Trustworthy, Useful Languages for Probabilistic Modeling and Inference

Neil Toronto

A dissertation submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

Brigham Young University

January 2014

ABSTRACT

Trustworthy, Useful Languages for Probabilistic Modeling and Inference

Neil Toronto
Department of Computer Science, BYU
Doctor of Philosophy

XXX: rewrite (was lifted from proposal)

The ideals of exact modeling, and of putting off approximations as long as possible, make Bayesian practice both successful and difficult. To address the difficulties, Bayesians have created languages for modeling and automatic inference. However, there are none that have a well-defined semantics, meaning that there is no way to distinguish between a bug and a feature, and there is no standard by which to prove optimizations correct.

Functional programming researchers have created probabilistic languages with well-defined semantics. Bayesians cannot use them in their day-to-day work, however, because they lack critical features; for example, all but one lack probabilistic conditioning.

I propose that it is possible to use measure-theoretic probability and functional programming theory to create languages with well-defined semantics, that are also useful to practicing Bayesians.

Keywords: Probability, Domain-Specific Languages, Semantics

ACKNOWLEDGMENTS

XXX: write

Table of Contents

List of Figures

"I think you're begging the question," said Haydock, "and I can see looming ahead one of those terrible exercises in probability where six men have white hats and six men have black hats and you have to work it out by mathematics how likely it is that the hats will get mixed up and in what proportion. If you start thinking about things like that, you would go round the bend. Let me assure you of that!"

Agatha Christie, *The Mirror Crack'd*

Chapter 1

Thesis Statement

Functional programming theory and measure-theoretic probability provide a solid foundation for trustworthy, useful languages for constructive probabilistic modeling and inference.

1.1 Motivation

1.2 Statement Terms

Funcional Programming Theory. Blah blah blah.

Blah.

Measure-Theoretic Probability. Blah blah blah.

Blah.

Trustworthy. Blah blah blah.

Blah.

Useful. Blah blah blah.

Blah.

Constructive Probabilitic Modeling. Blah blah blah.

Blah.

Probabilistic Inference. Blah blah blah.

Blah.

1.3 Proof and Supporting Evidence

Semantics. Blah blah blah.

Blah.

Properties. Blah blah blah.

Blah.

Test Cases. Blah blah blah.

Blah.

Chapter 2

Background

XXX: Candidates for exposition:

- Basic probability
- Bayesian inference
- Measure theory
- λ-calculus
- Operational semantics
- Monads, idioms, and arrows

This is a lot of background information. How do I choose? How do I decide how much to include?

Chapter 3

The Semantics of Countable Models

This chapter is derived from work published at the 22nd *Symposium on Implementation and Application of Functional Languages (IFL), 2010.*

This branch of mathematics [Probability] is the only one, I believe, in which good writers frequently get results which are entirely erroneous.

Charles S. Peirce

Probability is notorious for being stubbornly counterintuitive. Any automation of probabilistic calculations or reasoning is therefore helpful. In Bayesian statistics, automation is taking the form of probabilistic languages for specifying random processes, which compute answers to questions about them under constraints.

We believe that any such language should be made to meet a mathematical specification. The reason is simple: if a probabilistic language implementation is made to always meet its maker's expectations, it is almost certainly wrong.

Chapter 4

Interlude: Infinite Programs

Chapter 5

Preimage Computation: Running Probabilistic Programs Backwards

This chapter appears in condensed form in the proceedings of the 23rd *European Symposium on Programming (ESOP), 2014*.

I am so in favor of the actual infinite that instead of admitting that Nature abhors it, as is commonly said, I hold that Nature makes frequent use of it everywhere, in order to show more effectively the perfections of its Author.

Georg Cantor

5.1 Introduction

It is primarily Bayesian practice that drives probabilistic language development. To be useful, a probabilistic language must support **conditioning**, or imposing constraints in a way that preserves the relative probabilities of outputs.

Unfortunately, there is currently no efficient probabilistic language implementation that supports conditioning and does not restrict legal programs. Most commonly, languages that support conditioning disallow recursion, allow only discrete or continuous distributions, and restrict conditions to the form $x = c$.

5.1.1 Probability Densities

These common language restrictions arise from reasoning about probability using **densities**, which are functions from random values to *changes* in probability. While simple and convenient, densities have many limitations. For example, densities for random values with different dimension are incomparable, and they cannot be defined on infinite products. Either limitation rules out recursion.

Densities generally cannot define distributions for the outputs of discontinuous functions. For example, suppose we want to model a thermometer that reports in the range $[0, 100]$, and that the temperature it would report (if it could) is distributed according to a bell curve. We might encode the process as

$$\begin{aligned} \mathsf{t}' \;:=\; & \mathsf{let}\;\; \mathsf{t} := \mathsf{normal}\ \mu\ 1 \\ & \quad\mathsf{in}\;\; \mathsf{max}\ 0\ (\mathsf{min}\ 100\ \mathsf{t}) \end{aligned} \tag{5.1}$$

While t's distribution has a density, the distribution of t' does not.

Densities disallow all but the simplest conditions. **Bayes' law for densities** gives the density of x given an observed y in terms of other densities:

$$p_x(x \mid y) \;=\; \frac{p_y(y \mid x) \cdot \pi_x(x)}{\int p_y(y \mid x) \cdot \pi_x(x)\ dx} \tag{5.2}$$

Bayesians interpret probabilistic processes as defining p_y and π_x, and use (5.2) to find the distribution of "x given $y = c$." Even though "x given $x + y = 0$" has perfectly sensible distribution, Bayes' law for densities cannot express it.

5.1.2 Probability Measures

Measure-theoretic probability [27] is widely believed to be able to define every reasonable distribution that densities cannot. It mainly does this by *assigning probabilities to sets* instead of *assigning changes in probability to values.* Functions that do so are probability

measures. In contrast to densities, probabilities of sets of values with different dimension *are* comparable, and probability measures *can* be defined on infinite products.

If a probability measure P assigns probabilities to subsets of X and $f : X \to Y$, then **preimage measure** defines the distribution over subsets of Y:

$$\Pr[B] \;=\; P(f^{-1}(B)) \tag{5.3}$$

The preimage $f^{-1}(B) = \{a \in X \mid f(a) \in B\}$ is the subset of X for which f yields a value in B, and is well-defined for any f. In the thermometer example (5.1), f would be an interpretation of the program as a function, X would be the set of all random sources, and Y would be $\mathbb{R}$.

Measure-theoretic probability supports any kind of condition. If $\Pr[B] > 0$, the probability of $B' \subseteq Y$ given $B \subseteq Y$ is

$$\Pr[B' \,|\, B] \;=\; \Pr[B' \cap B] \;/\; \Pr[B] \tag{5.4}$$

If $\Pr[B] = 0$, conditional probabilities can be calculated as the limit of $\Pr[B' \,|\, B_n]$ for positive-probability $B_1 \supseteq B_2 \supseteq B_3 \supseteq \cdots$ whose intersection is B. For example, if $Y = \mathbb{R} \times \mathbb{R}$, the distribution of "$\langle x, y\rangle \in Y$ given $x + y = 0$" can be calculated using the descending sequence $B_n = \{\langle x, y\rangle \in Y \mid |x + y| < 2^{-n}\}$.

Only special families of **measurable** sets can be assigned probabilities. Proving measurability, taking limits, and other complications tend to make measure-theoretic probability less attractive, even though it is strictly more powerful.

5.1.3 Measure-Theoretic Semantics

Most purely functional languages allow only nontermination as a side effect, and not probabilistic choice. Programmers therefore encode probabilistic programs as functions from random sources to outputs. Monads and other categorical classes such as idioms (i.e. applicative functors) can make doing so easier [22, 45].

It seems this approach should make it easy to interpret probabilistic programs measure-theoretically. For a probabilistic program $f : X \to Y$, the probability measure on output sets $B \subseteq Y$ should be defined by preimages of B under f and the probability measure on X. Unfortunately, it is difficult to turn this simple-sounding idea into a compositional semantics, for the following reasons.

1. Preimages can be defined only for functions with observable domains, which excludes lambdas.
2. If subsets of X and Y must be measurable, taking preimages under f must preserve measurability (we say f itself is measurable). Proving the conditions under which this is true is difficult, especially if f may not terminate.
3. It is very difficult to define probability measures for arbitrary spaces of measurable functions [5].

Implementing a language based on such a semantics is complicated because

4. Contemporary mathematics is unlike any implementation's host language.
5. It requires running Turing-equivalent programs backward, efficiently, on possibly uncountable sets of outputs.

We address 1 and 4 by developing our semantics in λ_{ZFC} [46], a λ-calculus with infinite sets, and both extensional and intensional functions. We address 5 by deriving and implementing a *conservative approximation* of the semantics.

There seems to be no way to simplify difficulty 2, so we work through it in Section 5.9. The outcome is worth it: we prove all probabilistic programs are measurable, regardless of the inputs on which they do not terminate. This includes uncomputable programs; for example, those that contain real equality tests and limits. We believe this result is the first of its kind, and is general enough to apply to almost all past and future work on probabilistic programming languages.

For difficulty 3, we have discovered that the "first-orderness" of arrows [21] is a perfect fit for the "first-orderness" of measure theory.

5.1.4 Arrow Solution Overview

Using arrows, we define an *exact* semantics and an *approximating* semantics. The exact semantics includes

- A semantic function which, like the arrow calculus semantic function [32], transforms first-order programs into the computations of an arbitrary arrow.
- Arrows for evaluating expressions in different ways.

This commutative diagram describes the relationships among the arrows used to define the exact semantics:

$$\begin{tikzcd}
\mathsf{X} \rightsquigarrow_{\perp} \mathsf{Y} \arrow[r, "\mathsf{lift}_{\mathsf{map}}"] \arrow[d, "\eta_{\perp^*}"'] & \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y} \arrow[r, "\mathsf{lift}_{\mathsf{pre}}"] \arrow[d, "\eta_{\mathsf{map}^*}"] & \mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y} \arrow[d, "\eta_{\mathsf{pre}^*}"] \\
\mathsf{X} \rightsquigarrow_{\perp^*} \mathsf{Y} \arrow[r, "\mathsf{lift}_{\mathsf{map}^*}"'] & \mathsf{X} \underset{\mathsf{map}^*}{\rightsquigarrow} \mathsf{Y} \arrow[r, "\mathsf{lift}_{\mathsf{pre}^*}"'] & \mathsf{X} \underset{\mathsf{pre}^*}{\rightsquigarrow} \mathsf{Y}
\end{tikzcd} \tag{5.5}$$

From top-left to top-right, $\mathsf{X} \rightsquigarrow_{\perp} \mathsf{Y}$ arrow computations are intensional functions that may raise errors, $\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}$ instances produce extensional functions, and $\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}$ instances compute preimages. Instances of arrows in the bottom row are like those in the top, except they thread an infinite store of random values, and can be constructed to always terminate. Most of our correctness theorems rely on proofs that every morphism in (5.5) is a homomorphism.

The approximating semantics has the same semantic function, but its arrows $\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow}' \mathsf{Y}$ and $\mathsf{X} \underset{\mathsf{pre}^*}{\rightsquigarrow}' \mathsf{Y}$ compute conservative approximations. Given a library for representing and operating on rectangular sets, it is directly implementable.

5.2 Operational Metalanguage

We write programs in λ_{ZFC} [46], an untyped, call-by-value, operational λ-calculus designed for deriving implementable programs from contemporary mathematics.

Many mathematical areas are agnostic to their foundations, but measure theory is developed explicitly in **ZFC**: Zermelo-Fraenkel set theory with Choice. ZFC's intensional functions are first-order and it has no general recursion, which makes implementing a language defined by a transformation into ZFC difficult. Targeting λ_{ZFC} instead allows creating an exact semantics and deriving an approximating semantics 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. Almost everything definable in ZFC can be defined by a finite λ_{ZFC} program, and essentially every ZFC theorem applies to λ_{ZFC}'s set values without alteration. Proofs about λ_{ZFC}'s set values apply directly to ZFC sets, assuming the existence of an inaccessible cardinal.[1]

In λ_{ZFC}, algebraic data structures are encoded as sets; e.g. the pair $\langle \mathsf{x}, \mathsf{y} \rangle$ can be encoded as $\{\{\mathsf{x}\}, \{\mathsf{x}, \mathsf{y}\}\}$. Only the *existence* of encodings into sets is important, as it means data structures inherit a defining characteristic of sets: strictness. More precisely, in every data structure, every path between the root and a leaf must have finite length. Less precisely, data may be "infinitely wide" (such as $\mathbb{R}$) but not "infinitely tall" (such as infinite trees and lists).

Though λ_{ZFC} is untyped, it helps in this work to define an auxiliary type system. It is manually checked, polymorphic, and characterized by these rules:

- A free type variable is universally quantified; if uppercase, it denotes 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.

Because the type $\mathsf{Set}\ \mathsf{X}$ denotes the same values as the set $\mathcal{P}\ \mathsf{X}$ (i.e. subsets of the set X) we regard them as equivalent. Similarly, $\langle \mathsf{X}, \mathsf{Y} \rangle$ is equivalent to $\mathsf{X} \times \mathsf{Y}$.

Examples of types are those of the λ_{ZFC} primitives membership $(\in) : \mathsf{x} \Rightarrow \mathsf{Set}\ \mathsf{x} \Rightarrow \mathsf{Bool}$, powerset $\mathcal{P} : \mathsf{Set}\ \mathsf{x} \Rightarrow \mathsf{Set}\ (\mathsf{Set}\ \mathsf{x})$, big union $\bigcup : \mathsf{Set}\ (\mathsf{Set}\ \mathsf{x}) \Rightarrow \mathsf{Set}\ \mathsf{x}$, and the `map`-like $\mathsf{image} : (\mathsf{x} \Rightarrow \mathsf{y}) \Rightarrow \mathsf{Set}\ \mathsf{x} \Rightarrow \mathsf{Set}\ \mathsf{y}$.

[1] A modest assumption, as ZFC+κ is a smaller theory than Coq's [6].

We import ZFC theorems as lemmas; for example:

Lemma 5.1 (extensionality). *For all* $\mathsf{A} : \mathsf{Set}\ \mathsf{x}$ *and* $\mathsf{B} : \mathsf{Set}\ \mathsf{x}$*,* $\mathsf{A} = \mathsf{B}$ *if and only if* $\mathsf{A} \subseteq \mathsf{B}$ *and* $\mathsf{B} \subseteq \mathsf{A}$*.*

Or, $\mathsf{A} = \mathsf{B}$ if and only if they contain the same members.

5.2.1 Internal Equality and External Equivalence

Any λ_{ZFC} term e used as a truth statement means "e reduces to true." Therefore, the terms $(\lambda\,\mathsf{a}.\,\mathsf{a})\ 1$ and 1 are (externally) unequal, but $(\lambda\,\mathsf{a}.\,\mathsf{a})\ 1 = 1$.

Because of the way λ_{ZFC}'s lambda terms are defined, lambda equality is alpha equivalence. For example, $(\lambda\,\mathsf{a}.\,\mathsf{a}) = (\lambda\,\mathsf{b}.\,\mathsf{b})$, but not $(\lambda\,\mathsf{a}.\,2) = (\lambda\,\mathsf{a}.\,1+1)$.

If $e_1 = e_2$, then e_1 and e_2 both terminate, and substituting one for the other in an expression does not change its value. Substitution is also safe if both e_1 and e_2 do not terminate, leading to a coarser notion of equivalence.

Definition 5.2 (observational equivalence). *For terms e_1 and e_2, $e_1 \equiv e_2$ when $e_1 = e_2$, or both e_1 and e_2 do not terminate.*

It might seem helpful to define basic equivalence even more coarsely. However, we want internal and external equality to be similar, and we want to be able to extend "$\equiv$" with type-specific rules.

5.2.2 Additional Functions and Syntactic Forms

We use heavily sugared syntax, with automatic currying, binding forms such as indexed unions $\bigcup_{x \in e_A} e$, destructuring binds as in $\mathsf{swap}\ \langle \mathsf{x}, \mathsf{y} \rangle := \langle \mathsf{y}, \mathsf{x} \rangle$, and comprehensions like $\{\mathsf{x} \in \mathsf{A} \mid \mathsf{x} \in \mathsf{B}\}$. We assume logical operators, bounded quantifiers, and typical set operations are defined.

$$\begin{aligned}
&\mathsf{domain} : (\mathsf{X} \rightharpoonup \mathsf{Y}) \Rightarrow \mathsf{Set}\ \mathsf{X} \\
&\mathsf{domain}\ := \ \mathsf{image}\ \mathsf{fst} \\[1em]
&\mathsf{range} : (\mathsf{X} \rightharpoonup \mathsf{Y}) \Rightarrow \mathsf{Set}\ \mathsf{Y} \\
&\mathsf{range}\ := \ \mathsf{image}\ \mathsf{snd} \\[1em]
&\mathsf{preimage} : (\mathsf{X} \rightharpoonup \mathsf{Y}) \Rightarrow \mathsf{Set}\ \mathsf{Y} \Rightarrow \mathsf{Set}\ \mathsf{X} \\
&\mathsf{preimage}\ \mathsf{g}\ \mathsf{B}\ := \ \{\mathsf{a} \in \mathsf{domain}\ \mathsf{g} \mid \mathsf{g}\ \mathsf{a} \in \mathsf{B}\} \\[1em]
&\mathsf{restrict} : (\mathsf{X} \rightharpoonup \mathsf{Y}) \Rightarrow \mathsf{Set}\ \mathsf{X} \Rightarrow (\mathsf{X} \rightharpoonup \mathsf{Y}) \\
&\mathsf{restrict}\ \mathsf{g}\ \mathsf{A}\ := \ \lambda\,\mathsf{a} \in (\mathsf{A} \cap \mathsf{domain}\ \mathsf{g}).\,\mathsf{g}\ \mathsf{a}
\end{aligned}$$

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

Figure 5.1: Operations on mappings.

A less typical set operation we use is disjoint union:

$$\begin{aligned}
&(\uplus) : \mathsf{Set}\ \mathsf{x} \Rightarrow \mathsf{Set}\ \mathsf{x} \Rightarrow \mathsf{Set}\ \mathsf{x} \\
&\mathsf{A} \uplus \mathsf{B}\ := \ \mathsf{if}\ (\mathsf{A} \cap \mathsf{B} = \varnothing)\ (\mathsf{A} \cup \mathsf{B})\ (\mathsf{take}\ \varnothing)
\end{aligned} \tag{5.6}$$

The primitive $\mathsf{take} : \mathsf{Set}\ \mathsf{x} \Rightarrow \mathsf{x}$ returns the element in a singleton set, and does not terminate when applied to a non-singleton set. Thus, $\mathsf{A} \uplus \mathsf{B}$ terminates only when A and B are disjoint.

In set theory, extensional functions are encoded as sets of input-output pairs; e.g. the increment function for the natural numbers is $\{\langle 0,1\rangle, \langle 1,2\rangle, \langle 2,3\rangle, ...\}$. We call these **mappings** and intensional functions **lambdas**, and use **function** to mean either. As with lambdas, we use adjacency (e.g. $(\mathsf{f}\ \mathsf{x})$) to apply mappings.

Syntax for unnamed mappings is defined by

$$\lambda\, x_a \in e_A.\, e_b\ \ :\equiv\ \ \mathsf{mapping}\ (\lambda\, x_a.\, e_b)\ e_A \tag{5.7}$$

$$\begin{aligned}
&\mathsf{mapping} : (\mathsf{X} \Rightarrow \mathsf{Y}) \Rightarrow \mathsf{Set}\ \mathsf{X} \Rightarrow (\mathsf{X} \rightharpoonup \mathsf{Y}) \\
&\mathsf{mapping}\ \mathsf{f}\ \mathsf{A}\ := \ \mathsf{image}\ (\lambda\,\mathsf{a}.\,\langle \mathsf{a}, \mathsf{f}\ \mathsf{a}\rangle)\ \mathsf{A}
\end{aligned} \tag{5.8}$$

For symmetry with partial functions $\mathsf{x} \Rightarrow \mathsf{y}$, $\mathsf{mapping}$ returns a member of the set $\mathsf{X} \rightharpoonup \mathsf{Y}$ of all partial mappings from X to Y. Fig. 5.1 defines other mapping operations: domain, range,

$\mathsf{preimage}$, $\mathsf{restrict}$, pairing, composition, and disjoint union. The latter three are particularly important in the preimage arrow's derivation.

The set $\mathsf{J} \to \mathsf{X}$ contains all the *total* mappings from J to X; equivalently, all the vectors of X indexed by J (which may be infinite). The function

$$\begin{aligned} &\pi : \mathsf{J} \Rightarrow (\mathsf{J} \to \mathsf{X}) \Rightarrow \mathsf{X} \\ &\pi\ \mathsf{j}\ \mathsf{f}\ :=\ \mathsf{f}\ \mathsf{j} \end{aligned} \tag{5.9}$$

produces projections. This is particularly useful when f is unnamed.

5.3 Arrows and First-Order Semantics

Like monads [51] and idioms [34], arrows [21] thread effects through computations in a way that imposes structure. But arrow computations are always

- Function-like: An arrow computation of type $\mathsf{x} \rightsquigarrow \mathsf{y}$ must behave like a corresponding function of type $\mathsf{x} \Rightarrow \mathsf{y}$ (in a sense we explain shortly).
- First-order: There is no way to derive a computation $\mathsf{app} : \langle \mathsf{x} \rightsquigarrow \mathsf{y}, \mathsf{x} \rangle \rightsquigarrow \mathsf{y}$ from an arrow's minimal definition.

The first property makes arrows a good fit for a compositional translation from expressions to pure functions that operate on random sources. The second property makes arrows a good fit for a measure-theoretic semantics in particular, as app's corresponding function is generally not measurable [5].

5.3.1 Alternative Arrow Definitions and Laws

To make applying measure-theoretic theorems easier, and to simplify interpreting let-calculus expressions as arrow computations, we do not give typical minimal arrow definitions. For each arrow a, instead of $\mathsf{first}_\mathsf{a}$, we define $(\&\&\&_\mathsf{a})$. This combinator is typically called **fanout**, but its use will be clearer if we call it **pairing**. One way to strengthen an arrow a is to define an additional combinator left_a, which can be used to choose an arrow computation based on the result of another. Again, we define a different combinator, ifte_a ("if-then-else").

In a nonstrict λ-calculus, defining a choice combinator allows writing recursive functions using nothing but arrow combinators and lifted, pure functions. However, a strict λ-calculus needs an extra combinator lazy for deferring conditional branches. For example, define the **function arrow** with choice:

$$\begin{aligned}\mathsf{arr\ f} &:= \mathsf{f}\\ (\mathsf{f}_1 \ggg \mathsf{f}_2)\ \mathsf{a} &:= \mathsf{f}_2\ (\mathsf{f}_1\ \mathsf{a})\\ (\mathsf{f}_1\ \&\&\&\ \mathsf{f}_2)\ \mathsf{a} &:= \langle \mathsf{f}_1\ \mathsf{a}, \mathsf{f}_2, \mathsf{a}\rangle\\ \mathsf{ifte\ f}_1\ \mathsf{f}_2\ \mathsf{f}_3\ \mathsf{a} &:= \mathsf{if}\ (\mathsf{f}_1\ \mathsf{a})\ (\mathsf{f}_2\ \mathsf{a})\ (\mathsf{f}_3\ \mathsf{a})\\ \mathsf{lazy\ f\ a} &:= \mathsf{f}\ 0\ \mathsf{a}\end{aligned} \tag{5.10}$$

and try to define the following recursive function:

$$\mathsf{halt\text{-}on\text{-}true} := \mathsf{ifte\ (arr\ id)\ (arr\ id)\ halt\text{-}on\text{-}true} \tag{5.11}$$

In a strict λ-calculus, the defining expression does not terminate. But the following is well-defined in λ_{ZFC}, and loops only when applied to false:

$$\mathsf{halt\text{-}on\text{-}true} := \mathsf{ifte\ (arr\ id)\ (arr\ id)\ (lazy}\ \lambda 0.\, \mathsf{halt\text{-}on\text{-}true)} \tag{5.12}$$

All of our arrows are arrows with choice, so we simply call them arrows.

Definition 5.3 (arrow)**.** *Let* $1 := \{0\}$. *A binary type constructor* $(\rightsquigarrow_\mathsf{a})$ *and*

$$\begin{aligned}\mathsf{arr}_\mathsf{a} &: (\mathsf{x} \Rightarrow \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{y})\\ (\ggg_\mathsf{a}) &: (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{y}) \Rightarrow (\mathsf{y} \rightsquigarrow_\mathsf{a} \mathsf{z}) \Rightarrow (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{z})\\ (\&\&\&_\mathsf{a}) &: (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{z}) \Rightarrow (\mathsf{x} \rightsquigarrow_\mathsf{a} \langle \mathsf{y}, \mathsf{z}\rangle)\\ \mathsf{ifte}_\mathsf{a} &: (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{Bool}) \Rightarrow (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{y})\\ \mathsf{lazy}_\mathsf{a} &: (1 \Rightarrow (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{y})) \Rightarrow (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{y})\end{aligned} \tag{5.13}$$

define an ***arrow*** *if certain monoid, homomorphism, and structural laws hold.*

The arrow homomorphism laws can be put in terms of more general homomorphism properties that deal with distributing an arrow-to-arrow lift, which we use extensively to prove correctness.

Definition 5.4 (arrow homomorphism). *A function* $\mathsf{lift_b} : (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_\mathsf{b} \mathsf{y})$ *is an* ***arrow homomorphism*** *from arrow* a *to arrow* b *if the following distributive laws hold for appropriately typed* f*,* $\mathsf{f_1}$*,* $\mathsf{f_2}$ *and* $\mathsf{f_3}$*:*

$$\mathsf{lift_b}\ (\mathsf{arr_a}\ \mathsf{f}) \equiv \mathsf{arr_b}\ \mathsf{f} \tag{5.14}$$

$$\mathsf{lift_b}\ (\mathsf{f_1} \ggg_\mathsf{a} \mathsf{f_2}) \equiv (\mathsf{lift_b}\ \mathsf{f_1}) \ggg_\mathsf{b} (\mathsf{lift_b}\ \mathsf{f_2}) \tag{5.15}$$

$$\mathsf{lift_b}\ (\mathsf{f_1}\ \&\&\&_\mathsf{a}\ \mathsf{f_2}) \equiv (\mathsf{lift_b}\ \mathsf{f_1})\ \&\&\&_\mathsf{b}\ (\mathsf{lift_b}\ \mathsf{f_2}) \tag{5.16}$$

$$\mathsf{lift_b}\ (\mathsf{ifte_a}\ \mathsf{f_1}\ \mathsf{f_2}\ \mathsf{f_3}) \equiv \mathsf{ifte_b}\ (\mathsf{lift_b}\ \mathsf{f_1})\ (\mathsf{lift_b}\ \mathsf{f_2})\ (\mathsf{lift_b}\ \mathsf{f_3}) \tag{5.17}$$

$$\mathsf{lift_b}\ (\mathsf{lazy_a}\ \mathsf{f}) \equiv \mathsf{lazy_b}\ \lambda 0.\, \mathsf{lift_b}\ (\mathsf{f}\ 0) \tag{5.18}$$

The arrow homomorphism laws state that $\mathsf{arr_a} : (\mathsf{x} \Rightarrow \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{y})$ must be a homomorphism from the function arrow (5.10) to arrow a. Roughly, arrow computations that do not use additional combinators can be transformed into $\mathsf{arr_a}$ applied to a pure computation. They must be *function-like.*

Only a few of the other arrow laws play a role in our semantics and its correctness. We need associativity of $(\ggg_\mathsf{a})$, a pair extraction law, and distribution of pure computations over effectful:

$$(\mathsf{f_1} \ggg_\mathsf{a} \mathsf{f_2}) \ggg_\mathsf{a} \mathsf{f_3} \equiv \mathsf{f_1} \ggg_\mathsf{a} (\mathsf{f_2} \ggg_\mathsf{a} \mathsf{f_3}) \tag{5.19}$$

$$(\mathsf{arr_a}\ \mathsf{f_1}\ \&\&\&_\mathsf{a}\ \mathsf{f_2}) \ggg_\mathsf{a} \mathsf{arr_a}\ \mathsf{snd} \equiv \mathsf{f_2} \tag{5.20}$$

$$\mathsf{arr_a}\ \mathsf{f_1} \ggg_\mathsf{a} (\mathsf{f_2}\ \&\&\&_\mathsf{a}\ \mathsf{f_3}) \equiv (\mathsf{arr_a}\ \mathsf{f_1} \ggg_\mathsf{a} \mathsf{f_2})\ \&\&\&_\mathsf{a}\ (\mathsf{arr_a}\ \mathsf{f_1} \ggg_\mathsf{a} \mathsf{f_3}) \tag{5.21}$$

$$\begin{aligned}\mathsf{arr_a}\ \mathsf{f_1} \ggg_\mathsf{a} \mathsf{ifte_a}\ \mathsf{f_2}\ \mathsf{f_3}\ \mathsf{f_4} \equiv\ & \mathsf{ifte_a}\ (\mathsf{arr_a}\ \mathsf{f_1} \ggg_\mathsf{a} \mathsf{f_2}) \\ & \qquad (\mathsf{arr_a}\ \mathsf{f_1} \ggg_\mathsf{a} \mathsf{f_3}) \\ & \qquad (\mathsf{arr_a}\ \mathsf{f_1} \ggg_\mathsf{a} \mathsf{f_4})\end{aligned} \tag{5.22}$$

$$\mathsf{arr_a\ f_1} \ggg_{\mathsf{a}} \mathsf{lazy_a\ f_2} \ \equiv\ \mathsf{lazy_a}\ \lambda 0.\, \mathsf{arr_a\ f_1} \ggg_{\mathsf{a}} \mathsf{f_2}\ 0 \tag{5.23}$$

Equivalence between different arrow representations is usually proved in a strongly normalizing λ-calculus [31, 32], in which every function is free of effects, including nontermination. Such a λ-calculus has no need for $\mathsf{lazy_a}$, so we could not derive (5.23) from existing arrow laws. We follow Hughes's reasoning [21] for the original arrow laws: it is a function-like property (i.e. it holds for the function arrow), and it cannot not lose, reorder or duplicate effects.

The pair extraction law (5.20), which *can* be derived from existing arrow laws, is a more problematic, in nonstrict λ-calculii as well as λ_{ZFC}. If $\mathsf{f_1}$ can loop, using (5.20) to transform a computation can turn a nonterminating expression into a terminating one, or vice-versa. We could condition the pair extraction law on $\mathsf{f_1}$'s termination. Instead, we require every argument to $\mathsf{arr_a}$ to terminate, which simplifies more proofs.

Rather than prove each arrow law for each arrow, we prove arrows are *epimorphic* to arrows for which the laws are known to hold. (Isomorphism is sufficient but not necessary.)

Definition 5.5 (arrow epimorphism). *An arrow homomorphism* $\mathsf{lift_b} : (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{b}} \mathsf{y})$ *that has a right inverse is an* ***arrow epimorphism*** *from* a *to* b.

Theorem 5.6 (epimorphism implies arrow laws). *If* $\mathsf{lift_b} : (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{b}} \mathsf{y})$ *is an arrow epimorphism and the combinators of* a *define an arrow, then the combinators of* b *define an arrow.*

Proof. For the pair extraction law (5.20), rewrite in terms of $\mathsf{lift_b}$, apply homomorphism laws, and apply the pair extraction law for arrow a:

$$\begin{aligned}
&(\mathsf{arr_b\ f_1} \ \&\&\&_{\mathsf{b}}\ \mathsf{f_2}) \ggg_{\mathsf{b}} \mathsf{arr_b\ snd} \qquad (5.24)\\
&\quad\equiv\ (\mathsf{lift_b}\ (\mathsf{arr_a\ f_1}) \ \&\&\&_{\mathsf{b}}\ (\mathsf{lift_b}\ (\mathsf{lift_b^{-1}\ f_2}))) \ggg_{\mathsf{b}} \mathsf{arr_b\ snd}\\
&\quad\equiv\ \mathsf{lift_b}\ (\mathsf{arr_a\ f_1} \ \&\&\&_{\mathsf{a}}\ \mathsf{lift_b^{-1}\ f_2}) \ggg_{\mathsf{b}} \mathsf{lift_b}\ (\mathsf{arr_a\ snd})\\
&\quad\equiv\ \mathsf{lift_b}\ ((\mathsf{arr_a\ f_1} \ \&\&\&_{\mathsf{a}}\ \mathsf{lift_b^{-1}\ f_2}) \ggg_{\mathsf{b}} \mathsf{arr_a\ snd})
\end{aligned}$$

$$\begin{aligned}
p &::\equiv x := e;\ \ldots\ ; e \\
e &::\equiv x\ e \mid \mathsf{let}\ e\ e \mid \mathsf{env}\ n \mid \langle e, e\rangle \mid \mathsf{fst}\ e \mid \mathsf{snd}\ e \mid \mathsf{if}\ e\ e\ e \mid v \\
v &::\equiv \text{[first-order constants]}
\end{aligned}$$

$$[\![x := e;\ \ldots\ ; e_b]\!]_{\mathsf{a}} :\equiv x := [\![e]\!]_{\mathsf{a}}\ ;\ \ldots\ ; [\![e_b]\!]_{\mathsf{a}}$$

$$\begin{aligned}
[\![x\ e]\!]_{\mathsf{a}} &:\equiv [\![\langle e, \langle\rangle\rangle]\!]_{\mathsf{a}} \ggg_{\mathsf{a}} x \\
[\![\langle e_1, e_2\rangle]\!]_{\mathsf{a}} &:\equiv [\![e_1]\!]_{\mathsf{a}}\ \&\&\&_{\mathsf{a}}\ [\![e_2]\!]_{\mathsf{a}} \\
[\![\mathsf{fst}\ e]\!]_{\mathsf{a}} &:\equiv [\![e]\!]_{\mathsf{a}} \ggg_{\mathsf{a}} \mathsf{arr_a\ fst} \\
[\![\mathsf{snd}\ e]\!]_{\mathsf{a}} &:\equiv [\![e]\!]_{\mathsf{a}} \ggg_{\mathsf{a}} \mathsf{arr_a\ snd} \\
[\![v]\!]_{\mathsf{a}} &:\equiv \mathsf{arr_a}\ (\mathsf{const}\ v)
\end{aligned}
\qquad
\begin{aligned}
[\![\mathsf{let}\ e\ e_b]\!]_{\mathsf{a}} &:\equiv ([\![e]\!]_{\mathsf{a}}\ \&\&\&_{\mathsf{a}}\ \mathsf{arr_a\ id}) \ggg_{\mathsf{a}} [\![e_b]\!]_{\mathsf{a}} \\
[\![\mathsf{env}\ 0]\!]_{\mathsf{a}} &:\equiv \mathsf{arr_a\ fst} \\
[\![\mathsf{env}\ (n+1)]\!]_{\mathsf{a}} &:\equiv \mathsf{arr_a\ snd} \ggg_{\mathsf{a}} [\![\mathsf{env}\ n]\!]_{\mathsf{a}} \\
[\![\mathsf{if}\ e_c\ e_t\ e_f]\!]_{\mathsf{a}} &:\equiv \mathsf{ifte_a}\ [\![e_c]\!]_{\mathsf{a}}\ [\![\mathsf{lazy}\ e_t]\!]_{\mathsf{a}}\ [\![\mathsf{lazy}\ e_f]\!]_{\mathsf{a}} \\
[\![\mathsf{lazy}\ e]\!]_{\mathsf{a}} &:\equiv \mathsf{lazy_a}\ \lambda 0.\, [\![e]\!]_{\mathsf{a}}
\end{aligned}$$

$$\begin{aligned}
\mathsf{id} &:= \lambda\, \mathsf{a.\,a} \\
\mathsf{const\ b} &:= \lambda\, \mathsf{a.\,b}
\end{aligned}
\qquad
\text{subject to } [\![p]\!]_{\mathsf{a}} : \langle\rangle \rightsquigarrow_{\mathsf{a}} \mathsf{y} \text{ for some } \mathsf{y}$$

Figure 5.2: Interpretation of a let-calculus with first-order definitions and De-Bruijn-indexed bindings as arrow a computations.

$$\begin{aligned}
&\equiv \mathsf{lift_b}\ (\mathsf{lift_b^{-1}}\ \mathsf{f_2}) \\
&\equiv \mathsf{f_2}
\end{aligned}$$

The proofs for every other law are similar. □

5.3.2 First-Order Let-Calculus Semantics

Fig. 5.2 defines a transformation from a first-order let-calculus to arrow computations for any arrow a. A program is a sequence of definition statements followed by a final expression. The semantic function $[\![\cdot]\!]_{\mathsf{a}}$ transforms each defining expression and the final expression into arrow computations. Functions are named, but local variables and arguments are not. Instead, variables are referred to by De Bruijn indexes, with 0 referring to the innermost binding.

Perhaps unsurprisingly, interpretations act like stack machines. A final expression has type $\langle\rangle \rightsquigarrow_{\mathsf{a}} \mathsf{y}$, where y is the type of the program's value, and $\langle\rangle$ denotes an empty list, or stack. A let expression pushes a value onto the stack. First-order functions have type $\langle \mathsf{x}, \langle\rangle\rangle \rightsquigarrow_{\mathsf{a}} \mathsf{y}$ where x is the argument type and y is the return type. Application sends a stack containing just an x.

We generally regard programs as if they were their final expressions. Thus, the following definition applies to both programs and expressions.

