# 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

March 2014

## ABSTRACT

Trustworthy, Useful Languages for Probabilistic Modeling and Inference

Neil Toronto
Department of Computer Science, BYU
Doctor of Philosophy

The ideals of exact modeling, and of putting off approximations as long as possible, make Bayesian practice both successful and difficult. Automating it has been taking the form of languages for modeling probabilistic processes and implementations that answer questions about them.

Unfortunately, very few of these languages have a mathematical specification. This makes them difficult to trust: there is no way to distinguish between a bug and a feature, and there is no standard by which to prove optimizations correct. Further, because the languages are based on the incomplete theories of probability typically used in Bayesian practice, they place seemingly artificial restrictions on legal programs and queries, such as disallowing unbounded recursion and allowing only simple equality constraints.

We prove it is possible to make trustworthy probabilistic languages for Bayesian practice by using functional programming theory to define them mathematically and prove them correct. The specifications interpret programs using the most complete known theory of probability, so they place no restrictions on programs and allow arbitrary conditions with arbitrarily low probability.

We demonstrate that the results are useful by implementing the languages, and using them to model and answer questions about typical probabilistic processes. We also model and answer questions about processes that are either difficult or impossible to reason about precisely using Bayesian mathematical tools.

Keywords: Bayesian, Probability, Domain-Specific Languages, Functional Programming, Semantics

To my wife, Amy.

## ACKNOWLEDGMENTS

I must first thank my parents for bringing me up right.

I must secondly thank my parents again, for encouraging my curiosity and intellectual development. Only now do I hear stories of the difficulties they had finding age-appropriate computer accessories and programming books for my Christmas presents—in the mid-1980s, when such things were scarce and few were made for children.

I must thirdly thank my parents yet again, for trying to instill in me a love of hard work. It eventually took. I assure them it was worth it.

My PhD advisor, Jay McCarthy, has been stellar in every respect. He has shown me how to do programming language research with rigor, taught me how to structure my communication, and offered excellent insights and suggestions on research that crosses into a field he initially knew little about. Without Jay's guidance, there would be little to distinguish this work from other work in probabilistic languages. Together, we have done it in a way that is the most complete and reliable.

I cannot confine my thanks only to Jay's professional instruction. He has also ensured that I could take care of my family without going into debt, and ensured that I have the connections necessary to secure a good career. The wait for "when Daddy gets a real job" has been bearable, with no ever-increasing financial load and with good prospects to look forward to. The positive repurcussions will be nearly endless. I am eternally grateful.

My wife has constantly supported my academic pursuits for 11 years. This is truly marvellous in a way that words are too narrow to contain.

I thank my children, for praying for me every day, and for being sweet, loving, talented, and so very interesting.

I thank my God and Savior, whose justice and mercy makes life meaningful and beautiful.

# 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

*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

## 1.1 Introduction

Probability is notorious for being counterintuitive. Ask anyone who is wrestling with the birthday paradox, the Monty Hall problem, or the Bayesian phenomenon *explaining away.*

Probability's habit of violating intuition makes any automation of probabilistic reasoning helpful. In Bayesian statistics, automation has been taking the form of modeling languages for probabilistic processes. The languages' implementations compute answers to questions about the processes under constraints.

Probabilistic languages should have mathematical specifications. The reason is simple: if a probabilistic language is implemented to be faithful to its maker's intuitions instead of a specification, it is almost certainly faulty.

Unfortunately, there are currently few probabilistic languages that have mathematical specifications. This state of affairs is partly responsible for another: until now, every probabilistic language that can support Bayesian practice is artificially limited in what it can express. Most commonly, probabilistic languages disallow unbounded loops and recursion, allow only discrete or continuous distributions, and restrict constraints to the form $X = c$.

The thesis statement is essentially that these states of affairs need not continue.

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

## 1.2 Terms

To **model** something is to make it into a model of a theory, by developing the theory. For example, physicists model gravity by developing theories of gravitation; the physical phenomenon is a model of the theories. Likewise, Bayesians model probabilistic processes by developing probabilistic theories for which the physical processes are models. When there are mathematical models of theories, the mathematical models can be used to predict the physical models' behavior and discover their properties.

Bayesians write theories in many ways. One is to write them **constructively**: in such a way that the theory contains enough information to directly construct one of its mathematical models. This is often regarded as the ideal way to write them.

**Inference** means answering questions about theories. In this context, it implies **conditioning**: constraining the model in a way that preserves certain relative probabilities.

**Measure-theoretic probability** [27] is the most successful theory of probability in precision, maturity, and explanatory power. It was first developed in the early 1900s to formalize intuitive ideas about probability, to unify notions of discrete and continuous random variables, and to settle paradoxes that arise from incorrectly applying intuition to infinities.

**Functional programming theory** is used to give mathematically precise meaning to programs and to give rules for executing them. In it, the $\lambda$-calculus serves as a model of computation and as a minimal language in which to reason by substitution.

A **trustworthy** language has a mathematical meaning called a **semantics**. By defining a language mathematically, it is possible to prove theorems about it, which apply to all faithful implementations. Further, if a language implementation computes something unexpected, its semantics provides a way to determine whether its behavior is correct.

We generally think of languages as being **useful** when they save time by automating calculations. Languages are also useful when they allow us to express ideas naturally and reason about them precisely, or provide abstraction mechanisms so we can express ideas and reason about them at high levels.

## 1.3 Proof and Supporting Evidence

All but usefulness in the thesis can be proved, and for usefulness, we give evidence. To prove and demonstrate the thesis, we define two semantics, prove them correct, implement approximations of them, and test the implementations.

The first semantics is an initial investigation into our general approach: transforming Bayesian theories into $\lambda$-calculus terms that build exact measure-theoretic models of the theories, and then changing the transformation to build approximate models to carry out computations. To keep the investigation simple, we restrict theories to countable probability distributions and finitely many statements.

We ensure the first language is trustworthy by deriving its semantics from an idealized expected meaning of Bayesian theories, and computing answers to queries from approximate models in a way that converges to the correct answers according to the exact models. We demonstrate the language is useful by implementing the approximating transformation, and encoding theories and running queries that are difficult to model directly without it.

The second semantics handles uncountable probability distributions and recursion by transforming a first-order functional language with probabilistic choice into $\lambda$-calculus terms that build models. Again, we change the transformation to build approximate models and use them to carry out computations.

We show the language is trustworthy by proving

- Exact queries always terminate with correct answers (Theorem 7.51).
- All probabilistic programs have sensible output distributions, regardless of nontermination (Theorems 7.52 and 7.53).

- The approximations are sound, always terminate, and have other desirable properties (Theorems 7.58 through 7.61).
- Answers computed using the approximations correctly converge (Theorem **??**).

Further, Theorems 7.52 and 7.53 apply to any probabilistic programming language that can be transformed into ours. Because ours is Turing-equivalent (with a random oracle) and is easy to extend with uncomputable operations such as real limits and decidable equality, this includes all probabilistic programming languages to date, and likely almost all future probabilistic programming languages.

We demonstrate the second language is useful by implementing the approximating transformation and encoding some typical Bayesian theories and running queries. In all of our tests, the theory encodings are straightforward and the queries are efficient.

To demonstrate further usefulness, we encode theories and run queries that are impossible to reason about precisely using typical Bayesian mathematical tools. One example draws inferences from a correctly modeled thermometer. Another is a simple, direct theory of light transport and a query that together carry out stochastic ray tracing.

## 1.4 Exposition Transition System

While this work is designed to make sense when read straight through, readers may skip some depending on their goals.

In principle, answers to questions such as "Which chapters should I read if I am only interested in implementations of probabilistic languages with conditioning and recursion?" can be answered using a dependency graph. However, the graph would be a mess of arrows: for example, everything after Chapter 2 (Background) depends on Chapter 2; similarly for Chapter 4 (Using $\lambda_{\text{ZFC}}$). Figure 1.1 shows an alternative: a transition system on chapters for which following any path (with backtracking permitted) guarantees a reader will not miss out on prerequisites.

Chapter 2 gives the necessary background in Bayesian practice and functional pro-

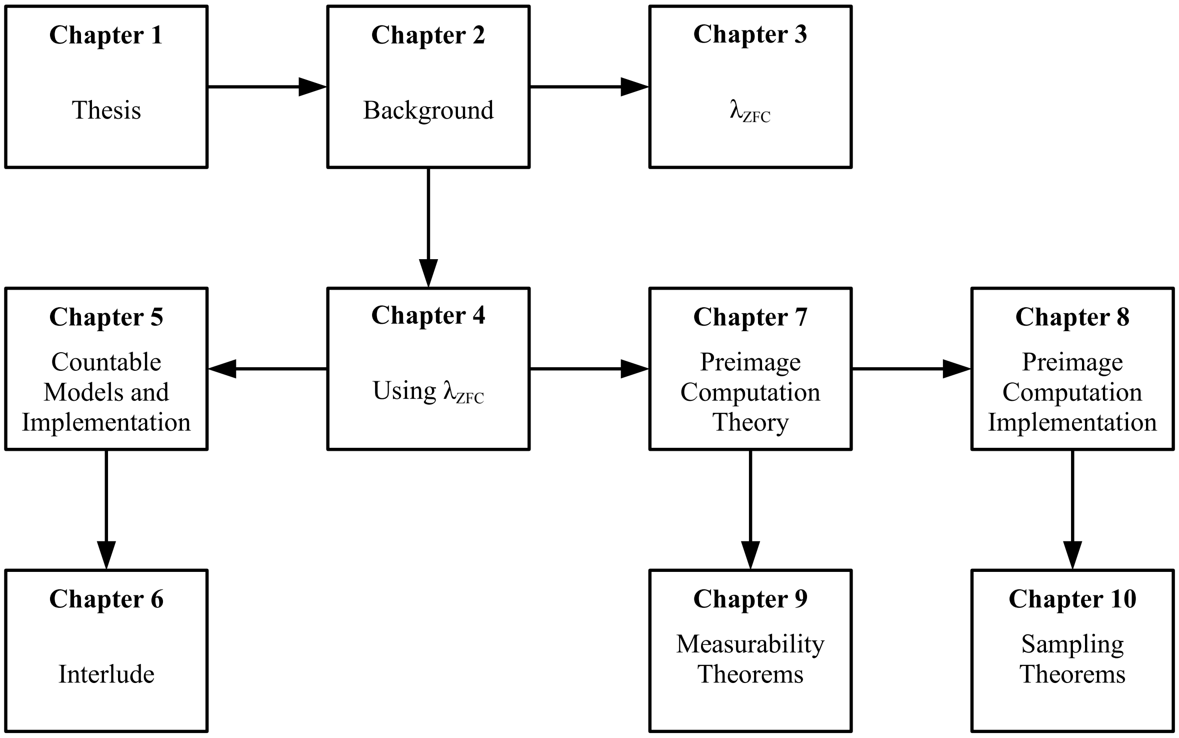


Figure 1.1: A transition system showing possible paths through this dissertation.

gramming theory, and motivates using measure theory.

The two semantics mentioned in the preceeding section transform programs into $\lambda$-calculus terms. This target language has three requirements that are unusual for a $\lambda$-calculus: it must be able to represent infinite objects and operations on them, it must have nonterminating programs, and measure-theoretic theorems must apply directly to its terms. Before this work, such a $\lambda$-calculus did not exist. Chapter 3 defines one, $\lambda_{\text{ZFC}}$, with the precision necessary to carry out proofs with it.

While this precision is necessary for doing the rest of our work and verifying it, such precision is not necessary for understanding it. Readers who are not verifying our work may therefore skip from Chapter 2 to Chapter 4, which gives an overview of $\lambda_{\text{ZFC}}$ and its relationship with contemporary mathematics, gives examples of use, and defines some common terminology and functions.

Chapter 5 defines a semantics for Bayesian notation restricted to countable probability distributions and finitely many statements. Chapter 6 explains why its specific way of transforming notation into models does not extend easily to theories with recursion, which motivates a slight change in tactics.

Following the new tactics, Chapter 7 defines a semantics for a probabilistic language with uncountable distributions, recursion, and arbitrary probabilistic conditions. Chapter 8 gives details that should be common to all implementations, and details specific to ours.

Chapter 9 contains proofs of theorems critical to correctness, but whose inclusion in Chapter 7 would interrupt the narrative flow too much for readers unfamiliar with measure theory. Chapter 10 is similar, but contains proofs of theorems from Chapter 8. While familiarity with measure theory is helpful while reading these two chapters, it is not strictly necessary: both explain the necessary concepts, and import enough definitions and lemmas from other sources to verify the proofs.

# Chapter 2

## Background

Our work bridges Bayesian practice and functional programming theory using measure-theoretic probability.

In this chapter, we attempt to give enough background to motivate and explain Bayesian practice and functional programming theory. We only *motivate* measure-theoretic probability here. Further on, we introduce, explain and use parts as we need them.

It is difficult to find two areas in computer science as different as Bayesian practice and functional programming theory. In Bayesian practice, we find deeply held belief that unknowns can and should be *quantified* (by probabilities), reasoning by probabilistic inference, willingness to accept many kinds of approximations, and common notation that is—to put it kindly—flexible. In functional programming theory, we find deeply held belief that unknowns should be *qualified* (usually universally), reasoning by logical inference, little tolerance for unsound approximations even if they converge, and common notation that is—to put it kindly—almost precise enough to feed a compiler.

There is one common trait that makes bridging both areas even conceivable. While Bayesians model processes and functional programming researchers model languages, both approach their tasks methodically, and both create theories in which every entity they want to reason about is represented explicitly. If something important is going on behind the scenes—whether a hidden Markov process or mutating the program's store—it is brought to the fore and fully characterized. In both areas, the extra time and cognitive burden are considered worthwhile payment for reliable artifacts and repeatable results.

It is this trait, explicit representation, that makes it possible to automate Bayesian inference, and again this trait that makes it possible to prove the automation correct.

## 2.1 Bayesian Practice

From Bayesian practice, the requisite background knowledge includes probability mass functions, probability densities, queries and manipulation rules, Bayesian modeling, and Bayesian inference. We assume readers know arithmetic, some set theory, functions, and the basic ideas behind integration.

### 2.1.1 Discrete Probability and Joint Distribution Models

In a probabilistic model of a real-world process, distinguished identifiers called **random variables** denote random values. These are regarded as free variables, but with additional information that quantifies the likelihoods of every *combination* of their values, or **observable outcomes**. The additional information is completely characterized by a function called a **joint distribution**.

For example, suppose $X, Y \in \{h, t\}$ are random variables that each represent the outcome of a coin toss. Further, let the joint distribution $p_{X,Y} : \{h,t\} \times \{h,t\} \to [0,1]$ quantify the likelihood of every possible combination of observable outcomes by defining

$$p_{X,Y} \;=\; \left[(h,h) \mapsto \frac{1}{4}, (h,t) \mapsto \frac{1}{4}, (t,h) \mapsto \frac{1}{6}, (t,t) \mapsto \frac{1}{3}\right] \tag{2.1}$$

(The subscript "$X, Y$" is just part of the function name, and has no special meaning.) This is a **probability mass function**: its outputs sum to 1.

The probability that $X = t$ and $Y = h$ is $p_{X,Y}(t,h) = \frac{1}{6}$. Another way to write "the probability that $X = t$ and $Y = h$" is with a **probability query**:

$$\Pr[X = t \wedge Y = h] \;=\; p_{X,Y}(t,h) \tag{2.2}$$

A probability query always implicitly refers to some ambient joint distribution. In general, the result of a probability query $\Pr[e]$ is the sum of probabilities of observable outcomes for which the proposition $e$ is true. Conjunctions are often separated by a comma; e.g. $\Pr[X = x, Y = y]$ means $\Pr[X = x \wedge Y = y]$.

Probability queries have manipulation rules. One is that random variables may be "summed out" to consider the probabilities of the values of others independently. For example, to consider just the probabilities of values of $X$, we may sum out $Y$:

$$\Pr[X = x] \;=\; \sum_{y \in \{h,t\}} \Pr[X = x, Y = y] \tag{2.3}$$

According to this rule, $X$ represents the outcome of a fair coin toss (independent of $Y$):

$$\begin{aligned} \Pr[X = h] &= \Pr[X = h, Y = h] + \Pr[X = h, Y = t] &&= \frac{1}{4} + \frac{1}{4} &&= \frac{1}{2} \\ \Pr[X = t] &= \Pr[X = t, Y = h] + \Pr[X = t, Y = t] &&= \frac{1}{6} + \frac{1}{3} &&= \frac{1}{2} \end{aligned} \tag{2.4}$$

Another manipulation rule allows fixing the value of one random variable to consider the probabilites of the values of others *dependently.* For example, if $x$ is fixed and $\Pr[X = x] > 0$, then the probability that $Y = y$ is

$$\Pr[Y = y \,|\, X = x] \;=\; \frac{\Pr[X = x, Y = y]}{\Pr[X = x]} \tag{2.5}$$

This **conditional probability** query is read "the probability that $Y = y$ given $X = x$." Using this rule, we can determine that, if $X$ is known to be $t$, then $Y$ represents a coin toss that is not fair:

$$\begin{aligned} \Pr[Y = h \,|\, X = t] &= \frac{\Pr[X = t, Y = h]}{\Pr[X = t]} &&= \frac{\frac{1}{6}}{\frac{1}{2}} &&= \frac{1}{3} \\ \Pr[Y = t \,|\, X = t] &= \frac{\Pr[X = t, Y = t]}{\Pr[X = t]} &&= \frac{\frac{1}{3}}{\frac{1}{2}} &&= \frac{2}{3} \end{aligned} \tag{2.6}$$

We could similarly show $\Pr[Y = h \,|\, X = h] = \frac{1}{2}$ and $\Pr[Y = t \,|\, X = h] = \frac{1}{2}$, so when $X$ is known to be $h$, $Y$ represents a fair coin toss.

To avoid the condition $\Pr[X = x] > 0$, the preceeding rule is often written

$$\begin{aligned} \Pr[X = x, Y = y] &= \Pr[X = x] \cdot \Pr[Y = y \,|\, X = x] \\ &= \Pr[Y = y] \cdot \Pr[X = x \,|\, Y = y] \end{aligned} \tag{2.7}$$

In this form, it is called the **chain rule**.

As a function of $x$, $\Pr[X = x]$ is a probability mass function, but over just $X$ instead of $X$ and $Y$ together. With any fixed $x$, as a function of $y$, $\Pr[Y = y \,|\, X = x]$ is also a probability mass function. Most Bayesian models are constructed by reifying these queries as functions called (respectively) **distributions** and **conditional distributions**, and using the chain rule to build a joint distribution. For the present example,

$$\begin{aligned} p_X &= \left[h \mapsto \frac{1}{2}, t \mapsto \frac{1}{2}\right] \\ p_{Y|X}(y \,|\, x) &= \begin{cases} x = h & \left[h \mapsto \frac{1}{2}, t \mapsto \frac{1}{2}\right](y) \\ x = t & \left[h \mapsto \frac{1}{3}, t \mapsto \frac{2}{3}\right](y) \end{cases} \\ p_{X,Y}(x, y) &= p_X(x) \cdot p_{Y|X}(y \,|\, x) \end{aligned} \tag{2.8}$$

In the conditional distribution $p_{Y|X}$, the "$Y|X$" subscript is simply part of the function name, and in applying it, $(y \,|\, x)$ is just another way to write the arguments $(y, x)$.[1] The syntax simply connotes that we expect $\Pr[Y = y \,|\, X = x] \;=\; p_{Y|X}(y \,|\, x)$.

This model can be more compactly specified by a constructive theory about $X$ and $Y$, which states only the properties that a joint distribution model must satisfy:

$$\begin{aligned} X &\sim \left[h \mapsto \frac{1}{2}, t \mapsto \frac{1}{2}\right] \\ Y &\sim \begin{cases} X = h & \left[h \mapsto \frac{1}{2}, t \mapsto \frac{1}{2}\right] \\ X = t & \left[h \mapsto \frac{1}{3}, t \mapsto \frac{2}{3}\right] \end{cases} \end{aligned} \tag{2.9}$$

Here, $X \sim e$ is read "$X$ is distributed $e$." In this leaner form, it is perhaps easier to understand

[1]It is common to leave off subscripts such as $Y|X$ and use the form of application to distinguish the different $p$s; e.g. $p(x)$ means $p_X(x)$ and $p(y \,|\, x)$ means $p_{Y|X}(y \,|\, x)$. Doing so is helpful when there are many random variables, and it is usually unambiguous, but the practice often confuses and frustrates newcomers.

the process being modeled, which is

1. Toss a coin and call its outcome $X$.
2. If $X = h$, toss a fair coin and call its outcome $Y$.
3. If $X = t$, toss a biased coin with heads probability $\frac{1}{3}$ and call its outcome $Y$.

It is usually easy to manually translate constructive theories into programs that sample random variable values.

By combining a conditional probability query and the chain rule (or using the chain rule twice), we get **Bayes' law**: if $\Pr[Y = y] > 0$, then

$$\Pr[X = x \,|\, Y = y] \;=\; \frac{\Pr[X = x] \cdot \Pr[Y = y \,|\, X = x]}{\Pr[Y = y]} \tag{2.10}$$

This is different than $\Pr[Y = y \,|\, X = x]$. This time, we are interested in the conditional probability that $X = x$ given we know $Y = y$ for some fixed $y$.

For example, suppose we did not observe $X$, but were allowed to observe $Y = t$. Given that we know the second coin toss is tails, the probability that $X = t$ is

$$\begin{aligned}\Pr[X = t \,|\, Y = t] \;&=\; \frac{\Pr[X = t] \cdot \Pr[Y = t \,|\, X = t]}{\Pr[Y = t]} \\ &=\; \frac{\frac{1}{2} \cdot \frac{2}{3}}{\sum_{x \in \{h,t\}} \Pr[X = x, Y = t]} \\ &=\; \frac{\frac{1}{2} \cdot \frac{2}{3}}{\frac{1}{4} + \frac{1}{3}} \;=\; \frac{\frac{1}{3}}{\frac{7}{12}} \;=\; \frac{4}{7}\end{aligned} \tag{2.11}$$

which is greater than $\Pr[X = t] = \frac{1}{2}$. Similarly, $\Pr[X = h \,|\, Y = t] = \frac{3}{7}$, which is less than $\Pr[X = h] = \frac{1}{2}$. Observing the effects of the first coin toss—even random effects—allows us to draw stronger conclusions about the first coin toss. In this case, we know it is more likely to be tails.

Using Bayes' law to draw probabilistic conclusions about probabilistic processes given observed probabilistic effects is called **Bayesian inference**.

It is easy for probability newcomers with logical background to think of conditional queries as a fuzzy kind of logical implication. A short example demonstrates that doing so leads to faulty intuition. To compute the probability that $Y = t \implies X = t$, we apply logical rules to the query until it is a disjunction of distinct observable outcomes, and add their probabilities:

$$\begin{aligned}
\Pr[Y = t \implies X = t] &= \Pr[\neg(Y = t) \vee X = t] \\
&= \Pr[Y = h \vee X = t] \\
&= \Pr[(Y = h \wedge X = t) \vee (Y = h \wedge X = h) \vee (Y = t \wedge X = t)] \\
&= \Pr[Y = h \wedge X = t] + \Pr[Y = h \wedge X = h] + \Pr[Y = t \wedge X = t] \\
&= \frac{1}{6} + \frac{1}{4} + \frac{1}{3} = \frac{3}{4}
\end{aligned} \tag{2.12}$$

This is clearly not $\Pr[X = t \,|\, Y = t] = \frac{4}{7}$. In a similar fashion, $\Pr[Y = t \implies X = h] = \frac{2}{3}$, which is not $\Pr[X = h \,|\, Y = t] = \frac{3}{7}$.

In conditioning on $Y = t$, we did not consider any outcomes in which $Y \neq t$: we restricted the possible outcomes to those for which $Y = t$ and renormalized the probabilities of $X = h$ and $X = t$. On the other hand, in computing the probability that $Y = t \implies X = t$, we had to consider *all* of the outcomes in which $Y \neq t$, and only *one* outcome in which $Y = t$.

### 2.1.2 Probability Densities and Density Models

Probability mass functions cannot quantify the likelihoods of uncountably many observable outcomes, such as when $X \in \mathbb{R}$. In these cases, the distributions, conditional distributions, and joint distributions are specified using **probability density functions**: functions over outcomes that *integrate* to 1 instead of sum to 1.[2]

For example, this probability density function defines the **standard normal distribution**, or bell curve:

$$f_N(x) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2}\right) \tag{2.13}$$

[2]For readers familiar with measure theory: we use the word *density* for densities with respect to Lebesgue measure, and *mass* for densities with respect to counting measure. We call all other densities *derivatives*.

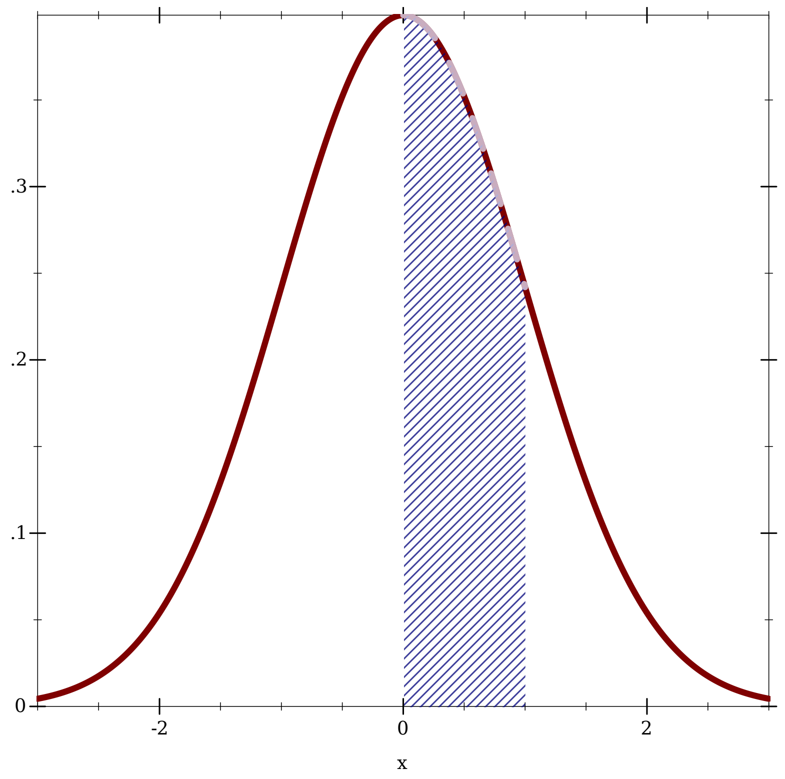


Figure 2.1: Integrating under the standard normal density to compute $\Pr[X \in (0,1)] \approx 0.34$.

If $X$ has a standard normal distribution, then the probability that $X \in (0,1)$ is

$$\Pr[X \in (0,1)] \;=\; \int_0^1 f_N(x)\,dx \;\approx\; 0.3413447460685 \tag{2.14}$$

Figure 2.1 plots this density and illustrates integrating under it to compute $\Pr[X \in (0,1)]$.

When probabilities are computed by integrating density functions, sets of outcomes may have positive probability, but every *single* outcome has zero probability. For any random variable $X \in \mathbb{R}$, outcome $x \in \mathbb{R}$, and density function $f_X : \mathbb{R} \to [0,+\infty)$,

$$\Pr[X = x] \;=\; \int_x^x f_X(x)\,dx \;=\; (f_X(x) - 0)\cdot(x - x) \;=\; f_X(x)\cdot 0 \;=\; 0 \tag{2.15}$$

As a consequence, interval endpoints do not matter; i.e. $\Pr[X \in [a,b]] \;=\; \Pr[X \in (a,b)]$. We discuss other, more difficult consequences further on.

The normal distribution can be extended to a **distribution family** by parameterizing

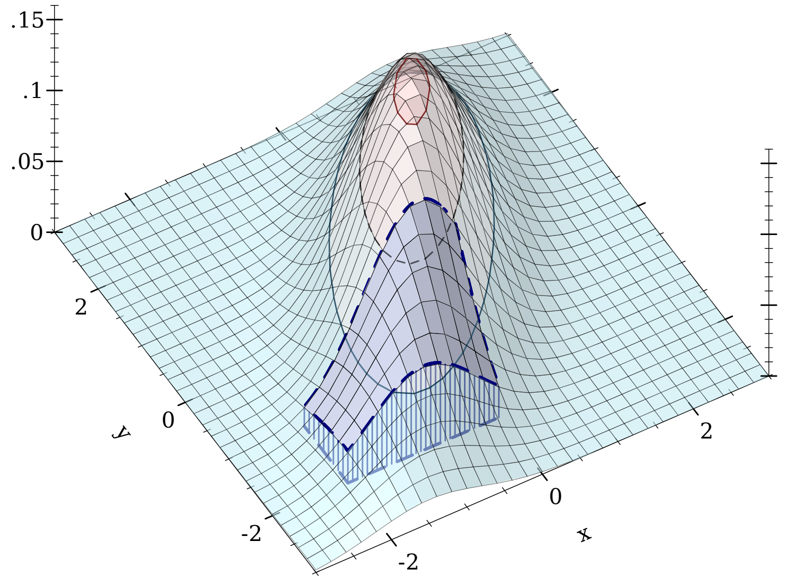


Figure 2.2: The joint density model $f_{X,Y}$ constructed from the density $f_X$ and the conditional density $f_{Y|X}$. Integrating under $f_{X,Y}$ on the set $(-2, 0) \times (-2, -1)$ computes $\Pr[X \in (-2, 0), Y \in (-2, -1)]$.

it on a *mean* $\mu$ and a *standard deviation* $\sigma$:

$$f_N(x \,|\, \mu, \sigma) \;=\; \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right) \tag{2.16}$$

(Again, the application syntax $(x \,|\, \mu, \sigma)$ simply means $(x, \mu, \sigma)$.) Using parameterized distributions, we can define a **joint density model** of a probabilistic process involving two random variables $X, Y \in \mathbb{R}$:

$$\begin{aligned} f_X(x) \;&=\; f_N(x \,|\, 0, 1) \\ f_{Y|X}(y \,|\, x) \;&=\; f_N(y \,|\, x, 1) \\ f_{X,Y}(x, y) \;&=\; f_X(x) \cdot f_{Y|X}(y \,|\, x) \end{aligned} \tag{2.17}$$

Integrating under $f_{X,Y}$ computes probability queries:

$$\Pr[X \in (a,b), Y \in (c,d)] \;=\; \int_a^b \int_c^d f_{X,Y}(x,y)\, dy\, dx \tag{2.18}$$

Figure 2.2 illustrates the joint density model $f_{X,Y}$ and computing a probability query.

As with discrete models, density models can be specified by constructive theories:

$$\begin{aligned} X &\sim \text{Normal}(0,1) \\ Y &\sim \text{Normal}(X,1) \end{aligned} \tag{2.19}$$

The probabilistic process being modeled is

1. Choose $X$ according to the standard normal distribution.
2. Choose $Y$ according to the normal distribution with mean $X$, standard deviation 1.

Again, it is usually easy to manually translate such theories into programs that sample random variable values.

When all single outcomes have zero probability, interpreting theories in terms of expected conditional probability queries is difficult. (In fact, doing so requires measure theory.) Fortunately, densities have rules analogous to rules for manipulating probability queries, which allow practitioners to derive joint density models from theories and compute a restricted class of conditional queries. Instances of the two most important rules are

$$\begin{aligned} f_X(x) &= \int_{-\infty}^{+\infty} f_{X,Y}(x,y)\, dy \\ f_{X,Y}(x,y) &= f_X(x) \cdot f_{Y|X}(y \,|\, x) \;=\; f_Y(y) \cdot f_{X|Y}(x \,|\, y) \end{aligned} \tag{2.20}$$

The second rule is the **chain rule for densities**, and it justifies constructing the joint density $f_{X,Y}$ from the density $f_X$ and the conditional density $f_{Y|X}$.

The rules for densities can be used to derive **Bayes' law for densities**: if $f_Y(y) > 0$, then, starting from the right-hand side of the chain rule,

$$\begin{aligned}
f_Y(y) \cdot f_{X|Y}(x \,|\, y) &= f_X(x) \cdot f_{Y|X}(y \,|\, x) \\
f_{X|Y}(x \,|\, y) &= \frac{f_X(x) \cdot f_{Y|X}(y \,|\, x)}{f_Y(y)} \\
&= \frac{f_X(x) \cdot f_{Y|X}(y \,|\, x)}{\displaystyle\int_{-\infty}^{+\infty} f_{X,Y}(x,y)\, dx} \\
&= \frac{f_X(x) \cdot f_{Y|X}(y \,|\, x)}{\displaystyle\int_{-\infty}^{+\infty} f_X(x) \cdot f_{Y|X}(y \,|\, x)\, dx}
\end{aligned} \tag{2.21}$$

The last form is conveniently in terms of $f_X$ and $f_{Y|X}$, which we have on-hand.

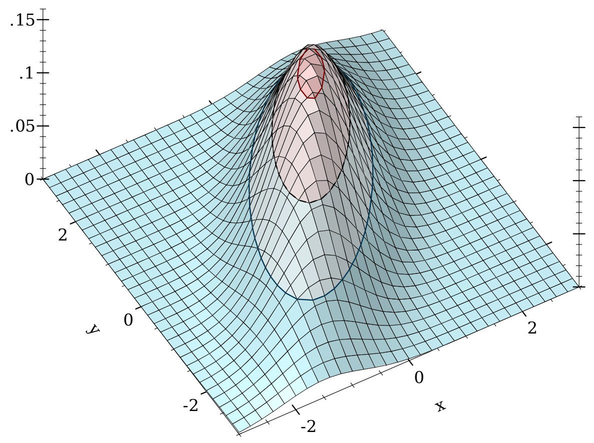
Using Bayes' law for densities, we can draw conclusions about $X$ given knowledge about $Y$. For example, suppose we want to know the distribution of $X$ given $Y = 2$, as a density. A quite lengthy derivation finally results in

$$\begin{aligned}
f_{X|Y}(x \,|\, 2) &= \frac{f_X(x) \cdot f_{Y|X}(2 \,|\, x)}{\displaystyle\int_{-\infty}^{+\infty} f_X(x) \cdot f_{Y|X}(2 \,|\, x)\, dx} \\
&= \frac{f_N(x \,|\, 0, 1) \cdot f_N(2 \,|\, x, 1)}{\displaystyle\int_{-\infty}^{+\infty} f_N(x \,|\, 0, 1) \cdot f_N(2 \,|\, x, 1)\, dx} \\
&\cdots \\
&= f_N(x \,|\, 1, \sqrt{\tfrac{1}{2}})
\end{aligned} \tag{2.22}$$

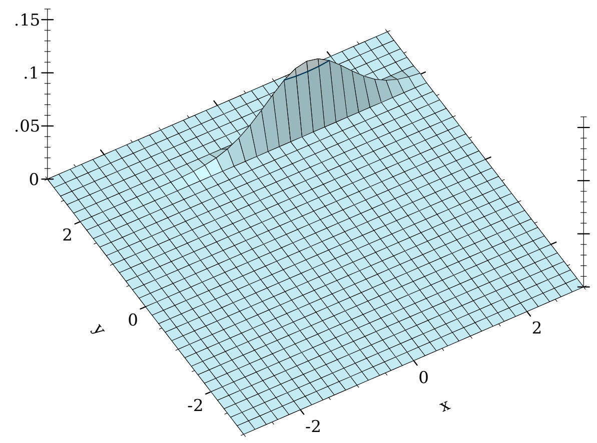
We can answer conditional probability queries such as "the probability that $X \in (a, b)$ given $Y = 2$" by integrating $f_{X|Y}(x \,|\, 2) \;=\; f_N(x \,|\, 1, \sqrt{\tfrac{1}{2}})$:

$$\Pr[X \in (a, b) \,|\, Y = 2] \;=\; \int_a^b f_N(x \,|\, 1, \sqrt{\tfrac{1}{2}})\, dx \tag{2.23}$$

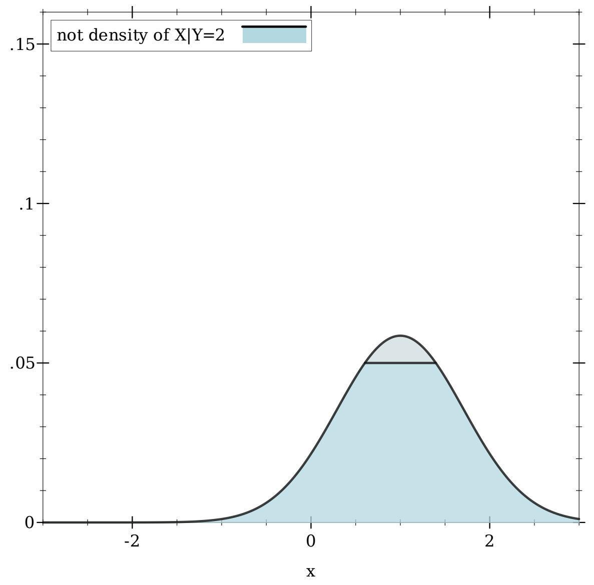
While using Bayes' law for densities is difficult, it is easy to visualize, at least in two dimensions. We may think of using it as happening in three steps: restrict, project, then renormalize. Figure 2.3 illustrates them. First, we restrict the joint density model (Figure 2.3a) to the subset of $\mathbb{R} \times \mathbb{R}$ where $y = 2$ (Figure 2.3b). Second, because the resulting model integrates to zero and therefore all answers to probability queries using it would be zero,



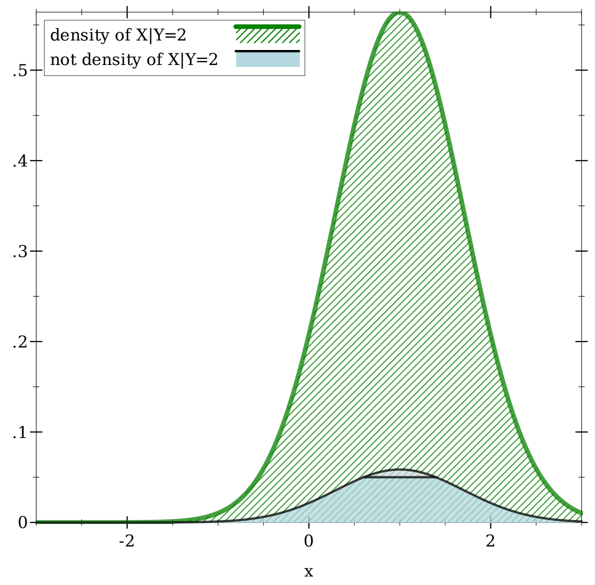
(a) The original joint density model.



