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—essentially infinite hash tables—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} [1], 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 **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} programs without alteration. Further, proofs about λ_{ZFC} objects 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 $\mathcal{P}\ \mathsf{A}$, and is the type of *subsets* of A. Similarly, the type $\langle \mathsf{A}, \mathsf{B} \rangle$ is equivalent to 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 a single inaccessible cardinal.

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

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

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 is defined by extending 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 by (conservatively) extending set theory with higher-order functions in the domain of discourse, and defining a reduction relation. Thus, all our reasoning about λ_{ZFC} programs is done in first-order logic.

While λ_{ZFC} does not have an equality primitive, extensional equality can be recovered internally using the $(\in)$ primitive. *Internal* extensional equality is defined by

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

which means

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

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

External extensional equality is usually too fine-grained—moreso than the internal $(=)$. For example, the terms $\{(\lambda x.\, x)\ 1,\ 1\}$ and $\{1\}$ are unequal, even though the term $\{(\lambda x.\, x)\ 1,\ 1\} = \{1\}$ reduces to true. We will therefore not use external extensional equality, but instead understand truth statements such as "$e_1 = e_2$" as shorthand for "$e_1 = e_2$ reduces to true."

When we do not want an equality statement to require or guarantee convergence, we use a slightly weaker equivalence.

Definition 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 equality, such as applicative or logical bisimilarity, but we do not want to deal with too much dissonance between external and internal equality. We therefore introduce type-specific equivalence as needed.

2.2 Additional Functions and Forms

XXX: syntactic sugar: automatic currying, matching, sectioning rules, set comprehensions, cardinality, indexed unions

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

XXX: $A \rightharpoonup B$ is the set of partial mappings...

XXX: put Englishy word thingies around the following so it's not just a wall of meaningless code:

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

$$\begin{aligned} &\mathsf{restrict} : (A \Rightarrow B) \Rightarrow \mathcal{P}\ A \Rightarrow (A \rightharpoonup B) \\ &\mathsf{restrict}\ f\ A' \ :=\ \mathsf{image}\ (\lambda x.\, (x, f\ x))\ A' \end{aligned} \tag{5}$$

$$\begin{aligned} f|_A \ &:\equiv\ \mathsf{restrict}\ f\ A \\ \lambda x \in A.\, e \ &:\equiv\ (\lambda x.\, e)|_A \end{aligned} \tag{6}$$

$$\begin{aligned} &\mathsf{domain} : (A \rightharpoonup B) \Rightarrow \mathsf{Set}\ A \\ &\mathsf{domain}\ f \ :=\ \mathsf{image}\ \mathsf{fst}\ f \end{aligned} \tag{7}$$

$$\begin{aligned} &\mathsf{range} : (A \rightharpoonup B) \Rightarrow \mathsf{Set}\ B \\ &\mathsf{range}\ f \ :=\ \mathsf{image}\ \mathsf{snd}\ f \end{aligned} \tag{8}$$

$$\begin{aligned} &\mathsf{preimage} : (A \rightharpoonup B) \Rightarrow \mathsf{Set}\ B \Rightarrow \mathsf{Set}\ A \\ &\mathsf{preimage}\ f\ B \ :=\ \{x \in \mathsf{domain}\ f \mid f\ x \in B\} \end{aligned} \tag{9}$$

$$\begin{aligned} &(\circ_{\mathsf{map}}) : (Y \rightharpoonup Z) \Rightarrow (X \rightharpoonup Y) \Rightarrow (X \rightharpoonup Z) \\ &g_2 \circ_{\mathsf{map}} g_1 \ :=\ \mathsf{let}\ A := \mathsf{preimage}\ g_1\ (\mathsf{domain}\ g_2) \\ &\qquad\qquad \mathsf{in}\ \ \lambda x \in A.\, g_2\ (g_1\ x) \end{aligned} \tag{10}$$

$$\begin{aligned} &\langle \cdot, \cdot \rangle_{\mathsf{map}} : (X \rightharpoonup Y) \Rightarrow (X \rightharpoonup Z) \Rightarrow (X \rightharpoonup Y \times Z) \\ &\langle g_1, g_2 \rangle_{\mathsf{map}} \ :=\ \mathsf{let}\ A := (\mathsf{domain}\ g_1) \cap (\mathsf{domain}\ g_2) \\ &\qquad\qquad \mathsf{in}\ \ \lambda x \in A.\, \langle g_1\ x, g_2\ x \rangle \end{aligned} \tag{11}$$

$$\begin{aligned} &(\uplus_{\mathsf{map}}) : (X \rightharpoonup Y) \Rightarrow (X \rightharpoonup Y) \Rightarrow (X \rightharpoonup Y) \\ &g_1 \uplus_{\mathsf{map}} g_2 \ :=\ \mathsf{let}\ A := (\mathsf{domain}\ g_1) \uplus (\mathsf{domain}\ g_2) \\ &\qquad\qquad \mathsf{in}\ \ g_1 \cup g_2 \end{aligned} \tag{12}$$

2.3 Ordinal Numbers and Transfinite Recursion

The **ordinal numbers**, or values of type Ord, are an extension of the natural numbers to infinite lengths. Ordinals are typically defined as the smallest sets that contain all their predecessors; e.g. the first four are

$$\begin{aligned} 0 \ &:=\ \varnothing & 2 \ &:=\ \{0, 1\} \\ 1 \ &:=\ \{0\} & 3 \ &:=\ \{0, 1, 2\} \end{aligned} \tag{13}$$

The smallest infinite ordinal ω is the set of all finite ordinals. Other infinite ordinals are defined in terms of ω:

$$\begin{aligned} \omega \ &:=\ \{0, 1, 2, 3, ...\} \\ \omega + 1 \ &:=\ \{0, 1, 2, 3, ..., \omega\} \\ \omega + 2 \ &:=\ \{0, 1, 2, 3, ..., \omega, \omega + 1\} \\ \omega + \omega \ &:=\ \{0, 1, 2, 3, ..., \omega, \omega + 1, \omega + 2, ...\} \end{aligned} \tag{14}$$

The above sets are all countable, meaning that their cardinality is ω. Generally, any ordinal α for which $\alpha = |\alpha|$ is also called a **cardinal number**.

Ordinals are totally ordered by membership; i.e. $\beta < \alpha$ is equivalent to $\beta \in \alpha$. Ordinals with an immediate predecessor (such as 3 and $\omega + 2$) are called **successor ordinals**. Nonzero ordinals without an immediate predecessor (roughly, those whose literal representations end in "...") are called **limit ordinals**.

Limit ordinals allow writing terminating functions that recur infinitely many times. Suppose we wanted a function that recursively generates all the integral successors of 0.5.

Consider this first attempt, which can be written in any Turing-equivalent language:

$$\begin{aligned}&\mathsf{succs} : \omega \Rightarrow \mathsf{Set}\ \mathbb{R} \Rightarrow \mathsf{Set}\ \mathbb{R}\\&\mathsf{succs\ n\ A}\ :=\\&\quad \mathsf{case\ \ n}\\&\qquad \mathsf{0} \quad \Longrightarrow\ \mathsf{A}\\&\qquad \mathsf{m}+1 \ \Longrightarrow\ \mathsf{A} \cup (\mathsf{image}\ (+\ 1.0)\ (\mathsf{succs\ m\ A}))\end{aligned} \tag{15}$$

This unfold over finite ordinals generates prefixes such as

$$\begin{aligned}\mathsf{succs\ 0}\ \{0.5\} &= \{0.5\}\\ \mathsf{succs\ 1}\ \{0.5\} &= \{0.5, 1.5\}\\ \mathsf{succs\ 2}\ \{0.5\} &= \{0.5, 1.5, 2.5\}\end{aligned} \tag{16}$$

but will never generate the full set of integral successors.

To close $\{0.5\}$ under increment, we can use **transfinite recursion**: unfolding over ordinals as above, but using an additional inductive case for limit ordinals:

$$\begin{aligned}&\mathsf{succs} : \mathsf{Ord} \Rightarrow \mathsf{Set}\ \mathbb{R} \Rightarrow \mathsf{Set}\ \mathbb{R}\\&\mathsf{succs}\ \alpha\ \mathsf{A}\ :=\\&\quad \mathsf{case}\ \ \alpha\\&\qquad \mathsf{0} \quad \Longrightarrow\ \mathsf{A}\\&\qquad \beta+1 \ \Longrightarrow\ \mathsf{A} \cup (\mathsf{image}\ (+\ 1.0)\ (\mathsf{succs}\ \beta\ \mathsf{A}))\\&\qquad \mathsf{else} \ \Longrightarrow\ \bigcup_{\beta<\alpha} \mathsf{succs}\ \beta\ \mathsf{A}\end{aligned} \tag{17}$$

With this, we can compute the closure as desired:

$$\begin{aligned}&\mathsf{succs}\ \omega\ \{0.5\}\\&\quad = (\mathsf{succs\ 0}\ \{0.5\}) \cup (\mathsf{succs\ 1}\ \{0.5\}) \cup \cdots\\&\quad = \{0.5, 1.5, 2.5, 3.5, ...\}\end{aligned} \tag{18}$$

The function terminates because each branch, though unbounded, is finite in length. As with strict data structures, the shape of the computation is "infinitely wide" but not "infinitely tall."

The succs function is a special case of a powerful general closure operator defined by

$$\begin{aligned}&\mathsf{close} : (\mathsf{Set\ x} \Rightarrow \mathsf{Set\ x}) \Rightarrow \mathsf{Ord} \Rightarrow \mathsf{Set\ x} \Rightarrow \mathsf{Set\ x}\\&\mathsf{close\ f}\ \alpha\ \mathsf{A}\ :=\ \mathsf{case}\ \ \alpha\\&\qquad\qquad \mathsf{0} \quad \Longrightarrow\ \mathsf{A}\\&\qquad\qquad \beta+1 \ \Longrightarrow\ \mathsf{A} \cup (\mathsf{f}\ (\mathsf{close\ f}\ \beta\ \mathsf{A}))\\&\qquad\qquad \mathsf{else} \ \Longrightarrow\ \bigcup_{\beta<\alpha} \mathsf{close\ f}\ \beta\ \mathsf{A}\end{aligned} \tag{19}$$

With this, succs := close (image (+ 1.0)).

The close function can construct the least fixpoint of any monotone set function, if a fixpoint exists. Such fixpoints include the languages of context-free grammars and the reduction relations defined by inductive rules in operational semantics. (For almost all of these, only countably many iterations is sufficient.) We will use close to construct σ-algebras and preimages, two of the main objects of study in measure theory.

3. Measure Theory

XXX: overview, with analogies to topology

3.1 Sigma-Algebras

XXX: motivate and define informally

Formally, using the following functions to generate $\varnothing$, complements, and countable unions from families of any type x:

$$\begin{aligned}&\sigma\text{-comps} : \mathsf{Set}\ (\mathsf{Set\ x}) \Rightarrow \mathsf{Set}\ (\mathsf{Set\ x})\\&\sigma\text{-comps}\ \mathcal{A}\ :=\ \{\mathsf{A} \backslash \mathsf{A}' \mid \mathsf{A}, \mathsf{A}' \in \mathcal{A}\}\end{aligned} \tag{20}$$

$$\begin{aligned}&\sigma\text{-unions} : \mathsf{Set}\ (\mathsf{Set\ x}) \Rightarrow \mathsf{Set}\ (\mathsf{Set\ x})\\&\sigma\text{-unions}\ \mathcal{A}\ :=\ \{\bigcup \mathcal{A}' \mid \mathcal{A}' \subseteq \mathcal{A} \wedge |\mathcal{A}'| \leq \omega\}\end{aligned} \tag{21}$$

$$\begin{aligned}&\sigma\text{-ops} : \mathsf{Set}\ (\mathsf{Set\ x}) \Rightarrow \mathsf{Set}\ (\mathsf{Set\ x})\\&\sigma\text{-ops}\ \mathcal{A}\ :=\ \{\varnothing\} \cup (\sigma\text{-comps}\ \mathcal{A}) \cup (\sigma\text{-unions}\ \mathcal{A})\end{aligned} \tag{22}$$

the following function identifies σ-algebras:

$$\begin{aligned}&\sigma\text{-algebra?} : \mathsf{Set}\ (\mathsf{Set\ x}) \Rightarrow \mathsf{Bool}\\&\sigma\text{-algebra?}\ \mathcal{A}\ :=\ (\sigma\text{-ops}\ \mathcal{A}) \subseteq \mathcal{A}\end{aligned} \tag{23}$$

Clearly, $\mathcal{P}\ \mathsf{A}$ for any set A is a σ-algebra. Unfortunately, this σ-algebra is "too large"—a concept we will formalize when discussing measures. (XXX: cover earlier, in motivation, using Banach-Tarski paradox?)

