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

XXX: motivate conditional probability

XXX: densities fail:

1. infinite or variable-length data
2. discontinuities in random variables
3. nonlinear conditions

XXX: measures instead of densities, preimage measure

Purely functional languages do not allow effects, including randomness. (Of course, to be Turing-equivalent, "pure" languages usually allow divergence, which is an applicative effect.) 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. The strength of this approach lies in how it factors each probabilistic program into an effectful part and a pure part.

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

- "Preimage under f" makes sense only if f has an observable domain of definition. Lambdas do not.
- Probability measures can be defined only for *measurable* subsets of X and of Y. Thus, f is constrained: 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.
- Probability measures cannot be defined for arbitrary function spaces without imposing difficult restrictions.

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

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

XXX: rewrite the following

In this paper, we

1. Define the *bottom arrow*, type $\mathsf{X} \rightsquigarrow_{\perp} \mathsf{Y}$, as a compilation target for first-order functions that may raise errors.
2. Derive the *mapping arrow* from the bottom arrow, type $\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}$. Prove that its instances return extensional functions, or mappings, that compute the same values as corresponding bottom arrow computations.
3. Derive the *preimage arrow* from the mapping arrow, type $\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}$. Prove that its instances compute preimages under corresponding mapping (or bottom) arrow instances.
4. Define an arrow transformer via a natural transformation η to thread random stores and avoid divergence.
5. Define a computable approximation of the transformed preimage arrow and report on its implementation.

Most of our correctness theorems can be described in terms of the following commutative diagram:

$$
\begin{tikzcd}
\mathsf{X} \rightsquigarrow_{\perp} \mathsf{Y} \arrow[r, "\mathsf{lift}_{\mathsf{map}}"] \arrow[d, "\eta_{\perp *}"'] & \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y} \arrow[r, "\mathsf{lift}_{\mathsf{pre}}"] \arrow[d, "\eta_{\mathsf{map}*}"] & \mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y} \arrow[d, "\eta_{\mathsf{pre}*}"] \\
\mathsf{X} \rightsquigarrow_{\perp *} \mathsf{Y} \arrow[r, "\mathsf{lift}_{\mathsf{map}*}"'] & \mathsf{X} \underset{\mathsf{map}*}{\rightsquigarrow} \mathsf{Y} \arrow[r, "\mathsf{lift}_{\mathsf{pre}*}"'] & \mathsf{X} \underset{\mathsf{pre}*}{\rightsquigarrow} \mathsf{Y}
\end{tikzcd}
\tag{1}
$$

We prove that all the functions between arrow types in (1) are homomorphisms, which implies that $\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}$ and $\mathsf{X} \underset{\mathsf{pre}*}{\rightsquigarrow} \mathsf{Y}$ instances compute preimages correctly.

XXX: something about how we know the approximations are correct

2. Operational Metalanguage

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

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.

Contemporary mathematics is generally 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 contemporary mathematics 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 terminate.[1]

Almost everything definable in contemporary mathematics can be formally defined by a finite λ_{ZFC} program, except objects that most mathematicians would agree are nonconstructive. More precisely, any object 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 contemporary theorem applies to λ_{ZFC}'s set values without alteration. Further, proofs about λ_{ZFC}'s set values apply to contemporary mathematical objects.[2]

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

We assume data structures, including pairs, are encoded as *primitive* ordered pairs with the first element a unique tag, so that 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 an informal, manually checked, polymorphic type system characterized by these rules:

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

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

[1] An example of a nonterminating λ_{ZFC} function is one that attempts to decide whether other λ_{ZFC} programs halt.

[2] Assuming the existence of an inaccessible cardinal.

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

$$\begin{array}{rr} \mathsf{true} : \mathsf{Bool} & (\in) : \mathsf{x} \Rightarrow \mathsf{Set}\ \mathsf{x} \Rightarrow \mathsf{Bool} \\ \mathsf{false} : \mathsf{Bool} & \mathcal{P} : \mathsf{Set}\ \mathsf{x} \Rightarrow \mathsf{Set}\ (\mathsf{Set}\ \mathsf{x}) \\ \varnothing : \mathsf{Set}\ \mathsf{x} & \bigcup : \mathsf{Set}\ (\mathsf{Set}\ \mathsf{x}) \Rightarrow \mathsf{Set}\ \mathsf{x} \\ \omega : \mathsf{Ord} & \mathsf{image} : (\mathsf{x} \Rightarrow \mathsf{y}) \Rightarrow \mathsf{Set}\ \mathsf{x} \Rightarrow \mathsf{Set}\ \mathsf{y} \\ \mathsf{take} : \mathsf{Set}\ \mathsf{x} \Rightarrow \mathsf{x} & |\cdot| : \mathsf{Set}\ \mathsf{x} \Rightarrow \mathsf{Ord} \end{array} \tag{2}$$

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

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

We import contemporary 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

$$\mathsf{x} = \mathsf{y}\ :=\ \mathsf{x} \in \{\mathsf{y}\} \tag{3}$$

which means

$$(=)\ :=\ \lambda \mathsf{x}.\, \lambda \mathsf{y}.\, \mathsf{x} \in \{\mathsf{y}\} \tag{4}$$

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

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

$$\mathsf{domain} : (\mathsf{X} \rightharpoonup \mathsf{Y}) \Rightarrow \mathsf{Set\ X}$$
$$\mathsf{domain} \;:=\; \mathsf{image\ fst}$$

$$\mathsf{range} : (\mathsf{X} \rightharpoonup \mathsf{Y}) \Rightarrow \mathsf{Set\ Y}$$
$$\mathsf{range} \;:=\; \mathsf{image\ snd}$$

$$\mathsf{preimage} : (\mathsf{X} \rightharpoonup \mathsf{Y}) \Rightarrow \mathsf{Set\ Y} \Rightarrow \mathsf{Set\ X}$$
$$\mathsf{preimage\ f\ B} \;:=\; \{\mathsf{a} \in \mathsf{domain\ f} \mid \mathsf{f\ a} \in \mathsf{B}\}$$

$$\mathsf{restrict} : (\mathsf{X} \rightharpoonup \mathsf{Y}) \Rightarrow \mathsf{Set\ X} \Rightarrow (\mathsf{X} \rightharpoonup \mathsf{Y})$$
$$\mathsf{restrict\ f\ A} \;:=\; \lambda\,\mathsf{a} \in (\mathsf{A} \cap \mathsf{domain\ f}).\,\mathsf{f\ a}$$

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

$$(\circ_{\mathsf{map}}) : (\mathsf{Y} \rightharpoonup \mathsf{Z}) \Rightarrow (\mathsf{X} \rightharpoonup \mathsf{Y}) \Rightarrow (\mathsf{X} \rightharpoonup \mathsf{Z})$$
$$\begin{aligned}\mathsf{g}_2 \circ_{\mathsf{map}} \mathsf{g}_1 \;:=\; &\mathsf{let}\ \ \mathsf{A} := \mathsf{preimage\ g}_1\ (\mathsf{domain\ g}_2)\\ &\mathsf{in}\ \ \lambda\,\mathsf{a} \in \mathsf{A}.\,\mathsf{g}_2\ (\mathsf{g}_1\ \mathsf{a})\end{aligned}$$

$$(\uplus_{\mathsf{map}}) : (\mathsf{X} \rightharpoonup \mathsf{Y}) \Rightarrow (\mathsf{X} \rightharpoonup \mathsf{Y}) \Rightarrow (\mathsf{X} \rightharpoonup \mathsf{Y})$$
$$\begin{aligned}\mathsf{g}_1 \uplus_{\mathsf{map}} \mathsf{g}_2 \;:=\; &\mathsf{let}\ \ \mathsf{A} := (\mathsf{domain\ g}_1) \uplus (\mathsf{domain\ g}_2)\\ &\mathsf{in}\ \ \lambda\,\mathsf{a} \in \mathsf{A}.\,\mathsf{if}\ (\mathsf{a} \in \mathsf{domain\ g}_1)\ (\mathsf{g}_1\ \mathsf{a})\ (\mathsf{g}_2\ \mathsf{a})\end{aligned}$$

Figure 1: Operations on mappings.

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 may be rather informal with the latter two binding forms when the meaning is clear.) We assume we have logical operators, bounded quantifiers (unbounded quantifiers are not λ_{ZFC}-definable), and typical set operations.

A less typical set operation we use is disjoint union:

$$\begin{aligned}&(\uplus) : \mathsf{Set\ x} \Rightarrow \mathsf{Set\ x} \Rightarrow \mathsf{Set\ x}\\ &\mathsf{A} \uplus \mathsf{B} \;:=\; \mathsf{if}\ (\mathsf{A} \cap \mathsf{B} = \varnothing)\ (\mathsf{A} \cup \mathsf{B})\ (\mathsf{take}\ \varnothing)\end{aligned} \tag{5}$$

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

In set theory, functions are encoded as sets of input-output pairs. The increment function for the natural numbers, for example, is $\{\langle 0,1\rangle, \langle 1,2\rangle, \langle 2,3\rangle, ...\}$. To distinguish these huge tables from lambdas, we call them ***mappings***, and use the word **function** to mean a lambda or mapping.

