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

TODO: equivalence relation for λ_{ZFC} terms, that at least handles divergence

1. Introduction

1. Define the *bottom arrow*, type $\mathsf{X} \Rightarrow \mathsf{Y}_\perp$, a compilation target for first-order functions that may raise errors.
2. Derive the *mapping arrow* from the bottom arrow, type $\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}$. Its instances return extensional functions, or mappings, that compute the same values as their corresponding bottom arrow computations, but have observable domains.
3. Derive the *preimage arrow* from the mapping arrow, type $\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}$. Instances compute preimages under their corresponding mapping arrow instances.
4. Derive *XXX* from the preimage arrow. Instances compute conservative approximations of the preimages computed by their corresponding preimage arrow instances.

Only the first and last artifacts—the bottom arrow and the XXX—can be implemented.

2. Mathematics and 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 mathematics in this paper in λ_{ZFC} [2], an untyped, call-by-value lambda calculus designed for manually deriving computable 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 $\mathsf{Set}\ \mathsf{A}$ denotes the same values as the powerset $\mathcal{P}\ \mathsf{A}$, or *subsets* of A. Similarly, the type $\langle \mathsf{A}, \mathsf{B} \rangle$ denotes the same values as the product set $\mathsf{A} \times \mathsf{B}$.

[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:

$$\begin{array}{rr}
\mathsf{true} : \mathsf{Bool} & (\in) : \mathsf{x} \Rightarrow \mathsf{Set}\ \mathsf{x} \Rightarrow \mathsf{Bool} \\
\mathsf{false} : \mathsf{Bool} & \mathcal{P} : \mathsf{Set}\ \mathsf{x} \Rightarrow \mathsf{Set}\ (\mathsf{Set}\ \mathsf{x}) \\
\varnothing : \mathsf{Set}\ \mathsf{x} & \bigcup : \mathsf{Set}\ (\mathsf{Set}\ \mathsf{x}) \Rightarrow \mathsf{Set}\ \mathsf{x} \\
\omega : \mathsf{Ord} & \mathsf{image} : (\mathsf{x} \Rightarrow \mathsf{y}) \Rightarrow \mathsf{Set}\ \mathsf{x} \Rightarrow \mathsf{Set}\ \mathsf{y} \\
\mathsf{take} : \mathsf{Set}\ \mathsf{x} \Rightarrow \mathsf{x} & \mathsf{card} : \mathsf{Set}\ \mathsf{x} \Rightarrow \mathsf{Ord}
\end{array} \tag{1}$$

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

$$\mathsf{x} = \mathsf{y}\ :=\ \mathsf{x} \in \{\mathsf{y}\} \tag{2}$$

which means

$$(=)\ :=\ \lambda \mathsf{x}.\, \lambda \mathsf{y}.\, \mathsf{x} \in \{\mathsf{y}\} \tag{3}$$

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 2.1.1 (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

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

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

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

$$\begin{aligned}
&(\uplus) : \mathsf{Set}\ \mathsf{x} \Rightarrow \mathsf{Set}\ \mathsf{x} \Rightarrow \mathsf{Set}\ \mathsf{x} \\
&\mathsf{A} \uplus \mathsf{B}\ :=\ \mathsf{if}\ (\mathsf{A} \cap \mathsf{B} = \varnothing)\ (\mathsf{A} \cup \mathsf{B})\ (\mathsf{take}\ \varnothing)
\end{aligned} \tag{4}$$

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:

$$\begin{aligned}
&(\cdot)|_{(\cdot)} : (\mathsf{X} \Rightarrow \mathsf{Y}) \Rightarrow \mathsf{Set}\ \mathsf{X} \Rightarrow (\mathsf{X} \rightharpoonup \mathsf{Y}) \\
&\mathsf{f}|_{\mathsf{A}}\ :=\ \mathsf{image}\ (\lambda \mathsf{x}.\, (\mathsf{x}, \mathsf{f}\ \mathsf{x}))\ \mathsf{A}
\end{aligned} \tag{5}$$

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

$$\begin{aligned}
&\pi : \mathsf{J} \Rightarrow (\mathsf{J} \to \mathsf{X}) \Rightarrow \mathsf{X} \\
&\pi\ \mathsf{j}\ \mathsf{f}\ :=\ \mathsf{f}\ \mathsf{j}
\end{aligned} \tag{6}$$

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

3. The Bottom Arrow

One way to be certain an arrow correctly computes preimages under functions f is to ultimately *derive* it from a simpler arrow used to construct f. One obvious candidate is the **function arrow**, in which $\mathsf{x} \rightsquigarrow \mathsf{y} ::= \mathsf{x} \Rightarrow \mathsf{y}$, whose implementation is extremely simple (e.g. $\mathsf{f}_1 \ggg \mathsf{f}_2$ is $\mathsf{f}_2 \circ \mathsf{f}_1$). 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 2 defines the ***bottom arrow***. Its computations are of type $\mathsf{x} \Rightarrow \mathsf{y}_\perp$, 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 $\mathsf{Bool} \cup \{\perp\}$.

Figure 2 does not give the typical minimal definition consisting of arr, $(\ggg)$ and first combinators. Instead of $\mathsf{first}_\perp$, it defines $(\&\&\&_\perp)$—typically called **fanout**, but its use in this paper will be clearer if we call it ***pairing***—which applies two functions to the same input and returns the pair of their outputs. Though any arrow's first may be defined in terms of its $(\&\&\&)$ and vice-versa [1], we give $(\&\&\&)$ definitions in this paper because the most well-known applicable contemporary measurability theorems are in terms of pairing functions.