Definition 5.7 (well-defined expression). *An expression e is* ***well-defined*** *under arrow* a *if* $[\![e]\!]_{\mathsf{a}} : \mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}$ *for some* x *and* y*, and* $[\![e]\!]_{\mathsf{a}}$ *terminates.*

From here on, we assume all expressions are well-defined. (The arrow a will be clear from context.) Well-definedness does not guarantee that *running* an interpretation terminates. It just simplifies statements about expressions, such as the following theorem, on which most of our semantic correctness results rely.

Theorem 5.8 (homomorphisms distribute over expressions). *Let* $\mathsf{lift_b} : (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{b}} \mathsf{y})$ *be an arrow homomorphism. For all e,* $[\![e]\!]_{\mathsf{b}} \equiv \mathsf{lift_b}\ [\![e]\!]_{\mathsf{a}}$.

Proof. By structural induction. Bases cases proceed by expansion and using $\mathsf{arr_b} \equiv \mathsf{lift_b} \circ \mathsf{arr_a}$ (5.14). For example, for constants:

$$\begin{aligned} [\![v]\!]_{\mathsf{b}} &\equiv \mathsf{arr_b}\ (\mathsf{const}\ v) \\ &\equiv \mathsf{lift_b}\ (\mathsf{arr_a}\ (\mathsf{const}\ v)) \\ &\equiv \mathsf{lift_b}\ [\![v]\!]_{\mathsf{a}} \end{aligned} \tag{5.25}$$

Inductive cases proceed by expansion, applying the inductive hypothesis on subterms, and applying distributive laws (5.15)–(5.18). For example, for pairing:

$$\begin{aligned} [\![\langle e_1, e_2\rangle]\!]_{\mathsf{b}} &\equiv [\![e_1]\!]_{\mathsf{b}}\ \&\&\&_{\mathsf{b}}\ [\![e_2]\!]_{\mathsf{b}} \\ &\equiv (\mathsf{lift_b}\ [\![e_1]\!]_{\mathsf{a}})\ \&\&\&_{\mathsf{b}}\ (\mathsf{lift_b}\ [\![e_2]\!]_{\mathsf{a}}) \\ &\equiv \mathsf{lift_b}\ ([\![e_1]\!]_{\mathsf{a}}\ \&\&\&_{\mathsf{a}}\ [\![e_2]\!]_{\mathsf{a}}) \\ &\equiv \mathsf{lift_b}\ [\![\langle e_1, e_2\rangle]\!]_{\mathsf{a}} \end{aligned} \tag{5.26}$$

It is not hard to check the remaining cases. □

$$\mathsf{X} \rightsquigarrow_\perp \mathsf{Y} \ ::= \ \mathsf{X} \Rightarrow \mathsf{Y}_\perp$$

$$\mathsf{arr}_\perp : (\mathsf{X} \Rightarrow \mathsf{Y}) \Rightarrow (\mathsf{X} \rightsquigarrow_\perp \mathsf{Y})$$
$$\mathsf{arr}_\perp\ \mathsf{f} \ := \ \mathsf{f}$$

$$(\ggg_\perp) : (\mathsf{X} \rightsquigarrow_\perp \mathsf{Y}) \Rightarrow (\mathsf{Y} \rightsquigarrow_\perp \mathsf{Z}) \Rightarrow (\mathsf{X} \rightsquigarrow_\perp \mathsf{Z})$$
$$(\mathsf{f}_1 \ggg_\perp \mathsf{f}_2)\ \mathsf{a} \ := \ \mathsf{if}\ (\mathsf{f}_1\ \mathsf{a} = \perp)\ \perp\ (\mathsf{f}_2\ (\mathsf{f}_1\ \mathsf{a}))$$

$$(\&\&\&_\perp) : (\mathsf{X} \rightsquigarrow_\perp \mathsf{Y}_1) \Rightarrow (\mathsf{X} \rightsquigarrow_\perp \mathsf{Y}_2) \Rightarrow (\mathsf{X} \rightsquigarrow_\perp \langle \mathsf{Y}_1, \mathsf{Y}_2 \rangle)$$
$$(\mathsf{f}_1\ \&\&\&_\perp\ \mathsf{f}_2)\ \mathsf{a} \ := \ \mathsf{if}\ (\mathsf{f}_1\ \mathsf{a} = \perp\ \mathsf{or}\ \mathsf{f}_2\ \mathsf{a} = \perp)\ \perp\ \langle \mathsf{f}_1\ \mathsf{a}, \mathsf{f}_2\ \mathsf{a} \rangle$$

$$\mathsf{ifte}_\perp : (\mathsf{X} \rightsquigarrow_\perp \mathsf{Bool}) \Rightarrow (\mathsf{X} \rightsquigarrow_\perp \mathsf{Y}) \Rightarrow (\mathsf{X} \rightsquigarrow_\perp \mathsf{Y}) \Rightarrow (\mathsf{X} \rightsquigarrow_\perp \mathsf{Y})$$
$$\begin{aligned}\mathsf{ifte}_\perp\ \mathsf{f}_1\ \mathsf{f}_2\ \mathsf{f}_3\ \mathsf{a} \ := \ \mathsf{case}\ \ &\mathsf{f}_1\ \mathsf{a} \\ &\mathsf{true} \ \longrightarrow\ \mathsf{f}_2\ \mathsf{a} \\ &\mathsf{false} \ \longrightarrow\ \mathsf{f}_3\ \mathsf{a} \\ &\perp \ \longrightarrow\ \perp\end{aligned}$$

$$\mathsf{lazy}_\perp : (1 \Rightarrow (\mathsf{X} \rightsquigarrow_\perp \mathsf{Y})) \Rightarrow (\mathsf{X} \rightsquigarrow_\perp \mathsf{Y})$$
$$\mathsf{lazy}_\perp\ \mathsf{f}\ \mathsf{a} \ := \ \mathsf{f}\ 0\ \mathsf{a}$$

Figure 5.3: Bottom arrow definitions.

If we assume $\mathsf{lift_b}$ defines correct behavior for arrow b in terms of arrow a, and prove that $\mathsf{lift_b}$ is a homomorphism, then by Theorem 5.8, $[\![\cdot]\!]_\mathsf{b}$ is correct.

5.4 The Bottom Arrow

Using the diagram in (5.5) as a sort of map, we start in the upper-left corner:

$$\begin{tikzcd}
\mathsf{X} \rightsquigarrow_\perp \mathsf{Y} \arrow[r, "\mathsf{lift_{map}}"] \arrow[d, "\eta_{\perp^*}"'] & \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y} \arrow[r, "\mathsf{lift_{pre}}"] \arrow[d, "\eta_{\mathsf{map}^*}"] & \mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y} \arrow[d, "\eta_{\mathsf{pre}^*}"] \\
\mathsf{X} \rightsquigarrow_{\perp^*} \mathsf{Y} \arrow[r, "\mathsf{lift_{map^*}}"'] & \mathsf{X} \underset{\mathsf{map}^*}{\rightsquigarrow} \mathsf{Y} \arrow[r, "\mathsf{lift_{pre^*}}"'] & \mathsf{X} \underset{\mathsf{pre}^*}{\rightsquigarrow} \mathsf{Y}
\end{tikzcd} \tag{5.27}$$

Through Section 5.7, we move across the top to $\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}$.

To use Theorem 5.8 to prove correct the interpretations of expressions as preimage arrow computations, we need the preimage arrow to be homomorphic to a simpler arrow with easily understood behavior. The function arrow (5.10) is an obvious candidate. However, we will need to explicitly handle nontermination as an error value, so we need a slightly more complicated arrow.

Fig. 5.3 defines the ***bottom arrow***. Its computations have type $\mathsf{X} \rightsquigarrow_\perp \mathsf{Y} ::= \mathsf{X} \Rightarrow \mathsf{Y}_\perp$, where $\mathsf{Y}_\perp ::= \mathsf{Y} \cup \{\perp\}$ and $\perp$ is a distinguished error value. The type $\mathsf{Bool}_\perp$, for example, denotes the members of $\mathsf{Bool} \cup \{\perp\} = \{\mathsf{true}, \mathsf{false}, \perp\}$.

To prove the arrow laws, we need a coarser notion of equivalence.

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

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

$$\begin{aligned}
&(\ggg_{\mathsf{map}}) : (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}) \Rightarrow (\mathsf{Y} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Z}) \Rightarrow (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Z}) \\
&(\mathsf{g_1} \ggg_{\mathsf{map}} \mathsf{g_2})\ \mathsf{A} \ := \ \mathsf{let}\ \ \mathsf{g'_1} := \mathsf{g_1}\ \mathsf{A} \\
&\qquad\qquad\qquad\qquad\quad\ \ \mathsf{g'_2} := \mathsf{g_2}\ (\mathsf{range}\ \mathsf{g'_1}) \\
&\qquad\qquad\qquad\qquad\ \ \mathsf{in}\ \ \mathsf{g'_2} \circ_{\mathsf{map}} \mathsf{g'_1}
\end{aligned}$$

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

$$\begin{aligned}
&\mathsf{ifte_{map}} : (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Bool}) \Rightarrow (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}) \\
&\mathsf{ifte_{map}}\ \mathsf{g_1}\ \mathsf{g_2}\ \mathsf{g_3}\ \mathsf{A} \ := \ \mathsf{let}\ \ \mathsf{g'_1} := \mathsf{g_1}\ \mathsf{A} \\
&\qquad\qquad\qquad\qquad\qquad\ \ \mathsf{g'_2} := \mathsf{g_2}\ (\mathsf{preimage}\ \mathsf{g'_1}\ \{\mathsf{true}\}) \\
&\qquad\qquad\qquad\qquad\qquad\ \ \mathsf{g'_3} := \mathsf{g_3}\ (\mathsf{preimage}\ \mathsf{g'_1}\ \{\mathsf{false}\}) \\
&\qquad\qquad\qquad\qquad\quad\ \mathsf{in}\ \ \mathsf{g'_2} \uplus_{\mathsf{map}} \mathsf{g'_3}
\end{aligned}$$

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

$$\begin{aligned}
&\mathsf{lift_{map}} : (\mathsf{X} \rightsquigarrow_{\perp} \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}) \\
&\mathsf{lift_{map}}\ \mathsf{f}\ \mathsf{A} := \{\langle \mathsf{a}, \mathsf{b} \rangle \in \mathsf{mapping}\ \mathsf{f}\ \mathsf{A} \mid \mathsf{b} \neq \perp\}
\end{aligned}$$

Figure 5.4: Mapping arrow definitions.

Definition 5.9 (bottom arrow equivalence)**.** *Two computations* $\mathsf{f_1} : \mathsf{X} \rightsquigarrow_{\perp} \mathsf{Y}$ *and* $\mathsf{f_2} : \mathsf{X} \rightsquigarrow_{\perp} \mathsf{Y}$ *are equivalent, or* $\mathsf{f_1} \equiv \mathsf{f_2}$*, when* $\mathsf{f_1}\ \mathsf{a} \equiv \mathsf{f_2}\ \mathsf{a}$ *for all* $\mathsf{a} \in \mathsf{X}$*.*

Theorem 5.10. $\mathsf{arr_{\perp}}$*,* $(\&\&\&_{\perp})$*,* $(\ggg_{\perp})$*,* $\mathsf{ifte_{\perp}}$ *and* $\mathsf{lazy_{\perp}}$ *define an arrow.*

Proof. The bottom arrow is epimorphic to (in fact, isomorphic to) the Maybe monad's Kleisli arrow. □

5.5 Deriving the Mapping Arrow

Theorems about functions in set theory tend to be about mappings, not about lambdas that may raise errors. As in intermediate step, then, we need an arrow whose computations produce mappings or are mappings themselves.

It is tempting to try to make the mapping arrow's computations mapping-valued; i.e. $\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y} ::= \mathsf{X} \rightharpoonup \mathsf{Y}$, with $\mathsf{f_1} \ggg_{\mathsf{map}} \mathsf{f_2} := \mathsf{f_2} \circ_{\mathsf{map}} \mathsf{f_1}$ and $\mathsf{f_1}\ \&\&\&_{\mathsf{map}}\ \mathsf{f_2} := \langle \mathsf{f_1}, \mathsf{f_2} \rangle_{\mathsf{map}}$. Unfortunately, we could not define $\mathsf{arr_{map}} : (\mathsf{X} \Rightarrow \mathsf{Y}) \Rightarrow (\mathsf{X} \rightharpoonup \mathsf{Y})$: to define a mapping, we need a domain, but lambdas' domains are unobservable.

To parameterize mapping arrow computations on a domain, we define the ***mapping arrow*** computation type as

$$\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y} \ ::= \ \mathsf{Set}\ \mathsf{X} \Rightarrow (\mathsf{X} \rightharpoonup \mathsf{Y}) \tag{5.28}$$

The absence of $\bot$ in $\mathsf{Set}\ \mathsf{X} \Rightarrow (\mathsf{X} \rightharpoonup \mathsf{Y})$, and the fact that type parameters X and Y denote sets, will make it easier to apply well-known theorems from measure theory, which know nothing of lambda types and propagating error values.

To use Theorem 5.8 to prove that expressions interpreted using $[\![\cdot]\!]_{\mathsf{map}}$ behave correctly with respect to $[\![\cdot]\!]_{\bot}$, we need to define correctness using a lift from the bottom arrow to the mapping arrow. It is helpful to have a standalone function $\mathsf{domain}_{\bot}$ that computes the subset of A on which f does not return $\bot$. We define that first, and then define $\mathsf{lift}_{\mathsf{map}}$ in terms of it:

$$\begin{aligned}
&\mathsf{domain}_{\bot} : (\mathsf{X} \rightsquigarrow_{\bot} \mathsf{Y}) \Rightarrow \mathsf{Set}\ \mathsf{X} \Rightarrow \mathsf{Set}\ \mathsf{X} \\
&\mathsf{domain}_{\bot}\ \mathsf{f}\ \mathsf{A}\ :=\ \{\mathsf{a} \in \mathsf{A} \mid \mathsf{f}\ \mathsf{a} \neq \bot\}
\end{aligned} \tag{5.29}$$

$$\begin{aligned}
&\mathsf{lift}_{\mathsf{map}} : (\mathsf{X} \rightsquigarrow_{\bot} \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}) \\
&\mathsf{lift}_{\mathsf{map}}\ \mathsf{f}\ \mathsf{A}\ :=\ \mathsf{mapping}\ \mathsf{f}\ (\mathsf{domain}_{\bot}\ \mathsf{f}\ \mathsf{A})
\end{aligned} \tag{5.30}$$

So $\mathsf{lift}_{\mathsf{map}}\ \mathsf{f}\ \mathsf{A}$ is like $\mathsf{mapping}\ \mathsf{f}\ \mathsf{A}$, except the domain does not contain inputs that produce errors or nontermination—a good notion of correctness.

If $\mathsf{lift}_{\mathsf{map}}$ is to be a homomorphism, mapping arrow computation equivalence needs to be more extensional.

Definition 5.11 (mapping arrow equivalence). *Two computations* $\mathsf{g}_1 : \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}$ *and* $\mathsf{g}_2 : \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}$ *are equivalent, or* $\mathsf{g}_1 \equiv \mathsf{g}_2$*, when* $\mathsf{g}_1\ \mathsf{A} \equiv \mathsf{g}_2\ \mathsf{A}$ *for all* $\mathsf{A} \subseteq \mathsf{X}$.

Clearly $\mathsf{arr}_{\mathsf{map}} := \mathsf{lift}_{\mathsf{map}} \circ \mathsf{arr}_{\bot}$ meets the first homomorphism law (5.14). The remainder of this section derives $(\&\&\&_{\mathsf{map}})$, $(\ggg_{\mathsf{map}})$, $\mathsf{ifte}_{\mathsf{map}}$ and $\mathsf{lazy}_{\mathsf{map}}$ from bottom arrow combinators, in a way that ensures $\mathsf{lift}_{\mathsf{map}}$ is an arrow homomorphism. Fig. 5.4 contains the resulting definitions.

5.5.1 Composition

Starting with the left side of (5.15), we expand definitions, simplify f by restricting it to a set for which $\mathsf{f_1\ a} \neq \bot$, and substitute f's definition:

$$\begin{aligned}
\mathsf{lift_{map}\ (f_1 \ggg f_2)\ A} &\equiv \mathsf{let}\quad \mathsf{f} := \lambda\mathsf{a}.\, \mathsf{if\ (f_1\ a = \bot)\ \bot\ (f_2\ (f_1\ a))} \\
&\qquad\quad \mathsf{A'} := \mathsf{domain_\bot\ f\ A} \\
&\qquad \mathsf{in\ \ mapping\ f\ A'} \\
&\equiv \mathsf{let}\quad \mathsf{f} := \lambda\mathsf{a}.\, \mathsf{f_2\ (f_1\ a)} \\
&\qquad\quad \mathsf{A'} := \mathsf{domain_\bot\ f\ (domain_\bot\ f_1\ A)} \\
&\qquad \mathsf{in\ \ mapping\ f\ A'} \\
&\equiv \mathsf{let}\ \ \mathsf{A'} := \{\mathsf{a} \in \mathsf{domain_\bot\ f_1\ A} \mid \mathsf{f_2\ (f_1\ a)} \neq \bot\} \\
&\qquad \mathsf{in}\ \ \lambda\mathsf{a} \in \mathsf{A'}.\, \mathsf{f_2\ (f_1\ a)}
\end{aligned} \tag{5.31}$$

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

$$\begin{aligned}
\mathsf{lift_{map}\ (f_1 \ggg f_2)\ A} &\equiv \mathsf{let}\ \ \mathsf{g_1} := \mathsf{lift_{map}\ f_1\ A} \\
&\qquad\quad \mathsf{A'} := \mathsf{preimage\ g_1\ (domain_\bot\ f_2\ (range\ g_1))} \\
&\qquad \mathsf{in}\ \ \lambda\mathsf{a} \in \mathsf{A'}.\, \mathsf{f_2\ (g_1\ a)} \\
&\equiv \mathsf{let}\ \ \mathsf{g_1} := \mathsf{lift_{map}\ f_1\ A} \\
&\qquad\quad \mathsf{g_2} := \mathsf{lift_{map}\ f_2\ (range\ g_1)} \\
&\qquad\quad \mathsf{A'} := \mathsf{preimage\ g_1\ (domain\ g_2)} \\
&\qquad \mathsf{in}\ \ \lambda\mathsf{a} \in \mathsf{A'}.\, \mathsf{g_2\ (g_1\ a)} \\
&\equiv \mathsf{let}\ \ \mathsf{g_1} := \mathsf{lift_{map}\ f_1\ A} \\
&\qquad\quad \mathsf{g_2} := \mathsf{lift_{map}\ f_2\ (range\ g_1)} \\
&\qquad \mathsf{in}\ \ \mathsf{g_2} \circ_{\mathsf{map}} \mathsf{g_1}
\end{aligned} \tag{5.32}$$

Substituting $\mathsf{g_1}$ for $\mathsf{lift_{map}\ f_1}$ and $\mathsf{g_2}$ for $\mathsf{lift_{map}\ f_2}$ gives a definition for $(\ggg_{\mathsf{map}})$ (Fig. 5.4) for which (5.15) holds.

5.5.2 Pairing

Starting with the left side of (5.16), we expand definitions and replace the definition of A' with one that does not depend on f:

$$\begin{aligned}
\mathsf{lift_{map}}\ (\mathsf{f_1}\ \&\&\&_\perp\ \mathsf{f_2})\ \mathsf{A} \ &\equiv\ \mathsf{let}\quad \mathsf{f} := \lambda\mathsf{a}.\,\mathsf{if}\ (\mathsf{f_1\ a} = \perp\ \mathsf{or}\ \mathsf{f_2\ a} = \perp)\ \perp\ \langle \mathsf{f_1\ a}, \mathsf{f_2\ a}\rangle \\
&\qquad\qquad \mathsf{A}' := \mathsf{domain}_\perp\ \mathsf{f\ A} \\
&\qquad\ \ \mathsf{in}\ \ \mathsf{mapping\ f\ A'} \\
&\equiv\ \mathsf{let}\ \ \mathsf{A}' := \mathsf{domain}_\perp\ \mathsf{f_1\ A} \cap \mathsf{domain}_\perp\ \mathsf{f_2\ A} \\
&\qquad\ \ \mathsf{in}\ \ \lambda\mathsf{a} \in \mathsf{A}'.\,\langle \mathsf{f_1\ a}, \mathsf{f_2\ a}\rangle
\end{aligned} \tag{5.33}$$

We finish by converting bottom arrow computations to the mapping arrow and rewriting in terms of $\langle\cdot,\cdot\rangle_{\mathsf{map}}$:

$$\begin{aligned}
\mathsf{lift_{map}}\ (\mathsf{f_1}\ \&\&\&_\perp\ \mathsf{f_2})\ \mathsf{A} \ &\equiv\ \mathsf{let}\ \ \mathsf{g_1} := \mathsf{lift_{map}\ f_1\ A} \\
&\qquad\quad\ \mathsf{g_2} := \mathsf{lift_{map}\ f_2\ A} \\
&\qquad\quad\ \mathsf{A}' := \mathsf{domain\ g_1} \cap \mathsf{domain\ g_2} \\
&\qquad\ \ \mathsf{in}\ \ \lambda\mathsf{a} \in \mathsf{A}'.\,\langle \mathsf{g_1\ a}, \mathsf{g_2\ a}\rangle \\
&\equiv\ \langle \mathsf{lift_{map}\ f_1\ A}, \mathsf{lift_{map}\ f_2\ A}\rangle_{\mathsf{map}}
\end{aligned} \tag{5.34}$$

Substituting $\mathsf{g_1}$ for $\mathsf{lift_{map}\ f_1}$ and $\mathsf{g_2}$ for $\mathsf{lift_{map}\ f_2}$ gives a definition for $(\&\&\&_{\mathsf{map}})$ (Fig. 5.4) for which (5.16) holds.

5.5.3 Conditional

Starting with the left side of (5.17), we expand definitions, and simplify f by restricting it to a domain for which $\mathsf{f_1\ a} \neq \perp$:

$$\begin{aligned}
\mathsf{lift_{map}}\ (\mathsf{ifte}_\perp\ \mathsf{f_1\ f_2\ f_3})\ \mathsf{A} \ \equiv\ \mathsf{let}\ \ &\mathsf{f} := \lambda\mathsf{a}.\,\mathsf{case}\ \ \mathsf{f_1\ a} \\
&\qquad\qquad\quad \mathsf{true} \longrightarrow \mathsf{f_2\ a} \\
&\qquad\qquad\quad \mathsf{false} \longrightarrow \mathsf{f_3\ a} \\
&\qquad\qquad\quad \perp \longrightarrow \perp \\
\mathsf{in}\ \ &\mathsf{mapping\ f\ (domain_\perp\ f\ A)}
\end{aligned} \tag{5.35}$$

$$\begin{aligned}\equiv\ \mathsf{let}\ \ \mathsf{g_1} &:= \mathsf{mapping\ f\ A}\\ \mathsf{A_2} &:= \mathsf{preimage\ g_1\ \{true\}}\\ \mathsf{A_3} &:= \mathsf{preimage\ g_1\ \{false\}}\\ \mathsf{f} &:= \lambda\mathsf{a.\,if\ (f_1\ a)\ (f_2\ a)\ (f_3\ a)}\\ \mathsf{in}\ \ &\mathsf{mapping\ f\ (domain_\perp\ f\ (A_2 \uplus A_3))}\end{aligned}$$

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

$$\begin{aligned}\mathsf{lift_{map}\ (ifte_\perp\ f_1\ f_2\ f_3)\ A} \equiv\ \mathsf{let}\ \ \mathsf{g_1} &:= \mathsf{lift_{map}\ f_1\ A}\\ \mathsf{g_2} &:= \mathsf{lift_{map}\ f_2\ (preimage\ g_1\ \{true\})}\\ \mathsf{g_3} &:= \mathsf{lift_{map}\ f_3\ (preimage\ g_1\ \{false\})}\\ \mathsf{A'} &:= \mathsf{domain\ g_2 \uplus domain\ g_3}\\ \mathsf{in}\ \ &\lambda\mathsf{a} \in \mathsf{A'}.\,\mathsf{if\ (a \in domain\ g_2)\ (g_2\ a)\ (g_3\ a)}\\[1em] \equiv\ \mathsf{let}\ \ \mathsf{g_1} &:= \mathsf{lift_{map}\ f_1\ A}\\ \mathsf{g_2} &:= \mathsf{lift_{map}\ f_2\ (preimage\ g_1\ \{true\})}\\ \mathsf{g_3} &:= \mathsf{lift_{map}\ f_3\ (preimage\ g_1\ \{false\})}\\ \mathsf{in}\ \ &\mathsf{g_2 \uplus_{map} g_3}\end{aligned} \tag{5.36}$$

Substituting $\mathsf{g_1}$ for $\mathsf{lift_{map}\ f_1}$, $\mathsf{g_2}$ for $\mathsf{lift_{map}\ f_2}$, and $\mathsf{g_3}$ for $\mathsf{lift_{map}\ f_3}$ gives a definition for $\mathsf{ifte_{map}}$ (Fig. 5.4) for which (5.17) holds.

5.5.4 Laziness

Starting with the left side of (5.18), we expand definitions:

$$\begin{aligned}\mathsf{lift_{map}\ (lazy_\perp\ f)\ A} \equiv\ \mathsf{let}\ \ &\mathsf{A'} := \mathsf{domain_\perp\ (\lambda a.\,f\ 0\ a)\ A}\\ \mathsf{in}\ \ &\mathsf{mapping\ (\lambda a.\,f\ 0\ a)\ A'}\end{aligned} \tag{5.37}$$

It appears we need an η rule to continue, which λ_{ZFC} does not have (i.e. $\lambda x.\, e\ x \not\equiv e$ because e may not terminate). Fortunately, we can use weaker facts. If $\mathsf{A} \neq \varnothing$, then $\mathsf{domain_\perp\ (\lambda a.\,f\ 0\ a)\ A} \equiv \mathsf{domain_\perp\ (f\ 0)\ A}$. Further, it terminates if and only if $\mathsf{mapping\ (f\ 0)\ A'}$ terminates. Therefore, if $\mathsf{A} \neq \varnothing$, we can replace $\lambda\mathsf{a.\,f\ 0\ a}$ with $\mathsf{f\ 0}$. If $\mathsf{A} = \varnothing$, then

$\mathsf{lift_{map}}\ (\mathsf{lazy_\perp}\ \mathsf{f})\ \mathsf{A} = \varnothing$ (the empty mapping), so

$$\begin{aligned} \mathsf{lift_{map}}\ (\mathsf{lazy_\perp}\ \mathsf{f})\ \mathsf{A} &\equiv \mathsf{if}\ (\mathsf{A} = \varnothing)\ \varnothing\ (\mathsf{mapping}\ (\mathsf{f}\ 0)\ (\mathsf{domain_\perp}\ (\mathsf{f}\ 0)\ \mathsf{A})) \\ &\equiv \mathsf{if}\ (\mathsf{A} = \varnothing)\ \varnothing\ (\mathsf{lift_{map}}\ (\mathsf{f}\ 0)\ \mathsf{A}) \end{aligned} \tag{5.38}$$

Substituting $\mathsf{g}\ 0$ for $\mathsf{lift_{map}}\ (\mathsf{f}\ 0)$ gives a $\mathsf{lazy_{map}}$ (Fig. 5.4) for which (5.18) holds.

5.5.5 Correctness

Theorem 5.12 (mapping arrow correctness). $\mathsf{lift_{map}}$ *is a homomorphism.*

Proof. By construction. □

Corollary 5.13 (semantic correctness). *For all e, $[\![e]\!]_{\mathsf{map}} \equiv \mathsf{lift_{map}}\ [\![e]\!]_\perp$.*

Without restrictions, mapping arrow computations can be quite unruly. For example, the following computation is well-typed, but returns the identity mapping on Bool when applied to an empty domain, and the empty mapping when applied to any other domain:

$$\begin{aligned} &\mathsf{nonmonotone} : \mathsf{Bool} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Bool} \\ &\mathsf{nonmonotone}\ \mathsf{A}\ := \ \mathsf{if}\ (\mathsf{A} = \varnothing)\ (\mathsf{mapping}\ \mathsf{id}\ \mathsf{Bool})\ \varnothing \end{aligned} \tag{5.39}$$

It would be nice if we could be sure that every $\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}$ is not only monotone, but acts as if it returned restricted mappings. The following equivalent property is easier to state, and makes proving the arrow laws simple.

Definition 5.14 (mapping arrow law). *Let $\mathsf{g} : \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}$. If there exists an $\mathsf{f} : \mathsf{X} \rightsquigarrow_\perp \mathsf{Y}$ such that $\mathsf{g} \equiv \mathsf{lift_{map}}\ \mathsf{f}$, then g obeys the* ***mapping arrow law****.*

By homomorphism of $\mathsf{lift_{map}}$, mapping arrow combinators preserve this law. It is therefore safe to assume that the mapping arrow law holds for all $\mathsf{g} : \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}$.

Theorem 5.15. $\mathsf{lift_{map}}$ *is an arrow epimorphism.*

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

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

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

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

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

$$\begin{aligned}
&\langle \cdot, \cdot \rangle_{\mathsf{pre}} : (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y_1}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y_2}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y_1} \times \mathsf{Y_2}) \\
&\langle \langle \mathsf{Y'_1}, \mathsf{p_1} \rangle, \langle \mathsf{Y'_2}, \mathsf{p_2} \rangle \rangle_{\mathsf{pre}} \ := \ \mathsf{let}\ \ \mathsf{Y'} := \mathsf{Y'_1} \times \mathsf{Y'_2} \\
&\qquad\qquad\qquad\qquad\qquad \mathsf{p} := \lambda \mathsf{B}. \bigcup_{\langle \mathsf{b_1}, \mathsf{b_2} \rangle \in \mathsf{B}} (\mathsf{p_1}\ \{\mathsf{b_1}\}) \cap (\mathsf{p_2}\ \{\mathsf{b_2}\}) \\
&\qquad\qquad\qquad\qquad \mathsf{in}\ \ \langle \mathsf{Y'}, \mathsf{p} \rangle
\end{aligned}$$

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

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

Figure 5.5: Lazy preimage mappings and operations.

Proof. Follows from Theorem 5.12 and restriction of $\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}$ to instances for which the mapping arrow law (Definition 5.14) holds. □

Corollary 5.16. $\mathsf{arr_{map}}$, $(\&\&\&_{\mathsf{map}})$, $(\ggg_{\mathsf{map}})$, $\mathsf{ifte_{map}}$ *and* $\mathsf{lazy_{map}}$ *define an arrow.*

5.6 Lazy Preimage Mappings

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

If we wish to compute with infinite sets in the language implementation, we will need an abstraction that makes it easy to replace computation on points with computation on sets whose representations allow efficient operations. Therefore, in the preimage arrow, we confine computation on points to instances of

$$\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y} \ ::= \ \langle \mathsf{Set\ Y}, \mathsf{Set\ Y} \Rightarrow \mathsf{Set\ X} \rangle \tag{5.40}$$

Like a mapping, an $\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y}$ has an observable domain—but computing the input-output pairs is delayed. We therefore call these ***lazy preimage mappings***.

Converting a mapping to a lazy preimage mapping requires constructing a delayed application of $\mathsf{preimage}$:

$$\begin{aligned} &\mathsf{pre} : (\mathsf{X} \rightharpoonup \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y}) \\ &\mathsf{pre}\ \mathsf{g}\ := \ \langle \mathsf{range}\ \mathsf{g}, \lambda \mathsf{B}.\, \mathsf{preimage}\ \mathsf{g}\ \mathsf{B} \rangle \end{aligned} \tag{5.41}$$

To apply a preimage mapping to some B, we intersect B with its range and apply the preimage function:

$$\begin{aligned} &\mathsf{ap_{pre}} : (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y}) \Rightarrow \mathsf{Set}\ \mathsf{Y} \Rightarrow \mathsf{Set}\ \mathsf{X} \\ &\mathsf{ap_{pre}}\ \langle \mathsf{Y}', \mathsf{p} \rangle\ \mathsf{B}\ := \ \mathsf{p}\ (\mathsf{B} \cap \mathsf{Y}') \end{aligned} \tag{5.42}$$

Preimage arrow correctness depends on this fact: that using $\mathsf{ap_{pre}}$ to compute preimages is the same as computing them from a mapping using $\mathsf{preimage}$.

Lemma 5.17. *Let* $\mathsf{g} \in \mathsf{X} \rightharpoonup \mathsf{Y}$. *For all* $\mathsf{B} \subseteq \mathsf{Y}$ *and* Y' *such that* $\mathsf{range}\ \mathsf{g} \subseteq \mathsf{Y}' \subseteq \mathsf{Y}$, $\mathsf{preimage}\ \mathsf{g}\ (\mathsf{B} \cap \mathsf{Y}') = \mathsf{preimage}\ \mathsf{g}\ \mathsf{B}$.

Theorem 5.18 ($\mathsf{ap_{pre}}$ computes preimages)**.** *Let* $\mathsf{g} \in \mathsf{X} \rightharpoonup \mathsf{Y}$. *For all* $\mathsf{B} \subseteq \mathsf{Y}$, $\mathsf{ap_{pre}}\ (\mathsf{pre}\ \mathsf{g})\ \mathsf{B} = \mathsf{preimage}\ \mathsf{g}\ \mathsf{B}$.

Proof. Expand definitions and apply Lemma 5.17 with $\mathsf{Y}' = \mathsf{range}\ \mathsf{g}$. □

Fig. 5.5 defines more operations on preimage mappings, including pairing, composition, and disjoint union operations corresponding to the mapping operations in Fig. 5.1. To prove them correct, we need preimage mappings to be equivalent when they compute the same preimages.

Definition 5.19 (preimage mapping equivalence)**.** $\mathsf{h}_1 : \mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y}$ *and* $\mathsf{h}_2 : \mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y}$ *are equivalent, or* $\mathsf{h}_1 \equiv \mathsf{h}_2$, *when* $\mathsf{ap_{pre}}\ \mathsf{h}_1\ \mathsf{B} \equiv \mathsf{ap_{pre}}\ \mathsf{h}_2\ \mathsf{B}$ *for all* $\mathsf{B} \subseteq \mathsf{Y}$.

Similarly to proving arrows correct, we prove the operations in Fig. 5.5 are correct by proving that pre is a homomorphism (though not an arrow homomorphism): it distributes over mapping operations to yield preimage mapping operations. The remainder of this section states these distributive properties as theorems and proves them. We will use these theorems to derive the preimage arrow from the mapping arrow.

5.6.1 Composition

To prove pre distributes over mapping composition, we can make more or less direct use of the fact that $\mathsf{preimage}$ distributes over mapping composition.

Lemma 5.20 ($\mathsf{preimage}$ distributes over $(\circ_{\mathsf{map}})$)**.** *Let* $\mathsf{g}_1 \in \mathsf{X} \rightharpoonup \mathsf{Y}$ *and* $\mathsf{g}_2 \in \mathsf{Y} \rightharpoonup \mathsf{Z}$*. 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 5.21 (pre distributes over $(\circ_{\mathsf{map}})$)**.** *Let* $\mathsf{g}_1 \in \mathsf{X} \rightharpoonup \mathsf{Y}$ *and* $\mathsf{g}_2 \in \mathsf{Y} \rightharpoonup \mathsf{Z}$*. Then* $\mathsf{pre}\ (\mathsf{g}_2 \circ_{\mathsf{map}} \mathsf{g}_1) \equiv (\mathsf{pre}\ \mathsf{g}_2) \circ_{\mathsf{pre}} (\mathsf{pre}\ \mathsf{g}_1)$*.*

Proof. Let $\langle \mathsf{Z}', \mathsf{p}_2 \rangle := \mathsf{pre}\ \mathsf{g}_2$ and $\mathsf{C} \subseteq \mathsf{Z}$. Starting from the right side, expand definitions, apply Theorem 5.18, apply Lemma 5.20, and apply Theorem 5.18 again:

$$\begin{aligned}
&\mathsf{ap}_{\mathsf{pre}}\ ((\mathsf{pre}\ \mathsf{g}_2) \circ_{\mathsf{pre}} (\mathsf{pre}\ \mathsf{g}_1))\ \mathsf{C} \\
&\quad \equiv \begin{array}[t]{l}\mathsf{let}\ \ \mathsf{h} := \lambda \mathsf{C}.\, \mathsf{ap}_{\mathsf{pre}}\ (\mathsf{pre}\ \mathsf{g}_1)\ (\mathsf{p}_2\ \mathsf{C}) \\ \quad \mathsf{in}\ \ \mathsf{h}\ (\mathsf{C} \cap \mathsf{Z}')\end{array} \\
&\quad \equiv \mathsf{ap}_{\mathsf{pre}}\ (\mathsf{pre}\ \mathsf{g}_1)\ (\mathsf{p}_2\ (\mathsf{C} \cap \mathsf{Z}')) \\
&\quad \equiv \mathsf{ap}_{\mathsf{pre}}\ (\mathsf{pre}\ \mathsf{g}_1)\ (\mathsf{ap}_{\mathsf{pre}}\ (\mathsf{pre}\ \mathsf{g}_2)\ \mathsf{C}) \\
&\quad \equiv \mathsf{preimage}\ \mathsf{g}_1\ (\mathsf{preimage}\ \mathsf{g}_2\ \mathsf{C}) \\
&\quad \equiv \mathsf{preimage}\ (\mathsf{g}_2 \circ_{\mathsf{map}} \mathsf{g}_1)\ \mathsf{C} \\
&\quad \equiv \mathsf{ap}_{\mathsf{pre}}\ (\mathsf{pre}\ (\mathsf{g}_2 \circ_{\mathsf{map}} \mathsf{g}_1))\ \mathsf{C}
\end{aligned} \tag{5.43}$$

□

5.6.2 Pairing

We have less luck with pairing than with composition, because $\mathsf{preimage}$ does not distribute over pairing. Fortunately, it distributes over pairing and cartesian product together.

