf{k_3}$ do. Yet it is still not good enough. When $\mathsf{A_2}$ and $\mathsf{A_3}$ 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 altered $\mathsf{ifte}^{\Downarrow}_{\mathsf{pre*}}$ to take *no branches* when it would normally take two. A program run with $\mathsf{T} \subset \mathsf{J} \Rightarrow \mathsf{Bool}_\bot$ XXX: now prove that programs interpreted using this always converge...

If both $\mathsf{A_2}$ and $\mathsf{A_3}$ are nonempty, we know the preimage mapping's range is a subset of Y and that the preimages it computes are subsets of $\mathsf{A_2} \vee \mathsf{A_3}$. Therefore, we might replace the let body $(\mathsf{k_2\ j_2\ A_2}) \uplus_{\mathsf{pre}} (\mathsf{k_3\ j_3\ A_3})$ with

$$
\begin{aligned}
&\mathsf{if}\ (\mathsf{A_2} = \varnothing\ \mathsf{or}\ \mathsf{A_3} = \varnothing)\\
&\quad (\mathsf{k_2\ j_2\ A_2}) \uplus_{\mathsf{pre}} (\mathsf{k_3\ j_3\ A_3})\\
&\quad \langle \mathsf{Y}, \lambda\,\mathsf{B}.\,\mathsf{A_2} \vee \mathsf{A_3}\rangle
\end{aligned}
\tag{71}
$$

which computes the same preimages if $\mathsf{A_2}$ or $\mathsf{A_3}$ is empty, and takes no branches and overapproximates otherwise.

We cannot refer to Y in a function definition, however; it is only part of the type of $\mathsf{ifte}^{\Downarrow}_{\mathsf{pre*}}$. Thus, we need to add a universal set $\top$ to every $\mathsf{Rect\ X}$

XXX: for all X, $\top \in \mathsf{Rect\ X}$

10. Implementation

11. Unrelated Work

12. Related Work

13. Conclusions and Future Work

References

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

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