The **trace** of a family of subsets $\mathcal{A}$ with a set A is the result of intersecting every $\mathsf{A}' \in \mathcal{A}$ with A:

$$\begin{aligned}&\mathsf{trace} : \mathsf{Set}\ (\mathsf{Set\ x}) \Rightarrow \mathsf{Set\ x} \Rightarrow \mathsf{Set}\ (\mathsf{Set\ x})\\&\mathsf{trace}\ \mathcal{A}\ \mathsf{A}\ :=\ \{\mathsf{A} \cap \mathsf{A}' \mid \mathsf{A}' \in \mathcal{A}\}\\&\mathcal{A}|_{\mathsf{A}}\ :=\ \mathsf{trace}\ \mathcal{A}\ \mathsf{A}\end{aligned} \tag{24}$$

Lemma 1 (traces of σ-algebras are σ-algebras)**.** *For any σ-algebra $\mathcal{A}$ and set* A, trace $\mathcal{A}$ A *is a σ-algebra on* $\mathsf{A} \cap \bigcup \mathcal{A}$.

Lemma 2 (σ-close distributes over trace)**.** *For any family of subsets $\mathcal{A}$ and set* A, σ-close (trace $\mathcal{A}$ A) = trace (σ-close $\mathcal{A}$) A.

3.2 Measurable Mappings

XXX: define and characterize measurable mappings

XXX: images of measurable sets under measurable functions are not always measurable; e.g. projections

3.3 Generated Sigma-Algebras

Often, σ-algebras are too complicated to work with directly. In such cases, we reason about them in terms of **generating families**: simpler families of sets that, when closed under σ-algebra operations, are σ-algebras with nice properties.

To generate σ-algebras, first note that σ-ops is monotone and has an upper bound (the powerset σ-algebra). Therefore, close can generate a least fixpoint from it:

$$\begin{aligned}&\sigma\text{-close} : \mathsf{Set}\ (\mathsf{Set\ x}) \Rightarrow \mathsf{Set}\ (\mathsf{Set\ x})\\&\sigma\text{-close}\ :=\ \mathsf{close}\ \sigma\text{-ops}\ \omega_1\end{aligned} \tag{25}$$

Here, ω_1 is the **first uncountable ordinal**, or the least ordinal containing every countable ordinal.[4]

Lemma 3 (generated σ-algebras)**.** *Let $\mathcal{A}$ be a family of sets and $\mathcal{A}' := \sigma$-close $\mathcal{A}$. Then $\mathcal{A}'$ is the smallest σ-algebra for which $\mathcal{A} \subseteq \mathcal{A}'$.*

Perhaps the most well-studied generated σ-algebras are **Borel σ-algebras**: those generated from topologies. For example, if τ is the standard topology on $\mathbb{R}$, containing the open intervals and uncountable unions of open intervals, then σ-close τ is the Borel σ-algebra for that topology, containing all intervals (open, closed and half-open), and their countable unions and complements.

Other well-studied, generated σ-algebras are **product σ-algebras**: those generated from the rectangles of, or the pairwise products of sets from, other σ-algebras. For exam-

[4] λ_{ZFC} does not decide the continuum hypothesis; i.e. whether $\omega_1 = |\mathbb{R}|$.

ple, the product σ-algebra $\mathcal{A}_1 \otimes \mathcal{A}_2$ can be defined by

$$\begin{aligned}&(\boxtimes) : \mathsf{Set}\ (\mathsf{Set}\ \mathsf{x}) \Rightarrow \mathsf{Set}\ (\mathsf{Set}\ \mathsf{y}) \Rightarrow \mathsf{Set}\ (\mathsf{Set}\ \langle \mathsf{x}, \mathsf{y}\rangle)\\ &\mathcal{A}_1 \boxtimes \mathcal{A}_2 \ :=\ \{\mathsf{A}_1 \times \mathsf{A}_2 \mid \mathsf{A}_1 \in \mathcal{A}_1, \mathsf{A}_2 \in \mathcal{A}_2\}\end{aligned} \tag{26}$$

$$\begin{aligned}&(\otimes) : \mathsf{Set}\ (\mathsf{Set}\ \mathsf{x}) \Rightarrow \mathsf{Set}\ (\mathsf{Set}\ \mathsf{y}) \Rightarrow \mathsf{Set}\ (\mathsf{Set}\ \langle \mathsf{x}, \mathsf{y}\rangle)\\ &\mathcal{A}_1 \otimes \mathcal{A}_2 \ :=\ \sigma\text{-}\mathsf{close}\ (\mathcal{A}_1 \boxtimes \mathcal{A}_2)\end{aligned} \tag{27}$$

However, the following theorem is usually taken (somewhat less constructively) as the definition of $\mathcal{A}_1 \otimes \mathcal{A}_2$.

Lemma 4 (projection mappings are product-measurable)**.** *Let $\mathcal{A} := \mathcal{A}_1 \otimes \mathcal{A}_2$ and $\mathsf{A} := \bigcup \mathcal{A}$. Then $\mathcal{A}$ is the smallest σ-algebra for which $\mathsf{fst}|_{\mathsf{A}}$ is $\mathcal{A} - \mathcal{A}_1$-measurable and $\mathsf{snd}|_{\mathsf{A}}$ is $\mathcal{A} - \mathcal{A}_2$-measurable.*

3.4 Measures and Probabilities

Now that we have defined what sets can be measured, we can define what it means to measure them.

Definition 2 (Measure)**.** *A mapping $\mu \in \mathcal{P}\ \mathsf{X} \rightharpoonup [0, \infty)$ is a **measure** if*

- $\mathsf{domain}\ \mu$ *is a σ-algebra on* X.
- $\mu\ \varnothing = 0$.
- *For all $\mathcal{A}' \subseteq \mathcal{A}$ such that $|\mathcal{A}'| \leq \omega$, $\sum_{\mathsf{A} \in \mathcal{A}'} (\mu\ \mathsf{A}) = \mu\ (\bigcup \mathcal{A}')$.*

XXX: Probabilities of output sets are preimage measures: if $\mathsf{f} \in \mathsf{A} \rightharpoonup \mathsf{B}$ is $\mathcal{A} - \mathcal{B}$-measurable and $\mathbb{P} \in \mathcal{A} \to [0, 1]$, then

$$\mathbb{P}\ (\mathsf{preimage}\ \mathsf{f}\ \mathsf{B}) \tag{28}$$

is the probability of B.

XXX: more explanation, and an example

4. The Bottom Arrow

XXX: motivation:

- derive preimage arrow from something simple and obviously correct
- eventually define functions that may diverge using this arrow; use derivation to do the same with the preimage arrow
- will be implemented to run programs on domain samples

XXX: Fig. 1 defines the bottom arrow...

XXX: the standard Kleisli conversion of the Maybe monad (using a $\perp$ instead of Just and Maybe), simplified; arrow laws therefore hold (XXX: check terminology)

XXX: point out that $\mathsf{if}_\perp$ receives thunks, and remind readers that $1 = \{0\}$

5. Deriving the Mapping Arrow

XXX: intermediate step between the bottom and preimage arrows; will not be implemented (no approximation will be implemented, either); computations are in terms of mappings, on which we can apply theorems from measure theory directly

XXX: the type of mapping arrow computations

$$\mathsf{X} \mathrel{\tilde{\rightarrow}_{\mathsf{map}}} \mathsf{Y}\ ::=\ \mathsf{Set}\ \mathsf{X} \Rightarrow (\mathsf{X} \rightharpoonup \mathsf{Y}) \tag{29}$$

XXX: notice $\mathsf{X} \rightharpoonup \mathsf{Y}$, not $\mathsf{X} \rightharpoonup \mathsf{Y}_\perp$

XXX: motivate removal of bottom

Lifting a bottom arrow computation $\mathsf{f} : \mathsf{X} \Rightarrow \mathsf{Y}_\perp$ to the mapping arrow requires restricting 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 $\mathsf{arr}_{\mathsf{map}}$ in terms of $\mathsf{domain}_\perp$:

$$\begin{aligned}&\mathsf{domain}_\perp : (\mathsf{X} \Rightarrow \mathsf{Y}_\perp) \Rightarrow \mathsf{Set}\ \mathsf{X} \Rightarrow \mathsf{Set}\ \mathsf{X}\\ &\mathsf{domain}_\perp\ \mathsf{f}\ \mathsf{A} \ :=\ \mathsf{preimage}\ \mathsf{f}|_{\mathsf{A}}\ ((\mathsf{image}\ \mathsf{f}\ \mathsf{A}) \backslash \{\perp\})\end{aligned} \tag{30}$$

$$\begin{aligned}&\mathsf{arr}_{\mathsf{map}} : (\mathsf{X} \Rightarrow \mathsf{Y}_\perp) \Rightarrow (\mathsf{X} \mathrel{\tilde{\rightarrow}_{\mathsf{map}}} \mathsf{Y})\\ &\mathsf{arr}_{\mathsf{map}}\ \mathsf{f}\ \mathsf{A} \ :=\ \mathsf{let}\ \ \mathsf{A}' := \mathsf{domain}_\perp\ \mathsf{f}\ \mathsf{A}\\ &\qquad\qquad\quad \mathsf{in}\ \ \mathsf{f}|_{\mathsf{A}'}\end{aligned} \tag{31}$$

XXX: the default equality relation, which for λ_{ZFC} terms is alpha equivalence of reduced terms, will not do; need something more extensional

Definition 3 (Mapping arrow equivalence)**.** *Two mapping arrow computations $\mathsf{g}_1 : \mathsf{X} \mathrel{\tilde{\rightarrow}_{\mathsf{map}}} \mathsf{Y}$ and $\mathsf{g}_2 : \mathsf{X} \mathrel{\tilde{\rightarrow}_{\mathsf{map}}} \mathsf{Y}$ are equivalent, written $\mathsf{g}_1 \mathrel{\tilde{\equiv}_{\mathsf{map}}} \mathsf{g}_2$, when $\mathsf{g}_1\ \mathsf{A} \equiv \mathsf{g}_2\ \mathsf{A}$ for all $\mathsf{A} \subseteq \mathsf{X}$.*

5.1 Natural Transformation

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 $\mathsf{arr}_{\mathsf{map}}$ into a **natural transformation**: a transformation that maintains the overall structure of the computations. As a bonus, the arrow laws naturally hold.

Formally, for $\mathsf{arr}_{\mathsf{map}}$ to be a natural transformation, we require the following identities to hold:

$$\mathsf{arr}_{\mathsf{map}}\ (\mathsf{pair}_\perp\ \mathsf{f}_1\ \mathsf{f}_2) \mathrel{\tilde{\equiv}_{\mathsf{map}}} \mathsf{pair}_{\mathsf{map}}\ (\mathsf{arr}_{\mathsf{map}}\ \mathsf{f}_1)\ (\mathsf{arr}_{\mathsf{map}}\ \mathsf{f}_2) \tag{32}$$

$$\mathsf{arr}_{\mathsf{map}}\ (\ggg_\perp\ \mathsf{f}_1\ \mathsf{f}_2) \mathrel{\tilde{\equiv}_{\mathsf{map}}} \ggg_{\mathsf{map}}\ (\mathsf{arr}_{\mathsf{map}}\ \mathsf{f}_1)\ (\mathsf{arr}_{\mathsf{map}}\ \mathsf{f}_2) \tag{33}$$

$$\begin{aligned}&\mathsf{arr}_{\mathsf{map}}\ (\mathsf{if}_\perp\ \mathsf{f}_1\ \mathsf{f}_2\ \mathsf{f}_3) \mathrel{\tilde{\equiv}_{\mathsf{map}}}\\ &\quad \mathsf{if}_{\mathsf{map}}\ (\mathsf{arr}_{\mathsf{map}}\ \mathsf{f}_1)\ (\lambda 0.\, \mathsf{arr}_{\mathsf{map}}\ (\mathsf{f}_2\ 0))\ (\lambda 0.\, \mathsf{arr}_{\mathsf{map}}\ (\mathsf{f}_3\ 0))\end{aligned} \tag{34}$$

i.e. $\mathsf{arr}_{\mathsf{map}}$ must distribute over bottom arrow computations.

Fig. 2 shows the final result of deriving the mapping arrow as a natural transformation from the bottom arrow.

Theorem 1 (mapping arrow correctness)**.** $\mathsf{arr}_{\mathsf{map}}$ *is a natural transformation from the bottom arrow.*

Proof. By structural induction; cases follow. □

Each case—pairing, composition, conditional—is proved by construction, yielding an implementation.

