Running Probabilistic Programs Backward

Neil Toronto Jay McCarthy

PLT @ Brigham Young University

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

Abstract

XXX

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

General Terms XXX, XXX

Keywords XXX, XXX

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

1. Introduction

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

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

2. Mathematics and Metalanguage

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

We write all of the mathematics in this paper in λ_{ZFC} [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 unique **Cardinality**). ZFC has only first-order functions and no general recursion, which makes implementing a language defined by a transformation into contemporary mathematics quite difficult. The problem is exacerbated if implementing the language requires approximation. Targeting λ_{ZFC} instead allows creating a precise mathematical specification and deriving an approximating implementation without changing languages.

In λ_{ZFC}, essentially every set is a value, as well as every lambda and every set of lambdas. All operations, including operations on infinite sets, are assumed to complete instantly if they terminate.[1]

Almost everything definable in contemporary mathematics can be formally defined by a finite λ_{ZFC} program, except objects that most mathematicians would agree are nonconstructive. More precisely, any object that *must* be defined by a statement of existence and uniqueness without giving a bounding set is not definable by a *finite* λ_{ZFC} program.

Because λ_{ZFC} includes an inner model of ZFC, essentially every contemporary theorem applies to λ_{ZFC}'s set values without alteration. Further, proofs about λ_{ZFC}'s set values apply to contemporary mathematical objects.[2]

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

We assume data structures, including pairs, are encoded as *primitive* ordered pairs with the first element a unique tag, so that they can be distinguished by checking tags. Accessors such as fst and snd are trivial to define.

λ_{ZFC} is untyped so its users can define an auxiliary type system that best suits their application area. For this work, we use an informal, manually checked, polymorphic type system characterized by these rules:

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

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

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

[2] Assuming the existence of an inaccessible cardinal.

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

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

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

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

2.1 Internal and External Equality

Set theory extends first-order logic with an axiom that defines equality to be extensional, and with axioms that ensure the existence of sets in the domain of discourse. λ_{ZFC} is defined the same way as any other operational λ-calculus: by (conservatively) extending the domain of discourse with expressions and defining a reduction relation.

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

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

which means

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

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

We understand any λ_{ZFC} term e used as a truth statement as shorthand for "e reduces to true." Therefore, while the terms $\{(\lambda \mathsf{x}.\, \mathsf{x})\ 1, 1\}$ and $\{1\}$ are (externally, extensionally) unequal, we can say that $\{(\lambda \mathsf{x}.\, \mathsf{x})\ 1, 1\} = \{1\}$.

Any truth statement e implies that e converges. We sometimes do not want this, particularly when we want to say that e_1 and e_2 are equivalent when they both diverge. In these cases, 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 equivalence, such as applicative or logical bisimilarity. However, we do not want internal equality and external equivalence to differ too much. We therefore introduce type-specific notions of equivalence as needed.

2.2 Additional Functions and Forms

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

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

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

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

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

XXX: logic: logical operators and quantifiers

In set theory, functions are encoded as sets of input-output pairs. The increment function for the natural numbers, for example, is $\{\langle 0, 1\rangle, \langle 1, 2\rangle, \langle 2, 3\rangle, ...\}$. To distinguish these hash tables from lambdas, we call them ***mappings***, and use the word **function** for either a lambda or a mapping. For convenience, as with lambdas, we use adjacency (i.e. $(\mathsf{f\ x})$) to apply mappings.

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

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

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

Figure 1 defines more operations on partial mappings: domain, range, $\mathsf{preimage}$, pairing, composition, and disjoint union. The latter three are particularly important in the preimage arrow's derivation, and $\mathsf{preimage}$ is critical in measure theory's account of probability.

XXX: lazy mappings

3. 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: Figure 2 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)

In a nonstrict or simply typed λ-calculus, $\mathsf{if}_{\bot}$ can be defined using the other combinators and a function $\mathsf{choose} : \langle \mathsf{Bool}, \langle \mathsf{X}, \mathsf{X}\rangle\rangle \Rightarrow \mathsf{X}$, whose boolean input determines which of the $\langle \mathsf{X}, \mathsf{X}\rangle$ it returns. However, λ_{ZFC} is call-by-value, so we need an explicitly lazy conditional. We would have had to define $\mathsf{if}_{\bot}$ in Section XXX (implementation) anyway, because the preimage arrow's lift returns unimplementable functions.

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

XXX: Figure 3...

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

XXX: example: suppose $\mathsf{x} \div \mathsf{y}$ diverges when $\mathsf{y} = 0$...

$$\mathsf{div}_{\bot} := [\![\mathsf{if}\ (\mathsf{snd}\ (\mathsf{env}\ 0) = 0)\ \bot\ (\mathsf{fst}\ (\mathsf{env}\ 0) \div \mathsf{snd}\ (\mathsf{env}\ 0))]\!] \tag{6}$$

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

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

Figure 1: Operations on mappings.

$$\begin{aligned}
&\mathsf{arr}_\perp : (\mathsf{X} \Rightarrow \mathsf{Y}) \Rightarrow (\mathsf{X} \Rightarrow \mathsf{Y}_\perp)\\
&\mathsf{arr}_\perp\ \mathsf{f}\ :=\ \mathsf{f}\\[1ex]
&(\ggg_\perp) : (\mathsf{X} \Rightarrow \mathsf{Y}_\perp) \Rightarrow (\mathsf{Y} \Rightarrow \mathsf{Z}_\perp) \Rightarrow (\mathsf{X} \Rightarrow \mathsf{Z}_\perp)\\
&(\mathsf{f_1} \ggg_\perp \mathsf{f_2})\ \mathsf{x}\ :=\ \mathsf{if}\ (\mathsf{f_1}\ \mathsf{x} = \perp)\ \perp\ (\mathsf{f_2}\ (\mathsf{f_1}\ \mathsf{x}))\\[1ex]
&(\&\&\&_\perp) : (\mathsf{X} \Rightarrow \mathsf{Y_1}_\perp) \Rightarrow (\mathsf{X} \Rightarrow \mathsf{Y_2}_\perp) \Rightarrow (\mathsf{X} \Rightarrow \langle\mathsf{Y_1},\mathsf{Y_2}\rangle_\perp)\\
&(\mathsf{f_1}\ \&\&\&_\perp\ \mathsf{f_2})\ \mathsf{x}\ :=\ \mathsf{if}\ ((\mathsf{f_1}\ \mathsf{x} = \perp) \vee (\mathsf{f_2}\ \mathsf{x} = \perp))\ \perp\ \langle\mathsf{f_1}\ \mathsf{x}, \mathsf{f_2}\ \mathsf{x}\rangle
\end{aligned}$$