Syntax for defining unnamed mappings is defined by

$$\lambda\, x_a \in e_A.\, e_b \;:\equiv\; \mathsf{mapping}\ (\lambda\, x_a.\, e_b)\ e_A \tag{6}$$

$$\begin{aligned}&\mathsf{mapping} : (\mathsf{X} \Rightarrow \mathsf{Y}) \Rightarrow \mathsf{Set\ X} \Rightarrow (\mathsf{X} \rightharpoonup \mathsf{Y})\\ &\mathsf{mapping\ f\ A} \;:=\; \mathsf{image}\ (\lambda\,\mathsf{a}.\,\langle \mathsf{a}, \mathsf{f\ a}\rangle)\ \mathsf{A}\end{aligned} \tag{7}$$

For convenience, as with lambdas, we use adjacency (e.g. $(\mathsf{f\ x})$) to apply mappings.

Figure 1 defines a few 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

$$\begin{aligned}&\pi : \mathsf{J} \Rightarrow (\mathsf{J} \to \mathsf{X}) \Rightarrow \mathsf{X}\\ &\pi\ \mathsf{j\ f} \;:=\; \mathsf{f\ j}\end{aligned} \tag{8}$$

which is useful in combinators, 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

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

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

In a nonstrict λ-calculus, simply defining a choice combinator allows writing recursive functions using nothing but arrow combinators and lifted, pure functions. However, any 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

$$\begin{aligned}\mathsf{arr\ f} \;&:=\; \mathsf{f}\\ (\mathsf{f}_1 \ggg \mathsf{f}_2)\ \mathsf{a} \;&:=\; \mathsf{f}_2\ (\mathsf{f}_1\ \mathsf{a})\\ (\mathsf{f}_1\ \&\&\&\ \mathsf{f}_2)\ \mathsf{a} \;&:=\; \langle \mathsf{f}_1\ \mathsf{a}, \mathsf{f}_2, \mathsf{a}\rangle\\ \mathsf{ifte\ f}_1\ \mathsf{f}_2\ \mathsf{f}_3\ \mathsf{a} \;&:=\; \mathsf{if}\ (\mathsf{f}_1\ \mathsf{a})\ (\mathsf{f}_2\ \mathsf{a})\ (\mathsf{f}_3\ \mathsf{a})\end{aligned} \tag{9}$$

and try to define the following recursive function:

$$\mathsf{halt\text{-}on\text{-}true} \;:=\; \mathsf{ifte\ (arr\ id)\ (arr\ id)\ halt\text{-}on\text{-}true} \tag{10}$$

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

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

$$\mathsf{halt\text{-}on\text{-}true} \;:=\; \mathsf{ifte\ (arr\ id)\ (arr\ id)\ (lazy}\ \lambda 0.\,\mathsf{halt\text{-}on\text{-}true)} \tag{11}$$

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*

$$\begin{aligned}\mathsf{arr_a} &: (\mathsf{x} \Rightarrow \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y})\\ (\ggg_{\mathsf{a}}) &: (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \Rightarrow (\mathsf{y} \rightsquigarrow_{\mathsf{a}} \mathsf{z}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{z})\\ (\&\&\&_{\mathsf{a}}) &: (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{z}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \langle \mathsf{y}, \mathsf{z}\rangle)\end{aligned} \tag{12}$$

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

$$\begin{aligned}\mathsf{ifte_a} &: (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{Bool}) \Rightarrow (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{y}) \\ \mathsf{lazy_a} &: (1 \Rightarrow (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{y})) \Rightarrow (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{y})\end{aligned} \tag{13}$$

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

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

$$\mathsf{lift_b}\ (\mathsf{arr_a}\ \mathsf{f}) \equiv \mathsf{arr_b}\ \mathsf{f} \tag{14}$$
$$\mathsf{lift_b}\ (\mathsf{f_1} \ggg_\mathsf{a} \mathsf{f_2}) \equiv (\mathsf{lift_b}\ \mathsf{f_1}) \ggg_\mathsf{b} (\mathsf{lift_b}\ \mathsf{f_2}) \tag{15}$$
$$\mathsf{lift_b}\ (\mathsf{f_1}\ \&\&\&_\mathsf{a}\ \mathsf{f_2}) \equiv (\mathsf{lift_b}\ \mathsf{f_1})\ \&\&\&_\mathsf{b}\ (\mathsf{lift_b}\ \mathsf{f_2}) \tag{16}$$
$$\mathsf{lift_b}\ (\mathsf{ifte_a}\ \mathsf{f_1}\ \mathsf{f_2}\ \mathsf{f_3}) \equiv \mathsf{ifte_b}\ (\mathsf{lift_b}\ \mathsf{f_1})\ (\mathsf{lift_b}\ \mathsf{f_2})\ (\mathsf{lift_b}\ \mathsf{f_3}) \tag{17}$$
$$\mathsf{lift_b}\ (\mathsf{lazy_a}\ \mathsf{f}) \equiv \mathsf{lazy_b}\ \lambda 0.\ \mathsf{lift_b}\ (\mathsf{f}\ 0) \tag{18}$$

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

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

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

3.2 First-Order Let-Calculus Semantics

Figure 2 shows a transformation $[\![\cdot]\!]_\mathsf{a}$ 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 $\mathsf{0}$ 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 $[\![\cdot]\!]_\mathsf{a}$ would complicate the simple proof of the following theorem, which underlies most of our semantic correctness claims.

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

Proof. By structural induction on program terms.

Bases cases proceed by expansion and using $\mathsf{lift_b} \circ \mathsf{arr_a} \equiv \mathsf{arr_b}$ (14). For example, for constants:

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

Inductive cases proceed by expansion, applying one or more distributive laws (15–18), and applying the inductive hypothesis on subterms. For example, for pairing:

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

It is not hard to check the remaining cases. □

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

4. The Bottom and Mapping Arrows

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

One obvious candidate for the simpler arrow is the function arrow, defined in (9). 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 $\mathsf{Bool} \cup \{\perp\}$.

Theorem 4.1. $\mathsf{arr}_\perp$, $(\&\&\&_\perp)$, $(\ggg_\perp)$, $\mathsf{ifte}_\perp$ *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

The contemporary theorems we need are about mappings, not about lambdas that may raise errors. As in intermediate step, then, we need an arrow whose computations produce mappings or are mappings themselves.

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

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

$$\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y} ::= \mathsf{Set}\ \mathsf{X} \Rightarrow (\mathsf{X} \rightharpoonup \mathsf{Y}) \tag{19}$$

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

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

$$\mathsf{g}\ \mathsf{A}' = \mathsf{restrict}\ (\mathsf{g}\ \mathsf{A})\ \mathsf{A}' \tag{20}$$

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

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

$$
\begin{aligned}
&\mathsf{x} \rightsquigarrow_{\perp} \mathsf{y} \ ::=\ \mathsf{x} \Rightarrow \mathsf{y}_{\perp} \\[4pt]
&\mathsf{arr}_{\perp} : (\mathsf{x} \Rightarrow \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\perp} \mathsf{y}) \\
&\mathsf{arr}_{\perp}\ \mathsf{f} \ :=\ \mathsf{f} \\[4pt]
&(\ggg_{\perp}) : (\mathsf{x} \rightsquigarrow_{\perp} \mathsf{y}) \Rightarrow (\mathsf{y} \rightsquigarrow_{\perp} \mathsf{z}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\perp} \mathsf{z}) \\
&(\mathsf{f}_1 \ggg_{\perp} \mathsf{f}_2)\ \mathsf{a} \ :=\ \mathsf{if}\ (\mathsf{f}_1\ \mathsf{a} = \perp)\ \perp\ (\mathsf{f}_2\ (\mathsf{f}_1\ \mathsf{a})) \\[4pt]
&(\&\&\&_{\perp}) : (\mathsf{x} \rightsquigarrow_{\perp} \mathsf{y}_1) \Rightarrow (\mathsf{x} \rightsquigarrow_{\perp} \mathsf{y}_2) \Rightarrow (\mathsf{x} \rightsquigarrow_{\perp} \langle \mathsf{y}_1, \mathsf{y}_2 \rangle) \\
&(\mathsf{f}_1\ \&\&\&_{\perp}\ \mathsf{f}_2)\ \mathsf{a} \ :=\ \mathsf{if}\ (\mathsf{f}_1\ \mathsf{a} = \perp\ \mathsf{or}\ \mathsf{f}_2\ \mathsf{a} = \perp)\ \perp\ \langle \mathsf{f}_1\ \mathsf{a}, \mathsf{f}_2\ \mathsf{a} \rangle
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{ifte}_{\perp} : (\mathsf{x} \rightsquigarrow_{\perp} \mathsf{Bool}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\perp} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\perp} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\perp} \mathsf{y}) \\
&\mathsf{ifte}_{\perp}\ \mathsf{f}_1\ \mathsf{f}_2\ \mathsf{f}_3\ \mathsf{a} \ :=\ \mathsf{case}\ \ \mathsf{f}_1\ \mathsf{a} \\
&\qquad\qquad \mathsf{true} \implies \mathsf{f}_2\ \mathsf{a} \\
&\qquad\qquad \mathsf{false} \implies \mathsf{f}_3\ \mathsf{a} \\
&\qquad\qquad \mathsf{else} \implies \perp \\[4pt]
&\mathsf{lazy}_{\perp} : (1 \Rightarrow (\mathsf{x} \rightsquigarrow_{\perp} \mathsf{y})) \Rightarrow (\mathsf{x} \rightsquigarrow_{\perp} \mathsf{y}) \\
&\mathsf{lazy}_{\perp}\ \mathsf{f}\ \mathsf{a} \ :=\ \mathsf{f}\ 0\ \mathsf{a}
\end{aligned}
$$

