Running Probabilistic Programs Backwards

Neil Toronto Jay McCarthy
PLT @ Brigham Young University
ntoronto@racket-lang.org jay@byu.edu

Chris Grant
Brigham Young University
grant@math.byu.edu

Abstract

XXX

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

General Terms XXX, XXX

Keywords XXX, XXX

1. Introduction

XXX

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} [**?**], 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.

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

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 $\{\{x\},\{x,y\}\}$. Only the *existence* of encodings into sets is important, as it means data structures inherit a defining characteristic of sets: strictness. More precisely, the lengths of paths to data structure leaves is unbounded, but each path must be finite. Less precisely, data may be "infinitely wide" (such as $\mathbb{R}$) but not "infinitely tall" (such as infinite 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.
- $\mathsf{Set\ x}$ denotes a set with members of type x.

The type $\mathsf{Set\ A}$ 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 $\mathsf{A} \times \mathsf{B}$.

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

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

Shortly, $\varnothing$ is the empty set, ω is the cardinality of the natural numbers, card returns the cardinality of a set, 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, and image applies a function to each member of a set and returns the set of results.

[2] Assuming the existence of a single inaccessible cardinal.

2.1 Additional Functions and Forms

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

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

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

$$\begin{aligned} &\mathsf{restrict} : (\mathsf{A} \Rightarrow \mathsf{B}) \Rightarrow \mathcal{P}\ \mathsf{A} \Rightarrow (\mathsf{A} \rightharpoonup \mathsf{B}) \\ &\mathsf{restrict}\ \mathsf{f}\ \mathsf{A}' \ :=\ \mathsf{image}\ (\lambda \mathsf{x}.\, (\mathsf{x}, \mathsf{f}\ \mathsf{x}))\ \mathsf{A}' \end{aligned} \tag{2}$$

$$\begin{aligned} \mathsf{f}|_{\mathsf{A}} &:\equiv\ \mathsf{restrict}\ \mathsf{f}\ \mathsf{A} \\ \lambda \mathsf{x} \in \mathsf{A}.\, \mathsf{e} &:\equiv\ (\lambda \mathsf{x}.\, \mathsf{e})|_{\mathsf{A}} \end{aligned} \tag{3}$$

$$\begin{aligned} &\mathsf{domain} : (\mathsf{A} \rightharpoonup \mathsf{B}) \Rightarrow \mathsf{Set}\ \mathsf{A} \\ &\mathsf{domain}\ \mathsf{f}\ :=\ \mathsf{image}\ \mathsf{fst}\ \mathsf{f} \end{aligned} \tag{4}$$

$$\begin{aligned} &\mathsf{range} : (\mathsf{A} \rightharpoonup \mathsf{B}) \Rightarrow \mathsf{Set}\ \mathsf{B} \\ &\mathsf{range}\ \mathsf{f}\ :=\ \mathsf{image}\ \mathsf{snd}\ \mathsf{f} \end{aligned} \tag{5}$$

$$\begin{aligned} &\mathsf{preimage} : (\mathsf{A} \rightharpoonup \mathsf{B}) \Rightarrow \mathsf{Set}\ \mathsf{B} \Rightarrow \mathsf{Set}\ \mathsf{A} \\ &\mathsf{preimage}\ \mathsf{f}\ \mathsf{B}\ :=\ \{\mathsf{x} \in \mathsf{domain}\ \mathsf{f} \mid \mathsf{f}\ \mathsf{x} \in \mathsf{B}\} \end{aligned} \tag{6}$$

$$\begin{aligned} &(\circ_{\mathsf{map}}) : (\mathsf{Y} \rightharpoonup \mathsf{Z}) \Rightarrow (\mathsf{X} \rightharpoonup \mathsf{Y}) \Rightarrow (\mathsf{X} \rightharpoonup \mathsf{Z}) \\ &\mathsf{g}_2 \circ_{\mathsf{map}} \mathsf{g}_1\ :=\ \mathsf{let}\ \ \mathsf{A} := \mathsf{preimage}\ \mathsf{g}_1\ (\mathsf{domain}\ \mathsf{g}_2) \\ &\qquad\qquad\qquad \mathsf{in}\ \ \lambda \mathsf{x} \in \mathsf{A}.\, \mathsf{g}_2\ (\mathsf{g}_1\ \mathsf{x}) \end{aligned} \tag{7}$$

$$\begin{aligned} &\langle \cdot, \cdot \rangle_{\mathsf{map}} : (\mathsf{X} \rightharpoonup \mathsf{Y}) \Rightarrow (\mathsf{X} \rightharpoonup \mathsf{Z}) \Rightarrow (\mathsf{X} \rightharpoonup \mathsf{Y} \times \mathsf{Z}) \\ &\langle \mathsf{g}_1, \mathsf{g}_2 \rangle_{\mathsf{map}}\ :=\ \mathsf{let}\ \ \mathsf{A} := (\mathsf{domain}\ \mathsf{g}_1) \cap (\mathsf{domain}\ \mathsf{g}_2) \\ &\qquad\qquad\qquad \mathsf{in}\ \ \lambda \mathsf{x} \in \mathsf{A}.\, \langle \mathsf{g}_1\ \mathsf{x}, \mathsf{g}_2\ \mathsf{x} \rangle \end{aligned} \tag{8}$$

2.2 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

$$\begin{aligned} 0\ &:=\ \varnothing \qquad & 2\ &:=\ \{0, 1\} \\ 1\ &:=\ \{0\} \qquad & 3\ &:=\ \{0, 1, 2\} \end{aligned} \tag{9}$$

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

$$\begin{aligned} \omega\ &:=\ \{0, 1, 2, 3, ...\} \\ \omega + 1\ &:=\ \{0, 1, 2, 3, ..., \omega\} \\ \omega + 2\ &:=\ \{0, 1, 2, 3, ..., \omega, \omega + 1\} \\ \omega + \omega\ &:=\ \{0, 1, 2, 3, ..., \omega, \omega + 1, \omega + 2, ...\} \end{aligned} \tag{10}$$

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:

$$\begin{aligned} &\mathsf{succs} : \omega \Rightarrow \mathsf{Set}\ \mathbb{R} \Rightarrow \mathsf{Set}\ \mathbb{R} \\ &\mathsf{succs}\ \mathsf{n}\ \mathsf{A}\ := \\ &\quad \mathsf{case}\ \ \mathsf{n} \\ &\qquad 0 \quad\ \Longrightarrow\ \mathsf{A} \\ &\qquad \mathsf{m} + 1\ \Longrightarrow\ \mathsf{A} \cup (\mathsf{image}\ (+\ 1.0)\ (\mathsf{succs}\ \mathsf{m}\ \mathsf{A})) \end{aligned} \tag{11}$$

This unfold over finite ordinals generates prefixes such as

$$\begin{aligned} \mathsf{succs}\ 0\ \{0.5\}\ &=\ \{0.5\} \\ \mathsf{succs}\ 1\ \{0.5\}\ &=\ \{0.5, 1.5\} \\ \mathsf{succs}\ 2\ \{0.5\}\ &=\ \{0.5, 1.5, 2.5\} \end{aligned} \tag{12}$$

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:

$$\begin{aligned} &\mathsf{succs} : \mathsf{Ord} \Rightarrow \mathsf{Set}\ \mathbb{R} \Rightarrow \mathsf{Set}\ \mathbb{R} \\ &\mathsf{succs}\ \alpha\ \mathsf{A}\ := \\ &\quad \mathsf{case}\ \ \alpha \\ &\qquad 0 \quad\ \Longrightarrow\ \mathsf{A} \\ &\qquad \beta + 1\ \Longrightarrow\ \mathsf{A} \cup (\mathsf{image}\ (+\ 1.0)\ (\mathsf{succs}\ \beta\ \mathsf{A})) \\ &\qquad \mathsf{else}\ \Longrightarrow\ \bigcup_{\beta < \alpha}\ \mathsf{succs}\ \beta\ \mathsf{A} \end{aligned} \tag{13}$$

With this, we can compute the closure as desired:

$$\begin{aligned} \mathsf{succs}\ \omega\ \{0.5\}\ &=\ (\mathsf{succs}\ 0\ \{0.5\}) \cup (\mathsf{succs}\ 1\ \{0.5\}) \cup \cdots \\ &=\ \{0.5, 1.5, 2.5, 3.5, ...\} \end{aligned} \tag{14}$$

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

$$\begin{aligned} &\mathsf{close} : (\mathsf{Set}\ \mathsf{x} \Rightarrow \mathsf{Set}\ \mathsf{x}) \Rightarrow \mathsf{Ord} \Rightarrow \mathsf{Set}\ \mathsf{x} \Rightarrow \mathsf{Set}\ \mathsf{x} \\ &\mathsf{close}\ \mathsf{f}\ \alpha\ \mathsf{A}\ := \\ &\quad \mathsf{case}\ \ \alpha \\ &\qquad 0 \quad\ \Longrightarrow\ \mathsf{A} \\ &\qquad \beta + 1\ \Longrightarrow\ \mathsf{A} \cup (\mathsf{f}\ (\mathsf{close}\ \mathsf{f}\ \beta\ \mathsf{A})) \\ &\qquad \mathsf{else}\ \Longrightarrow\ \bigcup_{\beta < \alpha}\ \mathsf{close}\ \mathsf{f}\ \beta\ \mathsf{A} \end{aligned} \tag{15}$$

With this, $\mathsf{succs} = \mathsf{close}\ (\mathsf{image}\ (+\ 1.0))$.