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

Figure 2: Bottom arrow definitions.

$$\begin{aligned}
[\![x := e;\ \cdots]\!]_a &:\equiv x := [\![e]\!]_a\,;\ \cdots\\
[\![x\ e]\!]_a &:\equiv [\![\langle e, 0\rangle]\!]_a \ggg_a x\\
[\![\langle e_1, e_2\rangle]\!]_a &:\equiv [\![e_1]\!]_a\ \&\&\&_a\ [\![e_2]\!]_a\\
[\![\mathsf{if}\ e_c\ e_t\ e_f]\!]_a &:\equiv \mathsf{if}_a\ [\![e_c]\!]_a\ (\lambda 0.\,[\![e_t]\!]_a)\ (\lambda 0.\,[\![e_f]\!]_a)\\
[\![\mathsf{let}\ e_0\ e_b]\!]_a &:\equiv ([\![e_0]\!]_a\ \&\&\&_a\ (\mathsf{arr}_a\ \mathsf{id})) \ggg_a [\![e_b]\!]_a\\
[\![\mathsf{env}\ n]\!]_a &:\equiv \mathsf{arr}_a\ \lambda\gamma.\,\gamma_n
\end{aligned}
\qquad
\begin{aligned}
[\![v]\!]_a &:\equiv \mathsf{arr}_a\ \lambda\gamma.\,v\\
[\![\mathsf{fst}\ e]\!]_a &:\equiv [\![e]\!]_a \ggg_a (\mathsf{arr}_a\ \mathsf{fst})\\
[\![\mathsf{snd}\ e]\!]_a &:\equiv [\![e]\!]_a \ggg_a (\mathsf{arr}_a\ \mathsf{snd})\\
[\![e_1 = e_2]\!]_a &:\equiv [\![\langle e_1, e_2\rangle]\!]_a \ggg_a (\mathsf{arr}_a\ \lambda\langle\mathsf{x},\mathsf{y}\rangle.\,\mathsf{x} = \mathsf{y})\\
[\![e_1 + e_2]\!]_a &:\equiv [\![\langle e_1, e_2\rangle]\!]_a \ggg_a (\mathsf{arr}_a\ \lambda\langle\mathsf{x},\mathsf{y}\rangle.\,\mathsf{x} + \mathsf{y})\\
&\cdots
\end{aligned}$$

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

4. 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

$$\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}\ ::=\ \mathsf{Set}\ \mathsf{X} \Rightarrow (\mathsf{X} \rightharpoonup \mathsf{Y}) \tag{7}$$

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

XXX: motivate removal of bottom (reasons: won't need to propagate it; its absence will be convenient when computing preimages under functions that may diverge)

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 $\mathsf{lift}_{\mathsf{map}}$ in terms of $\mathsf{domain}_\perp$:

$$\begin{aligned}
&\mathsf{domain}_\perp : (\mathsf{X} \Rightarrow \mathsf{Y}_\perp) \Rightarrow \mathsf{Set}\ \mathsf{X} \Rightarrow \mathsf{Set}\ \mathsf{X}\\
&\mathsf{domain}_\perp\ \mathsf{f}\ \mathsf{A}\ :=\ \mathsf{preimage}\ \mathsf{f}|_{\mathsf{A}}\ ((\mathsf{image}\ \mathsf{f}\ \mathsf{A}) \backslash \{\perp\})
\end{aligned} \tag{8}$$

$$\begin{aligned}
&\mathsf{lift}_{\mathsf{map}} : (\mathsf{X} \Rightarrow \mathsf{Y}_\perp) \Rightarrow (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y})\\
&\mathsf{lift}_{\mathsf{map}}\ \mathsf{f}\ \mathsf{A}\ :=\ \mathsf{let}\ \ \mathsf{A}' := \mathsf{domain}_\perp\ \mathsf{f}\ \mathsf{A}\\
&\qquad\qquad\qquad\ \ \mathsf{in}\ \ \mathsf{f}|_{\mathsf{A}'}
\end{aligned} \tag{9}$$

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

Definition 2 (Mapping arrow equivalence)**.** *Two mapping arrow computations* $\mathsf{g_1} : \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}$ *and* $\mathsf{g_2} : \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}$ *are equivalent, or* $\mathsf{g_1} \equiv \mathsf{g_2}$*, when* $\mathsf{g_1}\ \mathsf{A} \equiv \mathsf{g_2}\ \mathsf{A}$ *for all* $\mathsf{A} \subseteq \mathsf{X}$*.*

