Running Probabilistic Programs Backward

Neil Toronto Jay McCarthy

PLT @ Brigham Young University

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

Abstract

XXX

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

General Terms XXX, XXX

Keywords XXX, XXX

1. Introduction

XXX: what we want to do, a bit about measure-theory's take on probability (especially preimage measure)

XXX: the paper starts with an overview of the insanely powerful λ-calculus we use, and an overview of arrows, the categorical construct we use to define the meaning of let-calculus expressions compositionally

For the bulk of this paper, we

1. Define the *bottom arrow*, type $\mathsf{X} \rightsquigarrow_{\bot} \mathsf{Y}$, 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. Extend the preimage arrow to handle partial functions.
5. Approximate the preimage arrow by operating on rectangles instead of generally uncomputable sets.

We report on our implementation of the bottom arrow and the approximating preimage arrow.

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

Formal goals:

1. Preimages of pure, total functions computed using $\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow}{}' \mathsf{Y}$ are the preimages under $\mathsf{X} \rightsquigarrow_{\bot} \mathsf{Y}$ (have this, by homomorphism)
2. Preimages computed using $\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow}{}' \mathsf{Y}$ are the preimages under $\mathsf{X} \rightsquigarrow_{\bot}{}' \mathsf{Y}$ (need to prove $\mathsf{lift}'_{\mathsf{map}}$ and $\mathsf{lift}'_{\mathsf{pre}}$ are homomorphisms)
3. Running $\mathsf{X} \rightsquigarrow_{\bot}{}' \mathsf{Y}$ and $\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow}{}' \mathsf{Y}$ computations always converges

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

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).

[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 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 $\mathsf{X} \times \mathsf{Y}$.

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

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

Shortly, $\varnothing$ is the empty set, ω is the cardinality of the natural numbers, take removes the member from a singleton set, $(\in)$ is an infix operator that decides membership, $\mathcal{P}$ returns all the subsets of a set, $\bigcup$ returns the union of a set of sets, image applies a function to each member of a set and returns the set of return values, and card returns the cardinality of a set.

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

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 $(\lambda \mathsf{x}.\, \mathsf{x})\ 1 = 1$.

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

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 2.1.1 (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 could be 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. We therefore introduce type-specific notions of equivalence as needed.

2.2 Additional Functions and Forms

XXX: lambda syntactic sugar: automatic currying (including the two-argument primitives $(\in)$ and image), matching, sectioning rules

XXX: set syntactic sugar: set comprehensions, cardinality, indexed unions

XXX: functions: $\cup$, $\cap$, $\backslash$, $\subseteq$

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

XXX: logic: logical operators and quantifiers

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 hash tables from lambdas, we call them ***mappings***, and use the word **function** for either a lambda or a mapping. For convenience, as with lambdas, we use adjacency (i.e. $(\mathsf{f}\ \mathsf{x})$) to apply mappings.

The set $\mathsf{X} \rightharpoonup \mathsf{Y}$ contains all the *partial* mappings from X to Y. For example, $\mathsf{X} \rightharpoonup \mathsf{Y}$ is the return type for the restriction function:

$$\begin{aligned} &(\cdot)|_{(\cdot)} : (\mathsf{X} \Rightarrow \mathsf{Y}) \Rightarrow \mathsf{Set}\ \mathsf{X} \Rightarrow (\mathsf{X} \rightharpoonup \mathsf{Y}) \\ &\mathsf{f}|_{\mathsf{A}}\ :=\ \mathsf{image}\ (\lambda \mathsf{x}.\, \langle \mathsf{x}, \mathsf{f}\ \mathsf{x} \rangle)\ \mathsf{A} \end{aligned} \tag{6}$$

which converts a lambda or a mapping to a mapping with domain $\mathsf{A} \subseteq \mathsf{X}$. To create mappings using lambda syntax, we define $\lambda x \in e_A.\, e$ as shorthand for $(\lambda x.\, e)|_{e_A}$.

Figure 1 defines more operations on partial mappings: domain, range, $\mathsf{preimage}$, 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.

XXX: lazy mappings

XXX: total mappings are infinite vectors

XXX: projection functions

$$\begin{aligned} &\pi : \mathsf{J} \Rightarrow (\mathsf{J} \to \mathsf{X}) \Rightarrow \mathsf{X} \\ &\pi\ \mathsf{j}\ \mathsf{f}\ :=\ \mathsf{f}\ \mathsf{j} \end{aligned} \tag{7}$$

$$\begin{aligned} &\mathsf{proj} : \mathsf{J} \Rightarrow (\mathsf{J} \to \mathsf{X}) \Rightarrow \mathsf{Set}\ \mathsf{X} \\ &\mathsf{proj}\ \mathsf{j}\ \mathsf{A}\ :=\ \mathsf{image}\ (\pi\ \mathsf{j})\ \mathsf{A} \end{aligned} \tag{8}$$

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 $\mathsf{first}_{\mathsf{a}}$, we define $(\&\&\&_{\mathsf{a}})$—typically

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

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

Figure 1: Operations on mappings.

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, which are in terms of disjoint unions of mappings instead of explicit disjoint union types.

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

$$\begin{aligned}
\mathsf{arr\ f}\ &:= \ \mathsf{f} \\
(\mathsf{f_1} \ggg \mathsf{f_2})\ \mathsf{a}\ &:= \ \mathsf{f_2\ (f_1\ a)} \\
(\mathsf{f_1}\ \&\&\&\ \mathsf{f_2})\ \mathsf{a}\ &:= \ \langle \mathsf{f_1\ a}, \mathsf{f_2}, \mathsf{a}\rangle \\
\mathsf{ifte\ f_1\ f_2\ f_3\ a}\ &:= \ \mathsf{if\ (f_1\ a)\ (f_2\ a)\ (f_3\ a)}
\end{aligned} \tag{9}$$

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

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

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

Definition 3.1.1 (arrow+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+choice*** *if certain additional homomorphism and structural laws hold. (The* $\mathsf{lazy_a}$ *combinator may be omitted in nonstrict or strongly normalizing languages.)*

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 3.1.2 (arrow+choice 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, $\mathsf{f_1}$ *and* $\mathsf{f_2}$*:*

$$\mathsf{lift_b\ (arr_a\ f)}\ \equiv\ \mathsf{arr_b\ f} \tag{14}$$

$$\mathsf{lift_b}\ (\mathsf{f_1} \ggg_\mathsf{a} \mathsf{f_2})\ \equiv\ (\mathsf{lift_b\ f_1}) \ggg_\mathsf{b} (\mathsf{lift_b\ f_2}) \tag{15}$$

$$\mathsf{lift_b}\ (\mathsf{f_1}\ \&\&\&_\mathsf{a}\ \mathsf{f_2})\ \equiv\ (\mathsf{lift_b\ f_1})\ \&\&\&_\mathsf{b}\ (\mathsf{lift_b\ f_2}) \tag{16}$$

It is an ***arrow+choice homomorphism*** *if, additionally,*

$$\mathsf{lift_b\ (ifte_a\ f_1\ f_2\ f_3)}\ \equiv\ \mathsf{ifte_b\ (lift_b\ f_1)\ (lift_b\ f_2)\ (lift_b\ f_3)} \tag{17}$$

$$\mathsf{lift_b\ (lazy_a\ f)}\ \equiv\ \mathsf{lazy_b}\ \lambda 0.\, \mathsf{lift_b\ (f\ 0)} \tag{18}$$

hold for appropriately typed f, $\mathsf{f_1}$, $\mathsf{f_2}$ *and* $\mathsf{f_3}$.

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

3.2 First-Order Let-Calculus Semantics

XXX: Figure 2...