Figure 2 also defines the ***bottom arrow+choice*** by defining the additional combinators $\mathsf{ifte}_\perp$ (if-then-else) and $\mathsf{lazy}_\perp$. An arrow is typically strengthened to an arrow+choice (which is not quite as strong as a monad) by

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

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

Figure 1: Operations on mappings.

$$
\begin{aligned}
&\mathsf{arr}_\bot : (\mathsf{x} \Rightarrow \mathsf{y}) \Rightarrow (\mathsf{x} \Rightarrow \mathsf{y}_\bot)\\
&\mathsf{arr}_\bot\ \mathsf{f}\ :=\ \mathsf{f}\\[1ex]
&(\ggg_\bot) : (\mathsf{x} \Rightarrow \mathsf{y}_\bot) \Rightarrow (\mathsf{y} \Rightarrow \mathsf{z}_\bot) \Rightarrow (\mathsf{x} \Rightarrow \mathsf{z}_\bot)\\
&(\mathsf{f}_1 \ggg_\bot \mathsf{f}_2)\ \mathsf{x}\ :=\ \mathsf{if}\ (\mathsf{f}_1\ \mathsf{x} = \bot)\ \bot\ (\mathsf{f}_2\ (\mathsf{f}_1\ \mathsf{x}))\\[1ex]
&(\&\&\&_\bot) : (\mathsf{x} \Rightarrow \mathsf{y_1}_\bot) \Rightarrow (\mathsf{x} \Rightarrow \mathsf{y_2}_\bot) \Rightarrow (\mathsf{x} \Rightarrow \langle \mathsf{y}_1, \mathsf{y}_2\rangle_\bot)\\
&(\mathsf{f}_1\ \&\&\&_\bot\ \mathsf{f}_2)\ \mathsf{x}\ :=\ \mathsf{if}\ (\mathsf{f}_1\ \mathsf{x} = \bot\ \mathsf{or}\ \mathsf{f}_2\ \mathsf{x} = \bot)\ \bot\ \langle \mathsf{f}_1\ \mathsf{x}, \mathsf{f}_2\ \mathsf{x}\rangle
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{ifte}_\bot : (\mathsf{x} \Rightarrow \mathsf{Bool}_\bot) \Rightarrow (\mathsf{x} \Rightarrow \mathsf{y}_\bot) \Rightarrow (\mathsf{x} \Rightarrow \mathsf{y}_\bot) \Rightarrow (\mathsf{x} \Rightarrow \mathsf{y}_\bot)\\
&\mathsf{ifte}_\bot\ \mathsf{f}_1\ \mathsf{f}_2\ \mathsf{f}_3\ \mathsf{x}\ :=\ \mathsf{case}\ \ \mathsf{f}_1\ \mathsf{x}\\
&\qquad\qquad\qquad\qquad\quad \mathsf{true}\ \Longrightarrow\ \mathsf{f}_2\ \mathsf{x}\\
&\qquad\qquad\qquad\qquad\quad \mathsf{false}\ \Longrightarrow\ \mathsf{f}_3\ \mathsf{x}\\
&\qquad\qquad\qquad\qquad\quad \mathsf{else}\ \Longrightarrow\ \bot\\[1ex]
&\mathsf{lazy}_\bot : (1 \Rightarrow (\mathsf{x} \Rightarrow \mathsf{y}_\bot)) \Rightarrow (\mathsf{x} \Rightarrow \mathsf{y}_\bot)\\
&\mathsf{lazy}_\bot\ \mathsf{f}\ \mathsf{x}\ :=\ \mathsf{f}\ 0\ \mathsf{x}
\end{aligned}
$$

Figure 2: Bottom arrow definitions.

defining a left combinator. Again, however, defining $\mathsf{ifte}_\bot$ instead of $\mathsf{left}_\bot$ will make it easier to apply contemporary measurability theorems, which are in terms of disjoint unions of mappings instead of an explicit disjoint union type.

In a nonstrict or simply typed λ-calculus, $\mathsf{lazy}_\bot$ is unnecessary. For example, in a simply typed λ-calculus, the following recursive function cannot be typed:

$$\mathsf{halt\text{-}on\text{-}true}_\bot\ :=\ \mathsf{ifte}_\bot\ (\mathsf{arr}_\bot\ \mathsf{id})\ (\mathsf{arr}_\bot\ \mathsf{id})\ \mathsf{halt\text{-}on\text{-}true}_\bot \tag{8}$$

and it diverges in a nonstrict λ-calculus only when applied to false. However, its *defining expression* diverges in λ_{ZFC} and every other call-by-value λ-calculus.

We defer the inner $\mathsf{halt\text{-}on\text{-}true}_\bot$ until after the outer $\mathsf{halt\text{-}on\text{-}true}_\bot$ is applied, using $\mathsf{lazy}_\bot$:

$$
\begin{aligned}
&\mathsf{halt\text{-}on\text{-}true}_\bot\ :=\\
&\quad \mathsf{ifte}_\bot\ (\mathsf{arr}_\bot\ \mathsf{id})\ (\mathsf{arr}_\bot\ \mathsf{id})\ (\mathsf{lazy}_\bot\ \lambda 0.\, \mathsf{halt\text{-}on\text{-}true}_\bot)
\end{aligned} \tag{9}
$$

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

XXX: point out that $\mathsf{lazy}_\bot$ receives a thunk, and remind readers that $1 = \{0\}$

Theorem 3.0.1. *$\mathsf{arr}_\bot$, $(\&\&\&_\bot)$ and $(\ggg_\bot)$ define an arrow. With $\mathsf{ifte}_\bot$ and $\mathsf{lazy}_\bot$, they define an arrow+choice.*

Proof. The bottom arrow is the arrow in the Kleisli category of the Maybe monad with $\mathsf{Nothing} = \bot$. ☐

4. First-Order Let-Calculus Semantics

XXX: Figure 3...

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 ($\mathsf{let}\ e_0\ e_b$) pushes e_0 onto the stack. Variables are referenced using ($\mathsf{env}\ n$) with ($\mathsf{env}\ 0$) referring to the head.

5. 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 $(\&\&\&_{\mathsf{map}}) := \langle\cdot,\cdot\rangle_{\mathsf{map}}$ and $(\ggg_{\mathsf{map}}) := \mathsf{flip}\ (\circ_{\mathsf{map}})$. Unfortunately, we run into a problem defining $\mathsf{arr}_{\mathsf{map}} : (\mathsf{X} \Rightarrow \mathsf{Y}) \Rightarrow (\mathsf{X} \rightharpoonup \mathsf{Y})$: we cannot define it as $\mathsf{arr}_{\mathsf{map}}\ \mathsf{f} := \mathsf{f}|_{\mathsf{X}}$. Although X is a λ_{ZFC} value, it is not available within any definition because it is only part of the type.