5.2 Case: Pairing

Starting with the left-hand side of (32), we first expand definitions. For any $\mathsf{A} : \mathsf{Set}\ \mathsf{X}$,

$$\begin{aligned}&\mathsf{arr}_{\mathsf{map}}\ (\mathsf{pair}_\perp\ \mathsf{f}_1\ \mathsf{f}_2)\ \mathsf{A}\\ &\quad\equiv\ \mathsf{arr}_{\mathsf{map}}\ (\lambda \mathsf{x}.\, \mathsf{if}\ (\mathsf{f}_1\ \mathsf{x} = \perp \vee \mathsf{f}_2\ \mathsf{x} = \perp)\ \perp\ \langle \mathsf{f}_1\ \mathsf{x}, \mathsf{f}_2\ \mathsf{x}\rangle)\ \mathsf{A}\\ &\quad\equiv\ \mathsf{let}\quad \mathsf{f} := \lambda \mathsf{x}.\, \mathsf{if}\ (\mathsf{f}_1\ \mathsf{x} = \perp \vee \mathsf{f}_2\ \mathsf{x} = \perp)\ \perp\ \langle \mathsf{f}_1\ \mathsf{x}, \mathsf{f}_2\ \mathsf{x}\rangle\\ &\qquad\quad\ \ \mathsf{A}' := \mathsf{domain}_\perp\ \mathsf{f}\ \mathsf{A}\\ &\qquad\ \mathsf{in}\ \ \mathsf{f}|_{\mathsf{A}'}\end{aligned} \tag{35}$$

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

$$\begin{aligned}&\mathsf{arr}_{\mathsf{map}}\ (\mathsf{pair}_\perp\ \mathsf{f}_1\ \mathsf{f}_2)\ \mathsf{A}\\ &\quad\equiv\ \mathsf{let}\ \ \mathsf{A}_1 := (\mathsf{domain}_\perp\ \mathsf{f}_1\ \mathsf{A})\\ &\qquad\quad\ \ \mathsf{A}_2 := (\mathsf{domain}_\perp\ \mathsf{f}_2\ \mathsf{A})\\ &\qquad\quad\ \ \mathsf{A}' := \mathsf{A}_1 \cap \mathsf{A}_2\\ &\qquad\ \mathsf{in}\ \ \lambda \mathsf{x} \in \mathsf{A}'.\, \langle \mathsf{f}_1\ \mathsf{x}, \mathsf{f}_2\ \mathsf{x}\rangle\end{aligned}$$

$$\mathsf{arr}_\bot : (X \Rightarrow Y_\bot) \Rightarrow (X \Rightarrow Y_\bot)$$
$$\mathsf{arr}_\bot\ f\ :=\ f$$

$$\ggg_\bot : (X \Rightarrow Y_\bot) \Rightarrow (Y \Rightarrow Z_\bot) \Rightarrow (X \Rightarrow Z_\bot)$$
$$f_1 \ggg_\bot f_2\ x\ :=\ \mathsf{if}\ (f_1\ x = \bot)\ \bot\ (f_2\ (f_1\ x))$$

$$\mathsf{pair}_\bot : (X \Rightarrow Y_\bot) \Rightarrow (X \Rightarrow Z_\bot) \Rightarrow (X \Rightarrow \langle Y, Z\rangle_\bot)$$
$$\mathsf{pair}_\bot\ f_2\ f_2\ x\ :=\ \mathsf{if}\ (f_1\ x = \bot \vee f_2\ x = \bot)\ \bot\ \langle f_1\ x, f_2\ x\rangle$$

$$\mathsf{if}_\bot :\ (X \Rightarrow \mathsf{Bool}_\bot) \Rightarrow (1 \Rightarrow (X \Rightarrow Y_\bot)) \Rightarrow (1 \Rightarrow (X \Rightarrow Y_\bot)) \Rightarrow (X \Rightarrow Y_\bot)$$
$$\begin{aligned}\mathsf{if}_\bot\ f_1\ f_2\ f_3\ x\ :=\ \mathsf{case}\ & f_1\ x\\ & \mathsf{true}\ \Longrightarrow\ f_2\ 0\ x\\ & \mathsf{false}\ \Longrightarrow\ f_3\ 0\ x\\ & \mathsf{else}\ \Longrightarrow\ \bot\end{aligned}$$

Figure 1: Bottom arrow definitions.

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

$$\mathsf{arr}_{\mathsf{map}} : (X \Rightarrow Y_\bot) \Rightarrow (X \overset{\rightsquigarrow}{\scriptstyle\mathsf{map}} Y)$$
$$\begin{aligned}\mathsf{arr}_{\mathsf{map}}\ f\ A\ :=\ \mathsf{let}\ & A' := \mathsf{domain}_\bot\ f\ A\\ \mathsf{in}\ & f|_{A'}\end{aligned}$$

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

$$\mathsf{pair}_{\mathsf{map}} : (X \overset{\rightsquigarrow}{\scriptstyle\mathsf{map}} Y) \Rightarrow (X \overset{\rightsquigarrow}{\scriptstyle\mathsf{map}} Z) \Rightarrow (X \overset{\rightsquigarrow}{\scriptstyle\mathsf{map}} \langle Y, Z\rangle)$$
$$\mathsf{pair}_{\mathsf{map}}\ g_1\ g_2\ A\ :=\ \langle g_1\ A, g_2\ A\rangle_{\mathsf{map}}$$

$$\mathsf{if}_{\mathsf{map}} : (X \overset{\rightsquigarrow}{\scriptstyle\mathsf{map}} \mathsf{Bool}) \Rightarrow (1 \Rightarrow (X \overset{\rightsquigarrow}{\scriptstyle\mathsf{map}} Y)) \Rightarrow (1 \Rightarrow (X \overset{\rightsquigarrow}{\scriptstyle\mathsf{map}} Y)) \Rightarrow (X \overset{\rightsquigarrow}{\scriptstyle\mathsf{map}} Y)$$
$$\begin{aligned}\mathsf{if}_{\mathsf{map}}\ g_1\ g_2\ g_3\ A\ :=\ & \\ \mathsf{let}\ & g_1' := g_1\ A\\ & g_2' := \mathsf{lazy}_{\mathsf{map}}\ (g_2\ 0)\ (\mathsf{preimage}\ g_1'\ \{\mathsf{true}\})\\ & g_3' := \mathsf{lazy}_{\mathsf{map}}\ (g_3\ 0)\ (\mathsf{preimage}\ g_1'\ \{\mathsf{false}\})\\ \mathsf{in}\ & g_2' \uplus_{\mathsf{map}} g_3'\end{aligned}$$

$$\mathsf{domain}_\bot : (X \Rightarrow Y_\bot) \Rightarrow \mathsf{Set}\ X \Rightarrow \mathsf{Set}\ X$$
$$\mathsf{domain}_\bot\ f\ A\ :=\ \mathsf{preimage}\ f|_A\ ((\mathsf{image}\ f\ A)\backslash\{\bot\})$$

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

Figure 2: Mapping arrow definitions.

$$\begin{aligned}\equiv\ \mathsf{let}\ & g_1 := \mathsf{arr}_{\mathsf{map}}\ f_1\ A\\ & g_2 := \mathsf{arr}_{\mathsf{map}}\ f_2\ A\\ & A' := (\mathsf{domain}\ g_1) \cap (\mathsf{domain}\ g_2)\\ \mathsf{in}\ & \lambda x \in A'.\ \langle g_1\ x, g_2\ x\rangle\\ \equiv\ & \langle \mathsf{arr}_{\mathsf{map}}\ f_1\ A, \mathsf{arr}_{\mathsf{map}}\ f_2\ A\rangle_{\mathsf{map}}\end{aligned} \tag{36}$$

Substituting g_1 for $\mathsf{arr}_{\mathsf{map}}\ f_1$ and g_2 for $\mathsf{arr}_{\mathsf{map}}\ f_2$ in the last equality gives the definition for $\mathsf{pair}_{\mathsf{map}}$:

$$\begin{aligned}&\mathsf{pair}_{\mathsf{map}} : (X \overset{\rightsquigarrow}{\scriptstyle\mathsf{map}} Y) \Rightarrow (X \overset{\rightsquigarrow}{\scriptstyle\mathsf{map}} Z) \Rightarrow (X \overset{\rightsquigarrow}{\scriptstyle\mathsf{map}} \langle Y, Z\rangle)\\ &\mathsf{pair}_{\mathsf{map}}\ g_1\ g_2\ A\ :=\ \langle g_1\ A, g_2\ A\rangle_{\mathsf{map}}\end{aligned} \tag{37}$$

Thus, $\mathsf{arr}_{\mathsf{map}}$ distributes over $\mathsf{pair}_\bot$ by construction.

5.3 Case: Composition

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

XXX: include it?

5.4 Case: Conditional

The derivation of $\mathsf{if}_{\mathsf{map}}$ needs some care to maintain laziness of conditional branches in the presence of recursion.

We will use as an example the following bottom arrow computation, which returns true when applied to true and diverges on false:

$$\mathsf{halts\text{-}on\text{-}true}_\bot\ :=\ \mathsf{if}_\bot\ \mathsf{id}\ (\lambda 0.\,\mathsf{id})\ (\lambda 0.\,\mathsf{halts\text{-}on\text{-}true}_\bot) \tag{38}$$

Its natural transformation to the mapping arrow should diverge only if applied to a set containing false.

Starting with the left-hand-side of (34), we expand definitions, and simplify f by restricting it to a domain for which $f_1\ x$ cannot be $\bot$:

$$\begin{aligned}&\mathsf{arr}_{\mathsf{map}}\ (\mathsf{if}_\bot\ f_1\ f_2\ f_3)\ A\\ &\quad\equiv\ \mathsf{let}\quad f := \lambda x.\,\mathsf{case}\ f_1\ x\\ &\qquad\qquad\qquad\qquad \mathsf{true}\ \Longrightarrow\ f_2\ 0\ x\\ &\qquad\qquad\qquad\qquad \mathsf{false}\ \Longrightarrow\ f_3\ 0\ x\\ &\qquad\qquad\qquad\qquad \mathsf{else}\ \Longrightarrow\ \bot\\ &\qquad\quad A' := \mathsf{domain}_\bot\ f\ A\\ &\qquad \mathsf{in}\ \ f|_{A'}\\ &\quad\equiv\ \mathsf{let}\ \ A_2 := \mathsf{preimage}\ f_1|_A\ \{\mathsf{true}\}\\ &\qquad\qquad A_3 := \mathsf{preimage}\ f_1|_A\ \{\mathsf{false}\}\\ &\qquad\qquad\ f := \lambda x.\,\mathsf{if}\ (f_1\ x)\ (f_2\ 0\ x)\ (f_3\ 0\ x)\\ &\qquad\qquad A' := \mathsf{domain}_\bot\ f\ (A_2 \cup A_3)\\ &\qquad\ \mathsf{in}\ \ f|_{A'}\end{aligned} \tag{39}$$

It is tempting at this point to finish by simply converting bottom arrow computations to the mapping arrow; i.e.

$$\begin{aligned}&\mathsf{arr}_{\mathsf{map}}\ (\mathsf{if}_\bot\ f_1\ f_2\ f_3)\ A\\ &\quad\equiv\ \mathsf{let}\ \ g_1 := \mathsf{arr}_{\mathsf{map}}\ f_1\ A\\ &\qquad\qquad A_2 := \mathsf{preimage}\ g_1\ \{\mathsf{true}\}\\ &\qquad\qquad A_3 := \mathsf{preimage}\ g_1\ \{\mathsf{false}\}\\ &\qquad\qquad\ g_2 := \mathsf{arr}_{\mathsf{map}}\ (f_2\ 0)\ A_2\\ &\qquad\qquad\ g_3 := \mathsf{arr}_{\mathsf{map}}\ (f_3\ 0)\ A_3\\ &\qquad\qquad A' := (\mathsf{domain}\ g_2) \cup (\mathsf{domain}\ g_3)\\ &\qquad\ \mathsf{in}\ \ \lambda x \in A'.\,\mathsf{if}\ (g_1\ x)\ (g_2\ x)\ (g_3\ x)\end{aligned} \tag{40}$$

This is close to correct. Unfortunately, for $\mathsf{halts\text{-}on\text{-}true}_\bot$, computing $g_3 := \mathsf{arr}_{\mathsf{map}}\ (f_3\ 0)\ A_3$ always diverges. Wrapping the branch computations g_2 and g_3 in thunks will not help because A' is computed from their domains.