4.1 Distributive Laws

The clearest way to ensure that mapping arrow computations mean what we think they mean is to derive each combinator in a way that makes $\mathsf{lift_{map}}$ distribute over bottom arrow computations. Formally, we require the following distributive laws to hold:

$$\mathsf{lift_{map}}\ (\mathsf{arr_\bot}\ \mathsf{f})\ \equiv\ \mathsf{arr_{map}}\ \mathsf{f} \tag{10}$$

$$\mathsf{lift_{map}}\ (\mathsf{f_1}\ \&\&\&_\bot\ \mathsf{f_2})\ \equiv\ (\mathsf{lift_{map}}\ \mathsf{f_1})\ \&\&\&_{\mathsf{map}}\ (\mathsf{lift_{map}}\ \mathsf{f_2}) \tag{11}$$

$$\mathsf{lift_{map}}\ (\mathsf{f_1}\ \ggg_\bot\ \mathsf{f_2})\ \equiv\ (\mathsf{lift_{map}}\ \mathsf{f_1})\ \ggg_{\mathsf{map}}\ (\mathsf{lift_{map}}\ \mathsf{f_2}) \tag{12}$$

$$\begin{aligned}&\mathsf{lift_{map}}\ (\mathsf{if_\bot}\ \mathsf{f_1}\ \mathsf{f_2}\ \mathsf{f_3})\ \equiv\\ &\quad \mathsf{if_{map}}\ (\mathsf{lift_{map}}\ \mathsf{f_1})\ (\lambda 0.\,\mathsf{lift_{map}}\ (\mathsf{f_2}\ 0))\ (\lambda 0.\,\mathsf{lift_{map}}\ (\mathsf{f_3}\ 0))\end{aligned} \tag{13}$$

Clearly $\mathsf{arr_{map}}\ \mathsf{f} := \mathsf{lift_{map}}\ (\mathsf{arr_\bot}\ \mathsf{f})$ meets (10). Figure 4 shows the result of deriving the other combinators from the bottom arrow using distributive laws.

Theorem 1 (mapping arrow correctness)**.** $\mathsf{lift_{map}}$ *distributes over bottom arrow computations.*

Proof. By structural induction; cases follow. ☐

4.2 Case: Pairing

Starting with the left-hand side of (11), we first expand definitions. For any $\mathsf{f_1} : \mathsf{X} \Rightarrow \mathsf{Y}_\bot$, $\mathsf{f_2} : \mathsf{X} \Rightarrow \mathsf{Z}_\bot$, and $\mathsf{A} \subseteq \mathsf{X}$,

$$\begin{aligned}&\mathsf{lift_{map}}\ (\mathsf{f_1}\ \&\&\&_\bot\ \mathsf{f_2})\ \mathsf{A}\\ &\quad\equiv\ \mathsf{let}\ \ \mathsf{f} := \lambda \mathsf{x}.\,\mathsf{if}\ (\mathsf{f_1}\ \mathsf{x} = \bot \vee \mathsf{f_2}\ \mathsf{x} = \bot)\ \bot\ \langle \mathsf{f_1}\ \mathsf{x}, \mathsf{f_2}\ \mathsf{x}\rangle\\ &\qquad\qquad \mathsf{A}' := \mathsf{domain_\bot}\ \mathsf{f}\ \mathsf{A}\\ &\qquad\quad \mathsf{in}\ \ \mathsf{f}|_{\mathsf{A}'}\end{aligned} \tag{14}$$

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

$$\begin{aligned}&\mathsf{lift_{map}}\ (\mathsf{f_1}\ \&\&\&_\bot\ \mathsf{f_2})\ \mathsf{A}\\ &\quad\equiv\ \mathsf{let}\ \ \mathsf{A_1} := (\mathsf{domain_\bot}\ \mathsf{f_1}\ \mathsf{A})\\ &\qquad\qquad \mathsf{A_2} := (\mathsf{domain_\bot}\ \mathsf{f_2}\ \mathsf{A})\\ &\qquad\qquad \mathsf{A}' := \mathsf{A_1} \cap \mathsf{A_2}\\ &\qquad\quad \mathsf{in}\ \ \lambda \mathsf{x} \in \mathsf{A}'.\,\langle \mathsf{f_1}\ \mathsf{x}, \mathsf{f_2}\ \mathsf{x}\rangle\\ &\quad\equiv\ \mathsf{let}\ \ \mathsf{g_1} := \mathsf{lift_{map}}\ \mathsf{f_1}\ \mathsf{A}\\ &\qquad\qquad \mathsf{g_2} := \mathsf{lift_{map}}\ \mathsf{f_2}\ \mathsf{A}\\ &\qquad\qquad \mathsf{A}' := (\mathsf{domain}\ \mathsf{g_1}) \cap (\mathsf{domain}\ \mathsf{g_2})\\ &\qquad\quad \mathsf{in}\ \ \lambda \mathsf{x} \in \mathsf{A}'.\,\langle \mathsf{g_1}\ \mathsf{x}, \mathsf{g_2}\ \mathsf{x}\rangle\\ &\quad\equiv\ \langle \mathsf{lift_{map}}\ \mathsf{f_1}\ \mathsf{A}, \mathsf{lift_{map}}\ \mathsf{f_2}\ \mathsf{A}\rangle_{\mathsf{map}}\end{aligned} \tag{15}$$

Substituting $\mathsf{g_1}$ for $\mathsf{lift_{map}}\ \mathsf{f_1}$ and $\mathsf{g_2}$ for $\mathsf{lift_{map}}\ \mathsf{f_2}$ gives a definition for $(\&\&\&_{\mathsf{map}})$ (Figure 4) for which (11) holds.

4.3 Case: Composition

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

XXX: include it?

4.4 Case: Conditional

The derivation of $\mathsf{if_{map}}$ 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:

$$\mathsf{halts\text{-}on\text{-}true_\bot}\ :=\ \mathsf{if_\bot}\ \mathsf{id}\ (\lambda 0.\,\mathsf{id})\ (\lambda 0.\,\mathsf{halts\text{-}on\text{-}true_\bot}) \tag{16}$$

Its corresponding mapping arrow computation should diverge only if applied to a set containing false.