XXX: Stack machine...

XXX: Roughly, first-order application (x e) runs arrow computation x with a fresh stack with e at the head. The binding form (let e_0 e_b) pushes e_0 onto the stack. Variables are referenced using (env n) with (env 0) referring to the head.

$$\begin{aligned}
[\![x := e;\ \cdots]\!]_{\mathsf{a}} &:\equiv x := [\![e]\!]_{\mathsf{a}}\,;\ \cdots & [\![v]\!]_{\mathsf{a}} &:\equiv \mathsf{arr_a}\ \lambda\gamma.\,v\\
[\![x\ e]\!]_{\mathsf{a}} &:\equiv [\![\langle e, 0\rangle]\!]_{\mathsf{a}} \ggg_{\mathsf{a}} x & [\![\mathsf{fst}\ e]\!]_{\mathsf{a}} &:\equiv [\![e]\!]_{\mathsf{a}} \ggg_{\mathsf{a}} (\mathsf{arr_a}\ \mathsf{fst})\\
[\![\langle e_1, e_2\rangle]\!]_{\mathsf{a}} &:\equiv [\![e_1]\!]_{\mathsf{a}}\ \&\&\&_{\mathsf{a}}\ [\![e_2]\!]_{\mathsf{a}} & [\![\mathsf{snd}\ e]\!]_{\mathsf{a}} &:\equiv [\![e]\!]_{\mathsf{a}} \ggg_{\mathsf{a}} (\mathsf{arr_a}\ \mathsf{snd})\\
[\![\mathsf{let}\ e_0\ e_b]\!]_{\mathsf{a}} &:\equiv ([\![e_0]\!]_{\mathsf{a}}\ \&\&\&_{\mathsf{a}}\ (\mathsf{arr_a}\ \mathsf{id})) \ggg_{\mathsf{a}} [\![e_b]\!]_{\mathsf{a}} & [\![e_1 + e_2]\!]_{\mathsf{a}} &:\equiv [\![\langle e_1, e_2\rangle]\!]_{\mathsf{a}} \ggg_{\mathsf{a}} (\mathsf{arr_a}\ \lambda\langle \mathsf{x},\mathsf{y}\rangle.\,\mathsf{x}+\mathsf{y})\\
[\![\mathsf{env}\ n]\!]_{\mathsf{a}} &:\equiv \mathsf{arr_a}\ \lambda\gamma.\,\mathsf{list\text{-}ref}\ \gamma\ n & [\![e_1 < e_2]\!]_{\mathsf{a}} &:\equiv [\![\langle e_1, e_2\rangle]\!]_{\mathsf{a}} \ggg_{\mathsf{a}} (\mathsf{arr_a}\ \lambda\langle \mathsf{x},\mathsf{y}\rangle.\,\mathsf{x}<\mathsf{y})\\
[\![\mathsf{if}\ e_c\ e_t\ e_f]\!]_{\mathsf{a}} &:\equiv \mathsf{ifte_a}\ [\![e_c]\!]_{\mathsf{a}}\ (\mathsf{lazy_a}\ \lambda 0.\,[\![e_t]\!]_{\mathsf{a}})\ (\mathsf{lazy_a}\ \lambda 0.\,[\![e_f]\!]_{\mathsf{a}}) & &\cdots
\end{aligned}$$

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

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, it will be necessary to explicitly handle nonterminating functions, 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 4.0.1. *$\mathsf{arr}_\perp$, $(\&\&\&_\perp)$ and $(\ggg_\perp)$ define an arrow. With $\mathsf{ifte}_\perp$ and $\mathsf{lazy}_\perp$, they define an arrow+choice.*

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

4.1 Deriving the Mapping Arrow

Theorems in measure theory tend to be about mappings, not lambdas. As in intermediate step toward the preimage arrow, then, we need an arrow whose computations produce mappings or are mappings themselves.

It is tempting to try to define the mapping arrow's type constructor using $\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y} ::= \mathsf{X} \rightharpoonup \mathsf{Y}$, and define $\mathsf{f}_1 \ggg_{\mathsf{map}} \mathsf{f}_2 := \mathsf{f}_2 \circ_{\mathsf{map}} \mathsf{f}_1$ and $(\&\&\&_{\mathsf{map}}) := \langle\cdot,\cdot\rangle_{\mathsf{map}}$. Unfortunately, we run into a problem defining $\mathsf{arr_{map}} : (\mathsf{X} \Rightarrow \mathsf{Y}) \Rightarrow (\mathsf{X} \rightharpoonup \mathsf{Y})$: we cannot define it as $\mathsf{arr_{map}}\ \mathsf{f} := \mathsf{f}|_{\mathsf{X}}$. Although X is a λ_{ZFC} value, it is not in scope in $\mathsf{arr_{map}}$'s definition because it is only part of the type.

We need to parameterize computations on a domain, so 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 disregard nonterminating inputs when computing preimages further on. (XXX: section)

To make the correspondence between bottom arrow and mapping arrow computations as clear as possible, we start by defining a function $\mathsf{lift_{map}} : (\mathsf{X} \rightsquigarrow_\perp \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y})$ to lift bottom arrow computations. It must restrict its argument f's domain to a subset of X for which f does not return $\perp$. It is helpful to have a standalone function $\mathsf{domain}_\perp$ that computes such domains, so we define that first, and then define $\mathsf{lift_{map}}$ in terms of it:

$$\begin{aligned}
&\mathsf{domain}_\perp : (\mathsf{X} \rightsquigarrow_\perp \mathsf{Y}) \Rightarrow \mathsf{Set}\ \mathsf{X} \Rightarrow \mathsf{Set}\ \mathsf{X}\\
&\mathsf{domain}_\perp\ \mathsf{f}\ \mathsf{A} := \mathsf{preimage}\ \mathsf{f}|_{\mathsf{A}}\ ((\mathsf{image}\ \mathsf{f}\ \mathsf{A})\backslash\{\perp\})
\end{aligned} \tag{20}$$

$$\begin{aligned}
&\mathsf{lift_{map}} : (\mathsf{X} \rightsquigarrow_\perp \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y})\\
&\mathsf{lift_{map}}\ \mathsf{f}\ \mathsf{A} := \mathsf{let}\ \ \mathsf{A}' := \mathsf{domain}_\perp\ \mathsf{f}\ \mathsf{A}\\
&\qquad\qquad\qquad\ \mathsf{in}\ \ \mathsf{f}|_{\mathsf{A}'}
\end{aligned} \tag{21}$$

The clearest way to ensure that mapping arrow computations mean what we think they mean is to derive each combinator in a way that makes $\mathsf{lift_{map}}$ distribute over bottom arrow computations; i.e. it must be an arrow+choice homomorphism (Definition 3.1.2). Less generally but more concretely, for any let-calculus expression e, we would like $[\![e]\!]_{\mathsf{map}} \equiv \mathsf{lift_{map}}\ [\![e]\!]_\perp$.

To meet the homomorphism laws, we need equivalence to be more extensional for mapping computations.

Definition 4.1.1 (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+choice homomorphism. Figure 4 contains the resulting definitions.