(b) Restricting the model to the subset of its domain where $y = 2$. The probability of the subset is zero.



(c) Projecting the restricted model onto the $x$ axis results in a density that integrates to a nonzero constant that is less than 1.



(d) Normalizing the restricted, projected model results in a probability density that characterizes the distribution of $X$ given $Y = 2$.

Figure 2.3: Bayes' law for densities, in pictures.

we project it onto the $x$ axis (Figure 2.3c), on which it integrates to a positive value. Third, because the total probability is now less than 1, we renormalize the restricted, projected model by dividing its output by its area, and obtain a probability density (Figure 2.3d).

Using Bayes' law for densities is not only often technically challenging, but in general there are no closed-form solutions. In such cases, practitioners turn to Monte Carlo integration, or sampling, to answer conditional probability queries.

### 2.1.3 Motivating Measure-Theoretic Models

It is easy to define a random variable whose distribution cannot be characterized by a density. Suppose we have a thermometer whose output is not quite correct, but is usually within 1 degree. We could model the error with a normal distribution with standard deviation 1. If the thermometer cannot show a number greater than 100 and we want to model that fact, assuming the correct temperature is 99, we could write a theory about it like this:

$$\begin{aligned} T' &\sim \text{Normal}(99, 1) \\ T &= \min(T', 100) \end{aligned} \tag{2.24}$$

so that $T$ represents the thermometer's output. Because $T' \geq 100$ if and only if $T = 100$, $\Pr[T' \geq 100] = \Pr[T = 100] > 0$. But we know that if $T$ has a density, $\Pr[T = 100] = 0$.

Even though $T$ has no density, it must have a distribution, because $T \in (a, b)$ for any interval $(a, b)$ has a sensible probability, which we can compute by integrating the density of $T'$ up to 100 and adding $\Pr[T' \geq 100]$ if the interval happens to contain 100:

$$\Pr[T \in (a,b)] \;=\; \int_{\min(a,100)}^{\min(b,100)} f_N(x \mid 99, 1)\, dx + \begin{cases} \Pr[T' \geq 100] & \text{if } 100 \in (a,b) \\ 0 & \text{if } 100 \notin (a,b) \end{cases} \tag{2.25}$$

Further, it is easy to write a program that samples $T$ values, and we expect the outputs of programs with probabilistic choice to have well-defined probability distributions.

Another way to define a distribution that cannot be characterized by a density is to restrict a joint density model with a non-axial condition. For example, with the present theory and density model for $X, Y \in \mathbb{R}$, suppose we want to answer questions like

$$\Pr\Big[X \in (a,b), Y \in (c,d) \,|\, \sqrt{X^2+Y^2} = 1\Big] \tag{2.26}$$

which restricts the model to a unit circle. To do so with a density, we need $f_{X,Y|\sqrt{X^2+Y^2}}(x, y \,|\, z)$. But for any $z$, a joint density for $X, Y$ restricted to $\sqrt{X^2+Y^2} = z$ cannot exist because the area of that circular set is zero.

In the preceeding examples, on a part of the distribution's domain that has length or area 0 (i.e. the single outcome 100 or the set $\{x \in \mathbb{R}, y \in \mathbb{R} \mid \sqrt{x^2+x^2} = z\}$), the probability of that set is nonzero. Integrating on such a set can yield only 0, so the distributions cannot be defined by densities.

There are many more sensible distributions that cannot be defined by densities, such as the distribution of a random variable with outcomes in $\mathbb{R} \cup \mathbb{R}^2$ or $\mathbb{R}^{\mathbb{N}}$. Not only can sets of such outcomes have sensible probabilities, but as in the thermometer example, *it is easy to define random variables with these distributions.*

Measure theory's answer to this shocking lack of densities is to define probability distributions using **probability measures**, which map *sets to probabilities* instead of *values to instantaneous changes* in probability. The measure defining $T$'s distribution directly answers queries such as $\Pr[T \in (a,b)]$. There is also a measure defined on sets of $\mathbb{R} \times \mathbb{R}$ that directly answers queries such as $\Pr\Big[X \in (a,b), Y \in (c,d) \,|\, \sqrt{X^2+Y^2} = 1\Big]$. Probability measures exist on spaces with varying and infinite dimension.

Chapter 5 introduces measure theory by using it to interpret discrete Bayesian theories mechanically. Chapter 7 gives a measure-theoretic interpretation of probabilistic programs, which can encode Bayesian theories that do not have density models.

## 2.2 Functional Programming Theory

From functional programming theory, the requisite background knowledge includes the $\lambda$-calculus, big-step operational semantics, denotational semantics, categorical semantics, and abstract interpretation. We assume readers know basic computer science theory, including propositional logic, relations, functions, proof by induction, context-free grammars, and nondeterminism.

### 2.2.1 $\lambda$-Calculus

The following grammar defines a set of variable names $X$ and a language $E$ (a set of terms) called the **pure $\lambda$-calculus**.

$$\begin{aligned} e &::= x \mid e\ e \mid \lambda x.\, e \\ x &::= \text{[variable names]} \end{aligned} \tag{2.27}$$

Terms $\lambda x.\, e$ are unnamed functions of one argument, terms $x$ refer to function arguments, and terms $e_1\ e_2$ apply $e_1$ to $e_2$ (i.e. "call" function $e_1$ with argument $e_2$). For readers unused to the $\lambda$-calculus but familiar with other mathematical languages, perhaps the most difficult thing to get used to is that juxtaposition means application instead of multiplication.

As in most mathematical languages, parentheses are optional. Lambda terms greedily enclose their bodies in implicit parentheses, so $\lambda x.\, \lambda y.\, e$ (with some assumed-meaningful function body $e$) is the same term as $\lambda x.\, (\lambda y.\, e)$: a function that receives an $x$ and returns a function of $y$ in which $x$ is available. Application is left-associative, so $e_1\ e_2\ e_3$ is the same term as $(e_1\ e_2)\ e_3$. Here, $e_1\ e_2$ returns a function, which is applied to $e_3$.

This duality makes it easy to write two-argument functions using nested lambdas, and apply them using sequences of arguments. For example, $(\lambda x.\, \lambda y.\, e)\ e_x\ e_y$ defines a function of two arguments and applies it to $e_x$ and $e_y$.

Like its younger brother the Turing machine, the pure $\lambda$-calculus is a universal model of computation. Also like the Turing machine, it would be quite painful to program with it.

Unlike the Turing machine, it is easy to get something practical with a few extensions such as pairs and numbers, and a few primitive functions to operate on them.

In most programming languages, implementation details define the meaning of function application. It typically involves a jump from one machine address to another, and if the function returns, a jump back. However, in the pure λ-calculus and its extensions, there are no jumps or machine addresses. Function application is defined entirely in terms of substitution, as in algebra. For example, suppose *hypot* is a term in a λ-calculus extended with real numbers, defined by

$$\mathit{hypot} \;=\; \lambda x.\, \lambda y.\, \sqrt{x^2 + y^2} \tag{2.28}$$

Applying *hypot* eliminates lambdas by substituting their formal arguments with the supplied actual arguments:

$$\begin{aligned} \mathit{hypot}\ 3\ 4 \;&=\; \left(\lambda x.\, \lambda y.\, \sqrt{x^2 + y^2}\right)\ 3\ 4 \\ &=\; \left(\left(\lambda x.\, \left(\lambda y.\, \sqrt{x^2 + y^2}\right)\right)\ 3\right)\ 4 \\ &\equiv\; \left(\lambda y.\, \sqrt{3^2 + y^2}\right)\ 4 \\ &\equiv\; \sqrt{3^2 + 4^2} \end{aligned} \tag{2.29}$$

The two equivalences at the end of (2.29) are called **$\beta$-reductions**, or just **reductions**. We would expect $\sqrt{3^2 + 4^2}$ to further reduce to 5.

Computer implementations of an extended λ-calculus, such as the programming language Racket, necessarily use jumps and machine addresses to implement function application. However, the meanings of their programs are defined mathematically as the results of carrying out reductions. It is therefore possible to reason about programs algebraically and inductively, without having to consider complicating machine details.

It is sometimes convenient to define a λ-calculus whose variables refer to function arguments by *number* instead of by *name*. Such numeric references are called **De Bruijn**[3]

[3]Typically pronounced "deh brOIN," and named after Dutch mathematician Nicolaas de Bruijn.

**indexes**. One form of the pure $\lambda$-calculus with De Bruijn indexes is

$$\begin{aligned} e &::= \mathsf{env}\ n \mid e\ e \mid \lambda.\, e \\ n &::= 0 \mid 1 \mid 2 \mid \cdots \end{aligned} \tag{2.30}$$

where a "variable" term $\mathsf{env}\ 0$ refers to the innermost lambda's argument.

Suppose we define *hypot* as a term in a $\lambda$-calculus with De Bruijn indexes, extended with real numbers:

$$\mathit{hypot} \ = \ \lambda.\, \lambda.\, \sqrt{(\mathsf{env}\ 1)^2 + (\mathsf{env}\ 0)^2} \tag{2.31}$$

Here, $\mathsf{env}\ 1$ (which was previously $x$) refers to the outer lambda's argument and $\mathsf{env}\ 0$ refers to the inner lambda's argument. Reducing an application of *hypot* proceeds this way:

$$\begin{aligned} \mathit{hypot}\ 3\ 4 \ &= \ \left(\lambda.\, \lambda.\, \sqrt{(\mathsf{env}\ 1)^2 + (\mathsf{env}\ 0)^2}\right)\ 3\ 4 \\ &\equiv \ \left(\lambda.\, \sqrt{3^2 + (\mathsf{env}\ 0)^2}\right)\ 4 \\ &\equiv \ \sqrt{3^2 + 4^2} \end{aligned} \tag{2.32}$$

So far, we have been taking a certain evaluation order for granted when computing reductions. To highlight an ambiguity, consider this lambda term, which returns 0 given any argument:

$$\mathit{zero} \ = \ \lambda x.\, 0 \tag{2.33}$$

Suppose $1\ /\ 0$ does not reduce to any value, as in algebra. Should *zero* $(1\ /\ 0)$ reduce to 0, or likewise not reduce? In other words, should we accept this reduction:

$$\begin{aligned} \mathit{zero}\ (1\ /\ 0) \ &= \ (\lambda x.\, 0)\ (1\ /\ 0) \\ &\equiv \ 0 \end{aligned} \tag{2.34}$$

or should we require function arguments to reduce before substituting them? Always reducing function arguments first is **call-by-value** reduction, and substituting without reducing arguments is **call-by-name**. Both policies have their place, but we mostly use call-by-value reduction, in which *zero* (1 / 0) does not reduce.

### 2.2.2 Big-Step Operational Semantics

Instead of describing evaluation order using English phrases with scattered mathematical terms, we could instead give our $\lambda$-calculus a **semantics**: a precise mathematical definition of the meaning of its terms. To specify evaluation order and other operational aspects specifically, we would typically give it an **operational semantics**.

An operational semantics is defined by a **reduction relation**, which relates program terms to other program terms. There are two main kinds of operational semantics:

- **Small-step**, specified by a subset of $E \times E$, where $E$ is the set of program terms.
- **Big-step**, specified by a subset of $E \times V$, where $E$ is the set of program terms and $V \subseteq E$ is the set of irreducible program values (e.g. the number 4, the pair $\langle 10, 23 \rangle$).

For example, suppose we have a lambda term

$$\mathit{inc} \;=\; \lambda x.\, x + 1 \tag{2.35}$$

A small-step semantics would typically "stop" after a function application. If "$\Rightarrow$" is a small-step reduction relation, then $(\mathit{inc}\ 4) \Rightarrow (4 + 1)$ should be true, and also $(4 + 1) \Rightarrow 5$, so we can conclude *inc* 4 reduces to 5 in two **small steps**. On the other hand, a big-step semantics cannot "stop" after most function applications. If "$\Downarrow$" is a big-step reduction relation, then we cannot expect $(\mathit{inc}\ 4) \Downarrow (4 + 1)$ because by any reasonable definition, the term $4 + 1$ is an *expression* but not a *value*. We should expect, however, that $(\mathit{inc}\ 4) \Downarrow 5$ is true; i.e. *inc* reduces to 5 in one **big step**.

As with call-by-name and call-by-value, small-step and big-step semantics both have their place. However, as Chapter 3 contains the only operational semantics in this disseration and it is a big-step semantics, we concentrate on big-step in this overview.

Figure 2.4 defines a language and its semantics by giving a grammar and a big-step reduction relation "$\Downarrow$". The language is even simpler than the pure $\lambda$-calculus: its terms

$$\begin{aligned} e &::= v \mid \mathsf{add}\ e\ e \\ v &::= 0 \mid 1 \mid 2 \mid \cdots \end{aligned}$$

(a) A grammar to define sets $E$ and $V$

$$\frac{}{v \Downarrow v}\ (\text{val}) \qquad \frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{(\mathsf{add}\ e_1\ e_2) \Downarrow (v_1 + v_2)}\ (\text{add})$$

(b) Reduction rules to define $\Downarrow \subseteq E \times V$

Figure 2.4: A big-step operational semantics for a simple addition language.

simply represent adding concrete numbers. The relation "$\Downarrow$" is defined by **reduction rules** in the form

$$\frac{\textit{premise}_1 \quad \textit{premise}_2 \quad \cdots}{\textit{conclusion}}\ (\text{name}) \tag{2.36}$$

Grammar nonterminals are implicitly universally quantified, premises are implicitly conjuncted, and the rule is interpreted as an implication. For example, the (add) rule in Figure 2.4b means "for all $e_1, e_2 \in E$ and $v_1, v_2 \in V$, if $e_1$ reduces to $v_1$ and $e_2$ reduces to $v_2$, then $\mathsf{add}\ e_1\ e_2$ reduces to $v_1 + v_2$." The (val) rule means "for all $v \in V$, $v$ reduces to $v$" or equivalently, "for all $v \in V$, *true* implies $v$ reduces to $v$."

The reduction relation "$\Downarrow$" is defined as the *smallest* subset of $E \times V$ for which the reduction rules hold. Defining it as the smallest subset precludes unintended conclusions such as $4 \Downarrow 5$, which are not otherwise precluded by interpreting the rules as implications. Equivalently, it restricts "$\Downarrow$" to conclusions that are provable from the reduction rules.

Reduction rules can be used directly to build **derivation trees**, which represent both computation steps and proofs of conclusions. For example, suppose we want to use the reduction rules in Figure 2.4b to compute the value of $\mathsf{add}\ (\mathsf{add}\ 4\ 5)\ 90$. We start by writing it as a conclusion without premises:

$$\frac{}{(\mathsf{add}\ (\mathsf{add}\ 4\ 5)\ 90) \Downarrow v_1} \tag{2.37}$$

There is only one rule (add) with a matching conclusion, so we add its premises, renaming variables as appropriate:

$$\frac{\dfrac{}{(\mathsf{add}\ 4\ 5) \Downarrow v_2} \quad \dfrac{}{90 \Downarrow v_3}}{(\mathsf{add}\ (\mathsf{add}\ 4\ 5)\ 90) \Downarrow v_1} \tag{2.38}$$

There is only one rule (val) matching the conclusion $90 \Downarrow v_3$, and it has no premises. We thus only add premises for the (add) rule matching $(\textsf{add}\ 4\ 5)$:

$$\dfrac{\dfrac{\dfrac{}{4 \Downarrow v_4} \quad \dfrac{}{5 \Downarrow v_5}}{(\textsf{add}\ 4\ 5) \Downarrow v_2} \quad \dfrac{}{90 \Downarrow v_3}}{(\textsf{add}\ (\textsf{add}\ 4\ 5)\ 90) \Downarrow v_1} \tag{2.39}$$

It is easy to find values of $v_3$, $v_4$ and $v_5$ that make the leaf premises true, so we substitute them and recursively fill in the conclusions:

$$\dfrac{\dfrac{\dfrac{}{4 \Downarrow 4} \quad \dfrac{}{5 \Downarrow 5}}{(\textsf{add}\ 4\ 5) \Downarrow v_2} \quad \dfrac{}{90 \Downarrow v_3}}{(\textsf{add}\ (\textsf{add}\ 4\ 5)\ 90) \Downarrow v_1} \Longrightarrow \dfrac{\dfrac{\dfrac{}{4 \Downarrow 4} \quad \dfrac{}{5 \Downarrow 5}}{(\textsf{add}\ 4\ 5) \Downarrow 9} \quad \dfrac{}{90 \Downarrow 90}}{(\textsf{add}\ (\textsf{add}\ 4\ 5)\ 90) \Downarrow v_1} \Longrightarrow \dfrac{\dfrac{\dfrac{}{4 \Downarrow 4} \quad \dfrac{}{5 \Downarrow 5}}{(\textsf{add}\ 4\ 5) \Downarrow 9} \quad \dfrac{}{90 \Downarrow 90}}{(\textsf{add}\ (\textsf{add}\ 4\ 5)\ 90) \Downarrow 99} \tag{2.40}$$

Thus, the rightmost derivation tree in (2.40) is a proof that $(\textsf{add}\ (\textsf{add}\ 4\ 5)\ 90) \Downarrow 99$.

In most cases, reduction relations can be mathematically constructed by iterating a function that uses the reduction rules to add more conclusions given known premises. A fixpoint is reachable in countably many iterations, and as a consequence, derivation trees are always finite. On the other hand, Chapter 3 defines a λ-calculus in which the iterating function must be applied uncountably many times to reach a fixpoint, and as a consequence, its derivation trees may be infinite. Despite this minor difference in size, the basic principles behind the reduction relation's construction and use are the same.

If a big-step reduction relation "$\Downarrow$" relates each left-hand side term to exactly one right-hand side term, it is a total function, or $\Downarrow : E \to V$. If it relates each left-hand side term to *at most* one right-hand side term, it is a partial function, or $\Downarrow : E \rightharpoonup V$. In either case, if its derivation trees are finite, it can be implemented as a recursive function.

Figure 2.5 gives a Racket implementation of "$\Downarrow$" in Figure 2.4b. (We say Racket is the implementation's **host language**.) The implementation defines a structure type `add` to model add expressions, and uses Racket's built-in big integers to model $V$. Computation recursively interprets expressions, and proceeds similarly to the derivation tree construction in (2.37) through (2.40). As an example of use, at DrRacket's Read-Eval-Print Loop (REPL),

```
(define value? exact-nonnegative-integer?)
(struct add (e1 e2))

(define (interp e)
  (match e
    [(? value? v)  v]
    [(add e1 e2)  (define v1 (interp e1))
                  (define v2 (interp e2))
                  (+ v1 v2)]))
```

Figure 2.5: Racket implementation of the semantics defined in Figure 2.4.

$$
\begin{aligned}
e &::= v \mid \mathsf{add}\ e\ e \mid \mathsf{choose}\ e\ e \\
v &::= 0 \mid 1 \mid 2 \mid \cdots
\end{aligned}
$$

(a) A grammar to define sets $E$ and $V$

$$
\frac{}{v \Downarrow v}\ (\text{val}) \qquad \frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{(\mathsf{add}\ e_1\ e_2) \Downarrow (v_1 + v_2)}\ (\text{add})
$$

$$
\frac{e_1 \Downarrow v_1}{(\mathsf{choose}\ e_1\ e_2) \Downarrow v_1}\ (\text{left}) \qquad \frac{e_2 \Downarrow v_2}{(\mathsf{choose}\ e_1\ e_2) \Downarrow v_2}\ (\text{right})
$$

(b) Reduction rules to define $\Downarrow \subseteq E \times V$

Figure 2.6: Big-step operational semantics for a language with nondeterministic choice.

we get

```
> (interp (add (add 4 5) 90))
99
```

as expected.

Figure 2.6 extends the present example language with nondeterministic choice, which results in a reduction relation that is *not* a function. The culprits are the new rules (left) and (right), which can both match the same conclusion. For example, suppose we want to use the reduction rules in Figure 2.6b to compute the value of $\mathsf{add}\ (\mathsf{choose}\ 4\ 5)\ 90$. We start as before, by writing it as a conclusion without premises:

$$
\frac{}{(\mathsf{add}\ (\mathsf{choose}\ 4\ 5)\ 90) \Downarrow v_1} \tag{2.41}
$$

We match the conclusion to the (add) rule and add its premises:

$$\frac{\dfrac{}{(\mathsf{choose}\ 4\ 5) \Downarrow v_2} \quad \dfrac{}{90 \Downarrow v_3}}{(\mathsf{add}\ (\mathsf{choose}\ 4\ 5)\ 90) \Downarrow v_1} \tag{2.42}$$

Again, there is only one rule (val) matching the conclusion $90 \Downarrow v_3$, and it has no premises. For the conclusion $(\mathsf{choose}\ 4\ 5) \Downarrow v_2$, however, we may choose either (left) or (right), leading to two different derivation trees:

$$\frac{\dfrac{\dfrac{}{4 \Downarrow v_4}}{(\mathsf{choose}\ 4\ 5) \Downarrow v_2} \quad \dfrac{}{90 \Downarrow v_3}}{(\mathsf{add}\ (\mathsf{choose}\ 4\ 5)\ 90) \Downarrow v_1} \qquad \frac{\dfrac{\dfrac{}{5 \Downarrow v_5}}{(\mathsf{choose}\ 4\ 5) \Downarrow v_2} \quad \dfrac{}{90 \Downarrow v_3}}{(\mathsf{add}\ (\mathsf{choose}\ 4\ 5)\ 90) \Downarrow v_1} \tag{2.43}$$

After replacing $v_3$, $v_4$ and $v_5$ with the only values that make the leaf premises true and recursively filling in the conclusions, we would find that both $(\mathsf{add}\ (\mathsf{choose}\ 4\ 5)\ 90) \Downarrow 94$ and $(\mathsf{add}\ (\mathsf{choose}\ 4\ 5)\ 90) \Downarrow 95$ are true, and would have derivation trees to prove these facts.

An implementation of a nondeterministic semantics would be correct if, for every interpretation of a term $e$ that produced value $v$, $e \Downarrow v$ were a valid conclusion. For $\mathsf{choose}\ 4\ 5$, for example, a correct implementation may always choose 4, always choose 5, choose randomly, choose the number that gives the best or worst outcome according to some objective function, or always choose 4 on weekends or during the fall equinox. Its choice is simply not modeled by the semantics.

Suppose we wanted to compute results for every possible combination of nondeterministic choices. We could define a big-step relation $\Downarrow : E \to \mathcal{P}\ V$, which returns (when used as a function) a set of values, and implement an interpreter for it. However, we are saving that example for the next section.

### 2.2.3 Denotational Semantics

A **denotational semantics** is defined by a deterministic **semantic function** from language terms to values *in another language*. The other language is called the **metalanguage** or **target language**, and is often an axiomatic logic such as first-order set theory (i.e. ordinary

$$\llbracket\cdot\rrbracket : E \to \mathbb{N}$$

$$\begin{aligned} \llbracket v \rrbracket &= v \\ \llbracket \textsf{add}\ e_1\ e_2 \rrbracket &= \llbracket e_1 \rrbracket + \llbracket e_2 \rrbracket \end{aligned}$$

(a) A semantic function for the addition language

```
(define-syntax compile
  (syntax-rules (add)
    [(_ (add e1 e2))  (+ (compile e1)
                         (compile e2))]
    [(_ v)  v]))
```

(b) An implementation of the semantic function as a syntax transformer

Figure 2.7: A denotational semantics and its implementation.

mathematics).

Figure 2.7a defines a denotational semantics for the addition language without choose by defining a semantic function $\llbracket\cdot\rrbracket : E \to \mathbb{N}$. The double square brackets are simply a different application syntax: they connote nothing mathematically, but serve as a visual cue to read applications of the semantic function as "the meaning of" or "the denotation of." For example, the meaning of add (add 4 5) 90 is

$$\begin{aligned} \llbracket \textsf{add}\ (\textsf{add}\ 4\ 5)\ 90 \rrbracket &= \llbracket \textsf{add}\ 4\ 5 \rrbracket + \llbracket 90 \rrbracket \\ &= (\llbracket 4 \rrbracket + \llbracket 5 \rrbracket) + \llbracket 90 \rrbracket \\ &= (4+5)+90 \\ &= 99 \end{aligned} \tag{2.44}$$

The semantic function is **compositional**: it gives meaning to terms *by combining the meanings of their direct subterms.* Compositionality allows most proofs of program properties to be done by structural induction, as we will demonstrate shortly.

When the results of applying $\llbracket\cdot\rrbracket$ are computable, because it is compositional, it is often easy to implement it as local syntax transformation or compilation. Figure 2.7b shows a Racket implementation of $\llbracket\cdot\rrbracket$ as a transformation from meaningless parenthetical syntax (an add function does not exist) to runnable Racket syntax. The syntax transformer is barely more than a transcription of the semantic function's definition, with a little extra code to signal to Racket that it is to be applied to the syntax of expressions before compiling or

$$[\![\cdot]\!] : E \to \mathcal{P}\ \mathbb{N}$$

$$\begin{aligned} [\![v]\!] &= \{v\} \\ [\![\mathsf{add}\ e_1\ e_2]\!] &= \{v_1 + v_2 \mid v_1 \in [\![e_1]\!], v_2 \in [\![e_2]\!]\} \\ [\![\mathsf{choose}\ e_1\ e_2]\!] &= [\![e_1]\!] \cup [\![e_2]\!] \end{aligned}$$

Figure 2.8: A denotational semantics for the addition language with nondeterministic choice.

evaluating them (i.e. `define-syntax` instead of `define`) and to identify the symbol `add` as a terminal symbol.

The results of compilation seem to be equivalent to the results of interpretation:

```
> (compile (add (add 4 5) 90))
99
```

but the REPL does not show transformed syntax. Fortunately, `expand-syntax` can show it:

```
> (expand-syntax #'(add (add 4 5) 90))
#'(+ (+ 4 5) 90))
```

Figure 2.8 defines a compositional function $[\![\cdot]\!] : E \to \mathcal{P}\ \mathbb{N}$, which transforms the addition language with nondeterministic choice into sets of natural numbers. For example, the meaning of 4 is $\{4\}$, the meaning of $\mathsf{choose}\ 4\ 5$ is $\{4\} \cup \{5\} = \{4, 5\}$, and the meaning of $\mathsf{add}\ (\mathsf{choose}\ 4\ 5)\ 90$ is

$$\begin{aligned} [\![\mathsf{add}\ (\mathsf{choose}\ 4\ 5)\ 90]\!] &= \{v_1 + v_2 \mid v_1 \in [\![\mathsf{choose}\ 4\ 5]\!], v_2 \in [\![90]\!]\} \\ &= \{v_1 + v_2 \mid v_1 \in ([\![4]\!] \cup [\![5]\!]), v_2 \in [\![90]\!]\} \\ &= \{v_1 + v_2 \mid v_1 \in (\{4\} \cup \{5\}), v_2 \in \{90\}\} \\ &= \{v_1 + v_2 \mid v_1 \in \{4, 5\}, v_2 \in \{90\}\} \\ &= \{4 + 90, 5 + 90\} \\ &= \{94, 95\} \end{aligned} \tag{2.45}$$

We know that under "$\Downarrow$," $\mathsf{add}\ (\mathsf{choose}\ 4\ 5)\ 90$ reduces to both 94 and 95, so it appears $[\![\cdot]\!]$ is correct. It would be nice to know whether it is *always* correct. The following theorem states correctness precisely in terms of "$\Downarrow$," and critically uses $[\![\cdot]\!]$'s compositionality in a proof by induction on the structure of $e$.

**Theorem 2.1** (correctness)**.** *For all $v \in V$ and $e \in E$, $v \in [\![e]\!] \iff e \Downarrow v$.*

*Proof.* Let $v \in V$ and $e \in E$. The proof is by induction on the structure of $e$.

Base case $e \in V$. If $e = v$, then $v \in [\![e]\!] = \{e\} = \{v\}$ by definition of $[\![\cdot]\!]$, and $e \Downarrow v$ by the (val) rule. Similarly, if $e \neq v$, then $v \notin [\![e]\!]$, and not $e \Downarrow v$.

Inductive case $e = \mathsf{add}\ e_1\ e_2$ for some $e_1 \in E$ and $e_2 \in E$.

Suppose $v \in [\![\mathsf{add}\ e_1\ e_2]\!]$. By definition of $[\![\cdot]\!]$, there exist $v_1 \in [\![e_1]\!]$, $v_2 \in [\![e_2]\!]$ such that $v = v_1 + v_2$. By the inductive hypothesis, $e_1 \Downarrow v_1$ and $e_2 \Downarrow v_2$. By the (add) rule, $(\mathsf{add}\ e_1\ e_2) \Downarrow v$.

Conversely, if $(\mathsf{add}\ e_1\ e_2) \Downarrow v$, by (add), there exist $v_1, v_2$ such that $e_1 \Downarrow v_1$, $e_2 \Downarrow v_2$ and $v = v_1 + v_2$. By hypothesis, $v_1 \in [\![e_1]\!]$ and $v_2 \in [\![e_2]\!]$. By definition of $[\![\cdot]\!]$, $v \in [\![\mathsf{add}\ e_1\ e_2]\!]$.

Proof of the inductive case $e = \mathsf{choose}\ e_1\ e_2$ is similar to the preceeding, though each "$\iff$" direction has in inner case for nondeterministic choice. $\square$

Now that we know $[\![\cdot]\!]$ is correct, we can regard any implementation of it as an implementation of "$\Downarrow$" as well. In general, it is easy to transfer theorems about "$\Downarrow$" to a correct $[\![\cdot]\!]$, and vice-versa.

What if we wanted to represent nondeterministic choices using lists instead of sets, or model a different computational effect, such as mutation or probabilitistic choice? We could define a different semantic function for each model, but there is a more elegant way.

### 2.2.4 Categorical Semantics

When computer scientists from any area want to extend a fixed process without having to repeat themselves more than necessary, they **abstract**: they decouple the desired varying part from the fixed process, and parameterize the previously fixed process on the varying part. This characterizes modular, object-oriented, functional, and even semantic abstraction.

To abstract a denotational semantics, we parameterize its semantic function on the meaning it produces. The parameter takes the form of a **category**.[4] In semantics, the category is comprised of a collection of objects called **computations** (i.e. possible program meanings) and operations on them called **combinators**.

The appropriate category for the addition language with choose contains sets of numbers as computations, or $\mathcal{P}\ \mathbb{N}$, and operations on them. While there are many possible collections of combinators, one kind of collection that functional programmers and theorists have found very useful are **monads**.[5] The **set monad** operates on set-valued computations and is defined by these two combinators:

$$\begin{aligned} \mathit{return}_{\mathit{set}}\ v &= \{v\} \\ \mathit{bind}_{\mathit{set}}\ A\ f &= \bigcup_{v \in A} f\ v \end{aligned} \tag{2.46}$$

Evidently, from a semantic function $[\![\cdot]\!]_a$ parameterized on a monad $a$ we should expect $[\![v]\!]_{\mathit{set}} = \mathit{return}_{\mathit{set}}\ v \equiv \{v\}$. How to use $\mathit{bind}_{\mathit{set}}$ is less clear, however. It apparently applies $f$ to the objects in set $A$ to yield a set for each, and collects these sets' members in a big union. Turning the set comprehension in the definition of $[\![\mathsf{add}\ e_1\ e_2]\!]$ into an indexed union (as in $\mathit{bind}_{\mathit{set}}$) makes its use clearer:

$$\begin{aligned} [\![\mathsf{add}\ e_1\ e_2]\!] &= \{v_1 + v_2 \mid v_1 \in [\![e_1]\!], v_2 \in [\![e_2]\!]\} \\ &= \bigcup_{v_1 \in [\![e_1]\!]} \bigcup_{v_2 \in [\![e_2]\!]} \{v_1 + v_2\} \\ &\equiv \bigcup_{v_1 \in [\![e_1]\!]} \bigcup_{v_2 \in [\![e_2]\!]} \mathit{return}_{\mathit{set}}\ (v_1 + v_2) \\ &\equiv \bigcup_{v_1 \in [\![e_1]\!]} \mathit{bind}_{\mathit{set}}\ [\![e_2]\!]\ (\lambda v_2.\ \mathit{return}_{\mathit{set}}\ (v_1 + v_2)) \\ &\equiv \mathit{bind}_{\mathit{set}}\ [\![e_1]\!]\ (\lambda v_1.\ \mathit{bind}_{\mathit{set}}\ [\![e_2]\!]\ (\lambda v_2.\ \mathit{return}_{\mathit{set}}\ (v_1 + v_2))) \end{aligned} \tag{2.47}$$

Thus, we expect $[\![\mathsf{add}\ e_1\ e_2]\!]_{\mathit{set}} = \mathit{bind}_{\mathit{set}}\ [\![e_1]\!]_{\mathit{set}}\ (\lambda v_1.\ \mathit{bind}_{\mathit{set}}\ [\![e_2]\!]_{\mathit{set}}\ (\lambda v_2.\ \mathit{return}_{\mathit{set}}\ (v_1 + v_2)))$.

[4]The word "category" comes from category theory, an alternative axiomatization of mathematics. Fortunately, little knowledge of category theory is necesary to define or understand categorical semantics.

[5]Strictly speaking, in category theory, they are *strong* monads.

$$\llbracket\cdot\rrbracket_a : E \to M_a\ \mathbb{N}$$

$$\begin{aligned}
\llbracket v\rrbracket_a &= \mathit{return}_a\ v \\
\llbracket \mathsf{add}\ e_1\ e_2\rrbracket_a &= \mathit{bind}_a\ \llbracket e_1\rrbracket_a\ (\lambda v_1.\,\mathit{bind}_a\ \llbracket e_2\rrbracket_a\ (\lambda v_2.\,\mathit{return}_a\ (v_1+v_2))) \\
\llbracket \mathsf{choose}\ e_1\ e_2\rrbracket_a &= \mathit{merge}_a\ \llbracket e_1\rrbracket_a\ \llbracket e_2\rrbracket_a
\end{aligned}$$

Figure 2.9: A categorical semantics for the addition language with nondeterministic choice.

Finally, we need to extend the set monad with an operation for choose expressions. We define

$$\mathit{merge}_{set}\ A_1\ A_2 \ = \ A_1 \cup A_2 \tag{2.48}$$

so that $\llbracket \mathsf{choose}\ e_1\ e_2\rrbracket_{set} \ = \ \mathit{merge}_{set}\ \llbracket e_1\rrbracket_{set}\ \ \llbracket e_2\rrbracket_{set}$.