Starting with the left-hand-side of (13), we expand definitions, and simplify f by restricting it to a domain for which $\mathsf{f_1}\ \mathsf{x}$ cannot be $\bot$:

$$\begin{aligned}&\mathsf{lift_{map}}\ (\mathsf{if_\bot}\ \mathsf{f_1}\ \mathsf{f_2}\ \mathsf{f_3})\ \mathsf{A}\\ &\quad\equiv\ \mathsf{let}\ \ \mathsf{f} := \lambda \mathsf{x}.\,\mathsf{case}\ \ \mathsf{f_1}\ \mathsf{x}\\ &\qquad\qquad\qquad\qquad \mathsf{true} \implies \mathsf{f_2}\ 0\ \mathsf{x}\\ &\qquad\qquad\qquad\qquad \mathsf{false} \implies \mathsf{f_3}\ 0\ \mathsf{x}\\ &\qquad\qquad\qquad\qquad \mathsf{else} \implies \bot\\ &\qquad\qquad \mathsf{A}' := \mathsf{domain_\bot}\ \mathsf{f}\ \mathsf{A}\\ &\qquad\quad \mathsf{in}\ \ \mathsf{f}|_{\mathsf{A}'}\\ &\quad\equiv\ \mathsf{let}\ \ \mathsf{A_2} := \mathsf{preimage}\ \mathsf{f_1}|_{\mathsf{A}}\ \{\mathsf{true}\}\\ &\qquad\qquad \mathsf{A_3} := \mathsf{preimage}\ \mathsf{f_1}|_{\mathsf{A}}\ \{\mathsf{false}\}\\ &\qquad\qquad \mathsf{f} := \lambda \mathsf{x}.\,\mathsf{if}\ (\mathsf{f_1}\ \mathsf{x})\ (\mathsf{f_2}\ 0\ \mathsf{x})\ (\mathsf{f_3}\ 0\ \mathsf{x})\\ &\qquad\qquad \mathsf{A}' := \mathsf{domain_\bot}\ \mathsf{f}\ (\mathsf{A_2} \cup \mathsf{A_3})\\ &\qquad\quad \mathsf{in}\ \ \mathsf{f}|_{\mathsf{A}'}\end{aligned} \tag{17}$$

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

$$\begin{aligned}&\mathsf{lift_{map}}\ (\mathsf{if_\bot}\ \mathsf{f_1}\ \mathsf{f_2}\ \mathsf{f_3})\ \mathsf{A}\\ &\quad\equiv\ \mathsf{let}\ \ \mathsf{g_1} := \mathsf{lift_{map}}\ \mathsf{f_1}\ \mathsf{A}\\ &\qquad\qquad \mathsf{A_2} := \mathsf{preimage}\ \mathsf{g_1}\ \{\mathsf{true}\}\\ &\qquad\qquad \mathsf{A_3} := \mathsf{preimage}\ \mathsf{g_1}\ \{\mathsf{false}\}\\ &\qquad\qquad \mathsf{g_2} := \mathsf{lift_{map}}\ (\mathsf{f_2}\ 0)\ \mathsf{A_2}\\ &\qquad\qquad \mathsf{g_3} := \mathsf{lift_{map}}\ (\mathsf{f_3}\ 0)\ \mathsf{A_3}\\ &\qquad\qquad \mathsf{A}' := (\mathsf{domain}\ \mathsf{g_2}) \cup (\mathsf{domain}\ \mathsf{g_3})\\ &\qquad\quad \mathsf{in}\ \ \lambda \mathsf{x} \in \mathsf{A}'.\,\mathsf{if}\ (\mathsf{g_1}\ \mathsf{x})\ (\mathsf{g_2}\ \mathsf{x})\ (\mathsf{g_3}\ \mathsf{x})\end{aligned} \tag{18}$$

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

Note that the "true" branch needs to be taken only if $\mathsf{A_2}$ is nonempty; similarly for the "false" branch and $\mathsf{A_3}$. 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 $\mathsf{lift_{map}}\ (\mathsf{f_2}\ 0)$ and $\mathsf{lift_{map}}\ (\mathsf{f_3}\ 0)$ only to nonempty sets, using

$$\begin{aligned}&\mathsf{lazy_{map}} : (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y})\\ &\mathsf{lazy_{map}}\ \mathsf{f}\ \mathsf{A}\ :=\ \mathsf{if}\ (\mathsf{A} = \varnothing)\ \varnothing\ (\mathsf{f}\ \mathsf{A})\end{aligned} \tag{19}$$

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

$$\begin{aligned}&\mathsf{lift_{map}}\ (\mathsf{if_\bot}\ \mathsf{f_1}\ \mathsf{f_2}\ \mathsf{f_3})\ \mathsf{A}\\ &\quad\equiv\ \mathsf{let}\ \ \mathsf{g_1} := \mathsf{lift_{map}}\ \mathsf{f_1}\ \mathsf{A}\\ &\qquad\qquad \mathsf{g_2} := \mathsf{lazy_{map}}\ (\mathsf{lift_{map}}\ (\mathsf{f_2}\ 0))\ (\mathsf{preimage}\ \mathsf{g_1}\ \{\mathsf{true}\})\\ &\qquad\qquad \mathsf{g_3} := \mathsf{lazy_{map}}\ (\mathsf{lift_{map}}\ (\mathsf{f_3}\ 0))\ (\mathsf{preimage}\ \mathsf{g_1}\ \{\mathsf{false}\})\\ &\qquad\qquad \mathsf{A}' := (\mathsf{domain}\ \mathsf{g_2}) \cup (\mathsf{domain}\ \mathsf{g_3})\\ &\qquad\quad \mathsf{in}\ \ \lambda \mathsf{x} \in \mathsf{A}'.\,\mathsf{if}\ (\mathsf{g_1}\ \mathsf{x})\ (\mathsf{g_2}\ \mathsf{x})\ (\mathsf{g_3}\ \mathsf{x})\\ &\quad\equiv\ \mathsf{let}\ \ \mathsf{g_1} := \mathsf{lift_{map}}\ \mathsf{f_1}\ \mathsf{A}\\ &\qquad\qquad \mathsf{g_2} := \mathsf{lazy_{map}}\ (\mathsf{lift_{map}}\ (\mathsf{f_2}\ 0))\ (\mathsf{preimage}\ \mathsf{g_1}\ \{\mathsf{true}\})\\ &\qquad\qquad \mathsf{g_3} := \mathsf{lazy_{map}}\ (\mathsf{lift_{map}}\ (\mathsf{f_3}\ 0))\ (\mathsf{preimage}\ \mathsf{g_1}\ \{\mathsf{false}\})\\ &\qquad\quad \mathsf{in}\ \ \mathsf{g_2} \uplus_{\mathsf{map}} \mathsf{g_3}\end{aligned} \tag{20}$$