We need to parameterize computations on a domain, so

$$\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}\ ::=\ \mathsf{Set}\ \mathsf{X} \Rightarrow (\mathsf{X} \rightharpoonup \mathsf{Y}) \tag{10}$$

is the type of ***mapping arrow*** computations.

Notice that $\bot$ 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)

We want the correspondence between bottom arrow and mapping arrow computations as clear as possible. We therefore start by defining a function $\mathsf{lift}_{\mathsf{map}} : (\mathsf{X} \Rightarrow \mathsf{Y}_\bot) \Rightarrow (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y})$ to lift bottom arrow computations to the mapping arrow. It must restrict its argument f's domain to a subset of X for which f does not return $\bot$. It is helpful to have a

$$\llbracket x := e;\ \cdots \rrbracket_{\mathsf{a}} \ :\equiv\ x := \llbracket e \rrbracket_{\mathsf{a}} ;\ \cdots$$
$$\llbracket x\ e \rrbracket_{\mathsf{a}} \ :\equiv\ \llbracket \langle e, 0 \rangle \rrbracket_{\mathsf{a}} \ggg_{\mathsf{a}} x$$
$$\llbracket \langle e_1, e_2 \rangle \rrbracket_{\mathsf{a}} \ :\equiv\ \llbracket e_1 \rrbracket_{\mathsf{a}}\ \&\&\&_{\mathsf{a}}\ \llbracket e_2 \rrbracket_{\mathsf{a}}$$
$$\llbracket \mathsf{let}\ e_0\ e_b \rrbracket_{\mathsf{a}} \ :\equiv\ (\llbracket e_0 \rrbracket_{\mathsf{a}}\ \&\&\&_{\mathsf{a}}\ (\mathsf{arr_a\ id})) \ggg_{\mathsf{a}} \llbracket e_b \rrbracket_{\mathsf{a}}$$
$$\llbracket \mathsf{env}\ n \rrbracket_{\mathsf{a}} \ :\equiv\ \mathsf{arr_a}\ \lambda\gamma.\,\gamma_n$$
$$\llbracket \mathsf{if}\ e_c\ e_t\ e_f \rrbracket_{\mathsf{a}} \ :\equiv\ \mathsf{ifte_a}\ \llbracket e_c \rrbracket_{\mathsf{a}}\ (\mathsf{lazy_a}\ \lambda 0.\, \llbracket e_t \rrbracket_{\mathsf{a}})\ (\mathsf{lazy_a}\ \lambda 0.\, \llbracket e_f \rrbracket_{\mathsf{a}})$$
$$\llbracket v \rrbracket_{\mathsf{a}} \ :\equiv\ \mathsf{arr_a}\ \lambda\gamma.\, v$$
$$\llbracket \mathsf{fst}\ e \rrbracket_{\mathsf{a}} \ :\equiv\ \llbracket e \rrbracket_{\mathsf{a}} \ggg_{\mathsf{a}} (\mathsf{arr_a\ fst})$$
$$\llbracket \mathsf{snd}\ e \rrbracket_{\mathsf{a}} \ :\equiv\ \llbracket e \rrbracket_{\mathsf{a}} \ggg_{\mathsf{a}} (\mathsf{arr_a\ snd})$$
$$\llbracket e_1 = e_2 \rrbracket_{\mathsf{a}} \ :\equiv\ \llbracket \langle e_1, e_2 \rangle \rrbracket_{\mathsf{a}} \ggg_{\mathsf{a}} (\mathsf{arr_a}\ \lambda \langle \mathsf{x}, \mathsf{y} \rangle .\, \mathsf{x} = \mathsf{y})$$
$$\llbracket e_1 + e_2 \rrbracket_{\mathsf{a}} \ :\equiv\ \llbracket \langle e_1, e_2 \rangle \rrbracket_{\mathsf{a}} \ggg_{\mathsf{a}} (\mathsf{arr_a}\ \lambda \langle \mathsf{x}, \mathsf{y} \rangle .\, \mathsf{x} + \mathsf{y})$$
$$\cdots$$

Figure 3: Transformation from a let-calculus with first-order definitions and De-Bruijn-indexed bindings to computations in arrow 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:

$$\begin{aligned} &\mathsf{domain}_\perp : (\mathsf{X} \Rightarrow \mathsf{Y}_\perp) \Rightarrow \mathsf{Set\ X} \Rightarrow \mathsf{Set\ X} \\ &\mathsf{domain}_\perp\ \mathsf{f\ A}\ :=\ \mathsf{preimage\ f}|_{\mathsf{A}}\ ((\mathsf{image\ f\ A}) \backslash \{\perp\}) \end{aligned} \tag{11}$$

$$\begin{aligned} &\mathsf{lift_{map}} : (\mathsf{X} \Rightarrow \mathsf{Y}_\perp) \Rightarrow (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}) \\ &\mathsf{lift_{map}\ f\ A}\ :=\ \mathsf{let}\ \ \mathsf{A}' := \mathsf{domain}_\perp\ \mathsf{f\ A} \\ &\qquad\qquad\qquad \mathsf{in}\ \ \mathsf{f}|_{\mathsf{A}'} \end{aligned} \tag{12}$$

5.1 Distributive Laws

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 a particular kind of **homomorphism**. More concretely, for any let-calculus expression e, we would like $\llbracket e \rrbracket_{\mathsf{map}} \equiv \mathsf{lift_{map}}\ \llbracket e \rrbracket_\perp$.