In Figure 2.9, guided by our expectations for $\llbracket\cdot\rrbracket_{set}$, we define a categorical semantics for the addition language with choose, by defining a semantic function $\llbracket\cdot\rrbracket_a$ parameterized on a target monad $a$. The parameterized function $M_a$ returns the monad's computations. If $M_{set}\ X = \mathcal{P}\ X$, then $\llbracket\cdot\rrbracket_{set} : E \to M_{set}\ \mathbb{N}$ is equivalent to $\llbracket\cdot\rrbracket : E \to \mathcal{P}\ \mathbb{N}$ as defined in Figure 2.8, as expected.

Because Figure 2.9 does not refer to sets or set operations, it is abstract enough to interpret programs as many different kinds of computations. For example, let $M_{list}\ X = [X]$, where $[X]$ denotes all the lists of $X$, and define the **list monad** extended with *merge* by

$$\begin{aligned}
\mathit{return}_{list}\ v &= [v] \\
\mathit{bind}_{list}\ vs\ f &= \mathit{concat}\ (\mathit{map}\ f\ vs) \\
\mathit{merge}_{list}\ vs_1\ vs_2 &= \mathit{append}\ vs_1\ vs_2
\end{aligned} \tag{2.49}$$

Here, $[v]$ is a list containing just $v$, *map* applies a function to every element in a list and returns the list of results, and $\mathit{concat} : [[X]] \to [X]$ appends the elements in a list of lists. Now $\llbracket\cdot\rrbracket_{list} : E \to [\mathbb{N}]$ models nondeterminism with lists of numbers instead of sets of numbers.

For example, the meaning of $\mathsf{choose}\ 4\ 5$ as a list of nondeterministic choices is

$$\begin{aligned}
[\![\mathsf{choose}\ 4\ 5]\!]_{list} &= \mathit{merge}_{list}\ [\![4]\!]_{list}\ \ [\![5]\!]_{list} \\
&= \mathit{merge}_{list}\ (\mathit{return}_{list}\ 4)\ (\mathit{return}_{list}\ 5) \\
&\equiv \mathit{merge}_{list}\ [4]\ [5] \\
&\equiv \mathit{append}\ [4]\ [5] \\
&\equiv [4, 5]
\end{aligned} \tag{2.50}$$

The meaning of $\mathsf{add}\ (\mathsf{choose}\ 4\ 5)\ (\mathsf{choose}\ 4\ 5)$ is thus

$$\begin{aligned}
&[\![\mathsf{add}\ (\mathsf{choose}\ 4\ 5)\ (\mathsf{choose}\ 4\ 5)]\!]_{list} \\
&\quad\equiv \mathit{bind}_{list}\ [4,5]\ (\lambda v_1.\ \mathit{bind}_{list}\ [4,5]\ (\lambda v_2.\ \mathit{return}_{list}\ (v_1 + v_2))) \\
&\quad\equiv \mathit{concat}\ (\mathit{map}\ (\lambda v_1.\ \mathit{bind}_{list}\ [4,5]\ (\lambda v_2.\ \mathit{return}_{list}\ (v_1 + v_2)))\ [4,5]) \\
&\quad\equiv \mathit{concat}\ \ [\mathit{bind}_{list}\ [4,5]\ (\lambda v_2.\ \mathit{return}_{list}\ (4 + v_2)), \\
&\qquad\qquad\quad \mathit{bind}_{list}\ [4,5]\ (\lambda v_2.\ \mathit{return}_{list}\ (5 + v_2))] \\
&\quad\equiv \mathit{concat}\ \ [\mathit{concat}\ (\mathit{map}\ (\lambda v_2.\ \mathit{return}_{list}\ (4 + v_2))\ [4,5]), \\
&\qquad\qquad\quad \mathit{concat}\ (\mathit{map}\ (\lambda v_2.\ \mathit{return}_{list}\ (5 + v_2))\ [4,5])] \\
&\quad\equiv \mathit{concat}\ \ [\mathit{concat}\ [\mathit{return}_{list}\ (4+4), \mathit{return}_{list}\ (4+5)], \\
&\qquad\qquad\quad \mathit{concat}\ [\mathit{return}_{list}\ (5+4), \mathit{return}_{list}\ (5+5)]] \\
&\quad\equiv \mathit{concat}\ [\mathit{concat}\ [[8],[9]], \mathit{concat}\ [[9],[10]]] \\
&\quad\equiv \mathit{concat}\ [[8,9],[9,10]] \\
&\quad\equiv [8,9,9,10]
\end{aligned} \tag{2.51}$$

In contrast, $[\![\mathsf{add}\ (\mathsf{choose}\ 4\ 5)\ (\mathsf{choose}\ 4\ 5)]\!]_{set}\ \equiv\ \{8, 9, 10\}$.

The semantic function $[\![\cdot]\!]_a$ can be parameterized not just on the set and list monads, but any monad $a$ for which $\mathit{merge}_a$ can be sensibly defined. This includes monads for any kind of nondeterminism (e.g. all possibilities, angelic/demonic, random, probabilistic) with any kind of encoding for nondeterministic values (e.g. sets, lists, worst/best choices, execution paths, random values, probability distributions). It also includes monads that combine nondeterminism with other effects, such as input/output or backtracking search. Abstracting has granted the desired flexibility.

As evidenced by the long derivations in (2.50) and (2.51), like most other abstractions, semantic abstraction increases complexity in return for its flexibility and generalization. There are many ways to deal with this, including inferring the behavior of effects from computation types, and classifying effectful behaviors as belonging to different categories. The programming language Haskell benefits greatly from categorical semantics by using them to hide the encodings of effects, which, being an implementation of an effect-free $\lambda$-calculus, it cannot compute directly, by design. Its primary way to deal with the increase in complexity is to use just one built-in, standard semantic function that targets any monad, which transforms syntax that many Haskell programmers find (or learn to find) intuitive.

Besides increasing complexity, abstraction affects the semantics in another way that we have only hinted at by using "$\equiv$" instead of "$=$" in some of our equations: *it no longer targets first-order set theory.* Instead, the semantic function $[\![\cdot]\!]_a$ targets a $\lambda$-calculus.

Targeting a $\lambda$-calculus restricts a denotational semantics to be *directly implementable* as a syntax transformer. This restriction is generally regarded as good, because it makes the proof of the direct implementation's correctness trivial. However, we want to define semantic functions for Bayesian notation, which often denotes uncountable things such as probability distributions over $\mathbb{R}$. The entire reason for the work in Chapter 3 is to define a $\lambda$-calculus with a semantics that gives meaning to operations on infinite values of any size, so that we can define categorical semantics for probabilistic languages in Chapter 5 and Chapter 7.

Categorical abstraction has affected the semantics in a third way. Compare the rule for add in Figure 2.7a with the corresponding rule in Figure 2.9:

$$\begin{aligned} [\![\mathsf{add}\ e_1\ e_2]\!] &= \{v_1 + v_2 \mid v_1 \in [\![e_1]\!], v_2 \in [\![e_2]\!]\} \\ [\![\mathsf{add}\ e_1\ e_2]\!]_a &= \mathit{bind}_a\ [\![e_1]\!]_a\ (\lambda v_1.\, \mathit{bind}_a\ [\![e_2]\!]_a\ (\lambda v_2.\, \mathit{return}_a\ (v_1 + v_2))) \end{aligned} \tag{2.52}$$

Because $[\![\mathsf{add}\ e_1\ e_2]\!]$ does not specify the order of evaluating $[\![e_1]\!]$ and $[\![e_2]\!]$, an implementation is free to choose the order, evaluate them in parallel, or let the host language decide. On the other hand, $[\![e_1]\!]_{list}$ *must* be evaluated first, because changing the evaluation order changes the results:

$$\begin{aligned} bind_{list}\ [4,5]\ (\lambda v_1.\ bind_{list}\ [1,2,3]\ (\lambda v_2.\ return_{list}\ (v_1+v_2))) &\equiv [5,6,7,6,7,8] \\ bind_{list}\ [1,2,3]\ (\lambda v_1.\ bind_{list}\ [4,5]\ (\lambda v_2.\ return_{list}\ (v_1+v_2))) &\equiv [5,6,6,7,7,8] \end{aligned} \tag{2.53}$$

In general, parameterizing a semantics on a monad allows certain monads to impose a total order on evaluation, regardless of the host language's evaluation order.

The combinators in a category must obey certain laws. For example, to define a monad, $return_a$ and $bind_a$ most obey these laws:

$$\begin{aligned} bind_a\ (return_a\ x)\ f &\equiv f\ x && \text{left identity} \\ bind_a\ m\ return_a &\equiv m && \text{right identity} \\ bind_a\ (bind_a\ m\ f)\ g &\equiv bind_a\ m\ (\lambda x.\ bind_a\ (f\ x)\ g) && \text{associativity} \end{aligned} \tag{2.54}$$

It is not necessary for readers to understand these laws deeply, just that they exist, are expected to hold, are occasionally useful, and that we interpret them a little more broadly than is typical. In particular, "$\equiv$" is almost always understood to be the default equivalence for the $\lambda$-calculus in which the combinators are defined. When programming in Haskell, this helpfully ensures that using the laws to transform programs maintains program equivalence.

When defining categorical semantics, however, there is no reason for "$\equiv$" to be defined so narrowly. In fact, it is often useful to define equivalence per-category. For example, we might say that two lists are equivalent when the sets of their elements are equal. In Chapter 3's ***limit monad***, computations are infinite sequences, and are equivalent when they converge to the same value. Chapter 7 defines a notion of equivalence for each of the categories it uses to interpret probabilistic programs.

Two other kinds of categories besides monads are useful targets for categorical semantics: **idioms** and **arrows**. Each kind of category has its own combinators, orderings,

$$\mathcal{L}[\![\cdot]\!] : E \to \mathbb{N}$$

$$\begin{aligned}
\mathcal{L}[\![v]\!] &= 1 \\
\mathcal{L}[\![\mathsf{add}\ e_1\ e_2]\!] &= \mathcal{L}[\![e_1]\!] \cdot \mathcal{L}[\![e_2]\!] \\
\mathcal{L}[\![\mathsf{choose}\ e_1\ e_2]\!] &= \mathcal{L}[\![e_1]\!] + \mathcal{L}[\![e_2]\!]
\end{aligned}$$

Figure 2.10: An abstract semantics for the addition language with nondeterministic choice.

and laws. We do not review them here because they are not as well-known in functional programming theory as monads, so the chapters that use them also review them.

### 2.2.5 Abstract Interpretation

When we want to discover something about *every* evaluation of a program, we might do it with **abstract interpretation**: evaluating by operating on just the properties of terms instead of their actual values. Equivalently, we can think of an abstract interpretation as operating on sets of values for which those properties hold. The properties or sets of values are called **abstract values**. The actual values are called **concrete values**.

Perhaps the most common example of abstract interpretation is type checking. In this case, the abstract values are types, which represent properties such as "is a number" or "is a function from lists to natural numbers." During abstract interpretation, expressions are not evaluated on concrete values, but are checked to determine whether they preserve the properties that concrete values should have.

As with concrete interpretation, abstract interpretation is specified by a semantics. As with concrete semantics, any of a language's abstract semantics can be defined using rules or semantic functions. Type systems and type checkers are typically defined by rules with premises and conclusions similar to reduction rules. Because Chapters 7 and 8 define abstract interpretations using semantic functions, we give a small example of that approach here.

Figure 2.10 defines $\mathcal{L}[\![\cdot]\!]$, which defines an abstract semantics for the addition language with choose. (The prefix $\mathcal{L}$ means nothing mathematically; it simply differentiates this semantic function from the others we have defined.) The abstract values are the lengths of lists or cardinalities of finite sets; i.e. natural numbers. The abstract meaning of a term is an *upper bound* on the number of nondeterministic values it computes. For example, the abstract meaning of $\mathsf{add}$ $(\mathsf{choose}\ 4\ 5)$ $(\mathsf{choose}\ 4\ 5)$ is

$$\begin{aligned} \mathcal{L}[\![\mathsf{add}\ (\mathsf{choose}\ 4\ 5)\ (\mathsf{choose}\ 4\ 5)]\!] &= \mathcal{L}[\![\mathsf{choose}\ 4\ 5]\!] \cdot \mathcal{L}[\![\mathsf{choose}\ 4\ 5]\!] \\ &= (\mathcal{L}[\![4]\!] + \mathcal{L}[\![5]\!]) \cdot (\mathcal{L}[\![4]\!] + \mathcal{L}[\![5]\!]) \\ &= (1+1) \cdot (1+1) \\ &= 4 \end{aligned} \tag{2.55}$$

Indeed, $[\![\mathsf{add}\ (\mathsf{choose}\ 4\ 5)\ (\mathsf{choose}\ 4\ 5)]\!]_{set} \equiv \{8, 9, 10\}$, which is no more than 4 values.

This example demonstrates a pervasive fact about abstract semantics: almost every abstract semantics trades precision to get efficiency, tractability, or even computability. Certainly $|\{8, 9, 10\}| \neq 4$.

Usually, we need abstract interpretations to be **sound**, which roughly means that the abstract values are always a conservative approximation of the concrete values. (There is a way to formalize this notion using Galois connections, but that brings in more complexity than we need.) When abstraction interpretations must be sound, the abstract semantics must have a soundness theorem relating it to a concrete semantics, such as the following.

**Theorem 2.2** ($\mathcal{L}[\![\cdot]\!]$ soundness)**.** *For all* $e \in E$, $|[\![e]\!]_{set}| \leq \mathcal{L}[\![e]\!]$.

*Proof.* By structural induction on $e$. □

A soundness theorem sometimes suggests how abstraction interpretations might be used. For a type system, soundness implies that accepted programs never compute concrete values with the wrong type, so operations on concrete values may be specialized in ways that would otherwise be unsafe or incorrect. (A child class's methods may be inlined, for example.) By Theorem 2.2, we can use $\mathcal{L}[\![\cdot]\!]$ to determine how much space to preallocate for results in a less direct but faster implementation of $[\![\cdot]\!]_{set}$, and we will never allocate too little.

Sometimes the abstraction is both sound and precise, as $\mathcal{L}[\![\cdot]\!]$ is with respect to $[\![\cdot]\!]_{list}$.

**Theorem 2.3** ($\mathcal{L}[\![\cdot]\!]$ soundness and precision)**.** *For all $e \in E$, length $[\![e]\!]_{list} = \mathcal{L}[\![e]\!]$.*

*Proof.* By structural induction on $e$. $\square$

Having both soundness and precision is unusual.

Abstract interpretation is often used for program analysis in which determining precise properties is only semidecidable or is undecidable. An example is determining which functions are applied at every application site in a program written in a $\lambda$-calculus. In these cases, another concrete semantics is created, whose concrete interpretations—if they could be evaluated on a computer—would collect the necessary information. An abstract semantics is then created, whose abstract interpretations are computable, and which overapproximate the necessary information.

Such analyses are sometimes said to "embrace the infinite." In this work, we must do the same to interpret Bayesian notation—but instead embrace the *uncountably* infinite. Doing so with a concrete categorical semantics requires a powerful $\lambda$-calculus.

# Chapter 3

# 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

## 3.1 Motivation

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 certainly 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, $\lambda_{\text{ZFC}}$, 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 $\lambda_{\text{ZFC}}$, and derive a computable, directly implementable replacement monad by applying standard topological theorems. When certain proof obligations are met, the outputs of programs that use the computable monad converge to the same values as the outputs of programs that use the uncomputable monad.

Readers interested only in probabilistic programming languages may skip to Chapter 4, which reviews this chapter's highlights, without missing important prerequisites.

## 3.2 Language Tower and Terminology

$\lambda_{\text{ZFC}}$'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 3.3 reviews the axioms, from which we will derive $\lambda_{\text{ZFC}}$'s primitives.

To help ensure $\lambda_{\text{ZFC}}$'s definition conservatively extends set theory, we encode its terms as sets. For example, ordered pairs of sets $x$ and $y$ are encoded as $\langle x, y\rangle = \{\{x\}, \{x, y\}\}$, and $\langle t_{\mathcal{P}}, \mathbb{R}\rangle = \{\{t_{\mathcal{P}}\}, \{t_{\mathcal{P}}, \mathbb{R}\}\}$ encodes the expression that applies the powerset operator to $\mathbb{R}$.

$\lambda_{\text{ZFC}}$'s semantics reduces terms to terms; e.g. $\langle t_{\mathcal{P}}, \mathbb{R}\rangle$ reduces to the actual powerset of $\mathbb{R}$. Thus, $\lambda_{\text{ZFC}}$ 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 $\lambda^{-}_{\text{ZFC}}$ of finite terms and a function $\mathcal{F}[\![\cdot]\!]$ from $\lambda^{-}_{\text{ZFC}}$ to $\lambda_{\text{ZFC}}$. 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 $\lambda_{\text{ZFC}}^{-}$ terms in $\mathsf{sans\ serif}$ 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.

## 3.3 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 $\phi$ 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 ":="; e.g. $nand(\phi_1, \phi_2) := \neg(\phi_1 \wedge \phi_2)$. They must be nonrecursive so they can be exhaustively applied. Such definitions are **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 ":≡" 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 $A$ and $B$ mutually are subsets; i.e. assume $\forall A\, B.\, (A \subseteq B \wedge B \subseteq A \Rightarrow A = B)$. □

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 $\forall\, A\, x.\, (x \in \mathcal{P}(A) \iff x \subseteq A)$. □

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 arbitrary unions of sets of sets exist; i.e. $\forall\, A\, x.\, (x \in \bigcup A \iff \exists\, y.\, x \in y \wedge y \in A)$. □

For example, after $\{x, ...\}$ is defined, $\bigcup\{\{x, y\}, \{y, z\}\} = \{x, y, z\}$. Also, because all objects are sets, "$\bigcup$" 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 (read "there exists exactly one"). 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. □

An **axiom schema** represents countably many axioms. If $R(n, m) \iff m = n + 1$, for example, then there is an instance of Axiom 5 for $R(n, m)$.

It is not hard to show by Axiom 5 that (after $\mathbb{N}$ is defined) $\{m \mid n \in \mathbb{N} \wedge R(n, m)\}$ increments every natural number, yielding the set of positive naturals. But the syntax is cumbersome, so we 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 the more familiar $\{n + 1 \mid n \in \mathbb{N}\}$ is the positive naturals.

It might seem replacement should be *defined* functionally, but predicates allow powerful nonconstructivism. Suppose $Q(y)$ for exactly one $y$. The **description operator**

$$\iota\, y.\, Q(y) \;:\equiv\; \bigcup\{y \mid x \in \mathcal{P}(\varnothing) \wedge Q(y)\} \tag{3.1}$$

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 [43].

### 3.3.1 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 3.1** (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:

$$F_z(Z) \ := \ \{\varnothing\} \cup \{\langle \varnothing, z\rangle \mid z \in Z\} \tag{3.2}$$

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, so that every ordinal is defined as the set of its predecessors. 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 (3.2). 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 [42]),

$$\omega \;:=\; \bigcap\{N \subseteq I \mid N = F_n(N)\} \tag{3.3}$$

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

**Example 3.2** (interpreting a grammar)**.** We build the language defined by $z ::= \varnothing \mid \langle\varnothing, z\rangle$ recursively:

$$\begin{aligned} Z(0) &= \varnothing \\ Z(s(n)) &= F_z(Z(n)),\ n \in \omega \\ Z(\omega) &= \bigcup_{n\in\omega} Z(n) \end{aligned} \tag{3.4}$$

By induction, $Z(n)$ exists for every $n \in \omega$; therefore $Z(\omega)$ exists, so (3.4) is a conservative extension of set theory. It is not hard to prove (also 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 (3.4), we can build the set $\mathcal{V}(\omega)$ of all hereditarily finite sets (see Axiom 3) by iterating $\mathcal{P}$ instead of $F_z$:

$$\begin{aligned} \mathcal{V}(0) &= \varnothing \\ \mathcal{V}(s(n)) &= \mathcal{P}(\mathcal{V}(n)),\ n \in \omega \\ \mathcal{V}(\omega) &= \bigcup_{n\in\omega} \mathcal{V}(n) \end{aligned} \tag{3.5}$$

The set $\omega$ is not just a model of the natural numbers. It is also a number itself: the **first countable ordinal**. Indeed, $\omega$ 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 $\{0, 1, 2, ..., \omega\}$.) However, unlike finite, nonzero ordinals, $\omega$ has no *immediate* predecessor—it is a **limit ordinal**.

Defining more limit ordinals allows iterating $\mathcal{P}$ further. It is not hard to build $\omega + \omega$,

$\omega^2$ and $\omega^\omega$ as least fixpoints. The **Von Neumann hierarchy** generalizes (3.5):

$$\begin{aligned}
\mathcal{V}(0) &= \varnothing \\
\mathcal{V}(s(\alpha)) &= \mathcal{P}(V(\alpha)),\ \text{ordinal } \alpha \\
\mathcal{V}(\beta) &= \bigcup_{\alpha \in \beta} \mathcal{V}(\alpha),\ \text{limit ordinal } \beta
\end{aligned} \tag{3.6}$$

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

Equations (3.4,3.5,3.6) demonstrate **transfinite recursion**, set theory's $\mathsf{unfold}$: defining a function $V$ on ordinals, with $V(\beta)$ in terms of $V(\alpha)$ for every $\alpha \in \beta$.

### 3.3.2 Every Set Can Be Sequenced: Well-Ordering

A **sequence** is a total function from an ordinal to a codomain; e.g. $f \in 3 \to A$ 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 the predicate *Ord* identifies ordinals and $B \leftrightarrow A$ is the set of all 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 the bijective sequence $f$ is not unique, a well-ordering primitive could make $\lambda_{\text{ZFC}}$'s semantics nondeterministic. Fortunately, the existence of a cardinality operator is equivalent to well-ordering [52], so we will give $\lambda_{\text{ZFC}}$ a cardinality primitive.

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

### 3.3.3 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.

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

Figure 3.1: The definition of $\lambda^{-}_{\text{ZFC}}$, which represents countably many $\lambda_{\text{ZFC}}$ terms.

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

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

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 $\omega$. 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, 53, 55].

Constructing a set $A \notin \mathcal{V}(\kappa)$ requires assuming $\kappa$ 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 Section 3.7 is in $\mathcal{V}(\omega+11)$. Besides, if $\lambda_{\text{ZFC}}$ needed to contain large cardinals, we could always assume even larger ones.

## 3.4 $\lambda_{\text{ZFC}}$'s Grammar

We define $\lambda_{\text{ZFC}}$'s terms in three steps. First, we define $\lambda^{-}_{\text{ZFC}}$, 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 $\lambda_{\text{ZFC}}$ by defining its terms, most of which are infinite, as sets in $\mathcal{V}(\kappa)$.

Figure 3.1 shows $\lambda^{-}_{\text{ZFC}}$'s grammar. Expressions $e$ are typical: variables, values, appli-

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

$$\begin{array}{rcl rcl rcl}
\mathcal{F}[\![n]\!] & := & \langle t_{\text{var}}, n\rangle & \mathcal{F}[\![\varnothing]\!] & := & \mathit{set}(\varnothing) & \mathcal{F}[\![\omega]\!] & := & \mathit{set}(\omega) \\
\mathcal{F}[\![e_f\ e_x]\!] & := & \langle t_{\text{app}}, \mathcal{F}[\![e_f]\!], \mathcal{F}[\![e_x]\!]\rangle & \mathcal{F}[\![\mathsf{false}]\!] & := & a_{\text{false}} & a_{\text{false}} & := & \langle t_{\text{atom}}, t_{\text{false}}\rangle \\
\mathcal{F}[\![e_x \in e_A]\!] & := & \langle t_{\in}, \mathcal{F}[\![e_x]\!], \mathcal{F}[\![e_A]\!]\rangle & \mathcal{F}[\![\mathsf{true}]\!] & := & a_{\text{true}} & a_{\text{true}} & := & \langle t_{\text{atom}}, t_{\text{true}}\rangle \\
\cdots & & & \mathit{set}(A) & = & \multicolumn{4}{l}{\langle t_{\text{set}}, \{\mathit{set}(x) \mid x \in A\}\rangle}
\end{array}$$

Figure 3.2: The semantic function $\mathcal{F}[\![\cdot]\!]$ from $\lambda_{\text{ZFC}}^{-}$ terms to $\lambda_{\text{ZFC}}$ terms.

cation, if, and domain-specific primitives, for membership, union, extraction (take), powerset, functional replacement (image), and cardinality. Values $v$ are also typical: booleans and lambdas, and the domain-specific constants $\varnothing$ and $\omega$.

In set theory, $\bigcup \{A\} = A$ holds for all $A$, so $\bigcup$ can extract the element from a singleton. In $\lambda_{\text{ZFC}}$, the encoding of $\bigcup \{A\}$ reduces to $A$ only if $A$ is an encoded set. Therefore, the 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 $\lambda_{\text{ZFC}}$ terms as well-founded sets, by Axiom 2, countably many indexes is sufficient for $\lambda_{\text{ZFC}}$ as well as $\lambda_{\text{ZFC}}^{-}$.

Figure 3.2 shows part of the meta-metalanguage function $\mathcal{F}[\![\cdot]\!]$ that encodes $\lambda_{\text{ZFC}}^{-}$ terms as $\lambda_{\text{ZFC}}$ terms. It distinguishes sorts of terms in the standard way, by pairing them with tags; e.g. if $t_{\text{set}}$ is the "set" tag, then $\langle t_{\text{set}}, \varnothing\rangle$ encodes $\varnothing$.

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

### 3.4.1 An Infinite Set Rule For Finite BNF Grammars

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

To define $\lambda_{\text{ZFC}}$'s terms, we first extend BNF with a set rule: $\{y^{*\alpha}\}$, where $\alpha$ is a cardinal number. Roughly, it means sets comprised of no more than $\alpha$ terms from the language of $y$. Formally, it means $\mathcal{P}_{<}(Y, \alpha)$, where $Y$ is a subset of $y$'s language generated while building a least fixpoint, and the bounded powerset operation is defined by

$$\mathcal{P}_{<}(Y, \alpha) \ := \ \{x \in \mathcal{P}(Y) \mid |x| < \alpha\} \tag{3.7}$$

meaning $\mathcal{P}_{<}(Y, \alpha)$ returns all subsets of $Y$ with cardinality less than $\alpha$.

**Example 3.3** (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 3.4** (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 3.5.** *Let $F$ be a unary function. Define $V$ by transfinite recursion:*

$$\begin{aligned} V(0) &= \varnothing \\ V(s(\alpha)) &= F(V(\alpha)) \\ V(\beta) &= \bigcup_{\alpha \in \beta} V(\alpha), \ \textit{limit ordinal } \beta \end{aligned} \tag{3.8}$$

*Let $\gamma$ be an ordinal. If $F$ is monotone on $V(\gamma)$, $V$ is monotone on $\gamma$, 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 a property of $\bigcup$. $\square$

All the $F$s we define are monotone. In particular, the interpretations of $\{y^{*\alpha}\}$ rules are monotone because $\mathcal{P}$ is monotone. Further, all the $F$s 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 $F^\alpha$ instead of $V(\alpha)$ to mean $\alpha$ iterations of $F$.

**Theorem 3.6.** *Suppose a grammar with* $\{y^{*\alpha}\}$ *rules and iterating function* $F$. *The language of the grammar,* $F$*'s least fixpoint, is* $F^\gamma$, *where* $\gamma$ *is a regular cardinal not less than any* $\alpha$.

*Proof.* Fixpoint by Aczel [3, Theorem 1.3.4]; least fixpoint by Theorem 3.5. $\square$

**Example 3.7** (finite sets)**.** Because $\omega$ is regular, by Theorem 3.6, $F_h$'s least fixpoint is $F_h^\omega$. 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 3.8** (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 3.9** (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. $\diamondsuit$

### 3.4.2 The Grammar of Infinite, Encoded Terms

There are three main differences between $\lambda_{\text{ZFC}}$'s grammar in Fig. 3.3 and $\lambda_{\text{ZFC}}^{-}$'s grammar in Fig. 3.1. First, $\lambda_{\text{ZFC}}$'s grammar defines a language of terms that are already encoded as sets. Second, instead of the symbols $\varnothing$ and $\omega$, it includes, as values, encoded sets of values. Most of these value terms are infinite, such as the encoding of $\omega$. Third, it includes encoded sets of *expressions.*

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

Figure 3.3: $\lambda_{\text{ZFC}}$'s grammar. Here, $\{e^{*\kappa}\}$ means sets comprised of no more than $\kappa$ terms from the language of $e$.

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

$$\begin{aligned}
F_e(E, V) &:= N \cup V \cup \{\langle t_{\text{app}}, e_f, e_x \rangle \mid \langle e_f, e_x \rangle \in E \times E\} \cup \cdots \cup \{\langle t_{\text{set}}, e \rangle \mid e \in \mathcal{P}_<(E, \kappa)\} \\
F_v(E, V) &:= \{a_{\text{false}}, a_{\text{true}}\} \cup \{\langle t_{\lambda}, e \rangle \mid e \in E\} \cup \{\langle t_{\text{set}}, v \rangle \mid v \in \mathcal{P}_<(V, \kappa)\}
\end{aligned} \tag{3.9}$$

To use Theorem 3.6, 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 3.6, its least fixpoint is $F_{ev}^{\kappa}$, 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 $\lambda_{\text{ZFC}} := E'$.

## 3.5 $\lambda_{\text{ZFC}}$'s Big-Step Reduction Semantics

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

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

(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) $\lambda_{\text{ZFC}}$-specific rules

Figure 3.4: Reduction rules defining $\lambda_{\text{ZFC}}$'s big-step, call-by-value semantics.

resulting cardinal number. The rest are

$$\begin{aligned} \widehat{\mathcal{P}}(v_A) &:= \langle t_{\text{set}}, \{\langle t_{\text{set}}, v_x\rangle \mid v_x \in \mathcal{P}(\mathit{snd}(v_A))\}\rangle \\ \widehat{\bigcup}(v_A) &:= \langle t_{\text{set}}, \bigcup\{\mathit{snd}(v_x) \mid v_x \in \mathit{snd}(v_A)\}\rangle \\ \widehat{I}(v_f, v_A) &:= \langle t_{\text{set}}, \{\langle t_{\text{app}}, v_f, v_x\rangle \mid v_x \in \mathit{snd}(v_A)\}\rangle \end{aligned} \tag{3.10}$$

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 3.4 shows the reduction rules that define the reduction relation "$\Downarrow$". Figure 3.4a has standard call-by-value rules: values reduce to themselves, and applications reduce by substitution. Figure 3.4b has the $\lambda_{\text{ZFC}}$-specific rules. Most simply use $V_{\text{set}}$ 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.

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. 3.4 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 $R$:

$$F_{\Downarrow}(R) \ := \ \{c \in \mathcal{U} \mid D(R, c)\} \tag{3.11}$$

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.

For $F_{\Downarrow}$ itself to be nonmonotone, 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. 3.4 can do so.

Because $F_{\Downarrow}$ is monotone and iterating it is monotone, we can define $\Downarrow := F_{\Downarrow}^{\gamma}$ for some ordinal $\gamma$. If $\lambda_{\text{ZFC}}$ 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 3.10** (countably infinite term)**.** If $s$ is the successor function in $\lambda_{\text{ZFC}}$, 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. $\diamondsuit$

**Theorem 3.11.** $\Downarrow := F_{\Downarrow}^{\kappa}$ *is the least fixpoint of* $F_{\Downarrow}$.

*Proof.* Fixpoint by Aczel [3, Theorem 1.3.4]; least fixpoint by Theorem 3.5. $\square$

Lastly, ZFC theorems that do not depend on $\kappa$ can be applied to $\lambda_{\text{ZFC}}$ terms.

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

**Theorem 3.12.** *$\lambda_{ZFC}$'s set values and $\langle t_{\in}, \cdot, \cdot \rangle$ are a model of ZFC-$\kappa$.*

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

## 3.6 Syntactic Sugar and a Small Set Library

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

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

$\lambda_{\text{ZFC}}$ 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 $(\in)$, $\bigcup$, $\mathsf{take}$, $\mathcal{P}$, $\mathsf{image}$ and $\mathsf{card}$ 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 $0 := \varnothing$, 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 3.3, and we similarly elide most of the $\lambda_{\text{ZFC}}$ definitions. However, some deserve special mention.

Because $\lambda_{\text{ZFC}}$ has only *functional* replacement, we cannot define unbounded "$\forall$" and

"∃". But we can define bounded quantifiers in terms of bounded selection, or

$$\begin{aligned} \mathsf{select\ f\ A} &:= \bigcup\ (\mathsf{image}\ (\lambda \mathsf{x}.\, \mathsf{if}\ (\mathsf{f\ x})\ \{\mathsf{x}\}\ \varnothing)\ \mathsf{A}) \\ \forall\, x \in e_A.\, e_f &:\equiv (\mathsf{select}\ (\lambda\, x.\, e_f)\ e_A) = e_A \\ \exists\, x \in e_A.\, e_f &:\equiv (\mathsf{select}\ (\lambda\, x.\, e_f)\ e_A) = \varnothing \end{aligned} \tag{3.12}$$

We also define a bounded description operator using the `filter`-like select:

$$\iota\, x \in e_A.\, e_f \ :\equiv\ \mathsf{take}\ (\mathsf{select}\ (\lambda\, x.\, e_f)\ e_A) \tag{3.13}$$

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

Unlike in first-order logic, converting a predicate to an object in $\lambda_{\mathrm{ZFC}}$ requires a bounding set as well as unique existence. For example, if

$$\langle e_x, e_y \rangle \ :\equiv\ \{\{e_x\}, \{e_x, e_y\}\} \tag{3.14}$$

defines ordered pairs, then

$$\mathsf{fst\ p} \ :=\ \iota\, \mathsf{x} \in (\bigcup\ \mathsf{p}).\, \exists\, \mathsf{y} \in (\bigcup\ \mathsf{p}).\, \mathsf{p} = \langle \mathsf{x}, \mathsf{y} \rangle \tag{3.15}$$

takes the first element by identifying it in the bounding set $\bigcup\ \mathsf{p}$ using a predicate.

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

$$\begin{aligned} \mathsf{return_{set}\ a} &:= \{\mathsf{a}\} \\ \mathsf{bind_{set}\ A\ f} &:= \bigcup\ (\mathsf{image\ f\ A}) \end{aligned} \tag{3.16}$$

Using $\mathsf{bind\ m\ f} = \mathsf{join}\ (\mathsf{lift\ f\ m})$, 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

$$\{x \in e_A\}.\, e_f \ :\equiv\ \mathsf{bind_{set}}\ (\lambda\, x.\, e_f)\ e_A \tag{3.17}$$

read "choose $x$ in $e_A$, then $e_f$." For example, binary cartesian product is defined by

$$\mathsf{A} \times \mathsf{B} \;:=\; \{\mathsf{x} \in \mathsf{A}\}.\, \{\mathsf{y} \in \mathsf{B}\}.\, \mathsf{return}_{\mathsf{set}}\ \langle \mathsf{x}, \mathsf{y} \rangle \tag{3.18}$$

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 $\mathsf{A}$. To distinguish these hash tables from lambdas, we call them **mappings**. Mappings can be applied by $\mathsf{ap}\ \mathsf{f}\ \mathsf{x} := \iota\, \mathsf{y} \in (\mathsf{range}\ \mathsf{f}).\, \langle \mathsf{x}, \mathsf{y} \rangle \in \mathsf{f}$, but we write just $\mathsf{f}\ \mathsf{x}$. We define

$$\mathsf{mapping}\ \mathsf{f}\ \mathsf{A} \;:=\; \mathsf{image}\ (\lambda \mathsf{x}.\, \langle \mathsf{x}, \mathsf{f}\ \mathsf{x} \rangle)\ \mathsf{A} \tag{3.19}$$

to convert a lambda to a mapping or to restrict a mapping to $\mathsf{A}$. We usually use

$$\lambda x \in e_A.\, e_y \;:\equiv\; \mathsf{mapping}\ (\lambda x.\, e_y)\ e_A \tag{3.20}$$

to define mappings.

A sequence of $\mathsf{A}$ is a mapping $\mathsf{xs} \in \alpha \to \mathsf{A}$ for some ordinal $\alpha$. 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 $\mathsf{map}$, $\mathsf{map2}$ and $\mathsf{drop}$ are defined.