4.2 Case: Pairing

Starting with the left side of (16), we first expand definitions. For any $\mathsf{f}_1 : \mathsf{X} \rightsquigarrow_\perp \mathsf{Y}$, $\mathsf{f}_2 : \mathsf{X} \rightsquigarrow_\perp \mathsf{Z}$, and $\mathsf{A} \subseteq \mathsf{X}$,

$$\begin{aligned}
&\mathsf{lift_{map}}\ (\mathsf{f}_1\ \&\&\&_\perp\ \mathsf{f}_2)\ \mathsf{A}\\
&\quad\equiv \mathsf{lift_{map}}\ (\lambda \mathsf{x}.\,\mathsf{if}\ (\mathsf{f}_1\ \mathsf{x} = \perp\ \mathsf{or}\ \mathsf{f}_2\ \mathsf{x} = \perp)\ \perp\ \langle \mathsf{f}_1\ \mathsf{x}, \mathsf{f}_2\ \mathsf{x}\rangle)\ \mathsf{A}\\
&\quad\equiv \mathsf{let}\ \ \ \mathsf{f} := \lambda \mathsf{x}.\,\mathsf{if}\ (\mathsf{f}_1\ \mathsf{x} = \perp\ \mathsf{or}\ \mathsf{f}_2\ \mathsf{x} = \perp)\ \perp\ \langle \mathsf{f}_1\ \mathsf{x}, \mathsf{f}_2\ \mathsf{x}\rangle\\
&\qquad\qquad\ \mathsf{A}' := \mathsf{domain}_\perp\ \mathsf{f}\ \mathsf{A}\\
&\qquad\ \ \mathsf{in}\ \ \mathsf{f}|_{\mathsf{A}'}
\end{aligned} \tag{22}$$

$$x \rightsquigarrow_\perp y \ ::= \ x \Rightarrow y_\perp$$

$$\mathsf{arr}_\perp : (x \Rightarrow y) \Rightarrow (x \rightsquigarrow_\perp y)$$
$$\mathsf{arr}_\perp\ f \ := \ f$$

$$(\ggg_\perp) : (x \rightsquigarrow_\perp y) \Rightarrow (y \rightsquigarrow_\perp z) \Rightarrow (x \rightsquigarrow_\perp z)$$
$$(f_1 \ggg_\perp f_2)\ x \ := \ \mathsf{if}\ (f_1\ x = \perp)\ \perp\ (f_2\ (f_1\ x))$$

$$(\&\&\&_\perp) : (x \rightsquigarrow_\perp y_1) \Rightarrow (x \rightsquigarrow_\perp y_2) \Rightarrow (x \rightsquigarrow_\perp \langle y_1, y_2\rangle)$$
$$(f_1 \&\&\&_\perp f_2)\ x \ := \ \mathsf{if}\ (f_1\ x = \perp\ \mathsf{or}\ f_2\ x = \perp)\ \perp\ \langle f_1\ x, f_2\ x\rangle$$

$$\mathsf{ifte}_\perp : (x \rightsquigarrow_\perp \mathsf{Bool}) \Rightarrow (x \rightsquigarrow_\perp y) \Rightarrow (x \rightsquigarrow_\perp y) \Rightarrow (x \rightsquigarrow_\perp y)$$
$$\begin{aligned}\mathsf{ifte}_\perp\ f_1\ f_2\ f_3\ x \ := \ & \mathsf{case}\ \ f_1\ x \\ & \quad \mathsf{true} \implies f_2\ x \\ & \quad \mathsf{false} \implies f_3\ x \\ & \quad \mathsf{else} \implies \perp\end{aligned}$$

$$\mathsf{lazy}_\perp : (1 \Rightarrow (x \rightsquigarrow_\perp y)) \Rightarrow (x \rightsquigarrow_\perp y)$$
$$\mathsf{lazy}_\perp\ f\ x \ := \ f\ 0\ x$$

Figure 3: Bottom arrow definitions.

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

$$\mathsf{arr}_{\mathsf{map}} : (X \Rightarrow Y) \Rightarrow (X \underset{\mathsf{map}}{\rightsquigarrow} Y)$$
$$\mathsf{arr}_{\mathsf{map}} \ := \ \mathsf{lift}_{\mathsf{map}} \circ \mathsf{arr}_\perp$$

$$(\ggg_{\mathsf{map}}) : (X \underset{\mathsf{map}}{\rightsquigarrow} Y) \Rightarrow (Y \underset{\mathsf{map}}{\rightsquigarrow} Z) \Rightarrow (X \underset{\mathsf{map}}{\rightsquigarrow} Z)$$
$$\begin{aligned}(g_1 \ggg_{\mathsf{map}} g_2)\ A \ := \ & \mathsf{let}\ \ g_1' := g_1\ A \\ & \qquad g_2' := g_2\ (\mathsf{range}\ g_1') \\ & \mathsf{in}\ \ g_2' \circ_{\mathsf{map}} g_1'\end{aligned}$$

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

$$\mathsf{ifte}_{\mathsf{map}} : (X \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Bool}) \Rightarrow (X \underset{\mathsf{map}}{\rightsquigarrow} Y) \Rightarrow (X \underset{\mathsf{map}}{\rightsquigarrow} Y) \Rightarrow (X \underset{\mathsf{map}}{\rightsquigarrow} Y)$$
$$\begin{aligned}\mathsf{ifte}_{\mathsf{map}}\ g_1\ g_2\ g_3\ A \ := \ & \mathsf{let}\ \ g_1' := g_1\ A \\ & \qquad g_2' := g_2\ (\mathsf{preimage}\ g_1'\ \{\mathsf{true}\}) \\ & \qquad g_3' := g_3\ (\mathsf{preimage}\ g_1'\ \{\mathsf{false}\}) \\ & \mathsf{in}\ \ g_2' \uplus_{\mathsf{map}} g_3'\end{aligned}$$

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

$$\mathsf{lift}_{\mathsf{map}} : (X \rightsquigarrow_\perp Y) \Rightarrow (X \underset{\mathsf{map}}{\rightsquigarrow} Y)$$
$$\mathsf{lift}_{\mathsf{map}}\ f\ A \ := \ \{\langle x, y\rangle \in f|_A \mid y \neq \perp\}$$

Figure 4: Mapping arrow definitions.

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

$$\begin{aligned}& \mathsf{lift}_{\mathsf{map}}\ (f_1 \&\&\&_\perp f_2)\ A \\ & \quad \equiv \ \mathsf{let}\ \ A_1 := (\mathsf{domain}_\perp\ f_1\ A) \\ & \qquad\qquad A_2 := (\mathsf{domain}_\perp\ f_2\ A) \\ & \qquad\qquad A' := A_1 \cap A_2 \\ & \qquad\quad \mathsf{in}\ \ \lambda x \in A'.\ \langle f_1\ x, f_2\ x\rangle \\ & \quad \equiv \ \mathsf{let}\ \ g_1 := \mathsf{lift}_{\mathsf{map}}\ f_1\ A \\ & \qquad\qquad g_2 := \mathsf{lift}_{\mathsf{map}}\ f_2\ A \\ & \qquad\qquad A' := (\mathsf{domain}\ g_1) \cap (\mathsf{domain}\ g_2) \\ & \qquad\quad \mathsf{in}\ \ \lambda x \in A'.\ \langle g_1\ x, g_2\ x\rangle \\ & \quad \equiv \ \langle \mathsf{lift}_{\mathsf{map}}\ f_1\ A, \mathsf{lift}_{\mathsf{map}}\ f_2\ A\rangle_{\mathsf{map}}\end{aligned} \tag{23}$$

Substituting g_1 for $\mathsf{lift}_{\mathsf{map}}\ f_1$ and g_2 for $\mathsf{lift}_{\mathsf{map}}\ f_2$ gives a definition for $(\&\&\&_{\mathsf{map}})$ (Figure 4) for which (16) holds.

4.3 Case: Composition

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

XXX: include it?