Definition 5.1.1 (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* $\mathsf{f_2}$*:*

$$\mathsf{lift_b}\ (\mathsf{arr_a\ f})\ \equiv\ \mathsf{arr_b\ f} \tag{13}$$
$$\mathsf{lift_b}\ (\mathsf{f_1}\ \&\&\&_{\mathsf{a}}\ \mathsf{f_2})\ \equiv\ (\mathsf{lift_b\ f_1})\ \&\&\&_{\mathsf{b}}\ (\mathsf{lift_b\ f_2}) \tag{14}$$
$$\mathsf{lift_b}\ (\mathsf{f_1} \ggg_{\mathsf{a}} \mathsf{f_2})\ \equiv\ (\mathsf{lift_b\ f_1}) \ggg_{\mathsf{b}} (\mathsf{lift_b\ f_2}) \tag{15}$$

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

$$\mathsf{lift_b}\ (\mathsf{ifte_a\ f_1\ f_2\ f_3})\ \equiv\ \mathsf{ifte_b}\ (\mathsf{lift_b\ f_1})\ (\mathsf{lift_b\ f_2})\ (\mathsf{lift_b\ f_3}) \tag{16}$$
$$\mathsf{lift_b}\ (\mathsf{lazy_a\ f})\ \equiv\ \mathsf{lazy_b}\ \lambda 0.\, \mathsf{lift_b}\ (\mathsf{f}\ 0) \tag{17}$$

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

Recall that in nonstrict or strongly normalizing languages, a lazy combinator is unnecessary. In these languages, only (16) is necessary for arrow+choice homomorphisms.

Because mapping arrow computations are functions, we need to extend the notion of their equivalence—which by default is alpha-equivalence—to something more extensional.

Definition 5.1.2 (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* $\mathsf{g_1\ A} \equiv \mathsf{g_2\ A}$ *for all* $\mathsf{A} \subseteq \mathsf{X}$*.*

Clearly $\mathsf{arr_b} := \mathsf{lift_b} \circ \mathsf{arr_a}$ meets (13), 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.

5.2 Case: Pairing

Starting with the left side of (14), we first expand definitions. For any $\mathsf{f_1} : \mathsf{X} \Rightarrow \mathsf{Y}_\perp$, $\mathsf{f_2} : \mathsf{X} \Rightarrow \mathsf{Z}_\perp$, and $\mathsf{A} \subseteq \mathsf{X}$,

$$\begin{aligned} &\mathsf{lift_{map}}\ (\mathsf{f_1}\ \&\&\&_\perp\ \mathsf{f_2})\ \mathsf{A} \\ &\quad \equiv\ \mathsf{lift_{map}}\ (\lambda \mathsf{x}.\, \mathsf{if}\ (\mathsf{f_1\ x} = \perp\ \mathsf{or}\ \mathsf{f_2\ x} = \perp)\ \perp\ \langle \mathsf{f_1\ x}, \mathsf{f_2\ x} \rangle)\ \mathsf{A} \\ &\quad \equiv\ \mathsf{let}\ \ \ \mathsf{f} := \lambda \mathsf{x}.\, \mathsf{if}\ (\mathsf{f_1\ x} = \perp\ \mathsf{or}\ \mathsf{f_2\ x} = \perp)\ \perp\ \langle \mathsf{f_1\ x}, \mathsf{f_2\ x} \rangle \\ &\qquad\qquad \mathsf{A}' := \mathsf{domain}_\perp\ \mathsf{f\ A} \\ &\qquad\quad \mathsf{in}\ \ \mathsf{f}|_{\mathsf{A}'} \end{aligned} \tag{18}$$

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

$$\begin{aligned} &\mathsf{lift_{map}}\ (\mathsf{f_1}\ \&\&\&_\perp\ \mathsf{f_2})\ \mathsf{A} \\ &\quad \equiv\ \mathsf{let}\ \ \mathsf{A_1} := (\mathsf{domain}_\perp\ \mathsf{f_1\ A}) \\ &\qquad\qquad \mathsf{A_2} := (\mathsf{domain}_\perp\ \mathsf{f_2\ A}) \\ &\qquad\qquad \mathsf{A}' := \mathsf{A_1} \cap \mathsf{A_2} \\ &\qquad\quad \mathsf{in}\ \ \lambda \mathsf{x} \in \mathsf{A}'.\, \langle \mathsf{f_1\ x}, \mathsf{f_2\ x} \rangle \\ &\quad \equiv\ \mathsf{let}\ \ \mathsf{g_1} := \mathsf{lift_{map}\ f_1\ A} \\ &\qquad\qquad \mathsf{g_2} := \mathsf{lift_{map}\ f_2\ A} \\ &\qquad\qquad \mathsf{A}' := (\mathsf{domain\ g_1}) \cap (\mathsf{domain\ g_2}) \\ &\qquad\quad \mathsf{in}\ \ \lambda \mathsf{x} \in \mathsf{A}'.\, \langle \mathsf{g_1\ x}, \mathsf{g_2\ x} \rangle \\ &\quad \equiv\ \langle \mathsf{lift_{map}\ f_1\ A}, \mathsf{lift_{map}\ f_2\ A} \rangle_{\mathsf{map}} \end{aligned} \tag{19}$$

Substituting $\mathsf{g_1}$ for $\mathsf{lift_{map}\ f_1}$ and $\mathsf{g_2}$ for $\mathsf{lift_{map}\ f_2}$ gives a definition for $(\&\&\&_{\mathsf{map}})$ (Figure 4) for which (14) holds.

5.3 Case: Composition

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

XXX: include it?