For $\mathsf{halts\text{-}on\text{-}true_\bot}$, $\mathsf{lazy_{map}}\ (\mathsf{lift_{map}}\ (\mathsf{f_3}\ 0))\ \mathsf{A_3}$ does not diverge when $\mathsf{A_3}$ is empty.

Substituting $\mathsf{g_1}$ for $\mathsf{lift_{map}}\ \mathsf{f_1}$, $\mathsf{g_2}\ 0$ for $\mathsf{lift_{map}}\ (\mathsf{f_2}\ 0)$, and $\mathsf{g_3}\ 0$ for $\mathsf{lift_{map}}\ (\mathsf{f_3}\ 0)$ gives a definition for $\mathsf{if_{map}}$ (Figure 4) for which (13) holds.

4.5 Super-Saver Theorems

The following theorems are easy consequences of the fact that $\mathsf{lift_{map}}$ distributes over bottom arrow computations.

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

$$\mathsf{arr}_{\mathsf{map}} : (X \Rightarrow Y) \Rightarrow (X \underset{\mathsf{map}}{\leadsto} Y)$$
$$\mathsf{arr}_{\mathsf{map}}\ f\ A \;:=\; \mathsf{lift}_{\mathsf{map}}\ (\mathsf{arr}_{\bot}\ f)$$

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

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

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

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

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

Figure 4: Mapping arrow definitions.

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

Corollary 2. *If* $[\![e]\!]_{\bot} : X \Rightarrow Y_{\bot}$, *then* $[\![e]\!]_{\mathsf{map}} : X \underset{\mathsf{map}}{\leadsto} Y$ *and* $\mathsf{lift}_{\mathsf{map}}\ [\![e]\!]_{\bot} \equiv [\![e]\!]_{\mathsf{map}}$.

5. Lazy 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 infinite sets in the language implementation, we will need an abstraction that makes it easy to replace computation on points with computation on sets. Therefore, in the preimage arrow, we will confine computation on points to ***lazy preimage mappings***, or just *preimage mappings*, for which application is like applying preimage. The type is

$$X \underset{\mathsf{pre}}{\rightleftharpoons} Y \;::=\; \langle \mathsf{Set}\ Y, \mathsf{Set}\ Y \Rightarrow \mathsf{Set}\ X\rangle \tag{21}$$

Converting a mapping to a lazy preimage mapping:

$$\begin{aligned}&\mathsf{pre} : (X \rightharpoonup Y) \Rightarrow (X \underset{\mathsf{pre}}{\rightleftharpoons} Y)\\ &\mathsf{pre}\ g \;:=\; \mathsf{let}\ \ Y' := \mathsf{range}\ g\\ &\qquad\qquad\qquad\ p := \lambda B.\ \mathsf{preimage}\ g\ B\\ &\qquad\qquad\ \ \mathsf{in}\ \ \langle Y', p\rangle\end{aligned} \tag{22}$$

Applying a preimage mapping to any subset of its codomain:

$$\begin{aligned}&\mathsf{pre\text{-}ap} : (X \underset{\mathsf{pre}}{\rightleftharpoons} Y) \Rightarrow \mathsf{Set}\ Y \Rightarrow \mathsf{Set}\ X\\ &\mathsf{pre\text{-}ap}\ \langle Y', p\rangle\ B \;:=\; p\ (B \cap Y')\end{aligned} \tag{23}$$

The necessary property here is that using pre-ap to compute preimages is the same as computing them from a mapping using preimage.

Theorem 2 (pre-ap computes preimages)**.** *Let* $g \in X \rightharpoonup Y$. *For all* $B \subseteq Y$, $\mathsf{pre\text{-}ap}\ (\mathsf{pre}\ g)\ B = \mathsf{preimage}\ g\ B$.

Proof.

$$\begin{aligned}\mathsf{pre\text{-}ap}\ (\mathsf{pre}\ g)\ B \;&=\; \mathsf{let}\ \ Y' := \mathsf{range}\ g\\ &\qquad\quad\ p := \lambda B.\ \mathsf{preimage}\ g\ B\\ &\qquad\ \mathsf{in}\ \ p\ (B \cap Y')\\ &= \mathsf{preimage}\ g\ (B \cap (\mathsf{range}\ g))\\ &= \mathsf{preimage}\ g\ B\end{aligned}$$

□

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

First, we need a new notion of equivalence.

Definition 3. *Two preimage mappings* $h_1 : X \underset{\mathsf{pre}}{\rightleftharpoons} Y$ *and* $h_2 : X \underset{\mathsf{pre}}{\rightleftharpoons} Y$ *are equivalent, or* $h_1 \equiv h_2$, *when* $\mathsf{pre\text{-}ap}\ h_1\ B = \mathsf{pre\text{-}ap}\ h_2\ B$ *for all* $B \subseteq Y$.