Note that the "true" branch needs to be taken only if A_2 is nonempty; similarly for the "false" branch and A_3. Further, applying a mapping arrow computation to $\varnothing$ should always yield the empty mapping $\varnothing$. We can therefore maintain laziness in conditional branches by applying $\mathsf{arr}_{\mathsf{map}}\ (f_2\ 0)$ and $\mathsf{arr}_{\mathsf{map}}\ (f_3\ 0)$ only to nonempty sets, using

$$\begin{aligned} &\mathsf{lazy}_{\mathsf{map}} : (X \overset{\sim}{\underset{\mathsf{map}}{\rightarrow}} Y) \Rightarrow (X \overset{\sim}{\underset{\mathsf{map}}{\rightarrow}} Y) \\ &\mathsf{lazy}_{\mathsf{map}}\ f\ A\ :=\ \mathsf{if}\ (A = \varnothing)\ \varnothing\ (f\ A) \end{aligned} \tag{41}$$

In terms of $\mathsf{lazy}_{\mathsf{map}}$, we have

$$\begin{aligned} &\mathsf{arr}_{\mathsf{map}}\ (\mathsf{if}_\perp\ f_1\ f_2\ f_3)\ A \\ &\quad\equiv\ \mathsf{let}\ \ g_1 := \mathsf{arr}_{\mathsf{map}}\ f_1\ A \\ &\qquad\quad g_2 := \mathsf{lazy}_{\mathsf{map}}\ (\mathsf{arr}_{\mathsf{map}}\ (f_2\ 0))\ (\mathsf{preimage}\ g_1\ \{\mathsf{true}\}) \\ &\qquad\quad g_3 := \mathsf{lazy}_{\mathsf{map}}\ (\mathsf{arr}_{\mathsf{map}}\ (f_3\ 0))\ (\mathsf{preimage}\ g_1\ \{\mathsf{false}\}) \\ &\qquad\quad A' := (\mathsf{domain}\ g_2) \cup (\mathsf{domain}\ g_3) \\ &\qquad\mathsf{in}\ \ \lambda x \in A'.\, \mathsf{if}\ (g_1\ x)\ (g_2\ x)\ (g_3\ x) \\ &\quad\equiv\ \mathsf{let}\ \ g_1 := \mathsf{arr}_{\mathsf{map}}\ f_1\ A \\ &\qquad\quad g_2 := \mathsf{lazy}_{\mathsf{map}}\ (\mathsf{arr}_{\mathsf{map}}\ (f_2\ 0))\ (\mathsf{preimage}\ g_1\ \{\mathsf{true}\}) \\ &\qquad\quad g_3 := \mathsf{lazy}_{\mathsf{map}}\ (\mathsf{arr}_{\mathsf{map}}\ (f_3\ 0))\ (\mathsf{preimage}\ g_1\ \{\mathsf{false}\}) \\ &\qquad\mathsf{in}\ \ g_2 \uplus_{\mathsf{map}} g_3 \end{aligned} \tag{42}$$

For $\mathsf{halts\text{-}on\text{-}true}_\perp$, $\mathsf{lazy}_{\mathsf{map}}\ (\mathsf{arr}_{\mathsf{map}}\ (f_3\ 0))\ A_3$ does not diverge when A_3 is empty.

Substituting g_1 for $\mathsf{arr}_{\mathsf{map}}\ f_1$, $g_2\ 0$ for $\mathsf{arr}_{\mathsf{map}}\ (f_2\ 0)$, and $g_3\ 0$ for $\mathsf{arr}_{\mathsf{map}}\ (f_3\ 0)$ in the last equality gives the definition for $\mathsf{if}_{\mathsf{map}}$.

5.5 Super-Saver Theorems

The following two theorems are easy consequences of the fact that $\mathsf{arr}_{\mathsf{map}}$ is a natural transformation.

Corollary 1. $\mathsf{arr}_{\mathsf{map}}$, $\mathsf{pair}_{\mathsf{map}}$ *and* $\ggg_{\mathsf{map}}$ *define an arrow.*

Corollary 2. *Let* $f : X \Rightarrow Y_\perp$ *and* $g : X \overset{\sim}{\underset{\mathsf{map}}{\rightarrow}} Y$ *such that* $g \overset{\sim}{\underset{\mathsf{map}}{=}} \mathsf{arr}_{\mathsf{map}}\ f$. *Then for all* $A \subseteq X$, $g\ A$ *diverges if and only if there exists an* $x \in A$ *for which* $f\ x$ *diverges.*

6. Preimage Mappings

On a computer, we will not often have the luxury of testing each function input to see whether it belongs in a preimage set. Even for finite domains, doing so is often intractable.

If we wish to compute with abstract, infinite sets in the language implementation, we will need its semantics defined so that removing all point computations is easy. Therefore, in the preimage arrow, we will confine computation on points to ***preimage mappings***, for which application is like applying $\mathsf{preimage}$ to a mapping. The function space is constructed by

$$X \underset{\mathsf{pre}}{\rightrightarrows} Y\ :=\ \mathcal{P}\ Y \rightharpoonup \mathcal{P}\ X \tag{43}$$

XXX: convert a mapping to a preimage mapping:

$$\begin{aligned} &\mathsf{pre} : (X \rightharpoonup Y) \Rightarrow \mathsf{Set}\ (\mathsf{Set}\ Y) \Rightarrow (X \underset{\mathsf{pre}}{\rightrightarrows} Y) \\ &\mathsf{pre}\ g\ \mathcal{B}\ :=\ \lambda B \in \mathcal{B}.\, \mathsf{preimage}\ g\ B \end{aligned} \tag{44}$$

Note that $\mathcal{B}$ is not necessarily a σ-algebra; in particular, it may be a generating family such as $\mathcal{B}_1 \boxtimes \mathcal{B}_2$.

While preimages can be computed for any subset of a mapping's *codomain*, preimage mappings are usually defined on smaller sets, such as a mapping's *range*. We need a function to compute preimages of sub-codomains, so

$$\begin{aligned} &\mathsf{pre\text{-}ap} : (X \underset{\mathsf{pre}}{\rightrightarrows} Y) \Rightarrow \mathsf{Set}\ Y \Rightarrow \mathsf{Set}\ X \\ &\mathsf{pre\text{-}ap}\ h\ B\ :=\ h\ (B \cap \bigcup(\mathsf{domain}\ h)) \end{aligned} \tag{45}$$

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

Theorem 2 ($\mathsf{pre\text{-}ap}$ computes preimages)**.** *Let*

$$\begin{aligned} &g \in X \rightharpoonup Y,\ \textit{with } \mathcal{B} \textit{ a } \sigma\textit{-algebra on } Y \\ &Y' \textit{ a set such that } (\mathsf{range}\ g) \subseteq Y' \subseteq Y \\ &\mathcal{B}' := \mathsf{trace}\ \mathcal{B}\ Y' \end{aligned} \tag{46}$$

Then for all $B \in \mathcal{B}$, $\mathsf{pre\text{-}ap}\ (\mathsf{pre}\ g\ \mathcal{B}')\ B = \mathsf{preimage}\ g\ B$.

Proof. Expand the definitions of $\mathsf{pre\text{-}ap}$ and pre, and use $\bigcup(\mathsf{domain}\ h) = Y'$ and $B \cap Y' \in \mathcal{B}'$ (by definition of trace):

$$\begin{aligned} \mathsf{pre\text{-}ap}\ (\mathsf{pre}\ g\ \mathcal{B})\ B\ &=\ \mathsf{let}\ \ h := \lambda B \in \mathcal{B}'.\, \mathsf{preimage}\ g\ B \\ &\qquad\mathsf{in}\ \ h\ (B \cap \bigcup(\mathsf{domain}\ h)) \\ &=\ \mathsf{let}\ \ h := \lambda B \in \mathcal{B}'.\, \mathsf{preimage}\ g\ B \\ &\qquad\mathsf{in}\ \ h\ (B \cap Y') \\ &=\ \mathsf{preimage}\ g\ (B \cap Y') \\ &=\ \mathsf{preimage}\ g\ B \end{aligned}$$

□

6.1 Generated Preimage Mappings

XXX: for pairing, will need to define preimage mappings for a generating family, then close the domain (and thus the range) under σ-algebra operations

XXX: proceeds just like generating σ-algebras: define a single, monotone operation:

$$\begin{aligned} &\mathsf{pre\text{-}comps} : (X \underset{\mathsf{pre}}{\rightrightarrows} Y) \Rightarrow (X \underset{\mathsf{pre}}{\rightrightarrows} Y) \\ &\mathsf{pre\text{-}comps}\ h\ := \\ &\quad\mathsf{let}\ \ \mathcal{B} := \mathsf{domain}\ h \\ &\qquad\mathsf{in}\ \ \{\langle B_1 \backslash B_2, (h\ B_1) \backslash (h\ B_2)\rangle \mid B_1, B_2 \in \mathcal{B}\} \end{aligned} \tag{47}$$

$$\begin{aligned} &\mathsf{pre\text{-}unions} : (X \underset{\mathsf{pre}}{\rightrightarrows} Y) \Rightarrow (X \underset{\mathsf{pre}}{\rightrightarrows} Y) \\ &\mathsf{pre\text{-}unions}\ h\ := \\ &\quad\mathsf{let}\ \ \mathcal{B} := \mathsf{domain}\ h \\ &\qquad\mathsf{in}\ \ \{\langle \bigcup \mathcal{B}', \bigcup(\mathsf{image}\ h\ \mathcal{B}')\rangle \mid \mathcal{B}' \subseteq \mathcal{B} \wedge |\mathcal{B}'| \leq \omega\} \end{aligned} \tag{48}$$

$$\begin{aligned} &\mathsf{pre\text{-}ops} : (X \underset{\mathsf{pre}}{\rightrightarrows} Y) \Rightarrow (X \underset{\mathsf{pre}}{\rightrightarrows} Y) \\ &\mathsf{pre\text{-}ops}\ h\ :=\ \{\langle \varnothing, \varnothing\rangle\} \cup (\mathsf{pre\text{-}comps}\ h) \cup (\mathsf{pre\text{-}unions}\ h) \end{aligned} \tag{49}$$

XXX: then apply close to it with a sufficient number of iterations:

$$\begin{aligned} &\mathsf{pre\text{-}close} : (X \underset{\mathsf{pre}}{\rightrightarrows} Y) \Rightarrow (X \underset{\mathsf{pre}}{\rightrightarrows} Y) \\ &\mathsf{pre\text{-}close}\ :=\ \mathsf{close}\ \mathsf{pre\text{-}ops}\ \omega_1 \end{aligned} \tag{50}$$

XXX: central fact: closing a preimage mapping defined for a generating family is equivalent to defining that preimage mapping for the generated σ-algebra

Theorem 3 ($\mathsf{pre\text{-}close}$ distributes over pre)**.** *Let* $f \in X \rightharpoonup Y$ *and* $\mathcal{B}$ *a generating family for* Y. *Then* $\mathsf{pre\text{-}close}\ (\mathsf{pre}\ f\ \mathcal{B}) = \mathsf{pre}\ f\ (\sigma\text{-}\mathsf{close}\ \mathcal{B})$.

Proof. By transfinite induction, using: the preimage of $\varnothing$ under any function is $\varnothing$, and preimages distribute over complements and unions. □

6.2 Preimage Mapping Pairing

XXX: moar wurds in this section

$$\langle \cdot, \cdot \rangle_{\mathsf{pre}} : (\mathsf{X} \rightrightarrows_{\mathsf{pre}} \mathsf{Y}) \Rightarrow (\mathsf{X} \rightrightarrows_{\mathsf{pre}} \mathsf{Z}) \Rightarrow (\mathsf{X} \rightrightarrows_{\mathsf{pre}} \mathsf{Y} \times \mathsf{Z})$$
$$\langle \mathsf{h}_1, \mathsf{h}_2 \rangle_{\mathsf{pre}} := \mathsf{let}\ \mathcal{B} := (\mathsf{domain}\ \mathsf{h}_1) \boxtimes (\mathsf{domain}\ \mathsf{h}_2)\ \mathsf{h} := \lambda \mathsf{B}_1 \times \mathsf{B}_2 \in \mathcal{B}.\, (\mathsf{h}_1\ \mathsf{B}_1) \cap (\mathsf{h}_2\ \mathsf{B}_2)\ \mathsf{in}\ \mathsf{pre\text{-}close}\ \mathsf{h} \tag{51}$$

Lemma 5 (preimage distributes over pairing)**.** *Let* $\mathsf{g}_1 \in \mathsf{X} \rightharpoonup \mathsf{Y}$ *and* $\mathsf{g}_2 \in \mathsf{X} \rightharpoonup \mathsf{Z}$*. Then for all* $\mathsf{B}_1 \subseteq \mathsf{Y}$ *and* $\mathsf{B}_2 \subseteq \mathsf{Z}$*,* $\mathsf{preimage}\ \langle \mathsf{g}_1, \mathsf{g}_2 \rangle_{\mathsf{map}}\ (\mathsf{B}_1 \times \mathsf{B}_2) = (\mathsf{preimage}\ \mathsf{g}_1\ \mathsf{B}_1) \cap (\mathsf{preimage}\ \mathsf{g}_2\ \mathsf{B}_2)$.