5.4 Case: Conditional

Starting with the left side of (16), we expand definitions, and simplify f by restricting it to a domain for which $\mathsf{f_1\ x}$ cannot be $\perp$:

$$\begin{aligned} &\mathsf{lift_{map}}\ (\mathsf{ifte}_\perp\ \mathsf{f_1\ f_2\ f_3})\ \mathsf{A} \\ &\quad \equiv\ \mathsf{let}\ \ \ \mathsf{f} := \lambda \mathsf{x}.\, \mathsf{case}\ \ \mathsf{f_1\ x} \\ &\qquad\qquad\qquad\qquad \mathsf{true}\ \Longrightarrow\ \mathsf{f_2\ x} \\ &\qquad\qquad\qquad\qquad \mathsf{false}\ \Longrightarrow\ \mathsf{f_3\ x} \\ &\qquad\qquad\qquad\qquad \mathsf{else}\ \Longrightarrow\ \perp \\ &\qquad\qquad \mathsf{A}' := \mathsf{domain}_\perp\ \mathsf{f\ A} \\ &\qquad\quad \mathsf{in}\ \ \mathsf{f}|_{\mathsf{A}'} \\ &\quad \equiv\ \mathsf{let}\ \ \mathsf{A_2} := \mathsf{preimage\ f_1}|_{\mathsf{A}}\ \{\mathsf{true}\} \\ &\qquad\qquad \mathsf{A_3} := \mathsf{preimage\ f_1}|_{\mathsf{A}}\ \{\mathsf{false}\} \\ &\qquad\qquad\ \ \mathsf{f} := \lambda \mathsf{x}.\, \mathsf{if}\ (\mathsf{f_1\ x})\ (\mathsf{f_2\ x})\ (\mathsf{f_3\ x}) \\ &\qquad\qquad \mathsf{A}' := \mathsf{domain}_\perp\ \mathsf{f}\ (\mathsf{A_2} \uplus \mathsf{A_3}) \\ &\qquad\quad \mathsf{in}\ \ \mathsf{f}|_{\mathsf{A}'} \end{aligned} \tag{20}$$

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

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

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

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

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

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

$$\begin{aligned}&\mathsf{lift_{map}} : (\mathsf{X} \Rightarrow \mathsf{Y_\perp}) \Rightarrow (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y})\\ &\mathsf{lift_{map}}\ \mathsf{f}\ \mathsf{A} \ := \ \{\langle \mathsf{x}, \mathsf{y}\rangle \in \mathsf{f}|_{\mathsf{A}} \mid \mathsf{y} \neq \perp\}\end{aligned}$$

Figure 4: Mapping arrow definitions.

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

$$\begin{aligned}&\mathsf{lift_{map}}\ (\mathsf{ifte_\perp}\ \mathsf{f_1}\ \mathsf{f_2}\ \mathsf{f_3})\ \mathsf{A} \qquad (21)\\ &\quad \equiv \ \mathsf{let}\ \ \mathsf{g_1} := \mathsf{lift_{map}}\ \mathsf{f_1}\ \mathsf{A}\\ &\qquad\qquad\ \ \mathsf{g_2} := \mathsf{lift_{map}}\ \mathsf{f_2}\ (\mathsf{preimage}\ \mathsf{g_1}\ \{\mathsf{true}\})\\ &\qquad\qquad\ \ \mathsf{g_3} := \mathsf{lift_{map}}\ \mathsf{f_3}\ (\mathsf{preimage}\ \mathsf{g_1}\ \{\mathsf{false}\})\\ &\qquad\qquad\ \ \mathsf{A'} := (\mathsf{domain}\ \mathsf{g_2}) \uplus (\mathsf{domain}\ \mathsf{g_3})\\ &\qquad\quad\ \mathsf{in}\ \ \lambda \mathsf{x} \in \mathsf{A'}.\, \mathsf{if}\ (\mathsf{x} \in \mathsf{domain}\ \mathsf{g_2})\ (\mathsf{g_2}\ \mathsf{x})\ (\mathsf{g_3}\ \mathsf{x})\\ &\quad \equiv \ \mathsf{let}\ \ \mathsf{g_1} := \mathsf{lift_{map}}\ \mathsf{f_1}\ \mathsf{A}\\ &\qquad\qquad\ \ \mathsf{g_2} := \mathsf{lift_{map}}\ \mathsf{f_2}\ (\mathsf{preimage}\ \mathsf{g_1}\ \{\mathsf{true}\})\\ &\qquad\qquad\ \ \mathsf{g_3} := \mathsf{lift_{map}}\ \mathsf{f_3}\ (\mathsf{preimage}\ \mathsf{g_1}\ \{\mathsf{false}\})\\ &\qquad\quad\ \mathsf{in}\ \ \mathsf{g_2} \uplus_{\mathsf{map}} \mathsf{g_3}\end{aligned}$$

Substituting $\mathsf{g_1}$ for $\mathsf{lift_{map}}\ \mathsf{f_1}$, $\mathsf{g_2}$ for $\mathsf{lift_{map}}\ \mathsf{f_2}$, and $\mathsf{g_3}$ for $\mathsf{lift_{map}}\ \mathsf{f_3}$ gives a definition for $\mathsf{ifte_{map}}$ (Figure 4) for which (16) holds.

5.5 Case: Laziness

Starting with the left side of (17), we first expand definitions:

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

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

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

Substituting $\mathsf{g}\ 0$ for $\mathsf{lift_{map}}\ (\mathsf{f}\ 0)$ gives a definition for $\mathsf{lazy_{map}}$ (Figure 4) for which (17) holds.

5.6 Theorems

Theorem 5.6.1 (mapping arrow correctness)**.** $\mathsf{lift_{map}}$ *is an arrow+choice homomorphism.*

Proof. By construction. □

The following are easy consequences of the fact that $\mathsf{lift_{map}}$ is a homomorphism.

Corollary 5.6.2. $\mathsf{arr_{map}}$*,* $(\&\&\&_{\mathsf{map}})$ *and* $(\ggg_{\mathsf{map}})$ *define an arrow. With* $\mathsf{ifte_{map}}$ *and* $\mathsf{lazy_{map}}$*, they define an arrow+choice.*

Corollary 5.6.3. *If* $[\![e]\!]_\perp : \mathsf{X} \Rightarrow \mathsf{Y_\perp}$*, then* $\mathsf{lift_{map}}\ [\![e]\!]_\perp \equiv [\![e]\!]_{\mathsf{map}}$.