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

5.1 Preimage Mapping Pairing

XXX: moar wurds in this section

Lemma 1 (preimage distributes over $\langle\cdot,\cdot\rangle_{\mathsf{map}}$ and $(\times)$)**.** *Let* $g_1 \in X \rightharpoonup Y_1$ *and* $g_2 \in X \rightharpoonup Y_2$. *For all* $B_1 \subseteq Y_1$ *and* $B_2 \subseteq Y_2$, $\mathsf{preimage}\ \langle g_1, g_2\rangle_{\mathsf{map}}\ (B_1 \times B_2) = (\mathsf{preimage}\ g_1\ B_1) \cap (\mathsf{preimage}\ g_2\ B_2)$.

Theorem 3 (pre distributes over $\langle\cdot,\cdot\rangle_{\mathsf{map}}$)**.** *Let* $g_1 \in X \rightharpoonup Y_1$ *and* $g_2 \in X \rightharpoonup Y_2$. *Then* $\mathsf{pre}\ \langle g_1, g_2\rangle_{\mathsf{map}} \equiv \langle \mathsf{pre}\ g_1, \mathsf{pre}\ g_2\rangle_{\mathsf{pre}}$.

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

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

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

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

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

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

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

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

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

Figure 5: Lazy preimage mappings and operations.

□

5.2 Preimage Mapping Composition

XXX: moar wurds in this section

Lemma 2 (preimage distributes over $(\circ_{\mathsf{map}})$)**.** *Let* $g_1 \in X \rightharpoonup Y$ *and* $g_2 \in Y \rightharpoonup Z$. *For all* $C \subseteq Z$, $\mathsf{preimage}\ (g_2 \circ_{\mathsf{map}} g_1)\ C = \mathsf{preimage}\ g_1\ (\mathsf{preimage}\ g_2\ C)$.

Theorem 4 (pre distributes over $(\circ_{\mathsf{map}})$)**.** *Let* $g_1 \in X \rightharpoonup Y$ *and* $g_2 \in Y \rightharpoonup Z$. *Then* $\mathsf{pre}\ (g_2 \circ_{\mathsf{map}} g_1) \equiv (\mathsf{pre}\ g_2) \circ_{\mathsf{pre}} (\mathsf{pre}\ g_1)$.

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

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

□

5.3 Preimage Mapping Disjoint Union

XXX: moar wurds in this section

Lemma 3 (preimage distributes over $(\uplus_{\mathsf{map}})$)**.** *Let* $g_1 \in X \rightharpoonup Y$ *and* $g_2 \in X \rightharpoonup Y$ *be disjoint mappings. For all* $B \subseteq Y$, $\mathsf{preimage}\ (g_1 \uplus_{\mathsf{map}} g_2)\ B = (\mathsf{preimage}\ g_1\ B) \uplus (\mathsf{preimage}\ g_2\ B)$.

Theorem 5 (pre distributes over $(\uplus_{\mathsf{map}})$)**.** *Let* $g_1 \in X \rightharpoonup Y$ *and* $g_2 \in X \rightharpoonup Y$ *have disjoint domains. Then* $\mathsf{pre}\ (g_1 \uplus_{\mathsf{map}} g_2) \equiv (\mathsf{pre}\ g_1) \uplus_{\mathsf{pre}} (\mathsf{pre}\ g_2)$.

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

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

□

6. Deriving the Preimage Arrow

XXX: intro

$$X \underset{\mathsf{pre}}{\rightsquigarrow} Y \ ::= \ \mathsf{Set}\ X \Rightarrow (X \underset{\mathsf{pre}}{\rightharpoonup} Y) \tag{24}$$

$$\begin{aligned}
&\mathsf{lift}_{\mathsf{pre}} : (X \underset{\mathsf{map}}{\rightsquigarrow} Y) \Rightarrow (X \underset{\mathsf{pre}}{\rightsquigarrow} Y)\\
&\mathsf{lift}_{\mathsf{pre}}\ g\ A \ := \ \mathsf{pre}\ (g\ A)
\end{aligned} \tag{25}$$

Definition 4 (Preimage arrow equivalence)**.** *Two preimage arrow computations* $h_1 : X \underset{\mathsf{pre}}{\rightsquigarrow} Y$ *and* $h_2 : X \underset{\mathsf{pre}}{\rightsquigarrow} Y$ *are equivalent, or* $h_1 \equiv h_2$, *when* $h_1\ A \equiv h_2\ A$ *for all* $A \subseteq X$.

6.1 Distributive Laws

XXX: ensuring $\mathsf{lift}_{\mathsf{pre}}$ distributes over mapping arrow computations is awesome...

Formally, we require the following distributive laws to hold:

$$\mathsf{lift}_{\mathsf{pre}}\ (\mathsf{arr}_{\mathsf{map}}\ f) \ \equiv \ \mathsf{arr}_{\mathsf{pre}}\ f \tag{26}$$

$$\mathsf{lift}_{\mathsf{pre}}\ (g_1\ \&\&\&_{\mathsf{map}}\ g_2) \ \equiv \ (\mathsf{lift}_{\mathsf{pre}}\ g_1)\ \&\&\&_{\mathsf{pre}}\ (\mathsf{lift}_{\mathsf{pre}}\ g_2) \tag{27}$$

$$\mathsf{lift}_{\mathsf{pre}}\ (g_1 \ggg_{\mathsf{map}} g_2) \ \equiv \ (\mathsf{lift}_{\mathsf{pre}}\ g_1) \ggg_{\mathsf{pre}} (\mathsf{lift}_{\mathsf{pre}}\ g_2) \tag{28}$$