4.4 Case: Conditional

Starting with the left side of (17), we expand definitions, and simplify f by restricting it to a domain for which $f_1\ x$ cannot be $\perp$:

$$\begin{aligned}& \mathsf{lift}_{\mathsf{map}}\ (\mathsf{ifte}_\perp\ f_1\ f_2\ f_3)\ A \\ & \quad \equiv \ \mathsf{let}\ \ \ f := \lambda x.\ \mathsf{case}\ \ f_1\ x \\ & \qquad\qquad\qquad\qquad \mathsf{true} \implies f_2\ x \\ & \qquad\qquad\qquad\qquad \mathsf{false} \implies f_3\ x \\ & \qquad\qquad\qquad\qquad \mathsf{else} \implies \perp \\ & \qquad\qquad A' := \mathsf{domain}_\perp\ f\ A \\ & \qquad\quad \mathsf{in}\ \ f|_{A'} \\ & \quad \equiv \ \mathsf{let}\ \ A_2 := \mathsf{preimage}\ f_1|_A\ \{\mathsf{true}\} \\ & \qquad\qquad A_3 := \mathsf{preimage}\ f_1|_A\ \{\mathsf{false}\} \\ & \qquad\qquad\ \ f := \lambda x.\ \mathsf{if}\ (f_1\ x)\ (f_2\ x)\ (f_3\ x) \\ & \qquad\qquad A' := \mathsf{domain}_\perp\ f\ (A_2 \uplus A_3) \\ & \qquad\quad \mathsf{in}\ \ f|_{A'}\end{aligned} \tag{24}$$

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

$$\begin{aligned}& \mathsf{lift}_{\mathsf{map}}\ (\mathsf{ifte}_\perp\ f_1\ f_2\ f_3)\ A \\ & \quad \equiv \ \mathsf{let}\ \ g_1 := \mathsf{lift}_{\mathsf{map}}\ f_1\ A \\ & \qquad\qquad g_2 := \mathsf{lift}_{\mathsf{map}}\ f_2\ (\mathsf{preimage}\ g_1\ \{\mathsf{true}\}) \\ & \qquad\qquad g_3 := \mathsf{lift}_{\mathsf{map}}\ f_3\ (\mathsf{preimage}\ g_1\ \{\mathsf{false}\}) \\ & \qquad\qquad A' := (\mathsf{domain}\ g_2) \uplus (\mathsf{domain}\ g_3) \\ & \qquad\quad \mathsf{in}\ \ \lambda x \in A'.\ \mathsf{if}\ (x \in \mathsf{domain}\ g_2)\ (g_2\ x)\ (g_3\ x) \\ & \quad \equiv \ \mathsf{let}\ \ g_1 := \mathsf{lift}_{\mathsf{map}}\ f_1\ A \\ & \qquad\qquad g_2 := \mathsf{lift}_{\mathsf{map}}\ f_2\ (\mathsf{preimage}\ g_1\ \{\mathsf{true}\}) \\ & \qquad\qquad g_3 := \mathsf{lift}_{\mathsf{map}}\ f_3\ (\mathsf{preimage}\ g_1\ \{\mathsf{false}\}) \\ & \qquad\quad \mathsf{in}\ \ g_2 \uplus_{\mathsf{map}} g_3\end{aligned} \tag{25}$$

Substituting g_1 for $\mathsf{lift}_{\mathsf{map}}\ f_1$, g_2 for $\mathsf{lift}_{\mathsf{map}}\ f_2$, and g_3 for $\mathsf{lift}_{\mathsf{map}}\ f_3$ gives a definition for $\mathsf{ifte}_{\mathsf{map}}$ (Figure 4) for which (17) holds.

4.5 Case: Laziness

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

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

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

$$\begin{aligned}&\mathsf{lift}_{\mathsf{map}}\ (\mathsf{lazy}_\perp\ f)\ A\\ &\quad\equiv\ \mathsf{if}\ (A = \varnothing)\ \varnothing\ \mathsf{let}\ \ A' := \mathsf{domain}_\perp\ (f\ 0)\ A\\ &\qquad\qquad\qquad\qquad\quad \mathsf{in}\ \ (f\ 0)|_{A'}\\ &\quad\equiv\ \mathsf{if}\ (A = \varnothing)\ \varnothing\ (\mathsf{lift}_{\mathsf{map}}\ (f\ 0)\ A)\end{aligned}$$

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

4.6 Correctness

Theorem 4.6.1 (mapping arrow correctness)**.** $\mathsf{lift}_{\mathsf{map}}$ *is an arrow+choice homomorphism.*

Proof. By construction. □

Corollary 4.6.2 (semantic correctness)**.** *If* $[\![e]\!]_\perp : X \rightsquigarrow_\perp Y$, *then* $\mathsf{lift}_{\mathsf{map}}\ [\![e]\!]_\perp \equiv [\![e]\!]_{\mathsf{map}}$ *and* $[\![e]\!]_{\mathsf{map}} : X \underset{\mathsf{map}}{\rightsquigarrow} Y$.

5. Lazy Preimage Mappings

On a computer, we will 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 ***lazy preimage mappings***, or just *preimage mappings*, for which application is like applying $\mathsf{preimage}$ to a mapping. Further on, we will need their ranges to be observable, so we define their type as

$$X \underset{\mathsf{pre}}{\rightharpoonup} Y\ \ ::=\ \ \langle \mathsf{Set}\ Y, \mathsf{Set}\ Y \Rightarrow \mathsf{Set}\ X\rangle \tag{26}$$

Converting a mapping to a lazy preimage mapping:

$$\begin{aligned}&\mathsf{pre} : (X \rightharpoonup Y) \Rightarrow (X \underset{\mathsf{pre}}{\rightharpoonup} Y)\\ &\mathsf{pre}\ g\ :=\ \mathsf{let}\ \ Y' := \mathsf{range}\ g\\ &\qquad\qquad\qquad p := \lambda B.\, \mathsf{preimage}\ g\ B\\ &\qquad\qquad\ \ \mathsf{in}\ \ \langle Y', p\rangle\end{aligned} \tag{27}$$

Applying a preimage mapping to any subset of its codomain:

$$\begin{aligned}&\mathsf{pre\text{-}ap} : (X \underset{\mathsf{pre}}{\rightharpoonup} Y) \Rightarrow \mathsf{Set}\ Y \Rightarrow \mathsf{Set}\ X\\ &\mathsf{pre\text{-}ap}\ \langle Y', p\rangle\ B\ :=\ p\ (B \cap Y')\end{aligned} \tag{28}$$

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

Theorem 5.0.4 ($\mathsf{pre\text{-}ap}$ computes preimages)**.** *Let* $g \in X \rightharpoonup Y$. *For all* $B \subseteq Y$, $\mathsf{pre\text{-}ap}\ (\mathsf{pre}\ g)\ B = \mathsf{preimage}\ g\ B$.

Proof. Expand definitions and apply Lemma 5.0.3 with $Y' = \mathsf{range}\ g$. □

Figure 5 defines more operations on preimage mappings, including pairing, composition, and disjoint union operations corresponding to the mapping operations in Figure 1. Roughly, the correspondence is that pre distributes over mapping operations to yield preimage mapping operations. The precise correspondence is the subject of the next three theorems, which will be used to derive the preimage arrow from the mapping arrow.

First, we need a new notion of equivalence.

Definition 5.0.5. *Two preimage mappings* $h_1 : X \underset{\mathsf{pre}}{\rightharpoonup} Y$ *and* $h_2 : X \underset{\mathsf{pre}}{\rightharpoonup} Y$ *are equivalent, or* $h_1 \equiv h_2$, *when* $\mathsf{pre\text{-}ap}\ h_1\ B = \mathsf{pre\text{-}ap}\ h_2\ B$ *for all* $B \subseteq Y$.