6. 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

$$\mathsf{X} \underset{\mathsf{pre}}{\rightleftharpoons} \mathsf{Y} \ ::= \ \langle \mathsf{Set}\ \mathsf{Y}, \mathsf{Set}\ \mathsf{Y} \Rightarrow \mathsf{Set}\ \mathsf{X}\rangle \qquad (22)$$

Converting a mapping to a lazy preimage mapping:

$$\begin{aligned}&\mathsf{pre} : (\mathsf{X} \rightharpoonup \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightleftharpoons} \mathsf{Y})\\ &\mathsf{pre}\ \mathsf{g} \ := \ \mathsf{let}\ \ \mathsf{Y'} := \mathsf{range}\ \mathsf{g}\\ &\qquad\qquad\qquad\ \ \mathsf{p} := \lambda \mathsf{B}.\, \mathsf{preimage}\ \mathsf{g}\ \mathsf{B}\\ &\qquad\qquad\ \ \mathsf{in}\ \ \langle \mathsf{Y'}, \mathsf{p}\rangle\end{aligned} \qquad (23)$$

Applying a preimage mapping to any subset of its codomain:

$$\begin{aligned}&\mathsf{pre\text{-}ap} : (\mathsf{X} \underset{\mathsf{pre}}{\rightleftharpoons} \mathsf{Y}) \Rightarrow \mathsf{Set}\ \mathsf{Y} \Rightarrow \mathsf{Set}\ \mathsf{X}\\ &\mathsf{pre\text{-}ap}\ \langle \mathsf{Y'}, \mathsf{p}\rangle\ \mathsf{B} \ := \ \mathsf{p}\ (\mathsf{B} \cap \mathsf{Y'})\end{aligned} \qquad (24)$$

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 6.0.4. *Let* $\mathsf{g} \in \mathsf{X} \rightharpoonup \mathsf{Y}$*. For all* $\mathsf{B} \subseteq \mathsf{Y}$ *and* $\mathsf{Y'}$ *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 6.0.5 (pre-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. Apply Lemma 6.0.4 with $\mathsf{Y'} = \mathsf{range}\ \mathsf{g}$. □

Figure 5 defines more operations on preimage mappings, including pairing, composition, and disjoint union operations corresponding to the mapping operations in Figure 1. 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 6.0.6. *Two preimage mappings* $h_1 : X \underset{pre}{\rightharpoonup} Y$ *and* $h_2 : X \underset{pre}{\rightharpoonup} Y$ *are equivalent, or* $h_1 \equiv h_2$*, when* $\text{pre-ap}\ h_1\ B = \text{pre-ap}\ h_2\ B$ *for all* $B \subseteq Y$*.*

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

6.1 Preimage Mapping Pairing

XXX: moar wurds in this section

Lemma 6.1.1 (preimage distributes over $\langle\cdot,\cdot\rangle_{map}$ 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* $B_2 \subseteq Y_2$*,* $\text{preimage}\ \langle g_1, g_2\rangle_{map}\ (B_1 \times B_2) = (\text{preimage}\ g_1\ B_1) \cap (\text{preimage}\ g_2\ B_2)$*.*

Theorem 6.1.2 (pre distributes over $\langle\cdot,\cdot\rangle_{map}$). *Let* $g_1 \in X \rightharpoonup Y_1$ *and* $g_2 \in X \rightharpoonup Y_2$*. Then* $\text{pre}\ \langle g_1, g_2\rangle_{map} \equiv \langle \text{pre}\ g_1, \text{pre}\ g_2\rangle_{pre}$*.*

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

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

□

6.2 Preimage Mapping Composition

XXX: moar wurds in this section

Lemma 6.2.1 (preimage distributes over $(\circ_{map})$). *Let* $g_1 \in X \rightharpoonup Y$ *and* $g_2 \in Y \rightharpoonup Z$*. For all* $C \subseteq Z$*,* $\text{preimage}\ (g_2 \circ_{map} g_1)\ C = \text{preimage}\ g_1\ (\text{preimage}\ g_2\ C)$*.*

Theorem 6.2.2 (pre distributes over $(\circ_{map})$). *Let* $g_1 \in X \rightharpoonup Y$ *and* $g_2 \in Y \rightharpoonup Z$*. Then* $\text{pre}\ (g_2 \circ_{map} g_1) \equiv (\text{pre}\ g_2) \circ_{pre} (\text{pre}\ g_1)$*.*

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

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

□

6.3 Preimage Mapping Disjoint Union

XXX: moar wurds in this section

Lemma 6.3.1 (preimage distributes over $(\uplus_{map})$). *Let* $g_1 \in X \rightharpoonup Y$ *and* $g_2 \in X \rightharpoonup Y$ *be disjoint mappings. For all* $B \subseteq Y$*,* $\text{preimage}\ (g_1 \uplus_{map} g_2)\ B = (\text{preimage}\ g_1\ B) \uplus (\text{preimage}\ g_2\ B)$*.*

Theorem 6.3.2 (pre distributes over $(\uplus_{map})$). *Let* $g_1 \in X \rightharpoonup Y$ *and* $g_2 \in X \rightharpoonup Y$ *have disjoint domains. Then* $\text{pre}\ (g_1 \uplus_{map} g_2) \equiv (\text{pre}\ g_1) \uplus_{pre} (\text{pre}\ g_2)$*.*

Proof. Let $Y_1' := \text{range}\ g_1$ and $Y_2' := \text{range}\ g_2$. Starting from the right side, for all $B \subseteq Y$,

$$\begin{aligned}
&\text{pre-ap}\ ((\text{pre}\ g_1) \uplus_{pre} (\text{pre}\ g_2))\ B \\
&\quad = \text{let}\ \ Y' := Y_1' \cup Y_2' \\
&\qquad\qquad h := \lambda B.\ (\text{pre-ap}\ (\text{pre}\ g_1)\ B) \uplus (\text{pre-ap}\ (\text{pre}\ g_2)\ B) \\
&\qquad\ \ \text{in}\ \ h\ (B \cap Y') \\
&\quad = (\text{pre-ap}\ (\text{pre}\ g_1)\ (B \cap (Y_1' \cup Y_2'))) \uplus \\
&\qquad\ (\text{pre-ap}\ (\text{pre}\ g_2)\ (B \cap (Y_1' \cup Y_2'))) \\
&\quad = (\text{preimage}\ g_1\ (B \cap (Y_1' \cup Y_2'))) \uplus \\
&\qquad\ (\text{preimage}\ g_2\ (B \cap (Y_1' \cup Y_2'))) \\
&\quad = \text{preimage}\ (g_1 \uplus_{map} g_2)\ (B \cap (Y_1' \cup Y_2')) \\
&\quad = \text{preimage}\ (g_1 \uplus_{map} g_2)\ B \\
&\quad = \text{pre-ap}\ (\text{pre}\ (g_1 \uplus_{map} g_2))\ B
\end{aligned}$$