Lemma 5.22 ($\mathsf{preimage}$ distributes over $\langle\cdot,\cdot\rangle_{\mathsf{map}}$ and $(\times)$)**.** *Let* $\mathsf{g}_1 \in \mathsf{X} \rightharpoonup \mathsf{Y}_1$ *and* $\mathsf{g}_2 \in \mathsf{X} \rightharpoonup \mathsf{Y}_2$*. For all* $\mathsf{B}_1 \subseteq \mathsf{Y}_1$ *and* $\mathsf{B}_2 \subseteq \mathsf{Y}_2$*,* $\mathsf{preimage}\ \langle\mathsf{g}_1,\mathsf{g}_2\rangle_{\mathsf{map}}\ (\mathsf{B}_1 \times \mathsf{B}_2) = (\mathsf{preimage}\ \mathsf{g}_1\ \mathsf{B}_1) \cap (\mathsf{preimage}\ \mathsf{g}_2\ \mathsf{B}_2)$*.*

Theorem 5.23 (pre distributes over $\langle\cdot,\cdot\rangle_{\mathsf{map}}$)**.** *Let* $\mathsf{g}_1 \in \mathsf{X} \rightharpoonup \mathsf{Y}_1$ *and* $\mathsf{g}_2 \in \mathsf{X} \rightharpoonup \mathsf{Y}_2$*. Then* $\mathsf{pre}\ \langle\mathsf{g}_1,\mathsf{g}_2\rangle_{\mathsf{map}} \equiv \langle\mathsf{pre}\ \mathsf{g}_1, \mathsf{pre}\ \mathsf{g}_2\rangle_{\mathsf{pre}}$*.*

Proof. Let $\langle\mathsf{Y}'_1,\mathsf{p}_1\rangle := \mathsf{pre}\ \mathsf{g}_1$, $\langle\mathsf{Y}'_2,\mathsf{p}_2\rangle := \mathsf{pre}\ \mathsf{g}_2$ and $\mathsf{B} \in \mathsf{Y}_1 \times \mathsf{Y}_2$. Starting from the right side, expand definitions, apply Theorem 5.18, apply Lemma 5.22, note that a product of singletons is a singleton pair, distribute $\mathsf{preimage}$ over the union, apply Lemma 5.17, and apply Theorem 5.18 again:

$$
\begin{aligned}
&\mathsf{ap_{pre}}\ \langle\mathsf{pre}\ \mathsf{g}_1, \mathsf{pre}\ \mathsf{g}_2\rangle_{\mathsf{pre}}\ \mathsf{B} \\
&\quad\equiv \mathsf{let}\ \ \mathsf{Y}' := \mathsf{Y}'_1 \times \mathsf{Y}'_2 \\
&\qquad\qquad \mathsf{p} := \lambda\mathsf{B}. \bigcup_{\langle\mathsf{y}_1,\mathsf{y}_2\rangle\in\mathsf{B}} (\mathsf{p}_1\ \{\mathsf{y}_1\}) \cap (\mathsf{p}_2\ \{\mathsf{y}_2\}) \\
&\qquad\quad \mathsf{in}\ \ \mathsf{p}\ (\mathsf{B} \cap \mathsf{Y}') \\
&\quad\equiv \bigcup_{\langle\mathsf{y}_1,\mathsf{y}_2\rangle\in\mathsf{B}\cap(\mathsf{Y}'_1\times\mathsf{Y}'_2)} (\mathsf{p}_1\ \{\mathsf{y}_1\}) \cap (\mathsf{p}_2\ \{\mathsf{y}_2\}) \\
&\quad\equiv \bigcup_{\langle\mathsf{y}_1,\mathsf{y}_2\rangle\in\mathsf{B}\cap(\mathsf{Y}'_1\times\mathsf{Y}'_2)} (\mathsf{preimage}\ \mathsf{g}_1\ \{\mathsf{y}_1\}) \cap (\mathsf{preimage}\ \mathsf{g}_2\ \{\mathsf{y}_2\}) \\
&\quad\equiv \bigcup_{\langle\mathsf{y}_1,\mathsf{y}_2\rangle\in\mathsf{B}\cap(\mathsf{Y}'_1\times\mathsf{Y}'_2)} (\mathsf{preimage}\ \langle\mathsf{g}_1,\mathsf{g}_2\rangle_{\mathsf{map}}\ (\{\mathsf{y}_1\} \times \{\mathsf{y}_2\})) \\
&\quad\equiv \bigcup_{\langle\mathsf{y}_1,\mathsf{y}_2\rangle\in\mathsf{B}\cap(\mathsf{Y}'_1\times\mathsf{Y}'_2)} (\mathsf{preimage}\ \langle\mathsf{g}_1,\mathsf{g}_2\rangle_{\mathsf{map}}\ \{\langle\mathsf{y}_1,\mathsf{y}_2\rangle\}) \\
&\quad\equiv \mathsf{preimage}\ \langle\mathsf{g}_1,\mathsf{g}_2\rangle_{\mathsf{map}}\ (\mathsf{B} \cap (\mathsf{Y}'_1 \times \mathsf{Y}'_2)) \\
&\quad\equiv \mathsf{preimage}\ \langle\mathsf{g}_1,\mathsf{g}_2\rangle_{\mathsf{map}}\ \mathsf{B} \\
&\quad\equiv \mathsf{ap_{pre}}\ (\mathsf{pre}\ \langle\mathsf{g}_1,\mathsf{g}_2\rangle_{\mathsf{map}})\ \mathsf{B}
\end{aligned}
\tag{5.44}
$$

□

5.6.3 Disjoint Union

Like proving pre distributes over composition, the proof that it distributes over dijoint union simply lifts a lemma about $\mathsf{preimage}$ to lazy preimage mappings.

Lemma 5.24 ($\mathsf{preimage}$ distributes over $(\uplus_{\mathsf{map}})$)**.** *Let* $\mathsf{g}_1 \in \mathsf{X} \rightharpoonup \mathsf{Y}$ *and* $\mathsf{g}_2 \in \mathsf{X} \rightharpoonup \mathsf{Y}$ *have disjoint domains. For all* $\mathsf{B} \subseteq \mathsf{Y}$, $\mathsf{preimage}\ (\mathsf{g}_1 \uplus_{\mathsf{map}} \mathsf{g}_2)\ \mathsf{B} = (\mathsf{preimage}\ \mathsf{g}_1\ \mathsf{B}) \uplus (\mathsf{preimage}\ \mathsf{g}_2\ \mathsf{B})$.

Theorem 5.25 (pre distributes over $(\uplus_{\mathsf{map}})$)**.** *Let* $\mathsf{g}_1 \in \mathsf{X} \rightharpoonup \mathsf{Y}$ *and* $\mathsf{g}_2 \in \mathsf{X} \rightharpoonup \mathsf{Y}$ *have disjoint domains. Then* $\mathsf{pre}\ (\mathsf{g}_1 \uplus_{\mathsf{map}} \mathsf{g}_2) \equiv (\mathsf{pre}\ \mathsf{g}_1) \uplus_{\mathsf{pre}} (\mathsf{pre}\ \mathsf{g}_2)$.

Proof. Let $\mathsf{Y}'_1 := \mathsf{range}\ \mathsf{g}_1$, $\mathsf{Y}'_2 := \mathsf{range}\ \mathsf{g}_2$ and $\mathsf{B} \subseteq \mathsf{Y}$. Starting from the right side, expand definitions, apply Theorem 5.18, apply Lemma 5.24, appy Lemma 5.17, and apply Theorem 5.18 again:

$$\begin{aligned}
&\mathsf{ap}_{\mathsf{pre}}\ ((\mathsf{pre}\ \mathsf{g}_1) \uplus_{\mathsf{pre}} (\mathsf{pre}\ \mathsf{g}_2))\ \mathsf{B} \qquad (5.45)\\
&\quad\equiv\ \mathsf{let}\ \ \mathsf{Y}' := \mathsf{Y}'_1 \cup \mathsf{Y}'_2\\
&\qquad\qquad\quad \mathsf{h} := \lambda \mathsf{B}.\,(\mathsf{ap}_{\mathsf{pre}}\ (\mathsf{pre}\ \mathsf{g}_1)\ \mathsf{B}) \uplus (\mathsf{ap}_{\mathsf{pre}}\ (\mathsf{pre}\ \mathsf{g}_2)\ \mathsf{B})\\
&\qquad\quad\ \ \mathsf{in}\ \ \mathsf{h}\ (\mathsf{B} \cap \mathsf{Y}')\\
&\quad\equiv\ (\mathsf{ap}_{\mathsf{pre}}\ (\mathsf{pre}\ \mathsf{g}_1)\ (\mathsf{B} \cap (\mathsf{Y}'_1 \cup \mathsf{Y}'_2))) \uplus (\mathsf{ap}_{\mathsf{pre}}\ (\mathsf{pre}\ \mathsf{g}_2)\ (\mathsf{B} \cap (\mathsf{Y}'_1 \cup \mathsf{Y}'_2)))\\
&\quad\equiv\ (\mathsf{preimage}\ \mathsf{g}_1\ (\mathsf{B} \cap (\mathsf{Y}'_1 \cup \mathsf{Y}'_2))) \uplus (\mathsf{preimage}\ \mathsf{g}_2\ (\mathsf{B} \cap (\mathsf{Y}'_1 \cup \mathsf{Y}'_2)))\\
&\quad\equiv\ \mathsf{preimage}\ (\mathsf{g}_1 \uplus_{\mathsf{map}} \mathsf{g}_2)\ (\mathsf{B} \cap (\mathsf{Y}'_1 \cup \mathsf{Y}'_2))\\
&\quad\equiv\ \mathsf{preimage}\ (\mathsf{g}_1 \uplus_{\mathsf{map}} \mathsf{g}_2)\ \mathsf{B}\\
&\quad\equiv\ \mathsf{ap}_{\mathsf{pre}}\ (\mathsf{pre}\ (\mathsf{g}_1 \uplus_{\mathsf{map}} \mathsf{g}_2))\ \mathsf{B}
\end{aligned}$$

□

5.7 Deriving the Preimage Arrow

Now we can define an arrow that runs expressions backward on sets of outputs. Its computations should produce preimage mappings or be preimage mappings.

As with the mapping arrow and mappings, we cannot have $\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y} ::= \mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y}$: we run into trouble trying to define $\mathsf{arr}_{\mathsf{pre}}$ because a preimage mapping needs an observable range. To get one, it is easiest to parameterize preimage computations on a $\mathsf{Set\ X}$; therefore the ***preimage arrow*** type constructor is

$$\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}\ ::=\ \mathsf{Set\ X} \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y}) \qquad (5.46)$$

or $\mathsf{Set\ X} \Rightarrow \langle \mathsf{Set\ Y}, \mathsf{Set\ Y} \Rightarrow \mathsf{Set\ X} \rangle$. To deconstruct the type, a preimage arrow computation computes a range first, and returns the range and a lambda that computes preimages.

To use Theorem 5.8, we need to define correctness using a lift from the mapping arrow to the preimage arrow. A simple candidate with the right type is

$$\begin{aligned} \mathsf{lift_{pre}} &: (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}) \\ \mathsf{lift_{pre}\ g\ A} &:= \mathsf{pre\ (g\ A)} \end{aligned} \tag{5.47}$$

By $\mathsf{lift_{pre}}$'s definition and Theorem 5.18, for all $\mathsf{g} : \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}$, $\mathsf{A} \subseteq \mathsf{X}$ and $\mathsf{B} \subseteq \mathsf{Y}$,

$$\begin{aligned} \mathsf{ap_{pre}\ (lift_{pre}\ g\ A)\ B} &\equiv \mathsf{ap_{pre}\ (pre\ (g\ A))\ B} \\ &\equiv \mathsf{preimage\ (g\ A)\ B} \end{aligned} \tag{5.48}$$

Thus, lifted mapping arrow computations correctly compute preimages under restricted mappings, exactly as we should expect them to.

To derive the preimage arrow's combinators in a way that makes $\mathsf{lift_{pre}}$ a homomorphism, we need preimage arrow equivalence to mean "computes the same preimages."

Definition 5.26 (preimage arrow equivalence)**.** *Two computations* $\mathsf{h}_1 : \mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}$ *and* $\mathsf{h}_2 : \mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}$ *are equivalent, or* $\mathsf{h}_1 \equiv \mathsf{h}_2$*, when* $\mathsf{h}_1\ \mathsf{A} \equiv \mathsf{h}_2\ \mathsf{A}$ *for all* $\mathsf{A} \subseteq \mathsf{X}$.

As with $\mathsf{arr_{map}}$, defining $\mathsf{arr_{pre}}$ as a composition meets (5.14). The remainder of this section derives $(\&\&\&_{\mathsf{pre}})$, $(\ggg_{\mathsf{pre}})$, $\mathsf{ifte_{pre}}$ and $\mathsf{lazy_{pre}}$ from mapping arrow combinators, in a way that ensures $\mathsf{lift_{pre}}$ is an arrow homomorphism from the mapping arrow to the preimage arrow. Fig. 5.6 contains the resulting definitions.

$$
\begin{aligned}
&X \underset{\mathsf{pre}}{\rightsquigarrow} Y ::= \mathsf{Set}\ X \Rightarrow (X \underset{\mathsf{pre}}{\rightarrow} Y)\\[1ex]
&\mathsf{arr}_{\mathsf{pre}} : (X \Rightarrow Y) \Rightarrow (X \underset{\mathsf{pre}}{\rightsquigarrow} Y)\\
&\mathsf{arr}_{\mathsf{pre}}\ :=\ \mathsf{lift}_{\mathsf{pre}} \circ \mathsf{arr}_{\mathsf{map}}\\[1ex]
&(\ggg_{\mathsf{pre}}) : (X \underset{\mathsf{pre}}{\rightsquigarrow} Y) \Rightarrow (Y \underset{\mathsf{pre}}{\rightsquigarrow} Z) \Rightarrow (X \underset{\mathsf{pre}}{\rightsquigarrow} Z)\\
&(h_1 \ggg_{\mathsf{pre}} h_2)\ A\ :=\ \begin{array}[t]{l}\mathsf{let}\ \ h_1' := h_1\ A\\ \qquad\ \ h_2' := h_2\ (\mathsf{range}_{\mathsf{pre}}\ h_1')\\ \mathsf{in}\ \ h_2' \circ_{\mathsf{pre}} h_1'\end{array}\\[1ex]
&(\&\&\&_{\mathsf{pre}}) : (X \underset{\mathsf{pre}}{\rightsquigarrow} Y) \Rightarrow (X \underset{\mathsf{pre}}{\rightsquigarrow} Z) \Rightarrow (X \underset{\mathsf{pre}}{\rightsquigarrow} Y \times Z)\\
&(h_1\ \&\&\&_{\mathsf{pre}}\ h_2)\ A\ :=\ \langle h_1\ A, h_2\ A\rangle_{\mathsf{pre}}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{ifte}_{\mathsf{pre}} : (X \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Bool}) \Rightarrow (X \underset{\mathsf{pre}}{\rightsquigarrow} Y) \Rightarrow (X \underset{\mathsf{pre}}{\rightsquigarrow} Y) \Rightarrow (X \underset{\mathsf{pre}}{\rightsquigarrow} Y)\\
&\mathsf{ifte}_{\mathsf{pre}}\ h_1\ h_2\ h_3\ A\ :=\ \begin{array}[t]{l}\mathsf{let}\ \ h_1' := h_1\ A\\ \qquad\ \ h_2' := h_2\ (\mathsf{ap}_{\mathsf{pre}}\ h_1'\ \{\mathsf{true}\})\\ \qquad\ \ h_3' := h_3\ (\mathsf{ap}_{\mathsf{pre}}\ h_1'\ \{\mathsf{false}\})\\ \mathsf{in}\ \ h_2' \uplus_{\mathsf{pre}} h_3'\end{array}\\[1ex]
&\mathsf{lazy}_{\mathsf{pre}} : (1 \Rightarrow (X \underset{\mathsf{pre}}{\rightsquigarrow} Y)) \Rightarrow (X \underset{\mathsf{pre}}{\rightsquigarrow} Y)\\
&\mathsf{lazy}_{\mathsf{pre}}\ h\ A\ :=\ \mathsf{if}\ (A = \varnothing)\ (\mathsf{pre}\ \varnothing)\ (h\ 0\ A)
\end{aligned}
$$

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

Figure 5.6: Preimage arrow definitions.

5.7.1 Pairing

Starting with the left side of (5.16), we expand definitions, apply Theorem 5.23, and rewrite in terms of $\mathsf{lift}_{\mathsf{pre}}$:

$$
\begin{aligned}
\mathsf{ap}_{\mathsf{pre}}\ (\mathsf{lift}_{\mathsf{pre}}\ (g_1\ \&\&\&_{\mathsf{map}}\ g_2)\ A)\ B\ &\equiv\ \mathsf{ap}_{\mathsf{pre}}\ (\mathsf{pre}\ \langle g_1\ A, g_2\ A\rangle_{\mathsf{map}})\ B\\
&\equiv\ \mathsf{ap}_{\mathsf{pre}}\ \langle \mathsf{pre}\ (g_1\ A), \mathsf{pre}\ (g_2\ A)\rangle_{\mathsf{pre}}\ B\\
&\equiv\ \mathsf{ap}_{\mathsf{pre}}\ \langle \mathsf{lift}_{\mathsf{pre}}\ g_1\ A, \mathsf{lift}_{\mathsf{pre}}\ g_2\ A\rangle_{\mathsf{pre}}\ B
\end{aligned}
\tag{5.49}
$$

Substituting h_1 for $\mathsf{lift}_{\mathsf{pre}}\ g_1$ and h_2 for $\mathsf{lift}_{\mathsf{pre}}\ g_2$, and removing the application of $\mathsf{ap}_{\mathsf{pre}}$ from both sides of the equivalence gives a definition of $(\&\&\&_{\mathsf{pre}})$ (Fig. 5.6) for which (5.16) holds.

5.7.2 Composition

Starting with the left side of (5.15), we expand definitions, apply Theorem 5.21 and rewrite in terms of $\mathsf{lift}_{\mathsf{pre}}$:

$$
\mathsf{ap}_{\mathsf{pre}}\ (\mathsf{lift}_{\mathsf{pre}}\ (g_1 \ggg_{\mathsf{map}} g_2)\ A)\ C\ \equiv\ \begin{array}[t]{l}\mathsf{let}\ \ g_1' := g_1\ A\\ \qquad\ \ g_2' := g_2\ (\mathsf{range}\ g_1')\\ \mathsf{in}\ \ \mathsf{ap}_{\mathsf{pre}}\ (\mathsf{pre}\ (g_2' \circ_{\mathsf{map}} g_1'))\ C\end{array}
\tag{5.50}
$$

$$\begin{aligned}
&\equiv \text{let } g_1' := g_1\ A \\
&\qquad\quad g_2' := g_2\ (\text{range } g_1') \\
&\qquad \text{in } \text{ap}_{\text{pre}}\ ((\text{pre } g_1') \circ_{\text{pre}} (\text{pre } g_2'))\ C \\
&\equiv \text{let } h_1 := \text{lift}_{\text{pre}}\ g_1\ A \\
&\qquad\quad h_2 := \text{lift}_{\text{pre}}\ g_2\ (\text{range}_{\text{pre}}\ h_1) \\
&\qquad \text{in } \text{ap}_{\text{pre}}\ (h_2 \circ_{\text{pre}} h_1)\ C
\end{aligned}$$

Substituting h_1 for $\text{lift}_{\text{pre}}\ g_1$ and h_2 for $\text{lift}_{\text{pre}}\ g_2$, and removing the application of ap_{pre} from both sides of the equivalence gives a definition of $(\ggg_{\text{pre}})$ (Fig. 5.6) for which (5.15) holds.

5.7.3 Conditional

Starting with the left side of (5.17), we expand terms, apply Theorem 5.25, rewrite in terms of lift_{pre}, and apply Theorem 5.18 in h_2 and h_3:

$$\begin{aligned}
\text{ap}_{\text{pre}}\ (\text{lift}_{\text{pre}}\ (\text{ifte}_{\text{map}}\ g_1\ g_2\ g_3)\ A)\ B &\equiv \text{let } g_1' := g_1\ A \\
&\qquad\quad g_2' := g_2\ (\text{preimage } g_1'\ \{\text{true}\}) \\
&\qquad\quad g_3' := g_3\ (\text{preimage } g_1'\ \{\text{false}\}) \\
&\qquad \text{in } \text{ap}_{\text{pre}}\ (\text{pre } (g_2' \uplus_{\text{map}} g_3'))\ B \\
&\equiv \text{let } g_1' := g_1\ A \\
&\qquad\quad g_2' := g_2\ (\text{preimage } g_1'\ \{\text{true}\}) \\
&\qquad\quad g_3' := g_3\ (\text{preimage } g_1'\ \{\text{false}\}) \\
&\qquad \text{in } \text{ap}_{\text{pre}}\ ((\text{pre } g_2') \uplus_{\text{pre}} (\text{pre } g_3'))\ B \\
&\equiv \text{let } h_1 := \text{lift}_{\text{pre}}\ g_1\ A \\
&\qquad\quad h_2 := \text{lift}_{\text{pre}}\ g_2\ (\text{ap}_{\text{pre}}\ h_1\ \{\text{true}\}) \\
&\qquad\quad h_3 := \text{lift}_{\text{pre}}\ g_3\ (\text{ap}_{\text{pre}}\ h_1\ \{\text{false}\}) \\
&\qquad \text{in } \text{ap}_{\text{pre}}\ (h_2 \uplus_{\text{pre}} h_3)\ B
\end{aligned} \tag{5.51}$$

Substituting h_1 for $\text{lift}_{\text{pre}}\ g_1$, h_2 for $\text{lift}_{\text{pre}}\ g_2$ and h_3 for $\text{lift}_{\text{pre}}\ g_3$, and removing the application of ap_{pre} from both sides of the equivalence gives a definition of ifte_{pre} (Fig. 5.6) for which (5.17) holds.

5.7.4 Laziness

Starting with the left side of (5.18), expand definitions, distribute pre over the branches of if, and rewrite in terms of $\mathsf{lift_{pre}}\ (\mathsf{g}\ 0)$:

$$\begin{aligned}
\mathsf{ap_{pre}}\ (\mathsf{lift_{pre}}\ (\mathsf{lazy_{map}}\ \mathsf{g})\ \mathsf{A})\ \mathsf{B} &\equiv \mathsf{let}\ \ \mathsf{g}' := \mathsf{if}\ (\mathsf{A} = \varnothing)\ \varnothing\ (\mathsf{g}\ 0\ \mathsf{A}) \\
&\qquad \mathsf{in}\ \ \mathsf{ap_{pre}}\ (\mathsf{pre}\ \mathsf{g}')\ \mathsf{B} \\
&\equiv \mathsf{let}\ \ \mathsf{h} := \mathsf{if}\ (\mathsf{A} = \varnothing)\ (\mathsf{pre}\ \varnothing)\ (\mathsf{pre}\ (\mathsf{g}\ 0\ \mathsf{A})) \\
&\qquad \mathsf{in}\ \ \mathsf{ap_{pre}}\ \mathsf{h}\ \mathsf{B} \\
&\equiv \mathsf{let}\ \ \mathsf{h} := \mathsf{if}\ (\mathsf{A} = \varnothing)\ (\mathsf{pre}\ \varnothing)\ (\mathsf{lift_{pre}}\ (\mathsf{g}\ 0)\ \mathsf{A}) \\
&\qquad \mathsf{in}\ \ \mathsf{ap_{pre}}\ \mathsf{h}\ \mathsf{B}
\end{aligned} \tag{5.52}$$

Substituting $\mathsf{h}\ 0$ for $\mathsf{lift_{pre}}\ (\mathsf{g}\ 0)$ and removing the application of $\mathsf{ap_{pre}}$ from both sides of the equivalence gives a definition for $\mathsf{lazy_{pre}}$ (Fig. 5.6) for which (5.18) holds.

5.7.5 Correctness

Theorem 5.27 (preimage arrow correctness)**.** $\mathsf{lift_{pre}}$ *is a homomorphism.*

Proof. By construction. □

Corollary 5.28 (semantic correctness)**.** *For all e, $[\![e]\!]_{\mathsf{pre}} \equiv \mathsf{lift_{pre}}\ [\![e]\!]_{\mathsf{map}}$.*

As with the mapping arrow, preimage arrow computations can be unruly. We would like to assume that each $\mathsf{h} : \mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}$ acts as if it computes preimages under restricted mappings. The following equivalent property is easier to state, and makes proving the arrow laws simple.

Definition 5.29 (preimage arrow law)**.** *Let* $\mathsf{h} : \mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}$*. If there exists a* $\mathsf{g} : \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}$ *such that* $\mathsf{h} \equiv \mathsf{lift_{pre}}\ \mathsf{g}$*, then* h *obeys the* ***preimage arrow law****.*

By homomorphism of $\mathsf{lift_{pre}}$, preimage arrow combinators preserve this law. It is therefore safe to assume that the preimage arrow law holds for all $\mathsf{h} : \mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}$.

Theorem 5.30. $\mathsf{lift_{pre}}$ *is an arrow epimorphism.*

Proof. Follows from Theorem 5.27 and restriction of $\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}$ to instances for which the preimage arrow law (Definition 5.29) holds. $\square$

Corollary 5.31. $\mathsf{arr}_{\mathsf{pre}}$*,* $(\&\&\&_{\mathsf{pre}})$*,* $(\ggg_{\mathsf{pre}})$*,* $\mathsf{ifte}_{\mathsf{pre}}$ *and* $\mathsf{lazy}_{\mathsf{pre}}$ *define an arrow.*

5.8 Preimages Under Partial, Probabilistic Functions

We have defined everything on the top of our roadmap:

$$\begin{tikzcd}
\mathsf{X} \rightsquigarrow_{\perp} \mathsf{Y} \arrow[r, "\mathsf{lift}_{\mathsf{map}}"] \arrow[d, swap, "\eta_{\perp *}"] & \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y} \arrow[r, "\mathsf{lift}_{\mathsf{pre}}"] \arrow[d, "\eta_{\mathsf{map}*}"] & \mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y} \arrow[d, "\eta_{\mathsf{pre}*}"] \\
\mathsf{X} \rightsquigarrow_{\perp^*} \mathsf{Y} \arrow[r, swap, "\mathsf{lift}_{\mathsf{map}*}"] & \mathsf{X} \underset{\mathsf{map}^*}{\rightsquigarrow} \mathsf{Y} \arrow[r, swap, "\mathsf{lift}_{\mathsf{pre}*}"] & \mathsf{X} \underset{\mathsf{pre}^*}{\rightsquigarrow} \mathsf{Y}
\end{tikzcd} \tag{5.53}$$

and proved that $\mathsf{lift}_{\mathsf{map}}$ and $\mathsf{lift}_{\mathsf{pre}}$ are homomorphisms. Now we move down from all three top arrows simultaneously, and prove every morphism in (5.53) is an arrow homomorphism.

5.8.1 Motivation

Probabilistic functions that may not terminate, but do so with probability 1, are common. For example, suppose random retrieves numbers in $[0, 1]$ from an implicit random source. The following probabilistic function defines the well-known geometric distribution by counting the number of times $\mathsf{random} < \mathsf{p}$:

$$\mathsf{geometric\ p} \ := \ \mathsf{if\ (random} < \mathsf{p)}\ 0\ (1 + \mathsf{geometric\ p}) \tag{5.54}$$

For any $\mathsf{p} > 0$, $\mathsf{geometric\ p}$ may not terminate, but the probability of never taking the "else" branch is $(1 - \mathsf{p}) \cdot (1 - \mathsf{p}) \cdot (1 - \mathsf{p}) \cdot \cdots = 0$. Thus, $\mathsf{geometric\ p}$ terminates with probability 1.

Suppose we interpret $\mathsf{geometric\ p}$ as $\mathsf{h} : \mathsf{R} \underset{\mathsf{pre}}{\rightsquigarrow} \mathbb{N}$, a preimage arrow computation from random sources to naturals, and we have a probability measure $\mathsf{P} : \mathsf{Set\ R} \Rightarrow [0, 1]$. The probability of $\mathsf{N} \subseteq \mathbb{N}$ is $\mathsf{P}\ (\mathsf{ap}_{\mathsf{pre}}\ (\mathsf{h\ R})\ \mathsf{N})$. To compute this, we must

- Ensure $\mathsf{ap}_{\mathsf{pre}}\ (\mathsf{h\ R})\ \mathsf{N}$ terminates.
- Ensure each $\mathsf{r} \in \mathsf{R}$ contains enough random numbers.

- Determine how random indexes numbers in r.

Ensuring $\mathsf{ap_{pre}}\ (\mathsf{h\ R})\ \mathsf{N}$ terminates is the most difficult, but doing the other two will provide structure that makes it much easier.

5.8.2 Threading and Indexing

We clearly need to transform bottom, mapping, and preimage arrows so that they thread random sources. To ensure random sources contain enough numbers, they should be infinite.

In a pure λ-calculus, random sources are typically infinite streams, threaded monadically: each computation receives and produces a random source. A little-used alternative is for the random source to be a tree, threaded applicatively: each computation receives, but does not produce, a random source. Combinators split the tree and pass subtrees to subcomputations.

With either alternative, for arrows, the resulting definitions are large, conceptually difficult, and hard to manipulate. Fortunately, it is relatively easy to assign each subcomputation a unique index into a tree-shaped random source and pass the random source unchanged. To do this, we need an indexing scheme.

Definition 5.32 (binary indexing scheme)**.** *Let* J *be an index set,* $\mathsf{j_0} \in \mathsf{J}$ *a distinguished element, and* $\mathsf{left} : \mathsf{J} \Rightarrow \mathsf{J}$ *and* $\mathsf{right} : \mathsf{J} \Rightarrow \mathsf{J}$ *be total, injective functions. If for all* $\mathsf{j} \in \mathsf{J}$, $\mathsf{j} = \mathsf{next}\ \mathsf{j_0}$ *for some finite composition* next *of* left *and* right*, then* J*,* $\mathsf{j_0}$*,* left *and* right *define a* ***binary indexing scheme****.*

For example, let J be the set of lists of $\{0, 1\}$, $\mathsf{j_0} := \langle\rangle$, and $\mathsf{left\ j} := \langle 0, \mathsf{j}\rangle$ and $\mathsf{right\ j} := \langle 1, \mathsf{j}\rangle$. Alternatively, let J be the set of dyadic rationals in $(0, 1)$ (i.e. those with power-of-two denominators), $\mathsf{j_0} := \frac{1}{2}$ and

$$\begin{aligned} \mathsf{left}\ (\mathsf{p}/\mathsf{q}) &:= (\mathsf{p} - \tfrac{1}{2})/\mathsf{q} \\ \mathsf{right}\ (\mathsf{p}/\mathsf{q}) &:= (\mathsf{p} + \tfrac{1}{2})/\mathsf{q} \end{aligned} \tag{5.55}$$

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

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

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

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

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

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

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

Figure 5.7: AStore (associative store) arrow transformer definitions.

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

In any case, J is countable, and can be thought of as a set of indexes into an infinite binary tree. Values of type $J \to A$ encode an infinite binary tree of A values as an infinite vector (i.e. total mapping).

5.8.3 Applicative, Associative Store Transformer

We thread infinite binary trees through bottom, mapping, and preimage arrow computations by defining an **arrow transformer**: a type constructor that receives and produces an arrow type, and combinators for arrows of the produced type. The applicative store arrow transformer's type constructor takes a store type s and an arrow type $x \rightsquigarrow_a y$:

$$\mathsf{AStore}\ s\ (x \rightsquigarrow_a y) \ ::= \ J \Rightarrow (\langle s, x\rangle \rightsquigarrow_a y) \tag{5.56}$$

Reading the type, we see that computations receive an index $j \in J$ and produce a computation that receives a store as well as an x. Lifting extracts the x from the input pair and sends it on to the original computation, ignoring j:

$$\begin{aligned} \eta_{\mathsf{a}^*} &: (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \Rightarrow \mathsf{AStore\ s}\ (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \\ \eta_{\mathsf{a}^*}\ \mathsf{f\ j} &:= \mathsf{arr_a\ snd} \ggg_{\mathsf{a}} \mathsf{f} \end{aligned} \tag{5.57}$$

Fig. 5.7 defines the remaining combinators. Each subcomputation receives $\mathsf{left\ j}$, $\mathsf{right\ j}$, or some other unique binary index. We thus think of programs interpreted as AStore arrows as being completely unrolled into an infinite binary tree, with each expression labeled with its tree index.

5.8.4 Partial, Probabilistic Programs

To interpret probabilistic programs, we put an infinite random tree in the store.

Definition 5.33 (random source)**.** *Let* $\mathsf{R} := \mathsf{J} \to [0,1]$*. A* ***random source*** *is any infinite binary tree* $\mathsf{r} \in \mathsf{R}$*.*

To interpret partial programs, we need to ensure termination. One ultimately implementable way is to have the store dictate which branch of each conditional, if any, is taken.

Definition 5.34 (branch trace)**.** *A* ***branch trace*** *is any* $\mathsf{t} \in \mathsf{J} \to \mathsf{Bool}_\perp$ *such that* $\mathsf{t\ j} = \mathsf{true}$ *or* $\mathsf{t\ j} = \mathsf{false}$ *for no more than finitely many* $\mathsf{j} \in \mathsf{J}$*.*

Let $\mathsf{T} \subset \mathsf{J} \to \mathsf{Bool}_\perp$ *be the largest set of branch traces.*

Let $\mathsf{X} \rightsquigarrow_{\mathsf{a}^*} \mathsf{Y} ::= \mathsf{AStore}\ (\mathsf{R} \times \mathsf{T})\ (\mathsf{X} \rightsquigarrow_{\mathsf{a}} \mathsf{Y})$ be an AStore arrow type that threads both random stores and branch traces.

For probabilistic programs, we define a combinator $\mathsf{random}_{\mathsf{a}^*}$ that returns the number at its tree index in the random source, and extend $[\![\cdot]\!]_{\mathsf{a}^*}$ for arrows a^* for which $\mathsf{random}_{\mathsf{a}^*}$ is defined:

$$
\begin{aligned}
&\mathsf{random}_{\mathsf{a}^*} : \mathsf{X} \rightsquigarrow_{\mathsf{a}^*} [0,1] \\
&\mathsf{random}_{\mathsf{a}^*}\ \mathsf{j}\ :=\ \mathsf{arr}_{\mathsf{a}}\ (\mathsf{fst} \ggg \mathsf{fst} \ggg \pi\ \mathsf{j}) \\
&[\![\mathsf{random}]\!]_{\mathsf{a}^*}\ :\equiv\ \mathsf{random}_{\mathsf{a}^*}
\end{aligned}
\tag{5.58}
$$

For partial programs, we define a combinator that reads branch traces, and an if-then-else combinator that ensures its test expression agrees with the trace:

$$
\begin{aligned}
&\mathsf{branch}_{\mathsf{a}^*} : \mathsf{X} \rightsquigarrow_{\mathsf{a}^*} \mathsf{Bool} \\
&\mathsf{branch}_{\mathsf{a}^*}\ \mathsf{j}\ :=\ \mathsf{arr}_{\mathsf{a}}\ (\mathsf{fst} \ggg \mathsf{snd} \ggg \pi\ \mathsf{j}) \\
\\
&\mathsf{ifte}^{\Downarrow}_{\mathsf{a}^*} : (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{Bool}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y}) \\
&\mathsf{ifte}^{\Downarrow}_{\mathsf{a}^*}\ \mathsf{k}_1\ \mathsf{k}_2\ \mathsf{k}_3\ \mathsf{j}\ :=\ \mathsf{ifte}_{\mathsf{a}}\ ((\mathsf{k}_1\ (\mathsf{left}\ \mathsf{j})\ \&\!\&\!\&_{\mathsf{a}}\ \mathsf{branch}_{\mathsf{a}^*}\ \mathsf{j}) \ggg_{\mathsf{a}} \mathsf{arr}_{\mathsf{a}}\ \mathsf{agrees}) \\
&\qquad\qquad\qquad\qquad\qquad (\mathsf{k}_2\ (\mathsf{left}\ (\mathsf{right}\ \mathsf{j}))) \\
&\qquad\qquad\qquad\qquad\qquad (\mathsf{k}_3\ (\mathsf{right}\ (\mathsf{right}\ \mathsf{j})))
\end{aligned}
\tag{5.59}
$$

where $\mathsf{agrees}\ \langle \mathsf{b}_1, \mathsf{b}_2 \rangle := \mathsf{if}\ (\mathsf{b}_1 = \mathsf{b}_2)\ \mathsf{b}_1\ \bot$. Thus, if the branch trace does not agree with the test expression, it returns an error. We define a new semantic function $[\![\cdot]\!]^{\Downarrow}_{\mathsf{a}^*}$ by replacing the if rule in $[\![\cdot]\!]_{\mathsf{a}^*}$:

$$
[\![\mathsf{if}\ e_c\ e_t\ e_f]\!]^{\Downarrow}_{\mathsf{a}^*}\ :\equiv\ \mathsf{ifte}^{\Downarrow}_{\mathsf{a}^*}\ [\![e_c]\!]^{\Downarrow}_{\mathsf{a}^*}\ \ [\![\mathsf{lazy}\ e_t]\!]^{\Downarrow}_{\mathsf{a}^*}\ \ [\![\mathsf{lazy}\ e_f]\!]^{\Downarrow}_{\mathsf{a}^*}
\tag{5.60}
$$

For an AStore computation k, we obviously must run k on every branch trace in T and filter out $\bot$, or somehow find inputs $\langle\langle \mathsf{r}, \mathsf{t}\rangle, \mathsf{a}\rangle$ for which agrees never returns $\bot$. Preimage AStore arrows do the former by first computing an image, and the latter by computing preimages of sets that cannot contain $\bot$.