## 3.7 Example: The Reals From the Rationals

Here, we demonstrate that $\lambda_{\text{ZFC}}$ 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 $\lambda_{\text{ZFC}}$ 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

$$\begin{aligned}\mathsf{conv\text{-}zero?_R\ xs} &:= \forall\varepsilon\in\mathbb{Q}^+.\,\exists\,\mathsf{N}\in\omega.\,\forall\,\mathsf{n}\in\omega.\,(\mathsf{N}\in\mathsf{n}\Rightarrow|\mathsf{xs\ n}|_{\mathbb{Q}}<_{\mathbb{Q}}\varepsilon)\\ \mathsf{xs}=_{\mathsf{R}}\mathsf{ys} &:= \mathsf{conv\text{-}zero?_R\ (map2\ (-_{\mathbb{Q}})\ xs\ ys)}\end{aligned}\tag{3.21}$$

So a sequence $\mathsf{xs}\in\omega\to\mathbb{Q}$ converges to zero if, for any positive $\varepsilon$, there is some index $\mathsf{N}$ after which all $\mathsf{xs}$ are smaller than $\varepsilon$. 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:

$$\begin{aligned}\mathsf{conv?_R\ xs} &:= \forall\,\mathsf{n}\in\omega.\,\mathsf{xs}=_{\mathsf{R}}(\mathsf{drop\ n\ xs})\\ \mathsf{R} &:= \mathsf{select\ conv?_R}\ (\omega\to\mathbb{Q})\end{aligned}\tag{3.22}$$

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

$$\begin{aligned}\mathsf{quotient\ A}\ (=_{\mathsf{A}}) &:= \mathsf{image}\ (\lambda\mathsf{x}.\,\mathsf{select}\ (=_{\mathsf{A}}\mathsf{x})\ \mathsf{A})\ \mathsf{A}\\ \mathbb{R} &:= \mathsf{quotient\ R}\ (=_{\mathsf{R}})\end{aligned}\tag{3.23}$$

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 3.13.** *Suppose* $=_{\mathsf{X}}$ *is an equivalence relation on* $\mathsf{X}$*, and define its quotient* $\mathbb{X} := \mathsf{quotient}\ \mathsf{X}\ (=_{\mathsf{X}})$*. If* $\mathsf{op}$ *is surjective on* $\mathsf{X}$ *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 (3.21) 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 $\mathbb{Q} \subset \mathbb{R}$.

## 3.8 Example: Computable Real Limits

Exact real computation has been around since Turing's seminal paper [51]. The novelty here is how we do it. We define the ***limit monad*** in $\lambda_{\text{ZFC}}$ for expressing calculations involving limits, with $\mathsf{bind}_{\mathsf{lim}}$ defined in terms of a general $\mathsf{limit}$. We then derive a $\mathsf{limit}$-free, computable replacement $\mathsf{bind}'_{\mathsf{lim}}$. Replacing $\mathsf{bind}_{\mathsf{lim}}$ with $\mathsf{bind}'_{\mathsf{lim}}$ in a $\lambda_{\text{ZFC}}$ term incurs proof obligations. If they can be met, the computable $\lambda_{\text{ZFC}}$ term has the same limit as the original, uncomputable term.

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

The proof obligations are related to topological theorems [37] that we will import as lemmas. By Theorem 3.12, 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 $\lambda_{\text{ZFC}}$. $\mathsf{A} \Rightarrow \mathsf{B}$ is a lambda or mapping type. $\mathsf{A} \to \mathsf{B}$ is the set of total mappings from $\mathsf{A}$ to $\mathsf{B}$. A set is a membership proposition.

### 3.8.1 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 $\lambda_{\text{ZFC}}$-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 $\mathsf{run}$ function is $\mathsf{limit}$.

**Example 3.14** (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 convert it to a mapping: $\mathsf{partial\text{-}sums}\ \mathsf{xs} := \mathsf{mapping}\ (\mathsf{partial\text{-}sums}'\ \mathsf{xs})\ \omega$.

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 $\mathsf{xs}$ 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:

$$\begin{aligned} \mathsf{return}_{\mathsf{lim}}\ \mathsf{x} &:= \lambda \mathsf{n} \in \omega.\, \mathsf{x} \\ \mathsf{bind}_{\mathsf{lim}}\ \mathsf{xs}\ \mathsf{f} &:= \mathsf{f}\ (\mathsf{limit}\ \mathsf{xs}) \end{aligned} \tag{3.24}$$

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 3.15** (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 $\mathsf{limit}$ out of $\mathsf{f}$'s argument. ♢

**Example 3.16** (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 $\mathsf{limit}\ (\mathsf{exp\text{-}seq}\ \mathsf{x}) = \sum_{\mathsf{n}\in\omega} \frac{\mathsf{x}^{\mathsf{n}}}{\mathsf{n}!} = \mathsf{exp}\ \mathsf{x}$ for $\mathsf{x} \in \mathbb{R}$. To exponentiate converging sequences, define $\mathsf{exp}_{\mathsf{lim}}\ \mathsf{xs} := \mathsf{bind}_{\mathsf{lim}}\ \mathsf{xs}\ \mathsf{exp\text{-}seq}$. ♢

### 3.8.2 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 3.17** (continuity in metric spaces)**.** *Let* $\mathsf{f} : \mathsf{A} \Rightarrow \mathsf{B}$ *with* $\mathsf{A}$ *a metric space. Then* $\mathsf{f}$ *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* $\mathsf{f}$ *is defined on all elements of* $\mathsf{xs}$*,* $\mathsf{f}\ (\mathsf{limit}\ \mathsf{xs}) = \mathsf{limit}\ (\mathsf{map}\ \mathsf{f}\ \mathsf{xs})$.

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

$$\begin{aligned} \mathsf{limit}\ (\mathsf{lift}_{\mathsf{lim}}\ \mathsf{f}\ \mathsf{xs}) &= \mathsf{limit}\ (\mathsf{return}_{\mathsf{lim}}\ (\mathsf{f}\ (\mathsf{limit}\ \mathsf{xs}))) \\ &= \mathsf{limit}\ (\mathsf{return}_{\mathsf{lim}}\ (\mathsf{limit}\ (\mathsf{map}\ \mathsf{f}\ \mathsf{xs}))) \\ &= \mathsf{limit}\ (\mathsf{map}\ \mathsf{f}\ \mathsf{xs}) \end{aligned} \tag{3.25}$$

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 3.18** (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 $\mathsf{limit}$ by restricting $\mathsf{join_{lim}}$'s input.

**Definition 3.19** (uniform convergence)**.** *A sequence* $\mathsf{f} \in \omega \to \omega \to \mathbb{U}$ *converges* ***uniformly*** *if* $\forall\,\varepsilon \in \mathbb{R}^+.\,\exists\,\mathsf{N} \in \omega.\,\forall\,\mathsf{n},\mathsf{m} > \mathsf{N}.\,(\delta\ (\mathsf{f\ n\ m})\ (\mathsf{limit\ (f\ n)})) < \varepsilon$*.*

**Lemma 3.20** (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

$$\begin{aligned}\mathsf{limit\ (join_{lim}\ xss)} &= \mathsf{limit}\ \lambda\mathsf{n} \in \omega.\,\mathsf{limit\ (flip\ xss\ n)} \\ &= \mathsf{limit}\ \lambda\mathsf{n} \in \omega.\,\mathsf{flip\ xss\ (r\ n)\ (s\ n)}\end{aligned} \tag{3.26}$$

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 $\mathsf{f}$ is continuous at $\mathsf{limit\ xs}$ and defined on all $\mathsf{xs}$, and to prove that $\mathsf{flip\ (map\ f\ xs)}$ converges uniformly.

**Example 3.21** (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 3.22.** *Let* $\mathsf{f} : \mathsf{A} \Rightarrow (\omega \to \mathsf{B})$*. If* $\omega \to \mathsf{B}$ *has a product topology, then* $\mathsf{f}$ *is continuous if and only if* $(\mathsf{flip\ f})\ \mathsf{n}$ *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 $\mathsf{xs}$, 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 3.23** ($\pi$)**.** 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 $\pi$, we use

$$\begin{aligned}\pi_{\mathsf{lim}} \;:=\; & ((\mathsf{return}_{\mathsf{lim}}\ 16) \times_{\mathsf{lim}} (\mathsf{arctan}_{\mathsf{lim}}\ (\mathsf{return}_{\mathsf{lim}}\ \tfrac{1}{5}))) -_{\mathsf{lim}} \\ & ((\mathsf{return}_{\mathsf{lim}}\ 4) \times_{\mathsf{lim}} (\mathsf{arctan}_{\mathsf{lim}}\ (\mathsf{return}_{\mathsf{lim}}\ \tfrac{1}{239})))\end{aligned} \tag{3.27}$$

where $(\cdot)_{\mathsf{lim}}$ are lifted arithmetic operators. Because (3.27) does not directly use $\mathsf{bind}_{\mathsf{lim}}$, defining the limit-free $\pi'_{\mathsf{lim}}$ 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 $\pi'_{\mathsf{lim}}$ into Racket [16], using its built-in models of $\omega$ 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 $\pi'_{\mathsf{lim}}$ 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/`.

## 3.9 Related Work

O'Connor's completion monad [39] 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 $\lambda_{\text{ZFC}}$, 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 $\lambda_{\text{ZFC}}$ 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 $\kappa$-sized domains, while $\lambda_{\text{ZFC}}$ is interpreted in set theory using a straightforward extension of operational semantics to $\kappa$-sized relations.

The key difference between $\lambda_{\text{ZFC}}$ and these higher-order logics is that $\lambda_{\text{ZFC}}$ is not a logic. It is a programming language with infinite terms, which by design includes a transitive model of set theory (Theorem 3.12). 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 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 $\lambda_{\text{ZFC}}$, such up-front encodings are unnecessary.

Hypercomputation [40] describes many Turing machine extensions, including completion of transfinite computations. Much of the research is devoted to discovering the properties of computation in physically plausible extensions. While $\lambda_{\text{ZFC}}$ might offer a civilized way to program such machines, we do not think of our work as hypercomputation, but as approaching computability from above.

## 3.10 Conclusions

We defined $\lambda_{\text{ZFC}}$, 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.

Now that we have a suitably expressive target language for exact and approximating categorical semantics, we can get back to defining languages for Bayesian modeling and inference. But more generally, 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.

# Chapter 4

# Using $\lambda_{\mathbf{ZFC}}$

The previous chapter defined $\lambda_{\text{ZFC}}$, an untyped, call-by-value, operational $\lambda$-calculus. It is designed for deriving implementable programs from programs that carry out infinite computations. We will mostly use it as a target language for categorical semantics.

There are two reasonably accurate, short characterizations of $\lambda_{\text{ZFC}}$. First, it can be regarded as contemporary mathematics (Zermelo-Fraenkel set theory with the axiom of Choice, or ZFC) with well-defined lambdas. Second, it can be regarded as a pure functional programming language with infinite sets. The previous chapter defines $\lambda_{\text{ZFC}}$ in a way that makes these characterizations absolutely precise.

Fortunately, understanding and writing $\lambda_{\text{ZFC}}$ code, and knowing how to prove $\lambda_{\text{ZFC}}$ code correct, requires much less detail. We review the important details here.

## 4.1 Computations and Values

In $\lambda_{\text{ZFC}}$, essentially every set is a value, as well as every lambda and every set of lambdas. For example, these are all $\lambda_{\text{ZFC}}$ values:

$$\begin{aligned}
&\{1, 2, 3\} \\
&\{(\lambda\, \mathsf{a}.\, \mathsf{a}), (\lambda\, \mathsf{b}.\, \mathsf{b} + 1), (\lambda\, \mathsf{c}.\, \{\mathsf{c}, \mathsf{c} + 1\})\} \qquad (4.1) \\
&\mathbb{N},\ \mathbb{Q},\ \mathbb{R},\ \mathbb{R} \times \mathbb{R},\ \mathbb{R}^{\mathbb{N}},\ \mathbb{R}^{\mathbb{R}}
\end{aligned}$$

(We generally write $\lambda_{\text{ZFC}}$ terms in $\mathsf{sans\ serif}$.) All primitive operations on values, including operations on infinite sets, are assumed to complete instantly if they terminate.

Nonterminating $\lambda_{\text{ZFC}}$ programs are similar to nonterminating programs in any other call-by-value $\lambda$-calculus. For example, a function that does not terminate on any input because of runaway recursion (i.e. an infinite loop) is

$$\mathsf{count\text{-}from}\ \mathsf{n}\ :=\ \mathsf{count\text{-}from}\ (\mathsf{n}+1) \tag{4.2}$$

We could say that $\mathsf{count\text{-}from}\ 0$ does not terminate because it attempts "infinitely deep" computation. This limitation is necessary; for example, it prevents $\lambda_{\text{ZFC}}$ from having a program that solves its own halting problem, which would make it inconsistent.

Terminating, infinite computations are "infinitely wide," as in either of these equivalent expressions:

$$\mathsf{image}\ (\lambda\,\mathsf{n}.\,\mathsf{n}+1)\ \mathbb{N} \qquad\qquad \{\mathsf{n}+1 \mid \mathsf{n}\in\mathbb{N}\} \tag{4.3}$$

Both yield the set of all positive natural numbers. It is usually fine to think of terminating, infinite computations as being run in parallel.

As in ZFC, in $\lambda_{\text{ZFC}}$, all 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, as with computations, values may be "infinitely wide," such as $\mathbb{R}$, but not "infinitely deep," such as infinite trees and infinite lists made from nested pairs.

## 4.2 Auxiliary Type Systems

Though $\lambda_{\text{ZFC}}$ is untyped, it often helps to define an auxiliary type system. When we use a type system, we use a manually checked, polymorphic one 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 $\mathsf{x}$ and $\mathsf{y}$.
- $\mathsf{Set}\ \mathsf{x}$ denotes a set with members of type $\mathsf{x}$.

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

All function arrows are right-associative. Recall that in a $\lambda$-calculus, application is left-associative. This duality makes writing multi-argument function types easy. For example, $\mathsf{f} : \mathbb{N} \Rightarrow (\mathbb{N} \Rightarrow \mathbb{N})$ denotes that function $\mathsf{f}$ returns a $\mathbb{N} \Rightarrow \mathbb{N}$ function. If $\mathsf{m} : \mathbb{N}$ and $\mathsf{n} : \mathbb{N}$ (equivalently $\mathsf{m} \in \mathbb{N}$ and $\mathsf{n} \in \mathbb{N}$), it can be applied twice using $(\mathsf{f}\ \mathsf{m})\ \mathsf{n}$. Alternatively, it can be applied using $\mathsf{f}\ \mathsf{m}\ \mathsf{n}$, and its type may be written $\mathsf{f} : \mathbb{N} \Rightarrow \mathbb{N} \Rightarrow \mathbb{N}$.

Other examples of types are those of the $\lambda_{\text{ZFC}}$ primitives powerset $\mathcal{P} : \mathsf{Set}\ \mathsf{x} \Rightarrow \mathsf{Set}\ (\mathsf{Set}\ \mathsf{x})$, its left inverse, big union $\bigcup : \mathsf{Set}\ (\mathsf{Set}\ \mathsf{x}) \Rightarrow \mathsf{Set}\ \mathsf{x}$, and the `map`-like primitive $\mathsf{image} : (\mathsf{x} \Rightarrow \mathsf{y}) \Rightarrow \mathsf{Set}\ \mathsf{x} \Rightarrow \mathsf{Set}\ \mathsf{y}$.

It is often helpful to create type aliases. For example, to avoid repeatedly writing "$\cup\{\perp\}$" we might define

$$\mathsf{X} \rightsquigarrow_{\perp} \mathsf{Y} \ ::=\ \mathsf{X} \Rightarrow \mathsf{Y} \cup \{\perp\} \tag{4.4}$$

so that $\mathsf{f} : \mathsf{X} \rightsquigarrow_{\perp} \mathsf{Y}$ and $\mathsf{f} : \mathsf{X} \Rightarrow \mathsf{Y} \cup \{\perp\}$ are equivalent type-level statements.

## 4.3 Using ZFC Values and Theorems

Almost everything definable in ZFC can be defined by a finite $\lambda_{\text{ZFC}}$ program. The previous chapter, for example, defined the real numbers, arithmetic, and limits. The only ZFC values that cannot are those that *must* be defined **nonconstructively**: by proving existence and uniqueness, without giving a bound (no matter how loose) on the length or cardinality of the value. Mathematicians avoid such nonconstructive definitions, and most would consider that definition of "nonconstructive" too liberal.[1]

[1]Constructivists would object to allowing the law of excluded middle, which is derivable from $\lambda_{\text{ZFC}}$'s $\mathsf{if}$, and almost everyone else would object to allowing choice functions.

Almost all known ZFC theorems apply to $\lambda_{\text{ZFC}}$'s set values without alteration.[2] Proofs about $\lambda_{\text{ZFC}}$'s set values apply directly to ZFC sets.[3]

We often import well-known ZFC theorems as lemmas; for example:

**Lemma 4.1** (set equality is extensional)**.** *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.

## 4.4 Internal Equality and External Equivalence

Any $\lambda_{\text{ZFC}}$ term $e$ used as a truth statement means "$e$ reduces to $\mathsf{true}$" or "$e$ evalutes to $\mathsf{true}$." 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 $\lambda_{\text{ZFC}}$'s lambda terms are defined, lambda equality is alpha equivalence, or equivalence up to renaming identifiers. 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 4.2** (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, so that we can say $\lambda\, \mathsf{a}.\, 2$ is equivalent to $\lambda\, \mathsf{a}.\, 1 + 1$. However, we want internal equality and basic external equivalence to be similar, and we want to be able to extend "$\equiv$" with type-specific rules.

[2]The only exceptions are theorems that rely critically on the existence of an inaccessible cardinal.
[3]Assuming the existence of an inaccessible cardinal, which is a modest assumption, as ZFC+$\kappa$ is a smaller theory than Coq's [6].

## 4.5 Additional Functions and Syntactic Forms

We use heavily sugared syntax, with automatic currying (including primitive applications, so $\mathsf{image\ fst}$ means $\lambda \mathsf{A}.\,\mathsf{image\ fst\ A}$), binding forms such as indexed unions $\bigcup_{x \in e_A} e$, destructuring binds as in $\mathsf{swap}\ \langle \mathsf{a}, \mathsf{b}\rangle := \langle \mathsf{b}, \mathsf{a}\rangle$, and comprehensions like $\{\mathsf{a} \in \mathsf{A} \mid \mathsf{a} \in \mathsf{B}\}$. We assume logical operators, bounded quantifiers, and typical set operations are defined. To refer to binary operators as values, we enclose them in parentheses, as in $(\in)$ and $(\subseteq)$.

A less typical set operation we use is disjoint union:

$$\begin{aligned} &(\uplus) : \mathsf{Set\ x} \Rightarrow \mathsf{Set\ x} \Rightarrow \mathsf{Set\ x} \\ &\mathsf{A} \uplus \mathsf{B} \ := \ \mathsf{if}\ (\mathsf{A} \cap \mathsf{B} = \varnothing)\ (\mathsf{A} \cup \mathsf{B})\ (\mathsf{take}\ \varnothing) \end{aligned} \tag{4.5}$$

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

Operator precedence is the same as in ordinary mathematics; e.g. $\mathsf{a} + \mathsf{b} \cdot \mathsf{c} \ = \ \mathsf{a} + (\mathsf{b} \cdot \mathsf{c})$. Application has the highest precedence, so $\mathsf{f\ a} + \mathsf{g\ b} \ = \ (\mathsf{f\ a}) + (\mathsf{g\ b})$.

## 4.6 Extensional Functions

In mathematics, logic, and computer science, there are two general classes of functions:

- **Extensional**: functions whose equality, like that of sets, is determined only by their external properties, and not by how they are defined or constructed.
- **Intensional**: functions whose equality is determined only by their internal properties, or by how they are defined or constructed.

In $\lambda_{\mathrm{ZFC}}$, lambda equality is decided by comparing body expressions, so lambdas are intensional.

In ZFC and $\lambda_{\mathrm{ZFC}}$, function extensionality is achieved by encoding functions as sets of input-output pairs. For example, the increment function for the natural numbers is $\{\langle 0, 1\rangle, \langle 1, 2\rangle, \langle 2, 3\rangle, ...\}$. (It is fine to think of such encodings as infinite hash tables.) We call these encodings **mappings**. We use **function** to mean either a lambda or a mapping, and use adjacency (e.g. $(\mathsf{f\ a})$ or $\mathsf{f\ a}$) to apply either kind.

Syntax for constructing unnamed mappings is defined by

$$\lambda\, x_a \in e_A.\, e_b \;:=\; \mathsf{mapping}\ (\lambda\, x_a.\, e_b)\ e_A \tag{4.6}$$

$$\begin{aligned} &\mathsf{mapping} : (\mathsf{X} \Rightarrow \mathsf{Y}) \Rightarrow \mathsf{Set\ X} \Rightarrow (\mathsf{X} \rightharpoonup \mathsf{Y}) \\ &\mathsf{mapping\ f\ A} \;:=\; \mathsf{image}\ (\lambda\, \mathsf{a}.\, \langle \mathsf{a}, \mathsf{f\ a} \rangle)\ \mathsf{A} \end{aligned} \tag{4.7}$$

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 $\mathsf{X}$ to $\mathsf{Y}$; i.e. if $\mathsf{g} : \mathsf{X} \rightharpoonup \mathsf{Y}$, then $\mathsf{g}$'s domain may be a subset of $\mathsf{X}$. Two common partial mapping operations we use in the next chapter are

$$\begin{aligned} &\mathsf{domain} : (\mathsf{X} \rightharpoonup \mathsf{Y}) \Rightarrow \mathsf{Set\ X} \\ &\mathsf{domain\ g} \;:=\; \mathsf{image\ fst\ g} \end{aligned} \tag{4.8}$$

$$\begin{aligned} &\mathsf{preimage} : (\mathsf{X} \rightharpoonup \mathsf{Y}) \Rightarrow \mathsf{Set\ Y} \Rightarrow \mathsf{Set\ X} \\ &\mathsf{preimage\ g\ B} \;:=\; \{\mathsf{a} \in \mathsf{domain\ g} \mid \mathsf{g\ a} \in \mathsf{B}\} \end{aligned} \tag{4.9}$$

The $\mathsf{preimage}$ function finds $\mathsf{g}$'s inputs whose corresponding outputs are in $\mathsf{B}$.

The set $\mathsf{J} \to \mathsf{X}$ contains all the *total* mappings from $\mathsf{J}$ to $\mathsf{X}$; equivalently, all the vectors of $\mathsf{X}$ indexed by $\mathsf{J}$, which may be infinite. We use infinite vectors of type $\mathsf{J} \to [0,1]$ in Chapter 7 as infinite sources of uniformly random numbers.

In short, in addition to lambdas in $\lambda_{\text{ZFC}}$, we have every necessary mathematical object and theorem at our disposal.

# Chapter 5

# Countable Models and Implementation

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

---

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

John W. Tukey

## 5.1 Introduction

Bayesians write theories without regard to whether future calculations are closed-form or tractable. They are loath to make simplifying assumptions. (If answering questions about some probabilistic process involves an unsolvable integral, so be it.) When they must approximate, they often create two theories: an "ideal" theory first, and a second that approximates it.

Because they create theories without regard to future calculations, they usually must accept approximate answers to queries about them. Typically, they adapt algorithms that compute converging approximations in programming languages they are familiar with. The process is tedious and error-prone, and involves much performance tuning and manual optimization. It is by far the most time-consuming part of their work—and also the most automatable part.

They follow this process to adhere to an overriding philosophy: an approximate answer to the right question is worth more than an exact answer to an approximate question. Thus, they put off approximating as long as possible.

We must also adhere to this philosophy because Bayesian practitioners are unlikely to use a language that requires them to approximate early, or that approximates earlier than they would. We have found that a good way to put the philosophy into practice in language design is to create two semantics: an “ideal,” or *exact* semantics first, and a converging, *approximating* semantics.

### 5.1.1 Approach

Measure-theoretic probability is the most successful theory of probability in precision, maturity, and explanatory power. In particular, it is believed to explain every Bayesian theory. We therefore define the exact semantics as a transformation from Bayesian notation to measure-theoretic calculations.

Measure theory treats finite, countably infinite, and uncountably infinite probabilistic outcomes uniformly, but with significant complexity. Though there are relatively few important Bayesian models that require countably many outcomes but not uncountably many, in our preliminary work, we deal with only countable sets. This choice avoids most of measure theory’s complexity while retaining its functional structure, and still requires approximation.

For three syntactic categories of Bayesian notation, we

1. Manually interpret an unambiguous subclass of common notation.
2. Mechanize the interpretation with a semantic function.
3. If necessary, create an approximation and prove convergence.
4. Implement the approximation in Racket [16].

This approach is most effective if the target language can express measure-theoretic calculations and is similar to Racket in structure and semantics. We therefore use $\lambda_{\text{ZFC}}$.

The Bayesian notation we interpret falls into these syntactic categories:

- **Expressions**, which have no side effects, interpreted by $\mathcal{R}[\![\cdot]\!]$.
- **Statements**, which create side effects, interpreted by $\mathcal{M}[\![\cdot]\!]$.
- **Queries**, which observe side effects, interpreted by $\mathsf{P}[\![\cdot]\!]$ and $\mathsf{D}[\![\cdot]\!]$.

We write Bayesian notation in *italics*, Racket in `fixed width`, common keywords in **bold** and invented keywords in ***bold italics***. We omit proofs for space.

## 5.2 The Expression Language

### 5.2.1 Background Theory: Random Variables

Most practitioners of probability understand random variables as free variables whose values have ambient probabilities. But measure-theoretic probability defines a **random variable** $\mathsf{X}$ as a total mapping

$$\mathsf{X} : \Omega \to \mathsf{S}_\mathsf{X} \tag{5.1}$$

where $\Omega$ and $\mathsf{S}_\mathsf{X}$ are sets called **sample spaces**, with elements called **outcomes**. Random variables define and limit what is observable about any outcome $\omega \in \Omega$, so we call outcomes in $\mathsf{S}_\mathsf{X}$ ***observable outcomes***.

**Example 5.1.** Suppose we want to encode, as a random variable $\mathsf{E}$, the act of observing whether the outcome of a die roll is even or odd.

A complicated way is to define $\Omega$ as the possible states of the universe. $\mathsf{E} : \Omega \to \{\mathsf{even}, \mathsf{odd}\}$ must simulate the universe until the die is still, and then recognize the outcome. Hopefully, the probability that $\mathsf{E}\ \omega = \mathsf{even}$ is close to $\frac{1}{2}$.

A tractable way defines $\Omega := \{1, 2, 3, 4, 5, 6\}$ and $\mathsf{E} : \Omega \to \{\mathsf{even}, \mathsf{odd}\}$ so that $\mathsf{E}\ \omega = \mathsf{even}$ if $\omega \in \{2, 4, 6\}$, otherwise $\mathsf{odd}$. The probability that $\mathsf{E}\ \omega = \mathsf{even}$ is the sum of probabilities of every even $\omega \in \Omega$, or $\frac{1}{6} + \frac{1}{6} + \frac{1}{6} = \frac{1}{2}$.

If we are interested in observing only evenness, we can define $\Omega := \{\mathsf{even}, \mathsf{odd}\}$, each with probability $\frac{1}{2}$, and $\mathsf{E}\ \omega := \omega$. $\diamondsuit$

Random variables enable a kind of probabilistic abstraction. The example does it twice. The first makes calculating the probability that $\mathsf{E}\ \omega = \mathsf{even}$ tractable. The second is an optimization. In fact, redefining $\Omega$, the random variables, and the probabilities of outcomes—without changing the probabilities of *observable* outcomes—is the essence of measure-theoretic optimization.

Defining random variables as functions is also a good factorization: it separates nondeterminism from assigning probabilities. It allows us to interpret expressions involving random variables without considering probabilities at all.

### 5.2.2 Interpreting Random Variable Expressions As Computations

When random variables are regarded as free variables, arithmetic with random variables is no different from deterministic arithmetic. Measure-theoretic probability uses the same notation, but regards it as implicit pointwise lifting (as in vector arithmetic). For example, if $\mathsf{A}, \mathsf{B}, \mathsf{C} : \Omega \to \mathbb{R}$ are random variables, $C := A + B$ means $\mathsf{C}\ \omega := (\mathsf{A}\ \omega) + (\mathsf{B}\ \omega)$, and $B := 4 + A$ means $\mathsf{B}\ \omega := 4 + (\mathsf{A}\ \omega)$.

Because we use $\lambda_{\mathrm{ZFC}}$, we can extend the class of random variables from $\Omega \to \mathsf{S}_\mathsf{X}$ to $\Omega \Rightarrow \mathsf{S}_\mathsf{X}$. Including lambdas as well as mappings makes it easy to interpret unnamed random variables: $4 + A$, or in prefix form $((+)\ 4\ A)$, means $\lambda\omega.\,((+)\ 4\ (\mathsf{A}\ \omega))$. Lifting constants allows us to interpret expressions uniformly: if we interpret $(+)$ as $\mathsf{Plus} := \lambda\omega.\,(+)$ and 4 as $\mathsf{Four} := \lambda\omega.\,4$, then $((+)\ 4\ A)$ means

$$\lambda\omega.\,((\mathsf{Plus}\ \omega)\ (\mathsf{Four}\ \omega)\ (\mathsf{A}\ \omega)) \tag{5.2}$$

We abstract lifting and application with these combinators:

$$\begin{aligned} \mathsf{pure}_{\mathsf{rv}}\ \mathsf{c} &:= \lambda\omega.\,\mathsf{c} \\ \mathsf{ap}^*_{\mathsf{rv}}\ \mathsf{F}\ \mathsf{X}_1\ \ldots\ \mathsf{X}_n &:= \lambda\omega.\,((\mathsf{F}\ \omega)\ (\mathsf{X}_1\ \omega)\ \ldots\ (\mathsf{X}_n\ \omega)) \end{aligned} \tag{5.3}$$

$$\begin{aligned}
\mathcal{R}[\![X]\!] &:\equiv X \\
\mathcal{R}[\![x]\!] &:\equiv \mathsf{pure_{rv}}\ x \\
\mathcal{R}[\![v]\!] &:\equiv \mathsf{pure_{rv}}\ v \\
\mathcal{R}[\![e_f\ e_1\ \ldots\ e_n]\!] &:\equiv \mathsf{ap^*_{rv}}\ \mathcal{R}[\![e_f]\!]\ \mathcal{R}[\![e_1]\!]\ \ldots\ \mathcal{R}[\![e_n]\!] \\
\mathcal{R}[\![\lambda\, x_1 \ldots x_n .\, e]\!] &:\equiv \lambda\omega .\, \lambda\, x_1 \ldots x_n .\, (\mathcal{R}[\![e]\!]\ \omega)
\end{aligned}$$

$$\begin{aligned}
\mathsf{pure_{rv}}\ \mathsf{c} &:= \lambda\omega .\, \mathsf{c} \\
\mathsf{ap^*_{rv}}\ \mathsf{F}\ \mathsf{X}_1\ \ldots\ \mathsf{X}_n &:= \lambda\omega .\, ((\mathsf{F}\ \omega)\ (\mathsf{X}_1\ \omega)\ \ldots\ (\mathsf{X}_n\ \omega))
\end{aligned}$$

Figure 5.1: Random variable expression semantics. The source and target language are both $\lambda_{\text{ZFC}}$. Conditionals and primitive operators are trivial special cases of application.

In terms of $\mathsf{pure_{rv}}$ and $\mathsf{ap^*_{rv}}$, $4 + A$ means

$$\begin{aligned}
\mathsf{ap^*_{rv}}\ (\mathsf{pure_{rv}}\ (+))\ (\mathsf{pure_{rv}}\ 4)\ \mathsf{A} &\equiv \mathsf{ap^*_{rv}}\ (\lambda\omega .\, (+))\ (\lambda\omega .\, 4)\ \mathsf{A} \\
&\equiv \lambda\omega .\, ((\lambda\omega .\, (+))\ \omega)\ ((\lambda\omega .\, 4)\ \omega)\ (\mathsf{A}\ \omega) \\
&\equiv \lambda\omega .\, (+)\ 4\ (\mathsf{A}\ \omega) \\
&= \lambda\omega .\, 4 + (\mathsf{A}\ \omega)
\end{aligned} \tag{5.4}$$

as desired. These combinators define an **idiom** [35], which is like a monad but can impose a partial order on computations. The ***random variable idiom*** instantiates the environment idiom with the type constructor I $\mathsf{a}\ ::=\ \Omega \Rightarrow \mathsf{a}$ for some $\Omega$.

$\mathcal{R}[\![\cdot]\!]$ (Figure 5.1), the semantic function that interprets random variable expressions, targets this idiom. It does mechanically what we have done manually, and additionally interprets lambdas. For simplicity, it follows probability convention by assuming single uppercase letters are random variables. Figure 5.1 assumes syntactic sugar has been replaced; e.g. that application is in prefix form.

$\mathcal{R}[\![\cdot]\!]$ may return lambdas that do not terminate when applied to an $\omega$. For now, we assume they terminate for all $\omega \in \Omega$. (Chapter 7 deals with nonterminating programs.)

We will be able to recover mappings using the $\mathsf{mapping}$ function, which, given a domain, converts a lambda or mapping to a mapping, as in $\mathsf{mapping}\ \mathcal{R}[\![4 + A]\!]\ \Omega$.

```
(define-syntax (RV/kernel stx)
  (syntax-parse stx
    [(_ Xs:ids e:expr)
     (syntax-parse #'e #:literal-sets (kernel-literals)
       [X:id  #:when (free-id-in? #'Xs #'X)  #'X]
       [x:id  #'(pure x)]
       [(quote c)  #'(pure (quote c))]
       [(%#plain-app e ...)  #'(ap* (RV/kernel Xs e) ...)]
       ....)]))

(define-syntax (RV stx)
  (syntax-parse stx
    [(_ Xs:ids e:expr)
     #`(RV/kernel Xs #,(local-expand #'e 'expression empty))]))
```

Figure 5.2: A fragment of our implementation of $\mathcal{R}[\![\cdot]\!]$ in Racket.

### 5.2.3 Implementation in Racket

Figure 5.2 shows `RV` and a snippet of `RV/kernel`, the macros that implement $\mathcal{R}[\![\cdot]\!]$. `RV` fully expands expressions into Racket's kernel language, allowing `RV/kernel` to transform any pure Racket expression into a random variable. Both use Racket's new `syntax-parse` library [14]. `RV/kernel` raises a syntax error on `set!`, but there is no way to disallow applying functions that have effects.

Rather than differentiate between kinds of identifiers, `RV` takes a list of known random variable identifiers as an additional argument. It wraps other identifiers with `pure`, allowing arbitrary Racket values to be random variables.

## 5.3 The Query Language

It is best to regard statements in Bayesian theories as specifications for the results of later observations. We therefore interpret queries before interpreting statements. First, however, we must define the state objects that queries observe.

### 5.3.1 Background Theory: Probability Spaces

In practice, functions called **distributions** assign probabilities or probability densities to observable outcomes. Practitioners state distributions for certain random variables, and then calculate the distributions of others.

Measure-theoretic probability generalizes assigning probabilities and densities using **probability measures**, which assign probabilities to *sets* of outcomes. There are typically no special random variables: all random variable distributions are calculated from one global probability measure.

It is generally not possible to assign meaningful probabilities to all subsets of a sample space $\mathit{\Omega}$—except when $\mathit{\Omega}$ is countable. We thus deal here with **discrete probability measures** $\mathsf{P} : \mathsf{Set}\ \mathit{\Omega} \to [0,1]$, where $\mathit{\Omega}$ is countable. Any discrete probability measure is uniquely determined by its value on singleton sets, or by a **probability mass function** $\mathsf{p} : \mathit{\Omega} \to [0,1]$. It is easy to convert $\mathsf{p}$ to a probability measure:

$$\mathsf{sum\ p\ A} \ :=\ \sum_{\omega \in \mathsf{A}} \mathsf{p}\ \omega \tag{5.5}$$

Then $\mathsf{P} = \mathsf{sum\ p}$. Converting the other direction is also easy: $\mathsf{p\ e} = \mathsf{P}\ \{\mathsf{e}\}$.

A **discrete probability space** $\langle \mathit{\Omega}, \mathsf{p} \rangle$ embodies all probabilistic nondeterminism introduced by theory statements. It is fine to think of $\mathit{\Omega}$ as the set of all possible states of a write-once memory, with $\mathsf{p}$ assigning a probability to each state.

### 5.3.2 Background Theory: Queries

Any probability can be calculated from $\langle \mathit{\Omega}, \mathsf{p} \rangle$. For example, suppose we want to calculate, as in Example 5.1, the probability of an even die outcome. We must apply $\mathsf{P}$ to the correct subset of $\mathit{\Omega}$. Suppose $\mathit{\Omega} := \{1,2,3,4,5,6\}$ and that $\mathsf{p} := [1,2,3,4,5,6 \to \frac{1}{6}]$ determines $\mathsf{P}$. The probability that $\mathsf{E}$ outputs $\mathsf{even}$ is

$$\mathsf{P}\ \{\omega \in \mathit{\Omega} \mid \mathsf{E}\ \omega = \mathsf{even}\} \ =\ \mathsf{P}\ \{2,4,6\} \ =\ \mathsf{sum\ p}\ \{2,4,6\} \ =\ \tfrac{1}{2} \tag{5.6}$$

This is a **probability query**.