Theorem 4 (trace distributes over $(\boxtimes)$ and $(\times)$)**.** *For any families* $\mathcal{A}_1$ *and* $\mathcal{A}_2$*,* $\mathsf{trace}\ (\mathcal{A}_1 \boxtimes \mathcal{A}_2)\ (\mathsf{A}_1 \times \mathsf{A}_2) = (\mathsf{trace}\ \mathcal{A}_1\ \mathsf{A}_1) \boxtimes (\mathsf{trace}\ \mathcal{A}_2\ \mathsf{A}_2)$.

Proof. Expand $(\boxtimes)$ and trace, $(\cap)$ distributes over $(\times)$, rewrite in terms of $(\boxtimes)$, and rewrite in terms of trace.

$$\begin{aligned}
&\mathsf{trace}\ (\mathcal{A}_1 \boxtimes \mathcal{A}_2)\ (\mathsf{A}_1 \times \mathsf{A}_2) \\
&\quad = \{(\mathsf{A}_1 \times \mathsf{A}_2) \cap (\mathsf{A}'_1 \times \mathsf{A}'_2) \mid \mathsf{A}'_1 \in \mathcal{A}_1, \mathsf{A}'_2 \in \mathcal{A}_2\} \\
&\quad = \{(\mathsf{A}_1 \cap \mathsf{A}'_1) \times (\mathsf{A}_2 \cap \mathsf{A}'_2) \mid \mathsf{A}'_1 \in \mathcal{A}_1, \mathsf{A}'_2 \in \mathcal{A}_2\} \\
&\quad = \{\mathsf{A}_1 \cap \mathsf{A}'_1 \mid \mathsf{A}'_1 \in \mathcal{A}_1\} \boxtimes \{\mathsf{A}_2 \cap \mathsf{A}'_2 \mid \mathsf{A}'_2 \in \mathcal{A}_2\} \\
&\quad = (\mathsf{trace}\ \mathcal{A}_1\ \mathsf{A}_1) \boxtimes (\mathsf{trace}\ \mathcal{A}_2\ \mathsf{A}_2)
\end{aligned}$$

□

Theorem 5 (trace distributes over $(\otimes)$ and $(\times)$)**.** *For any* σ*-algebras* $\mathcal{A}_1$ *and* $\mathcal{A}_2$*,* $\mathsf{trace}\ (\mathcal{A}_1 \otimes \mathcal{A}_2)\ (\mathsf{A}_1 \times \mathsf{A}_2) = (\mathsf{trace}\ \mathcal{A}_1\ \mathsf{A}_1) \otimes (\mathsf{trace}\ \mathcal{A}_2\ \mathsf{A}_2)$.

Proof. Expand $(\otimes)$, apply Lemma 2, apply Theorem 4, and rewrite in terms of $(\otimes)$.

$$\begin{aligned}
&\mathsf{trace}\ (\mathcal{A}_1 \otimes \mathcal{A}_2)\ (\mathsf{A}_1 \times \mathsf{A}_2) \\
&\quad = \mathsf{trace}\ (\sigma\text{-}\mathsf{close}\ (\mathcal{A}_1 \boxtimes \mathcal{A}_2))\ (\mathsf{A}_1 \times \mathsf{A}_2) \\
&\quad = \sigma\text{-}\mathsf{close}\ (\mathsf{trace}\ (\mathcal{A}_1 \boxtimes \mathcal{A}_2)\ (\mathsf{A}_1 \times \mathsf{A}_2)) \\
&\quad = \sigma\text{-}\mathsf{close}\ ((\mathsf{trace}\ \mathcal{A}_1\ \mathsf{A}_1) \boxtimes (\mathsf{trace}\ \mathcal{A}_2\ \mathsf{A}_2)) \\
&\quad = (\mathsf{trace}\ \mathcal{A}_1\ \mathsf{A}_1) \otimes (\mathsf{trace}\ \mathcal{A}_2\ \mathsf{A}_2)
\end{aligned}$$

□

Theorem 6 ($\mathsf{pre\text{-}ap}$ of $\langle \cdot, \cdot \rangle_{\mathsf{pre}}$ computes preimages)**.** *Let*

$$\begin{array}{l}
\mathsf{g}_1 \in \mathsf{X} \rightharpoonup \mathsf{Y},\ \textit{with}\ \mathcal{B}_1\ \textit{a}\ \sigma\textit{-algebra on}\ \mathsf{Y} \\
\mathcal{B}'_1 := \mathsf{trace}\ \mathcal{B}_1\ (\mathsf{range}\ \mathsf{g}_1) \\
\mathsf{h}_1 := \mathsf{pre}\ \mathsf{g}_1\ \mathcal{B}'_1 \\
\mathsf{g}_2 \in \mathsf{X} \rightharpoonup \mathsf{Z},\ \textit{with}\ \mathcal{B}_2\ \textit{a}\ \sigma\textit{-algebra on}\ \mathsf{Z} \\
\mathcal{B}'_2 := \mathsf{trace}\ \mathcal{B}_2\ (\mathsf{range}\ \mathsf{g}_2) \\
\mathsf{h}_2 := \mathsf{pre}\ \mathsf{g}_2\ \mathcal{B}_2
\end{array} \tag{52}$$

Then for all pair sets $\mathsf{B} \in \mathcal{B}_1 \otimes \mathcal{B}_2$*,* $\mathsf{pre\text{-}ap}\ \langle \mathsf{h}_1, \mathsf{h}_2 \rangle_{\mathsf{pre}}\ \mathsf{B} = \mathsf{preimage}\ \langle \mathsf{g}_1, \mathsf{g}_2 \rangle_{\mathsf{map}}\ \mathsf{B}$.

Proof. Expand definitions, substitute h_1 and h_2, expand definition of pre, apply Lemma 5, rewrite in terms of pre, apply Theorem 3, apply Theorem 5, and apply Theorem 2.

$$\begin{aligned}
&\mathsf{pre\text{-}ap}\ \langle \mathsf{h}_1, \mathsf{h}_2 \rangle_{\mathsf{pre}}\ \mathsf{B} \\
&\quad = \mathsf{let}\ \mathcal{B}' := (\mathsf{domain}\ \mathsf{h}_1) \boxtimes (\mathsf{domain}\ \mathsf{h}_2)\ \mathsf{h} := \lambda \mathsf{B}_1 \times \mathsf{B}_2 \in \mathcal{B}'.\, (\mathsf{h}_1\ \mathsf{B}_1) \cap (\mathsf{h}_2\ \mathsf{B}_2)\ \mathsf{in}\ \mathsf{pre\text{-}ap}\ (\mathsf{pre\text{-}close}\ \mathsf{h})\ \mathsf{B} \\
&\quad = \mathsf{let}\ \mathcal{B}' := \mathcal{B}'_1 \boxtimes \mathcal{B}'_2\ \mathsf{h} := \lambda \mathsf{B}_1 \times \mathsf{B}_2 \in \mathcal{B}'.\, (\mathsf{pre}\ \mathsf{g}_1\ \mathcal{B}'_1\ \mathsf{B}_1) \cap (\mathsf{pre}\ \mathsf{g}_2\ \mathcal{B}'_2\ \mathsf{B}_2)\ \mathsf{in}\ \mathsf{pre\text{-}ap}\ (\mathsf{pre\text{-}close}\ \mathsf{h})\ \mathsf{B} \\
&\quad = \mathsf{let}\ \mathcal{B}' := \mathcal{B}'_1 \boxtimes \mathcal{B}'_2\ \mathsf{h} := \lambda \mathsf{B}_1 \times \mathsf{B}_2 \in \mathcal{B}'.\, (\mathsf{preimage}\ \mathsf{g}_1\ \mathsf{B}_1) \cap (\mathsf{preimage}\ \mathsf{g}_2\ \mathsf{B}_2)\ \mathsf{in}\ \mathsf{pre\text{-}ap}\ (\mathsf{pre\text{-}close}\ \mathsf{h})\ \mathsf{B} \\
&\quad = \mathsf{let}\ \mathcal{B}' := \mathcal{B}'_1 \boxtimes \mathcal{B}'_2\ \mathsf{h} := \lambda \mathsf{B} \in \mathcal{B}'.\, \mathsf{preimage}\ \langle \mathsf{g}_1, \mathsf{g}_2 \rangle_{\mathsf{map}}\ \mathsf{B}\ \mathsf{in}\ \mathsf{pre\text{-}ap}\ (\mathsf{pre\text{-}close}\ \mathsf{h})\ \mathsf{B} \\
&\quad = \mathsf{let}\ \mathcal{B}' := \mathcal{B}'_1 \boxtimes \mathcal{B}'_2\ \mathsf{in}\ \mathsf{pre\text{-}ap}\ (\mathsf{pre\text{-}close}\ (\mathsf{pre}\ \langle \mathsf{g}_1, \mathsf{g}_2 \rangle_{\mathsf{map}}\ \mathcal{B}'))\ \mathsf{B} \\
&\quad = \mathsf{let}\ \mathcal{B}' := \mathcal{B}'_1 \otimes \mathcal{B}'_2\ \mathsf{in}\ \mathsf{pre\text{-}ap}\ (\mathsf{pre}\ \langle \mathsf{g}_1, \mathsf{g}_2 \rangle_{\mathsf{map}}\ \mathcal{B}')\ \mathsf{B} \\
&\quad = \mathsf{let}\ \mathcal{B}' := \mathsf{trace}\ \mathcal{B}\ ((\mathsf{range}\ \mathsf{g}_1) \times (\mathsf{range}\ \mathsf{g}_2))\ \mathsf{in}\ \mathsf{pre\text{-}ap}\ (\mathsf{pre}\ \langle \mathsf{g}_1, \mathsf{g}_2 \rangle_{\mathsf{map}}\ \mathcal{B}')\ \mathsf{B} \\
&\quad = \mathsf{preimage}\ \langle \mathsf{g}_1, \mathsf{g}_2 \rangle_{\mathsf{map}}\ \mathsf{B}
\end{aligned}$$

□

6.3 Preimage Mapping Composition

XXX: moar wurds in this section

$$(\circ_{\mathsf{pre}}) : (\mathsf{Y} \rightrightarrows_{\mathsf{pre}} \mathsf{Z}) \Rightarrow (\mathsf{X} \rightrightarrows_{\mathsf{pre}} \mathsf{Y}) \Rightarrow (\mathsf{X} \rightrightarrows_{\mathsf{pre}} \mathsf{Z})$$
$$\mathsf{h}_2 \circ_{\mathsf{pre}} \mathsf{h}_1 := \mathsf{let}\ \mathcal{C} := \mathsf{domain}\ \mathsf{h}_2\ \mathsf{in}\ \lambda \mathsf{C} \in \mathcal{C}.\, \mathsf{pre\text{-}ap}\ \mathsf{h}_1\ (\mathsf{h}_2\ \mathsf{C}) \tag{53}$$

Lemma 6 (preimages under composition)**.** *Let* $\mathsf{g}_1 \in \mathsf{X} \rightharpoonup \mathsf{Y}$ *and* $\mathsf{g}_2 \in \mathsf{Y} \rightharpoonup \mathsf{Z}$*. Then for all* $\mathsf{C} \subseteq \mathsf{Z}$*,* $\mathsf{preimage}\ (\mathsf{g}_2 \circ_{\mathsf{map}} \mathsf{g}_1)\ \mathsf{C} = \mathsf{preimage}\ \mathsf{g}_1\ (\mathsf{preimage}\ \mathsf{g}_2\ \mathsf{C})$.

Theorem 7 (preimage mapping composition)**.** *Let*

$$\begin{array}{l}
\mathsf{g}_2 \in \mathsf{Y} \rightharpoonup \mathsf{Z},\ \textit{with}\ \mathcal{C}\ \textit{a}\ \sigma\textit{-algebra on}\ \mathsf{Z} \\
\mathcal{C}' := \sigma\text{-}\mathsf{trace}\ \mathcal{C}\ (\mathsf{range}\ \mathsf{g}_2) \\
\mathsf{h}_2 := \mathsf{pre}\ \mathsf{g}_2\ \mathcal{C}' \\
\mathsf{g}_1 \in \mathsf{X} \rightharpoonup \mathsf{Y} \\
\mathcal{B}' := \sigma\text{-}\mathsf{trace}\ (\mathsf{range}\ \mathsf{h}_2)\ (\mathsf{range}\ \mathsf{g}_1) \\
\mathsf{h}_1 := \mathsf{pre}\ \mathsf{g}_1\ \mathcal{B}'
\end{array} \tag{54}$$

Then for all $\mathsf{C} \in \mathcal{C}$*,* $\mathsf{pre\text{-}ap}\ (\mathsf{h}_2 \circ_{\mathsf{pre}} \mathsf{h}_1)\ \mathsf{C} = \mathsf{preimage}\ (\mathsf{g}_2 \circ_{\mathsf{map}} \mathsf{g}_1)\ \mathsf{C}$.