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:

$$\begin{aligned} &\sigma\text{-}\mathsf{comps} : \mathsf{Set}\ (\mathsf{Set}\ \mathsf{x}) \Rightarrow \mathsf{Set}\ (\mathsf{Set}\ \mathsf{x}) \\ &\sigma\text{-}\mathsf{comps}\ \mathcal{A}\ :=\ \{\mathsf{A} \backslash \mathsf{A}' \mid \mathsf{A}, \mathsf{A}' \in \mathcal{A}\} \\ \\ &\sigma\text{-}\mathsf{unions} : \mathsf{Set}\ (\mathsf{Set}\ \mathsf{x}) \Rightarrow \mathsf{Set}\ (\mathsf{Set}\ \mathsf{x}) \\ &\sigma\text{-}\mathsf{unions}\ \mathcal{A}\ :=\ \{\bigcup \mathcal{A}' \mid \mathcal{A}' \subseteq \mathcal{A} \wedge |\mathcal{A}'| \leq \omega\} \\ \\ &\sigma\text{-}\mathsf{ops} : \mathsf{Set}\ (\mathsf{Set}\ \mathsf{x}) \Rightarrow \mathsf{Set}\ (\mathsf{Set}\ \mathsf{x}) \\ &\sigma\text{-}\mathsf{ops}\ \mathcal{A}\ :=\ \{\varnothing\} \cup (\sigma\text{-}\mathsf{comps}\ \mathcal{A}) \cup (\sigma\text{-}\mathsf{unions}\ \mathcal{A}) \end{aligned} \tag{16}$$

the following function identifies σ-algebras:

$$\begin{aligned} &\sigma\text{-}\mathsf{algebra?} : \mathsf{Set}\ (\mathsf{Set}\ \mathsf{x}) \Rightarrow \mathsf{Bool} \\ &\sigma\text{-}\mathsf{algebra?}\ \mathcal{A}\ :=\ (\sigma\text{-}\mathsf{ops}\ \mathcal{A}) \subseteq \mathcal{A} \end{aligned} \tag{17}$$

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 σ-algebra $\mathcal{A}$ with a set A is the result of intersecting every $\mathsf{A}' \in \mathcal{A}$ with A:

$$\begin{array}{l}\sigma\text{-}\mathsf{trace} : \mathsf{Set}\ (\mathsf{Set}\ \mathsf{x}) \Rightarrow \mathsf{Set}\ \mathsf{x} \Rightarrow \mathsf{Set}\ (\mathsf{Set}\ \mathsf{x}) \\ \sigma\text{-}\mathsf{trace}\ \mathcal{A}\ \mathsf{A}\ :=\ \{\mathsf{A} \cap \mathsf{A}' \mid \mathsf{A}' \in \mathcal{A}\}\end{array} \tag{18}$$

There are no restrictions on A. In particular, it does not have to be in $\mathcal{A}$ (i.e. measurable).

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

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:

$$\begin{array}{l}\sigma\text{-}\mathsf{close} : \mathsf{Set}\ (\mathsf{Set}\ \mathsf{x}) \Rightarrow \mathsf{Set}\ (\mathsf{Set}\ \mathsf{x}) \\ \sigma\text{-}\mathsf{close}\ :=\ \mathsf{close}\ \sigma\text{-}\mathsf{ops}\ \omega_1\end{array} \tag{19}$$

Here, ω_1 is the **first uncountable ordinal**, or the least ordinal containing every countable ordinal.[3]

Lemma 2 (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 $\mathcal{A} \subseteq \mathcal{A}'$.*

The most well-studied generated σ-algebras are **Borel σ-algebras**: those generated from topologies. For example, if $\tau_{\mathbb{R}}$ is the standard topology on $\mathbb{R}$, containing the open intervals and uncountable unions of open intervals, then

$$\sigma\ \mathbb{R}\ =\ \sigma\text{-}\mathsf{close}\ \tau_{\mathbb{R}} \tag{20}$$

is the Borel σ-algebra for that topology, containing all intervals (open, closed and half-open), and their countable unions and complements. (We will formally define the σ function further on.) We call this the ***standard σ-algebra*** for $\mathbb{R}$.

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

$$\begin{array}{l}(\boxtimes) : \mathsf{Set}\ (\mathsf{Set}\ \mathsf{x}) \Rightarrow \mathsf{Set}\ (\mathsf{Set}\ \mathsf{y}) \Rightarrow \mathsf{Set}\ (\mathsf{Set}\ \langle \mathsf{x}, \mathsf{y} \rangle) \\ \mathcal{A}_1 \boxtimes \mathcal{A}_2\ :=\ \{\mathsf{A}_1 \times \mathsf{A}_2 \mid \mathsf{A}_1 \in \mathcal{A}_1, \mathsf{A}_2 \in \mathcal{A}_2\} \\ \\ (\otimes) : \mathsf{Set}\ (\mathsf{Set}\ \mathsf{x}) \Rightarrow \mathsf{Set}\ (\mathsf{Set}\ \mathsf{y}) \Rightarrow \mathsf{Set}\ (\mathsf{Set}\ \langle \mathsf{x}, \mathsf{y} \rangle) \\ \mathcal{A}_1 \otimes \mathcal{A}_2\ :=\ \sigma\text{-}\mathsf{close}\ (\mathcal{A}_1 \boxtimes \mathcal{A}_2)\end{array} \tag{21}$$

A product σ-algebra is the smallest σ-algebra for which pair projections (fst and snd) are measurable.

Lemma 3 (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] λ_{ZFC} does not decide the continuum hypothesis; i.e. whether $\omega_1 = |\mathbb{R}|$.

3.4 Measures and Probability

The mapping $\mu \in \mathcal{P}\ \mathsf{X} \rightharpoonup [0, \infty)$ is a **measure** if

- $\mathsf{domain}\ \mu$ is a σ-algebra.
- $\mu\ \varnothing = 0$.
- For all $\mathcal{A}' \subseteq \mathcal{A}$ countable, $\sum_{\mathsf{A} \in \mathcal{A}'} (\mu\ \mathsf{A}) = \mu\ (\bigcup \mathcal{A}')$.