Figure 3: Bottom arrow definitions.

$$
\begin{aligned}
&\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y} \ ::=\ \mathsf{Set\ X} \Rightarrow (\mathsf{X} \rightharpoonup \mathsf{Y}) \\[4pt]
&\mathsf{arr_{map}} : (\mathsf{X} \Rightarrow \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}) \\
&\mathsf{arr_{map}} \ :=\ \mathsf{lift_{map}} \circ \mathsf{arr}_{\perp} \\[4pt]
&(\ggg_{\mathsf{map}}) : (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}) \Rightarrow (\mathsf{Y} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Z}) \Rightarrow (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Z}) \\
&(\mathsf{g}_1 \ggg_{\mathsf{map}} \mathsf{g}_2)\ \mathsf{A} \ :=\ \mathsf{let}\ \ \mathsf{g}_1' := \mathsf{g}_1\ \mathsf{A} \\
&\qquad\qquad\qquad\qquad\quad\ \ \mathsf{g}_2' := \mathsf{g}_2\ (\mathsf{range}\ \mathsf{g}_1') \\
&\qquad\qquad\qquad\qquad\ \ \mathsf{in}\ \ \mathsf{g}_2' \circ_{\mathsf{map}} \mathsf{g}_1' \\[4pt]
&(\&\&\&_{\mathsf{map}}) : (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}_1) \Rightarrow (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}_2) \Rightarrow (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \langle \mathsf{Y}_1, \mathsf{Y}_2 \rangle) \\
&(\mathsf{g}_1\ \&\&\&_{\mathsf{map}}\ \mathsf{g}_2)\ \mathsf{A} \ :=\ \langle \mathsf{g}_1\ \mathsf{A}, \mathsf{g}_2\ \mathsf{A} \rangle_{\mathsf{map}}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{ifte_{map}} : (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Bool}) \Rightarrow (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}) \\
&\mathsf{ifte_{map}}\ \mathsf{g}_1\ \mathsf{g}_2\ \mathsf{g}_3\ \mathsf{A} \ :=\ \mathsf{let}\ \ \mathsf{g}_1' := \mathsf{g}_1\ \mathsf{A} \\
&\qquad\qquad\qquad\qquad\quad\ \ \mathsf{g}_2' := \mathsf{g}_2\ (\mathsf{preimage}\ \mathsf{g}_1'\ \{\mathsf{true}\}) \\
&\qquad\qquad\qquad\qquad\quad\ \ \mathsf{g}_3' := \mathsf{g}_3\ (\mathsf{preimage}\ \mathsf{g}_1'\ \{\mathsf{false}\}) \\
&\qquad\qquad\qquad\qquad\ \ \mathsf{in}\ \ \mathsf{g}_2' \uplus_{\mathsf{map}} \mathsf{g}_3' \\[4pt]
&\mathsf{lazy_{map}} : (1 \Rightarrow (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y})) \Rightarrow (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}) \\
&\mathsf{lazy_{map}}\ \mathsf{g}\ \mathsf{A} \ :=\ \mathsf{if}\ (\mathsf{A} = \varnothing)\ \varnothing\ (\mathsf{g}\ 0\ \mathsf{A}) \\[4pt]
&\mathsf{lift_{map}} : (\mathsf{X} \rightsquigarrow_{\perp} \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}) \\
&\mathsf{lift_{map}}\ \mathsf{f}\ \mathsf{A} \ :=\ \{ \langle \mathsf{a}, \mathsf{b} \rangle \in \mathsf{mapping\ f\ A} \mid \mathsf{b} \neq \perp \}
\end{aligned}
$$

Figure 4: Mapping arrow definitions.

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:

$$
\begin{aligned}
&\mathsf{domain}_{\perp} : (\mathsf{X} \rightsquigarrow_{\perp} \mathsf{Y}) \Rightarrow \mathsf{Set\ X} \Rightarrow \mathsf{Set\ X} \\
&\mathsf{domain}_{\perp}\ \mathsf{f}\ \mathsf{A} \ :=\ \{ \mathsf{a} \in \mathsf{A} \mid \mathsf{f}\ \mathsf{a} \neq \perp \}
\end{aligned}
\tag{21}
$$

$$
\begin{aligned}
&\mathsf{lift_{map}} : (\mathsf{X} \rightsquigarrow_{\perp} \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}) \\
&\mathsf{lift_{map}}\ \mathsf{f}\ \mathsf{A} \ :=\ \mathsf{mapping\ f}\ (\mathsf{domain}_{\perp}\ \mathsf{f\ A})
\end{aligned}
\tag{22}
$$

So $\mathsf{lift_{map}}\ \mathsf{f\ A}$ 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 than observational equivalence.

Definition 4.2 (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* $\mathsf{g}_1\ \mathsf{A} \equiv \mathsf{g}_2\ \mathsf{A}$ *for all* $\mathsf{A} \subseteq \mathsf{X}$.

Clearly $\mathsf{arr_b} := \mathsf{lift_b} \circ \mathsf{arr_a}$ meets the first homomorphism identity (14), so we define $\mathsf{arr_{map}}$ as a composition. The following subsections derive $(\&\&\&_{\mathsf{map}})$, $(\ggg_{\mathsf{map}})$, $\mathsf{ifte_{map}}$ 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.

4.1.1 Case: Pairing

Starting with the left side of (16), 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 $\mathsf{A} \subseteq \mathsf{X}$,

$$\begin{aligned}
&\mathsf{lift}_{\mathsf{map}}\ (\mathsf{f}_1\ \&\!\&\!\&_\bot\ \mathsf{f}_2)\ \mathsf{A} \\
&\quad \equiv \mathsf{lift}_{\mathsf{map}}\ (\lambda \mathsf{a}.\,\mathsf{if}\ (\mathsf{f}_1\ \mathsf{a} = \bot\ \mathsf{or}\ \mathsf{f}_2\ \mathsf{a} = \bot)\ \bot\ \langle \mathsf{f}_1\ \mathsf{a}, \mathsf{f}_2\ \mathsf{a}\rangle)\ \mathsf{A} \\
&\quad \equiv \mathsf{let}\ \ \mathsf{f} := \lambda \mathsf{a}.\,\mathsf{if}\ (\mathsf{f}_1\ \mathsf{a} = \bot\ \mathsf{or}\ \mathsf{f}_2\ \mathsf{a} = \bot)\ \bot\ \langle \mathsf{f}_1\ \mathsf{a}, \mathsf{f}_2\ \mathsf{a}\rangle \\
&\qquad\quad \mathsf{in}\ \ \mathsf{mapping}\ \mathsf{f}\ (\mathsf{domain}_\bot\ \mathsf{f}\ \mathsf{A})
\end{aligned} \tag{23}$$

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 $\mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_2$:

$$\begin{aligned}
&\mathsf{lift}_{\mathsf{map}}\ (\mathsf{f}_1\ \&\!\&\!\&_\bot\ \mathsf{f}_2)\ \mathsf{A} \\
&\quad \equiv \mathsf{let}\ \ \mathsf{A}_1 := (\mathsf{domain}_\bot\ \mathsf{f}_1\ \mathsf{A}) \\
&\qquad\qquad \mathsf{A}_2 := (\mathsf{domain}_\bot\ \mathsf{f}_2\ \mathsf{A}) \\
&\qquad\qquad \mathsf{A}' := \mathsf{A}_1 \cap \mathsf{A}_2 \\
&\qquad\quad \mathsf{in}\ \ \lambda \mathsf{a} \in \mathsf{A}'.\,\langle \mathsf{f}_1\ \mathsf{a}, \mathsf{f}_2\ \mathsf{a}\rangle \\
&\quad \equiv \mathsf{let}\ \ \mathsf{g}_1 := \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_1\ \mathsf{A} \\
&\qquad\qquad \mathsf{g}_2 := \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_2\ \mathsf{A} \\
&\qquad\qquad \mathsf{A}' := (\mathsf{domain}\ \mathsf{g}_1) \cap (\mathsf{domain}\ \mathsf{g}_2) \\
&\qquad\quad \mathsf{in}\ \ \lambda \mathsf{a} \in \mathsf{A}'.\,\langle \mathsf{g}_1\ \mathsf{a}, \mathsf{g}_2\ \mathsf{a}\rangle \\
&\quad \equiv \langle \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_1\ \mathsf{A}, \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_2\ \mathsf{A}\rangle_{\mathsf{map}}
\end{aligned} \tag{24}$$

Substituting g_1 for $\mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_1$ and g_2 for $\mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_2$ gives a definition for $(\&\!\&\!\&_{\mathsf{map}})$ (Figure 4) for which (16) holds.