Proof. Expand definitions, apply Theorem 2, expand h, rewrite in terms of $\mathsf{pre\text{-}ap}$, apply Theorem 2, and apply Lemma 6.

$$\begin{aligned}
&\mathsf{pre\text{-}ap}\ (\mathsf{h}_2 \circ_{\mathsf{pre}} \mathsf{h}_1)\ \mathsf{C} \\
&\quad = \mathsf{let}\ \mathsf{h} := \lambda \mathsf{C} \in \mathcal{C}'.\, \mathsf{pre\text{-}ap}\ (\mathsf{pre}\ \mathsf{g}_1\ \mathcal{B}')\ ((\mathsf{pre}\ \mathsf{g}_2\ \mathcal{C}')\ \mathsf{C})\ \mathsf{in}\ \mathsf{h}\ (\mathsf{C} \cap \bigcup(\mathsf{domain}\ \mathsf{h})) \\
&\quad = \mathsf{let}\ \mathsf{h} := \lambda \mathsf{C} \in \mathcal{C}'.\, \mathsf{preimage}\ \mathsf{g}_1\ ((\mathsf{pre}\ \mathsf{g}_2\ \mathcal{C}')\ \mathsf{C})\ \mathsf{in}\ \mathsf{h}\ (\mathsf{C} \cap \bigcup \mathcal{C}') \\
&\quad = \mathsf{preimage}\ \mathsf{g}_1\ ((\mathsf{pre}\ \mathsf{g}_2\ \mathcal{C}')\ (\mathsf{C} \cap \bigcup \mathcal{C}')) \\
&\quad = \mathsf{preimage}\ \mathsf{g}_1\ (\mathsf{pre\text{-}ap}\ (\mathsf{pre}\ \mathsf{g}_2\ \mathcal{C}')\ \mathsf{C}) \\
&\quad = \mathsf{preimage}\ \mathsf{g}_1\ (\mathsf{preimage}\ \mathsf{g}_2\ \mathsf{C}) \\
&\quad = \mathsf{preimage}\ (\mathsf{g}_2 \circ_{\mathsf{map}} \mathsf{g}_1)\ \mathsf{C}
\end{aligned} \tag{55}$$

□

6.4 Piecewise Preimage Mappings

XXX: moar wurds in this section

$$
\begin{array}{l}
(\uplus_{\mathsf{pre}}) : (\mathsf{X} \rightrightarrows_{\mathsf{pre}} \mathsf{Y}) \Rightarrow (\mathsf{X} \rightrightarrows_{\mathsf{pre}} \mathsf{Y}) \Rightarrow (\mathsf{X} \rightrightarrows_{\mathsf{pre}} \mathsf{Y}) \\
\mathsf{h}_1 \uplus_{\mathsf{pre}} \mathsf{h}_2 \ := \\
\quad \mathsf{let} \ \ \mathcal{B} := (\mathsf{domain}\ \mathsf{h}_1) \cup (\mathsf{domain}\ \mathsf{h}_2) \\
\qquad\quad \mathsf{h} := \lambda \mathsf{B} \in \mathcal{B}.\, (\mathsf{pre\text{-}ap}\ \mathsf{h}_1\ \mathsf{B}) \uplus (\mathsf{pre\text{-}ap}\ \mathsf{h}_2\ \mathsf{B}) \\
\quad\ \mathsf{in} \ \ \mathsf{pre\text{-}close}\ \mathsf{h}
\end{array}
\tag{56}
$$

Lemma 7 (preimages under piecewise mappings)**.** *Let* $\mathsf{g}_1 \in \mathsf{X} \rightharpoonup \mathsf{Y}$ *and* $\mathsf{g}_2 \in \mathsf{X} \rightharpoonup \mathsf{Y}$ *have disjoint domains. Then for all* $\mathsf{B} \subseteq \mathsf{Y}$, $\mathsf{preimage}\ (\mathsf{g}_1 \uplus_{\mathsf{map}} \mathsf{g}_2)\ \mathsf{B} = (\mathsf{preimage}\ \mathsf{g}_1\ \mathsf{B}) \uplus (\mathsf{preimage}\ \mathsf{g}_2\ \mathsf{B})$.

Theorem 8 (piecewise preimage mappings)**.** *Let* $\mathsf{g}_1 \in \mathsf{X} \rightharpoonup \mathsf{Y}$ *and* $\mathsf{g}_2 \in \mathsf{X} \rightharpoonup \mathsf{Y}$ *have disjoint domains. Let* $\mathcal{B}'$ *be a* σ*-algebra on* Y, $\mathcal{B}_1 := \sigma\text{-}\mathsf{trace}\ \mathcal{B}'\ (\mathsf{range}\ \mathsf{g}_1)$ *and* $\mathcal{B}_2 := \sigma\text{-}\mathsf{trace}\ \mathcal{B}'\ (\mathsf{range}\ \mathsf{g}_2)$. *Let* $\mathsf{h}_1 := \mathsf{pre}\ \mathsf{g}_1\ \mathcal{B}_1$ *and* $\mathsf{h}_2 := \mathsf{pre}\ \mathsf{g}_2\ \mathcal{B}_2$. *Then for all* $\mathsf{B} \in \mathcal{B}'$, $\mathsf{pre\text{-}ap}\ (\mathsf{h}_1 \uplus_{\mathsf{pre}} \mathsf{h}_2)\ \mathsf{B} = \mathsf{preimage}\ (\mathsf{g}_1 \uplus_{\mathsf{map}} \mathsf{g}_2)\ \mathsf{B}$.

Proof. Expand definitions, substitute h_1 and h_2, apply Theorem 2, apply Lemma 7, rewrite in terms of pre, apply Theorem 3, and rewrite in terms of $(\oplus)$:

$$
\begin{array}{l}
\mathsf{pre\text{-}ap}\ (\mathsf{h}_1 \uplus_{\mathsf{pre}} \mathsf{h}_2)\ \mathsf{B} \\
\quad = \ \mathsf{let} \ \ \mathcal{B} := (\mathsf{domain}\ \mathsf{h}_1) \cup (\mathsf{domain}\ \mathsf{h}_2) \\
\qquad\quad \mathsf{h} := \lambda \mathsf{B} \in \mathcal{B}.\, (\mathsf{pre\text{-}ap}\ \mathsf{h}_1\ \mathsf{B}) \uplus (\mathsf{pre\text{-}ap}\ \mathsf{h}_2\ \mathsf{B}) \\
\qquad\ \mathsf{in} \ \ \mathsf{pre\text{-}ap}\ (\mathsf{pre\text{-}close}\ \mathsf{h})\ \mathsf{B} \\
\quad = \ \mathsf{let} \ \ \mathcal{B} := \mathcal{B}_1 \cup \mathcal{B}_2 \\
\qquad\quad \mathsf{h} := \lambda \mathsf{B} \in \mathcal{B}.\, (\mathsf{pre\text{-}ap}\ (\mathsf{pre}\ \mathsf{g}_1\ \mathcal{B}_1)\ \mathsf{B}) \uplus \\
\qquad\qquad\qquad\qquad (\mathsf{pre\text{-}ap}\ (\mathsf{pre}\ \mathsf{g}_2\ \mathcal{B}_2)\ \mathsf{B}) \\
\qquad\ \mathsf{in} \ \ \mathsf{pre\text{-}ap}\ (\mathsf{pre\text{-}close}\ \mathsf{h})\ \mathsf{B} \\
\quad = \ \mathsf{let} \ \ \mathcal{B} := \mathcal{B}_1 \cup \mathcal{B}_2 \\
\qquad\quad \mathsf{h} := \lambda \mathsf{B} \in \mathcal{B}.\, (\mathsf{preimage}\ \mathsf{g}_1\ \mathsf{B}) \uplus \\
\qquad\qquad\qquad\qquad (\mathsf{preimage}\ \mathsf{g}_2\ \mathsf{B}) \\
\qquad\ \mathsf{in} \ \ \mathsf{pre\text{-}ap}\ (\mathsf{pre\text{-}close}\ \mathsf{h})\ \mathsf{B} \\
\quad = \ \mathsf{let} \ \ \mathcal{B} := \mathcal{B}_1 \cup \mathcal{B}_2 \\
\qquad\quad \mathsf{h} := \lambda \mathsf{B} \in \mathcal{B}.\, \mathsf{preimage}\ (\mathsf{g}_1 \uplus_{\mathsf{map}} \mathsf{g}_2)\ \mathsf{B} \\
\qquad\ \mathsf{in} \ \ \mathsf{pre\text{-}ap}\ (\mathsf{pre\text{-}close}\ \mathsf{h})\ \mathsf{B} \\
\quad = \ \mathsf{let} \ \ \mathcal{B} := \mathcal{B}_1 \cup \mathcal{B}_2 \\
\qquad\ \mathsf{in} \ \ \mathsf{pre\text{-}ap}\ (\mathsf{pre\text{-}close}\ (\mathsf{pre}\ (\mathsf{g}_1 \uplus_{\mathsf{map}} \mathsf{g}_2)\ \mathcal{B}))\ \mathsf{B} \\
\quad = \ \mathsf{let} \ \ \mathcal{B} := \mathcal{B}_1 \cup \mathcal{B}_2 \\
\qquad\ \mathsf{in} \ \ \mathsf{pre\text{-}ap}\ (\mathsf{pre}\ (\mathsf{g}_1 \uplus_{\mathsf{map}} \mathsf{g}_2)\ (\sigma\text{-}\mathsf{close}\ \mathcal{B}))\ \mathsf{B}
\end{array}
$$

Finally, noting that $\sigma\text{-}\mathsf{trace}\ \mathcal{B}'\ ((\mathsf{range}\ \mathsf{g}_1) \cup (\mathsf{range}\ \mathsf{g}_2)) \subseteq \sigma\text{-}\mathsf{close}\ (\mathcal{B}_1 \cup \mathcal{B}_2)$, apply Theorem 2. □

7. The Universe

XXX: to define standard σ-algebras, we need a universal set $\mathbb{U}$ and a universal σ-algebra $\mathcal{U}$

XXX: for any subset $\mathsf{A} \subseteq \mathbb{U}$, σ returns the ***standard σ-algebra*** for A

$$
\begin{array}{l}
\sigma : \mathsf{Set}\ \mathsf{X} \Rightarrow \mathsf{Set}\ (\mathsf{Set}\ \mathsf{X}) \\
\sigma\ \mathsf{A} \ := \ \mathsf{trace}\ \mathcal{U}\ \mathsf{A}
\end{array}
\tag{57}
$$

XXX: requirements: $\{\mathsf{true}\}$ and $\{\mathsf{false}\}$ must be measurable, must be closed under $(\otimes)$

8. Deriving the Preimage Arrow

XXX: intro

$$
\mathsf{X} \rightsquigarrow_{\mathsf{pre}} \mathsf{Y} \ ::= \ \mathsf{Set}\ \mathsf{X} \Rightarrow (\mathsf{X} \rightrightarrows_{\mathsf{pre}} \mathsf{Y}) \tag{58}
$$

$$
\begin{array}{l}
\mathsf{arr}_{\mathsf{pre}} : (\mathsf{X} \rightsquigarrow_{\mathsf{map}} \mathsf{Y}) \Rightarrow (\mathsf{X} \rightsquigarrow_{\mathsf{pre}} \mathsf{Y}) \\
\mathsf{arr}_{\mathsf{pre}}\ \mathsf{g}\ \mathsf{A} \ := \ \mathsf{let} \ \ \mathsf{g}' := \mathsf{g}\ \mathsf{A} \\
\qquad\qquad\qquad\quad \mathcal{B}' := \sigma\ (\mathsf{range}\ \mathsf{g}') \\
\qquad\qquad\quad \mathsf{in} \ \ \mathsf{pre}\ \mathsf{g}'\ \mathcal{B}'
\end{array}
\tag{59}
$$

Definition 4 (Preimage arrow equivalence)**.** *Two preimage arrow computations* $\mathsf{h}_1 : \mathsf{X} \rightsquigarrow_{\mathsf{pre}} \mathsf{Y}$ *and* $\mathsf{h}_2 : \mathsf{X} \rightsquigarrow_{\mathsf{pre}} \mathsf{Y}$ *are equivalent, written* $\mathsf{h}_1 \equiv_{\mathsf{pre}} \mathsf{h}_2$, *when* $\mathsf{pre\text{-}ap}\ (\mathsf{h}_1\ \mathsf{A})\ \mathsf{B} \equiv \mathsf{pre\text{-}ap}\ (\mathsf{h}_2\ \mathsf{A})\ \mathsf{B}$ *for any* $\mathsf{A} \subseteq \mathsf{X}$ *and* $\mathsf{B} \in \sigma\ \mathsf{Y}$.