Alternatively, we could use a **distribution query** to calculate $\mathsf{E}$'s distribution $\mathsf{P_E}$, and then apply it to $\{\mathsf{even}\}$. Measure-theoretic probability elegantly defines $\mathsf{P_E}$ as $\mathsf{P} \circ (\mathsf{preimage}\ \mathsf{E})$, but for now we do not need a measure. We only need the probability mass function $\mathsf{p_E} : \{\mathsf{even}, \mathsf{odd}\} \to [0,1]$, defined by $\mathsf{p_E}\ \mathsf{e} = \mathsf{sum}\ \mathsf{p}\ (\mathsf{preimage}\ \mathsf{E}\ \{\mathsf{e}\})$. Applying it to $\mathsf{even}$ yields

$$\mathsf{p_E}\ \mathsf{even} \ = \ \mathsf{sum}\ \mathsf{p}\ (\mathsf{preimage}\ \mathsf{E}\ \{\mathsf{even}\}) \ = \ \mathsf{sum}\ \mathsf{p}\ \{2,4,6\} \ = \ \tfrac{1}{2} \qquad (5.7)$$

More abstractly, we can calculate discrete distribution queries using

$$\begin{aligned} \mathsf{dist}\ \mathsf{X}\ \langle \Omega, \mathsf{p} \rangle \ := \ & \mathsf{let}\ \ \mathsf{S_X} := \mathsf{image}\ \mathsf{X}\ \Omega \\ & \mathsf{in}\ \ \lambda \mathsf{x} \in \mathsf{S_X}.\, \mathsf{sum}\ \mathsf{p}\ (\mathsf{preimage}\ (\mathsf{mapping}\ \mathsf{X}\ \Omega)\ \{\mathsf{x}\}) \end{aligned} \qquad (5.8)$$

Now $\mathsf{p_E} = \mathsf{dist}\ \mathsf{E}\ \langle \Omega, \mathsf{p} \rangle$. Recall that the special syntax $\lambda x \in e_A.\, e$ creates an unnamed mapping with domain $e_A$, and $\mathsf{mapping}\ \mathsf{X}\ \Omega$ converts $\mathsf{X}$, which may be a lambda, to a mapping with domain $\Omega$, on which preimages are well-defined.

### 5.3.3 Interpreting Query Notation

When random variables are regarded as free variables, special notation $\Pr[\cdot]$ replaces applying the probability measure $\mathsf{P}$ and sets become propositions. For example, a common way to write "the probability of an even die outcome" in practice is $\Pr[E = \mathit{even}]$.

The semantic function $\mathcal{R}[\![\cdot]\!]$ turns propositions about random variables into predicates on $\Omega$. The set corresponding to the proposition is the preimage of $\{\mathsf{true}\}$. For the proposition $E = \mathit{even}$, for example, it is $\mathsf{preimage}\ (\mathsf{mapping}\ \mathcal{R}[\![E = \mathit{even}]\!]\ \ \Omega)\ \{\mathsf{true}\}$. In general,

$$\mathsf{sum}\ \mathsf{p}\ (\mathsf{preimage}\ (\mathsf{mapping}\ \mathcal{R}[\![e]\!]\ \ \Omega)\ \{\mathsf{true}\}) \ = \ \mathsf{dist}\ \mathcal{R}[\![e]\!]\ \langle \Omega, \mathsf{p} \rangle\ \mathsf{true} \qquad (5.9)$$

calculates $\Pr[e]$ when $e$ is a proposition; i.e. when $\mathcal{R}[\![e]\!] : \Omega \Rightarrow \{\mathsf{true}, \mathsf{false}\}$.

Although probability queries have common notation, there seems to be no common notation that denotes distributions *per se*. The typical workarounds are to write implicit formulas like $\Pr[E = e]$ and to give distributions suggestive names like $p_E$. Some theorists use $\mathcal{L}[\cdot]$, with $\mathcal{L}$ for *law*, an obscure synonym of *distribution*. We define $\mathbf{D}[\![\cdot]\!]$ in place of $\mathcal{L}[\cdot]$. Then $\mathbf{D}[\![E]\!]$ denotes $\mathsf{E}$'s distribution.

Though we could define semantic functions $\mathbf{P}[\![\cdot]\!]$ and $\mathbf{D}[\![\cdot]\!]$ right now, we are putting them off until after interpreting statements.

### 5.3.4 Approximating Queries

Probabilities are real numbers. They remain real in the approximating semantics; we use floating-point approximation and exact rationals in the implementation.

Arbitrary countable sets are not finitely representable. In the approximating semantics, we restrict $\Omega$ to recursively enumerable sets. The implementation encodes them as lazy lists. We trust users to not create "sets" with duplicates.

When $\mathsf{A}$ is infinite, $\mathsf{sum}\ \mathsf{p}\ \mathsf{A}$ is an infinite series. With $\mathsf{A}$ represented by a lazy list, it is easy to compute a converging approximation—but then approximate answers to distribution queries sum to values less than 1. Instead, we approximate $\Omega$ and normalize $\mathsf{p}$, which makes the sum finite and the distributions proper.

Suppose $\langle\omega_1, \omega_2, ...\rangle$ is an enumeration of $\Omega$. Let $\mathsf{z} \in \mathbb{N}^+$ be the length of the prefix $\Omega_\mathsf{z} := \{\omega_1, ..., \omega_\mathsf{z}\}$ and define $\mathsf{p}_\mathsf{z} : \Omega \to [0,1]$ by $\mathsf{p}_\mathsf{z}\ \omega = (\mathsf{p}\ \omega)/(\mathsf{sum}\ \mathsf{p}\ \Omega_\mathsf{z})$ if $\omega \in \Omega_\mathsf{z}$; otherwise 0. Then $\mathsf{p}_\mathsf{z}$ converges to $\mathsf{p}$ pointwise. We define $\mathsf{finitize}\ \langle\Omega, \mathsf{p}\rangle := \langle\Omega_\mathsf{z}, \mathsf{p}_\mathsf{z}\rangle$ with $\mathsf{z} \in \mathbb{N}$ as a free variable.

### 5.3.5 Implementation in Racket

Figure 5.3 shows the implementations of $\mathsf{finitize}$ and $\mathsf{dist}$ in Racket. The free variable $\mathsf{z}$ appears as a *parameter* `appx-z`: a variable with static scope but dynamic extent. The `cotake` procedure returns the prefix of a lazy list as a finite list.

To implement $\mathsf{dist}$, we need to represent mappings in Racket. The applicable struct type `mapping` represents lazy mappings with possibly infinite domains. A `mapping` named `f` can be applied with `(f x)`. We do not ensure `x` is in the domain because checking is

```
(struct mapping (domain proc)
  #:property prop:procedure (λ (f x) ((mapping-proc f) x)))

(struct fmapping (default hash)
  #:property prop:procedure
  (λ (f x) (hash-ref (fmapping-hash f) x (fmapping-default f))))

(define appx-z (make-parameter +inf.0))
(define (finitize ps)
  (match-let* ([(mapping Ω P)  ps]
               [Ωn  (cotake Ω (appx-z))]
               [qn  (apply + (map P Ωn))])
    (mapping Ωn (λ (ω) (/ (P ω) qn)))))

(define ((dist X) ps)
  (match-define (mapping Ω P) ps)
  (fmapping 0 (for/fold ([h  (hash)]) ([ω  (in-list Ω)])
                (hash-set h (X ω) (+ (P ω) (hash-ref h (X ω) 0))))))
```

Figure 5.3: Implementation of finite approximation and distribution queries in Racket.

semidecidable and nontermination is a terrible error message. For distributions, checking is not important; the observable domain is.

However, we do not want `dist` to return lazy mappings. Doing so is inefficient: every application of the mapping would filter $\Omega$. Further, `dist` always receives a `finitized` probability space. We therefore define `fmapping` for mappings that are constant on all but a finite set. For these values, `dist` builds a hash table by computing the probabilities of all preimages in one pass through $\Omega$.

We do use `mapping`, but only for probability spaces and stated distributions.

## 5.4 Conditional Queries

For Bayesian practitioners, the most meaningful queries are **conditional** queries: those *conditioned on*, or *given*, some random variable's value. (For example, the probability an email is spam given it contains words like "madam," or the distribution over suspects given security footage.) A probabilistic language without conditional queries is of little more use to them than a general-purpose language with a `random` primitive.

Measure-theoretic conditional probability is too involved to accurately summarize here. When $\mathsf{P}$ is discrete, however, the conditional probability of set $\mathsf{A}$ given set $\mathsf{B}$ (i.e. asserting that $\omega \in \mathsf{B}$), simplifies to

$$\mathsf{Pr}[\mathsf{A} \,|\, \mathsf{B}] \;=\; \mathsf{P}\ (\mathsf{A} \cap \mathsf{B})\ /\ \mathsf{P}\ \mathsf{B} \tag{5.10}$$

In theory and practice, $\mathsf{Pr}[\,\cdot\,|\,\cdot\,]$ is special notation. As with $\Pr[\cdot]$, practitioners apply $\Pr[\,\cdot\,|\,\cdot\,]$ to propositions, and define it with $\Pr[e_A \,|\, e_B] \;:=\; \Pr[e_A \wedge e_B] / \Pr[e_B]$.

**Example 5.2.** Extend Example 5.1 with random variable $\mathsf{L} : \Omega \to \{\mathsf{low}, \mathsf{high}\}$ defined by $\mathsf{L}\ \omega = \mathsf{if}\ (\omega \le 3)\ \mathsf{low}\ \mathsf{high}$. The probability that $E = \mathit{even}$ given $L = \mathit{low}$ is

$$\Pr[E = \mathit{even} \,|\, L = \mathit{low}] \;=\; \frac{\Pr[E = \mathit{even} \wedge L = \mathit{low}]}{\Pr[L = \mathit{low}]} \;=\; \frac{\sum\limits_{\omega \in \{2\}} P\ \omega}{\sum\limits_{\omega \in \{1,2,3\}} P\ \omega} \;=\; \frac{\frac{1}{6}}{\frac{1}{2}} \;=\; \frac{1}{3} \tag{5.11}$$

Similarly, $\Pr[E = \mathit{odd} \,|\, L = \mathit{low}] \;=\; \frac{2}{3}$. Less precisely, there are proportionally fewer even outcomes when $L = \mathit{low}$. ♢

Conditional *distribution* queries ask how one random variable's output influences the distribution of another. As with unconditional distribution queries, practitioners work around a lack of common notation. For example, they might write the distribution of $\mathsf{E}$ given $\mathsf{L}$ as $\Pr[E = e \,|\, L = l]$ or $p_{E|L}$.

It is tempting to define $\mathbf{P}[\![\,\cdot\,|\,\cdot\,]\!]$ in terms of $\mathbf{P}[\![\cdot]\!]$, and $\mathbf{D}[\![\,\cdot\,|\,\cdot\,]\!]$ in terms of $\mathbf{D}[\![\cdot]\!]$. However, defining conditioning as an operation on probability spaces instead of on queries is more flexible. The following abstraction returns a discrete probability space in which $\Omega$ is restricted to the subset where random variable $\mathsf{Y}$ returns $\mathsf{y}$:

$$\begin{aligned} \mathsf{cond}\ \mathsf{Y}\ \mathsf{y}\ \langle \Omega, \mathsf{p} \rangle \;:=\; \mathsf{let}\ \ & \Omega' := \mathsf{preimage}\ (\mathsf{mapping}\ \mathsf{Y}\ \Omega)\ \{\mathsf{y}\} \\ & \mathsf{p}' := \lambda \omega \in \Omega'.\, (\mathsf{p}\ \omega)\ /\ (\mathsf{sum}\ \mathsf{p}\ \Omega') \\ \mathsf{in}\ \ & \langle \Omega', \mathsf{p}' \rangle \end{aligned} \tag{5.12}$$

Then $\Pr[E = \mathit{even} \,|\, L = \mathit{low}]$ means $\mathsf{dist}\ \mathsf{E}\ (\mathsf{cond}\ \mathsf{L}\ \mathsf{low}\ \langle \Omega, \mathsf{p} \rangle)\ \mathsf{even}$.

We approximate $\mathsf{cond}$ by applying $\mathsf{finitize}$ to the probability space first. Its implementation uses finite list procedures instead of set operators.

## 5.5 The Statement Language

Random variables influence each other through global probability spaces. However, because practitioners regard random variables as free variables instead of as functions of a probability space, they state facts about random variable distributions instead of facts about probability spaces. Though they call such collections of statements *models*,[1] to us they are ***probabilistic theories***. A ***model*** is a probability space and random variables that imply the stated facts.

Discrete ***conditional theories*** can always be written to conform to

$$t_i \;::\equiv\; X_i \sim e_i;\; t_{i+1} \mid X_i := e_i;\; t_{i+1} \mid e_a = e_b;\; t_{i+1} \mid \epsilon \tag{5.13}$$

Further, they can always be made ***well-formed***: an $e_j$ may refer to some $X_i$ only when $j > i$ (i.e. no circular bindings). We start by interpreting the most common kind of Bayesian theories, which contain only distribution statements.

### 5.5.1 Interpreting Common Conditional Theories

**Example 5.3.** Suppose we want to know only whether a die outcome is even or odd, high or low. If $\mathsf{L}$'s distribution is $\mathsf{p_L} := [\mathsf{low}, \mathsf{high} \mapsto \frac{1}{2}]$, then $\mathsf{E}$'s distribution depends on $\mathsf{L}$'s output.

Define $\mathsf{p_{E|L}} : \mathsf{S_L} \to \mathsf{S_E} \to [0,1]$ by $\mathsf{p_{E|L}}\ \mathsf{low} = [\mathsf{even} \mapsto \frac{1}{3}, \mathsf{odd} \mapsto \frac{2}{3}]$ and $\mathsf{p_{E|L}}\ \mathsf{high} = [\mathsf{even} \mapsto \frac{2}{3}, \mathsf{odd} \mapsto \frac{1}{3}]$.[2] The conditional theory could be written

$$L \sim p_L;\; E \sim \big(p_{E|L}\ L\big) \tag{5.14}$$

If $\mathsf{L}$ is a measure-theoretic random variable, $\big(p_{E|L}\ L\big)$ does not typecheck: $\mathsf{L} : \Omega \to \mathsf{S_L}$ is clearly not in $\mathsf{S_L}$. The *intent* is that $\mathsf{p_{E|L}}$ specifies how $\mathsf{E}$'s distribution depends on $\mathsf{L}$. $\diamondsuit$

[1] In the colloquial sense, probably to emphasize their essential incompleteness.

[2] Usually, $\mathsf{P_{E|L}} : \mathsf{S_E} \times \mathsf{S_L} \to [0,1]$. We reorder and curry to simplify interpretation.

We can regard $L \sim p_L$ as a constraint on models: $\mathsf{dist}\ \mathsf{L}\ \langle \Omega, \mathsf{p}\rangle$ must be $\mathsf{p_L}$ for every model $\langle \Omega, \mathsf{p}, \mathsf{L}\rangle$. Similarly, $E \sim \left(p_{E|L}\ L\right)$ means $\mathsf{E}$'s conditional distribution is $\mathsf{p_{E|L}}$. We have been using the model $\Omega := \{1,2,3,4,5,6\}$, $\mathsf{p} := [1,2,3,4,5,6 \mapsto \frac{1}{6}]$, and the obvious $\mathsf{E}$ and $\mathsf{L}$. It is not hard to verify that this is also a model:

$$\begin{aligned} \Omega &:= \{\mathsf{low}, \mathsf{high}\} \times \{\mathsf{even}, \mathsf{odd}\} \\ \mathsf{L}\ \langle \omega_1, \omega_2\rangle &:= \omega_1 \\ \mathsf{E}\ \langle \omega_1, \omega_2\rangle &:= \omega_2 \\ \mathsf{p} &:= [\langle \mathsf{low}, \mathsf{even}\rangle, \langle \mathsf{high}, \mathsf{odd}\rangle \mapsto \tfrac{1}{6}, \langle \mathsf{low}, \mathsf{odd}\rangle, \langle \mathsf{high}, \mathsf{even}\rangle \mapsto \tfrac{2}{6}] \end{aligned} \tag{5.15}$$

The construction of $\Omega$, $\mathsf{L}$ and $\mathsf{E}$ in (5.15) clearly generalizes, but $\mathsf{p}$ is trickier. Fully justifying the generalization (including that it meets implicit independence assumptions that we have not mentioned) is rather tedious, so we do not do it here. But, for the present example, it is not hard to check these facts:

$$\begin{aligned} \mathsf{p} &= \lambda \omega \in \Omega. (\mathsf{p_L}\ (\mathsf{L}\ \omega)) \cdot \left(\mathsf{p_{E|L}}\ (\mathsf{L}\ \omega)\ (\mathsf{E}\ \omega)\right) \\ &= \mathsf{mapping}\ \mathcal{R}\left[\!\left[(p_L\ \mathsf{L}) \cdot \left(\left(p_{E|L}\ L\right)\ \mathsf{E}\right)\right]\!\right]\ \Omega \end{aligned} \tag{5.16}$$

Let $\mathsf{K_L} := \mathcal{R}[\![p_L]\!]$ and $\mathsf{K_E} := \mathcal{R}\left[\!\left[p_{E|L}\ L\right]\!\right]$, which interpret (5.14)'s statements' right-hand sides. Then $\mathsf{p} = \mathsf{mapping}\ \mathcal{R}[\![(\mathsf{K_L}\ \mathsf{L}) \cdot (\mathsf{K_E}\ \mathsf{E})]\!]\ \Omega$. This can be generalized.

**Definition 5.4** (discrete product model)**.** *Given a well-formed, discrete conditional theory* $X_1 \sim e_1; ...; X_n \sim e_n$*, let* $\mathsf{K}_i : \Omega \Rightarrow \mathsf{S}_i \to [0,1]$*, defined by* $\mathsf{K}_i = \mathcal{R}[\![e_i]\!]$ *for each* $1 \leq i \leq n$*. The* ***discrete product model*** *of the theory is*

$$\begin{aligned} \Omega &:= \mathsf{S}_1 \times ... \times \mathsf{S}_n \\ \mathsf{X}_i\ \langle \omega_1, ..., \omega_i, ..., \omega_n\rangle &:= \omega_i \qquad (1 \leq \mathsf{i} \leq \mathsf{n}) \\ \mathsf{p} &:= \mathsf{mapping}\ \mathcal{R}[\![(\mathsf{K}_1\ \mathsf{X}_1) \cdot\ ...\ \cdot (\mathsf{K}_n\ \mathsf{X}_n)]\!]\ \Omega \end{aligned} \tag{5.17}$$

**Theorem 5.5** (semantic intent)**.** *The discrete product model induces the stated conditional distributions and meets implicit independence assumptions.*

When writing distribution statements, practitioners tend to apply first-order distributions to simple random variables. But the discrete product model allows any $\lambda_{\text{ZFC}}$ term $e_i$ whose interpretation is a discrete **transition kernel** $\mathcal{R}[\![e_i]\!] : \mathit{\Omega} \Rightarrow \mathsf{S}_i \rightarrow [0, 1]$. In measure theory, transition kernels are used to build **product spaces** such as $\langle \mathit{\Omega}, \mathsf{p} \rangle$. Thus, $\mathcal{R}[\![\cdot]\!]$ links Bayesian practice to measure theory and represents an increase in expressive power in specifying distributions, by turning properly typed $\lambda_{\text{ZFC}}$ terms into precisely what measure theory requires.

## 5.5.2 Interpreting Statements as Monadic Computations

Some conditional theories state more than just distributions [34, 50]. Interpreting theories with different kinds of statements requires recursive, rather than whole-theory, interpretation. Fortunately, well-formedness amounts to lexical scope, making it straightforward to interpret statements as monadic computations.

We use the state monad with probability-space-valued state: computations are functions from probability spaces to probability spaces paired with a statement-specific value. The probability space monad's return and bind are defined as

$$\begin{aligned} \mathsf{return_{ps}}\ \mathsf{x}\ \langle \mathit{\Omega}, \mathsf{p} \rangle &:= \langle \mathit{\Omega}, \mathsf{p}, \mathsf{x} \rangle \\ \mathsf{bind_{ps}}\ \mathsf{m}\ \mathsf{f}\ \langle \mathit{\Omega}, \mathsf{p} \rangle &:= \mathsf{let}\ \ \langle \mathit{\Omega}', \mathsf{p}', \mathsf{x} \rangle := \mathsf{m}\ \langle \mathit{\Omega}, \mathsf{p} \rangle \\ &\qquad\ \mathsf{in}\ \ \mathsf{f}\ \mathsf{x}\ \langle \mathit{\Omega}', \mathsf{p}' \rangle \end{aligned} \tag{5.18}$$

Figure 5.4 shows the additional $\mathsf{dist_{ps}}$, $\mathsf{cond_{ps}}$ and $\mathsf{extend_{ps}}$. The first two simply reimplement dist and cond. But $\mathsf{extend_{ps}}$, which interprets statements, needs more explanation.

According to (5.17), interpreting $X_i \sim e_i$ results in $\mathit{\Omega}_i = \mathit{\Omega}_{i-1} \times \mathsf{S}_i$, with $\mathsf{S}_i$ extracted from $\mathsf{K}_i : \mathit{\Omega}_{i-1} \Rightarrow \mathsf{S}_i \rightarrow [0, 1]$. A more precise type for $\mathsf{K}_i$ is the dependent type $(\omega : \mathit{\Omega}_{i-1}) \Rightarrow (\mathsf{S}'_i\ \omega) \rightarrow [0, 1]$, which reveals a complication. To extract $\mathsf{S}_i$, we first must extract the random variable $\mathsf{S}'_i : \mathit{\Omega}_{i-1} \rightarrow \mathsf{Set}\ \mathsf{S}_i$. So let $\mathsf{S}'_i\ \omega = \mathsf{domain}\ (\mathsf{K}_i\ \omega)$; then $\mathsf{S}_i = \bigcup (\mathsf{image}\ \mathsf{S}'_i\ \mathit{\Omega}_{i-1})$.

But this makes query implementation inefficient: if the union has little overlap or is disjoint, p will assign 0 to most $\omega$. In more general terms, we actually have a *dependent*

$$
\begin{aligned}
\mathsf{dist_{ps}}\ \mathsf{X}\ \langle \Omega, \mathsf{p}\rangle \ :=\ & \mathsf{let}\ \ \mathsf{S_X} := \mathsf{image}\ \mathsf{X}\ \Omega \\
& \qquad\quad \mathsf{p_X} := \lambda \mathsf{x} \in \mathsf{S_X}.\, \mathsf{sum}\ \mathsf{p}\ (\mathsf{preimage}\ (\mathsf{mapping}\ \mathsf{X}\ \Omega)\ \{\mathsf{x}\}) \\
& \mathsf{in}\ \ \langle \Omega, \mathsf{p}, \mathsf{p_X}\rangle
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{cond_{ps}}\ \mathsf{Y}\ \mathsf{y}\ \langle \Omega, \mathsf{p}\rangle \ :=\ & \mathsf{let}\ \ \Omega' := \mathsf{preimage}\ (\mathsf{mapping}\ \mathsf{Y}\ \Omega)\ \{\mathsf{y}\} \\
& \qquad\quad \mathsf{p}' := \lambda \omega \in \Omega'.\, (\mathsf{p}\ \omega)\ /\ (\mathsf{sum}\ \mathsf{p}\ \Omega') \\
& \mathsf{in}\ \ \langle \Omega', \mathsf{p}', \_\rangle
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{extend_{ps}}\ \mathsf{K}\ \langle \Omega, \mathsf{p}\rangle \ :=\ & \mathsf{let}\ \ \mathsf{S}'\ \omega := \mathsf{domain}\ (\mathsf{K}\ \omega) \\
& \qquad\quad \Omega' := (\omega \in \Omega) \times (\mathsf{S}'\ \omega) \\
& \qquad\quad \mathsf{X}\ \omega := \omega_j \quad \text{(where $j$ is the length of any $\omega \in \Omega$)} \\
& \qquad\quad \mathsf{p}' := \mathsf{mapping}\ \mathcal{R}[\![\mathsf{p} \cdot (\mathsf{K}\ \mathsf{X})]\!]\ \ \Omega' \\
& \mathsf{in}\ \ \langle \Omega', \mathsf{p}', \mathsf{X}\rangle
\end{aligned}
$$

$$
\mathsf{empty_{ps}}\ :=\ \langle \{\langle\rangle\}, \lambda \omega \in \{\langle\rangle\}.\, 1\rangle
$$

$$
\begin{aligned}
\mathsf{run_{ps}}\ \mathsf{m}\ :=\ & \mathsf{let}\ \ \langle \Omega, \mathsf{p}, \mathsf{x}\rangle := \mathsf{m}\ \mathsf{empty_{ps}} \\
& \mathsf{in}\ \ \mathsf{x}
\end{aligned}
$$

Figure 5.4: State monad functions that represent queries and statements. The state is probability-space-valued.

cartesian product $(\omega \in \Omega_{i-1}) \times (\mathsf{S}'_i\ \omega)$, a generalization of the cartesian product.[3] To extend $\Omega$, $\mathsf{extend_{ps}}$ calculates this product instead.

Dependent cartesian products are elegantly expressed using the set monad:

$$
\begin{aligned}
\mathsf{return_{set}}\ \mathsf{a}\ &:=\ \{\mathsf{a}\} \\
\mathsf{bind_{set}}\ \mathsf{A}\ \mathsf{f}\ &:=\ \bigcup(\mathsf{image}\ \mathsf{f}\ \mathsf{A})
\end{aligned}
\tag{5.19}
$$

Then $(\mathsf{a} \in \mathsf{A}) \times (\mathsf{B}\ \mathsf{a}) = \mathsf{bind_{set}}\ \mathsf{A}\ \lambda \mathsf{a}.\, \mathsf{bind_{set}}\ (\mathsf{B}\ \mathsf{a})\ \lambda \mathsf{b}.\, \mathsf{return_{set}}\ \langle \mathsf{a}, \mathsf{b}\rangle$.

Figure 5.5 defines $\mathcal{M}[\![\cdot]\!]$, which interprets conditional theories containing definition, distribution, and conditioning statements as probability space monad computations. After it exhausts the statements, it returns the random variables. Returning their names as well would be an obfuscating complication, which we avoid by implicitly extracting them from the theory

[3]The dependent cartesian product also generalizes disjoint union to arbitrary index sets. It is often called a *dependent sum* and denoted $\Sigma \mathsf{a} : \mathsf{A}.(\mathsf{B}\ \mathsf{a})$.

$$\begin{aligned}
\mathcal{M}[\![X_i := e_i;\ t_{i+1}]\!] &:\equiv \mathsf{bind_{ps}}\ (\mathsf{return_{ps}}\ \mathcal{R}[\![e_i]\!])\ \lambda\, X_i.\, \mathcal{M}[\![t_{i+1}]\!] \\
\mathcal{M}[\![X_i \sim e_i;\ t_{i+1}]\!] &:\equiv \mathsf{bind_{ps}}\ (\mathsf{extend_{ps}}\ \mathcal{R}[\![e_i]\!])\ \lambda\, X_i.\, \mathcal{M}[\![t_{i+1}]\!] \\
\mathcal{M}[\![e_a = e_b;\ t_{i+1}]\!] &:\equiv \mathsf{bind_{ps}}\ (\mathsf{cond_{ps}}\ \mathcal{R}[\![e_a]\!]\ \mathcal{R}[\![e_b]\!])\ \lambda\_.\, \mathcal{M}[\![t_{i+1}]\!] \\
\mathcal{M}[\![\epsilon]\!] &:\equiv \mathsf{return_{ps}}\ \langle X_1, ..., X_n\rangle
\end{aligned}$$

$$\begin{aligned}
\mathbf{D}[\![e]\!]\ \mathsf{m} &:\equiv \mathsf{run_{ps}}\, (\mathsf{bind_{ps}}\ \mathsf{m}\ \lambda\, \langle X_1, ..., X_n\rangle.\, \mathsf{dist_{ps}}\ \mathcal{R}[\![e]\!]) \\
\mathbf{D}[\![e_X \,|\, e_Y]\!]\ \mathsf{m} &:\equiv \lambda \mathsf{y}.\, \mathbf{D}[\![e_X]\!]\ (\mathsf{bind_{ps}}\ \mathsf{m}\ \lambda\, \langle X_1, ..., X_n\rangle.\, \mathcal{M}[\![e_Y = \mathsf{y}]\!]) \\
\mathbf{P}[\![e]\!]\ \mathsf{m} &:\equiv \mathbf{D}[\![e]\!]\ \mathsf{m}\ \mathsf{true} \\
\mathbf{P}[\![e_A \,|\, e_B]\!]\ \mathsf{m} &:\equiv \mathbf{D}[\![e_A \,|\, e_B]\!]\ \mathsf{m}\ \mathsf{true}\ \mathsf{true}
\end{aligned}$$

Figure 5.5: The conditional theory and query semantic functions.

before interpretation. (However, the implementation explicitly extracts and returns names.) Figure 5.5 also defines semantic functions for queries. $\mathbf{D}[\![e]\!]$ expands to a distribution-valued computation and runs it with a probability space with the single outcome $\langle\rangle$. $\mathbf{D}[\![e_X \,|\, e_Y]\!]$ conditions the probability space and hands off to $\mathbf{D}[\![e_X]\!]$. $\mathbf{P}[\![\cdot]\!]$ is defined in terms of $\mathbf{D}[\![\cdot]\!]$.

## 5.5.3 Approximating Models and Queries

We compute dependent cartesian products of sets represented by lazy lists in a way similar to enumerating $\mathbb{N} \times \mathbb{N}$. (It cannot be done with a monad as in the exact semantics, but we do not need it to.) The approximating versions of $\mathsf{dist_{ps}}$ and $\mathsf{cond_{ps}}$ apply $\mathsf{finitize}$ to the probability space.

## 5.5.4 Implementation in Racket

$\mathcal{M}[\![\cdot]\!]$'s implementation is `MDL`. Like `RV`, it passes random variable identifiers; unlike `RV`, `MDL` accumulates them. For example, `(MDL [] ([X ∼ Px]))` expands to

```
([X] (bind/ps (extend/ps (RV [] Px)) (λ (X) (ret/ps (list X)))))
```

where `[X]` is the updated list of identifiers and the rest is a model computation.

We store theories in transformer bindings so queries can expand them later. For example, `(define-model die-roll [L ∼ Pl] [E ∼ (Pe/l L)])` expands to

```
(define-syntax die-roll #'(MDL [] ([L ~ Pl] [E ~ (Pe/l L)])))
```

The macro with-model introduces a scope in which a theory's variables are visible. For example, (with-model die-roll (Dist L E)) looks up die-roll and expands it into its identifiers and computation. Using the identifiers as lambda arguments, Dist (the implementation of $\mathbf{D}[\![\cdot]\!]$) builds a query computation as in Figure 5.5, and runs it with (mapping (list empty) (λ (ω) 1)), the empty probability space.

Using these identifiers would break hygiene, except that Dist replaces the lambda arguments' lexical context. This puts the theory's exported identifiers in scope, even when the theory and query are defined in separate modules. Because queries can access only the exported identifiers, it is safe.

Aside from passing identifiers and monkeying with hygiene, the macros are almost transcribed from the semantic functions.

### 5.5.5 Examples

Consider a conditional distribution with the first-order definition

```
(define (Geometric p)
  (mapping N1 (λ (n) (* p (expt (- 1 p) (- n 1))))))
```

where N1 is a lazy list of natural numbers starting at 1. Nahin gives a delightfully morbid use for Geometric in his book of probability puzzlers [38].

Two idiots duel with one gun. They put only one bullet in it, and take turns spinning the chamber and firing at each other. They know that if they each take one shot at a time, player one usually wins. Therefore, player one takes one shot, and after that, the next player takes one more shot than the previous player, spinning the chamber before each shot. How probable is player two's demise?

The distribution over the number of shots when the gun fires is (Geometric 1/6). Using this procedure to determine whether player one fires shot n:

```
(define (p1-fires? n [shots 1])
  (cond [(n . <= . 0)  #f]
        [else  (not (p1-fires? (- n shots) (add1 shots)))]))
```

we compute the probability that player one wins with

```
(with-model (model [winning-shot ~ (Geometric 1/6)])
  (Pr (p1-fires? winning-shot)))
```

Nahin computes 0.5239191275550995247919843—25 decimal digits—with custom MATLAB code. At `appx-z` $\geq$ `321`, our solution computes the same digits. (Though it appends the digits 9..., so Nahin should have rounded up.) Implementing it took about five minutes. But the problem is not Bayesian.

This is: suppose player one slyly suggests a single coin flip to determine whether they spin the chamber before each shot. You do not see the duel, but learn that player two won. What is the probability they spun the chamber?

Suppose that the well-known `Bernoulli` and discrete `Uniform` conditional distributions are defined. Using these first-order conditional distributions and Racket's `cond`, we can state a fairly direct theory of the duel:

```
(define-model half-idiot-duel
  [spin? ~ (Bernoulli 1/2)]
  [winning-shot ~ (cond [spin?  (Geometric 1/6)]
                        [else   (Uniform 1 6)])])
```

Then `(Pr spin? (not (p1-fires? winning-shot)))` converges to about `0.588`.

Bayesian practitioners would normally create a new first-order conditional distribution `WinningShot`, and then state `[winning-shot ~ (WinningShot spin?)]`. Most would *like* to state something more direct—such as the above theory, which plainly shows how `spin?`'s value affects `winning-shot`'s distribution. However, without a semantics, they cannot be sure that using the value of a `cond` (or of any `if`-like expression) as a distribution is well-defined. That `winning-shot` has a *different range* for each value of `spin?` makes things more uncertain.

As specified by $\mathcal{R}[\![\cdot]\!]$, our implementation interprets `(cond ...)` above as a stochastic transition kernel. As specified by $\mathcal{M}[\![\cdot]\!]$, it builds the probability space using dependent cartesian products. Thus, the direct theory really is well-defined.

## 5.6 Why Separate Statements and Queries?

Whether queries should be allowed inside theories is a decision with subtle effects.

Theories are sets of facts. Well-formedness imposes a partial order, but every linearization should be interpreted equivalently. Thus, we can determine whether two kinds of statements can coexist in theories by determining whether they can be exchanged without changing the interpretation. This is equivalent to determining whether the corresponding monad functions commute.

The following definitions suppose a conditional theory $t_1; ...; t_n$ in which exchanging some $t_i$ and $t_{i+1}$ (where $i < n$) is well-formed. Applying semantic functions in the definitions yields definitions that are independent of syntax but difficult to read, so we give the syntactic versions.

**Definition 5.6** (commutativity)**.** *We say that $t_i$ and $t_{i+1}$* ***commute*** *when*
$\mathcal{M}[\![t_1; ...; t_i; t_{i+1}; ...; t_n]\!]\ \langle \Omega_0, \mathsf{p}_0\rangle \ = \ \mathcal{M}[\![t_1; ...; t_{i+1}; t_i; ...; t_n]\!]\ \langle \Omega_0, \mathsf{p}_0\rangle$.

Unfortunately, this notion of commutativity is usually too strong: distribution statements could never commute with each other. We need a weaker test than equality, based on *observable* outcomes.

**Definition 5.7** (equivalence in distribution)**.** *Suppose $X_1, ..., X_k$ are defined in $t_1, ..., t_n$. Let* $\mathsf{m} := \mathcal{M}[\![t_1, ..., t_n]\!]$*, and* $\mathsf{m}'$ *be a (usually different) probability space monad computation. We write* $\mathsf{m} \equiv_{\mathsf{D}} \mathsf{m}'$ *and call* $\mathsf{m}$ *and* $\mathsf{m}'$ ***equivalent in distribution*** *when* $\mathbf{D}[\![X_1, ..., X_k]\!]\ \mathsf{m} = \mathbf{D}[\![X_1, ..., X_k]\!]\ \mathsf{m}'$.

The following theorem says $\equiv_{\mathsf{D}}$ is like observational equivalence with query contexts:

**Theorem 5.8** (context)**.** $\mathsf{D}[\![e_X \mid e_Y]\!]\ \mathsf{m} \;=\; \mathsf{D}[\![e_X \mid e_Y]\!]\ \mathsf{m}'$ *for all random variables* $\mathcal{R}[\![e_X]\!]$ *and* $\mathcal{R}[\![e_Y]\!]$ *if and only if* $\mathsf{m} \equiv_{\mathsf{D}} \mathsf{m}'$.