XXX: define equivalence in terms of equivalence, check observational equivalence in the proofs (specifically divergence)

5.1 Preimage Mapping Pairing

XXX: moar wurds in this section

Lemma 5.1.1 ($\mathsf{preimage}$ distributes over $\langle\cdot,\cdot\rangle_{\mathsf{map}}$ and $(\times)$)**.** *Let* $g_1 \in X \rightharpoonup Y_1$ *and* $g_2 \in X \rightharpoonup Y_2$. *For all* $B_1 \subseteq Y_1$ *and* $B_2 \subseteq Y_2$, $\mathsf{preimage}\ \langle g_1, g_2\rangle_{\mathsf{map}}\ (B_1 \times B_2) = (\mathsf{preimage}\ g_1\ B_1) \cap (\mathsf{preimage}\ g_2\ B_2)$.

Theorem 5.1.2 (pre distributes over $\langle\cdot,\cdot\rangle_{\mathsf{map}}$)**.** *Let* $g_1 \in X \rightharpoonup Y_1$ *and* $g_2 \in X \rightharpoonup Y_2$. *Then* $\mathsf{pre}\ \langle g_1, g_2\rangle_{\mathsf{map}} \equiv \langle \mathsf{pre}\ g_1, \mathsf{pre}\ g_2\rangle_{\mathsf{pre}}$.

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

$$\begin{aligned}&\mathsf{pre\text{-}ap}\ \langle \mathsf{pre}\ g_1, \mathsf{pre}\ g_2\rangle_{\mathsf{pre}}\ B\\ &\quad=\ \mathsf{let}\ \ Y' := Y_1' \times Y_2'\\ &\qquad\qquad\quad p := \lambda B.\ \bigcup_{\langle y_1,y_2\rangle \in B} (p_1\ \{y_1\}) \cap (p_2\ \{y_2\})\\ &\qquad\quad\ \ \mathsf{in}\ \ p\ (B \cap Y')\\ &\quad=\ \bigcup_{\langle y_1,y_2\rangle \in B\cap(Y_1'\times Y_2')} (p_1\ \{y_1\}) \cap (p_2\ \{y_2\})\\ &\quad=\ \bigcup_{\langle y_1,y_2\rangle \in B\cap(Y_1'\times Y_2')} (\mathsf{preimage}\ g_1\ \{y_1\}) \cap (\mathsf{preimage}\ g_2\ \{y_2\})\\ &\quad=\ \bigcup_{y \in B\cap(Y_1'\times Y_2')} (\mathsf{preimage}\ \langle g_1, g_2\rangle_{\mathsf{map}}\ \{y\})\\ &\quad=\ \mathsf{preimage}\ \langle g_1, g_2\rangle_{\mathsf{map}}\ (B \cap (Y_1' \times Y_2'))\\ &\quad=\ \mathsf{preimage}\ \langle g_1, g_2\rangle_{\mathsf{map}}\ B\\ &\quad=\ \mathsf{pre\text{-}ap}\ (\mathsf{pre}\ \langle g_1, g_2\rangle_{\mathsf{map}})\ B\end{aligned}$$

□

5.2 Preimage Mapping Composition

XXX: moar wurds in this section

Lemma 5.2.1 ($\mathsf{preimage}$ distributes over $(\circ_{\mathsf{map}})$)**.** *Let* $g_1 \in X \rightharpoonup Y$ *and* $g_2 \in Y \rightharpoonup Z$. *For all* $C \subseteq Z$, $\mathsf{preimage}\ (g_2 \circ_{\mathsf{map}} g_1)\ C = \mathsf{preimage}\ g_1\ (\mathsf{preimage}\ g_2\ C)$.

Theorem 5.2.2 (pre distributes over $(\circ_{\mathsf{map}})$)**.** *Let* $g_1 \in X \rightharpoonup Y$ *and* $g_2 \in Y \rightharpoonup Z$. *Then* $\mathsf{pre}\ (g_2 \circ_{\mathsf{map}} g_1) \equiv (\mathsf{pre}\ g_2) \circ_{\mathsf{pre}} (\mathsf{pre}\ g_1)$.

$$
\begin{aligned}
&X \underset{\mathsf{pre}}{\rightharpoonup} Y ::= \langle \mathsf{Set}\ Y, \mathsf{Set}\ Y \Rightarrow \mathsf{Set}\ X \rangle \\[1ex]
&\mathsf{pre} : (X \underset{\mathsf{map}}{\rightsquigarrow} Y) \Rightarrow (X \underset{\mathsf{pre}}{\rightharpoonup} Y) \\
&\mathsf{pre}\ g \ := \ \langle \mathsf{range}\ g, \lambda B.\, \mathsf{preimage}\ g\ B \rangle \\[1ex]
&\mathsf{pre\text{-}ap} : (X \underset{\mathsf{pre}}{\rightharpoonup} Y) \Rightarrow \mathsf{Set}\ Y \Rightarrow \mathsf{Set}\ X \\
&\mathsf{pre\text{-}ap}\ \langle Y', p \rangle\ B \ := \ p\ (B \cap Y') \\[1ex]
&\mathsf{pre\text{-}range} : (X \underset{\mathsf{pre}}{\rightharpoonup} Y) \Rightarrow \mathsf{Set}\ Y \\
&\mathsf{pre\text{-}range} \ := \ \mathsf{fst}
\end{aligned}
$$

$$
\begin{aligned}
&\langle \cdot, \cdot \rangle_{\mathsf{pre}} : (X \underset{\mathsf{pre}}{\rightharpoonup} Y_1) \Rightarrow (X \underset{\mathsf{pre}}{\rightharpoonup} Y_2) \Rightarrow (X \underset{\mathsf{pre}}{\rightharpoonup} Y_1 \times Y_2) \\
&\langle \langle Y'_1, p_1 \rangle, \langle Y'_2, p_2 \rangle \rangle_{\mathsf{pre}} \ := \ \mathsf{let}\ \ Y' := Y'_1 \times Y'_2 \\
&\qquad\qquad\qquad\qquad\qquad\quad p := \lambda B.\ \bigcup_{\langle y_1, y_2 \rangle \in B} (p_1\ \{y_1\}) \cap (p_2\ \{y_2\}) \\
&\qquad\qquad\qquad\qquad\qquad \mathsf{in}\ \ \langle Y', p \rangle \\[1ex]
&(\circ_{\mathsf{pre}}) : (Y \underset{\mathsf{pre}}{\rightharpoonup} Z) \Rightarrow (X \underset{\mathsf{pre}}{\rightharpoonup} Y) \Rightarrow (X \underset{\mathsf{pre}}{\rightharpoonup} Z) \\
&\langle Z', p_2 \rangle \circ_{\mathsf{pre}} h_1 \ := \ \langle Z', \lambda C.\, \mathsf{pre\text{-}ap}\ h_1\ (p_2\ C) \rangle \\[1ex]
&(\uplus_{\mathsf{pre}}) : (X \underset{\mathsf{pre}}{\rightharpoonup} Y) \Rightarrow (X \underset{\mathsf{pre}}{\rightharpoonup} Y) \Rightarrow (X \underset{\mathsf{pre}}{\rightharpoonup} Y) \\
&h_1 \uplus_{\mathsf{pre}} h_2 \ := \ \mathsf{let}\ \ Y' := (\mathsf{pre\text{-}range}\ h_1) \cup (\mathsf{pre\text{-}range}\ h_2) \\
&\qquad\qquad\qquad\quad p := \lambda B.\, (\mathsf{pre\text{-}ap}\ h_1\ B) \uplus (\mathsf{pre\text{-}ap}\ h_2\ B) \\
&\qquad\qquad\quad \mathsf{in}\ \ \langle Y', p \rangle
\end{aligned}
$$

Figure 5: Lazy preimage mappings and operations.