Definition 5.35 (terminating, probabilistic arrows). *Define*

$$
\begin{aligned}
\mathsf{X} \rightsquigarrow_{\bot^*} \mathsf{Y}\ &::=\ \mathsf{AStore}\ (\mathsf{R} \times \mathsf{T})\ (\mathsf{X} \rightsquigarrow_{\bot} \mathsf{Y}) \\
\mathsf{X} \underset{\mathsf{map}^*}{\rightsquigarrow} \mathsf{Y}\ &::=\ \mathsf{AStore}\ (\mathsf{R} \times \mathsf{T})\ (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}) \\
\mathsf{X} \underset{\mathsf{pre}^*}{\rightsquigarrow} \mathsf{Y}\ &::=\ \mathsf{AStore}\ (\mathsf{R} \times \mathsf{T})\ (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y})
\end{aligned}
\tag{5.61}
$$

as the type constructors for the ***bottom****, ***mapping**** *and* ***preimage* arrows****.*

5.8.5 Correctness

We have two arrow lifts to prove homomorphic: one from pure computations to effectful (i.e. from those that do not access the store to those that do), and one from effectful computations to effectful. For both, we need AStore arrow equivalence to be more extensional.

Definition 5.36 (AStore arrow equivalence)**.** *Two* AStore *arrow computations* k_1 *and* k_2 *are equivalent, or* $\mathsf{k}_1 \equiv \mathsf{k}_2$*, when* $\mathsf{k}_1\ \mathsf{j} \equiv \mathsf{k}_2\ \mathsf{j}$ *for all* $\mathsf{j} \in \mathsf{J}$.

Pure Expressions Proving η_{a^*} is a homomorphism proves $[\![\cdot]\!]_{\mathsf{a}^*}$ correctly interprets pure expressions. Because AStore accepts any arrow type $\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}$, we can do so using only the arrow laws. From here on, we assume every AStore arrow's base type's combinators obey the arrow laws listed in Section 5.3.1.

Theorem 5.37 (pure AStore arrow correctness)**.** η_{a^*} *is a homomorphism.*

Proof. Defining $\mathsf{arr}_{\mathsf{a}^*}$ as a composition clearly meets the first homomorphism law (5.14). For homomorphism laws (5.15)–(5.17), start from the right side, expand definitions, and use arrow laws (5.20)–(5.22) to factor out $\mathsf{arr}_{\mathsf{a}}\ \mathsf{snd}$.

For (5.18), additionally β-expand within the outer thunk, then use the lazy distributive law (5.23) to extract $\mathsf{arr}_{\mathsf{a}}\ \mathsf{snd}$. □

Corollary 5.38 (pure semantic correctness)**.** *For all pure* e*,* $[\![e]\!]_{\mathsf{a}^*} \equiv \eta_{\mathsf{a}^*}\ [\![e]\!]_{\mathsf{a}}$.

Effectful Expressions To prove all interpretations of effectful expressions correct, we need a lift between AStore arrows. Let $\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y} ::= \mathsf{AStore}\ \mathsf{s}\ (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y})$ and $\mathsf{x} \rightsquigarrow_{\mathsf{b}^*} \mathsf{y} ::= \mathsf{AStore}\ \mathsf{s}\ (\mathsf{x} \rightsquigarrow_{\mathsf{b}^*} \mathsf{y})$. Define

$$\begin{aligned} &\mathsf{lift}_{\mathsf{b}^*} : (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{b}^*} \mathsf{y}) \\ &\mathsf{lift}_{\mathsf{b}^*}\ \mathsf{f}\ \mathsf{j}\ :=\ \mathsf{lift}_{\mathsf{b}}\ (\mathsf{f}\ \mathsf{j}) \end{aligned} \tag{5.62}$$

where $\mathsf{lift_b} : (\mathsf{x} \leadsto_{\mathsf{a}} \mathsf{y}) \Rightarrow (\mathsf{x} \leadsto_{\mathsf{b}} \mathsf{y})$. This shows the relationships more clearly:

$$\begin{tikzcd}
\mathsf{x} \leadsto_{\mathsf{a}} \mathsf{y} \arrow[r, "\mathsf{lift_b}"] \arrow[d, "\eta_{\mathsf{a}^*}"'] & \mathsf{x} \leadsto_{\mathsf{b}} \mathsf{y} \arrow[d, "\eta_{\mathsf{b}^*}"] \\
\mathsf{x} \leadsto_{\mathsf{a}^*} \mathsf{y} \arrow[r, "\mathsf{lift_{b^*}}"'] & \mathsf{x} \leadsto_{\mathsf{b}^*} \mathsf{y}
\end{tikzcd} \tag{5.63}$$

At minimum, we should expect to produce equivalent $\mathsf{x} \leadsto_{\mathsf{b}^*} \mathsf{y}$ computations from $\mathsf{x} \leadsto_{\mathsf{a}} \mathsf{y}$ computations whether a lift or an η is done first.

Theorem 5.39 (natural transformation)**.** *If* $\mathsf{lift_b}$ *is an arrow homomorphism, then* (5.63) *commutes.*

Proof. Expand definitions and apply homomorphism laws (5.15) and (5.14) for $\mathsf{lift_b}$:

$$\begin{aligned}
\mathsf{lift_{b^*}}\ (\eta_{\mathsf{a}^*}\ \mathsf{f}) &\equiv \lambda \mathsf{j}.\, \mathsf{lift_b}\ (\mathsf{arr_a\ snd} \ggg_{\mathsf{a}} \mathsf{f}) \\
&\equiv \lambda \mathsf{j}.\, \mathsf{lift_b}\ (\mathsf{arr_a\ snd}) \ggg_{\mathsf{b}} \mathsf{lift_b\ f} \\
&\equiv \lambda \mathsf{j}.\, \mathsf{arr_b\ snd} \ggg_{\mathsf{b}} \mathsf{lift_b\ f} \\
&\equiv \eta_{\mathsf{b}^*}\ (\mathsf{lift_b\ f})
\end{aligned} \tag{5.64}$$

□

Theorem 5.40 (effectful AStore arrow correctness)**.** *If* $\mathsf{lift_b}$ *is an arrow homomorphism from* a *to* b*, then* $\mathsf{lift_{b^*}}$ *is an arrow homomorphism from* a^* *to* b^**.*

Proof. For each homomorphism property (5.14)–(5.18), expand the definitions of $\mathsf{lift_{b^*}}$ and the combinator, distribute $\mathsf{lift_b}$, rewrite in terms of $\mathsf{lift_{b^*}}$, and rewrite using the definition of the combinator. For example, for distribution over pairing:

$$\begin{aligned}
\mathsf{lift_{b^*}}\ (\mathsf{k}_1 \mathbin{\&\&\&_{\mathsf{a}^*}} \mathsf{k}_2)\ \mathsf{j} &\equiv \mathsf{lift_b}\ ((\mathsf{k}_1 \mathbin{\&\&\&_{\mathsf{a}^*}} \mathsf{k}_2)\ \mathsf{j}) \\
&\equiv \mathsf{lift_b}\ (\mathsf{k}_1\ (\mathsf{left\ j}) \mathbin{\&\&\&_{\mathsf{a}}} \mathsf{k}_2\ (\mathsf{right\ j})) \\
&\equiv \mathsf{lift_b}\ (\mathsf{k}_1\ (\mathsf{left\ j})) \mathbin{\&\&\&_{\mathsf{b}}} \mathsf{lift_b}\ (\mathsf{k}_2\ (\mathsf{right\ j})) \\
&\equiv (\mathsf{lift_{b^*}}\ \mathsf{k}_1)\ (\mathsf{left\ j}) \mathbin{\&\&\&_{\mathsf{b}}} (\mathsf{lift_{b^*}}\ \mathsf{k}_2)\ (\mathsf{right\ j}) \\
&\equiv (\mathsf{lift_{b^*}}\ \mathsf{k}_1 \mathbin{\&\&\&_{\mathsf{b}^*}} \mathsf{lift_{b^*}}\ \mathsf{k}_2)\ \mathsf{j}
\end{aligned} \tag{5.65}$$

The remaining properties are similar, though distributing $\mathsf{lift}_{\mathsf{b}^*}$ over $\mathsf{lazy}_{\mathsf{a}^*}$ requires defining an extra thunk in the last step. □

Corollary 5.41 (effectful semantic correctness). *If* $\mathsf{lift}_{\mathsf{b}}$ *is an arrow homomorphism, then for all expressions* e, $[\![e]\!]_{\mathsf{b}^*} \equiv \mathsf{lift}_{\mathsf{b}^*}\ [\![e]\!]_{\mathsf{a}^*}$ *and* $[\![e]\!]^{\Downarrow}_{\mathsf{b}^*} \equiv \mathsf{lift}_{\mathsf{b}^*}\ [\![e]\!]^{\Downarrow}_{\mathsf{a}^*}$.

Corollary 5.42 (mapping* and preimage* arrow correctness). *The following diagram commutes:*

$$\begin{tikzcd}
\mathsf{X} \rightsquigarrow_{\bot} \mathsf{Y} \arrow[r, "\mathsf{lift}_{\mathsf{map}}"] \arrow[d, "\eta_{\bot^*}"'] & \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y} \arrow[r, "\mathsf{lift}_{\mathsf{pre}}"] \arrow[d, "\eta_{\mathsf{map}^*}"] & \mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y} \arrow[d, "\eta_{\mathsf{pre}^*}"] \\
\mathsf{X} \rightsquigarrow_{\bot^*} \mathsf{Y} \arrow[r, "\mathsf{lift}_{\mathsf{map}^*}"'] & \mathsf{X} \underset{\mathsf{map}^*}{\rightsquigarrow} \mathsf{Y} \arrow[r, "\mathsf{lift}_{\mathsf{pre}^*}"'] & \mathsf{X} \underset{\mathsf{pre}^*}{\rightsquigarrow} \mathsf{Y}
\end{tikzcd} \tag{5.66}$$

Further, $\mathsf{lift}_{\mathsf{map}^*}$ *and* $\mathsf{lift}_{\mathsf{pre}^*}$ *are arrow homomorphisms.*

Corollary 5.43 (effectful semantic correctness). *For all expressions* e,

$$\begin{aligned}
[\![e]\!]_{\mathsf{pre}^*} &\equiv \mathsf{lift}_{\mathsf{pre}^*}\ (\mathsf{lift}_{\mathsf{map}^*}\ [\![e]\!]_{\bot^*}) \\
[\![e]\!]^{\Downarrow}_{\mathsf{pre}^*} &\equiv \mathsf{lift}_{\mathsf{pre}^*}\ (\mathsf{lift}_{\mathsf{map}^*}\ [\![e]\!]^{\Downarrow}_{\bot^*})
\end{aligned} \tag{5.67}$$

5.8.6 Termination

Here, we relate $[\![e]\!]^{\Downarrow}_{\mathsf{a}^*}$ computations, which are interpreted using $\mathsf{ifte}^{\Downarrow}_{\mathsf{a}^*}$ and should always terminate, with $[\![e]\!]_{\mathsf{a}^*}$ computations, which are interpreted using $\mathsf{ifte}_{\mathsf{a}^*}$ and may not terminate. To do so, we need to find the largest domain on which $[\![e]\!]^{\Downarrow}_{\mathsf{a}^*}$ and $[\![e]\!]_{\mathsf{a}^*}$ should agree.

Definition 5.44 (maximal domain). *A computation's* ***maximal domain*** *is the largest* A^* *for which*

- *For* $\mathsf{f} : \mathsf{X} \rightsquigarrow_{\bot} \mathsf{Y}$, $\mathsf{domain}_{\bot}\ \mathsf{f}\ \mathsf{A}^* = \mathsf{A}^*$.
- *For* $\mathsf{g} : \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}$, $\mathsf{domain}\ (\mathsf{g}\ \mathsf{A}^*) = \mathsf{A}^*$.
- *For* $\mathsf{h} : \mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}$, $\mathsf{domain}_{\mathsf{pre}}\ (\mathsf{h}\ \mathsf{A}^*) = \mathsf{A}^*$.

The maximal domain of $\mathsf{k} : \mathsf{X} \rightsquigarrow_{\mathsf{a}^*} \mathsf{Y}$ *is that of* $\mathsf{k}\ \mathsf{j}_0$.

Because the above statements imply termination, A^* is a subset of the largest domain for which the computations terminate. It is not too hard to show (but is a bit tedious) that lifting computations preserves the maximal domain; e.g. the maximal domain of $\mathsf{lift}_{\mathsf{map}}\ \mathsf{f}$ is the same as f's, and the maximal domain of $\mathsf{lift}_{\mathsf{pre}^*}\ \mathsf{g}$ is the same as g's.

To ensure maximal domains exist, we need the domain operations above to have certain properties. For the mapping arrow, we must first make the intuition that computations "act as if they return restricted mappings" more precise.

Theorem 5.45 (mapping arrow restriction)**.** *Let* $\mathsf{g} : \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}$*, and* $\mathsf{A}^{\Downarrow} \subseteq \mathsf{X}$ *be the largest for which* $\mathsf{g}\ \mathsf{A}^{\Downarrow}$ *terminates. For all* $\mathsf{A} \subseteq \mathsf{A}^{\Downarrow}$*,* $\mathsf{g}\ \mathsf{A} = \mathsf{restrict}\ (\mathsf{g}\ \mathsf{A}^{\Downarrow})\ \mathsf{A}$*.*

Proof. By the mapping arrow law (Definition 5.14) there is an $\mathsf{f} : \mathsf{X} \rightsquigarrow_{\bot} \mathsf{Y}$ such that $\mathsf{g} \equiv \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}$. Thus,

$$\begin{aligned} \mathsf{restrict}\ (\mathsf{g}\ \mathsf{A}^{\Downarrow})\ \mathsf{A} &\equiv \mathsf{restrict}\ (\mathsf{lift}_{\mathsf{map}}\ \mathsf{f}\ \mathsf{A}^{\Downarrow})\ \mathsf{A} \\ &\equiv \mathsf{restrict}\ (\{\langle \mathsf{a}, \mathsf{b}\rangle \in \mathsf{mapping}\ \mathsf{f}\ \mathsf{A}^{\Downarrow} \mid \mathsf{b} \neq \bot\})\ \mathsf{A} \\ &\equiv \{\langle \mathsf{a}, \mathsf{b}\rangle \in \mathsf{mapping}\ \mathsf{f}\ \mathsf{A} \mid \mathsf{b} \neq \bot\} \\ &\equiv \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}\ \mathsf{A} \\ &\equiv \mathsf{g}\ \mathsf{A} \end{aligned} \tag{5.68}$$

□

Theorem 5.46 (domain closure operators)**.** *If* $\mathsf{f} : \mathsf{X} \rightsquigarrow_{\bot} \mathsf{Y}$, $\mathsf{g} : \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}$ *and* $\mathsf{h} : \mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}$*, then* $\mathsf{domain}_{\bot}\ \mathsf{f}$*,* $\mathsf{domain} \circ \mathsf{g}$*, and* $\mathsf{domain}_{\mathsf{pre}} \circ \mathsf{h}$ *are monotone, decreasing, and idempotent in the subdomains on which they terminate.*

Proof. These properties follow from the same properties of selection, restriction, and of preimages of images. □

Now we can relate $[\![e]\!]^{\Downarrow}_{\bot^*}$ computations to $[\![e]\!]_{\bot^*}$ computations. First, for any input for which $[\![e]\!]_{\bot^*}$ terminates, there should be a branch trace for which $[\![e]\!]^{\Downarrow}_{\bot^*}$ returns the correct output; it should otherwise return $\bot$.

Theorem 5.47. *Let* $\mathsf{f} := [\![e]\!]_{\bot^*} : \mathsf{X} \rightsquigarrow_{\bot^*} \mathsf{Y}$ *with maximal domain* A^**, and* $\mathsf{f}' := [\![e]\!]^{\Downarrow}_{\bot^*}$*. For all* $\langle\langle \mathsf{r}, \mathsf{t}\rangle, \mathsf{a}\rangle \in \mathsf{A}^*$*, there exists a* $\mathsf{T}' \subseteq \mathsf{T}$ *such that*

- *If* $\mathsf{t}' \in \mathsf{T}'$ *then* $\mathsf{f}'\ \mathsf{j}_0\ \langle\langle \mathsf{r}, \mathsf{t}'\rangle, \mathsf{a}\rangle = \mathsf{f}\ \mathsf{j}_0\ \langle\langle \mathsf{r}, \mathsf{t}\rangle, \mathsf{a}\rangle$*.*
- *If* $\mathsf{t}' \in \mathsf{T}\backslash\mathsf{T}'$ *then* $\mathsf{f}'\ \mathsf{j}_0\ \langle\langle \mathsf{r}, \mathsf{t}'\rangle, \mathsf{a}\rangle = \bot$*.*

Proof. Define T' as the set of all $\mathsf{t}' \in \mathsf{J} \to \mathsf{Bool}_\bot$ such that $\mathsf{t}'\ \mathsf{j} = \mathsf{z}$ if the subcomputation with index j is an if whose test returns z. Because $\mathsf{f}\ \mathsf{j}_0\ \langle\langle \mathsf{r}, \mathsf{t}\rangle, \mathsf{a}\rangle$ terminates, $\mathsf{t}'\ \mathsf{j} \neq \bot$ for at most finitely many j, so each $\mathsf{t}' \in \mathsf{T}$.

Let $\mathsf{t}' \in \mathsf{T}'$. Because the test of every if subcomputation at index j agrees with $\mathsf{t}'\ \mathsf{j}$ and f ignores branch traces, $\mathsf{f}'\ \mathsf{j}_0\ \langle\langle \mathsf{r}, \mathsf{t}'\rangle, \mathsf{a}\rangle = \mathsf{f}\ \mathsf{j}_0\ \langle\langle \mathsf{r}, \mathsf{t}\rangle, \mathsf{a}\rangle$.

Let $\mathsf{t}' \in \mathsf{T}\backslash\mathsf{T}'$. There exists an if subexpression with a test that does not agree with t'; therefore $\mathsf{f}'\ \mathsf{j}_0\ \langle\langle \mathsf{r}, \mathsf{t}'\rangle, \mathsf{a}\rangle = \bot$. □

Next, for any input for which $[\![e]\!]_{\bot *}$ does not terminate or returns $\bot$, $[\![e]\!]^{\Downarrow}_{\bot *}$ should return $\bot$. Proving this is a little easier if we first identify subsets of J that correspond with finite prefixes of an infinite binary tree.

Definition 5.48 (index prefix/suffix)**.** *A finite* $\mathsf{J}' \subset \mathsf{J}$ *is an* ***index prefix*** *if* $\mathsf{J}' = \{\mathsf{j}_0\}$ *or, for some index prefix* J'' *and* $\mathsf{j} \in \mathsf{J}''$, $\mathsf{J}' = \mathsf{J}'' \uplus \{\mathsf{left}\ \mathsf{j}\}$ *or* $\mathsf{J}' = \mathsf{J}'' \uplus \{\mathsf{right}\ \mathsf{j}\}$*. The corresponding* ***index suffix*** *is* $\mathsf{J}\backslash\mathsf{J}'$*.*

It is not hard to show that every index suffix is closed under left and right.

For a given $\mathsf{t} \in \mathsf{T}$, an index prefix J' serves as a convenient bounding set for the finitely many indexes j for which $\mathsf{t}\ \mathsf{j} \neq \bot$. Applying left and/or right repeatedly to any $\mathsf{j} \in \mathsf{J}'$ eventually yields a $\mathsf{j}' \in \mathsf{J}\backslash\mathsf{J}'$, for which $\mathsf{t}\ \mathsf{j}' = \bot$.

Theorem 5.49. *Let* $\mathsf{f} := [\![e]\!]_{\bot *} : \mathsf{X} \rightsquigarrow_{\bot *} \mathsf{Y}$ *with maximal domain* A^**, and* $\mathsf{f}' := [\![e]\!]^{\Downarrow}_{\bot *}$*. For all* $\mathsf{a} \in ((\mathsf{R} \times \mathsf{T}) \times \mathsf{X})\backslash\mathsf{A}^*$*,* $\mathsf{f}'\ \mathsf{j}_0\ \mathsf{a} = \bot$*.*

Proof. Let $\mathsf{t} := \mathsf{snd}\ (\mathsf{fst}\ \mathsf{a})$ be the branch trace element of a.

Suppose $\mathsf{f}\ \mathsf{j}_0\ \mathsf{a}$ terminates. If an if subcomputation's test does not agree with t, then $\mathsf{f}'\ \mathsf{j}_0\ \mathsf{a} = \bot$. If every if's test agrees, $\mathsf{f}'\ \mathsf{j}_0\ \mathsf{a} = \mathsf{f}\ \mathsf{j}_0\ \mathsf{a} = \bot$.

Suppose $\mathsf{f}\ \mathsf{j}_0\ \mathsf{a}$ does not terminate. The set of all indexes j for which $\mathsf{t}\ \mathsf{j} \neq \bot$ is contained within an index prefix J'. By hypothesis, there is an if subcomputation at some index j' such that $\mathsf{j}' \in \mathsf{J} \backslash \mathsf{J}'$. Because $\mathsf{t}\ \mathsf{j}' = \bot$, $\mathsf{f}'\ \mathsf{j}_0\ \mathsf{a} = \bot$. □

Corollary 5.50. *For all e, the maximal domain of $[\![e]\!]^{\Downarrow}_{\bot^*}$ is a subset of that of $[\![e]\!]_{\bot^*}$.*

Corollary 5.51. *Let $\mathsf{f}' := [\![e]\!]^{\Downarrow}_{\bot^*} : \mathsf{X} \rightsquigarrow_{\bot^*} \mathsf{Y}$ with maximal domain A^*, and $\mathsf{f} := [\![e]\!]_{\bot^*}$. For all $\mathsf{a} \in \mathsf{A}^*$, $\mathsf{f}'\ \mathsf{j}_0\ \mathsf{a} = \mathsf{f}\ \mathsf{j}_0\ \mathsf{a}$.*

Corollary 5.52 (correct computation everywhere)**.** *Let $[\![e]\!]^{\Downarrow}_{\bot^*} : \mathsf{X} \rightsquigarrow_{\bot^*} \mathsf{Y}$ have maximal domain A^*, and $\mathsf{X}' := (\mathsf{R} \times \mathsf{T}) \times \mathsf{X}$. For all $\mathsf{a} \in \mathsf{X}'$, $\mathsf{A} \subseteq \mathsf{X}'$ and $\mathsf{B} \subseteq \mathsf{Y}$,*

$$\begin{aligned}
[\![e]\!]^{\Downarrow}_{\bot^*}\ \mathsf{j}_0\ \mathsf{a} &= \mathsf{if}\ (\mathsf{a} \in \mathsf{A}^*)\ ([\![e]\!]_{\bot^*}\ \mathsf{j}_0\ \mathsf{a})\ \bot \\
[\![e]\!]^{\Downarrow}_{\mathsf{map}^*}\ \mathsf{j}_0\ \mathsf{A} &= [\![e]\!]_{\mathsf{map}^*}\ \mathsf{j}_0\ (\mathsf{A} \cap \mathsf{A}^*) \\
\mathsf{ap}_{\mathsf{pre}}\ ([\![e]\!]^{\Downarrow}_{\mathsf{pre}^*}\ \mathsf{j}_0\ \mathsf{A})\ \mathsf{B} &= \mathsf{ap}_{\mathsf{pre}}\ ([\![e]\!]_{\mathsf{pre}^*}\ \mathsf{j}_0\ (\mathsf{A} \cap \mathsf{A}^*))\ \mathsf{B}
\end{aligned} \tag{5.69}$$

In other words, preimages computed using $[\![\cdot]\!]^{\Downarrow}_{\mathsf{pre}^*}$ always terminate, never include inputs that give rise to errors or nontermination, and are correct.

5.9 Output Probabilities and Measurability

Typically, for $\mathsf{g} \in \mathsf{X} \rightharpoonup \mathsf{Y}$, the probability of $\mathsf{B} \subseteq \mathsf{Y}$ is $\mathsf{P}\ (\mathsf{preimage}\ \mathsf{g}\ \mathsf{B})$, where $\mathsf{P} \in \mathcal{P}\,\mathsf{X} \rightharpoonup [0,1]$ assigns probabilities to subsets of X.

However, a mapping* computation's domain is $(\mathsf{R} \times \mathsf{T}) \times \mathsf{X}$, not X. We assume each $\mathsf{r} \in \mathsf{R}$ is randomly chosen, but not each $\mathsf{t} \in \mathsf{T}$ nor each $\mathsf{x} \in \mathsf{X}$; therefore, neither T nor X should affect the probabilities of output sets. We clearly must measure *projections* of preimage sets, or $\mathsf{P}\ (\mathsf{image}\ (\mathsf{fst} \ggg \mathsf{fst})\ \mathsf{A})$ for preimage sets $\mathsf{A} \subseteq (\mathsf{R} \times \mathsf{T}) \times \mathsf{X}$.

Not all preimage sets have sensible measures. Sets that do are called **measurable**. Computing preimages and projecting them onto R must preserve measurability.

5.9.1 Measurability

From here on, we assume readers are familiar with topology or measure theory. While the remainder of this section is critical in that it establishes that the outputs of programs have sensible distributions, understanding it is not required to understand the rest of the paper. Therefore, readers unfamiliar with topology and meaure theory may skip to Section 5.10 without skipping prerequisite facts.

For readers familiar with topology but not measure theory, we review the necessary fundamentals by analogy to topology.

The analogue of a topology is a σ-algebra.

Definition 5.53 (σ-algebra, measurable set)**.** *A collection of sets $\mathcal{A} \subseteq \mathcal{P}\ \mathsf{X}$ is called a* ***σ-algebra*** *on X if it contains X and is closed under complements and countable unions. The sets in $\mathcal{A}$ are called* ***measurable sets****.*

$\mathsf{X} \backslash \mathsf{X} = \varnothing$, so $\varnothing \in \mathcal{A}$. Additionally, it follows from De Morgan's law that $\mathcal{A}$ is closed under countable intersections.

The analogue of continuity is measurability.

Definition 5.54 (measurable mapping)**.** *Let $\mathcal{A}$ and $\mathcal{B}$ be σ-algebras on X and Y. A mapping $\mathsf{g} : \mathsf{X} \rightharpoonup \mathsf{Y}$ is* ***$\mathcal{A}$-$\mathcal{B}$-measurable*** *if for all $\mathsf{B} \in \mathcal{B}$, $\mathsf{preimage\ g\ B} \in \mathcal{A}$.*

When the domain and codomain σ-algebras $\mathcal{A}$ and $\mathcal{B}$ are clear from context, we will simply say g is **measurable**.

Measurability is usually a weaker condition than continuity. For example, with respect to the σ-algebra generated from $\mathbb{R}$'s standard topology (i.e. using the standard topology as a sort of "base"), measurable $\mathbb{R} \rightharpoonup \mathbb{R}$ functions may have countably many discontinuities. Likewise, real comparison functions such as $(=)$, $(<)$, $(>)$ and their negations are measurable, but not continuous.

Product σ-algebras are defined analogously to product topologies.

Definition 5.55 (finite product σ-algebra)**.** *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be σ-algebras on X_1 and X_2, and define $\mathsf{X} := \mathsf{X}_1 \times \mathsf{X}_2$. The **product σ-algebra** $\mathcal{A}_1 \otimes \mathcal{A}_2$ is the smallest (i.e. coarsest) σ-algebra for which* $\mathsf{mapping\ fst\ X}$ *and* $\mathsf{mapping\ snd\ X}$ *are measurable.*

Definition 5.56 (arbitrary product σ-algebra)**.** *Let $\mathcal{A}$ be a σ-algebra on X. The **product σ-algebra** $\mathcal{A}^{\otimes \mathsf{J}}$ is the smallest σ-algebra for which, for all* $\mathsf{j} \in \mathsf{J}$, $\mathsf{mapping}\ (\pi\ \mathsf{j})\ (\mathsf{J} \to \mathsf{X})$ *is measurable.*

5.9.2 Measurable Pure Computations

It is easier to prove measurability of pure computations than to prove measurability of partial, probabilistic ones. Further, we can use the results to prove that the interpretations of all partial, probabilistic expressions are measurable.

We must first define what it means for a *computation* to be measurable.

Definition 5.57 (measurable mapping arrow computation)**.** *Let $\mathcal{A}$ and $\mathcal{B}$ be σ-algebras on X and Y. A computation* $\mathsf{g} : \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}$ *is $\mathcal{A}$-$\mathcal{B}$-**measurable** if* $\mathsf{g\ A}^*$ *is an $\mathcal{A}$-$\mathcal{B}$-measurable mapping, where* A^* *is* g*'s maximal domain.*

Theorem 5.58 (maximal domain measurability)**.** *Let* $\mathsf{g} : \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}$ *be an $\mathcal{A}$-$\mathcal{B}$-measurable mapping arrow computation. Its maximal domain* A^* *is in $\mathcal{A}$.*

Proof. Because $\mathsf{g\ A}^*$ is measurable, $\mathsf{preimage}\ (\mathsf{g\ A}^*)\ \mathsf{Y} = \mathsf{A}^*$ is in $\mathcal{A}$. □

Mapping arrow computations can be applied to sets other than their maximal domains. We need to ensure doing so yields a measurable mapping, at least for measurable subsets of A^*. Fortunately, that is true without any extra conditions.

Lemma 5.59. *Let* $\mathsf{g} : \mathsf{X} \rightharpoonup \mathsf{Y}$ *be an $\mathcal{A}$-$\mathcal{B}$-measurable mapping. For any* $\mathsf{A} \in \mathcal{A}$, $\mathsf{restrict\ g\ A}$ *is $\mathcal{A}$-$\mathcal{B}$-measurable.*

Theorem 5.60. *Let* $\mathsf{g} : \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}$ *be an $\mathcal{A}$-$\mathcal{B}$-measurable mapping arrow computation with maximal domain* A^**. For all* $\mathsf{A} \subseteq \mathsf{A}^*$ *with* $\mathsf{A} \in \mathcal{A}$, $\mathsf{g\ A}$ *is $\mathcal{A}$-$\mathcal{B}$-measurable.*

Proof. By Theorem 5.45 (mapping arrow restriction) and Lemma 5.59. □

We do not need to prove all interpretations using $[\![\cdot]\!]_{\mathsf{a}}$ are measurable. However, we do need to prove mapping arrow combinators preserve measurability.

Composition Proving compositions are measurable takes the most work. The main complication is that, under measurable mappings, while *preimages* of measurable sets are measurable, *images* of measurable sets may not be. We need the following four extra theorems to get around this.

Lemma 5.61 (images of preimages). *If* $\mathsf{g} : \mathsf{X} \rightharpoonup \mathsf{Y}$ *and* $\mathsf{B} \subseteq \mathsf{Y}$, $\mathsf{image\ g\ (preimage\ g\ B)} \subseteq \mathsf{B}$.

Lemma 5.62 (expanded post-composition). *Let* $\mathsf{g}_1 : \mathsf{X} \rightharpoonup \mathsf{Y}$ *and* $\mathsf{g}_2 : \mathsf{Y} \rightharpoonup \mathsf{Z}$ *such that* $\mathsf{range\ g}_1 \subseteq \mathsf{domain\ g}_2$, *and let* $\mathsf{g}_2' : \mathsf{Y} \rightharpoonup \mathsf{Z}$ *such that* $\mathsf{g}_2 \subseteq \mathsf{g}_2'$. *Then* $\mathsf{g}_2 \circ_{\mathsf{map}} \mathsf{g}_1 = \mathsf{g}_2' \circ_{\mathsf{map}} \mathsf{g}_1$.

Theorem 5.63 (mapping arrow monotonicity). *Let* $\mathsf{g} : \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}$. *For any* $\mathsf{A}' \subseteq \mathsf{A} \subseteq \mathsf{A}^*$, $\mathsf{g\ A}' \subseteq \mathsf{g\ A}$.

Proof. By Theorem 5.45 (mapping arrow restriction). □

Theorem 5.64 (maximal domain subsets). *Let* $\mathsf{g} : \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}$. *For any* $\mathsf{A} \subseteq \mathsf{A}^*$, $\mathsf{domain\ (g\ A)} = \mathsf{A}$.

Proof. Follows from Theorem 5.46. □

Now we can prove measurability.

Lemma 5.65 (($\circ_{\mathsf{map}}$) measurability). *If* $\mathsf{g}_1 : \mathsf{X} \rightharpoonup \mathsf{Y}$ *is* $\mathcal{A}$-$\mathcal{B}$*-measurable and* $\mathsf{g}_2 : \mathsf{Y} \rightharpoonup \mathsf{Z}$ *is* $\mathcal{B}$-$\mathcal{C}$*-measurable, then* $\mathsf{g}_2 \circ_{\mathsf{map}} \mathsf{g}_1$ *is* $\mathcal{A}$-$\mathcal{C}$*-measurable.*

Theorem 5.66 (($\ggg_{\mathsf{map}}$) measurability). *If* $\mathsf{g}_1 : \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}$ *is* $\mathcal{A}$-$\mathcal{B}$*-measurable and* $\mathsf{g}_2 : \mathsf{Y} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Z}$ *is* $\mathcal{B}$-$\mathcal{C}$*-measurable, then* $\mathsf{g}_1 \ggg_{\mathsf{map}} \mathsf{g}_2$ *is* $\mathcal{A}$-$\mathcal{C}$*-measurable.*

Proof. Let $\mathsf{A}^* \in \mathcal{A}$ and $\mathsf{B}^* \in \mathcal{B}$ be respectively g_1's and g_2's maximal domains. The maximal domain of $\mathsf{g}_1 \ggg_{\mathsf{map}} \mathsf{g}_2$ is $\mathsf{A}^{**} := \mathsf{preimage}\ (\mathsf{g}_1\ \mathsf{A}^*)\ \mathsf{B}^*$, which is in $\mathcal{A}$. By definition,

$$(\mathsf{g}_1 \ggg_{\mathsf{map}} \mathsf{g}_2)\ \mathsf{A}^{**} \;=\; \begin{array}{l} \mathsf{let}\ \ \mathsf{g}_1' := \mathsf{g}_1\ \mathsf{A}^{**} \\ \qquad\ \mathsf{g}_2' := \mathsf{g}_2\ (\mathsf{range}\ \mathsf{g}_1') \\ \mathsf{in}\ \ \mathsf{g}_2' \circ_{\mathsf{map}} \mathsf{g}_1' \end{array} \tag{5.70}$$

By Theorem 5.60, g_1' is an $\mathcal{A}$-$\mathcal{B}$-measurable mapping. Unfortunately, g_2' may not be $\mathcal{B}$-$\mathcal{C}$-measurable when $\mathsf{range}\ \mathsf{g}_1' \notin \mathcal{B}$.

Let $\mathsf{g}_2'' := \mathsf{g}_2\ \mathsf{B}^*$, which is a $\mathcal{B}$-$\mathcal{C}$-measurable mapping. By Lemma 5.65, $\mathsf{g}_2'' \circ_{\mathsf{map}} \mathsf{g}_1'$ is $\mathcal{A}$-$\mathcal{C}$-measurable. We need only show that $\mathsf{g}_2' \circ_{\mathsf{map}} \mathsf{g}_1' = \mathsf{g}_2'' \circ_{\mathsf{map}} \mathsf{g}_1'$, which by Lemma 5.62 is true if $\mathsf{range}\ \mathsf{g}_1' \subseteq \mathsf{domain}\ \mathsf{g}_2'$ and $\mathsf{g}_2' \subseteq \mathsf{g}_2''$.

By Theorem 5.64, $\mathsf{A}^{**} \subseteq \mathsf{A}^*$ implies $\mathsf{domain}\ \mathsf{g}_1' = \mathsf{A}^{**}$. By Theorem 5.63 and Lemma 5.61,

$$\begin{aligned} \mathsf{range}\ \mathsf{g}_1' &= \mathsf{image}\ (\mathsf{g}_1\ \mathsf{A}^{**})\ (\mathsf{preimage}\ (\mathsf{g}_1\ \mathsf{A}^*)\ \mathsf{B}^*) \\ &= \mathsf{image}\ (\mathsf{g}_1\ \mathsf{A}^{*})\ (\mathsf{preimage}\ (\mathsf{g}_1\ \mathsf{A}^*)\ \mathsf{B}^*) \\ &\subseteq \mathsf{B}^* \end{aligned} \tag{5.71}$$

$\mathsf{range}\ \mathsf{g}_1' \subseteq \mathsf{B}^*$ implies (by Theorem 5.64) that $\mathsf{domain}\ \mathsf{g}_2' = \mathsf{range}\ \mathsf{g}_1'$, and (by Theorem 5.63) that $\mathsf{g}_2' \subseteq \mathsf{g}_2''$. □

Pairing Proving pairing preserves measurability is straightforward given a corresponding theorem about mappings.