**Definition 5.9** (commutativity in distribution)**.** *We say* $t_i$ *and* $t_{i+1}$ *commute* ***in distribution*** *when* $\mathcal{M}[\![t_1; ...; t_i; t_{i+1}; ...; t_n]\!] \equiv_{\mathsf{D}} \mathcal{M}[\![t_1; ...; t_{i+1}; t_i; ...; t_n]\!]$.

**Theorem 5.10.** *The following table summarizes commutativity of* $\mathsf{cond}_{\mathsf{ps}}$*,* $\mathsf{dist}_{\mathsf{ps}}$ *and* $\mathsf{extend}_{\mathsf{ps}}$ *in the probability space monad:*

| | | | |
|---|---|---|---|
| $\mathsf{cond}_{\mathsf{ps}}$ | $=$ | | |
| $\mathsf{extend}_{\mathsf{ps}}$ | $=$ | $\equiv_{\mathsf{D}}$ | |
| $\mathsf{dist}_{\mathsf{ps}}$ | $\not\equiv_{\mathsf{D}}$ | $=$ | $=$ |
| | $\mathsf{cond}_{\mathsf{ps}}$ | $\mathsf{extend}_{\mathsf{ps}}$ | $\mathsf{dist}_{\mathsf{ps}}$ |

By Thm. 5.10, if we are to maintain the idea that theories are sets of facts, we cannot allow both conditioning and query statements.

## 5.7 Conclusions

For discrete Bayesian theories, we explained a large subclass of notation as measure-theoretic calculations by transformation into $\lambda_{\text{ZFC}}$. There is now at least one precisely defined set of expressions that denote discrete conditional distributions in conditional theories, and it is very large and expressive. We gave a converging approximating semantics and implemented it in Racket.

We could have interpreted notation as first-order set theory, in which measure theory is developed. Defining the exact semantics compositionally would have been difficult, and deriving an implementation from the semantics would have involved much hand-waving. By targeting $\lambda_{\text{ZFC}}$ instead, the path from notation to exact meaning to approximation to implementation is clear.

# Chapter 6

# Interlude: Uncountable Outcomes and Recursion

Now that we are satisfied that using $\lambda_{\text{ZFC}}$ as a target language for categorical semantics of constructive theories and queries works, we turn our attention to uncountable sample spaces and theories with general recursion.

It seems that, having followed measure-theoretic structure so far, the extension to uncountable sample spaces should be fairly smooth. Discrete probability spaces, probability mass functions, summation, conditioning, and discrete transition kernels all have uncountable analogues that can be composed in much the same way. The probability space monad thus has an uncountable analogue that can be used as a target for a categorial semantics for Bayesian notation. There are two difficulties, however.

The first difficulty is practical. As with the discrete probability space monad, we would like to *derive* an implementable semantics by approximating the target category. Unfortunately, this is complicated by the fact that the uncountable computations have large cardinalities. For example, a general probability space on $\mathbb{R}$ is defined as a triple $\langle \mathbb{R}, \mathit{\Sigma}, \mathsf{P} \rangle$, where $\mathit{\Sigma}$ is a subset of $\mathcal{P}\ \mathbb{R}$ and $\mathsf{P} : \mathit{\Sigma} \to [0, 1]$.

The second difficulty is theoretical. Suppose we define the following recursive function in a language with probabilistic choice, which counts the number of times $\mathsf{random} < \mathsf{p}$:

$$\mathsf{geometric\ p}\ :=\ \mathsf{if}\ (\mathsf{random} < \mathsf{p})\ 0\ (1 + \mathsf{geometric\ p}) \tag{6.1}$$

To interpret $\mathsf{geometric\ p}$ using the uncountable probability space monad, we must interpret both branches of the $\mathsf{if}$ as probability spaces and merge them. Unfortunately, doing so naïvely results in nontermination, as `geometric p` is applied in the *else* branch at every recurrence. Dealing with nontermination requires complicated fixpoint constructions and limits, which, with uncountable probability spaces, would put us on the frontier of research in mathematics instead of in computer science.

Fortunately, we can take a hint from measure-theoretic probability's general approach to infinite processes: define them with respect to a canonical, infinite-dimensional probability space, and encode branching and other complexities into the random variables. The next chapter takes this approach by interpreting whole programs as random variables in which every `random` expression indexes an infinite, random tree.

# Chapter 7

# Preimage Computation Theory: Running Programs Backwards

*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

## 7.1 Introduction

Measure-theoretic probability [27] is widely believed to be able to define every reasonable distribution, including distributions arising from discontinuous transformations and distributions on infinite spaces. It mainly does this by *assigning probabilities to sets*. Functions that do so are **probability measures**.

If a probability measure $\mathsf{P}$ assigns probabilities to subsets of $\mathsf{X}$ and $\mathsf{g} : \mathsf{X} \to \mathsf{Y}$, then the distribution over subsets of $\mathsf{Y}$ is

$$\mathsf{Pr}[\mathsf{B}] \;=\; \mathsf{P}\;(\mathsf{preimage}\ \mathsf{g}\ \mathsf{B}) \tag{7.1}$$

where $\mathsf{preimage}\ \mathsf{g}\ \mathsf{B} \;=\; \{\mathsf{a} \in \mathsf{domain}\ \mathsf{g} \mid \mathsf{g}\ \mathsf{a} \in \mathsf{B}\}$ is the subset of $\mathsf{X}$ for which $\mathsf{g}$ yields a value in $\mathsf{B}$. It is well-defined for any $\mathsf{g}$ and $\mathsf{B}$.

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

$$\mathsf{Pr}[\mathsf{B}' \,|\, \mathsf{B}] \;=\; \mathsf{Pr}[\mathsf{B}' \cap \mathsf{B}] \;/\; \mathsf{Pr}[\mathsf{B}] \tag{7.2}$$

If $\mathsf{Pr}[\mathsf{B}] = 0$, conditional probabilities can be calculated as the limit of $\mathsf{Pr}[\mathsf{B}' \,|\, \mathsf{B_n}]$ for certain positive-probability $\mathsf{B}_1 \supseteq \mathsf{B}_2 \supseteq \mathsf{B}_3 \supseteq \cdots$ whose intersection is $\mathsf{B}$ [46]. For example, if $\mathsf{Y} = \mathbb{R} \times \mathbb{R}$, the distribution of $\langle \mathsf{x}, \mathsf{y} \rangle \in \mathsf{Y}$ given $\mathsf{x} + \mathsf{y} = 0$ can be calculated using the descending sequence $\mathsf{B_n} = \{\langle \mathsf{x}, \mathsf{y} \rangle \in \mathsf{Y} \mid |\mathsf{x} + \mathsf{y}| < 2^{-\mathsf{n}}\}$.

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.

### 7.1.1 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, 48].

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

1. Preimages are definable only for functions with observable domains, which excludes lambdas.
2. If subsets of $\mathsf{X}$ and $\mathsf{Y}$ must be measurable, taking preimages under $\mathsf{g}$ must preserve measurability (we say $\mathsf{g}$ itself is measurable). Proving the conditions under which this is true is difficult, especially if $\mathsf{g}$ may not terminate.
3. It is difficult to define useful 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 backwards, efficiently, on possibly uncountable sets of outputs.

We address 1 and 4 by developing our semantics in $\lambda_{\text{ZFC}}$ [49], a $\lambda$-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 a proof of measurability. The outcome is worth it: 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. To maintain the flow of this chapter, we put it off until Chapter 9.

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

### 7.1.2 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 six arrows used to define the exact semantics:

$$\begin{tikzcd}
\mathsf{X} \rightsquigarrow_{\perp} \mathsf{Y} \arrow[r, "\mathsf{lift}_{\mathsf{map}}"] \arrow[d, "\eta_{\perp^*}"'] & \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y} \arrow[r, "\mathsf{lift}_{\mathsf{pre}}"] \arrow[d, "\eta_{\mathsf{map}^*}"] & \mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y} \arrow[d, "\eta_{\mathsf{pre}^*}"] \\
\mathsf{X} \rightsquigarrow_{\perp^*} \mathsf{Y} \arrow[r, "\mathsf{lift}_{\mathsf{map}^*}"'] & \mathsf{X} \underset{\mathsf{map}^*}{\rightsquigarrow} \mathsf{Y} \arrow[r, "\mathsf{lift}_{\mathsf{pre}^*}"'] & \mathsf{X} \underset{\mathsf{pre}^*}{\rightsquigarrow} \mathsf{Y}
\end{tikzcd} \tag{7.3}$$

At the top-left, $\mathsf{X} \rightsquigarrow_{\perp} \mathsf{Y}$ computations (or "bottom arrow computations") are intensional functions that may raise errors (i.e. return $\perp$, which is read "bottom"). From bottom arrow computations, the $\mathsf{lift}_{\mathsf{map}}$ combinator produces $\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}$ computations, which create equivalent extensional functions, or mappings. From mapping arrow computations, the $\mathsf{lift}_{\mathsf{pre}}$ combinator produces $\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}$ computations, which compute preimages.

Instances of arrows in the bottom row are like those in the top row, 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 combinator in (7.3) is a homomorphism; for example, that $\mathsf{lift}_{\mathsf{map}}$ distributes over all bottom arrow combinators.

The approximating semantics uses 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.

## 7.2 Arrows and First-Order Semantics

Like monads [54] and idioms [35], arrows [21] thread effects through computations in a way that imposes structure. But arrow computations are always

- Function-like: An arrow computation of type $\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}$ 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 $\mathsf{app}_{\mathsf{a}} : \langle \mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}, \mathsf{x} \rangle \rightsquigarrow_{\mathsf{a}} \mathsf{y}$ from the arrow $\mathsf{a}$'s minimal definition, so it is not possible for an arrow computation to apply another arrow computation.

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 it is difficult to define useful measurable sets of functions that make $\mathsf{app}$'s corresponding function measurable [5].

### 7.2.1 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 $\mathsf{a}$, instead of $\mathsf{first_a}$, 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 $\mathsf{a}$ is to define an additional combinator $\mathsf{left_a}$, which can be used to choose an arrow computation based on the result of another. Again, we define a different combinator, $\mathsf{ifte_a}$ ("if-then-else").

In a nonstrict $\lambda$-calculus, defining a choice combinator allows writing recursive functions using nothing but arrow combinators and lifted, pure functions. However, a strict $\lambda$-calculus needs an extra combinator $\mathsf{lazy}$ for deferring conditional branches. For example, define the **function arrow** with choice, in which $\mathsf{x} \rightsquigarrow \mathsf{y} \ ::= \ \mathsf{x} \Rightarrow \mathsf{y}$:

$$\begin{aligned}
\mathsf{arr\ f} \ &:= \ \mathsf{f} && \text{lift} \\
\mathsf{f_1} \ggg \mathsf{f_2} \ &:= \ \lambda\,\mathsf{a}.\,\mathsf{f_2}\ (\mathsf{f_1\ a}) && \text{composition} \\
\mathsf{f_1} \ \&\&\& \ \mathsf{f_2} \ &:= \ \lambda\,\mathsf{a}.\,\langle \mathsf{f_1\ a}, \mathsf{f_2}, \mathsf{a}\rangle && \text{pairing} \\
\mathsf{ifte\ f_1\ f_2\ f_3} \ &:= \ \lambda\,\mathsf{a}.\,\mathsf{if}\ (\mathsf{f_1\ a})\ (\mathsf{f_2\ a})\ (\mathsf{f_3\ a}) && \text{if-then-else} \\
\mathsf{lazy\ f} \ &:= \ \lambda\,\mathsf{a}.\,\mathsf{f}\ 0\ \mathsf{a} && \text{laziness}
\end{aligned} \tag{7.4}$$

and try to define the following recursive function:

$$\begin{aligned}
&\mathsf{halt\text{-}on\text{-}true} : \mathsf{Bool} \rightsquigarrow \mathsf{Bool} \qquad \text{(i.e. } \mathsf{halt\text{-}on\text{-}true} : \mathsf{Bool} \Rightarrow \mathsf{Bool}) \\
&\mathsf{halt\text{-}on\text{-}true} \ := \ \mathsf{ifte\ (arr\ id)\ (arr\ id)\ halt\text{-}on\text{-}true} \\
&\qquad\qquad\qquad \equiv \ \mathsf{ifte\ id\ id\ (ifte\ (arr\ id)\ (arr\ id)\ halt\text{-}on\text{-}true)} \\
&\qquad\qquad\qquad \equiv \ \mathsf{ifte\ id\ id\ (ifte\ id\ id\ (ifte\ (arr\ id)\ (arr\ id)\ halt\text{-}on\text{-}true))}
\end{aligned} \tag{7.5}$$

In a strict $\lambda$-calculus, the defining expression does not terminate. But the following is well-defined in $\lambda_{\mathrm{ZFC}}$, and loops only when applied to $\mathsf{false}$:

$$\begin{aligned}\mathsf{halt\text{-}on\text{-}true} \;&:=\; \mathsf{ifte}\ (\mathsf{arr}\ \mathsf{id})\ (\mathsf{arr}\ \mathsf{id})\ (\mathsf{lazy}\ \lambda 0.\, \mathsf{halt\text{-}on\text{-}true})\\ &\equiv\; \mathsf{ifte}\ \mathsf{id}\ \mathsf{id}\ (\lambda \mathsf{a}.\,(\lambda 0.\, \mathsf{halt\text{-}on\text{-}true})\ 0\ \mathsf{a})\\ &\equiv\; \lambda \mathsf{a}.\, \mathsf{if}\ (\mathsf{id}\ \mathsf{a})\ (\mathsf{id}\ \mathsf{a})\ ((\lambda \mathsf{a}.\,(\lambda 0.\, \mathsf{halt\text{-}on\text{-}true})\ 0\ \mathsf{a})\ \mathsf{a})\\ &\equiv\; \lambda \mathsf{a}.\, \mathsf{if}\ \mathsf{a}\ \mathsf{a}\ ((\lambda \mathsf{a}.\, \mathsf{halt\text{-}on\text{-}true}\ \mathsf{a})\ \mathsf{a})\\ &\equiv\; \lambda \mathsf{a}.\, \mathsf{if}\ \mathsf{a}\ \mathsf{a}\ (\mathsf{halt\text{-}on\text{-}true}\ \mathsf{a})\end{aligned} \tag{7.6}$$

All of our arrows are arrows with choice and $\mathsf{lazy}$, so we simply call them arrows.

**Definition 7.1** (arrow)**.** *Let* $1 := \{0\}$ *(Section 3.3.1). A binary type constructor* $(\rightsquigarrow_\mathsf{a})$ *and*

$$\begin{aligned}\mathsf{arr_a} &: (\mathsf{x} \Rightarrow \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{y}) && \textit{lift}\\ (\ggg_\mathsf{a}) &: (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{y}) \Rightarrow (\mathsf{y} \rightsquigarrow_\mathsf{a} \mathsf{z}) \Rightarrow (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{z}) && \textit{composition}\\ (\&\&\&_\mathsf{a}) &: (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{z}) \Rightarrow (\mathsf{x} \rightsquigarrow_\mathsf{a} \langle \mathsf{y}, \mathsf{z}\rangle) && \textit{pairing}\\ \mathsf{ifte_a} &: (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{Bool}) \Rightarrow (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{y}) && \textit{if-then-else}\\ \mathsf{lazy_a} &: (1 \Rightarrow (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{y})) \Rightarrow (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{y}) && \textit{laziness}\end{aligned} \tag{7.7}$$

*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 7.2** (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* $\mathsf{a}$ *to arrow* $\mathsf{b}$ *if the following distributive laws hold for* *appropriately typed* $\mathsf{f}$, $\mathsf{f}_1$, $\mathsf{f}_2$ *and* $\mathsf{f}_3$*:*

$$\mathsf{lift_b}\ (\mathsf{arr_a}\ \mathsf{f}) \equiv \mathsf{arr_b}\ \mathsf{f} \tag{7.8}$$

$$\mathsf{lift_b}\ (\mathsf{f}_1 \ggg_\mathsf{a} \mathsf{f}_2) \equiv (\mathsf{lift_b}\ \mathsf{f}_1) \ggg_\mathsf{b} (\mathsf{lift_b}\ \mathsf{f}_2) \tag{7.9}$$

$$\mathsf{lift_b}\ (\mathsf{f}_1\ \&\&\&_\mathsf{a}\ \mathsf{f}_2) \equiv (\mathsf{lift_b}\ \mathsf{f}_1)\ \&\&\&_\mathsf{b}\ (\mathsf{lift_b}\ \mathsf{f}_2) \tag{7.10}$$

$$\mathsf{lift_b}\ (\mathsf{ifte_a}\ \mathsf{f}_1\ \mathsf{f}_2\ \mathsf{f}_3) \equiv \mathsf{ifte_b}\ (\mathsf{lift_b}\ \mathsf{f}_1)\ (\mathsf{lift_b}\ \mathsf{f}_2)\ (\mathsf{lift_b}\ \mathsf{f}_3) \tag{7.11}$$

$$\mathsf{lift_b}\ (\mathsf{lazy_a}\ \mathsf{f}) \equiv \mathsf{lazy_b}\ \lambda 0.\, \mathsf{lift_b}\ (\mathsf{f}\ 0) \tag{7.12}$$

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 (7.4) to arrow $\mathsf{a}$. 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 $(\ggg_\mathsf{a})$ and a pair extraction law:

$$(\mathsf{f}_1 \ggg_\mathsf{a} \mathsf{f}_2) \ggg_\mathsf{a} \mathsf{f}_3 \equiv \mathsf{f}_1 \ggg_\mathsf{a} (\mathsf{f}_2 \ggg_\mathsf{a} \mathsf{f}_3) \tag{7.13}$$

$$(\mathsf{arr_a}\ \mathsf{f}_1\ \&\&\&_\mathsf{a}\ \mathsf{f}_2) \ggg_\mathsf{a} \mathsf{arr_a}\ \mathsf{snd} \equiv \mathsf{f}_2 \tag{7.14}$$

and distribution of pure computations over effectful:

$$\mathsf{arr_a}\ \mathsf{f}_1 \ggg_\mathsf{a} (\mathsf{f}_2\ \&\&\&_\mathsf{a}\ \mathsf{f}_3) \equiv (\mathsf{arr_a}\ \mathsf{f}_1 \ggg_\mathsf{a} \mathsf{f}_2)\ \&\&\&_\mathsf{a}\ (\mathsf{arr_a}\ \mathsf{f}_1 \ggg_\mathsf{a} \mathsf{f}_3) \tag{7.15}$$

$$\begin{aligned}\mathsf{arr_a}\ \mathsf{f}_1 \ggg_\mathsf{a} \mathsf{ifte_a}\ \mathsf{f}_2\ \mathsf{f}_3\ \mathsf{f}_4 \equiv\ & \mathsf{ifte_a}\ (\mathsf{arr_a}\ \mathsf{f}_1 \ggg_\mathsf{a} \mathsf{f}_2) \\ & \qquad (\mathsf{arr_a}\ \mathsf{f}_1 \ggg_\mathsf{a} \mathsf{f}_3) \\ & \qquad (\mathsf{arr_a}\ \mathsf{f}_1 \ggg_\mathsf{a} \mathsf{f}_4)\end{aligned} \tag{7.16}$$

$$\mathsf{arr_a}\ \mathsf{f}_1 \ggg_\mathsf{a} \mathsf{lazy_a}\ \mathsf{f}_2 \equiv \mathsf{lazy_a}\ \lambda 0.\, \mathsf{arr_a}\ \mathsf{f}_1 \ggg_\mathsf{a} \mathsf{f}_2\ 0 \tag{7.17}$$

Equivalence between different arrow representations is usually proved in a strongly normalizing $\lambda$-calculus [31, 32], in which every function is free of effects, including nontermination. Such a $\lambda$-calculus has no need for $\mathsf{lazy_a}$, so we could not derive (7.17) 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 (7.14), which *can* be derived from existing arrow laws, is a more problematic, in nonstrict $\lambda$-calculii as well as $\lambda_{\text{ZFC}}$. If $\mathsf{f}_1$ does not always terminate, using (7.14) to transform a computation can turn a nonterminating expression into a terminating one, or vice-versa. We could require $\mathsf{f}_1$ in the pair extraction law to always terminate. Instead, we require every argument to $\mathsf{arr}_\mathsf{a}$ 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 7.3** (arrow epimorphism)**.** *An arrow homomorphism* $\mathsf{lift}_\mathsf{b} : (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_\mathsf{b} \mathsf{y})$ *that has a right inverse is an* ***arrow epimorphism*** *from* a *to* b*.*

**Theorem 7.4** (epimorphism implies arrow laws)**.** *If* $\mathsf{lift}_\mathsf{b} : (\mathsf{x} \rightsquigarrow_\mathsf{a} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_\mathsf{b} \mathsf{y})$ *is an arrow epimorphism and the combinators of* a *define an arrow, then the combinators of* b *define an arrow.*

*Proof.* Let $\mathsf{lift}_\mathsf{b}^{-1}$ be $\mathsf{lift}_\mathsf{b}$'s right inverse. For the pair extraction law (7.14),

$$
\begin{aligned}
&(\mathsf{arr}_\mathsf{b}\ \mathsf{f}_1\ \&\&\&_\mathsf{b}\ \mathsf{f}_2) \ggg_\mathsf{b} \mathsf{arr}_\mathsf{b}\ \mathsf{snd} && (7.18)\\
&\quad\equiv (\mathsf{lift}_\mathsf{b}\ (\mathsf{arr}_\mathsf{a}\ \mathsf{f}_1)\ \&\&\&_\mathsf{b}\ (\mathsf{lift}_\mathsf{b}\ (\mathsf{lift}_\mathsf{b}^{-1}\ \mathsf{f}_2))) \ggg_\mathsf{b} \mathsf{lift}_\mathsf{b}\ (\mathsf{arr}_\mathsf{a}\ \mathsf{snd}) && \text{Rewrite with } \mathsf{lift}_\mathsf{b}\\
&\quad\equiv \mathsf{lift}_\mathsf{b}\ (\mathsf{arr}_\mathsf{a}\ \mathsf{f}_1\ \&\&\&_\mathsf{a}\ \mathsf{lift}_\mathsf{b}^{-1}\ \mathsf{f}_2) \ggg_\mathsf{b} \mathsf{lift}_\mathsf{b}\ (\mathsf{arr}_\mathsf{a}\ \mathsf{snd}) && \text{Homomorphism (7.10)}\\
&\quad\equiv \mathsf{lift}_\mathsf{b}\ ((\mathsf{arr}_\mathsf{a}\ \mathsf{f}_1\ \&\&\&_\mathsf{a}\ \mathsf{lift}_\mathsf{b}^{-1}\ \mathsf{f}_2) \ggg_\mathsf{a} \mathsf{arr}_\mathsf{a}\ \mathsf{snd}) && \text{Homomorphism (7.9)}\\
&\quad\equiv \mathsf{lift}_\mathsf{b}\ (\mathsf{lift}_\mathsf{b}^{-1}\ \mathsf{f}_2) && \text{Pair extraction (7.14)}\\
&\quad\equiv \mathsf{f}_2 && \text{Right inverse}
\end{aligned}
$$

The proofs for every other law are similar. □

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

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

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

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

Figure 7.1: Interpretation of a let-calculus with first-order definitions and De-Bruijn-indexed bindings as arrow a computations.

### 7.2.2 First-Order Let-Calculus Semantics

Figure 7.1 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 $\mathsf{y}$ 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 $\mathsf{x}$ is the argument type and $\mathsf{y}$ is the return type. Application sends a stack containing just an $\mathsf{x}$.

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

**Definition 7.5** (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 $\mathsf{a}$ 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 7.6** (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. Base cases proceed by expansion and using $\mathsf{arr_b} \equiv \mathsf{lift_b} \circ \mathsf{arr_a}$ (7.8). For example, for constants:

$$\begin{aligned}
[\![v]\!]_{\mathsf{b}} &\equiv \mathsf{arr_b}\ (\mathsf{const}\ v) && \text{Def of } [\![\cdot]\!]_{\mathsf{b}} \\
&\equiv \mathsf{lift_b}\ (\mathsf{arr_a}\ (\mathsf{const}\ v)) && \text{Homomorphism (7.8)} \\
&\equiv \mathsf{lift_b}\ [\![v]\!]_{\mathsf{a}} && \text{Def of } [\![\cdot]\!]_{\mathsf{a}}
\end{aligned} \tag{7.19}$$

Inductive cases proceed by expansion, applying the inductive hypothesis on subterms, and applying distributive laws (7.9)–(7.12). For example, for pairing:

$$\begin{aligned}
[\![\langle e_1, e_2\rangle]\!]_{\mathsf{b}} &\equiv [\![e_1]\!]_{\mathsf{b}}\ \&\&\&_{\mathsf{b}}\ [\![e_2]\!]_{\mathsf{b}} && \text{Def of } [\![\cdot]\!]_{\mathsf{b}} \\
&\equiv (\mathsf{lift_b}\ [\![e_1]\!]_{\mathsf{a}})\ \&\&\&_{\mathsf{b}}\ (\mathsf{lift_b}\ [\![e_2]\!]_{\mathsf{a}}) && \text{Ind hypothesis} \\
&\equiv \mathsf{lift_b}\ ([\![e_1]\!]_{\mathsf{a}}\ \&\&\&_{\mathsf{a}}\ [\![e_2]\!]_{\mathsf{a}}) && \text{Homomorphism (7.10)} \\
&\equiv \mathsf{lift_b}\ [\![\langle e_1, e_2\rangle]\!]_{\mathsf{a}} && \text{Def of } [\![\cdot]\!]_{\mathsf{a}}
\end{aligned} \tag{7.20}$$

It is not hard to check the remaining cases. □

If we assume $\mathsf{lift_b}$ defines correct behavior for arrow $\mathsf{b}$ in terms of arrow $\mathsf{a}$, and prove that $\mathsf{lift_b}$ is a homomorphism, then by Theorem 7.6, $[\![\cdot]\!]_{\mathsf{b}}$ is correct.

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

$$\begin{aligned}
&\mathsf{arr}_{\bot} : (\mathsf{X} \Rightarrow \mathsf{Y}) \Rightarrow (\mathsf{X} \rightsquigarrow_{\bot} \mathsf{Y}) \\
&\mathsf{arr}_{\bot}\ \mathsf{f} \ := \ \mathsf{f}
\end{aligned}$$

$$\begin{aligned}
&(\ggg_{\bot}) : (\mathsf{X} \rightsquigarrow_{\bot} \mathsf{Y}) \Rightarrow (\mathsf{Y} \rightsquigarrow_{\bot} \mathsf{Z}) \Rightarrow (\mathsf{X} \rightsquigarrow_{\bot} \mathsf{Z}) \\
&(\mathsf{f}_1 \ggg_{\bot} \mathsf{f}_2)\ \mathsf{a} \ := \ \mathsf{if}\ (\mathsf{f}_1\ \mathsf{a} = \bot)\ \bot\ (\mathsf{f}_2\ (\mathsf{f}_1\ \mathsf{a}))
\end{aligned}$$

$$\begin{aligned}
&(\&\&\&_{\bot}) : (\mathsf{X} \rightsquigarrow_{\bot} \mathsf{Y}_1) \Rightarrow (\mathsf{X} \rightsquigarrow_{\bot} \mathsf{Y}_2) \Rightarrow (\mathsf{X} \rightsquigarrow_{\bot} \langle \mathsf{Y}_1, \mathsf{Y}_2 \rangle) \\
&(\mathsf{f}_1 \ \&\&\&_{\bot}\ \mathsf{f}_2)\ \mathsf{a} \ := \ \mathsf{let}\ \ \mathsf{b}_1 := \mathsf{f}_1\ \mathsf{a} \\
&\qquad\qquad\qquad\qquad\quad\ \ \mathsf{b}_2 := \mathsf{f}_2\ \mathsf{a} \\
&\qquad\qquad\qquad\qquad\ \mathsf{in}\ \ \mathsf{if}\ (\mathsf{b}_1 = \bot\ \mathsf{or}\ \mathsf{b}_2 = \bot)\ \bot\ \langle \mathsf{b}_1, \mathsf{b}_2 \rangle
\end{aligned}$$

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

$$\begin{aligned}
&\mathsf{lazy}_{\bot} : (1 \Rightarrow (\mathsf{X} \rightsquigarrow_{\bot} \mathsf{Y})) \Rightarrow (\mathsf{X} \rightsquigarrow_{\bot} \mathsf{Y}) \\
&\mathsf{lazy}_{\bot}\ \mathsf{f}\ \mathsf{a} \ := \ \mathsf{f}\ 0\ \mathsf{a}
\end{aligned}$$

Figure 7.2: Bottom arrow definitions.

## 7.3 The Bottom Arrow

Using the diagram in (7.3) as a sort of map, we start in the upper-left corner:

$$\begin{tikzcd}
\mathsf{X} \rightsquigarrow_{\bot} \mathsf{Y} \arrow[r, "\mathsf{lift}_{\mathsf{map}}"] \arrow[d, "\eta_{\bot^*}"'] & \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y} \arrow[r, "\mathsf{lift}_{\mathsf{pre}}"] \arrow[d, "\eta_{\mathsf{map}^*}"] & \mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y} \arrow[d, "\eta_{\mathsf{pre}^*}"] \\
\mathsf{X} \rightsquigarrow_{\bot^*} \mathsf{Y} \arrow[r, "\mathsf{lift}_{\mathsf{map}^*}"'] & \mathsf{X} \underset{\mathsf{map}^*}{\rightsquigarrow} \mathsf{Y} \arrow[r, "\mathsf{lift}_{\mathsf{pre}^*}"'] & \mathsf{X} \underset{\mathsf{pre}^*}{\rightsquigarrow} \mathsf{Y}
\end{tikzcd} \tag{7.21}$$

Through Section 7.6, we move across the top to $\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}$.

To use Theorem 7.6 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 (7.4) is an obvious candidate. However, we will need to explicitly handle nontermination as an error value, so we need a slightly more complicated arrow.

Figure 7.2 defines the ***bottom arrow***. Its computations have type $\mathsf{X} \rightsquigarrow_{\bot} \mathsf{Y} ::= \mathsf{X} \Rightarrow \mathsf{Y}_{\bot}$, where $\mathsf{Y}_{\bot} ::= \mathsf{Y} \cup \{\bot\}$ and $\bot$ is a distinguished error value. The type $\mathsf{Bool}_{\bot}$, for example, denotes the members of $\mathsf{Bool} \cup \{\bot\} = \{\mathsf{true}, \mathsf{false}, \bot\}$.

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

**Definition 7.7** (bottom arrow equivalence)**.** *Two computations* $\mathsf{f}_1 : \mathsf{X} \rightsquigarrow_{\bot} \mathsf{Y}$ *and* $\mathsf{f}_2 : \mathsf{X} \rightsquigarrow_{\bot} \mathsf{Y}$

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

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

## 7.4 Deriving the Mapping Arrow

Computing preimages requires an observable domain, which lambdas do not have. Further, 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. $\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y} ::= \mathsf{X} \rightharpoonup \mathsf{Y}$. Unfortunately, we could not define $\mathsf{arr}_{\mathsf{map}} : (\mathsf{X} \Rightarrow \mathsf{Y}) \Rightarrow (\mathsf{X} \rightharpoonup \mathsf{Y})$: 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

$$\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y} \ ::= \ \mathsf{Set}\ \mathsf{X} \Rightarrow (\mathsf{X} \rightharpoonup \mathsf{Y}) \tag{7.22}$$

The absence of $\perp$ in $\mathsf{Set}\ \mathsf{X} \Rightarrow (\mathsf{X} \rightharpoonup \mathsf{Y})$, and the fact that type parameters $\mathsf{X}$ and $\mathsf{Y}$ 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 7.6 to prove that expressions interpreted using $[\![\cdot]\!]_{\mathsf{map}}$ behave correctly with respect to $[\![\cdot]\!]_\perp$, we need to define correctness using a lift from the bottom arrow to the mapping arrow. It is helpful to have a standalone function $\mathsf{domain}_\perp$ that computes the subset

$\mathsf{range} : (\mathsf{X} \rightharpoonup \mathsf{Y}) \Rightarrow \mathsf{Set\ Y}$
$\mathsf{range\ g} := \mathsf{image\ snd\ g}$

$(\circ_{\mathsf{map}}) : (\mathsf{Y} \rightharpoonup \mathsf{Z}) \Rightarrow (\mathsf{X} \rightharpoonup \mathsf{Y}) \Rightarrow (\mathsf{X} \rightharpoonup \mathsf{Z})$
$\mathsf{g_2} \circ_{\mathsf{map}} \mathsf{g_1} := \mathsf{let\ \ A := preimage\ g_1\ (domain\ g_2)}$
$\qquad\qquad\qquad \mathsf{in}\ \ \lambda \mathsf{a} \in \mathsf{A}.\, \mathsf{g_2\ (g_1\ a)}$

$\langle \cdot, \cdot \rangle_{\mathsf{map}} : (\mathsf{X} \rightharpoonup \mathsf{Y_1}) \Rightarrow (\mathsf{X} \rightharpoonup \mathsf{Y_2}) \Rightarrow (\mathsf{X} \rightharpoonup \mathsf{Y_1} \times \mathsf{Y_2})$
$\langle \mathsf{g_1}, \mathsf{g_2} \rangle_{\mathsf{map}} := \mathsf{let\ \ A := domain\ g_1} \cap \mathsf{domain\ g_2}$
$\qquad\qquad\qquad \mathsf{in}\ \ \lambda \mathsf{a} \in \mathsf{A}.\, \langle \mathsf{g_1\ a}, \mathsf{g_2\ a} \rangle$

$(\uplus_{\mathsf{map}}) : (\mathsf{X} \rightharpoonup \mathsf{Y}) \Rightarrow (\mathsf{X} \rightharpoonup \mathsf{Y}) \Rightarrow (\mathsf{X} \rightharpoonup \mathsf{Y})$
$\mathsf{g_1} \uplus_{\mathsf{map}} \mathsf{g_2} := \mathsf{let\ \ A := domain\ g_1} \uplus \mathsf{domain\ g_2}$
$\qquad\qquad\qquad \mathsf{in}\ \ \lambda \mathsf{a} \in \mathsf{A}.\, \mathsf{if\ (a} \in \mathsf{domain\ g_1)\ (g_1\ a)\ (g_2\ a)}$

Figure 7.3: Additional operations on partial mappings.

of $\mathsf{A}$ on which $\mathsf{f}$ does not return $\bot$. We define that first, and then define $\mathsf{lift_{map}}$ in terms of it:

$$\begin{aligned} &\mathsf{domain}_\bot : (\mathsf{X} \rightsquigarrow_\bot \mathsf{Y}) \Rightarrow \mathsf{Set\ X} \Rightarrow \mathsf{Set\ X} \\ &\mathsf{domain}_\bot\ \mathsf{f\ A} := \{\mathsf{a} \in \mathsf{A} \mid \mathsf{f\ a} \neq \bot\} \end{aligned} \tag{7.23}$$

$$\begin{aligned} &\mathsf{lift_{map}} : (\mathsf{X} \rightsquigarrow_\bot \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}) \\ &\mathsf{lift_{map}\ f\ A} := \mathsf{mapping\ f\ (domain_\bot\ f\ A)} \end{aligned} \tag{7.24}$$

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

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

**Definition 7.9** (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* $\mathsf{g_1\ A} \equiv \mathsf{g_2\ A}$ *for all* $\mathsf{A} \subseteq \mathsf{X}$.

Clearly $\mathsf{arr_{map}} := \mathsf{lift_{map}} \circ \mathsf{arr_\bot}$ meets the first homomorphism law (7.8). The remainder of this section derives $(\&\&\&_{\mathsf{map}})$, $(\ggg_{\mathsf{map}})$, $\mathsf{ifte_{map}}$ and $\mathsf{lazy_{map}}$ from bottom arrow combinators, in a way that ensures $\mathsf{lift_{map}}$ is an arrow homomorphism. Figure 7.3 defines the additional necessary mapping operations $\mathsf{range}$, composition, pairing, and disjoint union, and Figure 7.4 contains the resulting mapping arrow combinators.

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

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

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

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

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

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

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

Figure 7.4: Mapping arrow definitions.