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

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

□

5.3 Preimage Mapping Disjoint Union

XXX: moar wurds in this section

Lemma 5.3.1 (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 5.3.2 (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 $B \subseteq Y$,

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

□

6. Deriving the Preimage Arrow

XXX: intro

$$X \underset{\mathsf{pre}}{\rightsquigarrow} Y \ ::= \ \mathsf{Set}\ X \Rightarrow (X \underset{\mathsf{pre}}{\rightharpoonup} Y) \tag{29}$$

$$
\begin{aligned}
&\mathsf{lift}_{\mathsf{pre}} : (X \underset{\mathsf{map}}{\rightsquigarrow} Y) \Rightarrow (X \underset{\mathsf{pre}}{\rightsquigarrow} Y) \\
&\mathsf{lift}_{\mathsf{pre}}\ g\ A \ := \ \mathsf{pre}\ (g\ A)
\end{aligned}
\tag{30}
$$

Definition 6.0.3 (Preimage arrow equivalence)**.** *Two preimage arrow computations* $h_1 : X \underset{\mathsf{pre}}{\rightsquigarrow} Y$ *and* $h_2 : X \underset{\mathsf{pre}}{\rightsquigarrow} Y$ *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 $(\mathbin{\&\&\&}_{\mathsf{pre}})$, $(\ggg_{\mathsf{pre}})$, $\mathsf{ifte}_{\mathsf{pre}}$ and $\mathsf{lazy}_{\mathsf{pre}}$ from their corresponding mapping arrow combinators, in a way that ensures $\mathsf{lift}_{\mathsf{pre}}$ is an arrow+choice 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 5.1.2, and rewrite in terms of $\mathsf{lift}_{\mathsf{pre}}$:

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

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 $\mathsf{pre\text{-}ap}$ from both sides of the equivalence gives a definition of $(\mathbin{\&\&\&}_{\mathsf{pre}})$ (Figure 6) for which (16) holds.

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

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

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

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

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

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

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

Figure 6: Preimage arrow definitions.

6.2 Case: Composition

Starting with the left side of (15), we expand definitions, apply Theorem 5.2.2 and rewrite in terms of $\mathsf{lift}_{\mathsf{pre}}$:

$$\begin{aligned}
&\mathsf{pre\text{-}ap}\ (\mathsf{lift}_{\mathsf{pre}}\ (\mathsf{g}_1 \ggg_{\mathsf{map}} \mathsf{g}_2)\ \mathsf{A})\ \mathsf{C}\\
&\quad\equiv\ \mathsf{let}\ \ \mathsf{g}'_1 := \mathsf{g}_1\ \mathsf{A}\\
&\qquad\qquad\ \ \mathsf{g}'_2 := \mathsf{g}_2\ (\mathsf{range}\ \mathsf{g}'_1)\\
&\qquad\quad\ \mathsf{in}\ \ \mathsf{pre\text{-}ap}\ (\mathsf{pre}\ (\mathsf{g}'_2 \circ_{\mathsf{map}} \mathsf{g}'_1))\ \mathsf{C}\\
&\quad\equiv\ \mathsf{let}\ \ \mathsf{g}'_1 := \mathsf{g}_1\ \mathsf{A}\\
&\qquad\qquad\ \ \mathsf{g}'_2 := \mathsf{g}_2\ (\mathsf{range}\ \mathsf{g}'_1)\\
&\qquad\quad\ \mathsf{in}\ \ \mathsf{pre\text{-}ap}\ ((\mathsf{pre}\ \mathsf{g}'_1) \circ_{\mathsf{pre}} (\mathsf{pre}\ \mathsf{g}'_2))\ \mathsf{C}\\
&\quad\equiv\ \mathsf{let}\ \ \mathsf{h}_1 := \mathsf{lift}_{\mathsf{pre}}\ \mathsf{g}_1\ \mathsf{A}\\
&\qquad\qquad\ \ \mathsf{h}_2 := \mathsf{lift}_{\mathsf{pre}}\ \mathsf{g}_2\ (\mathsf{pre\text{-}range}\ \mathsf{h}_1)\\
&\qquad\quad\ \mathsf{in}\ \ \mathsf{pre\text{-}ap}\ (\mathsf{h}_2 \circ_{\mathsf{pre}} \mathsf{h}_1)\ \mathsf{C}
\end{aligned}\tag{31}$$

Substituting h_1 for $\mathsf{lift}_{\mathsf{pre}}\ \mathsf{g}_1$ and h_2 for $\mathsf{lift}_{\mathsf{pre}}\ \mathsf{g}_2$, and removing the application of $\mathsf{pre\text{-}ap}$ 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 5.3.2, rewrite in terms of $\mathsf{lift}_{\mathsf{pre}}$, and apply Theorem 5.0.4 in the definitions of h_2 and h_3:

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

Substituting h_1 for $\mathsf{lift}_{\mathsf{pre}}\ \mathsf{g}_1$, h_2 for $\mathsf{lift}_{\mathsf{pre}}\ \mathsf{g}_2$ and h_3 for $\mathsf{lift}_{\mathsf{pre}}\ \mathsf{g}_3$, and removing the application of $\mathsf{pre\text{-}ap}$ 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 $\mathsf{lift}_{\mathsf{pre}}\ (\mathsf{g}\ 0)$:

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

Substituting $\mathsf{h}\ 0$ for $\mathsf{lift}_{\mathsf{pre}}\ (\mathsf{g}\ 0)$ and removing the application of $\mathsf{pre\text{-}ap}$ from both sides of the equivalence gives a definition for $\mathsf{lazy}_{\mathsf{pre}}$ (Figure 6) for which (18) holds.

6.5 Correctness

Theorem 6.5.1 (preimage arrow correctness)**.** $\mathsf{lift}_{\mathsf{pre}}$ *is an arrow+choice homomorphism.*

Proof. By construction. □

Corollary 6.5.2 (semantic correctness)**.** *If* $[\![e]\!]_{\mathsf{map}} : X \underset{\mathsf{map}}{\rightsquigarrow} Y$, *then* $\mathsf{lift}_{\mathsf{pre}}\ [\![e]\!]_{\mathsf{map}} \equiv [\![e]\!]_{\mathsf{pre}}$ *and* $[\![e]\!]_{\mathsf{pre}} : X \underset{\mathsf{pre}}{\rightsquigarrow} Y$.

In particular, $[\![e]\!]_{\mathsf{pre}}$ correctly computes preimages under the interpretation of e as a function from a random source.

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 $\mathsf{x_j} \in [0,1]$ from a uniform random source x. 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:

$$\mathsf{geometric}\ \mathsf{p}\ :=\ \mathsf{if}\ (\mathsf{random} < \mathsf{p})\ 0\ (1 + \mathsf{geometric}\ \mathsf{p}) \tag{32}$$

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

Suppose we interpret (32) as $\mathsf{h} : \mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathbb{N}$, a preimage arrow computation from random sources in X to natural numbers, and that we have a probability measure $\mathsf{P} \in \mathcal{P}\ \mathsf{X} \rightharpoonup [0,1]$. We could compute the probability of any output set $\mathsf{N} \subseteq \mathbb{N}$ using $\mathsf{P}\ (\mathsf{h}\ \mathsf{A}\ \mathsf{N})$, where $\mathsf{A} \subseteq \mathsf{X}$ and $\mathsf{P}\ \mathsf{A} = 1$. We have three hurdles to overcome:

1. Ensuring $\mathsf{h}\ \mathsf{A}$ converges.
2. Determining how random indexes numbers in x.
3. Ensuring each $\mathsf{x} \in \mathsf{X}$ contains enough random numbers.