□

7. Deriving the Preimage Arrow

XXX: intro

$$X \underset{pre}{\rightsquigarrow} Y\ \ ::=\ \ \text{Set}\ X \Rightarrow (X \underset{pre}{\rightharpoonup} Y) \tag{25}$$

$$\begin{aligned}
&\text{lift}_{pre} : (X \underset{map}{\rightsquigarrow} Y) \Rightarrow (X \underset{pre}{\rightsquigarrow} Y) \\
&\text{lift}_{pre}\ g\ A\ \ :=\ \ \text{pre}\ (g\ A)
\end{aligned} \tag{26}$$

Definition 7.0.3 (Preimage arrow equivalence). *Two preimage arrow computations* $h_1 : X \underset{pre}{\rightsquigarrow} Y$ *and* $h_2 : X \underset{pre}{\rightsquigarrow} Y$ *are equivalent, or* $h_1 \equiv h_2$*, when* $h_1\ A \equiv h_2\ A$ *for all* $A \subseteq X$*.*

As with arr_{map}, defining arr_{pre} as a composition meets (13). The following subsections derive $(\&\&\&_{pre})$, $(\ggg_{pre})$, ifte_{pre} and lazy_{pre} from their corresponding mapping arrow combinators, in a way that ensures lift_{pre} is an arrow+choice homomorphism from the mapping arrow to the preimage arrow. Figure 6 contains the resulting definitions.

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

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

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

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

$$\begin{aligned}&\langle \cdot, \cdot \rangle_{\mathsf{pre}} : (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoondown} \mathsf{Y_1}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoondown} \mathsf{Y_2}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoondown} \mathsf{Y_1} \times \mathsf{Y_2})\\ &\langle \langle \mathsf{Y'_1}, \mathsf{p_1} \rangle, \langle \mathsf{Y'_2}, \mathsf{p_2} \rangle \rangle_{\mathsf{pre}} \ := \ \mathsf{let}\ \ \mathsf{Y'} := \mathsf{Y'_1} \times \mathsf{Y'_2}\\ &\qquad\qquad\qquad\qquad \mathsf{p} := \lambda \mathsf{B}. \bigcup_{\langle \mathsf{y_1}, \mathsf{y_2} \rangle \in \mathsf{B}} (\mathsf{p_1}\ \{\mathsf{y_1}\}) \cap (\mathsf{p_2}\ \{\mathsf{y_2}\})\\ &\qquad\qquad\qquad \mathsf{in}\ \ \langle \mathsf{Y'}, \mathsf{p} \rangle\end{aligned}$$

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

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

Figure 5: Lazy preimage mappings and operations.

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

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

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

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

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

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

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

Figure 6: Preimage arrow definitions.

7.1 Case: Pairing

Starting with the left side of (14), we expand definitions, apply Theorem 6.1.2, and rewrite in terms of $\mathsf{lift_{pre}}$:

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

Substituting $\mathsf{h_1}$ for $\mathsf{lift_{pre}}\ \mathsf{g_1}$ and $\mathsf{h_2}$ for $\mathsf{lift_{pre}}\ \mathsf{g_2}$, and removing the application of pre-ap from both sides of the equivalence gives a definition of $(\&\&\&_{\mathsf{pre}})$ (Figure 6) for which (14) holds.

7.2 Case: Composition

Starting with the left side of (15), we expand definitions, apply Theorem 6.2.2 and rewrite in terms of $\mathsf{lift_{pre}}$:

$$\begin{aligned}&\mathsf{pre\text{-}ap}\ (\mathsf{lift_{pre}}\ (\mathsf{g_1} \ggg_{\mathsf{map}} \mathsf{g_2})\ \mathsf{A})\ \mathsf{C}\\ &\quad\equiv \mathsf{let}\ \ \mathsf{g'_1} := \mathsf{g_1\ A}\\ &\qquad\qquad \mathsf{g'_2} := \mathsf{g_2}\ (\mathsf{range\ g'_1})\\ &\qquad\ \ \mathsf{in}\ \ \mathsf{pre\text{-}ap}\ (\mathsf{pre}\ (\mathsf{g'_2} \circ_{\mathsf{map}} \mathsf{g'_1}))\ \mathsf{C}\\ &\quad\equiv \mathsf{let}\ \ \mathsf{g'_1} := \mathsf{g_1\ A}\\ &\qquad\qquad \mathsf{g'_2} := \mathsf{g_2}\ (\mathsf{range\ g'_1})\\ &\qquad\ \ \mathsf{in}\ \ \mathsf{pre\text{-}ap}\ ((\mathsf{pre\ g'_1}) \circ_{\mathsf{pre}} (\mathsf{pre\ g'_2}))\ \mathsf{C}\\ &\quad\equiv \mathsf{let}\ \ \mathsf{h_1} := \mathsf{lift_{pre}\ g_1\ A}\\ &\qquad\qquad \mathsf{h_2} := \mathsf{lift_{pre}\ g_2}\ (\mathsf{pre\text{-}range\ h_1})\\ &\qquad\ \ \mathsf{in}\ \ \mathsf{pre\text{-}ap}\ (\mathsf{h_2} \circ_{\mathsf{pre}} \mathsf{h_1})\ \mathsf{C}\end{aligned} \tag{27}$$

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

7.3 Case: Conditional

Starting with the left side of (16), we expand terms, apply Theorem 6.3.2, rewrite in terms of $\mathsf{lift_{pre}}$, and apply Theorem 6.0.5 in the definitions of h_2 and h_3:

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

Substituting h_1 for $\mathsf{lift_{pre}}\ \mathsf{g}_1$, h_2 for $\mathsf{lift_{pre}}\ \mathsf{g}_2$ and h_3 for $\mathsf{lift_{pre}}\ \mathsf{g}_3$, and removing the application of pre-ap from both sides of the equivalence gives a definition of $\mathsf{ifte_{pre}}$ (Figure 6) for which (16) holds.