### 7.4.1 Composition

Starting with the left side of (7.9), we expand definitions, simplify $\mathsf{f}$ by restricting it to a set for which $\mathsf{f_1\ a} \neq \bot$:

$$\mathsf{lift_{map}}\ (\mathsf{f_1} \ggg_{\bot} \mathsf{f_2})\ \mathsf{A} \tag{7.25}$$

$$\begin{aligned}
\equiv\ &\mathsf{let}\ \ \ \mathsf{f} := \lambda \mathsf{a}.\, \mathsf{if}\ (\mathsf{f_1\ a} = \bot)\ \bot\ (\mathsf{f_2}\ (\mathsf{f_1\ a})) && \text{Def of } \mathsf{lift_{map}},\ (\ggg_{\bot}) \\
&\qquad \mathsf{A'} := \mathsf{domain_{\bot}\ f\ A} \\
&\ \ \mathsf{in}\ \ \mathsf{mapping\ f\ A'} \\[1em]
\equiv\ &\mathsf{let}\ \ \ \mathsf{f} := \lambda \mathsf{a}.\, \mathsf{f_2}\ (\mathsf{f_1\ a}) && \text{Simplify } \mathsf{f} \\
&\qquad \mathsf{A'} := \mathsf{domain_{\bot}\ f\ (domain_{\bot}\ f_1\ A)} \\
&\ \ \mathsf{in}\ \ \mathsf{mapping\ f\ A'} \\[1em]
\equiv\ &\mathsf{let}\ \ \mathsf{A'} := \{\mathsf{a} \in \mathsf{domain_{\bot}\ f_1\ A} \mid \mathsf{f_2}\ (\mathsf{f_1\ a}) \neq \bot\} && \text{Def of } \mathsf{domain_{\bot}},\ \mathsf{mapping} \\
&\ \ \mathsf{in}\ \ \lambda \mathsf{a} \in \mathsf{A'}.\, \mathsf{f_2}\ (\mathsf{f_1\ a})
\end{aligned}$$

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

$$\begin{aligned}
\equiv\ &\mathsf{let}\ \ \mathsf{g_1} := \mathsf{lift_{map}\ f_1\ A} && \text{Rewrite with } \mathsf{lift_{map}} \\
&\qquad \mathsf{A'} := \mathsf{preimage\ g_1\ (domain_{\bot}\ f_2\ (range\ g_1))} \\
&\ \ \mathsf{in}\ \ \lambda \mathsf{a} \in \mathsf{A'}.\, \mathsf{f_2}\ (\mathsf{g_1\ a})
\end{aligned}$$

$$
\begin{aligned}
&\equiv \ \mathsf{let}\ \ \mathsf{g}_1 := \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_1\ \mathsf{A} && \text{Rewrite with } \mathsf{lift}_{\mathsf{map}}\\
&\qquad\quad \mathsf{g}_2 := \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_2\ (\mathsf{range}\ \mathsf{g}_1)\\
&\qquad\quad \mathsf{A}' := \mathsf{preimage}\ \mathsf{g}_1\ (\mathsf{domain}\ \mathsf{g}_2)\\
&\qquad \mathsf{in}\ \ \lambda\,\mathsf{a} \in \mathsf{A}'.\, \mathsf{g}_2\ (\mathsf{g}_1\ \mathsf{a})
\end{aligned}
$$

$$
\begin{aligned}
&\equiv \ \mathsf{let}\ \ \mathsf{g}_1 := \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_1\ \mathsf{A} && \text{Rewrite with } (\circ_{\mathsf{map}})\\
&\qquad\quad \mathsf{g}_2 := \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_2\ (\mathsf{range}\ \mathsf{g}_1)\\
&\qquad \mathsf{in}\ \ \mathsf{g}_2 \circ_{\mathsf{map}} \mathsf{g}_1
\end{aligned}
$$

Substituting $\mathsf{g}_1$ for $\mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_1$ and $\mathsf{g}_2$ for $\mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_2$ gives a definition for $(\ggg_{\mathsf{map}})$ (Figure 7.4) for which (7.9) holds.

### 7.4.2 Pairing

Starting with the left side of (7.10), we expand definitions, and simplify $\mathsf{f}$ by restricting it to a set for which $\mathsf{f}_1\ \mathsf{a} \neq \bot$ and $\mathsf{f}_2\ \mathsf{a} \neq \bot$:

$$
\mathsf{lift}_{\mathsf{map}}\ (\mathsf{f}_1\ \&\&\&_{\bot}\ \mathsf{f}_2)\ \mathsf{A} \tag{7.26}
$$

$$
\begin{aligned}
&\equiv \ \mathsf{let}\ \ \ \mathsf{f} := \lambda\,\mathsf{a}.\,\mathsf{let}\ \ \mathsf{b}_1 := \mathsf{f}_1\ \mathsf{a} && \text{Def of } \mathsf{lift}_{\mathsf{map}},\ (\&\&\&_{\bot})\\
&\qquad\qquad\qquad\qquad\ \ \mathsf{b}_2 := \mathsf{f}_2\ \mathsf{a}\\
&\qquad\qquad\qquad\quad\ \mathsf{in}\ \ \mathsf{if}\ (\mathsf{b}_1 = \bot\ \mathsf{or}\ \mathsf{b}_2 = \bot)\ \bot\ \langle \mathsf{b}_1, \mathsf{b}_2\rangle\\
&\qquad\quad \mathsf{A}' := \mathsf{domain}_{\bot}\ \mathsf{f}\ \mathsf{A}\\
&\qquad \mathsf{in}\ \ \mathsf{mapping}\ \mathsf{f}\ \mathsf{A}'
\end{aligned}
$$

$$
\begin{aligned}
&\equiv \ \mathsf{let}\ \ \ \mathsf{f} := \lambda\,\mathsf{a}.\,\langle \mathsf{f}_1\ \mathsf{a}, \mathsf{f}_2\ \mathsf{a}\rangle && \text{Simplify } \mathsf{f}\\
&\qquad\quad \mathsf{A}' := \mathsf{domain}_{\bot}\ \mathsf{f}_1\ \mathsf{A} \cap \mathsf{domain}_{\bot}\ \mathsf{f}_2\ \mathsf{A}\\
&\qquad \mathsf{in}\ \ \mathsf{mapping}\ \mathsf{f}\ \mathsf{A}'
\end{aligned}
$$

$$
\begin{aligned}
&\equiv \ \mathsf{let}\ \ \mathsf{A}' := \mathsf{domain}_{\bot}\ \mathsf{f}_1\ \mathsf{A} \cap \mathsf{domain}_{\bot}\ \mathsf{f}_2\ \mathsf{A} && \text{Def of } \mathsf{mapping}\\
&\qquad \mathsf{in}\ \ \lambda\,\mathsf{a} \in \mathsf{A}'.\,\langle \mathsf{f}_1\ \mathsf{a}, \mathsf{f}_2\ \mathsf{a}\rangle
\end{aligned}
$$

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

$$\begin{aligned}
\equiv\ & \mathsf{let}\ \ \mathsf{g}_1 := \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_1\ \mathsf{A} && \text{Rewrite with } \mathsf{lift}_{\mathsf{map}}\\
& \qquad \mathsf{g}_2 := \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_2\ \mathsf{A}\\
& \qquad \mathsf{A}' := \mathsf{domain}\ \mathsf{g}_1 \cap \mathsf{domain}\ \mathsf{g}_2\\
& \quad \mathsf{in}\ \ \lambda\,\mathsf{a} \in \mathsf{A}'.\,\langle \mathsf{g}_1\ \mathsf{a}, \mathsf{g}_2\ \mathsf{a}\rangle\\
\equiv\ & \langle \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_1\ \mathsf{A}, \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_2\ \mathsf{A}\rangle_{\mathsf{map}} && \text{Rewrite with } \langle\cdot,\cdot\rangle_{\mathsf{map}}
\end{aligned}$$

Substituting $\mathsf{g}_1$ for $\mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_1$ and $\mathsf{g}_2$ for $\mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_2$ gives a definition for $(\&\&\&_{\mathsf{map}})$ (Figure 7.4) for which (7.10) holds.

### 7.4.3 Conditional

Starting with the left side of (7.11), we expand definitions, and simplify $\mathsf{f}$ by restricting it to a domain for which $\mathsf{f}_1\ \mathsf{a} \neq \bot$:

$$\mathsf{lift}_{\mathsf{map}}\ (\mathsf{ifte}_{\bot}\ \mathsf{f}_1\ \mathsf{f}_2\ \mathsf{f}_3)\ \mathsf{A} \tag{7.27}$$

$$\begin{aligned}
\equiv\ & \mathsf{let}\ \ \ \mathsf{f} := \lambda\,\mathsf{a}.\,\mathsf{case}\ \ \mathsf{f}_1\ \mathsf{a} && \text{Def of } \mathsf{lift}_{\mathsf{map}},\ \mathsf{ifte}_{\bot}\\
& \qquad\qquad\qquad \mathsf{true} \longrightarrow \mathsf{f}_2\ \mathsf{a}\\
& \qquad\qquad\qquad \mathsf{false} \longrightarrow \mathsf{f}_3\ \mathsf{a}\\
& \qquad\qquad\qquad \bot \longrightarrow \bot\\
& \qquad \mathsf{A}' := \mathsf{domain}_{\bot}\ \mathsf{f}\ \mathsf{A}\\
& \quad \mathsf{in}\ \ \mathsf{mapping}\ \mathsf{f}\ \mathsf{A}'\\
\equiv\ & \mathsf{let}\ \ \ \mathsf{f} := \lambda\,\mathsf{a}.\,\mathsf{if}\ (\mathsf{f}_1\ \mathsf{a})\ (\mathsf{f}_2\ \mathsf{a})\ (\mathsf{f}_3\ \mathsf{a}) && \text{Simplify } \mathsf{f}\\
& \qquad \mathsf{g}_1 := \mathsf{mapping}\ \mathsf{f}_1\ (\mathsf{domain}_{\bot}\ \mathsf{f}_1\ \mathsf{A})\\
& \qquad \mathsf{A}_2 := \mathsf{preimage}\ \mathsf{g}_1\ \{\mathsf{true}\}\\
& \qquad \mathsf{A}_3 := \mathsf{preimage}\ \mathsf{g}_1\ \{\mathsf{false}\}\\
& \qquad \mathsf{A}' := \mathsf{domain}_{\bot}\ \mathsf{f}_2\ \mathsf{A}_2 \uplus \mathsf{domain}_{\bot}\ \mathsf{f}_3\ \mathsf{A}_3\\
& \quad \mathsf{in}\ \ \mathsf{mapping}\ \mathsf{f}\ \mathsf{A}'\\
\equiv\ & \mathsf{let}\ \ \mathsf{g}_1 := \mathsf{mapping}\ \mathsf{f}_1\ (\mathsf{domain}_{\bot}\ \mathsf{f}_1\ \mathsf{A}) && \text{Def of } \mathsf{mapping}\\
& \qquad \mathsf{A}_2 := \mathsf{preimage}\ \mathsf{g}_1\ \{\mathsf{true}\}\\
& \qquad \mathsf{A}_3 := \mathsf{preimage}\ \mathsf{g}_1\ \{\mathsf{false}\}\\
& \qquad \mathsf{A}' := \mathsf{domain}_{\bot}\ \mathsf{f}_2\ \mathsf{A}_2 \uplus \mathsf{domain}_{\bot}\ \mathsf{f}_3\ \mathsf{A}_3\\
& \quad \mathsf{in}\ \ \lambda\,\mathsf{a} \in \mathsf{A}'.\,\mathsf{if}\ (\mathsf{f}_1\ \mathsf{a})\ (\mathsf{f}_2\ \mathsf{a})\ (\mathsf{f}_3\ \mathsf{a})
\end{aligned}$$

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

$$\begin{array}{lll} \equiv & \mathsf{let} & \mathsf{g}_1 := \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_1\ \mathsf{A} \\ & & \mathsf{g}_2 := \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_2\ (\mathsf{preimage}\ \mathsf{g}_1\ \{\mathsf{true}\}) \\ & & \mathsf{g}_3 := \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_3\ (\mathsf{preimage}\ \mathsf{g}_1\ \{\mathsf{false}\}) \\ & & \mathsf{A}' := \mathsf{domain}\ \mathsf{g}_2 \uplus \mathsf{domain}\ \mathsf{g}_3 \\ & \mathsf{in} & \lambda\,\mathsf{a} \in \mathsf{A}'.\,\mathsf{if}\ (\mathsf{a} \in \mathsf{domain}\ \mathsf{g}_2)\ (\mathsf{g}_2\ \mathsf{a})\ (\mathsf{g}_3\ \mathsf{a}) \end{array} \qquad \text{Rewrite with } \mathsf{lift}_{\mathsf{map}}$$

$$\begin{array}{lll} \equiv & \mathsf{let} & \mathsf{g}_1 := \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_1\ \mathsf{A} \\ & & \mathsf{g}_2 := \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_2\ (\mathsf{preimage}\ \mathsf{g}_1\ \{\mathsf{true}\}) \\ & & \mathsf{g}_3 := \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_3\ (\mathsf{preimage}\ \mathsf{g}_1\ \{\mathsf{false}\}) \\ & \mathsf{in} & \mathsf{g}_2 \uplus_{\mathsf{map}} \mathsf{g}_3 \end{array} \qquad \text{Rewrite with } (\uplus_{\mathsf{map}})$$

Substituting $\mathsf{g}_1$ for $\mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_1$, $\mathsf{g}_2$ for $\mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_2$, and $\mathsf{g}_3$ for $\mathsf{lift}_{\mathsf{map}}\ \mathsf{f}_3$ gives a definition for $\mathsf{ifte}_{\mathsf{map}}$ (Figure 7.4) for which (7.11) holds.

### 7.4.4 Laziness

Starting with the left side of (7.12), we expand definitions:

$$\mathsf{lift}_{\mathsf{map}}\ (\mathsf{lazy}_{\perp}\ \mathsf{f})\ \mathsf{A} \ \equiv\ \begin{array}{ll} \mathsf{let} & \mathsf{A}' := \mathsf{domain}_{\perp}\ (\lambda\,\mathsf{a}.\,\mathsf{f}\ 0\ \mathsf{a})\ \mathsf{A} \\ \mathsf{in} & \mathsf{mapping}\ (\lambda\,\mathsf{a}.\,\mathsf{f}\ 0\ \mathsf{a})\ \mathsf{A}' \end{array} \tag{7.28}$$

It appears we need an $\eta$ rule to continue, which $\lambda_{\text{ZFC}}$ 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 $\mathsf{domain}_{\perp}\ (\lambda\,\mathsf{a}.\,\mathsf{f}\ 0\ \mathsf{a})\ \mathsf{A} \ \equiv\ \mathsf{domain}_{\perp}\ (\mathsf{f}\ 0)\ \mathsf{A}$. Further, it terminates if and only if $\mathsf{mapping}\ (\mathsf{f}\ 0)\ \mathsf{A}'$ terminates. Therefore, if $\mathsf{A} \neq \varnothing$, we can replace $\lambda\,\mathsf{a}.\,\mathsf{f}\ 0\ \mathsf{a}$ with $\mathsf{f}\ 0$. If $\mathsf{A} = \varnothing$, then $\mathsf{lift}_{\mathsf{map}}\ (\mathsf{lazy}_{\perp}\ \mathsf{f})\ \mathsf{A} = \varnothing$ (the empty mapping), so

$$\begin{aligned} \mathsf{lift}_{\mathsf{map}}\ (\mathsf{lazy}_{\perp}\ \mathsf{f})\ \mathsf{A} \ &\equiv\ \mathsf{if}\ (\mathsf{A} = \varnothing)\ \varnothing\ (\mathsf{mapping}\ (\mathsf{f}\ 0)\ (\mathsf{domain}_{\perp}\ (\mathsf{f}\ 0)\ \mathsf{A})) \\ &\equiv\ \mathsf{if}\ (\mathsf{A} = \varnothing)\ \varnothing\ (\mathsf{lift}_{\mathsf{map}}\ (\mathsf{f}\ 0)\ \mathsf{A}) \end{aligned} \tag{7.29}$$

Substituting $\mathsf{g}\ 0$ for $\mathsf{lift}_{\mathsf{map}}\ (\mathsf{f}\ 0)$ gives a $\mathsf{lazy}_{\mathsf{map}}$ (Figure 7.4) for which (7.12) holds.

### 7.4.5 Correctness

**Theorem 7.10** (mapping arrow correctness)**.** $\mathsf{lift}_{\mathsf{map}}$ *is a homomorphism.*

*Proof.* By construction. □

**Corollary 7.11** (semantic correctness)**.** *For all $e$, $[\![e]\!]_{\mathsf{map}} \equiv \mathsf{lift}_{\mathsf{map}}\ [\![e]\!]_{\bot}$.*

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

$$\begin{aligned} &\mathsf{nonmonotone} : \mathsf{Bool} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Bool} \\ &\mathsf{nonmonotone}\ \mathsf{A}\ := \ \mathsf{if}\ (\mathsf{A} = \varnothing)\ (\lambda\, \mathsf{a} \in \mathsf{Bool}.\, \mathsf{a})\ \varnothing \end{aligned} \tag{7.30}$$

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

By homomorphism of $\mathsf{lift}_{\mathsf{map}}$, 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 7.13.** $\mathsf{lift}_{\mathsf{map}}$ *is an arrow epimorphism.*

*Proof.* Follows from Theorem 7.10 and restriction of $\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}$ to instances for which the mapping arrow law (Definition 7.12) holds. □

**Corollary 7.14.** $\mathsf{arr}_{\mathsf{map}}$*,* $(\&\&\&_{\mathsf{map}})$*,* $(\ggg_{\mathsf{map}})$*,* $\mathsf{ifte}_{\mathsf{map}}$ *and* $\mathsf{lazy}_{\mathsf{map}}$ *define an arrow.*

## 7.5 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.

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

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

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

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

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

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

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

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

Figure 7.5: Lazy preimage mappings and operations.

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

$$X \underset{\mathsf{pre}}{\rightharpoonup} Y \ ::= \ \langle \mathsf{Set}\ Y, \mathsf{Set}\ Y \Rightarrow \mathsf{Set}\ X \rangle \tag{7.31}$$

with the intention to replace $X \underset{\mathsf{pre}}{\rightharpoonup} Y$ instances with an approximation further on. Like a mapping, an $X \underset{\mathsf{pre}}{\rightharpoonup} Y$ 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 $\mathsf{preimage}$:

$$\begin{aligned}
&\mathsf{pre} : (X \rightharpoonup Y) \Rightarrow (X \underset{\mathsf{pre}}{\rightharpoonup} Y) \\
&\mathsf{pre}\ g \ := \ \langle \mathsf{range}\ g, \lambda B.\, \mathsf{preimage}\ g\ B \rangle
\end{aligned} \tag{7.32}$$

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

$$\begin{aligned}&\mathsf{ap_{pre}} : (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y}) \Rightarrow \mathsf{Set\ Y} \Rightarrow \mathsf{Set\ X} \\ &\mathsf{ap_{pre}}\ \langle \mathsf{Y}', \mathsf{p}\rangle\ \mathsf{B}\ := \ \mathsf{p}\ (\mathsf{B} \cap \mathsf{Y}')\end{aligned} \tag{7.33}$$

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

**Theorem 7.16** ($\mathsf{ap_{pre}}$ computes preimages)**.** *Let* $\mathsf{g} : \mathsf{X} \rightharpoonup \mathsf{Y}$. *For all* $\mathsf{B} \subseteq \mathsf{Y}$, $\mathsf{ap_{pre}}\ (\mathsf{pre\ g})\ \mathsf{B} = \mathsf{preimage\ g\ B}$.

*Proof.* Expand definitions and apply Lemma 7.15 with $\mathsf{Y}' = \mathsf{range\ g}$. □

Figure 7.5 defines more operations on preimage mappings, including pairing, composition, and disjoint union operations corresponding to the mapping operations in Figure 7.3. To prove them correct, we need preimage mappings to be equivalent when they compute the same preimages.

**Definition 7.17** (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 Figure 7.5 are correct by proving that $\mathsf{pre}$ 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.

### 7.5.1 Composition

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

**Lemma 7.18** (preimage distributes over $(\circ_{\mathsf{map}})$). *Let* $\mathsf{g}_1 : \mathsf{X} \rightharpoonup \mathsf{Y}$ *and* $\mathsf{g}_2 : \mathsf{Y} \rightharpoonup \mathsf{Z}$*. 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 7.19** (pre distributes over $(\circ_{\mathsf{map}})$). *Let* $\mathsf{g}_1 : \mathsf{X} \rightharpoonup \mathsf{Y}$ *and* $\mathsf{g}_2 : \mathsf{Y} \rightharpoonup \mathsf{Z}$*. 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-hand side of the equivalence,

$$
\begin{aligned}
&\mathsf{ap}_{\mathsf{pre}}\ ((\mathsf{pre}\ \mathsf{g}_2) \circ_{\mathsf{pre}} (\mathsf{pre}\ \mathsf{g}_1))\ \mathsf{C} && && (7.34)\\
&\quad \equiv \begin{array}{l}\mathsf{let}\ \ \mathsf{p} := \lambda \mathsf{C}.\, \mathsf{ap}_{\mathsf{pre}}\ (\mathsf{pre}\ \mathsf{g}_1)\ (\mathsf{p}_2\ \mathsf{C})\\ \quad \mathsf{in}\ \ \mathsf{p}\ (\mathsf{C} \cap \mathsf{Z}')\end{array} && \text{Def of } \mathsf{ap}_{\mathsf{pre}}, (\circ_{\mathsf{pre}})\\
&\quad \equiv \mathsf{ap}_{\mathsf{pre}}\ (\mathsf{pre}\ \mathsf{g}_1)\ (\mathsf{p}_2\ (\mathsf{C} \cap \mathsf{Z}')) && \text{Def of } \mathsf{p}\\
&\quad \equiv \mathsf{ap}_{\mathsf{pre}}\ (\mathsf{pre}\ \mathsf{g}_1)\ (\mathsf{ap}_{\mathsf{pre}}\ (\mathsf{pre}\ \mathsf{g}_2)\ \mathsf{C}) && \text{Rewrite with } \mathsf{ap}_{\mathsf{pre}}\\
&\quad \equiv \mathsf{preimage}\ \mathsf{g}_1\ (\mathsf{preimage}\ \mathsf{g}_2\ \mathsf{C}) && \text{Theorem 7.16}\\
&\quad \equiv \mathsf{preimage}\ (\mathsf{g}_2 \circ_{\mathsf{map}} \mathsf{g}_1)\ \mathsf{C} && \text{Lemma 7.18}\\
&\quad \equiv \mathsf{ap}_{\mathsf{pre}}\ (\mathsf{pre}\ (\mathsf{g}_2 \circ_{\mathsf{map}} \mathsf{g}_1))\ \mathsf{C} && \text{Theorem 7.16} && \square
\end{aligned}
$$

### 7.5.2 Pairing

We have less luck with pairing than with composition, because preimage does not distribute over pairing. Fortunately, preimage distributes over unions, and over pairing and cartesian product together.

**Lemma 7.20** (preimage distributes over $\langle \cdot, \cdot \rangle_{\mathsf{map}}$ and $(\times)$). *Let* $\mathsf{g}_1 : \mathsf{X} \rightharpoonup \mathsf{Y}_1$ *and* $\mathsf{g}_2 : \mathsf{X} \rightharpoonup \mathsf{Y}_2$*. For all* $\mathsf{B}_1 \subseteq \mathsf{Y}_1$ *and* $\mathsf{B}_2 \subseteq \mathsf{Y}_2$*,* $\mathsf{preimage}\ \langle \mathsf{g}_1, \mathsf{g}_2 \rangle_{\mathsf{map}}\ (\mathsf{B}_1 \times \mathsf{B}_2) = (\mathsf{preimage}\ \mathsf{g}_1\ \mathsf{B}_1) \cap (\mathsf{preimage}\ \mathsf{g}_2\ \mathsf{B}_2)$*.*

**Lemma 7.21** (preimage distributes over union). *Let* $\mathsf{g} : \mathsf{X} \rightharpoonup \mathsf{Y}$ *and* $\mathsf{B} : \mathsf{J} \Rightarrow \mathsf{Set}\ \mathsf{Y}$ *be an indexed collection of subsets of* $\mathsf{Y}$*. Then*

$$\bigcup_{\mathsf{j} \in \mathsf{J}} \mathsf{preimage}\ \mathsf{g}\ (\mathsf{B}\ \mathsf{j}) \ = \ \mathsf{preimage}\ \mathsf{g}\ \bigcup_{\mathsf{j} \in \mathsf{J}} \mathsf{B}\ \mathsf{j} \qquad (7.35)$$

**Theorem 7.22** ($\mathsf{pre}$ distributes over $\langle\cdot,\cdot\rangle_{\mathsf{map}}$)**.** *Let* $\mathsf{g}_1 : \mathsf{X} \rightharpoonup \mathsf{Y}_1$ *and* $\mathsf{g}_2 : \mathsf{X} \rightharpoonup \mathsf{Y}_2$*. 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 $\mathsf{B} \subseteq \mathsf{Y}_1 \times \mathsf{Y}_2$. Starting from the right-hand side of the equivalence,

$$
\begin{aligned}
&\mathsf{ap_{pre}}\ \langle \mathsf{pre}\ \mathsf{g}_1, \mathsf{pre}\ \mathsf{g}_2\rangle_{\mathsf{pre}}\ \mathsf{B} && (7.36)\\
&\quad\equiv \mathsf{let}\ \ \mathsf{p} := \lambda\mathsf{B}.\ \bigcup_{\langle \mathsf{y}_1,\mathsf{y}_2\rangle\in\mathsf{B}} \mathsf{p}_1\ \{\mathsf{y}_1\} \cap \mathsf{p}_2\ \{\mathsf{y}_2\} && \text{Def of } \mathsf{ap_{pre}},\ \langle\cdot,\cdot\rangle_{\mathsf{pre}}\\
&\qquad\quad \mathsf{in}\ \ \mathsf{p}\ (\mathsf{B} \cap (\mathsf{Y}'_1 \times \mathsf{Y}'_2))\\
&\quad\equiv \bigcup_{\langle \mathsf{y}_1,\mathsf{y}_2\rangle\in\mathsf{B}\cap(\mathsf{Y}'_1\times\mathsf{Y}'_2)} \mathsf{p}_1\ \{\mathsf{y}_1\} \cap \mathsf{p}_2\ \{\mathsf{y}_2\} && \text{Def of } \mathsf{p}\\
&\quad\equiv \bigcup_{\langle \mathsf{y}_1,\mathsf{y}_2\rangle\in\mathsf{B}\cap(\mathsf{Y}'_1\times\mathsf{Y}'_2)} \mathsf{preimage}\ \mathsf{g}_1\ \{\mathsf{y}_1\} \cap \mathsf{preimage}\ \mathsf{g}_2\ \{\mathsf{y}_2\} && \text{Theorem 7.16}\\
&\quad\equiv \bigcup_{\langle \mathsf{y}_1,\mathsf{y}_2\rangle\in\mathsf{B}\cap(\mathsf{Y}'_1\times\mathsf{Y}'_2)} \mathsf{preimage}\ \langle \mathsf{g}_1, \mathsf{g}_2\rangle_{\mathsf{map}}\ (\{\mathsf{y}_1\} \times \{\mathsf{y}_2\}) && \text{Lemma 7.20}\\
&\quad\equiv \bigcup_{\langle \mathsf{y}_1,\mathsf{y}_2\rangle\in\mathsf{B}\cap(\mathsf{Y}'_1\times\mathsf{Y}'_2)} \mathsf{preimage}\ \langle \mathsf{g}_1, \mathsf{g}_2\rangle_{\mathsf{map}}\ \{\langle \mathsf{y}_1, \mathsf{y}_2\rangle\} && \text{Def of } (\times)\\
&\quad\equiv \mathsf{preimage}\ \langle \mathsf{g}_1, \mathsf{g}_2\rangle_{\mathsf{map}}\ (\mathsf{B} \cap (\mathsf{Y}'_1 \times \mathsf{Y}'_2)) && \text{Lemma 7.21}\\
&\quad\equiv \mathsf{preimage}\ \langle \mathsf{g}_1, \mathsf{g}_2\rangle_{\mathsf{map}}\ \mathsf{B} && \text{Lemma 7.15}\\
&\quad\equiv \mathsf{ap_{pre}}\ (\mathsf{pre}\ \langle \mathsf{g}_1, \mathsf{g}_2\rangle_{\mathsf{map}})\ \mathsf{B} && \text{Theorem 7.16}
\end{aligned}
$$

We have an unmet proof obligation from using Lemma 7.15: that $\mathsf{range}\ \langle \mathsf{g}_1, \mathsf{g}_2\rangle_{\mathsf{map}} \subseteq \mathsf{Y}'_1 \times \mathsf{Y}'_2$.

Let $\mathsf{b} \in \mathsf{range}\ \langle \mathsf{g}_1, \mathsf{g}_2\rangle_{\mathsf{map}}$. By definition of $\langle\cdot,\cdot\rangle_{\mathsf{map}}$, there exists $\mathsf{a} \in \mathsf{domain}\ \mathsf{g}_1 \cap \mathsf{domain}\ \mathsf{g}_2$ such that $\mathsf{b} = \langle \mathsf{g}_1\ \mathsf{a}, \mathsf{g}_2\ \mathsf{a}\rangle$. Thus, $\mathsf{b} \in \mathsf{Y}'_1 \times \mathsf{Y}'_2$ if and only if $\mathsf{g}_1\ \mathsf{a} \in \mathsf{Y}'_1$ and $\mathsf{g}_2\ \mathsf{a} \in \mathsf{Y}'_2$.

By definition of $\mathsf{pre}$, $\mathsf{Y}'_1 = \mathsf{range}\ \mathsf{g}_1$ and $\mathsf{Y}'_2 = \mathsf{range}\ \mathsf{g}_2$. Because $\mathsf{a} \in \mathsf{domain}\ \mathsf{g}_1$, $\mathsf{g}_1\ \mathsf{a} \in \mathsf{range}\ \mathsf{g}_1 = \mathsf{Y}'_1$. Because $\mathsf{a} \in \mathsf{domain}\ \mathsf{g}_2$, $\mathsf{g}_2\ \mathsf{a} \in \mathsf{range}\ \mathsf{g}_2 = \mathsf{Y}'_2$. □

### 7.5.3 Disjoint Union

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

**Lemma 7.23** ($\mathsf{preimage}$ distributes over $(\uplus_{\mathsf{map}})$)**.** *Let* $\mathsf{g_1} : \mathsf{X} \rightharpoonup \mathsf{Y}$ *and* $\mathsf{g_2} : \mathsf{X} \rightharpoonup \mathsf{Y}$ *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 7.24** ($\mathsf{pre}$ distributes over $(\uplus_{\mathsf{map}})$)**.** *Let* $\mathsf{g_1} : \mathsf{X} \rightharpoonup \mathsf{Y}$ *and* $\mathsf{g_2} : \mathsf{X} \rightharpoonup \mathsf{Y}$ *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-hand side of the equivalence,

$$\begin{aligned}
&\mathsf{ap_{pre}}\ ((\mathsf{pre}\ \mathsf{g_1}) \uplus_{\mathsf{pre}} (\mathsf{pre}\ \mathsf{g_2}))\ \mathsf{B} && (7.37)\\
&\quad\equiv \mathsf{let}\ \ \mathsf{p} := \lambda\mathsf{B}.\, \mathsf{ap_{pre}}\ (\mathsf{pre}\ \mathsf{g_1})\ \mathsf{B} \uplus \mathsf{ap_{pre}}\ (\mathsf{pre}\ \mathsf{g_2})\ \mathsf{B} && \text{Def of } \mathsf{ap_{pre}},\ (\uplus_{\mathsf{pre}})\\
&\quad\qquad \mathsf{in}\ \ \mathsf{p}\ (\mathsf{B} \cap (\mathsf{Y}'_1 \cup \mathsf{Y}'_2)) &&\\
&\quad\equiv \mathsf{ap_{pre}}\ (\mathsf{pre}\ \mathsf{g_1})\ (\mathsf{B} \cap (\mathsf{Y}'_1 \cup \mathsf{Y}'_2)) \uplus \mathsf{ap_{pre}}\ (\mathsf{pre}\ \mathsf{g_2})\ (\mathsf{B} \cap (\mathsf{Y}'_1 \cup \mathsf{Y}'_2)) && \text{Def of } \mathsf{p}\\
&\quad\equiv \mathsf{preimage}\ \mathsf{g_1}\ (\mathsf{B} \cap (\mathsf{Y}'_1 \cup \mathsf{Y}'_2)) \uplus \mathsf{preimage}\ \mathsf{g_2}\ (\mathsf{B} \cap (\mathsf{Y}'_1 \cup \mathsf{Y}'_2)) && \text{Theorem 7.16}\\
&\quad\equiv \mathsf{preimage}\ (\mathsf{g_1} \uplus_{\mathsf{map}} \mathsf{g_2})\ (\mathsf{B} \cap (\mathsf{Y}'_1 \cup \mathsf{Y}'_2)) && \text{Lemma 7.23}\\
&\quad\equiv \mathsf{preimage}\ (\mathsf{g_1} \uplus_{\mathsf{map}} \mathsf{g_2})\ \mathsf{B} && \text{Lemma 7.15}\\
&\quad\equiv \mathsf{ap_{pre}}\ (\mathsf{pre}\ (\mathsf{g_1} \uplus_{\mathsf{map}} \mathsf{g_2}))\ \mathsf{B} && \text{Theorem 7.16}
\end{aligned}$$

We have an unmet proof obligation from using Lemma 7.15: that $\mathsf{range}\ (\mathsf{g_1} \uplus_{\mathsf{map}} \mathsf{g_2}) \subseteq \mathsf{Y}'_1 \cup \mathsf{Y}'_2$.

Let $\mathsf{b} \in \mathsf{range}\ (\mathsf{g_1} \uplus_{\mathsf{map}} \mathsf{g_2})$. By definition of $(\uplus_{\mathsf{map}})$, there exists $\mathsf{a} \in \mathsf{domain}\ \mathsf{g_1} \uplus \mathsf{domain}\ \mathsf{g_2}$ such that if $\mathsf{a} \in \mathsf{domain}\ \mathsf{g_1}$ then $\mathsf{b} = \mathsf{g_1}\ \mathsf{a}$ so $\mathsf{b} \in \mathsf{range}\ \mathsf{g_1} = \mathsf{Y}'_1$, and if $\mathsf{a} \in \mathsf{domain}\ \mathsf{g_2}$ then $\mathsf{b} = \mathsf{g_2}\ \mathsf{a}$ so $\mathsf{b} \in \mathsf{range}\ \mathsf{g_2} = \mathsf{Y}'_2$. Thus $\mathsf{b} \in \mathsf{Y}'_1 \cup \mathsf{Y}'_2$. □

## 7.6 Deriving the Preimage Arrow

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

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

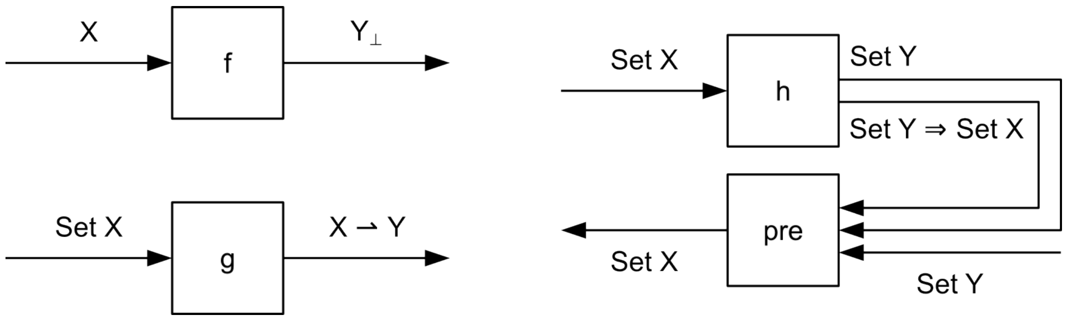


Figure 7.6: Comparison of arrows used as target categories. Computations $\mathsf{f} : \mathsf{X} \rightsquigarrow_\perp \mathsf{Y}$ may return an error value $\perp$. Computations $\mathsf{g} : \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}$ produce partial mappings on a given $\mathsf{A} \subseteq \mathsf{X}$, leaving out inputs for which $\mathsf{f}$ would return an error. Computations $\mathsf{h} : \mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}$ produce lazy preimage mappings; i.e. $\mathsf{h}\ \mathsf{A}$ computes preimages under $\mathsf{g}\ \mathsf{A}$.

***preimage arrow*** type constructor is

$$\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y} \ ::= \ \mathsf{Set\ X} \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y}) \tag{7.38}$$

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. Figure 7.6 illustrates this as a circuit diagram.