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

$$\mathbb{P}\ (\mathsf{preimage}\ \mathsf{f}\ \mathsf{B}) \tag{22}$$

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 $\bot$ instead of Just and Maybe), simplified; arrow laws therefore hold (XXX: check terminology)

XXX: point out that $\mathsf{if}_{\bot}$ 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

$$\mathsf{X} \overset{\rightrightarrows}{\mathsf{map}} \mathsf{Y}\ ::=\ \mathsf{Set}\ \mathsf{X} \Rightarrow (\mathsf{X} \rightharpoonup \mathsf{Y}) \tag{23}$$

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

XXX: motivate removal of bottom

Lifting a bottom arrow computation $\mathsf{f} : \mathsf{X} \Rightarrow \mathsf{Y}_{\bot}$ to the mapping arrow requires restricting f's domain to a subset of X for which f does not return $\bot$. It is helpful to have a standalone function $\mathsf{domain}_{\bot}$ that computes such domains, so we define that first, and $\mathsf{arr}_{\mathsf{map}}$ in terms of $\mathsf{domain}_{\bot}$:

$$\begin{array}{l}\mathsf{domain}_{\bot} : (\mathsf{X} \Rightarrow \mathsf{Y}_{\bot}) \Rightarrow \mathsf{Set}\ \mathsf{X} \Rightarrow \mathsf{Set}\ \mathsf{X} \\ \mathsf{domain}_{\bot}\ \mathsf{f}\ \mathsf{A}\ :=\ \mathsf{preimage}\ \mathsf{f}|_{\mathsf{A}}\ ((\mathsf{image}\ \mathsf{f}\ \mathsf{A}) \backslash \{\bot\}) \\ \\ \mathsf{arr}_{\mathsf{map}} : (\mathsf{X} \Rightarrow \mathsf{Y}_{\bot}) \Rightarrow (\mathsf{X} \overset{\rightrightarrows}{\mathsf{map}} \mathsf{Y}) \\ \mathsf{arr}_{\mathsf{map}}\ \mathsf{f}\ \mathsf{A}\ :=\ \mathsf{let}\ \ \mathsf{A}' := \mathsf{domain}_{\bot}\ \mathsf{f}\ \mathsf{A} \\ \qquad\qquad\qquad\qquad \mathsf{in}\ \ \mathsf{f}|_{\mathsf{A}'}\end{array} \tag{24}$$

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

Definition 1 (Mapping arrow equivalence)**.** *Two mapping arrow computations $\mathsf{g}_1 : \mathsf{X} \overset{\rightrightarrows}{\mathsf{map}} \mathsf{Y}$ and $\mathsf{g}_2 : \mathsf{X} \overset{\rightrightarrows}{\mathsf{map}} \mathsf{Y}$ are equivalent, written $\mathsf{g}_1 \overset{=}{\mathsf{map}} \mathsf{g}_2$, when $\mathsf{g}_1\ \mathsf{A} = \mathsf{g}_2\ \mathsf{A}$ for all $\mathsf{A} \subseteq \mathsf{X}$.*

$$\begin{aligned}
&\mathsf{arr}_\bot : (\mathsf{X} \Rightarrow \mathsf{Y}_\bot) \Rightarrow (\mathsf{X} \Rightarrow \mathsf{Y}_\bot)\\
&\mathsf{arr}_\bot\ \mathsf{f} \ :=\ \mathsf{f}\\
\\
&\ggg_\bot : (\mathsf{X} \Rightarrow \mathsf{Y}_\bot) \Rightarrow (\mathsf{Y} \Rightarrow \mathsf{Z}_\bot) \Rightarrow (\mathsf{X} \Rightarrow \mathsf{Z}_\bot)\\
&\ggg_\bot\ \mathsf{f}_1\ \mathsf{f}_2\ \mathsf{x} \ :=\ \mathsf{if}\ (\mathsf{f}_1\ \mathsf{x} = \bot)\ \bot\ (\mathsf{f}_2\ (\mathsf{f}_1\ \mathsf{x}))\\
\\
&\mathsf{pair}_\bot : (\mathsf{X} \Rightarrow \mathsf{Y}_\bot) \Rightarrow (\mathsf{X} \Rightarrow \mathsf{Z}_\bot) \Rightarrow (\mathsf{X} \Rightarrow \langle \mathsf{Y}, \mathsf{Z}\rangle_\bot)\\
&\mathsf{pair}_\bot\ \mathsf{f}_2\ \mathsf{f}_2\ \mathsf{x} \ :=\ \mathsf{if}\ (\mathsf{f}_1\ \mathsf{x} = \bot \vee \mathsf{f}_2\ \mathsf{x} = \bot)\ \bot\ \langle \mathsf{f}_1\ \mathsf{x}, \mathsf{f}_2\ \mathsf{x}\rangle
\end{aligned}$$

$$\begin{aligned}
&\mathsf{if}_\bot : (\mathsf{X} \Rightarrow \mathsf{Bool}_\bot) \Rightarrow (1 \Rightarrow (\mathsf{X} \Rightarrow \mathsf{Y}_\bot)) \Rightarrow (1 \Rightarrow (\mathsf{X} \Rightarrow \mathsf{Y}_\bot))\\
&\qquad \Rightarrow (\mathsf{X} \Rightarrow \mathsf{Y}_\bot)\\
&\mathsf{if}_\bot\ \mathsf{f}_1\ \mathsf{f}_2\ \mathsf{f}_3\ \mathsf{x} \ :=\ \mathsf{case}\ \ \mathsf{f}_1\ \mathsf{x}\\
&\qquad\qquad\qquad\quad \mathsf{true} \implies \mathsf{f}_2\ 0\ \mathsf{x}\\
&\qquad\qquad\qquad\quad \mathsf{false} \implies \mathsf{f}_3\ 0\ \mathsf{x}\\
&\qquad\qquad\qquad\quad \mathsf{else} \implies \bot
\end{aligned}$$

Figure 1: Bottom arrow definitions.

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:

$$\mathsf{arr}_{\mathsf{map}}\ (\mathsf{pair}_\bot\ \mathsf{f}_1\ \mathsf{f}_2) \underset{\widetilde{\mathsf{map}}}{=} \mathsf{pair}_{\mathsf{map}}\ (\mathsf{arr}_{\mathsf{map}}\ \mathsf{f}_1)\ (\mathsf{arr}_{\mathsf{map}}\ \mathsf{f}_2) \quad (25)$$

$$\mathsf{arr}_{\mathsf{map}}\ (\ggg_\bot\ \mathsf{f}_1\ \mathsf{f}_2) \underset{\widetilde{\mathsf{map}}}{=} \ggg_{\mathsf{map}}\ (\mathsf{arr}_{\mathsf{map}}\ \mathsf{f}_1)\ (\mathsf{arr}_{\mathsf{map}}\ \mathsf{f}_2) \quad (26)$$

$$\begin{aligned}
&\mathsf{arr}_{\mathsf{map}}\ (\mathsf{if}_\bot\ \mathsf{f}_1\ \mathsf{f}_2\ \mathsf{f}_3) \underset{\widetilde{\mathsf{map}}}{=}\\
&\quad \mathsf{if}_{\mathsf{map}}\ (\mathsf{arr}_{\mathsf{map}}\ \mathsf{f}_1)\ (\lambda 0.\ \mathsf{arr}_{\mathsf{map}}\ (\mathsf{f}_2\ 0))\ (\lambda 0.\ \mathsf{arr}_{\mathsf{map}}\ (\mathsf{f}_3\ 0))
\end{aligned} \quad (27)$$

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

We ensure (25) holds by construction and equational reasoning. Starting with its left-hand side, we first expand definitions. For any $\mathsf{A} : \mathsf{Set}\ \mathsf{X}$,

$$\begin{aligned}
&\mathsf{arr}_{\mathsf{map}}\ (\mathsf{pair}_\bot\ \mathsf{f}_1\ \mathsf{f}_2)\ \mathsf{A}\\
&\quad = \ \mathsf{arr}_{\mathsf{map}}\ (\lambda \mathsf{x}.\ \mathsf{if}\ (\mathsf{f}_1\ \mathsf{x} = \bot \vee \mathsf{f}_2\ \mathsf{x} = \bot)\ \bot\ \langle \mathsf{f}_1\ \mathsf{x}, \mathsf{f}_2\ \mathsf{x}\rangle)\ \mathsf{A}\\
&\quad = \ \mathsf{let} \quad \mathsf{f} := \lambda \mathsf{x}.\ \mathsf{if}\ (\mathsf{f}_1\ \mathsf{x} = \bot \vee \mathsf{f}_2\ \mathsf{x} = \bot)\ \bot\ \langle \mathsf{f}_1\ \mathsf{x}, \mathsf{f}_2\ \mathsf{x}\rangle\\
&\qquad\quad\ \ \mathsf{A}' := \mathsf{domain}_\bot\ \mathsf{f}\ \mathsf{A}\\
&\qquad\ \ \mathsf{in} \ \ \mathsf{f}|_{\mathsf{A}'}
\end{aligned} \quad (28)$$