Lemma 5.67 ($\langle\cdot,\cdot\rangle_{\mathsf{map}}$ measurability)**.** *If* $\mathsf{g}_1 : \mathsf{X} \rightharpoonup \mathsf{Y}_1$ *is* $\mathcal{A}$-$\mathcal{B}_1$*-measurable and* $\mathsf{g}_2 : \mathsf{X} \rightharpoonup \mathsf{Y}_2$ *is* $\mathcal{A}$-$\mathcal{B}_2$*-measurable, then* $\langle \mathsf{g}_1, \mathsf{g}_2 \rangle_{\mathsf{map}}$ *is* $\mathcal{A}$-$(\mathcal{B}_1 \otimes \mathcal{B}_2)$*-measurable.*

Theorem 5.68 (($\&\&\&_{\mathsf{map}}$) measurability)**.** *If* $\mathsf{g}_1 : \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}_1$ *is* $\mathcal{A}$-$\mathcal{B}_1$*-measurable and* $\mathsf{g}_2 : \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}_2$ *is* $\mathcal{A}$-$\mathcal{B}_2$*-measurable, then* $\mathsf{g}_1 \ \&\&\&_{\mathsf{map}}\ \mathsf{g}_2$ *is* $\mathcal{A}$-$(\mathcal{B}_1 \otimes \mathcal{B}_2)$*-measurable.*

Proof. Let A_1^* and A_2^* be respectively g_1's and g_2's maximal domains. The maximal domain of $\mathsf{g}_1 \;\&\&\&_{\mathsf{map}}\; \mathsf{g}_2$ is $\mathsf{A}^{**} := \mathsf{A}_1^* \cap \mathsf{A}_2^*$, which is in $\mathcal{A}$. By definition, $(\mathsf{g}_1 \;\&\&\&_{\mathsf{map}}\; \mathsf{g}_2)\ \mathsf{A}^{**} = \langle \mathsf{g}_1\ \mathsf{A}^{**}, \mathsf{g}_2\ \mathsf{A}^{**}\rangle_{\mathsf{map}}$, which by Lemma 5.67 is $\mathcal{A}$-$(\mathcal{B}_1 \otimes \mathcal{B}_2)$-measurable. □

Conditional Conditionals can be proved measurable given a theorem that ensures the measurability of *finite* unions of disjoint, measurable mappings. We will need the corresponding theorem for *countable* unions further on, however.

Lemma 5.69 (union of measurable mappings)**.** *The union of a countable set of $\mathcal{A}$-$\mathcal{B}$-measurable mappings with disjoint domains is $\mathcal{A}$-$\mathcal{B}$-measurable.*

Theorem 5.70 ($\mathsf{ifte}_{\mathsf{map}}$ measurability)**.** *If* $\mathsf{g}_1 : \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Bool}$*, and* $\mathsf{g}_2 : \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}$ *and* $\mathsf{g}_3 : \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}$ *are respectively $\mathcal{A}$-$(\mathcal{P}\ \mathsf{Bool})$-measurable and $\mathcal{A}$-$\mathcal{B}$-measurable, then* $\mathsf{ifte}_{\mathsf{map}}\ \mathsf{g}_1\ \mathsf{g}_2\ \mathsf{g}_3$ *is $\mathcal{A}$-$\mathcal{B}$-measurable.*

Proof. Let $\mathcal{A}_1^*$, $\mathcal{A}_2^*$ and $\mathcal{A}_3^*$ be g_1's, g_2's and g_3's maximal domains. The maximal domain of $\mathsf{ifte}_{\mathsf{map}}\ \mathsf{g}_1\ \mathsf{g}_2\ \mathsf{g}_3$ is A^{**}, defined by

$$\begin{aligned}
\mathsf{A}_2^{**} &:= \mathsf{A}_2^* \cap \mathsf{preimage}\ (\mathsf{g}_1\ \mathcal{A}_1^*)\ \{\mathsf{true}\} \\
\mathsf{A}_3^{**} &:= \mathsf{A}_3^* \cap \mathsf{preimage}\ (\mathsf{g}_1\ \mathcal{A}_1^*)\ \{\mathsf{false}\} \\
\mathsf{A}^{**} &:= \mathsf{A}_2^{**} \uplus \mathsf{A}_3^{**}
\end{aligned} \tag{5.72}$$

Because $\mathsf{preimage}\ (\mathsf{g}_1\ \mathcal{A}_1^*)\ \mathsf{B} \in \mathcal{A}$ for any $\mathsf{B} \subseteq \mathsf{Bool}$, $\mathsf{A}^{**} \in \mathcal{A}$. By definition,

$$\begin{aligned}
\mathsf{ifte}_{\mathsf{map}}\ \mathsf{g}_1\ \mathsf{g}_2\ \mathsf{g}_3\ \mathsf{A}^{**} = \ &\mathsf{let}\ \ \mathsf{g}_1' := \mathsf{g}_1\ \mathsf{A}^{**} \\
&\phantom{\mathsf{let}}\ \ \mathsf{g}_2' := \mathsf{g}_2\ (\mathsf{preimage}\ \mathsf{g}_1'\ \{\mathsf{true}\}) \\
&\phantom{\mathsf{let}}\ \ \mathsf{g}_3' := \mathsf{g}_3\ (\mathsf{preimage}\ \mathsf{g}_1'\ \{\mathsf{false}\}) \\
&\mathsf{in}\ \ \mathsf{g}_2' \uplus_{\mathsf{map}} \mathsf{g}_3'
\end{aligned} \tag{5.73}$$

By hypothesis, g_1', g_2' and g_3' are measurable mappings. By Theorem 5.45 (mapping arrow restriction), g_2' and g_3' have disjoint domains. Apply Lemma 5.69. □

Laziness We must first prove measurability of an often-ignored corner case.

Theorem 5.71 (measurability of $\varnothing$)**.** *For any σ-algebras $\mathcal{A}$ and $\mathcal{B}$, the empty mapping $\varnothing$ is $\mathcal{A}$-$\mathcal{B}$-measurable.*

Proof. For any $\mathsf{B} \in \mathcal{B}$, $\mathsf{preimage}\ \varnothing\ \mathsf{B} = \varnothing$, and $\varnothing \in \mathcal{A}$. $\square$

Theorem 5.72 (measurability under $\mathsf{lazy_{map}}$)**.** *Let* $\mathsf{g} : 1 \Rightarrow (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y})$. *If* $\mathsf{g}\ 0$ *is $\mathcal{A}$-$\mathcal{B}$-measurable, then* $\mathsf{lazy_{map}}\ \mathsf{g}$ *is $\mathcal{A}$-$\mathcal{B}$-measurable.*

Proof. The maximal domain A^{**} of $\mathsf{lazy_{map}}\ \mathsf{g}$ is that of $\mathsf{g}\ 0$. By definition,

$$\mathsf{lazy_{map}}\ \mathsf{g}\ \mathsf{A}^{**} \ = \ \mathsf{if}\ (\mathsf{A}^{**} = \varnothing)\ \varnothing\ (\mathsf{g}\ 0\ \mathsf{A}^{**}) \tag{5.74}$$

If $\mathsf{A}^{**} = \varnothing$, then $\mathsf{lazy_{map}}\ \mathsf{g}\ \mathsf{A}^{**} = \varnothing$; apply Theorem 5.71. If $\mathsf{A}^{**} \neq \varnothing$, then $\mathsf{lazy_{map}}\ \mathsf{g} = \mathsf{g}\ 0$, which is $\mathcal{A}$-$\mathcal{B}$-measurable. $\square$

5.9.3 Measurable Probabilistic Computations

As with pure computations, we must first define what it means for an effectful computation to be measurable.

Definition 5.73 (measurable mapping* arrow computation)**.** *Let $\mathcal{A}$ and $\mathcal{B}$ be σ-algebras on* $(\mathsf{R}{\times}\mathsf{T}){\times}\mathsf{X}$ *and* Y. *A computation* $\mathsf{g} : \mathsf{X} \underset{\mathsf{map^*}}{\rightsquigarrow} \mathsf{Y}$ *is $\mathcal{A}$-$\mathcal{B}$-**measurable** if* $\mathsf{g}\ \mathsf{j}_0$ *is an $\mathcal{A}$-$\mathcal{B}$-measurable mapping arrow computation.*

Theorem 5.74. *If* $\mathsf{g} : \mathsf{X} \underset{\mathsf{map^*}}{\rightsquigarrow} \mathsf{Y}$ *is $\mathcal{A}$-$\mathcal{B}$-measurable, then for all* $\mathsf{j} \in \mathsf{J}$, $\mathsf{g}\ \mathsf{j}$ *is an $\mathcal{A}$-$\mathcal{B}$-measurable mapping arrow computation.*

Proof. By induction on J: if $\mathsf{g}\ \mathsf{j}$ is measurable, so are $\mathsf{g}\ (\mathsf{left}\ \mathsf{j})$ and $\mathsf{g}\ (\mathsf{right}\ \mathsf{j})$. $\square$

To make general measurability statements about computations, whether they have flat or product types, it helps to have a notion of a standard σ-algebra.

Definition 5.75 (standard σ-algebra)**.** *For a set* X *used as a type,* $\Sigma\ \mathsf{X}$ *denotes its* ***standard σ-algebra****, which must be defined under the following constraints:*

$$\Sigma\ \langle \mathsf{X}_1, \mathsf{X}_2 \rangle = \Sigma\ \mathsf{X}_1 \otimes \Sigma\ \mathsf{X}_2 \tag{5.75}$$

$$\Sigma\ (\mathsf{J} \to \mathsf{X}) = (\Sigma\ \mathsf{X})^{\otimes \mathsf{J}} \tag{5.76}$$

From here on, when no σ-algebras are given, "measurable" means "measurable with respect to standard σ-algebras."

The following definitions allow distinguishing the results of conditional expressions and any two branch traces:

$$\Sigma\ \mathsf{Bool} \ ::= \ \mathcal{P}\ \mathsf{Bool} \tag{5.77}$$

$$\Sigma\ \mathsf{T} \ ::= \ \mathcal{P}\ \mathsf{T} \tag{5.78}$$

Lemma 5.76 (measurable mapping arrow lifts)**.** $\mathsf{arr}_{\mathsf{map}}\ \mathsf{id}$*,* $\mathsf{arr}_{\mathsf{map}}\ \mathsf{fst}$ *and* $\mathsf{arr}_{\mathsf{map}}\ \mathsf{snd}$ *are measurable.* $\mathsf{arr}_{\mathsf{map}}\ (\mathsf{const}\ \mathsf{b})$ *is measurable if* $\{\mathsf{b}\}$ *is a measurable set. For all* $\mathsf{j} \in \mathsf{J}$*,* $\mathsf{arr}_{\mathsf{map}}\ (\pi\ \mathsf{j})$ *is measurable.*

Corollary 5.77 (measurable mapping* arrow lifts)**.** $\mathsf{arr}_{\mathsf{map}^*}\ \mathsf{id}$*,* $\mathsf{arr}_{\mathsf{map}^*}\ \mathsf{fst}$ *and* $\mathsf{arr}_{\mathsf{map}^*}\ \mathsf{snd}$ *are measurable.* $\mathsf{arr}_{\mathsf{map}^*}\ (\mathsf{const}\ \mathsf{b})$ *is measurable if* $\{\mathsf{b}\}$ *is a measurable set.* $\mathsf{random}_{\mathsf{map}^*}$ *and* $\mathsf{branch}_{\mathsf{map}^*}$ *are measurable.*

Theorem 5.78 (AStore combinators preserve measurability)**.** *Every* AStore *arrow combinator produces measurable mapping* computations from measurable mapping* computations.*

Proof. AStore's combinators are defined in terms of the base arrow's combinators and $\mathsf{arr}_{\mathsf{map}}\ \mathsf{fst}$ and $\mathsf{arr}_{\mathsf{map}}\ \mathsf{snd}$. □

Theorem 5.79 ($\mathsf{ifte}^{\Downarrow}_{\mathsf{map}^*}$ measurability)**.** $\mathsf{ifte}^{\Downarrow}_{\mathsf{map}^*}$ *is measurable.*

Proof. $\mathsf{branch}_{\mathsf{map}^*}$ is measurable, and $\mathsf{arr}_{\mathsf{map}}\ \mathsf{agrees}$ is measurable by (5.77). □

We can now prove all nonrecursive programs measurable by induction.

Definition 5.80 (finite expression)**.** *A **finite expression** is any expression for which no subexpression is a first-order application.*

Theorem 5.81 (all finite expressions are measurable)**.** *For all finite expressions e, $[\![e]\!]_{\mathsf{map}^*}$ is measurable.*

Proof. By structural induction and the above theorems. ☐

Now all we need to do is represent recursive programs as a net of finite expressions, and take a sort of limit.

Theorem 5.82 (approximation with finite expressions)**.** *Let $\mathsf{g} := [\![e]\!]^{\Downarrow}_{\mathsf{map}^*} : \mathsf{X} \underset{\mathsf{map}^*}{\rightsquigarrow} \mathsf{Y}$ and $\mathsf{t} \in \mathsf{T}$. Define $\mathsf{A} := (\mathsf{R} \times \{\mathsf{t}\}) \times \mathsf{X}$. There is a finite expression e' for which $[\![e']\!]_{\mathsf{map}^*}\ \mathsf{j}_0\ \mathsf{A} = \mathsf{g}\ \mathsf{j}_0\ \mathsf{A}$.*

Proof. Let the index prefix J' contain every j for which $\mathsf{t}\ \mathsf{j} \neq \bot$. To construct e', exhaustively apply first-order functions in e, but replace any $\mathsf{ifte}^{\Downarrow}_{\mathsf{map}^*}$ whose index j is not in J' with the equivalent expression $\bot$. Because e is well-defined, recurrences must be guarded by if, so this process terminates after finitely many first-order applications. ☐

Theorem 5.83 (all probabilistic expressions are measurable)**.** *For all expressions e, $[\![e]\!]^{\Downarrow}_{\mathsf{map}^*}$ is measurable.*

Proof. Let $\mathsf{g} := [\![e]\!]^{\Downarrow}_{\mathsf{map}^*}$ and $\mathsf{g}' := \mathsf{g}\ \mathsf{j}_0\ ((\mathsf{R} \times \mathsf{T}) \times \mathsf{X})$. By Corollary 5.52 (correct computation everywhere), $\mathsf{g}' = \mathsf{g}\ \mathsf{j}_0\ \mathsf{A}^*$ where A^* is g's maximal domain; thus we need only show g' is a measurable mapping.

By Theorem 5.45 (mapping arrow restriction),

$$\mathsf{g}' \ = \ \bigcup_{\mathsf{t} \in \mathsf{T}} \mathsf{g}\ \mathsf{j}_0\ ((\mathsf{R} \times \{\mathsf{t}\}) \times \mathsf{X}) \tag{5.79}$$

By Theorem 5.82 (approximation with finite expressions), for every $\mathsf{t} \in \mathsf{T}$, there is a finite expression whose interpretation agrees with g on $(\mathsf{R} \times \{\mathsf{t}\}) \times \mathsf{X}$. Therefore, by Theorem 5.81 (all finite expressions are measurable), $\mathsf{g}\ \mathsf{j}_0\ ((\mathsf{R} \times \{\mathsf{t}\}) \times \mathsf{X})$ is a measurable mapping. By

Theorem 5.45 (mapping arrow restriction), they have disjoint domains. By Lemma 5.69 (union of measurable mappings), their union is measurable. $\square$

Theorem 5.83 remains true when $[\![\cdot]\!]_{\mathsf{a}}$ is extended with any rule whose right side is measurable, including rules for real arithmetic, equality, inequality and limits. More generally, any continuous or (countably) piecewise continuous function can be made available as a language primitive, as long as its domain's and codomain's standard σ-algebras are generated from their topologies.

It is not difficult to compose $[\![\cdot]\!]_{\mathsf{a}}$ with another semantic function that defunctionalizes lambda expressions. Thus, the interpretations of all expressions in higher-order languages are measurable.

5.9.4 Measurable Projections

If $\mathsf{g} := [\![e]\!]^{\Downarrow}_{\mathsf{map}^*} : \mathsf{X} \underset{\mathsf{map}^*}{\rightsquigarrow} \mathsf{Y}$, the probability of a measurable output set $\mathsf{B} \in \Sigma\ \mathsf{Y}$ is

$$\mathsf{P}\ (\mathsf{image}\ (\mathsf{fst} \ggg \mathsf{fst})\ (\mathsf{preimage}\ (\mathsf{g}\ \mathsf{j}_0\ \mathsf{A}^*)\ \mathsf{B})) \tag{5.80}$$

Unfortunately, projections are generally not measurable. Fortunately, for interpretations of programs $[\![p]\!]^{\Downarrow}_{\mathsf{map}^*}$, for which $\mathsf{X} = \{\langle\rangle\}$, we have a special case.

Theorem 5.84 (measurable finite projections). *Let* $\mathsf{A} \in \Sigma\ \langle \mathsf{X}_1, \mathsf{X}_2 \rangle$*. If* X_2 *is at most countable and* $\Sigma\ \mathsf{X}_2 = \mathcal{P}\ \mathsf{X}_2$*, then* $\mathsf{image}\ \mathsf{fst}\ \mathsf{A} \in \mathcal{A}_1$.

Proof. Because $\Sigma\ \mathsf{X}_2 = \mathcal{P}\ \mathsf{X}_2$, A is a countable union of rectangles of the form $\mathsf{A}_1 \times \{\mathsf{a}_2\}$, where $\mathsf{A}_1 \in \Sigma\ \mathsf{X}_1$ and $\mathsf{a}_2 \in \mathsf{X}_2$. Because $\mathsf{image}\ \mathsf{fst}$ distributes over unions, $\mathsf{image}\ \mathsf{fst}\ \mathsf{A}$ is a countable union of sets in $\Sigma\ \mathsf{X}_1$. $\square$

Theorem 5.85. *Let* $\mathsf{g} : \mathsf{X} \underset{\mathsf{map}^*}{\rightsquigarrow} \mathsf{Y}$ *be measurable. If* X *is at most countable and* $\Sigma\ \mathsf{X} = \mathcal{P}\ \mathsf{X}$*, for all* $\mathsf{B} \in \Sigma\ \mathsf{Y}$*,* $\mathsf{image}\ (\mathsf{fst} \ggg \mathsf{fst})\ (\mathsf{preimage}\ (\mathsf{g}\ \mathsf{j}_0\ \mathsf{A}^*)\ \mathsf{B}) \in \Sigma\ \mathsf{R}$.

Proof. T is countable and $\Sigma\ \mathsf{T} = \mathcal{P}\ \mathsf{T}$ by (5.78); apply Theorem 5.84 twice. $\square$

In particular, for $[\![p]\!]^{\Downarrow}_{\mathsf{map}^*} : \{\langle\rangle\} \underset{\mathsf{map}^*}{\rightsquigarrow} \mathsf{Y}$, the probabilities of $\Sigma\ \mathsf{Y}$ are well-defined.

$$
\begin{aligned}
\mathsf{id_{pre}}\ \mathsf{A} &:= \langle \mathsf{A}, \lambda \mathsf{B}.\, \mathsf{B} \rangle \\
\mathsf{const_{pre}}\ \mathsf{b}\ \mathsf{A} &:= \langle \{\mathsf{b}\}, \lambda \mathsf{B}.\, \mathsf{if}\ (\mathsf{B} = \varnothing)\ \varnothing\ \mathsf{A} \rangle \\
\mathsf{fst_{pre}}\ \mathsf{A} &:= \langle \mathsf{proj_1}\ \mathsf{A}, \mathsf{unproj_1}\ \mathsf{A} \rangle \\
\mathsf{snd_{pre}}\ \mathsf{A} &:= \langle \mathsf{proj_2}\ \mathsf{A}, \mathsf{unproj_2}\ \mathsf{A} \rangle \\
\pi_{\mathsf{pre}}\ \mathsf{j}\ \mathsf{A} &:= \langle \mathsf{proj}\ \mathsf{j}\ \mathsf{A}, \mathsf{unproj}\ \mathsf{j}\ \mathsf{A} \rangle
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathsf{unproj} : \mathsf{J} \Rightarrow \mathsf{Set}\ (\mathsf{J} \to \mathsf{X}) \Rightarrow \mathsf{Set}\ \mathsf{X} \Rightarrow \mathsf{Set}\ (\mathsf{J} \to \mathsf{X}) \\
\mathsf{unproj}\ \mathsf{j}\ \mathsf{A}\ \mathsf{B} &:= \mathsf{preimage}\ (\mathsf{mapping}\ (\pi\ \mathsf{j})\ \mathsf{A})\ \mathsf{B} \\
&\equiv \mathsf{A} \cap \textstyle\prod_{\mathsf{i} \in \mathsf{J}} \mathsf{if}\ (\mathsf{j} = \mathsf{i})\ \mathsf{B}\ (\mathsf{proj}\ \mathsf{j}\ \mathsf{A})
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{proj_1} : \mathsf{Set}\ \langle \mathsf{X_1}, \mathsf{X_2} \rangle \Rightarrow \mathsf{Set}\ \mathsf{X_1} \\
&\mathsf{proj_1} := \mathsf{image}\ \mathsf{fst}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{proj_2} : \mathsf{Set}\ \langle \mathsf{X_1}, \mathsf{X_2} \rangle \Rightarrow \mathsf{Set}\ \mathsf{X_2} \\
&\mathsf{proj_2} := \mathsf{image}\ \mathsf{snd}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{unproj_1} : \mathsf{Set}\ \langle \mathsf{X_1}, \mathsf{X_2} \rangle \Rightarrow \mathsf{Set}\ \mathsf{X_1} \Rightarrow \mathsf{Set}\ \langle \mathsf{X_1}, \mathsf{X_2} \rangle \\
\mathsf{unproj_1}\ \mathsf{A}\ \mathsf{A_1} &:= \mathsf{preimage}\ (\mathsf{mapping}\ \mathsf{fst}\ \mathsf{A})\ \mathsf{A_1} \\
&\equiv \mathsf{A} \cap (\mathsf{A_1} \times \mathsf{proj_2}\ \mathsf{A})
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{unproj_2} : \mathsf{Set}\ \langle \mathsf{X_1}, \mathsf{X_2} \rangle \Rightarrow \mathsf{Set}\ \mathsf{X_2} \Rightarrow \mathsf{Set}\ \langle \mathsf{X_1}, \mathsf{X_2} \rangle \\
\mathsf{unproj_2}\ \mathsf{A}\ \mathsf{A_2} &:= \mathsf{preimage}\ (\mathsf{mapping}\ \mathsf{snd}\ \mathsf{A})\ \mathsf{A_2} \\
&\equiv \mathsf{A} \cap (\mathsf{proj_1}\ \mathsf{A} \times \mathsf{A_2})
\end{aligned}
$$

Figure 5.8: Preimage arrow lifts needed to interpret probabilistic programs.

5.10 Approximating Semantics

If we were to confine preimage computation to finite sets, we could implement the preimage arrow directly. But we would like something that works efficiently on infinite sets, even if it means approximating. We focus on a specific method: approximating product sets with covering rectangles.

5.10.1 Implementable Lifts

We would like to be able to compute preimages of uncountable sets, such as real intervals—but $\mathsf{preimage}\ (\mathsf{g}\ \mathsf{A})\ \mathsf{B}$ is uncomputable for most uncountable sets A and B no matter how cleverly they are represented. Further, because pre, $\mathsf{lift_{pre}}$ and $\mathsf{arr_{pre}}$ are ultimately defined in terms of $\mathsf{preimage}$, we cannot implement them.

Fortunately, we need only certain lifts. Fig. 5.2 (which defines $[\![\cdot]\!]_{\mathsf{a}}$) lifts id, $\mathsf{const}\ \mathsf{b}$, fst and snd. Section 5.8.4, which defines the combinators used to interpret partial, probabilistic programs, lifts $\pi\ \mathsf{j}$ and agrees. Measurable functions made available as language primitives of course must be lifted to the preimage arrow.

Fig. 5.8 gives explicit definitions for $\mathsf{arr_{pre}\ id}$, $\mathsf{arr_{pre}\ fst}$, $\mathsf{arr_{pre}\ snd}$, $\mathsf{arr_{pre}\ (const\ b)}$ and $\mathsf{arr_{pre}}\ (\pi\ \mathsf{j})$. (We will deal with agrees separately.) To implement them, we must model sets in a way that makes $\mathsf{A} = \varnothing$ is decidable, and the following representable and finitely computable:

- $\mathsf{A} \cap \mathsf{B}$, $\varnothing$, $\{\mathsf{true}\}$, $\{\mathsf{false}\}$ and $\{\mathsf{b}\}$ for every $\mathsf{const\ b}$
- $\mathsf{A}_1 \times \mathsf{A}_2$, $\mathsf{proj}_1\ \mathsf{A}$ and $\mathsf{proj}_2\ \mathsf{A}$ (5.81)
- $\mathsf{J} \to \mathsf{X}$, $\mathsf{proj\ j\ A}$ and $\mathsf{unproj\ j\ A\ B}$

We first need to define families of sets under which these operations are closed.

Definition 5.86 (rectangular family)**.** $\mathsf{Rect\ X}$ *denotes the* ***rectangular family*** *of subsets of* X*.* $\mathsf{Rect\ X}$ *must contain* $\varnothing$ *and* X*, and be closed under finite intersections. Products must satisfy the following rules:*

$$\mathsf{Rect}\ \langle \mathsf{X}_1, \mathsf{X}_2 \rangle \ = \ (\mathsf{Rect\ X}_1) \boxtimes (\mathsf{Rect\ X}_2) \tag{5.82}$$

$$\mathsf{Rect}\ (\mathsf{J} \to \mathsf{X}) \ = \ (\mathsf{Rect\ X})^{\boxtimes \mathsf{J}} \tag{5.83}$$

where the following operations lift cartesian products to sets of sets:

$$\mathcal{A}_1 \boxtimes \mathcal{A}_2 \ := \ \{\mathsf{A}_1 \times \mathsf{A}_2 \mid \mathsf{A}_1 \in \mathcal{A}_1, \mathsf{A}_2 \in \mathcal{A}_2\} \tag{5.84}$$

$$\mathcal{A}^{\boxtimes \mathsf{J}} \ := \bigcup_{\mathsf{J}' \subset \mathsf{J}\ \textit{finite}} \left\{ \textstyle\prod_{\mathsf{j} \in \mathsf{J}} \mathsf{A_j} \,\middle|\, \mathsf{A_j} \in \mathcal{A}, \mathsf{j} \in \mathsf{J}' \iff \mathsf{A_j} \subset \bigcup \mathcal{A} \right\} \tag{5.85}$$

We additionally define $\mathsf{Rect\ Bool} ::= \mathcal{P}\ \mathsf{Bool}$. It is easy to show the collection of all rectangular families is closed under products, projections, and unproj.

Further, all of the operations in (5.81) can be exactly implemented if finite sets are modeled directly, sets in ordered spaces (such as $\mathbb{R}$) are modeled by intervals, and sets in $\mathsf{Rect}\ \langle \mathsf{X}_1, \mathsf{X}_2 \rangle$ are modeled by pairs of type $\langle \mathsf{Rect\ X}_1, \mathsf{Rect\ X}_2 \rangle$. By (5.85), sets in $\mathsf{Rect}\ (\mathsf{J} \to \mathsf{X})$ have no more than finitely many projections that are proper subsets of X. They can be modeled by *finite* binary trees, where unrepresented projections are implicitly X.

The set of branch traces T is nonrectangular, but we can model T subsets by $\mathsf{J} \to \mathsf{Bool}_\perp$ rectangles, implicitly intersected with T.

Theorem 5.87 (T model). *If* $\mathsf{T'} \in \mathsf{Rect}\ (\mathsf{J} \to \mathsf{Bool}_\perp)$ *and* $\mathsf{j} \in \mathsf{J}$*, then* $\mathsf{proj\ j\ T'} = \mathsf{proj\ j\ (T' \cap T)}$*. If* $\mathsf{B} \subseteq \mathsf{Bool}_\perp$*, then* $\mathsf{unproj\ j\ (T' \cap T)\ B} = \mathsf{unproj\ j\ T'\ B} \cap \mathsf{T}$.

Proof. Subset case is by projection monotonicity. For superset, let $\mathsf{b} \in \mathsf{proj\ j\ T'}$. Define t by $\mathsf{t\ j'} = \mathsf{b}$ if $\mathsf{j'} = \mathsf{j}$; $\mathsf{t\ j'} = \perp$ if $\perp \in \mathsf{proj\ j'\ T'}$; otherwise $\mathsf{t\ j'} \in \mathsf{proj\ j'\ T'}$.

By construction, $\mathsf{t} \in \mathsf{T'}$. For no more than finitely many $\mathsf{j'} \in \mathsf{J}$, $\mathsf{t\ j'} \neq \perp$, so $\mathsf{t} \in \mathsf{T}$. Thus, there exists a $\mathsf{t} \in \mathsf{T'} \cap \mathsf{T}$ such that $\mathsf{t\ j} = \mathsf{b}$, so $\mathsf{b} \in \mathsf{proj\ j\ (T' \cap T)}$.

The statement about unproj is an easy corollary. □

5.10.2 Approximate Preimage Mapping Operations

Implementing $\mathsf{lazy_{pre}}$ (defined in Fig. 5.6) requires computing pre, but only for the empty mapping, which is trivial: $\mathsf{pre}\ \varnothing \equiv \langle \varnothing, \lambda \mathsf{B}.\, \varnothing \rangle$. Implementing the other combinators requires $(\circ_{\mathsf{pre}})$, $\langle \cdot, \cdot \rangle_{\mathsf{pre}}$ and $(\uplus_{\mathsf{pre}})$.

From the preimage mapping definitions (Fig. 5.5), we see that $\mathsf{ap_{pre}}$ is defined using $(\cap)$ and that $(\circ_{\mathsf{pre}})$ is defined using $\mathsf{ap_{pre}}$, so $(\circ_{\mathsf{pre}})$ is directly implementable. Unfortunately, we hit a snag with $\langle \cdot, \cdot \rangle_{\mathsf{pre}}$: it loops over possibly uncountably many members of B in a big union. At this point, we need to approximate.

Theorem 5.88 (pair preimage approximation). *Let* $\mathsf{g_1} \in \mathsf{X} \rightharpoonup \mathsf{Y_1}$ *and* $\mathsf{g_2} \in \mathsf{X} \rightharpoonup \mathsf{Y_2}$*. For all* $\mathsf{B} \subseteq \mathsf{Y_1} \times \mathsf{Y_2}$*,* $\mathsf{preimage}\ \langle \mathsf{g_1}, \mathsf{g_2} \rangle_{\mathsf{map}}\ \mathsf{B} \subseteq \mathsf{preimage\ g_1\ (proj_1\ B)} \cap \mathsf{preimage\ g_2\ (proj_2\ B)}$.

Proof. By monotonicity of preimages and projections, and by Lemma 5.22. □

It is not hard to use Theorem 5.88 to show that

$$\begin{aligned} &\langle \cdot, \cdot \rangle'_{\mathsf{pre}} : (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y_1}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y_2}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y_1} \times \mathsf{Y_2}) \\ &\langle \langle \mathsf{Y'_1}, \mathsf{p_1} \rangle, \langle \mathsf{Y'_2}, \mathsf{p_2} \rangle \rangle'_{\mathsf{pre}} \ := \ \langle \mathsf{Y'_1} \times \mathsf{Y'_2}, \lambda \mathsf{B}.\, \mathsf{p_1\ (proj_1\ B)} \cap \mathsf{p_2\ (proj_2\ B)} \rangle \end{aligned} \tag{5.86}$$

computes covering rectangles of preimages under pairing.

For $(\uplus_{\mathsf{pre}})$, we need an approximating replacement for $(\cup)$ under which rectangular families are closed. In other words, we need a lattice join $(\vee)$ with respect to $(\subseteq)$, with the following additional properties:

$$\begin{aligned}(\mathsf{A}_1 \times \mathsf{A}_2) \vee (\mathsf{B}_1 \times \mathsf{B}_2) &= (\mathsf{A}_1 \vee \mathsf{B}_1) \times (\mathsf{A}_2 \vee \mathsf{B}_2) \\ (\textstyle\prod_{\mathsf{j} \in \mathsf{J}} \mathsf{A}_\mathsf{j}) \vee (\prod_{\mathsf{j} \in \mathsf{J}} \mathsf{B}_\mathsf{j}) &= \textstyle\prod_{\mathsf{j} \in \mathsf{J}} \mathsf{A}_\mathsf{j} \vee \mathsf{B}_\mathsf{j}\end{aligned} \tag{5.87}$$

If for every nonproduct type X, $\mathsf{Rect}\ \mathsf{X}$ is closed under $(\vee)$, then rectangular families are clearly closed under $(\vee)$. Further, for any A and B, $\mathsf{A} \cup \mathsf{B} \subseteq \mathsf{A} \vee \mathsf{B}$.

Replacing each union in $(\uplus_{\mathsf{pre}})$ with join yields the overapproximating $(\uplus'_{\mathsf{pre}})$:

$$\begin{aligned}&(\uplus'_{\mathsf{pre}}) : (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y}) \\ &\mathsf{h}_1 \uplus'_{\mathsf{pre}} \mathsf{h}_2 := \mathsf{let}\ \ \mathsf{Y}' := \mathsf{range}_{\mathsf{pre}}\ \mathsf{h}_1 \vee \mathsf{range}_{\mathsf{pre}}\ \mathsf{h}_2 \\ &\qquad\qquad\qquad\qquad \mathsf{p} := \lambda \mathsf{B}.\, \mathsf{ap}_{\mathsf{pre}}\ \mathsf{h}_1\ \mathsf{B} \vee \mathsf{ap}_{\mathsf{pre}}\ \mathsf{h}_2\ \mathsf{B} \\ &\qquad\qquad\qquad \mathsf{in}\ \ \langle \mathsf{Y}', \mathsf{p} \rangle\end{aligned} \tag{5.88}$$

To interpret programs that may not terminate, or that terminate with probability 1, we need to approximate $\mathsf{ifte}^{\Downarrow}_{\mathsf{pre}^*}$ (5.59), which is defined in terms of agrees. Defining its approximation in terms of an approximation of agrees would not allow us to preserve the fact that expressions interpreted using $\mathsf{ifte}^{\Downarrow}_{\mathsf{pre}^*}$ always terminate. The best approximation of the preimage of Bool under agrees (as a mapping) is $\mathsf{Bool} \times \mathsf{Bool}$, which contains $\langle \mathsf{true}, \mathsf{false} \rangle$ and $\langle \mathsf{false}, \mathsf{true} \rangle$, and thus would not constrain the test to agree with the branch trace.

A lengthy (elided) sequence of substitutions to the defining expression for $\mathsf{ifte}^{\Downarrow}_{\mathsf{pre}^*}$ results in an agrees-free equivalence:

$$\begin{aligned}\mathsf{ifte}^{\Downarrow}_{\mathsf{pre}^*}\ \mathsf{k}_1\ \mathsf{k}_2\ \mathsf{k}_3\ \mathsf{j}\ \mathsf{A} \equiv \mathsf{let}\ \ \langle \mathsf{C}_\mathsf{k}, \mathsf{p}_\mathsf{k} \rangle &:= \mathsf{k}_1\ \mathsf{j}_1\ \mathsf{A} \\ \langle \mathsf{C}_\mathsf{b}, \mathsf{p}_\mathsf{b} \rangle &:= \mathsf{branch}_{\mathsf{pre}^*}\ \mathsf{j}\ \mathsf{A} \\ \mathsf{C}_2 &:= \mathsf{C}_\mathsf{k} \cap \mathsf{C}_\mathsf{b} \cap \{\mathsf{true}\} \\ \mathsf{C}_3 &:= \mathsf{C}_\mathsf{k} \cap \mathsf{C}_\mathsf{b} \cap \{\mathsf{false}\} \\ \mathsf{A}_2 &:= \mathsf{p}_\mathsf{k}\ \mathsf{C}_2 \cap \mathsf{p}_\mathsf{b}\ \mathsf{C}_2 \\ \mathsf{A}_3 &:= \mathsf{p}_\mathsf{k}\ \mathsf{C}_3 \cap \mathsf{p}_\mathsf{b}\ \mathsf{C}_3 \\ \mathsf{in}\ \ \mathsf{k}_2\ \mathsf{j}_2\ \mathsf{A}_2 \uplus_{\mathsf{pre}} &\mathsf{k}_3\ \mathsf{j}_3\ \mathsf{A}_3\end{aligned} \tag{5.89}$$

where $\mathsf{j}_1 = \mathsf{left}\ \mathsf{j}$ and so on. Unfortunately, a straightforward approximation of this would still take unnecessary branches, when A_2 or A_3 overapproximates $\varnothing$.

$\mathsf{C_b}$ is the branch trace projection at j (with $\bot$ removed). The set of indexes for which $\mathsf{C_b}$ is either $\{\mathsf{true}\}$ or $\{\mathsf{false}\}$ is finite, so it is bounded by an index prefix, outside of which branch trace projections are $\{\mathsf{true}, \mathsf{false}\}$. Therefore, if the approximating $\mathsf{ifte}^{\Downarrow\prime}_{\mathsf{pre*}}$ takes *no branches* when $\mathsf{C_b} = \{\mathsf{true}, \mathsf{false}\}$, but approximates with a finite computation, expressions interpreted using $\mathsf{ifte}^{\Downarrow\prime}_{\mathsf{pre*}}$ will always terminate.

We need an overapproximation for the non-branching case. In the exact semantics, the returned preimage mapping's range is a subset of Y, and it returns subsets of $\mathsf{A}_2 \uplus \mathsf{A}_3$. Therefore, $\mathsf{ifte}^{\Downarrow\prime}_{\mathsf{pre*}}$ may return $\langle \mathsf{Y}, \lambda \mathsf{B}.\, \mathsf{A}_2 \vee \mathsf{A}_3 \rangle$ when $\mathsf{C_b} = \{\mathsf{true}, \mathsf{false}\}$. We cannot refer to the type Y in the function definition, so we represent it using $\top$ in the approximating semantics. Implementations can model it by a singleton "universe" instance for every $\mathsf{Rect\ Y}$.

Fig. 5.9b defines the final approximating preimage arrow. This arrow, the lifts in Fig. 5.8, and the semantic function $[\![\cdot]\!]_{\mathsf{a}}$ in Fig. 5.2 define an approximating semantics for partial, probabilistic programs.