4.1.2 Case: Composition

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

XXX: include it?

4.1.3 Case: Conditional

Starting with the left side of (17), we expand definitions, and simplify f by restricting it to a domain for which $\mathsf{f}_1\ \mathsf{a} \neq \bot$:

$$\begin{aligned}
&\mathsf{lift}_{\mathsf{map}}\ (\mathsf{ifte}_\bot\ \mathsf{f}_1\ \mathsf{f}_2\ \mathsf{f}_3)\ \mathsf{A} \\
&\quad \equiv \mathsf{let}\ \ \mathsf{f} := \lambda \mathsf{a}.\,\mathsf{case}\ \ \mathsf{f}_1\ \mathsf{a} \\
&\qquad\qquad\qquad\qquad \mathsf{true}\ \Longrightarrow\ \mathsf{f}_2\ \mathsf{a} \\
&\qquad\qquad\qquad\qquad \mathsf{false}\ \Longrightarrow\ \mathsf{f}_3\ \mathsf{a} \\
&\qquad\qquad\qquad\qquad \mathsf{else}\ \Longrightarrow\ \bot \\
&\qquad\quad \mathsf{in}\ \ \mathsf{mapping}\ \mathsf{f}\ (\mathsf{domain}_\bot\ \mathsf{f}\ \mathsf{A}) \\
&\quad \equiv \mathsf{let}\ \ \mathsf{g}_1 := \mathsf{mapping}\ \mathsf{f}\ \mathsf{A} \\
&\qquad\qquad \mathsf{A}_2 := \mathsf{preimage}\ \mathsf{g}_1\ \{\mathsf{true}\} \\
&\qquad\qquad \mathsf{A}_3 := \mathsf{preimage}\ \mathsf{g}_1\ \{\mathsf{false}\} \\
&\qquad\qquad \mathsf{f} := \lambda \mathsf{a}.\,\mathsf{if}\ (\mathsf{f}_1\ \mathsf{a})\ (\mathsf{f}_2\ \mathsf{a})\ (\mathsf{f}_3\ \mathsf{a}) \\
&\qquad\quad \mathsf{in}\ \ \mathsf{mapping}\ \mathsf{f}\ (\mathsf{domain}_\bot\ \mathsf{f}\ (\mathsf{A}_2 \uplus \mathsf{A}_3))
\end{aligned} \tag{25}$$

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

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

Substituting g_1 for $\mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_1$, g_2 for $\mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_2$, and g_3 for $\mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_3$ gives a definition for $\mathsf{ifte}_{\mathsf{map}}$ (Figure 4) for which (17) holds.

4.1.4 Case: Laziness

Starting with the left side of (18), we first expand definitions:

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

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

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

Substituting $\mathsf{g}\ 0$ for $\mathsf{lift}_{\mathsf{map}}\ (\mathsf{f}\ 0)$ gives a definition for $\mathsf{lazy}_{\mathsf{map}}$ (Figure 4) for which (18) holds.

4.2 Correctness

Theorem 4.3 (mapping arrow correctness)**.** $\mathsf{lift}_{\mathsf{map}}$ *is an arrow homomorphism.*

Proof. By construction. □

Corollary 4.4 (semantic correctness)**.** *For all programs e,* $[\![e]\!]_{\mathsf{map}} \equiv \mathsf{lift}_{\mathsf{map}}\ [\![e]\!]_\bot$.

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. Therefore, in the preimage arrow, we will confine computation on points to instances of

$$\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y}\ \ ::=\ \ \langle \mathsf{Set}\ \mathsf{Y}, \mathsf{Set}\ \mathsf{Y} \Rightarrow \mathsf{Set}\ \mathsf{X}\rangle \tag{27}$$

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

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

$$\begin{aligned}
&\mathsf{pre} : (\mathsf{X} \rightharpoonup \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y}) \\
&\mathsf{pre}\ \mathsf{g}\ \ :=\ \ \mathsf{let}\ \ \mathsf{Y}' := \mathsf{range}\ \mathsf{g} \\
&\qquad\qquad\qquad\quad \mathsf{p} := \lambda \mathsf{B}.\,\mathsf{preimage}\ \mathsf{g}\ \mathsf{B} \\
&\qquad\qquad\quad \mathsf{in}\ \ \langle \mathsf{Y}', \mathsf{p}\rangle
\end{aligned} \tag{28}$$

Applying a preimage mapping to any subset of its codomain:

$$\begin{aligned}
&\mathsf{pre\text{-}ap} : (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y}) \Rightarrow \mathsf{Set}\ \mathsf{Y} \Rightarrow \mathsf{Set}\ \mathsf{X} \\
&\mathsf{pre\text{-}ap}\ \langle \mathsf{Y}', \mathsf{p}\rangle\ \mathsf{B}\ \ :=\ \ \mathsf{p}\ (\mathsf{B} \cap \mathsf{Y}')
\end{aligned} \tag{29}$$

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

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

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

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

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

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

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

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

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

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

Figure 5: Lazy preimage mappings and operations.

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

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

Definition 5.3 (preimage mapping equivalence)**.** *Two preimage mappings* $\mathsf{h_1} : \mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y}$ *and* $\mathsf{h_2} : \mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y}$ *are equivalent, or* $\mathsf{h_1} \equiv \mathsf{h_2}$*, when* $\mathsf{pre\text{-}ap}\ \mathsf{h_1}\ \mathsf{B} \equiv \mathsf{pre\text{-}ap}\ \mathsf{h_2}\ \mathsf{B}$ *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 (preimage distributes over $\langle \cdot, \cdot \rangle_{\mathsf{map}}$ and $(\times)$)**.** *Let* $\mathsf{g_1} \in \mathsf{X} \rightharpoonup \mathsf{Y_1}$ *and* $\mathsf{g_2} \in \mathsf{X} \rightharpoonup \mathsf{Y_2}$*. For all* $\mathsf{B_1} \subseteq \mathsf{Y_1}$ *and* $\mathsf{B_2} \subseteq \mathsf{Y_2}$, $\mathsf{preimage}\ \langle \mathsf{g_1}, \mathsf{g_2} \rangle_{\mathsf{map}}\ (\mathsf{B_1} \times \mathsf{B_2}) = (\mathsf{preimage}\ \mathsf{g_1}\ \mathsf{B_1}) \cap (\mathsf{preimage}\ \mathsf{g_2}\ \mathsf{B_2})$.

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

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

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

□

5.0.2 Preimage Mapping Composition

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

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

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

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

□

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 $B \subseteq Y$,

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

□

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 $X \underset{\mathsf{pre}}{\leadsto} Y ::= X \underset{\mathsf{pre}}{\rightharpoonup} Y$: 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 $\mathsf{Set}\ X$:

$$X \underset{\mathsf{pre}}{\leadsto} Y\ \ ::=\ \ \mathsf{Set}\ X \Rightarrow (X \underset{\mathsf{pre}}{\rightharpoonup} Y) \tag{30}$$

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:

$$\begin{aligned}
&\mathsf{lift}_{\mathsf{pre}} : (X \underset{\mathsf{map}}{\leadsto} Y) \Rightarrow (X \underset{\mathsf{pre}}{\leadsto} Y) \\
&\mathsf{lift}_{\mathsf{pre}}\ g\ A\ :=\ \mathsf{pre}\ (g\ A)
\end{aligned} \tag{31}$$

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

$$\mathsf{pre\text{-}ap}\ (\mathsf{lift}_{\mathsf{pre}}\ g\ A)\ B \equiv \mathsf{preimage}\ (g\ A)\ B \tag{32}$$

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

We also need a coarser notion of equivalence.

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

As with $\mathsf{arr}_{\mathsf{map}}$, defining $\mathsf{arr}_{\mathsf{pre}}$ as a composition meets (14). The following subsections derive $(\&\&\&_{\mathsf{pre}})$, $(\ggg_{\mathsf{pre}})$, $\mathsf{ifte}_{\mathsf{pre}}$ and $\mathsf{lazy}_{\mathsf{pre}}$ from their corresponding mapping arrow combinators, in a way that ensures $\mathsf{lift}_{\mathsf{pre}}$ is an arrow homomorphism from the mapping arrow to the preimage arrow. Figure 6 contains the resulting definitions.