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

$$\begin{aligned}
&\mathsf{arr}_{\mathsf{map}}\ (\mathsf{pair}_\bot\ \mathsf{f}_1\ \mathsf{f}_2)\ \mathsf{A}\\
&\quad = \ \mathsf{let} \ \ \mathsf{A}_1 := (\mathsf{domain}_\bot\ \mathsf{f}_1\ \mathsf{A})\\
&\qquad\quad\ \ \mathsf{A}_2 := (\mathsf{domain}_\bot\ \mathsf{f}_2\ \mathsf{A})\\
&\qquad\quad\ \ \mathsf{A}' := \mathsf{A}_1 \cap \mathsf{A}_2\\
&\qquad\ \ \mathsf{in} \ \ \lambda \mathsf{x} \in \mathsf{A}'.\ \langle \mathsf{f}_1\ \mathsf{x}, \mathsf{f}_2\ \mathsf{x}\rangle\\
&\quad = \ \mathsf{let} \ \ \mathsf{g}_1 := \mathsf{arr}_{\mathsf{map}}\ \mathsf{f}_1\ \mathsf{A}\\
&\qquad\quad\ \ \mathsf{g}_2 := \mathsf{arr}_{\mathsf{map}}\ \mathsf{f}_2\ \mathsf{A}\\
&\qquad\quad\ \ \mathsf{A}' := (\mathsf{domain}\ \mathsf{g}_1) \cap (\mathsf{domain}\ \mathsf{g}_2)\\
&\qquad\ \ \mathsf{in} \ \ \lambda \mathsf{x} \in \mathsf{A}'.\ \langle \mathsf{g}_1\ \mathsf{x}, \mathsf{g}_2\ \mathsf{x}\rangle\\
&\quad = \ \langle \mathsf{arr}_{\mathsf{map}}\ \mathsf{f}_1\ \mathsf{A}, \mathsf{arr}_{\mathsf{map}}\ \mathsf{f}_2\ \mathsf{A}\rangle_{\mathsf{map}}
\end{aligned} \quad (29)$$

Substituting g_1 for $\mathsf{arr}_{\mathsf{map}}\ \mathsf{f}_1$ and g_2 for $\mathsf{arr}_{\mathsf{map}}\ \mathsf{f}_2$ in the last equality gives the definition for $\mathsf{pair}_{\mathsf{map}}$:

$$\begin{aligned}
&\mathsf{pair}_{\mathsf{map}} : (\mathsf{X} \overset{\sim}{\underset{\mathsf{map}}{\rightsquigarrow}} \mathsf{Y}) \Rightarrow (\mathsf{X} \overset{\sim}{\underset{\mathsf{map}}{\rightsquigarrow}} \mathsf{Z}) \Rightarrow (\mathsf{X} \overset{\sim}{\underset{\mathsf{map}}{\rightsquigarrow}} \langle \mathsf{Y}, \mathsf{Z}\rangle)\\
&\mathsf{pair}_{\mathsf{map}}\ \mathsf{g}_1\ \mathsf{g}_2\ \mathsf{A} \ :=\ \langle \mathsf{g}_1\ \mathsf{A}, \mathsf{g}_2\ \mathsf{A}\rangle_{\mathsf{map}}
\end{aligned} \quad (30)$$

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:

$$\begin{aligned}
&\mathsf{halts\text{-}on\text{-}true}_\bot \ :=\\
&\quad \mathsf{if}_\bot\ \mathsf{id}\ (\lambda 0.\ \mathsf{id})\ (\lambda 0.\ \mathsf{halts\text{-}on\text{-}true}_\bot)
\end{aligned} \quad (31)$$

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

Starting with the left-hand-side of (27), we expand definitions, and simplify f by restricting it to a domain for which $\mathsf{f}_1\ \mathsf{x}$ cannot be $\bot$:

$$\begin{aligned}
&\mathsf{arr}_{\mathsf{map}}\ (\mathsf{if}_\bot\ \mathsf{f}_1\ \mathsf{f}_2\ \mathsf{f}_3)\ \mathsf{A}\\
&\quad = \ \mathsf{let} \quad \mathsf{f} := \lambda \mathsf{x}.\ \mathsf{case}\ \ \mathsf{f}_1\ \mathsf{x}\\
&\qquad\qquad\qquad\quad \mathsf{true} \implies \mathsf{f}_2\ 0\ \mathsf{x}\\
&\qquad\qquad\qquad\quad \mathsf{false} \implies \mathsf{f}_3\ 0\ \mathsf{x}\\
&\qquad\qquad\qquad\quad \mathsf{else} \implies \bot\\
&\qquad\quad\ \ \mathsf{A}' := \mathsf{domain}_\bot\ \mathsf{f}\ \mathsf{A}\\
&\qquad\ \ \mathsf{in} \ \ \mathsf{f}|_{\mathsf{A}'}\\
&\quad = \ \mathsf{let} \ \ \mathsf{A}_2 := \mathsf{preimage}\ \mathsf{f}_1|_{\mathsf{A}}\ \{\mathsf{true}\}\\
&\qquad\quad\ \ \mathsf{A}_3 := \mathsf{preimage}\ \mathsf{f}_1|_{\mathsf{A}}\ \{\mathsf{false}\}\\
&\qquad\quad\ \ \mathsf{f} := \lambda \mathsf{x}.\ \mathsf{if}\ (\mathsf{f}_1\ \mathsf{x})\ (\mathsf{f}_2\ 0\ \mathsf{x})\ (\mathsf{f}_3\ 0\ \mathsf{x})\\
&\qquad\quad\ \ \mathsf{A}' := \mathsf{domain}_\bot\ \mathsf{f}\ (\mathsf{A}_2 \cup \mathsf{A}_3)\\
&\qquad\ \ \mathsf{in} \ \ \mathsf{f}|_{\mathsf{A}'}
\end{aligned} \quad (32)$$

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

$$\begin{aligned}
&\mathsf{arr}_{\mathsf{map}}\ (\mathsf{if}_\bot\ \mathsf{f}_1\ \mathsf{f}_2\ \mathsf{f}_3)\ \mathsf{A}\\
&\quad = \ \mathsf{let} \ \ \mathsf{g}_1 := \mathsf{arr}_{\mathsf{map}}\ \mathsf{f}_1\ \mathsf{A}\\
&\qquad\quad\ \ \mathsf{A}_2 := \mathsf{preimage}\ \mathsf{g}_1\ \{\mathsf{true}\}\\
&\qquad\quad\ \ \mathsf{A}_3 := \mathsf{preimage}\ \mathsf{g}_1\ \{\mathsf{false}\}\\
&\qquad\quad\ \ \mathsf{g}_2 := \mathsf{arr}_{\mathsf{map}}\ (\mathsf{f}_2\ 0)\ \mathsf{A}_2\\
&\qquad\quad\ \ \mathsf{g}_3 := \mathsf{arr}_{\mathsf{map}}\ (\mathsf{f}_3\ 0)\ \mathsf{A}_3\\
&\qquad\quad\ \ \mathsf{A}' := (\mathsf{domain}\ \mathsf{g}_2) \cup (\mathsf{domain}\ \mathsf{g}_3)\\
&\qquad\ \ \mathsf{in} \ \ \lambda \mathsf{x} \in \mathsf{A}'.\ \mathsf{if}\ (\mathsf{g}_1\ \mathsf{x})\ (\mathsf{g}_2\ \mathsf{x})\ (\mathsf{g}_3\ \mathsf{x})
\end{aligned} \quad (33)$$

This is close to correct. Unfortunately, for $\mathsf{halts\text{-}on\text{-}true}_\bot$, computing $\mathsf{g}_3 := \mathsf{arr}_{\mathsf{map}}\ (\mathsf{f}_3\ 0)\ \mathsf{A}_3$ always diverges. Wrapping

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

$$\begin{array}{l}
\text{arr}_{\text{map}} : (X \Rightarrow Y_\perp) \Rightarrow (X \overset{\rightsquigarrow}{\underset{\text{map}}{}} Y) \\
\text{arr}_{\text{map}}\ f\ A \ := \ \text{let}\ \ A' := \text{domain}_\perp\ f\ A \\
\qquad\qquad\qquad\qquad \text{in}\ \ f|_{A'}
\end{array}$$