5.10.3 Correctness

From here on, $[\![\cdot]\!]^{\Downarrow\prime}_{\mathsf{pre*}}$ interprets programs as approximating preimage* arrow computations using $\mathsf{ifte}^{\Downarrow\prime}_{\mathsf{pre*}}$. The following theorems assume $\mathsf{h} := [\![e]\!]^{\Downarrow}_{\mathsf{pre*}} : \mathsf{X} \underset{\mathsf{pre*}}{\leadsto} \mathsf{Y}$ and $\mathsf{h}' := [\![e]\!]^{\Downarrow\prime}_{\mathsf{pre*}} : \mathsf{X} \underset{\mathsf{pre*}}{\leadsto}{}' \mathsf{Y}$ for some expression e.

Theorem 5.89 (sound)**.** *For all* $\mathsf{A} \in \mathsf{Rect}\ \langle\langle \mathsf{R}, \mathsf{T}\rangle, \mathsf{X}\rangle$ *and* $\mathsf{B} \in \mathsf{Rect\ Y}$, $\mathsf{ap_{pre}}\ (\mathsf{h}\ \mathsf{j}_0\ \mathsf{A})\ \mathsf{B} \subseteq \mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}'\ \mathsf{j}_0\ \mathsf{A})\ \mathsf{B}$.

Proof. By construction. □

To use structural induction on the interpretation of e, we need a theorem that allows representing it as a finite expression (Definition 5.80). Because $\mathsf{ifte}^{\Downarrow\prime}_{\mathsf{pre*}}$ does not branch when either branch could be taken, an equivalent finite expression exists for each rectangular domain subset A.

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

$$\varnothing'_{\mathsf{pre}} := \langle \varnothing, \lambda \mathsf{B}.\, \varnothing \rangle$$

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

$$\begin{aligned} &(\circ'_{\mathsf{pre}}) : (\mathsf{Y} \underset{\mathsf{pre}}{\rightharpoonup}' \mathsf{Z}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup}' \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup}' \mathsf{Z}) \\ &\langle \mathsf{Z}', \mathsf{p}_2 \rangle \circ'_{\mathsf{pre}} \mathsf{h}_1 := \langle \mathsf{Z}', \lambda \mathsf{C}.\, \mathsf{ap}'_{\mathsf{pre}}\ \mathsf{h}_1\ (\mathsf{p}_2\ \mathsf{C}) \rangle \end{aligned}$$

$$\begin{aligned} &\langle \cdot, \cdot \rangle'_{\mathsf{pre}} : (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup}' \mathsf{Y}_1) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup}' \mathsf{Y}_2) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup}' \mathsf{Y}_1 \times \mathsf{Y}_2) \\ &\langle \langle \mathsf{Y}'_1, \mathsf{p}_1 \rangle, \langle \mathsf{Y}'_2, \mathsf{p}_2 \rangle \rangle'_{\mathsf{pre}} := \\ &\quad \langle \mathsf{Y}'_1 \times \mathsf{Y}'_2, \lambda \mathsf{B}.\, \mathsf{p}_1\ (\mathsf{proj}_1\ \mathsf{B}) \cap \mathsf{p}_2\ (\mathsf{proj}_2\ \mathsf{B}) \rangle \end{aligned}$$

$$\begin{aligned} &(\uplus'_{\mathsf{pre}}) : (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup}' \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup}' \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup}' \mathsf{Y}) \\ &\langle \mathsf{Y}'_1, \mathsf{p}_1 \rangle \uplus'_{\mathsf{pre}} \langle \mathsf{Y}'_2, \mathsf{p}_2 \rangle := \\ &\quad \langle \mathsf{Y}'_1 \vee \mathsf{Y}'_2, \lambda \mathsf{B}.\, \mathsf{ap}'_{\mathsf{pre}}\ \langle \mathsf{Y}'_1, \mathsf{p}_1 \rangle\ \mathsf{B} \vee \mathsf{ap}'_{\mathsf{pre}}\ \langle \mathsf{Y}'_2, \mathsf{p}_2 \rangle\ \mathsf{B} \rangle \end{aligned}$$

(a) Definitions for preimage mappings that compute rectangular covers.

$$\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow}' \mathsf{Y} ::= \mathsf{Rect\ X} \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup}' \mathsf{Y})$$

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

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

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

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

(b) Approximating preimage arrow, defined using approximating preimage mappings.

$$\mathsf{X} \underset{\mathsf{pre}^*}{\rightsquigarrow}' \mathsf{Y} ::= \mathsf{AStore}\ (\mathsf{R} \times \mathsf{T})\ (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow}' \mathsf{Y})$$

$$\begin{aligned} &\mathsf{random}'_{\mathsf{pre}^*} : \mathsf{X} \underset{\mathsf{pre}^*}{\rightsquigarrow}' [0,1] \\ &\mathsf{random}'_{\mathsf{pre}^*}\ \mathsf{j} := \\ &\quad \mathsf{fst}_{\mathsf{pre}} \ggg'_{\mathsf{pre}} \mathsf{fst}_{\mathsf{pre}} \ggg'_{\mathsf{pre}} \pi_{\mathsf{pre}}\ \mathsf{j} \end{aligned}$$

$$\begin{aligned} &\mathsf{branch}'_{\mathsf{pre}^*} : \mathsf{X} \underset{\mathsf{pre}^*}{\rightsquigarrow}' \mathsf{Bool} \\ &\mathsf{branch}'_{\mathsf{pre}^*}\ \mathsf{j} := \\ &\quad \mathsf{fst}_{\mathsf{pre}} \ggg'_{\mathsf{pre}} \mathsf{snd}_{\mathsf{pre}} \ggg'_{\mathsf{pre}} \pi_{\mathsf{pre}}\ \mathsf{j} \end{aligned}$$

$$\mathsf{fst}'_{\mathsf{pre}^*} := \eta'_{\mathsf{pre}^*}\ \mathsf{fst}_{\mathsf{pre}};\ \cdots$$

$$\begin{aligned} &\mathsf{ifte}^{\Downarrow\prime}_{\mathsf{pre}^*} : (\mathsf{X} \underset{\mathsf{pre}^*}{\rightsquigarrow}' \mathsf{Bool}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}^*}{\rightsquigarrow}' \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}^*}{\rightsquigarrow}' \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}^*}{\rightsquigarrow}' \mathsf{Y}) \\ &\mathsf{ifte}^{\Downarrow\prime}_{\mathsf{pre}^*}\ \mathsf{k}_1\ \mathsf{k}_2\ \mathsf{k}_3\ \mathsf{j} := \\ &\quad \mathsf{let}\ \ \langle \mathsf{C}_\mathsf{k}, \mathsf{p}_\mathsf{k} \rangle := \mathsf{k}_1\ (\mathsf{left\ j})\ \mathsf{A} \\ &\qquad\quad\ \langle \mathsf{C}_\mathsf{b}, \mathsf{p}_\mathsf{b} \rangle := \mathsf{branch}_{\mathsf{pre}^*}\ \mathsf{j}\ \mathsf{A} \\ &\qquad\qquad\quad \mathsf{C}_2 := \mathsf{C}_\mathsf{k} \cap \mathsf{C}_\mathsf{b} \cap \{\mathsf{true}\} \\ &\qquad\qquad\quad \mathsf{C}_3 := \mathsf{C}_\mathsf{k} \cap \mathsf{C}_\mathsf{b} \cap \{\mathsf{false}\} \\ &\qquad\qquad\quad \mathsf{A}_2 := \mathsf{p}_\mathsf{k}\ \mathsf{C}_2 \cap \mathsf{p}_\mathsf{b}\ \mathsf{C}_2 \\ &\qquad\qquad\quad \mathsf{A}_3 := \mathsf{p}_\mathsf{k}\ \mathsf{C}_3 \cap \mathsf{p}_\mathsf{b}\ \mathsf{C}_3 \\ &\qquad \mathsf{in}\ \ \mathsf{if}\ (\mathsf{C}_\mathsf{b} = \{\mathsf{true}, \mathsf{false}\}) \\ &\qquad\quad\ \langle \top, \lambda \mathsf{B}.\, \mathsf{A}_2 \vee \mathsf{A}_3 \rangle \\ &\qquad\quad\ (\mathsf{k}_2\ (\mathsf{left\ (right\ j)})\ \mathsf{A}_2 \uplus'_{\mathsf{pre}} \mathsf{k}_3\ (\mathsf{right\ (right\ j)})\ \mathsf{A}_3) \end{aligned}$$

(c) Preimage* arrow combinators for probabilistic choice and guaranteed termination. Fig. 5.7 (AStore arrow transformer) defines η'_{pre^*}, $(\ggg'_{\mathsf{pre}^*})$, $(\&\&\&'_{\mathsf{pre}^*})$, $\mathsf{ifte}'_{\mathsf{pre}^*}$ and $\mathsf{lazy}'_{\mathsf{pre}^*}$.

Figure 5.9: Implementable arrows that approximate preimage arrows. Specific lifts such as $\mathsf{fst}_{\mathsf{pre}} := \mathsf{arr}_{\mathsf{pre}}\ \mathsf{fst}$ are computable (see Fig. 5.8), but $\mathsf{arr}'_{\mathsf{pre}}$ is not.

Theorem 5.90 (equivalent finite expression)**.** *Let* $\mathsf{A}' \in \mathsf{Rect}\ \langle \langle \mathsf{R}, \mathsf{T} \rangle, \mathsf{X} \rangle$. *There is a finite expression* e' *for which* $\mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}''\ \mathsf{j}_0\ \mathsf{A}')\ \mathsf{B} = \mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}'\ \mathsf{j}_0\ \mathsf{A}')\ \mathsf{B}$ *for all* $\mathsf{B} \in \mathsf{Rect\ Y}$*, where* $\mathsf{h}'' := [\![e']\!]^{\Downarrow\prime}_{\mathsf{pre}^*}$.

Proof. Let $\mathsf{T}' := \mathsf{proj}_2\ (\mathsf{proj}_1\ \mathsf{A}')$, and let the index prefix J' contain every j' for which $(\mathsf{proj}\ \mathsf{j}'\ \mathsf{T}')\backslash\{\bot\}$ is either $\{\mathsf{true}\}$ or $\{\mathsf{false}\}$. To construct e', exhaustively apply first-order functions in e, but replace any $\mathsf{if}\ e_1\ e_2\ e_3$ whose index is not in J' with the equivalent expression $\mathsf{if}\ e_1\ \bot\ \bot$. Because e is well-defined, recurrences must be guarded by if, so this process terminates after finitely many applications. □

Corollary 5.91 (terminating)**.** *For all* $\mathsf{A}' \in \mathsf{Rect}\ \langle\langle\mathsf{R},\mathsf{T}\rangle,\mathsf{X}\rangle$ *and* $\mathsf{B} \in \mathsf{Rect}\ \mathsf{Y}$*,* $\mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}'\ \mathsf{j}_0\ \mathsf{A}')\ \mathsf{B}$ *terminates.*

Theorem 5.92 (monotone)**.** $\mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}'\ \mathsf{j}_0\ \mathsf{A})\ \mathsf{B}$ *is monotone in both* A *and* B*.*

Proof. Lattice operators $(\cap)$ and $(\vee)$ are monotone, as is $(\times)$. Therefore, $\mathsf{id}_{\mathsf{pre}}$ and the other lifts in Fig. 5.8 are monotone, and each approximating preimage arrow combinator preserves monotonicity. Approximating preimage* arrow combinators, which are defined in terms of approximating preimage arrow combinators (Fig. 5.9b) likewise preserve monotonicity, as does η'_{pre^*}; therefore $\mathsf{id}_{\mathsf{pre}^*}$ and other lifts are monotone.

The definition of $\mathsf{ifte}^{\Downarrow\prime}_{\mathsf{pre}^*}$ can be written in terms of lattice operators and approximating preimage arrow combinators for any A for which $\mathsf{C_b} \subset \{\mathsf{true},\mathsf{false}\}$, and thus preserves monotonicity in that case. If $\mathsf{C_b} = \{\mathsf{true},\mathsf{false}\}$, which is an upper bound for $\mathsf{C_b}$, the returned value is an upper bound.

For monotonicity in A, suppose $\mathsf{A}_1 \subseteq \mathsf{A}_2$. Apply Theorem 5.90 with $\mathsf{A}' := \mathsf{A}_1$ to yield e'; clearly, it is also an equivalent finite expression for A_2. Monotonicity follows from structural induction on the interpretation of e'.

For monotonicity in B, use Theorem 5.90 with fixed $\mathsf{A}' := \mathsf{A}$. □

Theorem 5.93 (decreasing)**.** *For all* $\mathsf{A} \in \mathsf{Rect}\ \langle\langle\mathsf{R},\mathsf{T}\rangle,\mathsf{X}\rangle$ *and* $\mathsf{B} \in \mathsf{Rect}\ \mathsf{Y}$*,* $\mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}'\ \mathsf{j}_0\ \mathsf{A})\ \mathsf{B} \subseteq \mathsf{A}$*.*

Proof. Because they compute exact preimages of rectangular sets under restriction to rectangular domains, $\mathsf{id}_{\mathsf{pre}}$ and the other lifts in Fig. 5.8 are decreasing.

By definition and applying basic lattice properties,

$$\begin{aligned}
\mathsf{ap}'_{\mathsf{pre}}\ ((\mathsf{h}_1 \ggg'_{\mathsf{pre}} \mathsf{h}_2)\ \mathsf{A})\ \mathsf{B} &\equiv \mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}_1\ \mathsf{A})\ \mathsf{B}' \quad \text{for some } \mathsf{B}' \\
\mathsf{ap}'_{\mathsf{pre}}\ ((\mathsf{h}_1 \mathbin{\&\&\&'_{\mathsf{pre}}} \mathsf{h}_2)\ \mathsf{A})\ \mathsf{B} &\equiv \mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}_1\ \mathsf{A})\ (\mathsf{proj}_1\ \mathsf{B}) \cap \mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}_2\ \mathsf{A})\ (\mathsf{proj}_2\ \mathsf{B}) \\
\mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{ifte}'_{\mathsf{pre}}\ \mathsf{h}_1\ \mathsf{h}_2\ \mathsf{h}_3\ \mathsf{A})\ \mathsf{B} &\equiv \mathsf{let}\ \ \mathsf{A}_2 := \mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}_1\ \mathsf{A})\ \{\mathsf{true}\} \\
&\qquad\quad\ \mathsf{A}_3 := \mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}_1\ \mathsf{A})\ \{\mathsf{false}\} \\
&\qquad \mathsf{in}\ \ \mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}_2\ \mathsf{A}_2)\ \mathsf{B} \vee \mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}_3\ \mathsf{A}_3)\ \mathsf{B} \\
\mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{lazy}'_{\mathsf{pre}}\ \mathsf{h}\ \mathsf{A})\ \mathsf{B} &\equiv \mathsf{if}\ (\mathsf{A} = \varnothing)\ \varnothing\ (\mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}\ 0\ \mathsf{A})\ \mathsf{B})
\end{aligned} \tag{5.90}$$

Thus, approximating preimage arrow combinators return decreasing computations when given decreasing computations. This property transfers trivially to approximating preimage* arrow combinators. Use Theorem 5.90 with $\mathsf{A}' := \mathsf{A}$, and structural induction. □

5.10.4 Preimage Refinement Algorithm

Given these properties, we might try to compute preimages of B by computing preimages with respect to increasingly fine discretizations of A.

Definition 5.94 (preimage refinement algorithm). *Let* $\mathsf{B} \in \mathsf{Rect}\ \mathsf{Y}$ *and*

$$\begin{aligned}
&\mathsf{refine} : \mathsf{Rect}\ \langle\langle \mathsf{R}, \mathsf{T}\rangle, \mathsf{X}\rangle \Rightarrow \mathsf{Rect}\ \langle\langle \mathsf{R}, \mathsf{T}\rangle, \mathsf{X}\rangle \\
&\mathsf{refine}\ \mathsf{A} := \mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}'\ \mathsf{j}_0\ \mathsf{A})\ \mathsf{B}
\end{aligned} \tag{5.91}$$

Define $\mathsf{partition} : \mathsf{Rect}\ \langle\langle \mathsf{R}, \mathsf{T}\rangle, \mathsf{X}\rangle \Rightarrow \mathsf{Set}\ (\mathsf{Rect}\ \langle\langle \mathsf{R}, \mathsf{T}\rangle, \mathsf{X}\rangle)$ *to produce positive-measure, disjoint rectangles, and define*

$$\begin{aligned}
&\mathsf{refine}^* : \mathsf{Set}\ (\mathsf{Rect}\ \langle\langle \mathsf{R}, \mathsf{T}\rangle, \mathsf{X}\rangle) \Rightarrow \mathsf{Set}\ (\mathsf{Rect}\ \langle\langle \mathsf{R}, \mathsf{T}\rangle, \mathsf{X}\rangle) \\
&\mathsf{refine}^*\ \mathcal{A} := \mathsf{image}\ \mathsf{refine}\ \left(\textstyle\bigcup_{\mathsf{A} \in \mathcal{A}} \mathsf{partition}\ \mathsf{A}\right)
\end{aligned} \tag{5.92}$$

For any $\mathsf{A} \in \mathsf{Rect}\ \langle\langle \mathsf{R}, \mathsf{T}\rangle, \mathsf{X}\rangle$*, iterate* refine^* *on* $\{\mathsf{A}\}$.

Theorem 5.93 (decreasing) guarantees $\mathsf{refine}\ \mathsf{A}$ is never larger than A. Theorem 5.92 (monotone) guarantees refining a *partition* of A never does worse than refining A itself.

Theorem 5.89 (sound) guarantees the algorithm is **sound**: the preimage of B is always contained in the covering partition refine* returns.

We would like it to be **complete** in the limit, up to null sets: covering partitions' measures should converge to the true preimage measure. Unfortunately, preimage refinement appears to compute the **Jordan outer measure** of a preimage, which is not always its measure. A counterexample is the expression rational? random, where rational? returns true when its argument is rational and loops otherwise. (This is definable using a $(\leq)$ primitive.) The preimage of $\{$true$\}$ (the rational numbers) has measure 0, but its Jordan outer measure is 1.

We conjecture that a minimal requirement for preimage refinement's measures to converge is that a program must converge with probability 1. There are certainly other requirements. We leave these and proof of convergence of measures for future work.

For now, we use algorithms that depend only on soundness.

5.11 Implementations

We have four implementations: one of the exact semantics, two direct implementations of the approximating semantics, and a less direct but more efficient implementation of the approximating semantics, which we call ***Dr. Bayes***.

5.11.1 Direct Implementations

If sets are restricted to be finite, the arrows used as translation targets in the exact semantics, defined in Figs. 5.1, 5.3, 5.4, 5.5, 5.6 and 5.7, can be implemented directly in any practical λ-calculus. Computing exact preimages is very inefficient, even under the interpretations of very small programs. Still, we have found our Typed Racket [44] implementation useful for finding theorem candidates.

Given a rectangular set library, the approximating preimage arrows defined in Figs. 5.8 and 5.9b can be implemented with few changes in any practical λ-calculus. We have done so in Typed Racket and Haskell [1]. Both implementations' arrow combinator definitions are almost line-for-line transliterations from the figures. They are at `https://github.com/ntoronto/writing/tree/master/2014esop-code`.

Making the rectangular set type polymorphic seems to require the equivalent of a typeclass system. In Haskell, it also requires multi-parameter typeclasses or indexed type families [12] to associate set types with the types of their members. Using indexed type families, the only significant differences between the Haskell implementation and the approximating semantics are type contexts, `newtype` wrappers for arrow types, and using `Maybe` types as bottom arrow return types.

Typed Racket has no typeclass system on top of its type system, so the rectangular set type is monomorphic; thus, so are the arrow types. The lack of type variables in the combinator types is the only significant difference between the implementation and the approximating semantics.

5.11.2 Dr. Bayes

Our main implementation, ***Dr. Bayes***, is written in Typed Racket. It consists of the semantic function $[\![\cdot]\!]_{\mathsf{a}^*}$ from Fig. 5.2 and its extension $[\![\cdot]\!]^{\Downarrow}_{\mathsf{a}^*}$, the bottom* arrow as defined in Figs. 5.3 and 5.7, the approximating preimage and preimage* arrows as defined in Figs. 5.8 and 5.9b, and algorithms to compute approximate probabilities. We use it to test the feasibility of solving real-world problems by computing approximate preimages.

Dr. Bayes's arrows operate on a monomorphic rectangular set data type. It includes floating-point intervals to overapproximate real intervals, with which we compute approximate preimages under arithmetic and inequalities. Finding the smallest covering rectangle for images and preimages under $\mathsf{add} : \langle\mathbb{R},\mathbb{R}\rangle \Rightarrow \mathbb{R}$ and other monotone functions is fairly straightforward. For piecewise monotone functions, we distinguish cases using $\mathsf{ifte}_{\mathsf{pre}}$; e.g.

$$\begin{aligned}\mathsf{mul}_{\mathsf{pre}} \ := \ & \mathsf{ifte}_{\mathsf{pre}}\ (\mathsf{fst}_{\mathsf{pre}} \ggg_{\mathsf{pre}} \mathsf{pos?}_{\mathsf{pre}}) \\ & \quad (\mathsf{ifte}_{\mathsf{pre}}\ (\mathsf{snd}_{\mathsf{pre}} \ggg_{\mathsf{pre}} \mathsf{pos?}_{\mathsf{pre}}) \\ & \qquad \mathsf{mul}^{++}_{\mathsf{pre}} \\ & \qquad (\mathsf{ifte}_{\mathsf{pre}}\ (\mathsf{snd}_{\mathsf{pre}} \ggg_{\mathsf{pre}} \mathsf{neg?}_{\mathsf{pre}}) \\ & \qquad\quad \mathsf{mul}^{+-}_{\mathsf{pre}} \\ & \qquad\quad (\mathsf{const}_{\mathsf{pre}}\ 0))) \\ & \quad \ldots \end{aligned} \tag{5.93}$$

To support data types, the set type includes tagged rectangles; for ad-hoc polymorphism, it includes disjoint unions.

Section 5.10.4 outlines preimage refinement: a discretization algorithm that seems to converge for programs that halt with probability 1, consisting of repeatedly shrinking and repartitioning a program's domain. We do not use this algorithm directly in Dr. Bayes because it is inefficient. Good accuracy requires fine discretization, which is *exponential* in the number of discretized axes. For example, a nonrecursive program that contains only 10 uses of random would need to partition 10 axes of R, the set of random sources. Splitting each axis into only 4 disjoint intervals yields a partition of R of size $4^{10} = 1,048,576$.

Fortunately, Bayesian practitioners tend to be satisified with sampling methods, which are usually more efficient than exact methods based on enumeration.

Let $\mathsf{g} : \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}$ be the interpretation of a program as a mapping arrow computation. A Bayesian is primarily interested in the probability of $\mathsf{B}' \subseteq \mathsf{Y}$ given some condition set $\mathsf{B} \subseteq \mathsf{Y}$. This can be approximated using **rejection sampling**. If $\mathsf{A} := \mathsf{preimage}\ (\mathsf{g}\ \mathsf{X})\ \mathsf{B}$ and $\mathsf{A}' := \mathsf{preimage}\ (\mathsf{g}\ \mathsf{X})\ \mathsf{B}'$, and xs is a list of samples from any superset of A that has at least one element in A, then

$$\Pr[\mathsf{B}'|\mathsf{B}] \ \approx \ \frac{\mathsf{length}\ (\mathsf{filter}\ (\in \mathsf{A}' \cap \mathsf{A})\ \mathsf{xs})}{\mathsf{length}\ (\mathsf{filter}\ (\in \mathsf{A})\ \mathsf{xs})} \tag{5.94}$$

where "$\approx$" (rather loosely) denotes convergence as the length of xs increases. The probability that any given element of xs is in A is often extremely small, so it would clearly be best to sample only within A. While we cannot do that, we can easily sample from a partition covering A.

For a fixed number d of uses of random, n samples, and m repartitions that split each rectangle in two, enumerating and sampling from a covering partition has time complexity $O(2^{md} + n)$. Fortunately, we do not have to enumerate the rectangles in the partition: we sample them instead, and sample one value from each rectangle, which is $O(mdn)$.

We cannot directly compute $\mathsf{a} \in \mathsf{A}$ or $\mathsf{a} \in \mathsf{A}' \cap \mathsf{A}$ in (5.94), but we can use the fact that A and A' are preimages, and use the interpretation of the program as a bottom arrow computation $\mathsf{f} : \mathsf{X} \rightsquigarrow_\perp \mathsf{Y}$:

$$\begin{aligned} \mathsf{filter}\ (\in \mathsf{A})\ \mathsf{xs} &= \mathsf{filter}\ (\in \mathsf{preimage}\ (\mathsf{g}\ \mathsf{X})\ \mathsf{B})\ \mathsf{xs} \\ &= \mathsf{filter}\ (\lambda\,\mathsf{a}.\,\mathsf{g}\ \mathsf{X}\ \mathsf{a} \in \mathsf{B})\ \mathsf{xs} \\ &= \mathsf{filter}\ (\lambda\,\mathsf{a}.\,\mathsf{f}\ \mathsf{a} \in \mathsf{B})\ \mathsf{xs} \end{aligned} \tag{5.95}$$

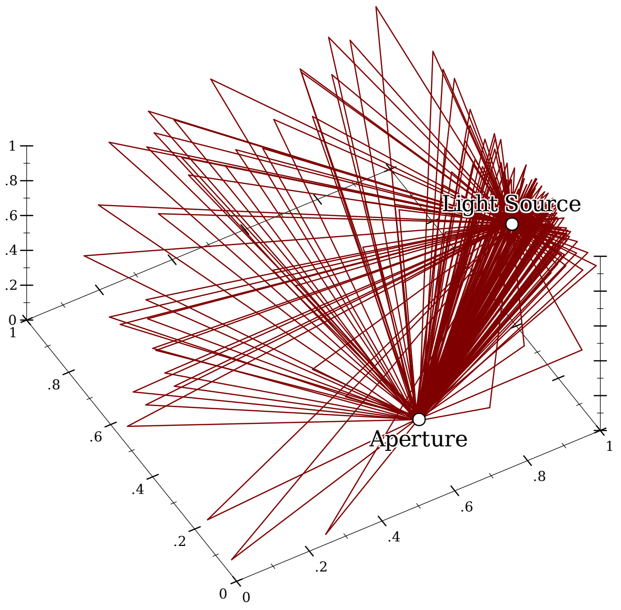
Substituting into (5.94) gives

$$\mathsf{Pr}[\mathsf{B}'|\mathsf{B}] \approx \frac{\mathsf{length}\ (\mathsf{filter}\ (\lambda\,\mathsf{a}.\,\mathsf{f}\ \mathsf{a} \in \mathsf{B}' \cap \mathsf{B})\ \mathsf{xs})}{\mathsf{length}\ (\mathsf{filter}\ (\lambda\,\mathsf{a}.\,\mathsf{f}\ \mathsf{a} \in \mathsf{B})\ \mathsf{xs})} \tag{5.96}$$

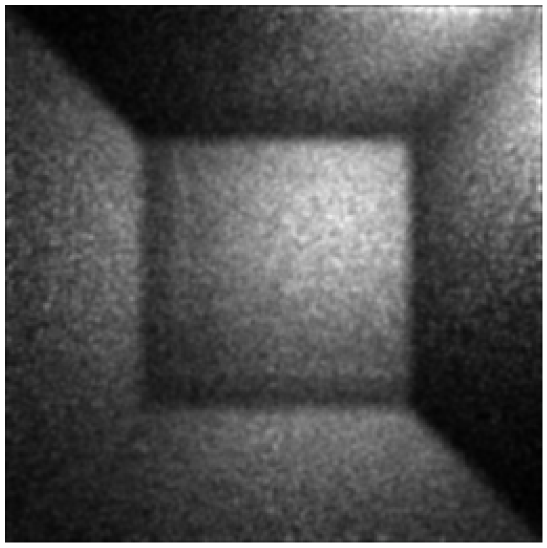
which converges to the probability of B' given B as the number of samples xs from the covering partition increases.

For simplicity, the preceeding discussion does not deal with projecting preimages from the domain of programs $(\mathsf{R} \times \mathsf{T}) \times \{\langle\rangle\}$ onto the set of random sources R. Shortly, Dr. Bayes samples rectangles from covering partitions of $(\mathsf{R} \times \mathsf{T}) \times \{\langle\rangle\}$ subsets, weights each rectangle by the inverse of the probability with which it is sampled, and projects onto R. This alogorithm is a variant of **importance sampling** [14, Section 12.4], where the candidate distribution is defined by the sampling algorithm's partitioning choices, and the target distribution is P.

Fig. 5.10 shows the result of using Dr. Bayes for stochastic ray tracing [50]. In this instance, photons are cast from a light source in a uniformly random direction and are



(a) Random paths from a single light source, conditioned on passing through an aperture.



(b) Random paths that pass through the aperture, projected onto a plane and accumulated.

```
(struct/drbayes collision (time point normal))

(define/drbayes (ray-plane-intersect p0 v n d)
  (let ([denom  (- (vec-dot v n))])
    (if (positive? denom)
        (let ([t  (/ (+ d (vec-dot p0 n)) denom)])
          (if (positive? t) (collision t (vec+ p0 (vec-scale v t)) n) #f))
        #f)))
```

(c) Part of the ray tracer implementation. Sampling involves computing approximate preimages under functions like this.

Figure 5.10: Stochastic ray tracing in Dr. Bayes is little more than physics simulation.

reflected by the walls of a square room, generating paths. The objective is to sample, with the correct distribution, only those paths that pass through an aperture. The smaller the aperture, the smaller the probability a path passes through it, and the more focused the resulting image.

All efficient implementations of stochastic ray tracing to date use sophisticated, specialized sampling methods that bear little resemblance to the physical processes they simulate. The proof-of-concept ray tracer, written in Dr. Bayes, is little more than a simple physics simulation and a conditional query.

5.12 Related Work

Any programming language research described by the words "bijective" or "reversible" might seem to have much in common with ours. Unfortunately, when we look more closely, we can usually draw only loose analogies and perhaps inspiration. An example is lenses [19], which are transformations from X to Y that can be run forward and backward, in a way that maintains some relationship between X and Y. Usually, a destructive, external process is assumed, so that, for example, a change from $\mathsf{y} \in \mathsf{Y}$ to $\mathsf{y}' \in \mathsf{Y}$ induces a corresponding change from $\mathsf{x} \in \mathsf{X}$ to some $\mathsf{x}' \in \mathsf{X}$. When transformations lose information, lenses must satisfy certain behavioral laws. In our work, no input or output is updated, and preimages are always definable regardless of non-injectivity.

Many multi-paradigm languages [18], especially constraint functional languages, bear a strong resemblance to our work. In fact, it is easy to add a fail expression to our semantics, or to transform constraints into boolean program outputs. The most obvious difference is evaluation strategy. The most important difference is that our interpretation of programs returns *distributions* of constrained outputs, rather than arbitrary single values that meet constraints.

The forward phase in computing preimages takes a subdomain and returns an over-approximation of the function's range for that subdomain. This clearly generalizes interval arithmetic [24] to all first-order algebraic types.

Our approximating semantics can be regarded as an abstract interpretation [13] where the concrete domain consists of measurable sets and the abstract domain consists of rectangular sets. In some ways, it is quite typical: it is sound, it labels expressions, the abstract domain is a lattice, and the exact semantics it approximates performs infinite computations. However, it is far from typical in other ways. It is used to run programs, not for static analysis. The abstraction boundaries are the if branches of completely unrolled, infinite programs, and are not fixed. There is no Kleene iteration. Infinite computations are done in a library of λ_{ZFC}-computable combinators, not by a semantic function. This cleanly separates the syntax from the semantics, and allows us to prove the exact semantics correct mostly by proving simple categorical properties.

Probabilistic languages can be approximately placed into two groups: those defined by an implementation, and those defined by a semantics.

Some languages defined by an implementation are a probabilistic Scheme by Koller and Pfeffer [28], BUGS [33], BLOG [35], BLAISE [10], Church [17], and Kiselyov's embedded language for O'Caml based on continuations [26]. The reports on these languages generally describe interpreters, compilers, and algorithms for sampling with probabilistic conditions. Recently, Wingate et al [53, 54] have defined the semantics of *nonstandard interpretations* that enable efficient inference, but do not define the languages.

Early work in probabilistic language semantics is not motivated by Bayesian concerns, and thus does not address conditioning. Kozen [29] defines the meaning of bounded-space, imperative "while" programs as functions from probability measures to probability measures. Hurd [22] proves properties about programs with binary random choice by encoding programs and portions of measure theory in HOL. Jones [23] develops a domain-theoretic variation of probability theory, and with it defines the probability monad, whose discrete version is a distribution-valued variation of the set or list monad. Ramsey and Pfeffer [43] define the probability monad measure-theoretically and implement a language for finite probability. Park [39] extends a λ-calculus with probabilistic choice from a general class of probability measures using inverse transform sampling.

Some recent work in probabilistic language semantics tackles conditioning. Pfeffer's IBAL [42] is the earliest lambda calculus with finite probabilistic choice that also defines conditional queries. Borgström et al [11] develop Fun, a first-order functional language without recursion, extended with probabilistic choice and conditioning. Its semantics interprets programs as *measure transformers* by transforming expressions into arrow-like combinators. The implementation generates a decomposition of the probability density represented by the program, if it exists. Bhat et al [9] replaces Fun's `if` with `match`, and interprets programs more directly as probability density functions by compositionally transforming expressions into an extension of the probability monad.

5.13 Conclusions and Future Work

To allow recursion and arbitrary conditions in probabilistic programs, we combined the power of measure theory with the unifying elegance of arrows. We

1. Defined a transformation from first-order programs to arbitrary arrows.
2. Defined the bottom arrow as a standard translation target.
3. Derived the uncomputable preimage arrow as an alternative target.
4. Derived a sound, computable approximation of the preimage arrow, and enough computable lifts to transform programs.

Critically, the preimage arrow's lift from the bottom arrow distributes over bottom arrow computations. Our semantics thus generalizes this process to all programs: 1) encode a program as a bottom arrow computation; 2) lift this computation to get an uncomputable function that computes preimages; 3) distribute the lift; and 4) replace uncomputable expressions with sound approximations.

Our semantics trades efficiency for simplicity by threading a constant, tree-shaped random source (Section 5.8.2). Passing subtrees instead would make `random` a constant-time primitive, and allow combinators to detect lack of change and return cached values. Other future optimization work includes creating new sampling algorithms, and using other easily measured but more expressive set representations, such as parallelotopes [4]. On the theory side, we intend to explore preimage computation's connection to type checking and type inference, investigate ways to integrate and leverage polymorphic type systems, and find the conditions under which preimage refinement is complete in the limit.

More broadly, we hope to advance Bayesian practice by providing a rich modeling language with an efficient, correct implementation, which allows general recursion and any computable, probabilistic condition.

Chapter 6

Implementation of Preimage Computation

An approximate answer to the right question is worth a good deal more than the exact answer to an approximate problem.

John W. Tukey

Chapter 7

Computing in Cantor's Paradise With $\lambda_{\mathbf{ZFC}}$

This chapter is derived from work published at the 11th *International Symposium on Functional and Logic Programming (FLOPS), 2012.*

No one shall expel us from the Paradise that Cantor has created.

David Hilbert

7.1 Introduction

Georg Cantor first proved some of the surprising consequences of assuming infinite sets exist. David Hilbert passionately defended Cantor's set theory as a mathematical foundation, coining the term "Cantor's Paradise" to describe the universe of transfinite sets in which most mathematics now takes place.

The calculations done in Cantor's Paradise range from computable to unimaginably uncomputable. Still, its inhabitants increasingly use computers to answer questions. We want to make domain-specific languages (DSLs) for writing these questions, with implementations that compute exact and approximate answers.

Such a DSL should have two meanings: an exact mathematical semantics, and an approximate computational one. A traditional, denotational approach is to give the exact as a transformation to first-order set theory, and because set theory is unlike any intended implementation language, the approximate as a transformation to a lambda calculus. However, deriving approximations while switching target languages is rife with opportunities to commit errors.

A more certain way is to define the exact semantics in a proof assistant like HOL [30] or Coq [8], prove theorems, and extract programs. The type systems confer an advantage: if the right theorems are proved, the programs are correct.

Unfortunately, reformulating and re-proving theorems in such an exacting way causes significant delays. For example, half of Joe Hurd's 2002 dissertation on probabilistic algorithms [22] is devoted to formalizing early-1900s measure theory in HOL. Our work in Bayesian inference would require at least three times as much formalization, even given the work we could build on.

Some middle ground is clearly needed: something between the traditional, error-prone way and the slow, absolutely certain way.

Instead of using a typed, higher-order logic, suppose we defined, in first-order set theory, an untyped lambda calculus that contained infinite sets and operations on them. We could interpret DSL terms exactly as uncomputable programs in this lambda calculus. But instead of redoing a century of work to extract programs that compute approximations, we could directly reuse first-order theorems to derive them from the uncomputable programs.

Conversely, set theory, which lacks lambdas and general recursion, is an awkward target language for a semantics that is intended to be implemented. Suppose we extended set theory with untyped lambdas (as objects, not quantifiers). We could still interpret DSL terms as operations on infinite objects. But instead of leaping from infinite sets and operations on them to implementations, we could replace those operations with computable approximations piece at a time.

If we had a lambda calculus with infinite sets as values, we could approach computability from above in a principled way, gradually changing programs for Cantor's Paradise until they can be implemented in Church's Purgatory.