6.1 Case: Pairing

Starting with the left side of (16), we expand definitions, apply Theorem 5.5, and rewrite in terms of $\mathsf{lift}_{\mathsf{pre}}$:

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

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

6.2 Case: Composition

Starting with the left side of (15), we expand definitions, apply Theorem 5.7 and rewrite in terms of $\mathsf{lift}_{\mathsf{pre}}$:

$$\begin{aligned}
&\mathsf{pre\text{-}ap}\ (\mathsf{lift}_{\mathsf{pre}}\ (g_1 \ggg_{\mathsf{map}} g_2)\ A)\ C \\
&\quad \equiv \mathsf{let}\ \ g'_1 := g_1\ A \\
&\qquad\qquad g'_2 := g_2\ (\mathsf{range}\ g'_1) \\
&\qquad\quad \mathsf{in}\ \ \mathsf{pre\text{-}ap}\ (\mathsf{pre}\ (g'_2 \circ_{\mathsf{map}} g'_1))\ C \\
&\quad \equiv \mathsf{let}\ \ g'_1 := g_1\ A \\
&\qquad\qquad g'_2 := g_2\ (\mathsf{range}\ g'_1) \\
&\qquad\quad \mathsf{in}\ \ \mathsf{pre\text{-}ap}\ ((\mathsf{pre}\ g'_1) \circ_{\mathsf{pre}} (\mathsf{pre}\ g'_2))\ C \\
&\quad \equiv \mathsf{let}\ \ h_1 := \mathsf{lift}_{\mathsf{pre}}\ g_1\ A \\
&\qquad\qquad h_2 := \mathsf{lift}_{\mathsf{pre}}\ g_2\ (\mathsf{pre\text{-}range}\ h_1) \\
&\qquad\quad \mathsf{in}\ \ \mathsf{pre\text{-}ap}\ (h_2 \circ_{\mathsf{pre}} h_1)\ C
\end{aligned} \tag{33}$$

Substituting h_1 for $\mathsf{lift}_{\mathsf{pre}}\ g_1$ and h_2 for $\mathsf{lift}_{\mathsf{pre}}\ g_2$, and removing the application of pre-ap from both sides of the equivalence gives a definition of $(\ggg_{\mathsf{pre}})$ (Figure 6) for which (15) holds.

6.3 Case: Conditional

Starting with the left side of (17), we expand terms, apply Theorem 5.9, rewrite in terms of $\mathsf{lift}_{\mathsf{pre}}$, and apply Theorem 5.2 in the definitions of h_2 and h_3:

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

Substituting h_1 for $\mathsf{lift}_{\mathsf{pre}}\ g_1$, h_2 for $\mathsf{lift}_{\mathsf{pre}}\ g_2$ and h_3 for $\mathsf{lift}_{\mathsf{pre}}\ g_3$, and removing the application of pre-ap from both sides of the equivalence gives a definition of $\mathsf{ifte}_{\mathsf{pre}}$ (Figure 6) for which (17) holds.

6.4 Case: Laziness

Starting with the left side of (18), expand definitions, distribute pre over the branches of if, and rewrite in terms of

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

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

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

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

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

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

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

Figure 6: Preimage arrow definitions.

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

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

Substituting $\mathsf{h}\ 0$ for $\mathsf{lift_{pre}}\ (\mathsf{g}\ 0)$ and removing the application of pre-ap from both sides of the equivalence gives a definition for $\mathsf{lazy_{pre}}$ (Figure 6) for which (18) 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

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

$$\mathsf{geometric}\ \mathsf{p} \ := \ \mathsf{if}\ (\mathsf{random} < \mathsf{p})\ 0\ (1 + \mathsf{geometric}\ \mathsf{p}) \tag{34}$$

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

Suppose we interpret (34) as $\mathsf{h} : \mathsf{R} \underset{\mathsf{pre}}{\rightsquigarrow} \mathbb{N}$, a preimage arrow computation from random sources in R to natural numbers, and that we have a probability measure $\mathsf{P} \in \mathcal{P}\ \mathsf{R} \rightharpoonup [0,1]$. We could compute the probability of any output set $\mathsf{N} \subseteq \mathbb{N}$ using $\mathsf{P}\ (\mathsf{h}\ \mathsf{R}'\ \mathsf{N})$, 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 sub-trees to sub-computations.

We have tried both ways with arrows defined using pairing. The resulting definitions are large, making them conceptually difficult and hard to manipulate. Fortunately, assigning each sub-computation a unique index into a tree-shaped random source, and passing it unchanged, is relatively easy.

We need a way to assign unique indexes to expressions.

Definition 7.1 (binary indexing scheme). *Let* J *be an index set,* $\mathsf{j_0} \in \mathsf{J}$ *a distinguished element, and* $\mathsf{left} : \mathsf{J} \Rightarrow \mathsf{J}$ *and* $\mathsf{right} : \mathsf{J} \Rightarrow \mathsf{J}$ *be total functions. These define a* ***binary indexing scheme*** *if*

- *Every finite composition* next *of* left *and* right *is injective.*
- *For all* $\mathsf{j} \in \mathsf{J}$, $\mathsf{j} = \mathsf{next}\ \mathsf{j_0}$ *for some finite composition* next.

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

$$\begin{aligned}\mathsf{left}\ (\mathsf{p}/\mathsf{q}) \ &:= \ (\mathsf{p} - \tfrac{1}{2})/\mathsf{q}\\ \mathsf{right}\ (\mathsf{p}/\mathsf{q}) \ &:= \ (\mathsf{p} + \tfrac{1}{2})/\mathsf{q}\end{aligned} \tag{35}$$

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.

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:

$$\mathsf{AStore\ s\ (x \rightsquigarrow_a y)} \ ::= \ \mathsf{J} \Rightarrow (\langle \mathsf{s}, \mathsf{x} \rangle \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \tag{36}$$

XXX: motivational wurds for definition of lift:

$$\begin{aligned} &\eta_{\mathsf{a}^*} : (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \Rightarrow \mathsf{AStore\ s\ (x \rightsquigarrow_a y)} \\ &\eta_{\mathsf{a}^*}\ \mathsf{f\ j} \ := \ \mathsf{arr_a\ snd} \ggg_{\mathsf{a}} \mathsf{f} \end{aligned} \tag{37}$$

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 $\mathsf{AStore\ s\ (x \rightsquigarrow_a y)}$.

Definition 7.2 (AStore arrow equivalence)**.** *Two* AStore *arrow computations* $\mathsf{k_1}$ *and* $\mathsf{k_2}$ *are equivalent, or* $\mathsf{k_1} \equiv \mathsf{k_2}$*, when* $\mathsf{k_1\ j} \equiv \mathsf{k_2\ j}$ *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* $\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y} ::= \mathsf{AStore\ s\ (x \rightsquigarrow_a y)}$*. Then* η_{a^*} *is an arrow homomorphism.*

Proof. Defining $\mathsf{arr_{a^*}}$ as a composition clearly meets the first homomorphism identity (14).

Composition. Starting with the right side of (15), expand definitions and use $(\mathsf{arr_a\ f} \ \&\&\&_{\mathsf{a}}\ \mathsf{f_1}) \ggg_{\mathsf{a}} \mathsf{arr_a\ snd} \equiv \mathsf{f_1}$:

$$\begin{aligned} &(\eta_{\mathsf{a}^*}\ \mathsf{f_1} \ggg_{\mathsf{a}^*} \eta_{\mathsf{a}^*}\ \mathsf{f_2})\ \mathsf{j} \\ &\quad \equiv \ (\mathsf{arr_a\ fst} \ \&\&\&_{\mathsf{a}}\ (\mathsf{arr_a\ snd} \ggg_{\mathsf{a}} \mathsf{f_1})) \ggg_{\mathsf{a}} \mathsf{arr_a\ snd} \ggg_{\mathsf{a}} \mathsf{f_2} \\ &\quad \equiv \ \mathsf{arr_a\ snd} \ggg_{\mathsf{a}} \mathsf{f_1} \ggg_{\mathsf{a}} \mathsf{f_2} \\ &\quad \equiv \ \eta_{\mathsf{a}^*}\ (\mathsf{f_1} \ggg_{\mathsf{a}} \mathsf{f_2})\ \mathsf{j} \end{aligned}$$

Pairing. Starting with the right side of (16), expand definitions and use the arrow law $\mathsf{arr_a\ f} \ggg_{\mathsf{a}} (\mathsf{f_1} \ \&\&\&_{\mathsf{a}}\ \mathsf{f_2}) \equiv (\mathsf{arr_a\ f} \ggg_{\mathsf{a}} \mathsf{f_1}) \ \&\&\&_{\mathsf{a}}\ (\mathsf{arr_a\ f} \ggg_{\mathsf{a}} \mathsf{f_2})$:

$$\begin{aligned} &(\eta_{\mathsf{a}^*}\ \mathsf{f_1} \ \&\&\&_{\mathsf{a}^*}\ \eta_{\mathsf{a}^*}\ \mathsf{f_2})\ \mathsf{j} \\ &\quad \equiv \ (\mathsf{arr_a\ snd} \ggg_{\mathsf{a}} \mathsf{f_1}) \ \&\&\&_{\mathsf{a}}\ (\mathsf{arr_a\ snd} \ggg_{\mathsf{a}} \mathsf{f_2}) \\ &\quad \equiv \ \mathsf{arr_a\ snd} \ggg_{\mathsf{a}} (\mathsf{f_1} \ \&\&\&_{\mathsf{a}}\ \mathsf{f_2}) \\ &\quad \equiv \ \eta_{\mathsf{a}^*}\ (\mathsf{f_1} \ \&\&\&_{\mathsf{a}}\ \mathsf{f_2})\ \mathsf{j} \end{aligned}$$