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

$$\begin{array}{l}
\text{pair}_{\text{map}} : (X \overset{\rightsquigarrow}{\underset{\text{map}}{}} Y) \Rightarrow (X \overset{\rightsquigarrow}{\underset{\text{map}}{}} Z) \Rightarrow (X \overset{\rightsquigarrow}{\underset{\text{map}}{}} \langle Y, Z\rangle) \\
\text{pair}_{\text{map}}\ g_1\ g_2\ A \ := \ \langle g_1\ A, g_2\ A\rangle_{\text{map}}
\end{array}$$

$$\begin{array}{l}
\text{if}_{\text{map}} : (X \overset{\rightsquigarrow}{\underset{\text{map}}{}} \text{Bool}) \Rightarrow (1 \Rightarrow (X \overset{\rightsquigarrow}{\underset{\text{map}}{}} Y)) \Rightarrow (1 \Rightarrow (X \overset{\rightsquigarrow}{\underset{\text{map}}{}} Y)) \\
\qquad \Rightarrow (X \overset{\rightsquigarrow}{\underset{\text{map}}{}} Y) \\
\text{if}_{\text{map}}\ g_1\ g_2\ g_3\ A \ := \\
\quad \text{let}\ \ g_1' := g_1\ A \\
\qquad\ g_2' := \text{lazy}_{\text{map}}\ (g_2\ 0)\ (\text{preimage}\ g_1'\ \{\text{true}\}) \\
\qquad\ g_3' := \text{lazy}_{\text{map}}\ (g_3\ 0)\ (\text{preimage}\ g_1'\ \{\text{false}\}) \\
\qquad\ A' := (\text{domain}\ g_2') \cup (\text{domain}\ g_3') \\
\quad \text{in}\ \ \lambda x \in A'.\, \text{if}\ (g_1'\ x)\ (g_2'\ x)\ (g_3'\ x)
\end{array}$$

$$\begin{array}{l}
\text{domain}_\perp : (X \Rightarrow Y_\perp) \Rightarrow \text{Set}\ X \Rightarrow \text{Set}\ X \\
\text{domain}_\perp\ f\ A \ := \ \text{preimage}\ f|_A\ ((\text{image}\ f\ A) \backslash \{\perp\})
\end{array}$$

$$\begin{array}{l}
\text{lazy}_{\text{map}} : (X \overset{\rightsquigarrow}{\underset{\text{map}}{}} Y) \Rightarrow (X \overset{\rightsquigarrow}{\underset{\text{map}}{}} Y) \\
\text{lazy}_{\text{map}}\ g\ A \ := \ \text{if}\ (A = \varnothing)\ \varnothing\ (g\ A)
\end{array}$$

Figure 2: Mapping arrow definitions.

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 $\text{arr}_{\text{map}}\ (f_2\ 0)$ and $\text{arr}_{\text{map}}\ (f_3\ 0)$ only to nonempty sets, using

$$\begin{array}{l}
\text{lazy}_{\text{map}} : (X \overset{\rightsquigarrow}{\underset{\text{map}}{}} Y) \Rightarrow (X \overset{\rightsquigarrow}{\underset{\text{map}}{}} Y) \\
\text{lazy}_{\text{map}}\ f\ A \ := \ \text{if}\ (A = \varnothing)\ \varnothing\ (f\ A)
\end{array}$$

$$\begin{array}{l}
\text{arr}_{\text{map}}\ (\text{if}_\perp\ f_1\ f_2\ f_3)\ A \\
\quad = \ \text{let}\ \ g_1 := \text{arr}_{\text{map}}\ f_1\ A \\
\qquad\quad\ g_2 := \text{lazy}_{\text{map}}\ (\text{arr}_{\text{map}}\ (f_2\ 0))\ (\text{preimage}\ g_1\ \{\text{true}\}) \\
\qquad\quad\ g_3 := \text{lazy}_{\text{map}}\ (\text{arr}_{\text{map}}\ (f_3\ 0))\ (\text{preimage}\ g_1\ \{\text{false}\}) \\
\qquad\quad\ A' := (\text{domain}\ g_2) \cup (\text{domain}\ g_3) \\
\qquad\ \text{in}\ \ \lambda x \in A'.\, \text{if}\ (g_1\ x)\ (g_2\ x)\ (g_3\ x)
\end{array} \tag{34}$$

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

5.5 Super-Saver Theorems

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

Corollary 1. *Let* $f : X \Rightarrow Y_\perp$, $A \subseteq X$, *and* $g : X \overset{\rightsquigarrow}{\underset{\text{map}}{}} Y$ *such that* $g \overset{=}{\underset{\text{map}}{}} \text{arr}_{\text{map}}\ f$. *Then* $g\ A$ *diverges if and only if there exists an* $x \in A$ *for which* $f\ x$ *diverges.*

Corollary 2. arr_{map}, pair_{map} *and* $\ggg_{\text{map}}$ *define an arrow.*

6. Preimage Mappings

XXX: almost ready to derive the preimage arrow; unfortunately, data structures used in measure theory do not always lend themselves to approximation

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.

To compute with abstract, infinite sets in the language implementation, we need to be able to work with preimages in the semantics without resorting to applying functions to points. We therefore introduce ***preimage mappings***. The function space is constructed by

$$X \overset{\rightarrow}{\underset{\text{pre}}{}} Y \ := \ \mathcal{P}\ Y \rightharpoonup \mathcal{P}\ X \tag{35}$$

XXX: converting a mapping to a preimage mapping:

$$\begin{array}{l}
\text{pre} : (X \rightharpoonup Y) \Rightarrow \text{Set}\ (\text{Set}\ Y) \Rightarrow (X \overset{\rightarrow}{\underset{\text{pre}}{}} Y) \\
\text{pre}\ g\ \mathcal{B} \ := \ \lambda B \in \mathcal{B}.\, \text{preimage}\ g\ B
\end{array} \tag{36}$$

Note: $\mathcal{B}$ is not necessarily a σ-algebra

XXX: preimage mappings are defined for the *range* of a mapping, but preimages can be computed for any subset of a mapping's *codomain*; use this function to compute preimages of any measurable codomain subset:

$$\begin{array}{l}
\text{pre-ap} : (X \overset{\rightarrow}{\underset{\text{pre}}{}} Y) \Rightarrow \text{Set}\ Y \Rightarrow \text{Set}\ X \\
\text{pre-ap}\ h\ B \ := \ h\ (B \cap \bigcup(\text{domain}\ h))
\end{array} \tag{37}$$

XXX: for measurable sets in a function's codomain, using pre-ap to compute preimages from a preimage mapping is the same as computing them from the original mapping:

Theorem 2 (pre-ap of measurable sets)**.** *Let* $g \in X \rightharpoonup Y$, $\mathcal{B}'$ *a* σ*-algebra on* Y, *and* $\mathcal{B} := \sigma\text{-trace}\ \mathcal{B}'\ (\text{range}\ g)$. *Then for all* $B \in \mathcal{B}'$, $\text{pre-ap}\ (\text{pre}\ g\ \mathcal{B})\ B = \text{preimage}\ g\ B$.

Proof. Expanding the definitions of pre-ap and pre, and substituting $\bigcup(\text{domain}\ h) = \text{range}\ g$ yields

$$\begin{array}{rcl}
\text{pre-ap}\ (\text{pre}\ g\ \mathcal{B})\ B & = & \text{let}\ \ h := \lambda B \in \mathcal{B}.\, \text{preimage}\ g\ B \\
& & \qquad \text{in}\ \ h\ (B \cap \bigcup(\text{domain}\ h)) \\
& = & \text{let}\ \ h := \lambda B \in \mathcal{B}.\, \text{preimage}\ g\ B \\
& & \qquad \text{in}\ \ h\ (B \cap (\text{range}\ g))
\end{array} \tag{38}$$

By definition of σ-trace, $B \cap (\text{range}\ g) \in \mathcal{B}'$, so

$$\begin{array}{rcl}
\text{pre-ap}\ (\text{pre}\ g\ \mathcal{B})\ B & = & \text{preimage}\ g\ (B \cap (\text{range}\ g)) \\
& = & \text{preimage}\ g\ B
\end{array} \tag{39}$$