We define that lambda calculus, λ_{ZFC}, and a call-by-value, big-step reduction semantics. To show that it is expressive enough, we code up the real numbers, arithmetic and limits, following standard analysis. To show that it simplifies language design, we define the uncomputable limit monad in λ_{ZFC}, and derive a computable, directly implementable replacement monad by applying standard topological theorems. When certain proof obligations are met, the output of programs that use the computable monad converge to the same values as the output of programs that use the uncomputable monad but are otherwise identical.

7.1.1 Language Tower and Terminology

λ_{ZFC}'s metalanguage is **first-order set theory**: first-order logic with equality extended with ZFC, or the Zermelo Fraenkel axioms and Choice (equivalently well-ordering). We also assume the existence of an inaccessible cardinal. Section 7.2 reviews the axioms, which λ_{ZFC}'s primitives are derived from.

To help ensure λ_{ZFC}'s definition conservatively extends set theory, we encode its terms as sets. For example, $\langle t_{\mathcal{P}}, \mathbb{R}\rangle$ is the encoding of "the powerset of the reals" as a pair, where $\langle x, y\rangle = \{\{x\}, \{x, y\}\}$ for any sets x and y.

λ_{ZFC}'s semantics reduces terms to terms; e.g. $\langle t_{\mathcal{P}}, \mathbb{R}\rangle$ reduces to the actual powerset of $\mathbb{R}$. Thus, λ_{ZFC} contains infinite terms. Infinitary languages are useful and definable: the infinitary lambda calculus [25] is an example, and Aczel's broadly used work [3] on inductive sets treats infinite inference rules explicitly.

For convenience, we define a language $\lambda^{-}_{\mathrm{ZFC}}$ of finite terms and a function $\mathcal{F}[\![\cdot]\!]$ from $\lambda^{-}_{\mathrm{ZFC}}$ to λ_{ZFC}. We can then write $\mathcal{P}\ \mathbb{R}$, meaning $\mathcal{F}[\![\mathcal{P}\ \mathbb{R}]\!] = \langle t_{\mathcal{P}}, \mathbb{R}\rangle$.

Semantic functions like $\mathcal{F}[\![\cdot]\!]$ and the interpretation of BNF grammars are defined in set theory's metalanguage, or the *meta*-metalanguage. Distinguishing metalanguages helps avoid paradoxes of definition such as Berry's paradox, which are particularly easy to stumble onto when dealing with infinities.

We write λ_{ZFC}^{-} terms in sans serif font, and the metalanguage and meta-metalanguage in *math font*. We write common keywords in **bold** and invented keywords in ***bold italics***. We abbreviate proofs for space.

7.2 Metalanguage: First-Order Set Theory

We assume readers are familiar with classical first-order logic with equality and its inference rules, but not set theory. Hrbacek and Jech [20] is a fine introduction.

Set theory extends classical first-order logic with equality, which distinguishes between truth-valued formulas ϕ and object-valued terms x. Set theory allows only sets as objects, and quantifiers like '$\forall$' may range only over sets.

We define predicates and functions using '$:=$'; for example, $nand(\phi_1, \phi_2) := \neg(\phi_1 \wedge \phi_2)$. They must be nonrecursive so they can be exhaustively applied. They are therefore **conservative extensions**: they do not prove more theorems.

To develop set theory, we make **proper extensions**, which prove more theorems, by adding symbols and axioms to first-order logic. For example, we first add '$\varnothing$' and '$\in$', and the **empty set axiom** $\forall x.\, x \notin \varnothing$.

We use '$:\equiv$' to define syntax; e.g. $\forall x \in A.\, P(x) :\equiv \forall x.\, (x \in A \Rightarrow P(x))$, where predicate application $P(x)$ represents a formula that may depend on x. We allow recursion in meta-metalanguage definitions if substitution terminates, so $\forall x_1\, x_2 \ldots x_n.\, \phi :\equiv \forall x_1.\, \forall x_2 \ldots x_n.\, \phi$ can bind any number of names.

We already have Axiom 0 (empty set). Now for the rest.

Axiom 1 (extensionality)**.** Define $A \subseteq B := \forall x \in A.\, x \in B$ and assume $A = B$ if they mutually are subsets; i.e. $\forall A\, B.\, (A \subseteq B \wedge B \subseteq A \Rightarrow A = B)$. □

The converse follows from substituting A for B or B for A.

Axiom 2 (foundation)**.** Define $A \not\pitchfork B := \forall x.\, (x \in A \Rightarrow x \notin B)$ ("A and B are disjoint") and assume $\forall A.\, (A = \varnothing) \vee \exists x \in A.\, x \not\pitchfork A$. □

Foundation implies that the following nondeterministic procedure always terminates: If input $A = \varnothing$, return A; otherwise restart with any $A' \in A$.

Thus, sets are roots of trees in which every upward path is unbounded but finite. Foundation is analogous to "all data constructors are strict."

Axiom 3 (powerset)**.** Add '$\mathcal{P}$' and assume $\forall\, A\, x.\, (x \in \mathcal{P}(A) \iff x \subseteq A)$. $\square$

A **hereditarily finite** set is finite and has only hereditarily finite members. Each such set first appears in some $\mathcal{P}(\mathcal{P}(...\mathcal{P}(\varnothing)...))$. For example, after $\{x, ...\}$ (literal set syntax) is defined, $\{\varnothing\} \in \mathcal{P}(\mathcal{P}(\varnothing))$. $\{\mathbb{R}\}$ is not hereditarily finite.

Axiom 4 (union)**.** Add '$\bigcup$' ("big" union) and assume $\forall\, A\, x.\, (x \in \bigcup A \iff \exists\, y.\, x \in y \wedge y \in A)$. $\square$

After $\{x, ...\}$ is defined, $\bigcup\{\{x, y\}, \{y, z\}\} = \{x, y, z\}$. Also, '$\bigcup$' can extract the object in a singleton set: if $A = \{x\}$, then $x = \bigcup A$.

Axiom 5 (replacement schema)**.** A binary predicate R can act as a function if it relates each x to exactly one y; i.e. $\forall\, x \in A.\, \exists!\; y.\, R(x, y)$, where '$\exists!$' means unique existence. We cannot quantify over predicates in first-order logic, but we can assume, for each such definable R, that $\forall\, y.\, (y \in \{y' \mid x \in A \wedge R(x, y')\} \iff \exists\, x \in A.\, R(x, y))$. Roughly, treating R as a function, if R's domain is a set, its image (range) is also a set. $\square$

A **schema** represents countably many axioms. If $R(n, m) \iff m = n + 1$, for example, then $\{m \mid n \in \mathbb{N} \wedge R(n, m)\}$ increments the natural numbers.

Define $\{F(x) \mid x \in A\} :\equiv \{y \mid x \in A \wedge y = F(x)\}$, analogous to map F A, for **functional replacement**. Now $\{n + 1 \mid n \in \mathbb{N}\}$ increments the naturals.

It seems replacement should be *defined* functionally, but predicates allow powerful nonconstructivism. Suppose $Q(y)$ for exactly one y. The **description operator** $\iota\, y.\, Q(y) :\equiv \bigcup\{y \mid x \in \mathcal{P}(\varnothing) \wedge Q(y)\}$ finds "the y such that $Q(y)$."

From the six axioms so far, we can define $A \cup B$ (binary union), $\{x, ...\}$ (literal finite sets), $\langle x, y, z, ...\rangle$ (ordered pairs and lists), $\{x \in A \mid Q(x)\}$ (bounded selection), $A \backslash B$ (relative complement), $\bigcap A$ ("big" intersection), $\bigcup_{x \in A} F(x)$ (indexed union), $A \times B$ (cartesian product), and $A \to B$ (total function spaces). For details, we recommend Paulson's remarkably lucid development in HOL [41].

7.2.1 The Gateway to Cantor's Paradise: Infinity

From the six axioms so far, we cannot construct a set that is closed under unboundedly many operations, such as the language of a recursive grammar.

Example 7.1 (interpreting a grammar)**.** We want to interpret $z ::= \varnothing \mid \langle \varnothing, z \rangle$. It should mean the least fixpoint of a function F_z, which, given a subset of z's language, returns a larger subset. To define F_z, replace '|' with '$\cup$', the terminal $\varnothing$ with $\{\varnothing\}$, and the rule $\langle \varnothing, z \rangle$ with functional replacement:

$$F_z(Z) \;:=\; \{\varnothing\} \cup \{\langle \varnothing, z \rangle \mid z \in Z\} \tag{7.1}$$

We could define $Z(0) := \varnothing$, then $Z(1) := F_z(Z(0)) = \{\varnothing, \langle \varnothing, \varnothing \rangle\}$, then $Z(2) = F_z(Z(1)) = \{\varnothing, \langle \varnothing, \varnothing \rangle, \langle \varnothing, \varnothing, \varnothing \rangle\}$, and so on. The language should be the union of all the $Z(n)$, but we cannot construct it without a set of all n. $\diamondsuit$

We follow Von Neumann, defining $0 := \varnothing$ as the **first ordinal number** and $s(n) := n \cup \{n\}$ to generate **successor ordinals**. Then $1 := s(0) = \{0\}$, $2 := s(1) = \{0, 1\}$, and $3 := s(2) = \{0, 1, 2\}$, and so on. The set of such numbers is the language of $n ::= 0 \mid s(n)$, which should be the least fixpoint of $F_n(N) := \{0\} \cup \{s(n) \mid n \in N\}$, similar to (7.1). Before we can prove this set exists, we must assume *some* fixpoint exists.

Axiom 6 (infinity)**.** $\exists I.\, I = F_n(I)$. $\square$

I is a bounding set, so it may contain more than just finite ordinals. But F_n is monotone in I, so by the Knaster-Tarksi theorem (suitably restricted [40]),

$$\omega \;:=\; \bigcap\{N \subseteq I \mid N = F_n(N)\} \tag{7.2}$$

is the least fixpoint of F_n: the finite ordinals, a model of the natural numbers.

Example 7.2 (interpreting a grammar)**.** We build the language of z recursively:

$$\begin{aligned} Z(0) &= \varnothing \\ Z(s(n)) &= F_z(Z(n)),\ n \in \omega \\ Z(\omega) &= \bigcup_{n\in\omega} Z(n) \end{aligned} \tag{7.3}$$

By induction, $Z(n)$ exists for every $n \in \omega$; therefore $Z(\omega)$ exists, so (7.3) is a conservative extension of set theory. It is not hard to prove (by induction) that $Z(\omega)$ is the set of all finite lists of $\varnothing$, and that it is the least fixpoint of F_z. $\diamondsuit$

Similarly to building the language $Z(\omega)$ of z in (7.3), we can build the set $\mathcal{V}(\omega)$ of all hereditarily finite sets (see Axiom 3) by iterating $\mathcal{P}$ instead of F_z:

$$\begin{aligned} \mathcal{V}(0) &= \varnothing \\ \mathcal{V}(s(n)) &= \mathcal{P}(\mathcal{V}(n)),\ n \in \omega \\ \mathcal{V}(\omega) &= \bigcup_{n\in\omega} \mathcal{V}(n) \end{aligned} \tag{7.4}$$

The set ω is not just a model of the natural numbers. It is also a number itself: the **first countable ordinal**. Indeed, ω is strikingly similar to every finite ordinal in two ways. First, it is defined as the set of its predecessors. Second, it has a successor $s(\omega) = \omega \cup \{\omega\}$. (Imagine it as $\{0, 1, 2, ..., \omega\}$.) Unlike finite, nonzero ordinals, ω has no *immediate* predecessor—it is a **limit ordinal**.

More limit ordinals allow iterating $\mathcal{P}$ further. It is not hard to build $\omega + \omega$, ω^2 and ω^ω as least fixpoints. The **Von Neumann hierarchy** generalizes (7.4):

$$\begin{aligned} \mathcal{V}(0) &= \varnothing \\ \mathcal{V}(s(\alpha)) &= \mathcal{P}(V(\alpha)),\ \text{ordinal } \alpha \\ \mathcal{V}(\beta) &= \bigcup_{\alpha\in\beta} \mathcal{V}(\alpha),\ \text{limit ordinal } \beta \end{aligned} \tag{7.5}$$

It is a theorem of ZFC that every set first appears in $\mathcal{V}(\alpha)$ for some ordinal α.

Equations (7.3,7.4,7.5) demonstrate **transfinite recursion**, set theory's unfold: defining a function V on ordinals, with $V(\beta)$ in terms of $V(\alpha)$ for every $\alpha \in \beta$.

7.2.2 Every Set Can Be Sequenced: Well-Ordering

A **sequence** is a total function from an ordinal to a codomain; e.g. $f \in 3 \rightarrow A$ is a length-3 sequence of A's elements. (An ordinal is comprised of its predecessors, so $3 = \{0, 1, 2\}$.) A **well-order** of A is a bijective sequence of A's elements.

Axiom 7 (well-ordering)**.** Suppose *Ord* identifies ordinals and $B \leftrightarrow A$ is the bijective mappings from B to A. Assume $\forall\, A.\, \exists\, \alpha\, f.\, \mathit{Ord}(\alpha) \wedge f \in \alpha \leftrightarrow A$; i.e. every set can be well-ordered. □

Because f is not unique, a well-ordering primitive could make λ_{ZFC}'s semantics nondeterministic. Fortunately, the existence of a cardinality operator is equivalent to well-ordering [48], so we will give λ_{ZFC} a cardinality primitive.

The **cardinality** of a set A is the smallest ordinal that can be put in bijection with A. Formally, if F contains A's well-orderings, $|A| = \bigcap\{\mathit{domain}(f) \mid f \in F\}$.

7.2.3 Infinity's Infinity: An Inaccessible Cardinal

The set $\mathcal{V}(\omega)$ of hereditarily finite sets is closed under powerset, union, replacement (with predicates restricted to $\mathcal{V}(\omega)$), and cardinality. It is also **transitive**: if $A \in \mathcal{V}(\omega)$, then $x \in \mathcal{V}(\omega)$ for all $x \in A$. These closure properties make it a **Grothendieck universe**: a set that acts like a set of all sets.

λ_{ZFC}'s values should contain ω and be closed under its primitives. But a Grothendieck universe containing ω cannot be proved from the typical axioms. If it exists, it must be equal to $\mathcal{V}(\kappa)$ for some **inaccessible cardinal** κ.

Axiom 8 (inaccessible cardinal)**.** Suppose $GU(V)$ if and only if V is a Grothendieck universe. Add 'κ' and assume $\mathit{Ord}(\kappa) \wedge (\kappa > \omega) \wedge GU(\mathcal{V}(\kappa))$. □

$$
\begin{array}{rcl}
e & ::= & n \mid v \mid e\ e \mid \textsf{if}\ e\ e\ e \mid e \in e \mid \bigcup e \mid \textsf{take}\ e \mid \mathcal{P}\ e \mid \textsf{image}\ e\ e \mid \textsf{card}\ e \\
v & ::= & \textsf{false} \mid \textsf{true} \mid \lambda.\, e \mid \emptyset \mid \boldsymbol{\omega} \\
n & ::= & 0 \mid 1 \mid 2 \mid \cdots
\end{array}
$$

Figure 7.1: The definition of λ^{-}_{ZFC}, which represents countably many λ_{ZFC} terms.

We call the sets in $\mathcal{V}(\kappa)$ **hereditarily accessible**.

Inaccessible cardinals are not usually assumed but are widely believed consistent. Set theorists regard them as no more dangerous than ω. Interpreting category theory with small and large categories, second-order set theory, or CIC in first-order set theory requires at least one inaccessible cardinal [6, 49, 52].

Constructing a set $A \notin \mathcal{V}(\kappa)$ requires assuming κ or an equivalent, so $\mathcal{V}(\kappa)$ easily contains most mathematics. In fact, most can be modeled well within $\mathcal{V}(2^{\omega})$; e.g. the model of $\mathbb{R}$ we define in Sect. 7.6 is in $\mathcal{V}(\omega + 11)$. Besides, if λ_{ZFC} needed to contain large cardinals, we could always assume even larger ones.

7.3 λ_{ZFC}'s Grammar

We define λ_{ZFC}'s terms in three steps. First, we define λ^{-}_{ZFC}, a language of finite terms with primitives that correspond with the ZFC axioms. Second, we encode these terms as sets. Third, guided by the first two steps, we define λ_{ZFC} by defining its terms, most of which are infinite, as sets in $\mathcal{V}(\kappa)$.

Figure 7.1 shows λ^{-}_{ZFC}'s grammar. Expressions e are typical: variables, values, application, if, and domain-specific primitives for membership, union, extraction ($\textsf{take}$), powerset, functional replacement ($\textsf{image}$), and cardinality. Values v are also typical: booleans and lambdas, and the domain-specific constants $\emptyset$ and $\boldsymbol{\omega}$.

In set theory, $\bigcup \{A\} = A$ holds for all A, so $\bigcup$ can extract the element from a singleton. In λ_{ZFC}, the encoding of $\bigcup \{A\}$ reduces to A only if A is an encoded set. Therefore, the

$$\text{Distinct } t_{\text{var}}, t_{\text{app}}, t_{\text{if}}, t_{\in}, t_{\cup}, t_{\text{take}}, t_{\mathcal{P}}, t_{\text{image}}, t_{\text{card}}, t_{\text{set}}, t_{\text{atom}}, t_{\lambda}, t_{\text{false}}, t_{\text{true}}$$

$$\begin{array}{rcl rcl rcl}
\mathcal{F}[\![n]\!] & := & \langle t_{\text{var}}, n\rangle & \mathcal{F}[\![\emptyset]\!] & := & set(\varnothing) & \mathcal{F}[\![\boldsymbol{\omega}]\!] & := & set(\omega) \\
\mathcal{F}[\![e_f\ e_x]\!] & := & \langle t_{\text{app}}, \mathcal{F}[\![e_f]\!], \mathcal{F}[\![e_x]\!]\rangle & \mathcal{F}[\![\mathsf{false}]\!] & := & a_{\text{false}} & a_{\text{false}} & := & \langle t_{\text{atom}}, t_{\text{false}}\rangle \\
\mathcal{F}[\![e_x \in e_A]\!] & := & \langle t_{\in}, \mathcal{F}[\![e_x]\!], \mathcal{F}[\![e_A]\!]\rangle & \mathcal{F}[\![\mathsf{true}]\!] & := & a_{\text{true}} & a_{\text{true}} & := & \langle t_{\text{atom}}, t_{\text{true}}\rangle \\
 & \cdots & & set(A) & = & \langle t_{\text{set}}, \{set(x) \mid x \in A\}\rangle & & &
\end{array}$$

Figure 7.2: The semantic function $\mathcal{F}[\![\cdot]\!]$ from λ^{-}_{ZFC} terms to λ_{ZFC} terms.

primitives must include take, which extracts A from $\{A\}$. In particular, extracting a lambda from an ordered pair requires take.

We use De Bruijn indexes with 0 referring to the innermost binding. Because we will define λ_{ZFC} terms as well-founded sets, by Axiom 2, countably many indexes is sufficient for λ_{ZFC} as well as λ^{-}_{ZFC}.

Figure 7.2 shows part of the meta-metalanguage function $\mathcal{F}[\![\cdot]\!]$ that encodes λ^{-}_{ZFC} terms as λ_{ZFC} terms. It distinguishes sorts of terms in the standard way, by pairing them with tags; e.g. if t_{set} is the "set" tag, then $\langle t_{\text{set}}, \varnothing\rangle$ encodes $\varnothing$.

To recursively tag sets, we add the axiom $set(A) = \langle t_{\text{set}}, \{set(x) \mid x \in A\}\rangle$. The **well-founded recursion theorem** proves that for all A, $set(A)$ exists, so this axiom is a conservative extension. The actual proof is tedious, but in short, *set* is structurally recursive. Now $set(\varnothing) = \langle t_{\text{set}}, \varnothing\rangle$ and $set(\omega)$ encodes $\boldsymbol{\omega}$.

7.3.1 An Infinite Set Rule For Finite BNF Grammars

There is no sensible reduction relation for λ^{-}_{ZFC}. (For example, $\mathcal{P}\ \emptyset$ cannot correctly reduce to a value because no value in λ^{-}_{ZFC} corresponds to $\{\varnothing\}$.) The easiest way to ensure a reduction relation exists for λ_{ZFC} is to include encodings of all the sets in $\mathcal{V}(\kappa)$ as values.

To define λ_{ZFC}'s terms, we first extend BNF with a set rule: $\{y^{*\alpha}\}$, where α is a cardinal. Roughly, it means sets comprised of no more than α terms from the language of y. Formally, it means $\mathcal{P}_{<}(Y, \alpha) := \{x \in \mathcal{P}(Y) \mid |x| < \alpha\}$, where Y is a subset of y's language generated while building a least fixpoint.

Example 7.3 (finite sets)**.** The grammar $h ::= \{h^{*\omega}\}$ should represent all hereditarily finite sets, or $\mathcal{V}(\omega)$. Intuitively, the single rule for h should be equivalent to countably many rules $h ::= \{\} \mid \{h\} \mid \{h, h\} \mid \{h, h, h\} \mid \cdots$.

Its language is the least fixpoint of $F_h(H) := \mathcal{P}_<(H, \omega)$. Further on, we will prove that F_h's least fixpoint is $\mathcal{V}(\omega)$ using a general theorem. $\diamondsuit$

Example 7.4 (accessible sets)**.** The language of $a ::= \{a^{*\kappa}\}$ is the least fixpoint of $F_a(A) := \mathcal{P}_<(A, \kappa)$, which should be $\mathcal{V}(\kappa)$. $\diamondsuit$

The following theorem schemas will make it easy to find least fixpoints.

Theorem 7.5. *Let F be a unary function. Define V by transfinite recursion:*

$$\begin{aligned} V(0) &= \varnothing \\ V(s(\alpha)) &= F(V(\alpha)) \\ V(\beta) &= \bigcup_{\alpha\in\beta} V(\alpha), \ \textit{limit ordinal } \beta \end{aligned} \tag{7.6}$$

Let γ be an ordinal. If F is monotone on $V(\gamma)$, V is monotone on γ, and $V(\gamma)$ is a fixpoint of F, then $V(\gamma)$ is also the ***least*** *fixpoint of F.*

Proof. By induction: successor case by monotonicity; limit by property of $\bigcup$. $\square$

All the Fs we define are monotone. In particular, the interpretations of $\{y^{*\alpha}\}$ rules are monotone because $\mathcal{P}$ is monotone. Further, all the Fs we define give rise to a monotone V. Grammar terminals "seed" every iteration with singleton sets, and $\{y^{*\alpha}\}$ rules seed every iteration with $\varnothing$.

From here on, we write F^α instead of $V(\alpha)$ to mean α iterations of F.

Theorem 7.6. *Suppose a grammar with $\{y^{*\alpha}\}$ rules and iterating function F. Then F's least fixpoint is F^γ, where γ is a regular cardinal not less than any α.*

Proof. Fixpoint by Aczel [3, Theorem 1.3.4]; least fixpoint by Theorem 7.5. $\square$

$$\begin{aligned}
e &::= n \mid v \mid \langle t_{\text{app}}, e, e\rangle \mid \langle t_{\text{if}}, e, e, e\rangle \mid \langle t_{\in}, e, e\rangle \mid \langle t_{\cup}, e\rangle \mid \langle t_{\text{take}}, e\rangle \mid \langle t_{\mathcal{P}}, e\rangle \mid \\
&\quad\ \langle t_{\text{image}}, e, e\rangle \mid \langle t_{\text{card}}, e\rangle \mid \langle t_{\text{set}}, \{e^{*\kappa}\}\rangle \\
v &::= a_{\text{false}} \mid a_{\text{true}} \mid \langle t_{\lambda}, e\rangle \mid \langle t_{\text{set}}, \{v^{*\kappa}\}\rangle \\
n &::= \langle t_{\text{var}}, 0\rangle \mid \langle t_{\text{var}}, 1\rangle \mid \cdots
\end{aligned}$$

Figure 7.3: λ_{ZFC}'s grammar. Here, $\{e^{*\kappa}\}$ means sets comprised of no more than κ terms from the language of e.

Example 7.7 (finite sets)**.** Because ω is regular, by Theorem 7.6, F_h's least fixpoint is F_h^{ω}. Further, $F_h(H) = \mathcal{P}(H)$ for all hereditarily finite H, and $\mathcal{V}(\omega)$ is closed under $\mathcal{P}$, so $F_h^{\omega} = \mathcal{V}(\omega)$, the set of all hereditarily finite sets. $\diamondsuit$

Example 7.8 (accessible sets)**.** By a similar argument, F_a's least fixpoint is $F_a^{\kappa} = \mathcal{V}(\kappa)$, the set of all hereditarily accessible sets. $\diamondsuit$

Example 7.9 (encoded accessible sets)**.** The language of $v ::= \langle t_{\text{set}}, \{v^{*\kappa}\}\rangle$ is comprised of the *encodings* of all the hereditarily accessible sets. $\diamondsuit$

7.3.2 The Grammar of Infinite, Encoded Terms

There are three main differences between λ_{ZFC}'s grammar in Fig. 7.3 and λ_{ZFC}^{-}'s grammar in Fig. 7.1. First, λ_{ZFC}'s grammar defines a language of terms that are already encoded as sets. Second, instead of the symbols $\emptyset$ and $\boldsymbol{\omega}$, it includes, as values, encoded sets of values. Most of these value terms are infinite, such as the encoding of $\boldsymbol{\omega}$. Third, it includes encoded sets of *expressions*.

The language of n is $N := \{\langle t_{\text{var}}, i\rangle \mid i \in \omega\}$. The rules for e and v are mutually recursive. Interpreted, but leaving out some of e's rules, they are

$$\begin{aligned}
F_e(E, V) &:= N \cup V \cup \{\langle t_{\text{app}}, e_f, e_x\rangle \mid \langle e_f, e_x\rangle \in E \times E\} \cup \cdots \cup \{\langle t_{\text{set}}, e\rangle \mid e \in \mathcal{P}_{<}(E, \kappa)\} \\
F_v(E, V) &:= \{a_{\text{false}}, a_{\text{true}}\} \cup \{\langle t_{\lambda}, e\rangle \mid e \in E\} \cup \{\langle t_{\text{set}}, v\rangle \mid v \in \mathcal{P}_{<}(V, \kappa)\}
\end{aligned} \tag{7.7}$$

To use Theorem 7.6, we need to iterate a single function. Note that the language pair $\langle E, V\rangle = \langle \{e, ...\}, \{v, ...\}\rangle$ is isomorphic to the single set of tagged terms $EV = \{\langle 0, e\rangle, ..., \langle 1, v\rangle, ...\}$.

Binary **disjoint union**, denoted $E \sqcup V$, creates such sets. We define F_{ev} by $F_{ev}(E \sqcup V) = F_e(E,V) \sqcup F_v(E,V)$. By Theorem 7.6, its least fixpoint is F_{ev}^{κ}, so we define E and V by $E \sqcup V = F_{ev}^{\kappa}$.

To make well-founded substitution easy, we will use capturing substitution, which does not capture when used on closed terms. Let $Cl(e)$ indicate whether a term is closed—this is structurally recursive. Then $E' := \{e \in E \mid Cl(e)\}$ and $V' := \{v \in V \mid Cl(v)\}$ contain only closed terms. Lastly, we define $\lambda_{\text{ZFC}} := E'$.

7.4 λ_{ZFC}'s Big-Step Reduction Semantics

We distinguish sets from other expressions using E_{set} and V_{set}, which merely check tags. We also lift set constructors to operate on encoded sets. For example, for cardinality, $\widehat{C}(v_A) := \mathit{set}(|\mathit{snd}(v_A)|)$ extracts the tagged set from v_A, applies $|\cdot|$, and recursively tags the resulting cardinal number. The rest are

$$\begin{aligned}
\widehat{\mathcal{P}}(v_A) &:= \langle t_{\text{set}}, \{\langle t_{\text{set}}, v_x\rangle \mid v_x \in \mathcal{P}(\mathit{snd}(v_A))\}\rangle \\
\widehat{\bigcup}(v_A) &:= \langle t_{\text{set}}, \bigcup\{\mathit{snd}(v_x) \mid v_x \in \mathit{snd}(v_A)\}\rangle \\
\widehat{I}(v_f, v_A) &:= \langle t_{\text{set}}, \{\langle t_{\text{app}}, v_f, v_x\rangle \mid v_x \in \mathit{snd}(v_A)\}\rangle
\end{aligned} \tag{7.8}$$

All but $\widehat{I}$ return values. Sets returned by $\widehat{I}$ are intended to be reduced further.

We use $e[n\backslash v]$ for De Bruijn substitution. Because e and v are closed, it is easy to define it using simple structural recursion on terms; it is thus conservative.

Figure 7.4 shows the reduction rules that define the reduction relation '$\Downarrow$'. Figure 7.4a has standard call-by-value rules: values reduce to themselves, and applications reduce by substitution. Figure 7.4b has the λ_{ZFC}-specific rules. Most simply use V_{set} to check tags before applying a lifted operator. The (image) rule replaces each value v_x in the set v_A with an application, generating a set expression, and the (set) rule reduces all the terms inside a set expression.

$$\frac{}{v \Downarrow v}\ \text{(val)} \quad \frac{e_f \Downarrow \langle t_\lambda, e_y\rangle \quad e_x \Downarrow v_x \quad e_y[0\backslash v_x] \Downarrow v_y}{\langle t_{\text{app}}, e_f, e_x\rangle \Downarrow v_y}\ \text{(ap)} \quad \frac{e_c \Downarrow a_{\text{true}} \quad e_t \Downarrow v_t}{\langle t_{\text{if}}, e_c, e_t, e_f\rangle \Downarrow v_t} \ \frac{e_c \Downarrow a_{\text{false}} \quad e_f \Downarrow v_f}{\langle t_{\text{if}}, e_c, e_t, e_f\rangle \Downarrow v_f}\ \text{(if)}$$

(a) Standard call-by-value reduction rules

$$\frac{e_A \Downarrow v_A \quad V_{\text{set}}(v_A) \quad e_x \Downarrow v_x \quad v_x \in \mathit{snd}(v_A)}{\langle t_\in, e_x, e_A\rangle \Downarrow a_{\text{true}}} \qquad \frac{e_A \Downarrow v_A \quad V_{\text{set}}(v_A) \quad e_x \Downarrow v_x \quad v_x \notin \mathit{snd}(v_A)}{\langle t_\in, e_x, e_A\rangle \Downarrow a_{\text{false}}}\ \text{(in)}$$

$$\frac{e_A \Downarrow v_A \quad V_{\text{set}}(v_A) \quad \forall\, v_x \in \mathit{snd}(v_A).\, V_{\text{set}}(v_x)}{\langle t_\cup, e_A\rangle \Downarrow \widehat{\bigcup}(v_A)}\ \text{(union)} \qquad \frac{e_A \Downarrow v_A \quad V_{\text{set}}(v_A)}{\langle t_{\mathcal{P}}, e_A\rangle \Downarrow \widehat{\mathcal{P}}(v_A)}\ \text{(pow)}$$

$$\frac{e_A \Downarrow v_A \quad V_{\text{set}}(v_A) \quad e_f \Downarrow \langle t_\lambda, e_y\rangle \quad \widehat{I}(\langle t_\lambda, e_y\rangle, v_A) \Downarrow v_y}{\langle t_{\text{image}}, e_f, e_A\rangle \Downarrow v_y}\ \text{(image)} \qquad \frac{e_A \Downarrow v_A \quad V_{\text{set}}(v_A)}{\langle t_{\text{card}}, e_A\rangle \Downarrow \widehat{C}(v_A)}\ \text{(card)}$$

$$\frac{E_{\text{set}}(e_A) \quad \forall\, e_x \in \mathit{snd}(e_A).\, \exists\, v_x.\, e_x \Downarrow v_x}{e_A \Downarrow \langle t_{\text{set}}, \{v_x \mid e_x \in \mathit{snd}(e_A) \wedge e_x \Downarrow v_x\}\rangle}\ \text{(set)} \qquad \frac{e_A \Downarrow \langle t_{\text{set}}, \{v_x\}\rangle}{\langle t_{\text{take}}, e_A\rangle \Downarrow v_x}\ \text{(take)}$$

(b) λ_{ZFC}-specific rules

Figure 7.4: Reduction rules defining λ_{ZFC}'s big-step, call-by-value semantics.

To define '$\Downarrow$' as a least fixpoint, we adapt Aczel's treatment [3]. We first define a bounding set for '$\Downarrow$' using closed terms, or $\mathcal{U} := E' \times V'$, so that $\Downarrow \subseteq \mathcal{U}$.

The rules in Fig. 7.4 can be used to define a predicate $D(R, \langle e, v\rangle)$. This predicate indicates whether some reduction rule, after replacing every '$\Downarrow$' in its premise with the approximation R, derives the conclusion $e \Downarrow v$.[1] Using D, we define a function that derives new conclusions from the known conclusions in R:

$$F_\Downarrow(R) \;:=\; \{c \in \mathcal{U} \mid D(R, c)\} \tag{7.9}$$

For example, $F_\Downarrow(\varnothing) = \{\langle v, v\rangle \mid v \in V\}$, by the (val) rule. $F_\Downarrow(F_\Downarrow(\varnothing))$ includes all pairs of non-value expressions and the values they reduce to in one derivation, as well as $\{\langle v, v\rangle \mid v \in V\}$. Generally, (val) ensures that iterating $F_\Downarrow$ is monotone.

[1] D is definable in first-order logic, but its definition does not aid understanding much.

For $F_{\Downarrow}$ itself to be non-monotone, for some $R \subseteq R' \subseteq \mathcal{U}$, there would have to be a conclusion $c \in F_{\Downarrow}(R)$ that is not in $F_{\Downarrow}(R')$. In other words, having more known conclusions could falsify a premise. None of the rules in Fig. 7.4 can do so.

Because $F_{\Downarrow}$ is monotone and iterating it is monotone, we can define $\Downarrow := F_{\Downarrow}^{\gamma}$ for some ordinal γ. If λ_{ZFC} had only finite terms, $\gamma = \omega$ iterations would reach a fixpoint. But a simple countable term shows why '$\Downarrow$' cannot be $F_{\Downarrow}^{\omega}$.

Example 7.10 (countably infinite term)**.** If s is the successor function in λ_{ZFC}, the term $t := \langle t_{\text{set}}, \{0, \langle t_{\text{app}}, s, 0\rangle, \langle t_{\text{app}}, s, \langle t_{\text{app}}, s, 0\rangle\rangle, ...\}\rangle$ should reduce to $\mathit{set}(\omega)$. The (set) rule's premises require each of t's subterms to reduce—using at least $F_{\Downarrow}^{\omega}$ because each subterm requires a finite, unbounded number of (ap) derivations. Though $F_{\Downarrow}^{s(\omega)}$ reduces t, for larger terms, we must iterate $F_{\Downarrow}$ much further. ♢

Theorem 7.11. *$\Downarrow := F_{\Downarrow}^{\kappa}$ is the least fixpoint of $F_{\Downarrow}$.*

Proof. Fixpoint by Aczel [3, Theorem 1.3.4]; least fixpoint by Theorem 7.5. □

Lastly, ZFC theorems that do not depend on κ can be applied to λ_{ZFC} terms.

Theorem 7.12. *λ_{ZFC}'s set values and $\langle t_{\in}, \cdot, \cdot\rangle$ are a model of ZFC-κ.*

Proof. $\mathcal{V}(\kappa)$, a model of ZFC-κ, is isomorphic to $v ::= \langle t_{\text{set}}, \{v^{*\kappa}\}\rangle$. □

7.5 Syntactic Sugar and a Small Set Library

From here on, we write only λ_{ZFC}^{-} terms, assume $\mathcal{F}[\![\cdot]\!]$ is applied, and no longer distinguish λ_{ZFC}^{-} from λ_{ZFC}.

We use names instead of De Bruijn indexes and assume names are converted. We get alpha equivalence for free; for example, $\lambda \mathsf{x}.\, \mathsf{x} = \langle t_{\lambda}, \langle t_{\text{var}}, 0\rangle\rangle = \lambda \mathsf{y}.\, \mathsf{y}$.

λ_{ZFC} does not contain terms with free variables. To get around this technical limitation, we assume free variables are metalanguage names for closed terms.

We allow the primitives $(\in)$, $\bigcup$, take, $\mathcal{P}$, image and card to be used as if they were functions. Enclosing infix operators in parenthesis refers to them as functions, as in $(\in)$. We partially apply infix functions using Haskell-like sectioning rules, so $(\mathsf{x}\in)$ means $\lambda \mathsf{A}.\,\mathsf{x}\in\mathsf{A}$ and $(\in\mathsf{A})$ means $\lambda \mathsf{x}.\,\mathsf{x}\in\mathsf{A}$.

We define first-order objects using ":=", as in $0 := \emptyset$, and syntax with ":≡", as in $\lambda\, x_1\, x_2 \ldots x_n.\, e :\equiv \lambda\, x_1.\, \lambda\, x_2 \ldots x_n.\, e$ to automatically curry. Function definitions expand to lambdas (using fixpoint combinators for recursion); for example, $\mathsf{x} = \mathsf{y} \;:=\; \mathsf{x} \in \{\mathsf{y}\}$ and $(=) := \lambda \mathsf{x}\,\mathsf{y}.\,\mathsf{x}\in\{\mathsf{y}\}$ equivalently define $(=)$ in terms of $(\in)$. We destructure pairs implicitly in binding patterns, as in $\lambda\,\langle\mathsf{x},\mathsf{y}\rangle.\,\mathsf{f}\;\mathsf{x}\;\mathsf{y}$.

To do anything useful, we need a small set library. The definitions are similar to the metalanguage definitions we omitted in Section 7.2, and we similarly elide most of the λ_{ZFC} definitions. However, some deserve special mention.