Ensuring $\mathsf{h}\ \mathsf{A}$ converges is the most difficult, but doing the other two will provide enough structure to make it easy.

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 lists, 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 an infinite 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. In each case, the resulting definitions are large, conceptually difficult, and hard to formally manipulate. Fortunately, assigning each sub-computation a unique index into a tree-shaped random source, and passing it unchanged, is relatively easy.

We first need a way to assign unique indexes.

Definition 7.1.1 (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. If for every finite composition* f *of* left *and* right*,* $\mathsf{f}\ \mathsf{j}_0$ *is unique, then* J*,* j_0*,* left *and* right *define a* ***binary indexing scheme****.*

For example, let J be the set of lists of booleans, j_0 the empty list, and define

$$\begin{aligned} \mathsf{left}\ \mathsf{xs} &:= \langle \mathsf{true}, \mathsf{xs} \rangle \\ \mathsf{right}\ \mathsf{xs} &:= \langle \mathsf{false}, \mathsf{xs} \rangle \end{aligned} \tag{33}$$

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

$$\begin{aligned} \mathsf{left}\ (\mathsf{p}/\mathsf{q}) &:= (\mathsf{p} - \tfrac{1}{2})/\mathsf{q} \\ \mathsf{right}\ (\mathsf{p}/\mathsf{q}) &:= (\mathsf{p} + \tfrac{1}{2})/\mathsf{q} \end{aligned} \tag{34}$$

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

7.2 Applicative, Associative Store

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

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

$$\mathsf{AStore}\ \mathsf{s}\ (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \ ::=\ \mathsf{J} \Rightarrow (\langle \mathsf{s}, \mathsf{x} \rangle \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \tag{35}$$

XXX: motivational wurds for definition of lift:

$$\begin{aligned} \eta_{\mathsf{a}^*} &: (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \Rightarrow \mathsf{AStore}\ \mathsf{s}\ (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \\ \eta_{\mathsf{a}^*}\ \mathsf{f}\ \mathsf{j} &:= \mathsf{arr}_{\mathsf{a}}\ \mathsf{snd} \ggg_{\mathsf{a}} \mathsf{f} \end{aligned} \tag{36}$$

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

Definition 7.2.1 (AStore arrow equivalence)**.** *Two* AStore *arrow computations* k_1 *and* k_2 *are equivalent, or* $\mathsf{k}_1 \equiv \mathsf{k}_2$*, when* $\mathsf{k}_1\ \mathsf{j} \equiv \mathsf{k}_2\ \mathsf{j}$ *for all* $\mathsf{j} \in \mathsf{J}$*.*

Theorem 7.2.2 (AStore arrow correctness)**.** *Let* $\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y} ::= \mathsf{AStore}\ \mathsf{s}\ (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y})$*.* η_{a^*} *is an arrow+choice homomorphism.*

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

Composition. Starting with the right side of (15), expand definitions and use $(\mathsf{arr}_{\mathsf{a}}\ \mathsf{f}\ \&\&\&_{\mathsf{a}}\ \mathsf{f}_1) \ggg_{\mathsf{a}} \mathsf{arr}_{\mathsf{a}}\ \mathsf{snd} \equiv \mathsf{f}_1$:

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

Pairing. Starting with the right side of (16), expand definitions and use the arrow law $\mathsf{arr}_{\mathsf{a}}\ \mathsf{f} \ggg_{\mathsf{a}} (\mathsf{f}_1\ \&\&\&_{\mathsf{a}}\ \mathsf{f}_2) \equiv (\mathsf{arr}_{\mathsf{a}}\ \mathsf{f} \ggg_{\mathsf{a}} \mathsf{f}_1)\ \&\&\&_{\mathsf{a}}\ (\mathsf{arr}_{\mathsf{a}}\ \mathsf{f} \ggg_{\mathsf{a}} \mathsf{f}_2)$:

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

Conditional. Starting with the right side of (17), expand definitions and use the arrow law $\mathsf{arr}_{\mathsf{a}}\ \mathsf{f} \ggg_{\mathsf{a}} \mathsf{ifte}_{\mathsf{a}}\ \mathsf{f}_1\ \mathsf{f}_2\ \mathsf{f}_3 \equiv \mathsf{ifte}_{\mathsf{a}}\ (\mathsf{arr}_{\mathsf{a}}\ \mathsf{f} \ggg_{\mathsf{a}} \mathsf{f}_1)\ (\mathsf{arr}_{\mathsf{a}}\ \mathsf{f} \ggg_{\mathsf{a}} \mathsf{f}_2)\ (\mathsf{arr}_{\mathsf{a}}\ \mathsf{f} \ggg_{\mathsf{a}} \mathsf{f}_3)$:

$$\begin{aligned} &(\mathsf{ifte}_{\mathsf{a}^*}\ (\eta_{\mathsf{a}^*}\ \mathsf{f}_1)\ (\eta_{\mathsf{a}^*}\ \mathsf{f}_2)\ (\eta_{\mathsf{a}^*}\ \mathsf{f}_3))\ \mathsf{j} \\ &\quad\equiv \mathsf{ifte}_{\mathsf{a}}\ (\mathsf{arr}_{\mathsf{a}}\ \mathsf{snd} \ggg_{\mathsf{a}} \mathsf{f}_1) \\ &\qquad\qquad (\mathsf{arr}_{\mathsf{a}}\ \mathsf{snd} \ggg_{\mathsf{a}} \mathsf{f}_2) \\ &\qquad\qquad (\mathsf{arr}_{\mathsf{a}}\ \mathsf{snd} \ggg_{\mathsf{a}} \mathsf{f}_3) \\ &\quad\equiv \mathsf{arr}_{\mathsf{a}}\ \mathsf{snd} \ggg_{\mathsf{a}} (\mathsf{ifte}\ \mathsf{f}_1\ \mathsf{f}_2\ \mathsf{f}_3) \\ &\quad\equiv \eta_{\mathsf{a}^*}\ (\mathsf{ifte}\ \mathsf{f}_1\ \mathsf{f}_2\ \mathsf{f}_3)\ \mathsf{j} \end{aligned}$$

Laziness. Starting with the right side of (18), expand definitions, β-expand within the outer thunk, and use the arrow law $\mathsf{arr}_{\mathsf{a}}\ \mathsf{f} \ggg_{\mathsf{a}} \mathsf{lazy}_{\mathsf{a}}\ \mathsf{f}_1 \equiv \mathsf{lazy}_{\mathsf{a}}\ \lambda 0.\, \mathsf{arr}_{\mathsf{a}}\ \mathsf{f} \ggg_{\mathsf{a}} \mathsf{f}_1\ 0$:

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

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

Corollary 7.2.3 (semantic correctness)**.** *Let* $\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y} ::= \mathsf{AStore}\ \mathsf{s}\ (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y})$*. 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 7.2.3 implies that, if a pure let-calculus expression is interpreted as a computation $\mathsf{k} : \mathsf{AStore}\ \mathsf{S}\ (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y})$, then $\mathsf{k}\ \mathsf{j}_0$ correctly computes preimages. We still need to know that preimages under functions that use the store are computed correctly, which we will get to after defining stores and combinators that access them.

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

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

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

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

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

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

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

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

7.3 Probabilistic Programs

Define $R := J \to [0,1]$ as the set of infinite random sources. Each $r \in R$ is a total mapping from an expression index to a random number. Equivalently, each $r \in R$ is an infinite vector of random numbers indexed by J.