□

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:

$$\begin{array}{l}\mathsf{pre\text{-}comps} : (\mathsf{X} \rightrightarrows_{\mathsf{pre}} \mathsf{Y}) \Rightarrow (\mathsf{X} \rightrightarrows_{\mathsf{pre}} \mathsf{Y}) \\ \mathsf{pre\text{-}comps\ h} := \\ \quad \mathsf{let}\ \ \mathcal{B} := \mathsf{domain\ h} \\ \quad\quad \mathsf{in}\ \ \{\langle \mathsf{B}_1 \backslash \mathsf{B}_2, (\mathsf{h}\ \mathsf{B}_1) \backslash (\mathsf{h}\ \mathsf{B}_2)\rangle \mid \mathsf{B}_1, \mathsf{B}_2 \in \mathcal{B}\} \\ \\ \mathsf{pre\text{-}unions} : (\mathsf{X} \rightrightarrows_{\mathsf{pre}} \mathsf{Y}) \Rightarrow (\mathsf{X} \rightrightarrows_{\mathsf{pre}} \mathsf{Y}) \\ \mathsf{pre\text{-}unions\ h} := \\ \quad \mathsf{let}\ \ \mathcal{B} := \mathsf{domain\ h} \\ \quad\quad \mathsf{in}\ \ \{\langle \bigcup \mathcal{B}', \bigcup(\mathsf{image\ h}\ \mathcal{B}')\rangle \mid \mathcal{B}' \subseteq \mathcal{B} \wedge |\mathcal{B}'| \leq \omega\} \\ \\ \mathsf{pre\text{-}ops} : (\mathsf{X} \rightrightarrows_{\mathsf{pre}} \mathsf{Y}) \Rightarrow (\mathsf{X} \rightrightarrows_{\mathsf{pre}} \mathsf{Y}) \\ \mathsf{pre\text{-}ops\ h} := \{\langle \varnothing, \varnothing\rangle\} \cup (\mathsf{pre\text{-}comps\ h}) \cup (\mathsf{pre\text{-}unions\ h}) \end{array} \tag{40}$$

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

$$\begin{array}{l}\mathsf{pre\text{-}close} : (\mathsf{X} \rightrightarrows_{\mathsf{pre}} \mathsf{Y}) \Rightarrow (\mathsf{X} \rightrightarrows_{\mathsf{pre}} \mathsf{Y}) \\ \mathsf{pre\text{-}close} := \mathsf{close\ pre\text{-}ops}\ \omega_1 \end{array} \tag{41}$$

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. *Let* $\mathsf{f} \in \mathsf{X} \rightharpoonup \mathsf{Y}$ *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, noting that the preimage of $\varnothing$ under any function is $\varnothing$, and that preimages distribute over complements and unions. □

XXX: as with mappings, it helps to have basic composition and pairing functions for preimage mappings

6.2 Preimage Mapping Composition

XXX: moar wurds in this section

$$\begin{array}{l}(\circ_{\mathsf{pre}}) : (\mathsf{Y} \rightrightarrows_{\mathsf{pre}} \mathsf{Z}) \Rightarrow (\mathsf{X} \rightrightarrows_{\mathsf{pre}} \mathsf{Y}) \Rightarrow (\mathsf{X} \rightrightarrows_{\mathsf{pre}} \mathsf{Z}) \\ \mathsf{h}_2 \circ_{\mathsf{pre}} \mathsf{h}_1 := \lambda \mathsf{C} \in \mathsf{domain\ h}_2.\, \mathsf{pre\text{-}ap\ h}_1\ (\mathsf{h}_2\ \mathsf{C}) \end{array} \tag{42}$$

Lemma 4 (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 4 (Preimage mapping composition)**.** *Let* $\mathsf{g}_1 \in \mathsf{X} \rightharpoonup \mathsf{Y}$, $\mathcal{B}'$ *a* σ*-algebra on* Y, *and* $\mathcal{B} := \sigma\text{-}\mathsf{trace}\ \mathcal{B}'\ (\mathsf{range\ g}_1)$. *Let* $\mathsf{g}_2 \in \mathsf{Y} \rightharpoonup \mathsf{Z}$, $\mathcal{C}'$ *a* σ*-algebra on* Z, *and* $\mathcal{C} := \sigma\text{-}\mathsf{trace}\ \mathcal{C}'\ (\mathsf{range\ g}_2)$. *Then for all* $\mathsf{C} \in \mathcal{C}'$, $\mathsf{pre\text{-}ap}\ ((\mathsf{pre\ g}_2\ \mathcal{C}) \circ_{\mathsf{pre}} (\mathsf{pre\ g}_1\ \mathcal{B}))\ \mathsf{C} = \mathsf{preimage}\ (\mathsf{g}_2 \circ_{\mathsf{map}} \mathsf{g}_1)\ \mathsf{C}$.

Proof. Expand definitions and apply Theorem 2 and Lemma 4.

$$\begin{array}{ll} & \mathsf{pre\text{-}ap}\ ((\mathsf{pre\ g}_2\ \mathcal{C}) \circ_{\mathsf{pre}} (\mathsf{pre\ g}_1\ \mathcal{B}))\ \mathsf{C} \\ = & \mathsf{let}\ \ \mathsf{h} := \lambda \mathsf{C} \in \mathcal{C}.\, \mathsf{pre\text{-}ap}\ (\mathsf{pre\ g}_1\ \mathcal{B})\ ((\mathsf{pre\ g}_2\ \mathcal{C})\ \mathsf{C}) \\ & \quad \mathsf{in}\ \ \mathsf{h}\ (\mathsf{C} \cap \bigcup(\mathsf{domain\ h})) \\ = & \mathsf{let}\ \ \mathsf{h} := \lambda \mathsf{C} \in \mathcal{C}.\, \mathsf{preimage\ g}_1\ ((\mathsf{pre\ g}_2\ \mathcal{C})\ \mathsf{C}) \\ & \quad \mathsf{in}\ \ \mathsf{h}\ (\mathsf{C} \cap \bigcup \mathcal{C}) \\ = & \mathsf{preimage\ g}_1\ ((\mathsf{pre\ g}_2\ \mathcal{C})\ (\mathsf{C} \cap \bigcup \mathcal{C})) \\ = & \mathsf{preimage\ g}_1\ (\mathsf{pre\text{-}ap}\ (\mathsf{pre\ g}_2\ \mathcal{C})\ \mathsf{C}) \\ = & \mathsf{preimage\ g}_1\ (\mathsf{preimage\ g}_2\ \mathsf{C}) \\ = & \mathsf{preimage}\ (\mathsf{g}_2 \circ_{\mathsf{map}} \mathsf{g}_1)\ \mathsf{C} \end{array} \tag{43}$$

□

6.3 Preimage Mapping Pairing

XXX: moar wurds in this section

$$\begin{array}{l}\langle \cdot, \cdot \rangle_{\mathsf{pre}} : (\mathsf{X} \rightrightarrows_{\mathsf{pre}} \mathsf{Y}) \Rightarrow (\mathsf{X} \rightrightarrows_{\mathsf{pre}} \mathsf{Z}) \Rightarrow (\mathsf{X} \rightrightarrows_{\mathsf{pre}} \mathsf{Y} \times \mathsf{Z}) \\ \langle \mathsf{h}_1, \mathsf{h}_2 \rangle_{\mathsf{pre}} := \mathsf{let}\ \ \mathcal{B} := (\mathsf{domain\ h}_1) \boxtimes (\mathsf{domain\ h}_2) \\ \qquad\qquad\qquad\qquad \mathsf{h} := \lambda \mathsf{B}_1 \times \mathsf{B}_2 \in \mathcal{B}.\, (\mathsf{h}_1\ \mathsf{B}_1) \cap (\mathsf{h}_2\ \mathsf{B}_2) \\ \qquad\qquad\qquad\quad \mathsf{in}\ \ \mathsf{pre\text{-}close\ h} \end{array} \tag{44}$$

Lemma 5 (Preimages of rectangles under 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* $\mathsf{B}_2 \subseteq \mathsf{Z}$, $\mathsf{preimage}\ \langle \mathsf{g}_1, \mathsf{g}_2 \rangle_{\mathsf{map}}\ (\mathsf{B}_1 \times \mathsf{B}_2) = (\mathsf{preimage\ g}_1\ \mathsf{B}_1) \cap (\mathsf{preimage\ g}_2\ \mathsf{B}_2)$.