Conditional. Starting with the right side of (17), expand definitions and use the arrow law $\mathsf{arr_a\ f} \ggg_{\mathsf{a}} \mathsf{ifte_a\ f_1\ f_2\ f_3} \equiv \mathsf{ifte_a}\ (\mathsf{arr_a\ f} \ggg_{\mathsf{a}} \mathsf{f_1})\ (\mathsf{arr_a\ f} \ggg_{\mathsf{a}} \mathsf{f_2})\ (\mathsf{arr_a\ f} \ggg_{\mathsf{a}} \mathsf{f_3})$:

$$\begin{aligned} &(\mathsf{ifte_{a^*}}\ (\eta_{\mathsf{a}^*}\ \mathsf{f_1})\ (\eta_{\mathsf{a}^*}\ \mathsf{f_2})\ (\eta_{\mathsf{a}^*}\ \mathsf{f_3}))\ \mathsf{j} \\ &\quad \equiv \ \mathsf{ifte_a}\ (\mathsf{arr_a\ snd} \ggg_{\mathsf{a}} \mathsf{f_1}) \\ &\qquad\qquad\quad (\mathsf{arr_a\ snd} \ggg_{\mathsf{a}} \mathsf{f_2}) \\ &\qquad\qquad\quad (\mathsf{arr_a\ snd} \ggg_{\mathsf{a}} \mathsf{f_3}) \\ &\quad \equiv \ \mathsf{arr_a\ snd} \ggg_{\mathsf{a}} (\mathsf{ifte\ f_1\ f_2\ f_3}) \\ &\quad \equiv \ \eta_{\mathsf{a}^*}\ (\mathsf{ifte\ f_1\ f_2\ f_3})\ \mathsf{j} \end{aligned}$$

Laziness. Starting with the right side of (18), expand definitions, β-expand within the outer thunk, and use the arrow law $\mathsf{arr_a\ f} \ggg_{\mathsf{a}} \mathsf{lazy_a\ f_1} \equiv \mathsf{lazy_a}\ \lambda 0.\, \mathsf{arr_a\ f} \ggg_{\mathsf{a}} \mathsf{f_1}\ 0$:

$$\begin{aligned} &(\mathsf{lazy_{a^*}}\ \lambda 0.\, \eta_{\mathsf{a}^*}\ (\mathsf{f}\ 0))\ \mathsf{j} \\ &\quad \equiv \ \mathsf{lazy_a}\ \lambda 0.\, (\lambda 0.\, \lambda \mathsf{j}.\, \mathsf{arr_a\ snd} \ggg_{\mathsf{a}} \mathsf{f}\ 0)\ 0\ \mathsf{j} \\ &\quad \equiv \ \mathsf{lazy_a}\ \lambda 0.\, \mathsf{arr_a\ snd} \ggg_{\mathsf{a}} \mathsf{f}\ 0 \\ &\quad \equiv \ \mathsf{arr_a\ snd} \ggg_{\mathsf{a}} \mathsf{lazy_a\ f} \\ &\quad \equiv \ \eta_{\mathsf{a}^*}\ (\mathsf{lazy_a\ f})\ \mathsf{j} \end{aligned}$$

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

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

In particular, Corollary 7.4 implies that, if a pure let-calculus expression is interpreted as a computation $\mathsf{k} : \mathsf{AStore\ S}\ (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y})$, then $\mathsf{k\ j_0}$ 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* $\mathsf{R} := \mathsf{J} \to [0,1]$*. A* ***random source*** *is a total mapping* $\mathsf{r} \in \mathsf{R}$*; equivalently, an infinite vector of random numbers indexed by* J.

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

$$\begin{aligned} &\mathsf{random_{a^*}} : \mathsf{x} \rightsquigarrow_{\mathsf{a}^*} [0,1] \\ &\mathsf{random_{a^*}\ j} \ := \ \mathsf{arr_a}\ (\mathsf{fst} \ggg \pi\ \mathsf{j}) \end{aligned} \tag{38}$$

We extend the let-calculus semantic function with

$$[\![\mathsf{random}]\!]_{\mathsf{a}^*} :\equiv \mathsf{random_{a^*}} \tag{39}$$

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

7.4 Partial Programs

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

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

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

$$\begin{aligned} &\mathsf{branch_{a^*}} : \mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{Bool} \\ &\mathsf{branch_{a^*}\ j} \ := \ \mathsf{arr_a}\ (\mathsf{fst} \ggg \pi\ \mathsf{j}) \end{aligned} \tag{40}$$

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

$$\begin{aligned} &\mathsf{agrees} : \langle \mathsf{Bool}, \mathsf{Bool} \rangle \Rightarrow \mathsf{Bool}_\perp \\ &\mathsf{agrees}\ \langle \mathsf{b_1}, \mathsf{b_2} \rangle \ := \ \mathsf{if}\ (\mathsf{b_1} = \mathsf{b_2})\ \mathsf{b_1}\ \perp \end{aligned} \tag{41}$$

$$\begin{aligned} &\mathsf{ifte'_{a^*}} : (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{Bool}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y}) \\ &\mathsf{ifte'_{a^*}\ k_1\ k_2\ k_3} \ := \\ &\quad \mathsf{ifte_{a^*}}\ ((\mathsf{k_1} \ \&\&\&_{\mathsf{a}^*}\ \mathsf{branch_{a^*}}) \ggg_{\mathsf{a}^*} \mathsf{arr_{a^*}\ agrees})\ \mathsf{k_2\ k_3} \end{aligned} \tag{42}$$

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

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

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

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

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

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

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

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

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

overriding only the if rule:

$$\begin{aligned}[\![\mathsf{if}\ e_c\ e_t\ e_f]\!]'_{\mathsf{a}^*} \ &:\equiv \ \mathsf{ifte}'_{\mathsf{a}^*}\ [\![e_c]\!]_{\mathsf{a}^*}\\ &\qquad (\mathsf{lazy}_{\mathsf{a}}\ \lambda 0.\, [\![e_t]\!]_{\mathsf{a}^*})\\ &\qquad (\mathsf{lazy}_{\mathsf{a}}\ \lambda 0.\, [\![e_f]\!]_{\mathsf{a}^*})\\ [\![e]\!]'_{\mathsf{a}^*} \ &:\equiv \ [\![e]\!]_{\mathsf{a}^*}\end{aligned} \tag{43}$$

7.5 Partial, Probabilistic Programs

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

$$\begin{aligned}&\mathsf{random}_{\mathsf{a}^*} : \mathsf{x} \rightsquigarrow_{\mathsf{a}^*} [0,1]\\ &\mathsf{random}_{\mathsf{a}^*}\ \mathsf{j} \ := \ \mathsf{arr}_{\mathsf{a}}\ (\mathsf{fst} \ggg \mathsf{fst} \ggg \pi\ \mathsf{j})\end{aligned} \tag{44}$$

$$\begin{aligned}&\mathsf{branch}_{\mathsf{a}^*} : \mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{Bool}\\ &\mathsf{branch}_{\mathsf{a}^*}\ \mathsf{j} \ := \ \mathsf{arr}_{\mathsf{a}}\ (\mathsf{fst} \ggg \mathsf{snd} \ggg \pi\ \mathsf{j})\end{aligned} \tag{45}$$

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

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

7.6 Correctness

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

$$\begin{aligned}&\mathsf{lift}_{\mathsf{b}^*} : (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{b}^*} \mathsf{y})\\ &\mathsf{lift}_{\mathsf{b}^*}\ \mathsf{f}\ \mathsf{j} \ := \ \mathsf{lift}_{\mathsf{b}}\ (\mathsf{f}\ \mathsf{j})\end{aligned} \tag{46}$$

The following diagram commutes:

$$\begin{tikzcd} \mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y} \arrow[r, "\mathsf{lift}_{\mathsf{b}}"] \arrow[d, "\eta_{\mathsf{a}^*}"'] & \mathsf{x} \rightsquigarrow_{\mathsf{b}} \mathsf{y} \arrow[d, "\eta_{\mathsf{b}^*}"] \\ \mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y} \arrow[r, "\mathsf{lift}_{\mathsf{b}^*}"'] & \mathsf{x} \rightsquigarrow_{\mathsf{b}^*} \mathsf{y} \end{tikzcd} \tag{47}$$

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

Proof. Starting from the right side of the equivalence, expand definitions and apply homomorphism identities (15) and (14) for $\mathsf{lift}_{\mathsf{b}}$:

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

Further, because η_{a^*}, η_{b^*}, and $\mathsf{lift}_{\mathsf{b}}$ 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 ☐