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

$$\begin{aligned} &\mathsf{random}_{a^*} : x \rightsquigarrow_{a^*} [0,1] \\ &\mathsf{random}_{a^*}\ j \ := \ \mathsf{arr}_a\ (\mathsf{fst} \ggg \pi\ j) \end{aligned} \tag{37}$$

We extend the let-calculus semantic function with

$$[\![\mathsf{random}]\!]_{a^*} :\equiv \mathsf{random}_{a^*} \tag{38}$$

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

7.4 Partial Programs

Divergence is an applicative effect, so it should not be too surprising that it can be modeled using an applicative store. Roughly, the store determines which branch of each conditional, if any, is allowed to continue computing.

Definition 7.4.1 (branch trace)**.** *A **branch trace** is a total mapping* $t \in J \to \mathsf{Bool}_\perp$ *such that* $t\ j = \mathsf{true}$ *or* $t\ j = \mathsf{false}$ *for no more than finitely many* $j \in J$.

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

$$\begin{aligned} &\mathsf{branch}_{a^*} : x \rightsquigarrow_{a^*} \mathsf{Bool} \\ &\mathsf{branch}_{a^*}\ j \ := \ \mathsf{arr}_a\ (\mathsf{fst} \ggg \pi\ j) \end{aligned} \tag{39}$$

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

$$\begin{aligned} &\mathsf{agrees} : \langle \mathsf{Bool}, \mathsf{Bool}\rangle \Rightarrow \mathsf{Bool}_\perp \\ &\mathsf{agrees}\ \langle a_1, a_2\rangle \ := \ \mathsf{if}\ (a_1 = a_2)\ a_1\ \perp \end{aligned} \tag{40}$$

$$\begin{aligned} &\mathsf{ifte}'_{a^*} : (x \rightsquigarrow_{a^*} \mathsf{Bool}) \Rightarrow (x \rightsquigarrow_{a^*} y) \Rightarrow (x \rightsquigarrow_{a^*} y) \Rightarrow (x \rightsquigarrow_{a^*} y) \\ &\mathsf{ifte}'_{a^*}\ k_1\ k_2\ k_3 \ := \\ &\quad \mathsf{ifte}_{a^*}\ ((k_1\ \&\&\&_{a^*}\ \mathsf{branch}_{a^*}) \ggg_{a^*} \mathsf{arr}_{a^*}\ \mathsf{agrees})\ k_2\ k_3 \end{aligned} \tag{41}$$

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

We will prove that this correctly returns $\perp$ instead of diverging after defining the meaning of partial, probabilistic let-calculus programs.

7.5 Partial, Probabilistic Programs

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

$$\begin{aligned} &\mathsf{random}_{a^*} : x \rightsquigarrow_{a^*} [0,1] \\ &\mathsf{random}_{a^*}\ j \ := \ \mathsf{arr}_a\ (\mathsf{fst} \ggg \mathsf{fst} \ggg \pi\ j) \end{aligned} \tag{42}$$

$$\begin{aligned} &\mathsf{branch}_{a^*} : x \rightsquigarrow_{a^*} \mathsf{Bool} \\ &\mathsf{branch}_{a^*}\ j \ := \ \mathsf{arr}_a\ (\mathsf{fst} \ggg \mathsf{snd} \ggg \pi\ j) \end{aligned} \tag{43}$$

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

We define a new semantic function $[\![\cdot]\!]'_{a^*}$ by extending $[\![\cdot]\!]_{a^*}$:

$$\begin{aligned} [\![\mathsf{if}\ e_c\ e_t\ e_f]\!]'_{a^*} &:\equiv \mathsf{ifte}'_{a^*}\ [\![e_c]\!]_{a^*}\ (\mathsf{lazy}_a\ \lambda 0.\, [\![e_t]\!]_{a^*})\ (\mathsf{lazy}_a\ \lambda 0.\, [\![e_f]\!]_{a^*}) \\ [\![e]\!]'_{a^*} &:\equiv [\![e]\!]_{a^*} \end{aligned} \tag{44}$$

7.6 Correctness

Theorem 7.6.1 (natural transformation)**.** *Let* $x \rightsquigarrow_{a^*} y ::= \mathsf{AStore}\ s\ (x \rightsquigarrow_a y)$ *and* $x \rightsquigarrow_{b^*} y ::= \mathsf{AStore}\ s\ (x \rightsquigarrow_{b^*} y)$. *Let* $\mathsf{lift}_b : (x \rightsquigarrow_a y) \Rightarrow (x \rightsquigarrow_b y)$ *be an arrow+choice homomorphism, and define*

$$\begin{aligned} &\mathsf{lift}_{b^*} : (x \rightsquigarrow_{a^*} y) \Rightarrow (x \rightsquigarrow_{b^*} y) \\ &\mathsf{lift}_{b^*}\ f\ j \ := \ \mathsf{lift}_b\ (f\ j) \end{aligned} \tag{45}$$

Then the following diagram commutes:

$$\begin{tikzcd} x \rightsquigarrow_a y \arrow[r, "\mathsf{lift}_b"] \arrow[d, "\eta_{a^*}"'] & x \rightsquigarrow_b y \arrow[d, "\eta_{b^*}"] \\ x \rightsquigarrow_{a^*} y \arrow[r, "\mathsf{lift}_{b^*}"'] & x \rightsquigarrow_{b^*} y \end{tikzcd} \tag{46}$$

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

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

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

Lastly, because $\eta_{\mathsf{a^*}}$, $\eta_{\mathsf{b^*}}$, and $\mathsf{lift_b}$ are homomorphisms, $\mathsf{lift_{b^*}}$ is a homomorphism by composition.

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

Corollary 7.6.2 (semantic correctness)**.** *The following diagram commutes:*

$$\begin{tikzcd} \mathsf{X} \rightsquigarrow_{\perp} \mathsf{Y} \arrow[r, "\mathsf{lift_{pre}} \circ \mathsf{lift_{map}}"] \arrow[d, "\eta_{\perp^*}"'] & \mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y} \arrow[d, "\eta_{\mathsf{pre}^*}"] \\ \mathsf{X} \rightsquigarrow_{\perp^*} \mathsf{Y} \arrow[r, "\mathsf{lift_{pre^*}} \circ \mathsf{lift_{map^*}}"'] & \mathsf{X} \underset{\mathsf{pre}^*}{\rightsquigarrow} \mathsf{Y} \end{tikzcd} \tag{47}$$

Further, $\mathsf{lift_{pre^*}} \circ \mathsf{lift_{map^*}}$ *is an arrow+choice homomorphism.*

In particular, $[\![e]\!]'_{\mathsf{pre}^*}$ correctly computes preimages under the interpretation of e as a partial function from an infinite random source.

8. Approximating Preimage Arrow

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

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

Figure 4:

- pre: can't implement
- $\mathsf{pre\text{-}ap}$: 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:

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

Figure 8:

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

9. 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.

```
id_pre : X ⇝_pre X
id_pre A  :=  ⟨A, λB. B⟩

const_pre : Y ⇒ X ⇝_pre Y
const_pre y A  :=  ⟨{y}, λB. if (B = ∅) ∅ A⟩

π_pre : J ⇒ (J → X) ⇝_pre X
π_pre j A  :=  let  A_j := proj j A
                    p := λB. A ∩ ∏_{i∈J} if (j = i) B (proj i A)
                in  ⟨A_j, p⟩

fst_pre : ⟨X, Y⟩ ⇝_pre X
fst_pre A  :=  let  A_1 := image fst A
                    A_2 := image snd A
                in  ⟨A_1, λB. A ∩ (B × A_2)⟩

snd_pre : ⟨X, Y⟩ ⇝_pre Y
snd_pre A  :=  let  A_1 := image fst A
                    A_2 := image snd A
                in  ⟨A_2, λB. A ∩ (A_1 × B)⟩
```

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