Theorem 5 (Preimage mapping pairing)**.** *Let* $\mathsf{g}_1 \in \mathsf{X} \rightharpoonup \mathsf{Y}$, $\mathcal{B}_1'$ *a* σ*-algebra on* Y, *and* $\mathcal{B}_1 := \sigma\text{-}\mathsf{trace}\ \mathcal{B}_1'\ (\mathsf{range\ g}_1)$. *Let* $\mathsf{g}_2 \in \mathsf{X} \rightharpoonup \mathsf{Z}$, $\mathcal{B}_2'$ *a* σ*-algebra on* Z, *and* $\mathcal{B}_2 := \sigma\text{-}\mathsf{trace}\ \mathcal{B}_2'\ (\mathsf{range\ g}_2)$. *Then for all* $\mathsf{B} \in \mathcal{B}_1' \otimes \mathcal{B}_2'$, $\mathsf{pre\text{-}ap}\ \langle \mathsf{pre\ g}_1\ \mathcal{B}_1, \mathsf{pre\ g}_2, \mathcal{B}_2 \rangle_{\mathsf{pre}}\ \mathsf{B} = \mathsf{preimage}\ \langle \mathsf{g}_1, \mathsf{g}_2 \rangle_{\mathsf{map}}\ \mathsf{B}$.

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

$$\begin{array}{ll} & \mathsf{pre\text{-}ap}\ \langle \mathsf{pre\ g}_1\ \mathcal{B}_1, \mathsf{pre\ g}_2\ \mathcal{B}_2 \rangle_{\mathsf{pre}}\ \mathsf{B} \\ = & \mathsf{let}\ \ \mathsf{h}_1 := \mathsf{pre\ g}_1\ \mathcal{B}_1 \\ & \qquad \mathsf{h}_2 := \mathsf{pre\ g}_2\ \mathcal{B}_2 \\ & \qquad \mathcal{B} := \mathsf{rects}\ (\mathsf{domain\ h}_1)\ (\mathsf{domain\ h}_2) \\ & \qquad \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) \\ & \quad \mathsf{in}\ \ \mathsf{pre\text{-}ap\ (pre\text{-}close\ h)\ B} \\ = & \mathsf{let}\ \ \mathcal{B} := \mathsf{rects}\ \mathcal{B}_1\ \mathcal{B}_2 \\ & \qquad \mathsf{h} := \lambda \mathsf{B}_1 \times \mathsf{B}_2 \in \mathcal{B}. \\ & \qquad\qquad (\mathsf{pre\ g}_1\ \mathcal{B}_1\ \mathsf{B}_1) \cap (\mathsf{pre\ g}_2\ \mathcal{B}_2\ \mathsf{B}_2) \\ & \quad \mathsf{in}\ \ \mathsf{pre\text{-}ap\ (pre\text{-}close\ h)\ B} \\ = & \mathsf{let}\ \ \mathcal{B} := \mathsf{rects}\ \mathcal{B}_1\ \mathcal{B}_2 \\ & \qquad \mathsf{h} := \lambda \mathsf{B}_1 \times \mathsf{B}_2 \in \mathcal{B}. \\ & \qquad\qquad (\mathsf{preimage\ g}_1\ \mathsf{B}_1) \cap (\mathsf{preimage\ g}_2\ \mathsf{B}_2) \\ & \quad \mathsf{in}\ \ \mathsf{pre\text{-}ap\ (pre\text{-}close\ h)\ B} \\ = & \mathsf{let}\ \ \mathcal{B} := \mathsf{rects}\ \mathcal{B}_1\ \mathcal{B}_2 \\ & \qquad \mathsf{h} := \lambda \mathsf{B} \in \mathcal{B}.\, \mathsf{preimage}\ \langle \mathsf{g}_1, \mathsf{g}_2 \rangle_{\mathsf{map}}\ \mathsf{B} \\ & \quad \mathsf{in}\ \ \mathsf{pre\text{-}ap\ (pre\text{-}close\ h)\ B} \\ = & \mathsf{let}\ \ \mathcal{B} := \mathsf{rects}\ \mathcal{B}_1\ \mathcal{B}_2 \\ & \quad \mathsf{in}\ \ \mathsf{pre\text{-}ap\ (pre\text{-}close\ (pre}\ \langle \mathsf{g}_1, \mathsf{g}_2 \rangle_{\mathsf{map}}\ \mathcal{B}))\ \mathsf{B} \\ = & \mathsf{let}\ \ \mathcal{B} := \mathsf{rects}\ \mathcal{B}_1\ \mathcal{B}_2 \\ & \quad \mathsf{in}\ \ \mathsf{pre\text{-}ap\ (pre}\ \langle \mathsf{g}_1, \mathsf{g}_2 \rangle_{\mathsf{map}}\ (\sigma\text{-}\mathsf{close}\ \mathcal{B}))\ \mathsf{B} \\ = & \mathsf{pre\text{-}ap\ (pre}\ \langle \mathsf{g}_1, \mathsf{g}_2 \rangle_{\mathsf{map}}\ (\mathcal{B}_1 \otimes \mathcal{B}_2))\ \mathsf{B} \\ = & \mathsf{preimage}\ \langle \mathsf{g}_1, \mathsf{g}_2 \rangle_{\mathsf{map}}\ \mathsf{B} \end{array} \tag{45}$$

□

7. The Preimage Arrow

XXX: intro

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

$$\begin{array}{l}\sigma : \mathsf{Set}\ \mathbb{U} \Rightarrow \mathsf{Set}\ (\mathsf{Set}\ \mathbb{U}) \\ \sigma\ \mathsf{A} := \sigma\text{-}\mathsf{trace}\ \mathcal{U}\ \mathsf{A} \end{array} \tag{46}$$

A preimage arrow computation is a function from a domain to a mapping from measurable output sets to measur-

able input sets (where the input sets are restricted to the given domain)

$$\mathsf{X} \overset{\rightsquigarrow}{\underset{\mathsf{pre}}{}} \mathsf{Y} \;::=\; \mathsf{Set}\ \mathsf{X} \Rightarrow (\mathsf{X} \overset{\rightharpoonup}{\underset{\mathsf{pre}}{}} \mathsf{Y}) \tag{47}$$

where X and Y are subsets of $\mathbb{U}$

$$\begin{array}{l} \mathsf{arr}_{\mathsf{pre}} : (\mathsf{X} \overset{\rightsquigarrow}{\underset{\mathsf{map}}{}} \mathsf{Y}) \Rightarrow (\mathsf{X} \overset{\rightsquigarrow}{\underset{\mathsf{pre}}{}} \mathsf{Y}) \\ \mathsf{arr}_{\mathsf{pre}}\ \mathsf{g}\ \mathsf{A} \;:=\; \mathsf{let}\ \ \mathsf{g}' := \mathsf{g}\ \mathsf{A} \\ \qquad\qquad\qquad\quad \mathcal{B} := \sigma\ (\mathsf{range}\ \mathsf{g}') \\ \qquad\qquad\quad \mathsf{in}\ \ \mathsf{pre}\ \mathsf{g}'\ \mathcal{B} \end{array} \tag{48}$$

Definition 2 (Preimage arrow equivalence)**.** *Two preimage arrow computations* $\mathsf{h}_1 : \mathsf{X} \overset{\rightsquigarrow}{\underset{\mathsf{pre}}{}} \mathsf{Y}$ *and* $\mathsf{h}_2 : \mathsf{X} \overset{\rightsquigarrow}{\underset{\mathsf{pre}}{}} \mathsf{Y}$ *are equivalent, written* $\mathsf{h}_1 \overset{\equiv}{\underset{\mathsf{pre}}{}} \mathsf{h}_2$*, when for any* σ*-algebra* $\mathcal{B}$ *for* Y*,* $\mathsf{pre\text{-}ap}\ (\mathsf{h}_1\ \mathsf{A})\ \mathsf{B} = \mathsf{pre\text{-}ap}\ (\mathsf{h}_2\ \mathsf{A})\ \mathsf{B}$ *for all* $\mathsf{A} \subseteq \mathsf{X}$ *and* $\mathsf{B} \in \mathcal{B}$*.*