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

$$\begin{tikzcd} \mathsf{X} \rightsquigarrow_{\perp} \mathsf{Y} \arrow[r, "\mathsf{lift}_{\mathsf{map}}"] \arrow[d, "\eta_{\perp^*}"'] & \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y} \arrow[r, "\mathsf{lift}_{\mathsf{pre}}"] \arrow[d, "\eta_{\mathsf{map}^*}"] & \mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y} \arrow[d, "\eta_{\mathsf{pre}^*}"] \\ \mathsf{X} \rightsquigarrow_{\perp^*} \mathsf{Y} \arrow[r, "\mathsf{lift}_{\mathsf{map}^*}"'] & \mathsf{X} \underset{\mathsf{map}^*}{\rightsquigarrow} \mathsf{Y} \arrow[r, "\mathsf{lift}_{\mathsf{pre}^*}"'] & \mathsf{X} \underset{\mathsf{pre}^*}{\rightsquigarrow} \mathsf{Y} \end{tikzcd} \tag{48}$$

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

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

Corollary 7.10 (semantic′ correctness)**.** *If* $[\![e]\!]'_{\perp^*} : \mathsf{X} \rightsquigarrow_{\perp^*} \mathsf{Y}$*, then* $\mathsf{lift}_{\mathsf{map}^*}\ [\![e]\!]'_{\perp^*} \equiv [\![e]\!]'_{\mathsf{map}^*}$ *and* $\mathsf{lift}_{\mathsf{pre}^*}\ [\![e]\!]'_{\mathsf{map}^*} \equiv [\![e]\!]'_{\mathsf{pre}^*}$*.*

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

Theorem 7.11 (divergence implies error)**.** *Let* $\mathsf{f} := [\![e]\!]_{\perp^*}$ *and* $\mathsf{f}' := [\![e]\!]'_{\perp^*}$ *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. Let $\mathsf{m} : \mathsf{J} \Rightarrow \mathsf{J}$ invertibly map sub-computation indexes in f' to corresponding sub-computation indexes in f. (Defining m formally is tedious and unilluminating. XXX: check this)

Case 1. Define $\mathsf{t}' \in \mathsf{J} \to \mathsf{Bool}_{\perp}$ such that $\mathsf{t}'\ \mathsf{j} = \mathsf{z}$ if the sub-computation with index $\mathsf{m}\ \mathsf{j}$ in f is an if condition that 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 $t' \in T$. There exists an infinite *suffix* $J' \subset J$ closed under left and right, such that for all $j' \in J'$, $t'\ j' = \bot$. Because $f\ \langle\langle r, t\rangle, a\rangle$ diverges, the indexes of its if conditions are unbounded; there is therefore a condition with index j such that $m^{-1}\ j \in J'$. It returns $\bot$, so $f'\ \langle\langle r, t'\rangle, a\rangle = \bot$. ☐

To compare preimages computed by arrow instances produced by $[\![\cdot]\!]_{\mathsf{pre}^*}$ and $[\![\cdot]\!]'_{\mathsf{pre}^*}$, 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* $A^* \subseteq (R \times T) \times X$ *for which*

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

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

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

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

Proof. XXX: do this ☐

Corollary 7.14 (computed halting set)**.** *Let* $[\![e]\!]_{\bot^*} : X \rightsquigarrow_{\bot^*} Y$ *converge. Then* $A^* = \mathsf{pre\text{-}ap}\ ([\![e]\!]'_{\mathsf{pre}^*}\ j_0\ (S \times X))\ Y$*.*

Corollary 7.15 (semantic correctness (final))**.** *Let* $[\![e]\!]_{\bot^*} : X \rightsquigarrow_{\bot^*} Y$ *converge, with halting set* A^**. For* $A \subseteq X$ *and* $B \subseteq Y$*,* $\mathsf{pre\text{-}ap}\ ([\![e]\!]'_{\mathsf{pre}^*}\ j_0\ A)\ B = \mathsf{preimage}\ ([\![e]\!]_{\mathsf{map}^*}\ j_0\ (A \cap A^*))\ B$*.*

In other words, preimages computed using $[\![\cdot]\!]'_{\mathsf{pre}^*}$ 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.

Recall that, for a mapping $g : X \rightharpoonup Y$, the probability of an output set $B \subseteq Y$ is

$$P\ (\mathsf{preimage}\ g\ B) \tag{49}$$

where $P \in \mathcal{P}\ X \rightharpoonup [0, 1]$ is a probability measure on X. This was the motivation for defining arrows to compute preimages in the first place. Note again that P is a *partial* function. We had left $\mathsf{domain}\ P$, and which $B \subseteq Y$ have preimages in $\mathsf{domain}\ P$, as technical conditions to be proved later. Now we determine those conditions.

To save space, we assume readers are familiar with either topology or measure theory. (Readers unfamiliar with both may wish to skip to the next section.) Many concepts in measure theory can be understood by analogy to topology.

The analogue of a topology is a σ-algebra.

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

$X \backslash X = \varnothing$, 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* $g : X \rightharpoonup Y$ *is* ***$\mathcal{A}$-$\mathcal{B}$-measurable*** *if for all* $B \in \mathcal{B}$*,* $\mathsf{preimage}\ g\ B \in \mathcal{A}$*.*

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

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

8.1 Measurable Pure Computations

It is easier to prove measurability of pure computations than to prove measurability of probabilistic ones. Further, we can use the resulting theorems to prove that probabilistic computations 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 when applying a computation to its halting set.

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

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

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

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

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

Proof. Use the mapping arrow restriction law (20) 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. We therefore need a case for each arrow combinator.

8.1.1 Case: Composition

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

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

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

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

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

Proof. Use the mapping arrow restriction law (20). ☐

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,

$$(\mathsf{g}_1 \ggg_{\mathsf{map}} \mathsf{g}_2)\ \mathsf{A}^{**} \ = \ \begin{array}{l}\mathsf{let}\ \ \mathsf{g}_1' := \mathsf{g}_1\ \mathsf{A}^{**} \\ \phantom{\mathsf{let}}\ \ \mathsf{g}_2' := \mathsf{g}_2\ (\mathsf{range}\ \mathsf{g}_1') \\ \mathsf{in}\ \ \mathsf{g}_2' \circ_{\mathsf{map}} \mathsf{g}_1'\end{array} \tag{50}$$

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

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

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

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

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

8.1.2 Case: Pairing

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

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

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

8.1.3 Case: Conditional

Lemma 8.17 (measurability under $\uplus_{\mathsf{map}}$)**.** *If* $\mathsf{g}_1 : \mathsf{X} \rightharpoonup \mathsf{Y}$ *and* $\mathsf{g}_2 : \mathsf{X} \rightharpoonup \mathsf{Y}$ *are* $\mathcal{A}$*-*$\mathcal{B}$*-measurable and have disjoint domains,* $\mathsf{g}_1 \uplus_{\mathsf{map}} \mathsf{g}_2$ *is* $\mathcal{A}$*-*$\mathcal{B}$*-measurable.*

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

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

$$\begin{aligned}\mathsf{A}_2^{**} \ &:= \ \mathsf{A}_2^* \cap \mathsf{preimage}\ (\mathsf{g}_1\ \mathcal{A}_1^*)\ \{\mathsf{true}\} \\ \mathsf{A}_3^{**} \ &:= \ \mathsf{A}_3^* \cap \mathsf{preimage}\ (\mathsf{g}_1\ \mathcal{A}_1^*)\ \{\mathsf{false}\} \\ \mathsf{A}^{**} \ &:= \ \mathsf{A}_2^{**} \uplus \mathsf{A}_3^{**}\end{aligned} \tag{51}$$

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

$$\mathsf{ifte}_{\mathsf{map}}\ \mathsf{g}_1\ \mathsf{g}_2\ \mathsf{g}_3\ \mathsf{A}^{**} \ = \ \begin{array}{l}\mathsf{let}\ \ \mathsf{g}_1' := \mathsf{g}_1\ \mathsf{A}^{**} \\ \phantom{\mathsf{let}}\ \ \mathsf{g}_2' := \mathsf{g}_2\ (\mathsf{preimage}\ \mathsf{g}_1'\ \{\mathsf{true}\}) \\ \phantom{\mathsf{let}}\ \ \mathsf{g}_3' := \mathsf{g}_3\ (\mathsf{preimage}\ \mathsf{g}_1'\ \{\mathsf{false}\}) \\ \mathsf{in}\ \ \mathsf{g}_2' \uplus_{\mathsf{map}} \mathsf{g}_3'\end{array} \tag{52}$$

By hypothesis, g_1', g_2' and g_3' are measurable mappings, and the mapping arrow restriction law (20) 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.*

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

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

$$\mathsf{lazy}_{\mathsf{map}}\ \mathsf{g}\ \mathsf{A}^{**} \ = \ \mathsf{if}\ (\mathsf{A}^{**} = \varnothing)\ \varnothing\ (\mathsf{g}\ 0\ \mathsf{A}^{**}) \tag{53}$$