8.1 Natural Transformation

XXX: ensuring $\mathsf{arr}_{\mathsf{pre}}$ is a natural transformation from the mapping arrow...

Formally, for $\mathsf{arr}_{\mathsf{pre}}$ to be a natural transformation,

$$
\mathsf{arr}_{\mathsf{pre}}\ (\mathsf{pair}_{\mathsf{map}}\ \mathsf{g}_1\ \mathsf{g}_2) \ \equiv_{\mathsf{pre}} \ \mathsf{pair}_{\mathsf{pre}}\ (\mathsf{arr}_{\mathsf{pre}}\ \mathsf{g}_1)\ (\mathsf{arr}_{\mathsf{pre}}\ \mathsf{g}_2) \tag{60}
$$

$$
\mathsf{arr}_{\mathsf{pre}}\ (\ggg_{\mathsf{map}}\ \mathsf{g}_1\ \mathsf{g}_2) \ \equiv_{\mathsf{pre}} \ \ggg_{\mathsf{pre}}\ (\mathsf{arr}_{\mathsf{pre}}\ \mathsf{g}_1)\ (\mathsf{arr}_{\mathsf{pre}}\ \mathsf{g}_2) \tag{61}
$$

$$
\begin{array}{l}
\mathsf{arr}_{\mathsf{pre}}\ (\mathsf{if}_{\mathsf{map}}\ \mathsf{g}_1\ \mathsf{g}_2\ \mathsf{g}_3) \ \equiv_{\mathsf{pre}} \\
\quad \mathsf{if}_{\mathsf{pre}}\ (\mathsf{arr}_{\mathsf{pre}}\ \mathsf{g}_1)\ (\lambda 0.\, \mathsf{arr}_{\mathsf{pre}}\ (\mathsf{g}_2\ 0))\ (\lambda 0.\, \mathsf{arr}_{\mathsf{pre}}\ (\mathsf{g}_3\ 0))
\end{array}
\tag{62}
$$

i.e. $\mathsf{arr}_{\mathsf{pre}}$ must distribute over mapping arrow computations.

Fig. 3 shows the final result of deriving the preimage arrow as a natural transformation from the mapping arrow.

Theorem 9 (preimage arrow correctness)**.** $\mathsf{arr}_{\mathsf{pre}}$ *is a natural transformation from the mapping arrow.*

Proof. By structural induction; cases follow. □

8.2 Case: Pairing

Starting with the left-hand side of (60), we expand definitions and apply Theorem 2 with $\mathcal{B}' = \mathsf{trace}\ \mathcal{B}\ (\mathsf{range}\ \mathsf{g}')$ and $\mathcal{B} = \mathsf{trace}\ \mathcal{U}\ (\mathsf{Y} \times \mathsf{Z})$:

$$
\begin{array}{l}
\mathsf{pre\text{-}ap}\ (\mathsf{arr}_{\mathsf{pre}}\ (\mathsf{pair}_{\mathsf{map}}\ \mathsf{g}_1\ \mathsf{g}_2)\ \mathsf{A})\ \mathsf{B} \\
\quad \equiv \ \mathsf{let} \ \ \mathsf{g}' := \langle \mathsf{g}_1\ \mathsf{A}, \mathsf{g}_2\ \mathsf{A} \rangle_{\mathsf{map}} \\
\qquad\quad \mathcal{B}' := \sigma\ (\mathsf{range}\ \mathsf{g}') \\
\qquad\ \mathsf{in} \ \ \mathsf{pre\text{-}ap}\ (\mathsf{pre}\ \mathsf{g}'\ \mathcal{B}')\ \mathsf{B} \\
\quad \equiv \ \mathsf{preimage}\ \langle \mathsf{g}_1\ \mathsf{A}, \mathsf{g}_2\ \mathsf{A} \rangle_{\mathsf{map}}\ \mathsf{B}
\end{array}
$$

Next, we apply Theorem 6 with $\mathcal{B}_1' = \mathsf{trace}\ \mathcal{B}_1\ (\mathsf{range}\ \mathsf{g}_1')$ and $\mathcal{B}_1 = \mathsf{trace}\ \mathcal{U}\ \mathsf{Y}$ (similarly for $\mathcal{B}_2'$ and $\mathcal{B}_2$), and rewrite in terms of $\mathsf{arr}_{\mathsf{pre}}$:

$$
\begin{array}{l}
\mathsf{pre\text{-}ap}\ (\mathsf{arr}_{\mathsf{pre}}\ (\mathsf{pair}_{\mathsf{map}}\ \mathsf{g}_1\ \mathsf{g}_2)\ \mathsf{A})\ \mathsf{B} \\
\quad \equiv \ \mathsf{let} \ \ \mathsf{g}_1' := \mathsf{g}_1\ \mathsf{A} \\
\qquad\quad \mathcal{B}_1' := \sigma\ (\mathsf{range}\ \mathsf{g}_1') \\
\qquad\quad \mathsf{h}_1 := \mathsf{pre}\ \mathsf{g}_1'\ \mathcal{B}_1' \\
\qquad\quad \mathsf{g}_2' := \mathsf{g}_2\ \mathsf{A} \\
\qquad\quad \mathcal{B}_2' := \sigma\ (\mathsf{range}\ \mathsf{g}_2') \\
\qquad\quad \mathsf{h}_2 := \mathsf{pre}\ \mathsf{g}_2'\ \mathcal{B}_2' \\
\qquad\ \mathsf{in} \ \ \mathsf{pre\text{-}ap}\ \langle \mathsf{h}_1, \mathsf{h}_2 \rangle_{\mathsf{pre}}\ \mathsf{B} \\
\quad \equiv \ \mathsf{let} \ \ \mathsf{h}_1 := \mathsf{arr}_{\mathsf{pre}}\ \mathsf{g}_1\ \mathsf{A} \\
\qquad\quad \mathsf{h}_2 := \mathsf{arr}_{\mathsf{pre}}\ \mathsf{g}_2\ \mathsf{A} \\
\qquad\ \mathsf{in} \ \ \mathsf{pre\text{-}ap}\ \langle \mathsf{h}_1, \mathsf{h}_2 \rangle_{\mathsf{pre}}\ \mathsf{B}
\end{array}
\tag{63}
$$

Substituting h_1 for $\mathsf{arr}_{\mathsf{pre}}\ \mathsf{g}_1$ and h_2 for $\mathsf{arr}_{\mathsf{pre}}\ \mathsf{g}_2$ in the last equality gives a definition for $\mathsf{pair}_{\mathsf{pre}}$ for which (60) holds:

$$
\begin{array}{l}
\mathsf{pair}_{\mathsf{pre}} : (\mathsf{X} \rightsquigarrow_{\mathsf{pre}} \mathsf{Y}) \Rightarrow (\mathsf{X} \rightsquigarrow_{\mathsf{pre}} \mathsf{Z}) \Rightarrow (\mathsf{X} \rightsquigarrow_{\mathsf{pre}} \mathsf{Y} \times \mathsf{Z}) \\
\mathsf{pair}_{\mathsf{pre}}\ \mathsf{h}_1\ \mathsf{h}_2\ \mathsf{A} \ := \ \langle \mathsf{h}_1\ \mathsf{A}, \mathsf{h}_2\ \mathsf{A} \rangle_{\mathsf{pre}}
\end{array}
\tag{64}
$$

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

$$\mathsf{arr}_{\mathsf{pre}} : (\mathsf{X} \underset{\mathsf{map}}{\rightarrow} \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y})$$
$$\begin{array}{l}\mathsf{arr}_{\mathsf{pre}}\ \mathsf{g}\ \mathsf{A} := \mathsf{let}\ \ \mathsf{g}' := \mathsf{g}\ \mathsf{A} \\ \qquad\qquad\quad \mathcal{B} := \sigma\ (\mathsf{range}\ \mathsf{g}') \\ \qquad\qquad \mathsf{in}\ \ \mathsf{pre}\ \mathsf{g}'\ \mathcal{B}\end{array}$$

$$\ggg_{\mathsf{pre}} : (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}) \Rightarrow (\mathsf{Y} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Z}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Z})$$
$$\begin{array}{l}\ggg_{\mathsf{pre}}\ \mathsf{h}_1\ \mathsf{h}_2\ \mathsf{A} := \mathsf{let}\ \ \mathsf{h}_1' := \mathsf{h}_1\ \mathsf{A} \\ \qquad\qquad\qquad\quad \mathsf{h}_2' := \mathsf{h}_2\ (\bigcup(\mathsf{domain}\ \mathsf{h}_1')) \\ \qquad\qquad\quad \mathsf{in}\ \ \mathsf{h}_2 \circ_{\mathsf{pre}} \mathsf{h}_1\end{array}$$

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

$$\mathsf{if}_{\mathsf{pre}} : (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Bool}) \Rightarrow (1 \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y})) \Rightarrow (1 \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y})) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y})$$
$$\begin{array}{l}\mathsf{if}_{\mathsf{pre}}\ \mathsf{h}_1\ \mathsf{h}_2\ \mathsf{h}_3\ \mathsf{A} := \\ \mathsf{let}\ \ \mathsf{h}_1' := \mathsf{h}_1\ \mathsf{A} \\ \qquad \mathsf{h}_2' := \mathsf{lazy}_{\mathsf{pre}}\ (\mathsf{h}_2\ 0)\ (\mathsf{pre\text{-}ap}\ \mathsf{h}_1'\ \{\mathsf{true}\}) \\ \qquad \mathsf{h}_3' := \mathsf{lazy}_{\mathsf{pre}}\ (\mathsf{h}_3\ 0)\ (\mathsf{pre\text{-}ap}\ \mathsf{h}_1'\ \{\mathsf{false}\}) \\ \quad \mathsf{in}\ \ \mathsf{h}_2' \uplus_{\mathsf{pre}} \mathsf{h}_3'\end{array}$$

$$\mathsf{lazy}_{\mathsf{pre}} : (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y})$$
$$\mathsf{lazy}_{\mathsf{pre}}\ \mathsf{h}\ \mathsf{A} := \mathsf{if}\ (\mathsf{A} = \varnothing)\ (\lambda \varnothing.\, \varnothing)\ (\mathsf{h}\ \mathsf{A})$$

Figure 3: Preimage arrow definitions.

8.3 Case: Composition

Let $\mathcal{C}$ be a sigma-algebra on Z. Starting with the left-hand side of (61), we expand definitions and apply Theorem 2:

$$\begin{array}{l}\mathsf{pre\text{-}ap}\ (\mathsf{arr}_{\mathsf{pre}}\ (\ggg_{\mathsf{map}}\ \mathsf{g}_1\ \mathsf{g}_2)\ \mathsf{A}\ \mathcal{C})\ \mathsf{C} \\ \quad \equiv \mathsf{let}\ \ \mathsf{g}_1' := \mathsf{g}_1\ \mathsf{A} \\ \qquad\quad \mathsf{g}_2' := \mathsf{g}_2\ (\mathsf{range}\ \mathsf{g}_1') \\ \qquad\quad \mathsf{g}' := \mathsf{g}_2' \circ_{\mathsf{map}} \mathsf{g}_1' \\ \qquad\quad \mathcal{C}' := \sigma\text{-trace}\ \mathcal{C}\ (\mathsf{range}\ \mathsf{g}') \\ \qquad \mathsf{in}\ \ \mathsf{pre\text{-}ap}\ (\mathsf{pre}\ \mathsf{g}'\ \mathcal{C})\ \mathsf{C} \\ \quad \equiv \mathsf{let}\ \ \mathsf{g}_1' := \mathsf{g}_1\ \mathsf{A} \\ \qquad\quad \mathsf{g}_2' := \mathsf{g}_2\ (\mathsf{range}\ \mathsf{g}_1') \\ \qquad \mathsf{in}\ \ \mathsf{preimage}\ (\mathsf{g}_2' \circ_{\mathsf{map}} \mathsf{g}_1')\ \mathsf{C}\end{array} \tag{65}$$