7.1 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,

$$\mathsf{arr}_{\mathsf{pre}}\ (\mathsf{pair}_{\mathsf{map}}\ \mathsf{g}_1\ \mathsf{g}_2) \overset{\equiv}{\underset{\mathsf{pre}}{}} \mathsf{pair}_{\mathsf{pre}}\ (\mathsf{arr}_{\mathsf{pre}}\ \mathsf{g}_1)\ (\mathsf{arr}_{\mathsf{pre}}\ \mathsf{g}_2) \tag{49}$$

$$\mathsf{arr}_{\mathsf{pre}}\ (\ggg_{\mathsf{map}}\ \mathsf{g}_1\ \mathsf{g}_2) \overset{\equiv}{\underset{\mathsf{pre}}{}} \ggg_{\mathsf{pre}}\ (\mathsf{arr}_{\mathsf{pre}}\ \mathsf{g}_1)\ (\mathsf{arr}_{\mathsf{pre}}\ \mathsf{g}_2) \tag{50}$$

$$\begin{array}{l} \mathsf{arr}_{\mathsf{pre}}\ (\mathsf{if}_{\mathsf{map}}\ \mathsf{g}_1\ \mathsf{g}_2\ \mathsf{g}_3) \overset{\equiv}{\underset{\mathsf{pre}}{}} \\ \quad \mathsf{if}_{\mathsf{pre}}\ (\mathsf{arr}_{\mathsf{pre}}\ \mathsf{g}_1)\ (\lambda 0.\ \mathsf{arr}_{\mathsf{pre}}\ (\mathsf{g}_2\ 0))\ (\lambda 0.\ \mathsf{arr}_{\mathsf{pre}}\ (\mathsf{g}_3\ 0)) \end{array} \tag{51}$$

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

Proof. By structural induction; cases follow. □

7.2 Case: Pairing

Starting with the left-hand side of (49), we expand definitions and apply Theorem 2:

$$\begin{array}{l} \mathsf{pre\text{-}ap}\ (\mathsf{arr}_{\mathsf{pre}}\ (\mathsf{pair}_{\mathsf{map}}\ \mathsf{g}_1\ \mathsf{g}_2)\ \mathsf{A})\ \mathsf{B} \\ \quad = \ \mathsf{let}\ \ \mathsf{g}' := \langle \mathsf{g}_1\ \mathsf{A}, \mathsf{g}_2\ \mathsf{A} \rangle_{\mathsf{map}} \\ \qquad\quad \mathcal{B} := \sigma\ (\mathsf{range}\ \mathsf{g}') \\ \qquad\quad \mathsf{h} := \mathsf{pre}\ \mathsf{g}'\ \mathcal{B} \\ \qquad \mathsf{in}\ \ \mathsf{pre\text{-}ap}\ \mathsf{h}\ \mathsf{B} \\ \quad = \ \mathsf{preimage}\ \langle \mathsf{g}_1\ \mathsf{A}, \mathsf{g}_2\ \mathsf{A} \rangle_{\mathsf{map}}\ \mathsf{B} \end{array} \tag{52}$$

Next, we apply Theorem 5 and replace g_1 and g_2 with $\mathsf{arr}_{\mathsf{pre}}\ \mathsf{g}_1$ and $\mathsf{arr}_{\mathsf{pre}}\ \mathsf{g}_2$:

$$\begin{array}{l} \mathsf{pre\text{-}ap}\ (\mathsf{arr}_{\mathsf{pre}}\ (\mathsf{pair}_{\mathsf{map}}\ \mathsf{g}_1\ \mathsf{g}_2)\ \mathsf{A})\ \mathsf{B} \\ \quad = \ \mathsf{let}\ \ \mathsf{g}_1' := \mathsf{g}_1\ \mathsf{A} \\ \qquad\quad \mathsf{g}_2' := \mathsf{g}_2\ \mathsf{A} \\ \qquad\quad \mathcal{B}_1 := \sigma\ (\mathsf{range}\ \mathsf{g}_1') \\ \qquad\quad \mathcal{B}_2 := \sigma\ (\mathsf{range}\ \mathsf{g}_2') \\ \qquad \mathsf{in}\ \ \mathsf{pre\text{-}ap}\ \langle \mathsf{pre}\ \mathsf{g}_1'\ \mathcal{B}_1, \mathsf{pre}\ \mathsf{g}_2'\ \mathcal{B}_2 \rangle_{\mathsf{pre}}\ \mathsf{B} \\ \quad = \ \mathsf{let}\ \ \mathsf{h}_1 := \mathsf{arr}_{\mathsf{pre}}\ \mathsf{g}_1\ \mathsf{A} \\ \qquad\quad \mathsf{h}_2 := \mathsf{arr}_{\mathsf{pre}}\ \mathsf{g}_2\ \mathsf{A} \\ \qquad \mathsf{in}\ \ \mathsf{pre\text{-}ap}\ \langle \mathsf{h}_1, \mathsf{h}_2 \rangle_{\mathsf{pre}}\ \mathsf{B} \end{array} \tag{53}$$

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 $\mathsf{pair}_{\mathsf{pre}}$:

$$\begin{array}{l} \mathsf{pair}_{\mathsf{pre}} : (\mathsf{X} \overset{\rightsquigarrow}{\underset{\mathsf{pre}}{}} \mathsf{Y}) \Rightarrow (\mathsf{X} \overset{\rightsquigarrow}{\underset{\mathsf{pre}}{}} \mathsf{Z}) \Rightarrow (\mathsf{X} \overset{\rightsquigarrow}{\underset{\mathsf{pre}}{}} \mathsf{Y} \times \mathsf{Z}) \\ \mathsf{pair}_{\mathsf{pre}}\ \mathsf{h}_1\ \mathsf{h}_2\ \mathsf{A} \;:=\; \langle \mathsf{h}_1\ \mathsf{A}, \mathsf{h}_2\ \mathsf{A} \rangle_{\mathsf{pre}} \end{array} \tag{54}$$

7.3 Case: Composition

XXX: Starting with the left-hand side of (51), we expand definitions...

$$\begin{array}{l} \mathsf{pre\text{-}ap}\ (\mathsf{arr}_{\mathsf{pre}}\ (\ggg_{\mathsf{map}}\ \mathsf{g}_1\ \mathsf{g}_2)\ \mathsf{A})\ \mathsf{C} \\ \quad = \ \mathsf{let}\ \ \mathsf{g}_1' := \mathsf{g}_1\ \mathsf{A} \\ \qquad\quad \mathsf{g}_2' := \mathsf{g}_2\ (\mathsf{range}\ \mathsf{g}_1') \\ \qquad\quad \mathsf{A}' := \mathsf{preimage}\ \mathsf{g}_1'\ (\mathsf{domain}\ \mathsf{g}_2') \\ \qquad\quad \mathsf{g}' := \lambda \mathsf{x} \in \mathsf{A}'.\, \mathsf{g}_2'\ (\mathsf{g}_1'\ \mathsf{x}) \\ \qquad\quad \mathcal{C} := \sigma\ (\mathsf{range}\ \mathsf{g}') \\ \qquad \mathsf{in}\ \ \mathsf{pre\text{-}ap}\ (\mathsf{pre}\ \mathsf{g}'\ \mathcal{C})\ \mathsf{C} \end{array} \tag{55}$$

7.4 Case: Conditional

XXX: do this

7.5 Super-Saver Theorems

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

Corollary 3. *Let* $\mathsf{g} : \mathsf{X} \overset{\rightsquigarrow}{\underset{\mathsf{map}}{}} \mathsf{Y}$*,* $\mathsf{A} \subseteq \mathsf{X}$*, and* $\mathsf{h} : \mathsf{X} \overset{\rightsquigarrow}{\underset{\mathsf{pre}}{}} \mathsf{Y}$ *such that* $\mathsf{h} \overset{\equiv}{\underset{\mathsf{map}}{}} \mathsf{arr}_{\mathsf{pre}}\ \mathsf{g}$*. Then* $\mathsf{h}\ \mathsf{A}$ *diverges if and only if* $\mathsf{g}\ \mathsf{A}$ *diverges.*

Corollary 4. $\mathsf{arr}_{\mathsf{pre}}$*,* $\mathsf{pair}_{\mathsf{pre}}$ *and* $\ggg_{\mathsf{pre}}$ *define an arrow.*

8. Preimages of Partial Functions

$$\begin{array}{l} \bot_{\mathsf{pre}} : \mathsf{X} \overset{\rightsquigarrow}{\underset{\mathsf{pre}}{}} \varnothing \\ \bot_{\mathsf{pre}} \;:=\; \mathsf{arr}_{\mathsf{pre}}\ (\mathsf{arr}_{\mathsf{map}}\ \lambda \mathsf{x}.\, \bot) \end{array} \tag{56}$$

References

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

Figure 3: Preimage arrow definitions.