7.4 Case: Laziness

Starting with the left side of (17), expand definitions, distribute pre over the branches of if, and rewrite in terms of $\mathsf{lift_{pre}}\ (\mathsf{g}\ 0)$:

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

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 (17) holds.

7.5 Theorems

Theorem 7.5.1 (preimage arrow correctness)**.** $\mathsf{lift_{pre}}$ *is an arrow+choice homomorphism.*

Proof. By construction. ☐

The following are easy consequences of the fact that $\mathsf{lift_{pre}}$ is a homomorphism.

Corollary 7.5.2. $\mathsf{arr_{pre}}$, $(\&\&\&_{\mathsf{pre}})$ *and* $(\ggg_{\mathsf{pre}})$ *define an arrow. With* $\mathsf{ifte_{pre}}$ *and* $\mathsf{lazy_{pre}}$, *they define an arrow+choice.*

Corollary 7.5.3. *If* $[\![e]\!]_{\mathsf{map}} : \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}$, *then for all* $\mathsf{A} \subseteq \mathsf{X}$ *and* $\mathsf{B} \subseteq \mathsf{Y}$, $\mathsf{preimage}\ ([\![e]\!]_{\mathsf{map}}\ \mathsf{A})\ \mathsf{B} \equiv \mathsf{pre\text{-}ap}\ ([\![e]\!]_{\mathsf{pre}}\ \mathsf{A})\ \mathsf{B}$.

In other words, $[\![e]\!]_{\mathsf{pre}}$ returns a computation that correctly computes preimages under $[\![e]\!]_{\mathsf{map}}$ or, again by homomorphism, $[\![e]\!]_{\perp}$.

8. Approximating Preimage Arrow

XXX: $\mathsf{arr_{pre}}$ is generally uncomputable, but we don't need that many lifts; Figure 7 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:

- $\mathsf{arr_{pre}}$ (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 7:

- $\mathsf{id_{pre}}$: 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. Preimages of Partial Functions

Probabilistic functions that may diverge, but converge with probability 1, are common. They often come up when practitioners want to build data with random size or structure, but they may show up in simpler circumstances as well.

Suppose $\mathsf{boolean\ p}$ returns true with probability p, by retrieving a number $\mathsf{x_i} \in [0,1]$ from a uniform random source x and returning true when $\mathsf{x_i} < \mathsf{p}$. The following recursive function, which defines the well-known **geometric distribution**, counts the number of times $\mathsf{boolean\ p}$ is false:

$$\mathsf{geometric\ p}\ :=\ \mathsf{if}\ (\mathsf{boolean\ p})\ 0\ (1 + \mathsf{geometric\ p}) \tag{28}$$

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

Suppose we interpret (28) as $\mathsf{h} : \mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathbb{N}$, a preimage arrow computation from random sources in X to natural numbers. Given a uniform probability measure $\mathsf{P} \in \mathcal{P}\ \mathsf{X} \rightharpoonup [0,1]$, we could compute the probability of some output set $\mathsf{N} \subseteq \mathbb{N}$ using $\mathsf{P}\ (\mathsf{h}\ \mathsf{A}\ \mathsf{N})$, where $\mathsf{A} \subseteq \mathsf{X}$, $\mathsf{P}\ \mathsf{A} = 1$ and $\mathsf{h}\ \mathsf{A}$ converges.

We have three hurdles to overcome:

1. Determining how $\mathsf{boolean\ p}$ retrieves unique real numbers $\mathsf{x_i}$ from x.
2. Ensuring that each $\mathsf{x} \in \mathsf{X}$ contains enough random numbers.
3. Ensuring that $\mathsf{h}\ \mathsf{A}$ converges.

For hurdle 3, we will ensure that $\mathsf{h}\ \mathsf{A}$ converges for any $\mathsf{A} \subseteq \mathsf{X}$. Hurdle 2 is easy: we make each x infinite. Hurdle 1 relies on an expression indexing scheme for unrolled programs, which we develop to address 3.

9.1 Expression Indexing

XXX: overall idea: define an arrow transformer that assigns every combinator a unique index; define another arrow transformer that threads

threads an infinite vector of random numbers through its computations

Definition 9.1.1 (binary indexing scheme)**.** *Let* J *be an index set and* $\mathsf{j}_0 \in \mathsf{J}$ *a distinguished element. Let* $\mathsf{left} : \mathsf{J} \Rightarrow \mathsf{J}$ *and* $\mathsf{right} : \mathsf{J} \Rightarrow \mathsf{J}$ *be total functions. If for any 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****.*

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

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

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

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

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

Figure 7: Specific instances of $\mathsf{arr_{pre}}\ \mathsf{f}$

For example, let J be the set of lists of booleans, $\mathsf{j_0}$ the empty list, and define

$$\begin{aligned} \mathsf{left}\ \mathsf{xs} \ &:= \ \langle \mathsf{true}, \mathsf{xs} \rangle \\ \mathsf{right}\ \mathsf{xs} \ &:= \ \langle \mathsf{false}, \mathsf{xs} \rangle \end{aligned} \tag{29}$$

Alternatively, let $\mathsf{J} := (0,1) \cap \mathbb{Q}$, $\mathsf{j_0} := \frac{1}{2}$ and

$$\begin{aligned} \mathsf{left}\ (\mathsf{p}/\,\mathsf{q}) \ &:= \ (\mathsf{p} - \tfrac{1}{2})/\,\mathsf{q} \\ \mathsf{right}\ (\mathsf{p}/\,\mathsf{q}) \ &:= \ (\mathsf{p} + \tfrac{1}{2})/\,\mathsf{q} \end{aligned} \tag{30}$$

If J ... then $\mathsf{x} \in \mathsf{J} \to [0,1]$

9.2 Applicative Store-Passing

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.