$$\begin{aligned}
&\mathsf{lift}_{\mathsf{pre}}\ (\mathsf{if}_{\mathsf{map}}\ g_1\ g_2\ g_3) \ \equiv\\
&\quad \mathsf{if}_{\mathsf{pre}}\ (\mathsf{lift}_{\mathsf{pre}}\ g_1)\ (\lambda 0.\, \mathsf{lift}_{\mathsf{pre}}\ (g_2\ 0))\ (\lambda 0.\, \mathsf{lift}_{\mathsf{pre}}\ (g_3\ 0))
\end{aligned} \tag{29}$$

$$
\begin{aligned}
&\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y} ::= \mathsf{Set\ X} \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y}) \\[1ex]
&\mathsf{arr_{pre}} : (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}) \\
&\mathsf{arr_{pre}\ g\ A} \ := \ \mathsf{pre\ (g\ A)} \\[1ex]
&(\ggg_{\mathsf{pre}}) : (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}) \Rightarrow (\mathsf{Y} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Z}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Z}) \\
&(\mathsf{h_1} \ggg_{\mathsf{pre}} \mathsf{h_2})\ \mathsf{A} \ := \ \mathsf{let}\ \ \mathsf{h_1'} := \mathsf{h_1\ A} \\
&\qquad\qquad\qquad\qquad\quad \mathsf{h_2'} := \mathsf{h_2\ (pre\text{-}range\ h_1')} \\
&\qquad\qquad\qquad\qquad \mathsf{in}\ \ \mathsf{h_2} \circ_{\mathsf{pre}} \mathsf{h_1} \\[1ex]
&(\&\&\&_{\mathsf{pre}}) : (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Z}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y} \times \mathsf{Z}) \\
&(\mathsf{h_1} \ \&\&\&_{\mathsf{pre}}\ \mathsf{h_2})\ \mathsf{A} \ := \ \langle \mathsf{h_1\ A}, \mathsf{h_2\ A} \rangle_{\mathsf{pre}}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{if_{pre}} : (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Bool}) \Rightarrow (1 \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y})) \Rightarrow (1 \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y})) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}) \\
&\mathsf{if_{pre}\ h_1\ h_2\ h_3\ A} \ := \\
&\mathsf{let}\ \ \mathsf{h_1'} := \mathsf{h_1\ A} \\
&\qquad \mathsf{h_2'} := \mathsf{lazy_{pre}\ (h_2\ 0)\ (pre\text{-}ap\ h_1'\ \{true\})} \\
&\qquad \mathsf{h_3'} := \mathsf{lazy_{pre}\ (h_3\ 0)\ (pre\text{-}ap\ h_1'\ \{false\})} \\
&\quad \mathsf{in}\ \ \mathsf{h_2'} \uplus_{\mathsf{pre}} \mathsf{h_3'} \\[2ex]
&\mathsf{lazy_{pre}} : (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}) \\
&\mathsf{lazy_{pre}\ h\ A} \ := \ \mathsf{if}\ (\mathsf{A} = \varnothing)\ (\mathsf{pre}\ \varnothing)\ (\mathsf{h\ A})
\end{aligned}
$$

Figure 6: Preimage arrow definitions.

Clearly $\mathsf{arr_{pre}\ f} := \mathsf{lift_{pre}\ (arr_{map}\ f)}$ meets (26). Figure 6 shows the result of deriving the other combinators from the mapping arrow using distributive laws.

Theorem 6 (preimage arrow correctness)**.** $\mathsf{lift_{pre}}$ *distributes over mapping arrow computations.*

Proof. By structural induction; cases follow. □

6.2 Case: Pairing

Starting with the left-hand side of (27), we expand definitions, apply Theorem 3, and rewrite in terms of $\mathsf{lift_{pre}}$:

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

Substituting $\mathsf{h_1}$ for $\mathsf{lift_{pre}\ g_1}$ and $\mathsf{h_2}$ for $\mathsf{lift_{pre}\ g_2}$, and removing the application of pre-ap from both sides of the equivalence gives a definition of $(\&\&\&_{\mathsf{pre}})$ (Figure 6) for which (27) holds.

6.3 Case: Composition

Starting with the left-hand side of (28), we expand definitions, apply Theorem 4 and rewrite in terms of $\mathsf{lift_{pre}}$:

$$
\begin{aligned}
&\mathsf{pre\text{-}ap\ (lift_{pre}\ (g_1 \ggg_{map} g_2)\ A)\ C} \\
&\quad \equiv \mathsf{let}\ \ \mathsf{g_1'} := \mathsf{g_1\ A} \\
&\qquad\qquad \mathsf{g_2'} := \mathsf{g_2\ (range\ g_1')} \\
&\qquad\quad \mathsf{in}\ \ \mathsf{pre\text{-}ap\ (pre\ (g_2' \circ_{map} g_1'))\ C} \\
&\quad \equiv \mathsf{let}\ \ \mathsf{g_1'} := \mathsf{g_1\ A} \\
&\qquad\qquad \mathsf{g_2'} := \mathsf{g_2\ (range\ g_1')} \\
&\qquad\quad \mathsf{in}\ \ \mathsf{pre\text{-}ap\ ((pre\ g_1') \circ_{pre} (pre\ g_2'))\ C} \\
&\quad \equiv \mathsf{let}\ \ \mathsf{h_1} := \mathsf{lift_{pre}\ g_1\ A} \\
&\qquad\qquad \mathsf{h_2} := \mathsf{lift_{pre}\ g_2\ (pre\text{-}range\ h_1)} \\
&\qquad\quad \mathsf{in}\ \ \mathsf{pre\text{-}ap\ (h_2 \circ_{pre} h_1)\ C}
\end{aligned}
\tag{30}
$$

Substituting $\mathsf{h_1}$ for $\mathsf{lift_{pre}\ g_1}$ and $\mathsf{h_2}$ for $\mathsf{lift_{pre}\ g_2}$, and removing the application of pre-ap from both sides of the equivalence gives a definition of $(\ggg_{\mathsf{pre}})$ (Figure 6) for which (28) holds.