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

8.2 Measurable Probabilistic Computations

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

Definition 8.21 (measurable mapping* arrow computation)**.** *Let* $\mathcal{A}$ *and* $\mathcal{B}$ *be* σ*-algebras on* $(\mathsf{R} \times \mathsf{T}) \times \mathsf{X}$ *and* Y*. A computation* $\mathsf{g} : \mathsf{X} \underset{\mathsf{map^*}}{\rightsquigarrow} \mathsf{Y}$ *is* $\mathcal{A}$*-*$\mathcal{B}$***-measurable*** *if* $\mathsf{g}\ \mathsf{j}_0\ \mathsf{A}^*$ *is an* $\mathcal{A}$*-*$\mathcal{B}$*-measurable mapping, where* A^* *is* g*'s halting set.*

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

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

$$\Sigma\ \langle \mathsf{X}_1, \mathsf{X}_2 \rangle \ = \ \Sigma\ \mathsf{X}_1 \otimes \Sigma\ \mathsf{X}_2 \tag{54}$$

$$\Sigma\ (\mathsf{J} \to \mathsf{X}) \ = \ (\Sigma\ \mathsf{X})^{\otimes \mathsf{J}} \tag{55}$$

$$|\mathsf{X}| \leq \omega \implies \Sigma\ \mathsf{X} \ = \ \mathcal{P}\ \mathsf{X} \tag{56}$$

$$\mathsf{X}' \in \Sigma\ \mathsf{X} \implies \Sigma\ \mathsf{X}' \ = \ \{\mathsf{X}' \cap \mathsf{A} \mid \mathsf{A} \in \Sigma\ \mathsf{X}\} \tag{57}$$

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

Examples: by (56), $\Sigma\ \mathsf{Bool} = \mathcal{P}\ \mathsf{Bool}$. By both (54) and (56), $\Sigma\ \langle \mathsf{Bool}, \mathsf{Bool} \rangle = \mathcal{P}\ (\mathsf{Bool} \times \mathsf{Bool})$. By (57), $\Sigma\ [0, 1]$ is a sub-σ-algebra of $\Sigma\ \mathbb{R}$.

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

Proof. Follows from (54) and Definition 8.3. ☐

Theorem 8.24 (AStore measurability transfer)**.** *Every* AStore *arrow combinator produces measurable mapping* computations from measurable mapping* computations.*

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

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

Proof. Follows from (55) and Definition 8.4. □

Corollary 8.26. $\mathsf{random}_{\mathsf{map}^*}$ *and* $\mathsf{branch}_{\mathsf{map}^*}$ *are measurable.*

Theorem 8.27. $\mathsf{ifte}'_{\mathsf{map}^*}$ *is measurable.*

Proof. $\mathsf{branch}_{\mathsf{map}^*}$ is measurable, and $\mathsf{arr}_{\mathsf{map}^*}$ agrees is measurable by (56). □

Theorem 8.28 (probabilistic programs are measurable)**.** *If* $[\![e]\!]_{\mathsf{map}^*}$ *converges, it is measurable. If* $[\![e]\!]'_{\mathsf{map}^*}$ *converges, it is measurable.*

Proof. By structural induction over e and the above measurability theorems. □

Of course, Theorem 8.28 remains true when $[\![\cdot]\!]_{\mathsf{a}}$ is extended with any rule whose right side is measurable. Examples include arithmetic, equalities and inequalities, and lambda expressions transformed into closures.

8.3 Random Store Probabilities

Preimages under probabilistic programs are measurable subsets of $(\mathsf{R} \times \mathsf{T}) \times \mathsf{X}$. While it is possible to put probability measures on such domains, doing so would be surprising for end-users. For example, the probabilities of outputs of the geometric function defined in (34) would depend not only on the probability of random $< \mathsf{p}$, but also on some arbitrary probability that each branch is taken. It would not define the geometric distribution.

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

As previously, we start with measuring the halting set.

Definition 8.29 (standard probability measure)**.** *For a type* X*, a* ***standard probability measure*** *is a probability measure* $\mathsf{P} \in \mathcal{P}\ \mathsf{X} \rightharpoonup [0,1]$ *where* $\mathsf{domain}\ \mathsf{P} = \Sigma\ \mathsf{X}$.

Definition 8.30 (halting probability)**.** *Let* $\mathsf{g} : \mathsf{X} \underset{\mathsf{map}^*}{\rightsquigarrow} \mathsf{Y}$ *be measurable, with* 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* $\mathsf{P}\ (\mathsf{image}\ (\mathsf{fst} \ggg \mathsf{fst})\ \mathsf{A}^*)$.

Theorem 8.31 (measurable finite projections)**.** *Let* $\mathsf{A} \in \Sigma\ \langle \mathsf{X}_1, \mathsf{X}_2 \rangle$*. If* 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 8.32. *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 8.31 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 8.33. *Let* $\mathsf{g} : \mathsf{X} \underset{\mathsf{map}^*}{\rightsquigarrow} \mathsf{Y}$ *be measurable and* $\mathsf{A} \subseteq \mathsf{A}^*$ *a measurable set.* $\mathsf{P}\ (\mathsf{image}\ (\mathsf{fst} \ggg \mathsf{fst})\ \mathsf{A})$*, the probability of* A*, is well-defined.*

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

9. Implementable Approximation

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

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

Figure 4:

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

Figure 6:

- $\mathsf{arr}_{\mathsf{pre}}$ (and $\mathsf{lift}_{\mathsf{pre}}$): can't implement
- $\ggg_{\mathsf{pre}}$: no change
- $\&\&\&_{\mathsf{pre}}$: use approximating $\langle \cdot, \cdot \rangle_{\mathsf{pre}}$
- $\mathsf{ifte}_{\mathsf{pre}}$: need {true} and {false}; use approximating $\uplus_{\mathsf{pre}}$
- $\mathsf{lazy}_{\mathsf{pre}}$: need $(= \varnothing)$, $(\mathsf{pre}\ \varnothing)$

Figure 8:

- $\mathsf{id}_{\mathsf{pre}}$: no change
- $\mathsf{const}_{\mathsf{pre}}$: need $\{\mathsf{y}\}$, $(= \varnothing)$, $\varnothing$
- $\mathsf{fst}_{\mathsf{pre}}$ and $\mathsf{snd}_{\mathsf{pre}}$: need projections, i, $\times$
- π_{pre}: need projections, $\cap$, arbitrary products

10. Computable Approximation

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.

$$\mathsf{id}_{\mathsf{pre}} : \mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{X}$$
$$\mathsf{id}_{\mathsf{pre}}\ \mathsf{A} \ := \ \langle \mathsf{A}, \lambda \mathsf{B}.\, \mathsf{B} \rangle$$

$$\mathsf{const}_{\mathsf{pre}} : \mathsf{Y} \Rightarrow \mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}$$
$$\mathsf{const}_{\mathsf{pre}}\ \mathsf{y}\ \mathsf{A} \ := \ \langle \{\mathsf{y}\}, \lambda \mathsf{B}.\, \mathsf{if}\ (\mathsf{B} = \varnothing)\ \varnothing\ \mathsf{A} \rangle$$

$$\pi_{\mathsf{pre}} : \mathsf{J} \Rightarrow (\mathsf{J} \to \mathsf{X}) \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{X}$$
$$\begin{aligned} \pi_{\mathsf{pre}}\ \mathsf{j}\ \mathsf{A} \ := \ \mathsf{let}\ \ & \mathsf{A_j} := \mathsf{proj}\ \mathsf{j}\ \mathsf{A} \\ & \mathsf{p} := \lambda \mathsf{B}.\, \mathsf{A} \cap \textstyle\prod_{\mathsf{i} \in \mathsf{J}} \mathsf{if}\ (\mathsf{j} = \mathsf{i})\ \mathsf{B}\ (\mathsf{proj}\ \mathsf{i}\ \mathsf{A}) \\ \mathsf{in}\ \ & \langle \mathsf{A_j}, \mathsf{p} \rangle \end{aligned}$$

$$\mathsf{fst}_{\mathsf{pre}} : \langle \mathsf{X}, \mathsf{Y} \rangle \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{X}$$
$$\begin{aligned} \mathsf{fst}_{\mathsf{pre}}\ \mathsf{A} \ := \ \mathsf{let}\ \ & \mathsf{A_1} := \mathsf{image}\ \mathsf{fst}\ \mathsf{A} \\ & \mathsf{A_2} := \mathsf{image}\ \mathsf{snd}\ \mathsf{A} \\ \mathsf{in}\ \ & \langle \mathsf{A_1}, \lambda \mathsf{B}.\, \mathsf{A} \cap (\mathsf{B} \times \mathsf{A_2}) \rangle \end{aligned}$$

$$\mathsf{snd}_{\mathsf{pre}} : \langle \mathsf{X}, \mathsf{Y} \rangle \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}$$
$$\begin{aligned} \mathsf{snd}_{\mathsf{pre}}\ \mathsf{A} \ := \ \mathsf{let}\ \ & \mathsf{A_1} := \mathsf{image}\ \mathsf{fst}\ \mathsf{A} \\ & \mathsf{A_2} := \mathsf{image}\ \mathsf{snd}\ \mathsf{A} \\ \mathsf{in}\ \ & \langle \mathsf{A_2}, \lambda \mathsf{B}.\, \mathsf{A} \cap (\mathsf{A_1} \times \mathsf{B}) \rangle \end{aligned}$$

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