Next, we apply Theorem 7 and rewrite in terms of $\mathsf{arr}_{\mathsf{pre}}$:

$$\begin{array}{l}\mathsf{pre\text{-}ap}\ (\mathsf{arr}_{\mathsf{pre}}\ (\ggg_{\mathsf{map}}\ \mathsf{g}_1\ \mathsf{g}_2)\ \mathsf{A})\ \mathsf{C} \\ \quad \equiv \mathsf{let}\ \ \mathsf{g}_1' := \mathsf{g}_1\ \mathsf{A} \\ \qquad\quad \mathcal{B} := \sigma\ (\mathsf{range}\ \mathsf{g}_1') \\ \qquad\quad \mathsf{h}_1 := \mathsf{pre}\ \mathsf{g}_1'\ \mathcal{B} \\ \qquad\quad \mathsf{g}_2' := \mathsf{g}_2\ (\mathsf{range}\ \mathsf{g}_1') \\ \qquad\quad \mathcal{C} := \sigma\ (\mathsf{range}\ \mathsf{g}_2') \\ \qquad\quad \mathsf{h}_2 := \mathsf{pre}\ \mathsf{g}_2'\ \mathcal{C} \\ \qquad \mathsf{in}\ \ \mathsf{pre\text{-}ap}\ (\mathsf{h}_2 \circ_{\mathsf{pre}} \mathsf{h}_1)\ \mathsf{C} \\ \quad \equiv \mathsf{let}\ \ \mathsf{h}_1 := \mathsf{arr}_{\mathsf{pre}}\ \mathsf{g}_1\ \mathsf{A} \\ \qquad\quad \mathsf{h}_2 := \mathsf{arr}_{\mathsf{pre}}\ \mathsf{g}_2\ (\mathsf{range}\ (\mathsf{g}_1\ \mathsf{A})) \\ \qquad \mathsf{in}\ \ \mathsf{pre\text{-}ap}\ (\mathsf{h}_2 \circ_{\mathsf{pre}} \mathsf{h}_1)\ \mathsf{C} \\ \quad \equiv \mathsf{let}\ \ \mathsf{h}_1 := \mathsf{arr}_{\mathsf{pre}}\ \mathsf{g}_1\ \mathsf{A} \\ \qquad\quad \mathsf{h}_2 := \mathsf{arr}_{\mathsf{pre}}\ \mathsf{g}_2\ (\bigcup(\mathsf{domain}\ \mathsf{h}_1)) \\ \qquad \mathsf{in}\ \ \mathsf{pre\text{-}ap}\ (\mathsf{h}_2 \circ_{\mathsf{pre}} \mathsf{h}_1)\ \mathsf{C}\end{array} \tag{66}$$

Substituting h_1 for $\mathsf{arr}_{\mathsf{pre}}\ \mathsf{g}_1$ and h_2 for $\mathsf{arr}_{\mathsf{pre}}\ \mathsf{g}_2$ in the last equality gives the definition for $\ggg_{\mathsf{pre}}$:

$$\begin{array}{l}\ggg_{\mathsf{pre}} : (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}) \Rightarrow (\mathsf{Y} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Z}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Z}) \\ \ggg_{\mathsf{pre}}\ \mathsf{h}_1\ \mathsf{h}_2\ \mathsf{A} := \mathsf{let}\ \ \mathsf{h}_1' := \mathsf{h}_1\ \mathsf{A} \\ \qquad\qquad\qquad\quad \mathsf{h}_2' := \mathsf{h}_2\ (\bigcup(\mathsf{domain}\ \mathsf{h}_1')) \\ \qquad\qquad\quad \mathsf{in}\ \ \mathsf{h}_2 \circ_{\mathsf{pre}} \mathsf{h}_1\end{array} \tag{67}$$

8.4 Case: Conditional

Starting with the left-hand side of (62), expanding terms, and applying Theorem 2 results in

$$\begin{array}{l}\mathsf{pre\text{-}ap}\ (\mathsf{arr}_{\mathsf{pre}}\ (\mathsf{if}_{\mathsf{map}}\ \mathsf{g}_1\ \mathsf{g}_2\ \mathsf{g}_3)\ \mathsf{A})\ \mathsf{B} \\ \quad \equiv \mathsf{let}\ \ \mathsf{g}_1' := \mathsf{g}_1\ \mathsf{A} \\ \qquad\quad \mathsf{g}_2' := \mathsf{lazy}_{\mathsf{map}}\ (\mathsf{g}_2\ 0)\ (\mathsf{preimage}\ \mathsf{g}_1'\ \{\mathsf{true}\}) \\ \qquad\quad \mathsf{g}_3' := \mathsf{lazy}_{\mathsf{map}}\ (\mathsf{g}_3\ 0)\ (\mathsf{preimage}\ \mathsf{g}_1'\ \{\mathsf{false}\}) \\ \qquad \mathsf{in}\ \ \mathsf{preimage}\ (\mathsf{g}_2' \uplus_{\mathsf{map}} \mathsf{g}_3')\ \mathsf{B}\end{array}$$

Next, we apply Theorem 8 and rewrite in terms of $\mathsf{arr}_{\mathsf{pre}}$:

$$\begin{array}{l}\mathsf{pre\text{-}ap}\ (\mathsf{arr}_{\mathsf{pre}}\ (\mathsf{if}_{\mathsf{map}}\ \mathsf{g}_1\ \mathsf{g}_2\ \mathsf{g}_3)\ \mathsf{A})\ \mathsf{B} \\ \quad \equiv \mathsf{let}\ \ \mathsf{g}_1' := \mathsf{g}_1\ \mathsf{A} \\ \qquad\quad \mathsf{g}_2' := \mathsf{lazy}_{\mathsf{map}}\ (\mathsf{g}_2\ 0)\ (\mathsf{preimage}\ \mathsf{g}_1'\ \{\mathsf{true}\}) \\ \qquad\quad \mathcal{B}_2 := \sigma\ (\mathsf{range}\ \mathsf{g}_2') \\ \qquad\quad \mathsf{h}_2 := \mathsf{pre}\ \mathsf{g}_2\ \mathcal{B}_2 \\ \qquad\quad \mathsf{g}_3' := \mathsf{lazy}_{\mathsf{map}}\ (\mathsf{g}_3\ 0)\ (\mathsf{preimage}\ \mathsf{g}_1'\ \{\mathsf{false}\}) \\ \qquad\quad \mathcal{B}_3 := \sigma\ (\mathsf{range}\ \mathsf{g}_3') \\ \qquad\quad \mathsf{h}_3 := \mathsf{pre}\ \mathsf{g}_3\ \mathcal{B}_3 \\ \qquad \mathsf{in}\ \ \mathsf{pre\text{-}ap}\ (\mathsf{h}_2 \uplus_{\mathsf{pre}} \mathsf{h}_3)\ \mathsf{B} \\ \quad \equiv \mathsf{let}\ \ \mathsf{g}_1' := \mathsf{g}_1\ \mathsf{A} \\ \qquad\quad \mathsf{h}_2 := \mathsf{arr}_{\mathsf{pre}}\ (\mathsf{lazy}_{\mathsf{map}}\ (\mathsf{g}_2\ 0))\ (\mathsf{preimage}\ \mathsf{g}_1'\ \{\mathsf{true}\}) \\ \qquad\quad \mathsf{h}_3 := \mathsf{arr}_{\mathsf{pre}}\ (\mathsf{lazy}_{\mathsf{map}}\ (\mathsf{g}_3\ 0))\ (\mathsf{preimage}\ \mathsf{g}_1'\ \{\mathsf{false}\}) \\ \qquad \mathsf{in}\ \ \mathsf{pre\text{-}ap}\ (\mathsf{h}_2 \uplus_{\mathsf{pre}} \mathsf{h}_3)\ \mathsf{B} \\ \quad \equiv \mathsf{let}\ \ \mathsf{h}_1 := \mathsf{arr}_{\mathsf{pre}}\ \mathsf{g}_1\ \mathsf{A} \\ \qquad\quad \mathsf{h}_2 := \mathsf{arr}_{\mathsf{pre}}\ (\mathsf{lazy}_{\mathsf{map}}\ (\mathsf{g}_2\ 0))\ (\mathsf{pre\text{-}ap}\ \mathsf{h}_1\ \{\mathsf{true}\}) \\ \qquad\quad \mathsf{h}_3 := \mathsf{arr}_{\mathsf{pre}}\ (\mathsf{lazy}_{\mathsf{map}}\ (\mathsf{g}_3\ 0))\ (\mathsf{pre\text{-}ap}\ \mathsf{h}_1\ \{\mathsf{false}\}) \\ \qquad \mathsf{in}\ \ \mathsf{pre\text{-}ap}\ (\mathsf{h}_2 \uplus_{\mathsf{pre}} \mathsf{h}_3)\ \mathsf{B}\end{array}$$

Suppose we have defined $\mathsf{lazy}_{\mathsf{pre}}$ so that for any domain subset A,

$$\mathsf{arr}_{\mathsf{pre}}\ (\mathsf{lazy}_{\mathsf{map}}\ \mathsf{g})\ \mathsf{A} \equiv \mathsf{lazy}_{\mathsf{pre}}\ (\mathsf{arr}_{\mathsf{pre}}\ \mathsf{g})\ \mathsf{A} \tag{68}$$

(We discharge this proof obligation after defining $\mathsf{if}_{\mathsf{pre}}$.) Then

$$\begin{array}{l}\mathsf{pre\text{-}ap}\ (\mathsf{arr}_{\mathsf{pre}}\ (\mathsf{if}_{\mathsf{map}}\ \mathsf{g}_1\ \mathsf{g}_2\ \mathsf{g}_3)\ \mathsf{A})\ \mathsf{B} \\ \quad \equiv \mathsf{let}\ \ \mathsf{h}_1 := \mathsf{arr}_{\mathsf{pre}}\ \mathsf{g}_1\ \mathsf{A} \\ \qquad\quad \mathsf{h}_2 := \mathsf{lazy}_{\mathsf{pre}}\ (\mathsf{arr}_{\mathsf{pre}}\ (\mathsf{g}_2\ 0))\ (\mathsf{pre\text{-}ap}\ \mathsf{h}_1\ \{\mathsf{true}\}) \\ \qquad\quad \mathsf{h}_3 := \mathsf{lazy}_{\mathsf{pre}}\ (\mathsf{arr}_{\mathsf{pre}}\ (\mathsf{g}_3\ 0))\ (\mathsf{pre\text{-}ap}\ \mathsf{h}_1\ \{\mathsf{false}\}) \\ \qquad \mathsf{in}\ \ \mathsf{pre\text{-}ap}\ (\mathsf{h}_2 \uplus_{\mathsf{pre}} \mathsf{h}_3)\ \mathsf{B}\end{array}$$

Substituting h_1 for $\mathsf{arr}_{\mathsf{pre}}\ \mathsf{g}_1$, $\mathsf{h}_2\ 0$ for $\mathsf{arr}_{\mathsf{pre}}\ (\mathsf{g}_2\ 0)$, and $\mathsf{h}_3\ 0$ for $\mathsf{arr}_{\mathsf{pre}}\ (\mathsf{g}_3\ 0)$ in the last equality gives the definition for $\mathsf{if}_{\mathsf{pre}}$.

Lastly, starting with the left-hand side of (68),

$$\begin{array}{l}\mathsf{arr}_{\mathsf{pre}}\ (\mathsf{lazy}_{\mathsf{map}}\ \mathsf{g})\ \mathsf{A} \equiv \\ \quad \equiv \mathsf{let}\ \ \mathsf{g}' := \mathsf{if}\ (\mathsf{A} = \varnothing)\ \varnothing\ (\mathsf{g}\ \mathsf{A}) \\ \qquad\quad \mathcal{B} := \sigma\ (\mathsf{range}\ \mathsf{g}') \\ \qquad \mathsf{in}\ \ \mathsf{pre}\ \mathsf{g}'\ \mathcal{B}\end{array}$$

$$\equiv \text{ if } (A = \varnothing)\ (\text{pre}\ \varnothing\ (\sigma\ \varnothing))\ (\text{arr}_{\text{pre}}\ g\ A)$$
$$\equiv \text{ if } (A = \varnothing)\ (\lambda \varnothing.\, \varnothing)\ (\text{arr}_{\text{pre}}\ g\ A)$$

Substituting h for $\text{arr}_{\text{pre}}\ g$ in the last equality gives the definition for lazy_{pre}.

8.5 Super-Saver Theorems

The following two theorems are easy consequences of the fact that arr_{pre} is a natural transformation.

Corollary 3. arr_{pre}, pair_{pre} *and* $\ggg_{\text{pre}}$ *define an arrow.*

Corollary 4. *Let* $g : X \overset{\sim}{\underset{\text{map}}{\to}} Y$ *and* $h : X \overset{\sim}{\underset{\text{pre}}{\to}} Y$ *such that* $h \underset{\text{map}}{\equiv} \text{arr}_{\text{pre}}\ g$. *Then for all* $A \subseteq X$, $h\ A$ *diverges if and only if* $g\ A$ *diverges.*

XXX: simple reason for this:

Corollary 5. *XXX: measurability*

9. Preimages of Partial Functions

$$\begin{aligned} &\bot_{\text{pre}} : X \overset{\sim}{\underset{\text{pre}}{\to}} \varnothing \\ &\bot_{\text{pre}} \ := \ \text{arr}_{\text{pre}}\ (\text{arr}_{\text{map}}\ \lambda x.\, \bot) \end{aligned} \tag{69}$$

References

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