6.4 Case: Conditional

Starting with the left-hand side of (29), we expand terms, apply Theorem 5, rewrite in terms of $\mathsf{lift_{pre}}$, and apply Theorem 2 in the definitions of $\mathsf{h_2}$ and $\mathsf{h_3}$:

$$
\begin{aligned}
&\mathsf{pre\text{-}ap\ (lift_{pre}\ (if_{map}\ g_1\ g_2\ g_3)\ A)\ B} \\
&\quad \equiv \mathsf{let}\ \ \mathsf{g_1'} := \mathsf{g_1\ A} \\
&\qquad\qquad \mathsf{g_2'} := \mathsf{lazy_{map}\ (g_2\ 0)\ (preimage\ g_1'\ \{true\})} \\
&\qquad\qquad \mathsf{g_3'} := \mathsf{lazy_{map}\ (g_3\ 0)\ (preimage\ g_1'\ \{false\})} \\
&\qquad\quad \mathsf{in}\ \ \mathsf{pre\text{-}ap\ (pre\ (g_2' \uplus_{map} g_3'))\ B} \\
&\quad \equiv \mathsf{let}\ \ \mathsf{g_1'} := \mathsf{g_1\ A} \\
&\qquad\qquad \mathsf{g_2'} := \mathsf{lazy_{map}\ (g_2\ 0)\ (preimage\ g_1'\ \{true\})} \\
&\qquad\qquad \mathsf{g_3'} := \mathsf{lazy_{map}\ (g_3\ 0)\ (preimage\ g_1'\ \{false\})} \\
&\qquad\quad \mathsf{in}\ \ \mathsf{pre\text{-}ap\ ((pre\ g_2') \uplus_{pre} (pre\ g_3'))\ B} \\
&\quad \equiv \mathsf{let}\ \ \mathsf{g_1'} := \mathsf{g_1\ A} \\
&\qquad\qquad \mathsf{h_2} := \mathsf{lift_{pre}\ (lazy_{map}\ (g_2\ 0))\ (preimage\ g_1'\ \{true\})} \\
&\qquad\qquad \mathsf{h_3} := \mathsf{lift_{pre}\ (lazy_{map}\ (g_3\ 0))\ (preimage\ g_1'\ \{false\})} \\
&\qquad\quad \mathsf{in}\ \ \mathsf{pre\text{-}ap\ (h_2 \uplus_{pre} h_3)\ B} \\
&\quad \equiv \mathsf{let}\ \ \mathsf{h_1} := \mathsf{lift_{pre}\ g_1\ A} \\
&\qquad\qquad \mathsf{h_2} := \mathsf{lift_{pre}\ (lazy_{map}\ (g_2\ 0))\ (pre\text{-}ap\ h_1\ \{true\})} \\
&\qquad\qquad \mathsf{h_3} := \mathsf{lift_{pre}\ (lazy_{map}\ (g_3\ 0))\ (pre\text{-}ap\ h_1\ \{false\})} \\
&\qquad\quad \mathsf{in}\ \ \mathsf{pre\text{-}ap\ (h_2 \uplus_{pre} h_3)\ B}
\end{aligned}
$$

Replacing mappings with lazy preimage mappings requires removing $\mathsf{lazy_{map}}$. First, we define $\mathsf{lazy_{pre}}$ as in Figure 6. It is not hard to check that

$$
\mathsf{lift_{pre}\ (lazy_{map}\ g)} \ \equiv \ \mathsf{lazy_{pre}\ (lift_{pre}\ g)} \tag{31}
$$

In terms of $\mathsf{lazy_{pre}}$, we have

$$
\begin{aligned}
&\mathsf{pre\text{-}ap\ (lift_{pre}\ (if_{map}\ g_1\ g_2\ g_3)\ A)\ B} \\
&\quad \equiv \mathsf{let}\ \ \mathsf{h_1} := \mathsf{lift_{pre}\ g_1\ A} \\
&\qquad\qquad \mathsf{h_2} := \mathsf{lazy_{pre}\ (lift_{pre}\ (g_2\ 0))\ (pre\text{-}ap\ h_1\ \{true\})} \\
&\qquad\qquad \mathsf{h_3} := \mathsf{lazy_{pre}\ (lift_{pre}\ (g_3\ 0))\ (pre\text{-}ap\ h_1\ \{false\})} \\
&\qquad\quad \mathsf{in}\ \ \mathsf{pre\text{-}ap\ (h_2 \uplus_{pre} h_3)\ B}
\end{aligned}
$$

Substituting $\mathsf{h_1}$ for $\mathsf{lift_{pre}\ g_1}$, $\mathsf{h_2\ 0}$ for $\mathsf{lift_{pre}\ (g_2\ 0)}$ and $\mathsf{h_3\ 0}$ for $\mathsf{lift_{pre}\ (g_3\ 0)}$, and removing the application of pre-ap from both sides of the equivalence gives a definition of $\mathsf{if_{pre}}$ (Figure 6) for which (29) holds.

6.5 Super-Saver Theorems

The following theorems are easy consequences of the fact that $\mathsf{lift_{pre}}$ distributes over mapping arrow computations.

Corollary 3. $\mathsf{arr_{pre}}$*,* $(\&\&\&_{\mathsf{pre}})$ *and* $(\ggg_{\mathsf{pre}})$ *define an arrow.*

Corollary 4. *If* $[\![\mathsf{e}]\!]_{\mathsf{map}} : \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}$*, then for all* $\mathsf{A} \subseteq \mathsf{X}$ *and* $\mathsf{B} \subseteq \mathsf{Y}$*,* $\mathsf{preimage\ ([\![e]\!]_{map}\ \ A)\ B} \equiv \mathsf{pre\text{-}ap\ ([\![e]\!]_{pre}\ \ A)\ B}$.

7. Computable Approximation

8. Preimages of Partial Functions

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.