Because λ_{ZFC} has only *functional* replacement, we cannot define unbounded $\forall$ and $\exists$. But we can define bounded quantifiers in terms of bounded selection, or

$$\mathsf{select\;f\;A} \;:=\; \bigcup\;(\mathsf{image}\;(\lambda\mathsf{x}.\,\mathsf{if}\;(\mathsf{f\;x})\;\{\mathsf{x}\}\;\emptyset)\;\mathsf{A}) \tag{7.10}$$

We also define a bounded description operator:

$$\iota\, x \in e_A.\, e_f \;:\equiv\; \mathsf{take}\;(\mathsf{select}\;(\lambda\, x.\, e_f)\;e_A) \tag{7.11}$$

Note $\iota\, x \in e_A.\, e_f$ reduces only if $e_f \Downarrow \mathsf{true}$ for exactly one $x \in e_A$.

Thus, converting a predicate to an object requires both unique existence and a bounding set. For example, if

$$\langle e_x, e_y\rangle \;:\equiv\; \{\{e_x, \{e_x, e_y\}\}\} \tag{7.12}$$

defines ordered pairs, then

$$\mathsf{fst\;p} \;:=\; \iota\,\mathsf{x} \in (\bigcup\;\mathsf{p}).\,\exists\,\mathsf{y} \in (\bigcup\;\mathsf{p}).\,\mathsf{p} = \langle\mathsf{x},\mathsf{y}\rangle \tag{7.13}$$

takes the first element.

The ***set monad*** simulates nondeterministic choice. We define it by

$$\begin{aligned} \mathsf{return_{set}}\ \mathsf{a} &:= \{\mathsf{a}\} \\ \mathsf{bind_{set}}\ \mathsf{A}\ \mathsf{f} &:= \bigcup\ (\mathsf{image}\ \mathsf{f}\ \mathsf{A}) \end{aligned} \tag{7.14}$$

Using $\mathsf{bind\ m\ f} = \mathsf{join\ (lift\ f\ m)}$, evidently $\mathsf{lift_{set}} := \mathsf{image}$ and $\mathsf{join_{set}} := \bigcup$. The proofs of the monad laws follow the proofs for the list monad. We also define $\{x \in e_A\}.\, e_f :\equiv \mathsf{bind_{set}}\ (\lambda x.\, e_f)\ e_A$, read "choose x in e_A, then e_f." For example, binary cartesian product is $\mathsf{A} \times \mathsf{B} := \{\mathsf{x} \in \mathsf{A}\}.\, \{\mathsf{y} \in \mathsf{B}\}.\, \mathsf{return_{set}}\ \langle \mathsf{x}, \mathsf{y} \rangle$.

Every $\mathsf{f} \in \mathsf{A} \to \mathsf{B}$ is shaped $\mathsf{f} = \{\langle \mathsf{x}_1, \mathsf{y}_1 \rangle, \langle \mathsf{x}_2, \mathsf{y}_2 \rangle, ...\}$ and is total on A. To distinguish these hash tables from lambdas, we call them **mappings**. They can be applied by $\mathsf{ap\ f\ x} := \iota\, \mathsf{y} \in (\mathsf{range\ f}).\, \langle \mathsf{x}, \mathsf{y} \rangle \in \mathsf{f}$, but we write just $\mathsf{f\ x}$. We define $e_f|_{e_A} :\equiv \mathsf{image}\ (\lambda \mathsf{x}.\, \langle \mathsf{x}, e_f\ \mathsf{x} \rangle)\ e_A$ to convert a lambda or to restrict a mapping to $\mathsf{e_A}$. We usually use $\lambda x \in e_A.\, e_y :\equiv (\lambda x.\, e_y)|_{e_A}$ to define mappings.

A sequence of A is a mapping $\mathsf{xs} \in \alpha \to \mathsf{A}$ for some ordinal α. For example, $\mathsf{ns} := \lambda \mathsf{n} \in \omega.\, \mathsf{n}$ is a countable sequence in $\omega \to \omega$ of increasing finite ordinals. We assume useful sequence functions like map, $\mathsf{map2}$ and drop are defined.

7.6 Example: The Reals From the Rationals

Here, we demonstrate that λ_{ZFC} is computationally powerful enough to construct the real numbers. For a clear, well-motivated, rigorous treatment in first-order set theory without lambdas, we recommend Abbott's excellent introductory text [2].

Assume we have a model $\mathbb{Q}, +_{\mathbb{Q}}, -_{\mathbb{Q}}, \times_{\mathbb{Q}}, \div_{\mathbb{Q}}$ of the rationals and rational arithmetic.[2] To get the reals, we close the rationals under countable limits.

We represent limits of rationals with sequences in $\omega \to \mathbb{Q}$. To select only the converging ones, we must define what convergence means. We start with convergence to zero

[2]Though the λ_{ZFC} development of $\mathbb{Q}$ is short and elegant, it does not fit in this paper.

and equivalence. Given $\mathbb{Q}^+$, '$<_{\mathbb{Q}}$' and $|\cdot|_{\mathbb{Q}}$, define

$$\begin{aligned} \mathsf{conv\text{-}zero?_R\ xs} &:= \forall \varepsilon \in \mathbb{Q}^+.\, \exists \mathsf{N} \in \omega.\, \forall \mathsf{n} \in \omega.\, (\mathsf{N} \in \mathsf{n} \Rightarrow |\mathsf{xs\ n}|_{\mathbb{Q}} <_{\mathbb{Q}} \varepsilon) \\ \mathsf{xs} =_{\mathsf{R}} \mathsf{ys} &:= \mathsf{conv\text{-}zero?_R\ (map2\ }(-_{\mathbb{Q}})\ \mathsf{xs\ ys)} \end{aligned} \tag{7.15}$$

So a sequence $\mathsf{xs} \in \omega \to \mathbb{Q}$ converges to zero if, for any positive ε, there is some index N after which all xs are smaller than ε. Two sequences are equivalent ($=_{\mathsf{R}}$) if their pointwise difference converges to zero.

We should be able to drop finitely many elements from a converging sequence without changing its limit. Therefore, a sequence of rationals converges to *something* when it is equivalent to all of its suffixes. We thus define an equivalent to the Cauchy convergence test, and use it to select the converging sequences:

$$\begin{aligned} \mathsf{conv?_R\ xs} &:= \forall \mathsf{n} \in \omega.\, \mathsf{xs} =_{\mathsf{R}} (\mathsf{drop\ n\ xs}) \\ \mathsf{R} &:= \mathsf{select\ conv?_R}\ (\omega \to \mathbb{Q}) \end{aligned} \tag{7.16}$$

But R (equipped with the equivalence relation $=_{\mathsf{R}}$) is not the real numbers as they are normally defined: converging sequences in R may be equivalent but not equal. To decide real equality using λ_{ZFC}'s '$=$', we partition R into disjoint sets of equivalent sequences—we make a **quotient space**. Thus,

$$\begin{aligned} \mathsf{quotient\ A}\ (=_{\mathsf{A}}) &:= \mathsf{image}\ (\lambda \mathsf{x}.\, \mathsf{select}\ (=_{\mathsf{A}} \mathsf{x})\ \mathsf{A})\ \mathsf{A} \\ \mathbb{R} &:= \mathsf{quotient\ R}\ (=_{\mathsf{R}}) \end{aligned} \tag{7.17}$$

defines the reals with extensional equality.

To define real arithmetic, we must lift rational arithmetic to sequences and then to sets of sequences. The $\mathsf{map2}$ function lifts, say, $+_{\mathbb{Q}}$ to sequences, as in $(+_{\mathsf{R}}) := \mathsf{map2}\ (+_{\mathbb{Q}})$. To lift $+_{\mathsf{R}}$ to sets of sequences, note that sets of sequences are models of nondeterministic sequences, suggesting the set monad. We define $\mathsf{lift2_{set}\ f\ A\ B} := \{\mathsf{a} \in \mathsf{A}\}.\, \{\mathsf{b} \in \mathsf{B}\}.\, \mathsf{return_{set}\ (f\ a\ b)}$ to lift two-argument functions to the set monad. Now $(+) := \mathsf{lift2_{set}}\ (+_{\mathsf{R}})$, and similarly for the other operators.

Using $\mathsf{lift2}_{\mathsf{set}}$ is atypical, so we prove that $\mathsf{A} + \mathsf{B} \in \mathbb{R}$ when $\mathsf{A} \in \mathbb{R}$ and $\mathsf{B} \in \mathbb{R}$, and similarly for the other operators. It follows from the fact that the rational operators lifted to sequences are surjective morphisms, and this theorem:

Theorem 7.13. *Suppose $=_{\mathsf{X}}$ is an equivalence relation on X, and define its quotient $\mathbb{X} :=$ $\mathsf{quotient}\ \mathsf{X}\ (=_{\mathsf{X}})$. If op is surjective on X and a binary morphism for $=_{\mathsf{X}}$, then $(\mathsf{lift2}_{\mathsf{set}}\ \mathsf{op}\ \mathsf{A}\ \mathsf{B}) \in \mathbb{X}$ for all $\mathsf{A} \in \mathbb{X}$ and $\mathsf{B} \in \mathbb{X}$.*

Proof. Reduce to an equality. Case '$\subseteq$' by morphism; case '$\supseteq$' by surjection. $\square$

Now for real limits. If $\mathbb{R}^+$, '$<$', and $|\cdot|$ are defined, we can define $\mathsf{conv\text{-}zero?}_{\mathbb{R}}$, which is like (7.15) but operates on real sequences $\mathsf{xs} \in \omega \to \mathbb{R}$. We then define $\mathsf{limit}_{\mathbb{R}}\ \mathsf{xs} :=$ $\iota\, \mathsf{y} \in \mathbb{R}.\, \mathsf{conv\text{-}zero?}_{\mathbb{R}}\ (\mathsf{map}\ (-\ \mathsf{y})\ \mathsf{xs})$ to calculate their limits.

From here, it is not difficult to treat $\mathbb{Q}$ and $\mathbb{R}$ uniformly by redefining $\mathbb{Q} \subset \mathbb{R}$.

7.7 Example: Computable Real Limits

Exact real computation has been around since Turing's seminal paper [47]. The novelty here is how we do it. We define the ***limit monad*** in λ_{ZFC} for expressing calculations involving limits, with $\mathsf{bind}_{\mathsf{lim}}$ defined in terms of a general limit. We then derive a limit-free, computable replacement $\mathsf{bind}'_{\mathsf{lim}}$. Replacing $\mathsf{bind}_{\mathsf{lim}}$ with $\mathsf{bind}'_{\mathsf{lim}}$ in a λ_{ZFC} term incurs proof obligations. If they can be met, the computable λ_{ZFC} term has the same limit as the original, uncomputable term.

In other words, entirely in λ_{ZFC}, we define uncomputable things, and gradually turn them into computable, directly implementable approximations.

The proof obligations are related to topological theorems [36] that we will import as lemmas. By Theorem 7.12, we are allowed to use them directly.

At this point, it is helpful to have a simple, informal type system, which we can easily add to the untyped λ_{ZFC}. $\mathsf{A} \Rightarrow \mathsf{B}$ is a lambda or mapping type. $\mathsf{A} \to \mathsf{B}$ is the set of total mappings from A to B. A set is a membership proposition.

7.7.1 The Limit Monad

We first need a universe $\mathbb{U}$ of values that is closed under sequencing; i.e. if $\mathsf{A} \subset \mathbb{U}$ then so is $\omega \to \mathsf{A}$. Define $\mathbb{U}$ as the language of $\mathsf{u} ::= \mathbb{R} \mid \omega \to \mathsf{u}$. A complete product metric $\delta : \mathbb{U} \Rightarrow \mathbb{U} \Rightarrow \mathbb{R}$ exists; therefore, a function $\mathsf{limit} : (\omega \to \mathbb{U}) \Rightarrow \mathbb{U}$ similar to $\mathsf{limit}_{\mathbb{R}}$ exists that calculates limits. These are all λ_{ZFC}-definable.

The limit monad's computations are of type $\omega \to \mathbb{U}$. The type does not imply convergence, which must be proved separately. Its run function is limit.

Example 7.14 (infinite series)**.** Define $\mathsf{partial\text{-}sums} : (\omega \to \mathbb{R}) \Rightarrow (\omega \to \mathbb{R})$ first by $\mathsf{partial\text{-}sums}'\ \mathsf{xs} := \lambda \mathsf{n}.\, \mathsf{if}\ (\mathsf{n} = 0)\ (\mathsf{xs}\ 0)\ ((\mathsf{xs}\ \mathsf{n}) + (\mathsf{partial\text{-}sums}'\ \mathsf{xs}\ (\mathsf{n} - 1)))$. (The sequence is recursively defined, so we cannot use $\lambda \mathsf{n} \in \omega.\, e$ to immediately create it.) Then restrict its output: $\mathsf{partial\text{-}sums}\ \mathsf{xs} := (\mathsf{partial\text{-}sums}'\ \mathsf{xs})|_{\omega}$.

Now $\sum_{n\in\omega} e :\equiv \mathsf{limit}\ (\mathsf{partial\text{-}sums}\ \lambda n \in \omega.\, e)$, or the limit of partial sums. Even if xs converges, $\mathsf{partial\text{-}sums}\ \mathsf{xs}$ may not; e.g. if $\mathsf{xs} = \lambda \mathsf{n} \in \omega.\, \frac{1}{\mathsf{n}+1}$. ♢

The limit monad's $\mathsf{return}_{\mathsf{lim}} : \mathbb{U} \Rightarrow (\omega \to \mathbb{U})$ creates constant sequences, and its $\mathsf{bind}_{\mathsf{lim}} : (\omega \to \mathbb{U}) \Rightarrow (\mathbb{U} \Rightarrow (\omega \to \mathbb{U})) \Rightarrow (\omega \to \mathbb{U})$ simply takes a limit:

$$\begin{aligned} \mathsf{return}_{\mathsf{lim}}\ \mathsf{x} &:= \lambda \mathsf{n} \in \omega.\, \mathsf{x} \\ \mathsf{bind}_{\mathsf{lim}}\ \mathsf{xs}\ \mathsf{f} &:= \mathsf{f}\ (\mathsf{limit}\ \mathsf{xs}) \end{aligned} \tag{7.18}$$

The left identity and associativity monad laws hold using '$=$' for equivalence. However, right identity holds only in the limit, so we define equivalence by $\mathsf{xs} =_{\mathsf{lim}} \mathsf{ys} \ := \ \mathsf{limit}\ \mathsf{xs} = \mathsf{limit}\ \mathsf{ys}$.

Example 7.15 (lifting)**.** Define $\mathsf{lift}_{\mathsf{lim}}\ \mathsf{f}\ \mathsf{xs} := \mathsf{bind}_{\mathsf{lim}}\ \mathsf{xs}\ \lambda \mathsf{x}.\, \mathsf{return}_{\mathsf{lim}}\ (\mathsf{f}\ \mathsf{x})$, as is typical. Substituting $\mathsf{bind}_{\mathsf{lim}}$ and reducing reveals that $\mathsf{f}\ (\mathsf{limit}\ \mathsf{xs}) = \mathsf{limit}\ (\mathsf{lift}_{\mathsf{lim}}\ \mathsf{f}\ \mathsf{xs})$. That is, using $\mathsf{lift}_{\mathsf{lim}}$ pulls limit out of f's argument. ♢

Example 7.16 (exponential)**.** The Taylor series expansion of the exponential function is $\mathsf{exp\text{-}seq} : \mathbb{R} \Rightarrow (\omega \to \mathbb{R})$, defined by $\mathsf{exp\text{-}seq}\ \mathsf{x} := \mathsf{partial\text{-}sums}\ \lambda \mathsf{n} \in \omega.\, \frac{\mathsf{x}^{\mathsf{n}}}{\mathsf{n}\text{-}}$. It always converges, so $\mathsf{limit}\ (\mathsf{exp\text{-}seq}\ \mathsf{x}) = \sum_{\mathsf{n}\in\omega} \frac{\mathsf{x}^{\mathsf{n}}}{\mathsf{n\text{-}}} = \mathsf{exp}\ \mathsf{x}$ for $\mathsf{x} \in \mathbb{R}$. To exponentiate converging sequences, define $\mathsf{exp}_{\mathsf{lim}}\ \mathsf{xs} := \mathsf{bind}_{\mathsf{lim}}\ \mathsf{xs}\ \mathsf{exp\text{-}seq}$. ♢

7.7.2 The Computable Limit Monad

We derive the computable limit monad in two steps. In the first, longest step, we replace the limit monad's defining functions with those that do not use limit. But computations will still have type $\omega \to \mathbb{U}$, whose inhabitants are not directly implementable, so in the second step, we give them a lambda type.

We define $\mathsf{return}'_{\mathsf{lim}} := \mathsf{return}_{\mathsf{lim}}$. A drop-in, limit-free replacement for $\mathsf{bind}_{\mathsf{lim}}$ does not exist, but there is one that incurs three proof obligations. Without imposing rigid constraints on using $\mathsf{bind}_{\mathsf{lim}}$, we cannot meet them automatically. But we can separate them by factoring $\mathsf{bind}_{\mathsf{lim}}$ into $\mathsf{lift}_{\mathsf{lim}}$ and $\mathsf{join}_{\mathsf{lim}}$.

Limit-Free Lift. Substituting to get $\mathsf{lift}_{\mathsf{lim}}\ \mathsf{f}\ \mathsf{xs} = \mathsf{return}_{\mathsf{lim}}\ (\mathsf{f}\ (\mathsf{limit}\ \mathsf{xs}))$ exposes the use of limit. Removing it requires continuity and definedness.

Lemma 7.17 (continuity in metric spaces)**.** *Let* $\mathsf{f} : \mathsf{A} \Rightarrow \mathsf{B}$ *with* A *a metric space. Then* f *is continuous at* $\mathsf{x} \in \mathsf{A}$ *if and only if for all* $\mathsf{xs} \in \omega \to \mathsf{A}$ *for which* $\mathsf{limit}\ \mathsf{xs} = \mathsf{x}$ *and* f *is defined on all elements of* xs*,* $\mathsf{f}\ (\mathsf{limit}\ \mathsf{xs}) = \mathsf{limit}\ (\mathsf{map}\ \mathsf{f}\ \mathsf{xs})$*.*

So if $\mathsf{f} : \mathbb{U} \Rightarrow \mathbb{U}$ is continuous at $\mathsf{limit}\ \mathsf{xs}$, and f is defined on all xs, then

$$\begin{aligned}
\mathsf{limit}\ (\mathsf{lift}_{\mathsf{lim}}\ \mathsf{f}\ \mathsf{xs}) &= \mathsf{limit}\ (\mathsf{return}_{\mathsf{lim}}\ (\mathsf{f}\ (\mathsf{limit}\ \mathsf{xs}))) \\
&= \mathsf{limit}\ (\mathsf{return}_{\mathsf{lim}}\ (\mathsf{limit}\ (\mathsf{map}\ \mathsf{f}\ \mathsf{xs}))) \qquad (7.19) \\
&= \mathsf{limit}\ (\mathsf{map}\ \mathsf{f}\ \mathsf{xs})
\end{aligned}$$

Thus, $\mathsf{lift}_{\mathsf{lim}}\ \mathsf{f}\ \mathsf{xs} =_{\mathsf{lim}} \mathsf{map}\ \mathsf{f}\ \mathsf{xs}$, so $\mathsf{lift}'_{\mathsf{lim}}\ \mathsf{f}\ \mathsf{xs} := \mathsf{map}\ \mathsf{f}\ \mathsf{xs}$. Using $\mathsf{lift}_{\mathsf{lim}}\ \mathsf{f}\ \mathsf{xs}$ instead of $\mathsf{lift}'_{\mathsf{lim}}\ \mathsf{f}\ \mathsf{xs}$ requires f to be continuous at $\mathsf{limit}\ \mathsf{xs}$ and defined on all xs.

Limit-Free Join. Using $\mathsf{join}\ \mathsf{m} = \mathsf{bind}\ \mathsf{m}\ \lambda\mathsf{x}.\,\mathsf{x}$ results in $\mathsf{join}_{\mathsf{lim}} = \mathsf{limit}$. Removing limit might seem hopeless—until we distribute it pointwise over xss.

Lemma 7.18 (limits of sequences). *Let* $\mathsf{f} \in \omega \to \omega \to \mathsf{A}$*, where* $\omega \to \mathsf{A}$ *has a product topology. Then* $\mathsf{limit\ f} = \lambda\mathsf{n} \in \omega.\,\mathsf{limit\ (flip\ f\ n)}$*, where* $\mathsf{flip\ f\ x\ y} := \mathsf{f\ y\ x}$*.*

A countable product metric defines a product topology, so $\mathsf{join_{lim}\ xss} := \lambda\mathsf{n} \in \omega.\,\mathsf{limit\ (flip\ xss\ n)}$. Now we can remove limit by restricting $\mathsf{join_{lim}}$'s input.

Definition 7.19 (uniform convergence). *A sequence* $\mathsf{f} \in \omega \to \omega \to \mathbb{U}$ *converges* ***uniformly*** *if* $\forall\,\varepsilon \in \mathbb{R}^+.\,\exists\,\mathsf{N} \in \omega.\,\forall\,\mathsf{n},\mathsf{m} > \mathsf{N}.\,(\delta\ \mathsf{(f\ n\ m)\ (limit\ (f\ n))}) < \varepsilon$*.*

Lemma 7.20 (collapsing limits). *If* $\mathsf{f} \in \omega \to \omega \to \mathbb{U}$ *converges uniformly, and* $\mathsf{r},\mathsf{s} : \omega \Rightarrow \omega$ *increase, then* $\mathsf{limit}\ \lambda\mathsf{n} \in \omega.\,\mathsf{limit\ (f\ n)} = \mathsf{limit}\ \lambda\mathsf{n} \in \omega.\,\mathsf{f\ (r\ n)\ (s\ n)}$*.*

So if $\mathsf{flip\ xss}$ converges uniformly, then

$$\begin{aligned} \mathsf{limit\ (join_{lim}\ xss)} &= \mathsf{limit}\ \lambda\mathsf{n} \in \omega.\,\mathsf{limit\ (flip\ xss\ n)} \\ &= \mathsf{limit}\ \lambda\mathsf{n} \in \omega.\,\mathsf{flip\ xss\ (r\ n)\ (s\ n)} \end{aligned} \tag{7.20}$$

We define $\mathsf{join'_{lim}} : (\omega \to \omega \to \mathbb{U}) \Rightarrow (\omega \to \mathbb{U})$ by $\mathsf{join'_{lim}\ xss} := \lambda\mathsf{n} \in \omega.\,\mathsf{xss\ n\ n}$. Replacing $\mathsf{join_{lim}\ xss}$ with $\mathsf{join'_{lim}\ xss}$ requires that $\mathsf{flip\ xss}$ converge uniformly.

Limit-Free Bind. Define $\mathsf{bind'_{lim}\ xs\ f} := \mathsf{join'_{lim}\ (lift'_{lim}\ f\ xs)}$. It inherits obligations to prove that f is continuous at $\mathsf{limit\ xs}$ and defined on all xs, and to prove that $\mathsf{flip\ (map\ f\ xs)}$ converges uniformly.

Example 7.21 (exponential cont.). Define $\mathsf{exp'_{lim}}$ by replacing $\mathsf{bind_{lim}}$ by $\mathsf{bind'_{lim}}$ in $\mathsf{exp_{lim}}$, so $\mathsf{exp'_{lim}\ xs} := \mathsf{bind'_{lim}\ xs\ exp\text{-}seq}$. We now meet the proof obligations.

Lemma 7.22. *Let* $\mathsf{f} : \mathsf{A} \Rightarrow (\omega \to \mathsf{B})$*. If* $\omega \to \mathsf{B}$ *has a product topology, then* f *is continuous if and only if* $\mathsf{(flip\ f)\ n}$ *is continuous for every* $\mathsf{n} \in \omega$*.*

We have a product topology, so for the first obligation, pointwise continuity is enough. Let $\mathsf{g} := \mathsf{flip\ exp\text{-}seq}$. Every $\mathsf{g\ n}$ is a finite polynomial, and thus continuous. The second obligation, that $\mathsf{exp\text{-}seq}$ is defined on all xs, is obvious. The third, that $\mathsf{flip\ (map\ exp\text{-}seq\ xs)}$ converges uniformly, can be proved using the Weierstrass M test [2, Theorem 6.4.5]. ♢

Example 7.23 (π)**.** The definition of $\mathsf{arctan}_{\mathsf{lim}}$ is like $\mathsf{exp}_{\mathsf{lim}}$'s. Defining $\mathsf{arctan}'_{\mathsf{lim}}$, including proving correctness, is like defining $\mathsf{exp}'_{\mathsf{lim}}$. To compute π, we use

$$\begin{aligned}\pi_{\mathsf{lim}} \ := \ & ((\mathsf{return}_{\mathsf{lim}}\ 16) \times_{\mathsf{lim}} (\mathsf{arctan}_{\mathsf{lim}}\ (\mathsf{return}_{\mathsf{lim}}\ \tfrac{1}{5}))) -_{\mathsf{lim}} \\ & ((\mathsf{return}_{\mathsf{lim}}\ 4) \times_{\mathsf{lim}} (\mathsf{arctan}_{\mathsf{lim}}\ (\mathsf{return}_{\mathsf{lim}}\ \tfrac{1}{239})))\end{aligned} \tag{7.21}$$

where $(\cdot)_{\mathsf{lim}}$ are lifted arithmetic operators. Because (7.21) does not directly use $\mathsf{bind}_{\mathsf{lim}}$, defining the limit-free π'_{lim} imposes no proof obligations. $\diamondsuit$

In general, using functions defined in terms of $\mathsf{bind}'_{\mathsf{lim}}$ requires little more work than using functions on finite values. The implicit limits are pulled outward and collapse on their own, hidden within monadic computations.

Computable Sequences. Lambdas are the simplest model of $\omega \to \mathbb{U}$. After manipulating some terms, we define the final, computable limit monad by $\mathsf{return}'_{\mathsf{lim}}\ \mathsf{x} := \lambda\mathsf{n}.\,\mathsf{x}$ and $\mathsf{bind}'_{\mathsf{lim}}\ \mathsf{xs}\ \mathsf{f} \ := \ \lambda\mathsf{n}.\,\mathsf{f}\ (\mathsf{xs}\ \mathsf{n})\ \mathsf{n}$. Computations have type $\omega \Rightarrow \mathbb{U}'$, where $\mathbb{U}'$ contains countable sequences of rationals.

Implementation. We have transliterated $\mathsf{return}'_{\mathsf{lim}}$, $\mathsf{bind}'_{\mathsf{lim}}$, $\mathsf{exp}'_{\mathsf{lim}}$, $\mathsf{arctan}'_{\mathsf{lim}}$ and π'_{lim} into Racket [16], using its built-in models of ω and $\mathbb{Q}$. Even without optimizations, $\pi'_{\mathsf{lim}}\ 141$ yields a rational approximation in a few milliseconds that is correct to 200 digits. More importantly, $\mathsf{exp}'_{\mathsf{lim}}$, $\mathsf{arctan}'_{\mathsf{lim}}$ and π'_{lim} are almost identical to their counterparts in the uncomputable limit monad, and meet their proof obligations. The code is clean, short, correct and reasonably fast, and resides in a directory named `flops2012` at `https://github.com/ntoronto/plt-stuff/`.

7.8 Related Work

O'Connor's completion monad [37] is quite similar to the limit monad. Both operate on general metric spaces and compute to arbitrary precision. O'Connor starts with computable approximations and completes them using a monad. Implementing it in Coq took five months. It is certainly correct.

We start with a monad for exact values and define a computable replacement. It was two weeks from conception to implementation. Between directly using well-known theorems, and deriving the computable monad from the uncomputable monad without switching languages, we are as certain as we can be without mechanically verifying it. We have found our middle ground.

Higher-order logics such as HOL [30], CIC [8], MT [7] (Map Theory) and EFL* [15] continue Church's programme to found mathematics on the lambda calculus. Like λ_{ZFC}, interpreting them in set theory seems to require a slightly stronger theory than plain ZFC. HOL and CIC ensure consistency using types, and use the Curry-Howard correspondence to extract programs.

MT and EFL* are more like λ_{ZFC} in that they are untyped. MT ensures consistency partly by making nontermination a truth value, and EFL* partly by tagging propositions. Both support classical reasoning. MT and EFL* are interpreted in set theory using a straightforward extension of Scott-style denotational semantics to κ-sized domains, while λ_{ZFC} is interpreted in set theory using a straightforward extension of operational semantics to κ-sized relations.

The key difference between λ_{ZFC} and these higher-order logics is that λ_{ZFC} is not a logic. It is a programming language with infinite terms, which by design includes a transitive model of set theory (Theorem 7.12). Therefore, ZFC theorems can be applied to its set-valued terms with only trivial interpretation, whereas the interpretation it takes to apply ZFC theorems to lambda terms that represent sets in MT or EFL* can be highly nontrivial. Applying a ZFC theorem in HOL or CIC requires re-proving it to the satisfaction of a type checker.

The infinitary lambda calculus [25] has “infinitely deep” terms. Although it exists for investigating laziness, cyclic data, and undefinedness in finitary languages, it is possible to encode uncomputable mathematics in it. In λ_{ZFC}, such up-front encodings are unnecessary.

Hypercomputation [38] describes many Turing machine extensions, including completion of transfinite computations. Much of the research is for discovering the properties of computation in physically plausible extensions. While λ_{ZFC} might offer a civilized way to program such machines, we do not think of our work as hypercomputation, but as approaching computability from above.

7.9 Conclusions and Future Work

We defined λ_{ZFC}, which can express essentially anything constructible in contemporary mathematics, in a way that makes it compatible with existing first-order theorems. We demonstrated that it makes deriving computational meaning easier by defining the limit monad in it, deriving a computable replacement, and computing real numbers to arbitrary accuracy with acceptable speed.

Our main future work is using λ_{ZFC} to define languages for Bayesian inference, then deriving implementations that compute converging probabilities. We have already done this successfully for countable Bayesian models [45]. Having built our previous work’s foundation, we can now proceed with it.

Overall, we no longer have to hold back when a set-theoretic construction could be defined elegantly with untyped lambdas or recursion, or generalized precisely with higher-order functions. If we can derive a computable replacement, we might help someone in Cantor’s Paradise compute the apparently uncomputable.

References

[1] Haskell 98 language and libraries, the revised report, December 2002. URL `http://www.haskell.org/onlinereport/`.

[2] Stephen Abbott. *Understanding Analysis*. Springer, 2001.

[3] Peter Aczel. An introduction to inductive definitions. *Studies in Logic and the Foundations of Mathematics*, 90:739–782, 1977.

[4] G. Amato and F. Scozzari. The abstract domain of parallelotopes. *Electronic Notes in Theoretical Computer Science*, 287:17–28, November 2012.

[5] Robert J. Aumann. Borel structures for function spaces. *Illinois Journal of Mathematics*, 5:614–630, 1961.

[6] Bruno Barras. Sets in Coq, Coq in sets. *Journal of Formalized Reasoning*, 3(1), 2010.

[7] C. Berline and K. Grue. A κ-denotational semantics for Map Theory in ZFC+SI. *Theoretical Computer Science*, 179(1–2):137–202, 1997.

[8] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004. URL `http://www.labri.fr/publications/l3a/2004/BC04`.

[9] Sooraj Bhat, Johannes Borgström, Andrew D. Gordon, and Claudio Russo. Deriving probability density functions from probabilistic functional programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2013.

[10] Keith A Bonawitz. *Composable Probabilistic Inference with Blaise*. PhD thesis, Massachusetts Institute of Technology, 2008.

[11] Johannes Borgström, Andrew D. Gordon, Michael Greenberg, James Margetson, and Jurgen Van Gael. Measure transformer semantics for Bayesian machine learning. In *European Symposium on Programming*, pages 77–96, 2011.

[12] Manuel M. T. Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. Associated types with class. In *Principles of Programming Languages*, pages 1–13, 2005.

[13] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages*, pages 238–252, 1977.

[14] M.H. DeGroot and M.J. Schervish. *Probability and Statistics*. Addison Wesley Publishing Company, Inc., 2012. ISBN 9780321500465.

[15] R. C. Flagg and J. Myhill. A type-free system extending ZFC. *Annals of Pure and Applied Logic*, 43:79–97, 1989.

[16] Matthew Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Inc., 2010. `http://racket-lang.org/tr1/`.

[17] Noah Goodman, Vikash Mansinghka, Daniel Roy, Keith Bonawitz, and Joshua Tenenbaum. Church: a language for generative models. In *Uncertainty in Artificial Intelligence*, 2008.

[18] Michael Hanus. Multi-paradigm declarative languages. In *Logic Programming*, pages 45–75. 2007.

[19] Martin Hofmann, Benjamin C. Pierce, , and Daniel Wagner. Edit lenses. In *Principles of Programming Languages*, 2012.

[20] K. Hrbacek and T.J. Jech. *Introduction to set theory*. Pure and Applied Mathematics. M. Dekker, 1999.

[21] John Hughes. Generalizing monads to arrows. In *Science of Computer Programming*, volume 37, pages 67–111, 2000.

[22] Joe Hurd. *Formal Verification of Probabilistic Algorithms*. PhD thesis, University of Cambridge, 2002.

[23] Claire Jones. *Probabilistic Non-Determinism*. PhD thesis, Univ. of Edinburgh, 1990.

[24] R. B. Kearfott. Interval computations: Introduction, uses, and resources. *Euromath Bulletin*, 2:95–112, 1996.

[25] Richard Kennaway, Jan Willem Klop, M. Ronan Sleep, and Fer jan De Vries. Infinitary lambda calculus. *Theoretical Computer Science*, 175:93–125, 1997.

[26] Oleg Kiselyov and Chung-chieh Shan. Monolingual probabilistic programming using generalized coroutines. In *Uncertainty in Artificial Intelligence*, 2008.

[27] Achim Klenke. *Probability Theory: A Comprehensive Course.* Springer, 2006. ISBN 978-1-84800-047-6.

[28] Daphne Koller, David McAllester, and Avi Pfeffer. Effective Bayesian inference for stochastic programs. In *14th National Conference on Artificial Intelligence*, August 1997.

[29] Dexter Kozen. Semantics of probabilistic programs. In *Foundations of Computer Science*, 1979.

[30] Daniel Leivant. Higher order logic. In *In Handbook of Logic in Artificial Intelligence and Logic Programming*, pages 229–321. Clarendon Press, 1994.

[31] Sam Lindley, Philip Wadler, and Jeremy Yallop. Idioms are oblivious, arrows are meticulous, monads are promiscuous. *Electronic Notes in Theoretical Computer Science*, 2008.

[32] Sam Lindley, Philip Wadler, and Jeremy Yallop. The arrow calculus. *Journal of Functional Programming*, 20:51–69, 2010.

[33] David J. Lunn, Andrew Thomas, Nicky Best, and David Spiegelhalter. WinBUGS – a Bayesian modelling framework. *Statistics and Computing*, 10(4), 2000.

[34] Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(1), 2008.

[35] Brian Milch, Bhaskara Marthi, Stuart Russell, David Sontag, Daniel Ong, and Andrey Kolobov. BLOG: Probabilistic models with unknown objects. In *International Joint Conference on Artificial Intelligence*, 2005.

[36] James R. Munkres. *Topology.* Prentice Hall, second edition, 2000.

[37] Russell O'Connor. Certified exact transcendental real number computation in Coq. In *TPHOLs'08*, pages 246–261, 2008.

[38] Toby Ord. The many forms of hypercomputation. *Applied Mathematics and Computation*, 178:143–153, 2006.

[39] Sungwoo Park, Frank Pfenning, and Sebastian Thrun. A probabilistic language based upon sampling functions. *Transactions on Programming Languages and Systems*, 31(1), 2008.

[40] L. C. Paulson. Set theory for verification: II. Induction and recursion. *Journal of Automated Reasoning*, 15:167–215, 1995.

[41] Lawrence C. Paulson. Set theory for verification: I. From foundations to functions. *Journal of Automated Reasoning*, 11:353–389, 1993.

[42] Avi Pfeffer. The design and implementation of IBAL: A general-purpose probabilistic language. In *Statistical Relational Learning*. MIT Press, 2007.

[43] Norman Ramsey and Avi Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *Principles of Programming Languages*, 2002.

[44] Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of typed Scheme. In *Principles of Programming Languages*, pages 395–406, 2008.

[45] Neil Toronto and Jay McCarthy. From Bayesian notation to pure Racket, via measure-theoretic probability in λ_{ZFC}. In *Impl. and Appl. of Functional Languages*, 2010.

[46] Neil Toronto and Jay McCarthy. Computing in Cantor's paradise with λ_{ZFC}. In *Functional and Logic Programming Symposium*, pages 290–306, 2012.

[47] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. In *Proceedings of the London Mathematical Society*, volume 42, pages 230–265, 1936.

[48] Athanassios Tzouvaras. Cardinality without enumeration. *Studia Logica: An International Journal for Symbolic Logic*, 80(1):121–141, June 2005.

[49] Gabriel Uzquiano. Models of second-order Zermelo set theory. *The Bulletin of Symbolic Logic*, 5(3):289–302, 1999.

[50] Eric Veach and Leonidas J. Guibas. Metropolis light transport. In *ACM SIGGRAPH*, pages 65–76, 1997.

[51] Philip Wadler. Monads for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*. 2001.

[52] Benjamin Werner. Sets in types, types in sets. In *TACS'97*, pages 530–546, 1997.

[53] David Wingate, Noah D. Goodman, Andreas Stuhlmüller, and Jeffrey M. Siskind. Nonstandard interpretations of probabilistic programs for efficient inference. In *Neural Information Processing Systems*, pages 1152–1160, 2011.

[54] David Wingate, Andreas Stuhlmüller, and Noah D. Goodman. Lightweight implementations of probabilistic programming languages via transformational compilation. In *Artificial Intelligence and Statistics*, 2011.