To use Theorem 7.6, we need to define correctness using a lift from the mapping arrow to the preimage arrow. A simple candidate with the right type is

$$\begin{aligned} \mathsf{lift_{pre}} &: (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}) \\ \mathsf{lift_{pre}}\ \mathsf{g}\ \mathsf{A} &:= \ \mathsf{pre}\ (\mathsf{g}\ \mathsf{A}) \end{aligned} \tag{7.39}$$

By definition of $\mathsf{lift_{pre}}$ and Theorem 7.16, for all $\mathsf{g} : \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}$, and $\mathsf{A} \subseteq \mathsf{X}$ and $\mathsf{B} \subseteq \mathsf{Y}$,

$$\begin{aligned} \mathsf{ap_{pre}}\ (\mathsf{lift_{pre}}\ \mathsf{g}\ \mathsf{A})\ \mathsf{B} &\equiv \ \mathsf{ap_{pre}}\ (\mathsf{pre}\ (\mathsf{g}\ \mathsf{A}))\ \mathsf{B} \\ &\equiv \ \mathsf{preimage}\ (\mathsf{g}\ \mathsf{A})\ \mathsf{B} \end{aligned} \tag{7.40}$$

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."

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

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

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

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

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

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

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

Figure 7.7: Preimage arrow definitions.

**Definition 7.25** (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* $\mathsf{h_1\ A} \equiv \mathsf{h_2\ A}$ *for all* $\mathsf{A} \subseteq \mathsf{X}$*.*

As with $\mathsf{arr_{map}}$, defining $\mathsf{arr_{pre}}$ as a composition meets (7.8). 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. Figure 7.7 contains the resulting definitions.

### 7.6.1 Composition

Starting with the left-hand side of (7.9),

$$\begin{aligned}
&\mathsf{ap_{pre}}\ (\mathsf{lift_{pre}}\ (\mathsf{g_1} \ggg_{\mathsf{map}} \mathsf{g_2})\ \mathsf{A})\ \mathsf{C} && (7.41)\\
&\quad\equiv\ \mathsf{let}\ \ \mathsf{g'_1} := \mathsf{g_1\ A} && \text{Def of } \mathsf{lift_{pre}},\ (\ggg_{\mathsf{map}})\\
&\qquad\qquad\ \ \mathsf{g'_2} := \mathsf{g_2}\ (\mathsf{range}\ \mathsf{g'_1})\\
&\qquad\quad\ \mathsf{in}\ \ \mathsf{ap_{pre}}\ (\mathsf{pre}\ (\mathsf{g'_2} \circ_{\mathsf{map}} \mathsf{g'_1}))\ \mathsf{C}\\
&\quad\equiv\ \mathsf{let}\ \ \mathsf{g'_1} := \mathsf{g_1\ A} && \text{Theorem 7.19}\\
&\qquad\qquad\ \ \mathsf{g'_2} := \mathsf{g_2}\ (\mathsf{range}\ \mathsf{g'_1})\\
&\qquad\quad\ \mathsf{in}\ \ \mathsf{ap_{pre}}\ ((\mathsf{pre}\ \mathsf{g'_1}) \circ_{\mathsf{pre}} (\mathsf{pre}\ \mathsf{g'_2}))\ \mathsf{C}\\
&\quad\equiv\ \mathsf{let}\ \ \mathsf{h_1} := \mathsf{lift_{pre}}\ \mathsf{g_1\ A} && \text{Rewrite with } \mathsf{lift_{pre}}\\
&\qquad\qquad\ \ \mathsf{h_2} := \mathsf{lift_{pre}}\ \mathsf{g_2}\ (\mathsf{range_{pre}}\ \mathsf{h_1})\\
&\qquad\quad\ \mathsf{in}\ \ \mathsf{ap_{pre}}\ (\mathsf{h_2} \circ_{\mathsf{pre}} \mathsf{h_1})\ \mathsf{C}
\end{aligned}$$

Substituting $\mathsf{h}_1$ for $\mathsf{lift_{pre}}\ \mathsf{g}_1$ and $\mathsf{h}_2$ for $\mathsf{lift_{pre}}\ \mathsf{g}_2$, and removing the application of $\mathsf{ap_{pre}}$ from both sides of the equivalence gives a definition of $(\ggg_{\mathsf{pre}})$ (Figure 7.7) for which (7.9) holds.

### 7.6.2 Pairing

Starting with the left-hand side of (7.10),

$$\begin{aligned}
&\mathsf{ap_{pre}}\ (\mathsf{lift_{pre}}\ (\mathsf{g}_1\ \&\&\&_{\mathsf{map}}\ \mathsf{g}_2)\ \mathsf{A})\ \mathsf{B} && (7.42)\\
&\quad\equiv\ \mathsf{ap_{pre}}\ (\mathsf{pre}\ \langle \mathsf{g}_1\ \mathsf{A}, \mathsf{g}_2\ \mathsf{A}\rangle_{\mathsf{map}})\ \mathsf{B} && \text{Def of } \mathsf{lift_{pre}},\ (\&\&\&_{\mathsf{map}})\\
&\quad\equiv\ \mathsf{ap_{pre}}\ \langle \mathsf{pre}\ (\mathsf{g}_1\ \mathsf{A}), \mathsf{pre}\ (\mathsf{g}_2\ \mathsf{A})\rangle_{\mathsf{pre}}\ \mathsf{B} && \text{Theorem 7.22}\\
&\quad\equiv\ \mathsf{ap_{pre}}\ \langle \mathsf{lift_{pre}}\ \mathsf{g}_1\ \mathsf{A}, \mathsf{lift_{pre}}\ \mathsf{g}_2\ \mathsf{A}\rangle_{\mathsf{pre}}\ \mathsf{B} && \text{Rewrite with } \mathsf{lift_{pre}}
\end{aligned}$$

Substituting $\mathsf{h}_1$ for $\mathsf{lift_{pre}}\ \mathsf{g}_1$ and $\mathsf{h}_2$ for $\mathsf{lift_{pre}}\ \mathsf{g}_2$, and removing the application of $\mathsf{ap_{pre}}$ from both sides of the equivalence gives a definition of $(\&\&\&_{\mathsf{pre}})$ (Figure 7.7) for which (7.10) holds.

### 7.6.3 Conditional

Starting with the left-hand side of (7.11),

$$\begin{aligned}
&\mathsf{ap_{pre}}\ (\mathsf{lift_{pre}}\ (\mathsf{ifte_{map}}\ \mathsf{g}_1\ \mathsf{g}_2\ \mathsf{g}_3)\ \mathsf{A})\ \mathsf{B} && (7.43)\\
&\quad\equiv\ \begin{aligned}[t]\mathsf{let}\ \ &\mathsf{g}_1' := \mathsf{g}_1\ \mathsf{A}\\ &\mathsf{g}_2' := \mathsf{g}_2\ (\mathsf{preimage}\ \mathsf{g}_1'\ \{\mathsf{true}\})\\ &\mathsf{g}_3' := \mathsf{g}_3\ (\mathsf{preimage}\ \mathsf{g}_1'\ \{\mathsf{false}\})\\ \mathsf{in}\ \ &\mathsf{ap_{pre}}\ (\mathsf{pre}\ (\mathsf{g}_2' \uplus_{\mathsf{map}} \mathsf{g}_3'))\ \mathsf{B}\end{aligned} && \text{Def of } \mathsf{lift_{pre}},\ \mathsf{ifte_{map}}\\
&\quad\equiv\ \begin{aligned}[t]\mathsf{let}\ \ &\mathsf{g}_1' := \mathsf{g}_1\ \mathsf{A}\\ &\mathsf{g}_2' := \mathsf{g}_2\ (\mathsf{preimage}\ \mathsf{g}_1'\ \{\mathsf{true}\})\\ &\mathsf{g}_3' := \mathsf{g}_3\ (\mathsf{preimage}\ \mathsf{g}_1'\ \{\mathsf{false}\})\\ \mathsf{in}\ \ &\mathsf{ap_{pre}}\ ((\mathsf{pre}\ \mathsf{g}_2') \uplus_{\mathsf{pre}} (\mathsf{pre}\ \mathsf{g}_3'))\ \mathsf{B}\end{aligned} && \text{Theorem 7.24}\\
&\quad\equiv\ \begin{aligned}[t]\mathsf{let}\ \ &\mathsf{g}_1' := \mathsf{g}_1\ \mathsf{A}\\ &\mathsf{g}_2' := \mathsf{g}_2\ (\mathsf{ap_{pre}}\ (\mathsf{pre}\ \mathsf{g}_1')\ \{\mathsf{true}\})\\ &\mathsf{g}_3' := \mathsf{g}_3\ (\mathsf{ap_{pre}}\ (\mathsf{pre}\ \mathsf{g}_1')\ \{\mathsf{false}\})\\ \mathsf{in}\ \ &\mathsf{ap_{pre}}\ ((\mathsf{pre}\ \mathsf{g}_2') \uplus_{\mathsf{pre}} (\mathsf{pre}\ \mathsf{g}_3'))\ \mathsf{B}\end{aligned} && \text{Theorem 7.16}
\end{aligned}$$

$$\begin{aligned}\equiv\ & \mathsf{let}\ \ \mathsf{h}_1 := \mathsf{lift}_{\mathsf{pre}}\ \mathsf{g}_1\ \mathsf{A} && \text{Rewrite with } \mathsf{lift}_{\mathsf{pre}}\\ & \qquad \mathsf{h}_2 := \mathsf{lift}_{\mathsf{pre}}\ \mathsf{g}_2\ (\mathsf{ap}_{\mathsf{pre}}\ \mathsf{h}_1\ \{\mathsf{true}\})\\ & \qquad \mathsf{h}_3 := \mathsf{lift}_{\mathsf{pre}}\ \mathsf{g}_3\ (\mathsf{ap}_{\mathsf{pre}}\ \mathsf{h}_1\ \{\mathsf{false}\})\\ & \quad \mathsf{in}\ \ \mathsf{ap}_{\mathsf{pre}}\ (\mathsf{h}_2 \uplus_{\mathsf{pre}} \mathsf{h}_3)\ \mathsf{B}\end{aligned}$$

Substituting $\mathsf{h}_1$, $\mathsf{h}_2$ and $\mathsf{h}_3$ for $\mathsf{lift}_{\mathsf{pre}}\ \mathsf{g}_1$, $\mathsf{lift}_{\mathsf{pre}}\ \mathsf{g}_2$ and $\mathsf{lift}_{\mathsf{pre}}\ \mathsf{g}_3$, and removing the application of $\mathsf{ap}_{\mathsf{pre}}$ from both sides of the equivalence gives a definition of $\mathsf{ifte}_{\mathsf{pre}}$ (Figure 7.7) for which (7.11) holds.

### 7.6.4 Laziness

Starting with the left-hand side of (7.12),

$$\begin{aligned}& \mathsf{ap}_{\mathsf{pre}}\ (\mathsf{lift}_{\mathsf{pre}}\ (\mathsf{lazy}_{\mathsf{map}}\ \mathsf{g})\ \mathsf{A})\ \mathsf{B} && (7.44)\\ & \quad \equiv\ \mathsf{let}\ \ \mathsf{g}' := \mathsf{if}\ (\mathsf{A} = \varnothing)\ \varnothing\ (\mathsf{g}\ 0\ \mathsf{A}) && \text{Def of } \mathsf{lift}_{\mathsf{pre}},\ \mathsf{lazy}_{\mathsf{map}}\\ & \qquad\quad \mathsf{in}\ \ \mathsf{ap}_{\mathsf{pre}}\ (\mathsf{pre}\ \mathsf{g}')\ \mathsf{B}\\ & \quad \equiv\ \mathsf{let}\ \ \mathsf{h} := \mathsf{if}\ (\mathsf{A} = \varnothing)\ (\mathsf{pre}\ \varnothing)\ (\mathsf{pre}\ (\mathsf{g}\ 0\ \mathsf{A})) && \text{Dist } \mathsf{pre} \text{ over } \mathsf{if}\\ & \qquad\quad \mathsf{in}\ \ \mathsf{ap}_{\mathsf{pre}}\ \mathsf{h}\ \mathsf{B}\\ & \quad \equiv\ \mathsf{let}\ \ \mathsf{h} := \mathsf{if}\ (\mathsf{A} = \varnothing)\ (\mathsf{pre}\ \varnothing)\ (\mathsf{lift}_{\mathsf{pre}}\ (\mathsf{g}\ 0)\ \mathsf{A}) && \text{Rewrite with } \mathsf{lift}_{\mathsf{pre}}\\ & \qquad\quad \mathsf{in}\ \ \mathsf{ap}_{\mathsf{pre}}\ \mathsf{h}\ \mathsf{B}\end{aligned}$$

Substituting $\mathsf{h}\ 0$ for $\mathsf{lift}_{\mathsf{pre}}\ (\mathsf{g}\ 0)$ and removing the application of $\mathsf{ap}_{\mathsf{pre}}$ from both sides of the equivalence gives a definition for $\mathsf{lazy}_{\mathsf{pre}}$ (Figure 7.7) for which (7.12) holds.

### 7.6.5 Correctness

**Theorem 7.26** (preimage arrow correctness). $\mathsf{lift}_{\mathsf{pre}}$ *is a homomorphism.*

*Proof.* By construction. □

**Corollary 7.27** (semantic correctness). *For all $e$,* $[\![e]\!]_{\mathsf{pre}} \equiv \mathsf{lift}_{\mathsf{pre}}\ [\![e]\!]_{\mathsf{map}}$.

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 7.28** (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* $\mathsf{h} \equiv \mathsf{lift}_{\mathsf{pre}}\ \mathsf{g}$*, then* $\mathsf{h}$ *obeys the* ***preimage arrow law****.*

By homomorphism of $\mathsf{lift}_{\mathsf{pre}}$, 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 7.29.** $\mathsf{lift}_{\mathsf{pre}}$ *is an arrow epimorphism.*

*Proof.* Follows from Theorem 7.26 and restriction of $\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}$ to instances for which the preimage arrow law (Definition 7.28) holds. □

**Corollary 7.30.** $\mathsf{arr}_{\mathsf{pre}}$*,* $(\&\&\&_{\mathsf{pre}})$*,* $(\ggg_{\mathsf{pre}})$*,* $\mathsf{ifte}_{\mathsf{pre}}$ *and* $\mathsf{lazy}_{\mathsf{pre}}$ *define an arrow.*

## 7.7 Preimages Under Partial, Probabilistic Functions

We have defined everything on the top of our roadmap:

$$\begin{tikzcd}
\mathsf{X} \rightsquigarrow_{\bot} \mathsf{Y} \arrow[r, "\mathsf{lift}_{\mathsf{map}}"] \arrow[d, "\eta_{\bot^*}"'] & \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y} \arrow[r, "\mathsf{lift}_{\mathsf{pre}}"] \arrow[d, "\eta_{\mathsf{map}^*}"] & \mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y} \arrow[d, "\eta_{\mathsf{pre}^*}"] \\
\mathsf{X} \rightsquigarrow_{\bot^*} \mathsf{Y} \arrow[r, "\mathsf{lift}_{\mathsf{map}^*}"'] & \mathsf{X} \underset{\mathsf{map}^*}{\rightsquigarrow} \mathsf{Y} \arrow[r, "\mathsf{lift}_{\mathsf{pre}^*}"'] & \mathsf{X} \underset{\mathsf{pre}^*}{\rightsquigarrow} \mathsf{Y}
\end{tikzcd} \tag{7.45}$$

and proved that $\mathsf{lift}_{\mathsf{map}}$ and $\mathsf{lift}_{\mathsf{pre}}$ are homomorphisms. At this point, we can interpret an expression $e$ in three ways using the same semantic function for first-order programs:

1. As $[\![e]\!]_{\bot} : \mathsf{X} \rightsquigarrow_{\bot} \mathsf{Y}$, an intensional function that may raise errors.
2. As $[\![e]\!]_{\mathsf{map}} : \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}$, which produces mappings, or extensional functions, on a restricted domain (correct by homomorphism of $\mathsf{lift}_{\mathsf{map}}$).
3. As $[\![e]\!]_{\mathsf{pre}} : \mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}$, which computes preimages under mappings produced by $[\![e]\!]_{\mathsf{map}}$ (correct by homomorphism of $\mathsf{lift}_{\mathsf{pre}}$).

These interpretations have two shortcomings:

1. They do not pass an implicit random source through $e$'s subexpressions.
2. Using them requires knowing the set of inputs on which $e$ terminates. If $[\![e]\!]_{\bot}$ does not terminate on just one input in $\mathsf{A} \subseteq \mathsf{X}$, neither $[\![e]\!]_{\mathsf{map}}\ \mathsf{A}$ nor $[\![e]\!]_{\mathsf{pre}}\ \mathsf{A}$ terminates.

In this section, we define the arrows on the bottom of the roadmap (7.45) by transforming the arrows on the top into arrows that pass an implicit random source and always terminate. Their correctness again comes down to proving that the combinators between them are homomorphisms, though guaranteed termination needs special treatment.

### 7.7.1 Motivation

Probabilistic functions that may not terminate, but do so with probability 1, are common. For example, suppose $\mathsf{random}$ 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 $\mathsf{random} < \mathsf{p}$:

$$\mathsf{geometric\ p} \ := \ \mathsf{if}\ (\mathsf{random} < \mathsf{p})\ 0\ (1 + \mathsf{geometric\ p}) \tag{7.46}$$

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

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

- Ensure $\mathsf{ap}_{\mathsf{pre}}\ (\mathsf{h}\ \mathit{\Omega})\ \mathsf{N}$ terminates.
- Ensure each $\omega \in \mathit{\Omega}$ contains enough random numbers.
- Determine how $\mathsf{random}$ indexes numbers in $\omega$.

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

### 7.7.2 Threading and Indexing

To ensure random sources contain enough numbers, they should be infinite.

Typically, to thread a random source $\omega \in \mathit{\Omega}$ through computations, $\omega$ is made an infinite stream. Each computation receives and returns an $\omega$. The interpretation of $\mathsf{random}$

as a computation takes $\omega$'s head and returns its tail. Combinators pass $\omega$ unchanged to one subcomputation, and pass the resulting $\omega'$ unchanged to the next. This is typically done with a monad, and it imposes a total order on evaluation.

A little-used alternative that imposes only a partial order makes $\omega$ an infinite binary tree. Each computation receives an $\omega$ but does not return one. The interpretation of $\mathsf{random}$ as a computation simply returns $\omega$'s root value. Combinators ignore the root, split $\omega$ into a left subtree $\omega_{\mathsf{left}}$ and a right subtree $\omega_{\mathsf{right}}$, and pass each to their subcomputations.

Arrows can thread a stream or a tree in the same manner, but the resulting combinators have large definitions, and are conceptually difficult and hard to manipulate. Fortunately, it is relatively easy to assign each arrow computation a unique index into a tree-shaped random source and pass the random source unchanged. To do this, we need an indexing scheme.

**Definition 7.31** (binary indexing scheme)**.** *Let* $\mathsf{J}$ *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* $\mathsf{next}$ *of* $\mathsf{left}$ *and* $\mathsf{right}$*, then* $\mathsf{J}$*,* $\mathsf{j}_0$*,* $\mathsf{left}$ *and* $\mathsf{right}$ *define a* ***binary indexing scheme****.*

For example, let $\mathsf{J}$ be the set of lists of $\{0,1\}$, $\mathsf{j}_0 := \langle\rangle$, and $\mathsf{left}\ \mathsf{j} := \langle 0, \mathsf{j}\rangle$ and $\mathsf{right}\ \mathsf{j} := \langle 1, \mathsf{j}\rangle$. Alternatively, let $\mathsf{J}$ be the set of dyadic rationals in $(0,1)$ (i.e. those with power-of-two denominators), $\mathsf{j}_0 := \frac{1}{2}$ and

$$\begin{aligned} \mathsf{left}\ (\mathsf{p}/\mathsf{q}) &:= (\mathsf{p} - \tfrac{1}{2})/\mathsf{q} \\ \mathsf{right}\ (\mathsf{p}/\mathsf{q}) &:= (\mathsf{p} + \tfrac{1}{2})/\mathsf{q} \end{aligned} \tag{7.47}$$

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

In any case, $\mathsf{J}$ is countable, and can be thought of as a set of indexes into an infinite binary tree. Values of type $\mathsf{J} \to \mathsf{A}$ encode such trees of values in $\mathsf{A}$ as total mappings (i.e. infinite vectors).

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

$$\begin{aligned}
&\mathsf{arr_{a^*}} : (\mathsf{x} \Rightarrow \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a^*}} \mathsf{y}) \\
&\mathsf{arr_{a^*}} \;:=\; \eta_{\mathsf{a^*}} \circ \mathsf{arr_a}
\end{aligned}$$

$$\begin{aligned}
&(\ggg_{\mathsf{a^*}}) : (\mathsf{x} \rightsquigarrow_{\mathsf{a^*}} \mathsf{y}) \Rightarrow (\mathsf{y} \rightsquigarrow_{\mathsf{a^*}} \mathsf{z}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a^*}} \mathsf{z}) \\
&(\mathsf{k_1} \ggg_{\mathsf{a^*}} \mathsf{k_2})\ \mathsf{j} \;:= \\
&\quad (\mathsf{arr_a\ fst}\ \&\&\&_{\mathsf{a}}\ \mathsf{k_1}\ (\mathsf{left\ j})) \ggg_{\mathsf{a}} \mathsf{k_2}\ (\mathsf{right\ j})
\end{aligned}$$

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

$$\begin{aligned}
&\mathsf{ifte_{a^*}} : (\mathsf{x} \rightsquigarrow_{\mathsf{a^*}} \mathsf{Bool}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a^*}} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a^*}} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a^*}} \mathsf{y}) \\
&\mathsf{ifte_{a^*}}\ \mathsf{k_1\ k_2\ k_3\ j} \;:=\; \mathsf{ifte_a}\ (\mathsf{k_1}\ (\mathsf{left\ j})) \\
&\qquad\qquad\qquad\qquad\quad (\mathsf{k_2}\ (\mathsf{left}\ (\mathsf{right\ j}))) \\
&\qquad\qquad\qquad\qquad\quad (\mathsf{k_3}\ (\mathsf{right}\ (\mathsf{right\ j})))
\end{aligned}$$

$$\begin{aligned}
&\mathsf{lazy_{a^*}} : (1 \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a^*}} \mathsf{y})) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a^*}} \mathsf{y}) \\
&\mathsf{lazy_{a^*}}\ \mathsf{k\ j} \;:=\; \mathsf{lazy_a}\ \lambda 0.\, \mathsf{k}\ 0\ \mathsf{j}
\end{aligned}$$

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

Figure 7.8: AStore (associative store) arrow transformer definitions.

### 7.7.3 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 $\mathsf{s}$ and an arrow type $\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}$:

$$\mathsf{AStore\ s}\ (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \;::=\; \mathsf{J} \Rightarrow (\langle \mathsf{s},\mathsf{x}\rangle \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \tag{7.48}$$

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

$$\begin{aligned}
&\eta_{\mathsf{a^*}} : (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \Rightarrow \mathsf{AStore\ s}\ (\mathsf{x} \rightsquigarrow_{\mathsf{a}} \mathsf{y}) \\
&\eta_{\mathsf{a^*}}\ \mathsf{f\ j} \;:=\; \mathsf{arr_a\ snd} \ggg_{\mathsf{a}} \mathsf{f}
\end{aligned} \tag{7.49}$$

Figure 7.8 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 subcomputation labeled with its tree index.

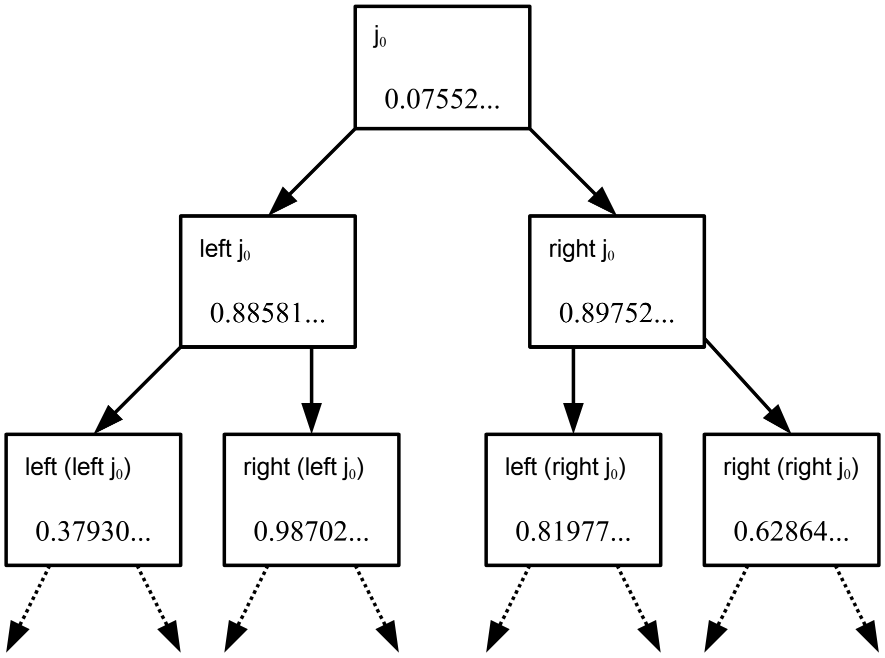


Figure 7.9: An $\omega \in \Omega$ is an infinite binary tree of random values encoded as a total mapping from tree indexes in $\mathsf{J}$ to real numbers in $[0, 1]$.

### 7.7.4 Partial, Probabilistic Programs

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

**Definition 7.32** (random source)**.** *Let $\Omega := \mathsf{J} \to [0,1]$. A **random source** is any infinite binary tree $\omega \in \Omega$.*

Figure 7.9 illustrates a single $\omega \in \Omega$.

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 7.33** (branch trace)**.** *A **branch trace** is any $\mathsf{t} : \mathsf{J} \to \mathsf{Bool}_\bot$ 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 $\mathsf{T} \subset \mathsf{J} \to \mathsf{Bool}_\bot$ be the largest set of branch traces.*

Now $\mathsf{X} \rightsquigarrow_{\mathsf{a}^*} \mathsf{Y} \ ::= \ \mathsf{AStore}\ (\mathit{\Omega} \times \mathsf{T})\ (\mathsf{X} \rightsquigarrow_{\mathsf{a}} \mathsf{Y})$ is an AStore arrow type whose computations thread 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 $\mathsf{a}^*$ for which $\mathsf{random}_{\mathsf{a}^*}$ is defined:

$$\begin{aligned}
&\mathsf{random}_{\mathsf{a}^*} : \mathsf{X} \rightsquigarrow_{\mathsf{a}^*} [0,1] \\
&\mathsf{random}_{\mathsf{a}^*}\ \mathsf{j} \ := \ \mathsf{arr}_{\mathsf{a}}\ (\mathsf{fst} \ggg \mathsf{fst} \ggg \pi\ \mathsf{j}) \\
&[\![\mathsf{random}]\!]_{\mathsf{a}^*} \ :\equiv \ \mathsf{random}_{\mathsf{a}^*}
\end{aligned} \tag{7.50}$$

Here, $\pi\ \mathsf{j}$ projects its argument onto the argument's $\mathsf{j}$th coordinate:

$$\begin{aligned}
&\pi : \mathsf{J} \Rightarrow (\mathsf{J} \to \mathsf{X}) \Rightarrow \mathsf{X} \\
&\pi\ \mathsf{j}\ \mathsf{f} \ := \ \mathsf{f}\ \mathsf{j}
\end{aligned} \tag{7.51}$$

So $\pi\ \mathsf{j}$ is analogous to fst and snd for pairs, but for vectors at index j.

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:

$$\begin{aligned}
&\mathsf{branch}_{\mathsf{a}^*} : \mathsf{X} \rightsquigarrow_{\mathsf{a}^*} \mathsf{Bool} \\
&\mathsf{branch}_{\mathsf{a}^*}\ \mathsf{j} \ := \ \mathsf{arr}_{\mathsf{a}}\ (\mathsf{fst} \ggg \mathsf{snd} \ggg \pi\ \mathsf{j}) \\
&\\
&\mathsf{ifte}^{\Downarrow}_{\mathsf{a}^*} : (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{Bool}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y}) \Rightarrow (\mathsf{x} \rightsquigarrow_{\mathsf{a}^*} \mathsf{y}) \\
&\mathsf{ifte}^{\Downarrow}_{\mathsf{a}^*}\ \mathsf{k}_1\ \mathsf{k}_2\ \mathsf{k}_3\ \mathsf{j} \ := \ \mathsf{ifte}_{\mathsf{a}}\ ((\mathsf{k}_1\ (\mathsf{left}\ \mathsf{j})\ \&\&\&_{\mathsf{a}}\ \mathsf{branch}_{\mathsf{a}^*}\ \mathsf{j}) \ggg_{\mathsf{a}} \mathsf{arr}_{\mathsf{a}}\ \mathsf{agrees}) \\
&\qquad\qquad\qquad\qquad\qquad (\mathsf{k}_2\ (\mathsf{left}\ (\mathsf{right}\ \mathsf{j}))) \\
&\qquad\qquad\qquad\qquad\qquad (\mathsf{k}_3\ (\mathsf{right}\ (\mathsf{right}\ \mathsf{j})))
\end{aligned} \tag{7.52}$$

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 $[\![\cdot]\!]_{\mathsf{a}^*}$:

$$[\![\mathsf{if}\ e_c\ e_t\ e_f]\!]^{\Downarrow}_{\mathsf{a}^*} \ :\equiv \ \mathsf{ifte}^{\Downarrow}_{\mathsf{a}^*}\ [\![e_c]\!]^{\Downarrow}_{\mathsf{a}^*}\ [\![\mathsf{lazy}\ e_t]\!]^{\Downarrow}_{\mathsf{a}^*}\ [\![\mathsf{lazy}\ e_f]\!]^{\Downarrow}_{\mathsf{a}^*} \tag{7.53}$$

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

**Definition 7.34** (terminating, probabilistic arrows)**.** *Define*

$$\begin{aligned}\mathsf{X} \rightsquigarrow_{\bot^*} \mathsf{Y} &::= \mathsf{AStore}\ (\Omega \times \mathsf{T})\ (\mathsf{X} \rightsquigarrow_{\bot} \mathsf{Y}) \\ \mathsf{X} \underset{\mathsf{map}^*}{\rightsquigarrow} \mathsf{Y} &::= \mathsf{AStore}\ (\Omega \times \mathsf{T})\ (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}) \\ \mathsf{X} \underset{\mathsf{pre}^*}{\rightsquigarrow} \mathsf{Y} &::= \mathsf{AStore}\ (\Omega \times \mathsf{T})\ (\mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y})\end{aligned} \tag{7.54}$$

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

### 7.7.5 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 $\mathsf{AStore}$ arrow equivalence to be more extensional.

**Definition 7.35** ($\mathsf{AStore}$ arrow equivalence)**.** *Two* $\mathsf{AStore}$ *arrow computations* $\mathsf{k}_1$ *and* $\mathsf{k}_2$ *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 $\eta_{\mathsf{a}^*}$ is a homomorphism proves $[\![\cdot]\!]_{\mathsf{a}^*}$ correctly interprets pure expressions. Because $\mathsf{AStore}$ 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 $\mathsf{AStore}$ arrow's base type's combinators obey the arrow laws listed in Section 7.2.1.

**Theorem 7.36** (pure $\mathsf{AStore}$ arrow correctness)**.** $\eta_{\mathsf{a}^*}$ *is a homomorphism.*

*Proof.* Defining $\mathsf{arr}_{\mathsf{a}^*}$ as a composition clearly meets the first homomorphism law (7.8). For homomorphism laws (7.9)–(7.11), start from the right side, expand definitions, and use arrow laws (7.14)–(7.16) to factor out $\mathsf{arr}_{\mathsf{a}}\ \mathsf{snd}$.

For (7.12), additionally $\beta$-reduce within the outer thunk, then use the lazy distributive law (7.17) to extract $\mathsf{arr_a\ snd}$. □

**Corollary 7.37** (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 $\mathsf{AStore}$ arrows. Let $\mathsf{x} \leadsto_{\mathsf{a}^*} \mathsf{y} ::= \mathsf{AStore\ s\ (x} \leadsto_{\mathsf{a}} \mathsf{y)}$ and $\mathsf{x} \leadsto_{\mathsf{b}^*} \mathsf{y} ::= \mathsf{AStore\ s\ (x} \leadsto_{\mathsf{b}^*} \mathsf{y)}$. Define

$$\begin{aligned} &\mathsf{lift}_{\mathsf{b}^*} : (\mathsf{x} \leadsto_{\mathsf{a}^*} \mathsf{y}) \Rightarrow (\mathsf{x} \leadsto_{\mathsf{b}^*} \mathsf{y}) \\ &\mathsf{lift}_{\mathsf{b}^*}\ \mathsf{f\ j}\ :=\ \mathsf{lift_b\ (f\ j)} \end{aligned} \tag{7.55}$$

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

$$\begin{tikzcd} \mathsf{x} \leadsto_{\mathsf{a}} \mathsf{y} \arrow[r, "\mathsf{lift_b}"] \arrow[d, "\eta_{\mathsf{a}^*}"'] & \mathsf{x} \leadsto_{\mathsf{b}} \mathsf{y} \arrow[d, "\eta_{\mathsf{b}^*}"] \\ \mathsf{x} \leadsto_{\mathsf{a}^*} \mathsf{y} \arrow[r, "\mathsf{lift}_{\mathsf{b}^*}"'] & \mathsf{x} \leadsto_{\mathsf{b}^*} \mathsf{y} \end{tikzcd} \tag{7.56}$$

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 $\mathsf{lift}$ or an $\eta$ is done first.

**Theorem 7.38** (natural transformation). *If $\mathsf{lift_b}$ is an arrow homomorphism, then* (7.56) *commutes.*

*Proof.* Expand definitions and apply homomorphism laws (7.9) and (7.8) for $\mathsf{lift_b}$:

$$\begin{aligned} \mathsf{lift}_{\mathsf{b}^*}\ (\eta_{\mathsf{a}^*}\ \mathsf{f}) &\equiv \lambda \mathsf{j}.\, \mathsf{lift_b\ (arr_a\ snd} \ggg_{\mathsf{a}} \mathsf{f)} \\ &\equiv \lambda \mathsf{j}.\, \mathsf{lift_b\ (arr_a\ snd)} \ggg_{\mathsf{b}} \mathsf{lift_b\ f} \\ &\equiv \lambda \mathsf{j}.\, \mathsf{arr_b\ snd} \ggg_{\mathsf{b}} \mathsf{lift_b\ f} \\ &\equiv \eta_{\mathsf{b}^*}\ (\mathsf{lift_b\ f}) \end{aligned} \tag{7.57}$$

□

**Theorem 7.39** (effectful $\mathsf{AStore}$ arrow correctness). *If $\mathsf{lift_b}$ is an arrow homomorphism from $\mathsf{a}$ to $\mathsf{b}$, then $\mathsf{lift}_{\mathsf{b}^*}$ is an arrow homomorphism from $\mathsf{a}^*$ to $\mathsf{b}^*$.*

*Proof.* For each homomorphism property (7.8)–(7.12), expand the definitions of $\mathsf{lift}_{\mathsf{b}^*}$ and the combinator, distribute $\mathsf{lift}_{\mathsf{b}}$, rewrite in terms of $\mathsf{lift}_{\mathsf{b}^*}$, and rewrite using the definition of the combinator. For example, for distribution over pairing:

$$\begin{aligned}
\mathsf{lift}_{\mathsf{b}^*}\ (\mathsf{k}_1\ \&\&\&_{\mathsf{a}^*}\ \mathsf{k}_2)\ \mathsf{j} &\equiv \mathsf{lift}_{\mathsf{b}}\ ((\mathsf{k}_1\ \&\&\&_{\mathsf{a}^*}\ \mathsf{k}_2)\ \mathsf{j}) \\
&\equiv \mathsf{lift}_{\mathsf{b}}\ (\mathsf{k}_1\ (\mathsf{left\ j})\ \&\&\&_{\mathsf{a}}\ \mathsf{k}_2\ (\mathsf{right\ j})) \\
&\equiv \mathsf{lift}_{\mathsf{b}}\ (\mathsf{k}_1\ (\mathsf{left\ j}))\ \&\&\&_{\mathsf{b}}\ \mathsf{lift}_{\mathsf{b}}\ (\mathsf{k}_2\ (\mathsf{right\ j})) \\
&\equiv (\mathsf{lift}_{\mathsf{b}^*}\ \mathsf{k}_1)\ (\mathsf{left\ j})\ \&\&\&_{\mathsf{b}}\ (\mathsf{lift}_{\mathsf{b}^*}\ \mathsf{k}_2)\ (\mathsf{right\ j}) \\
&\equiv (\mathsf{lift}_{\mathsf{b}^*}\ \mathsf{k}_1\ \&\&\&_{\mathsf{b}^*}\ \mathsf{lift}_{\mathsf{b}^*}\ \mathsf{k}_2)\ \mathsf{j}
\end{aligned} \tag{7.58}$$

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 7.40** (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 7.41** (mapping* and preimage* arrow correctness). *The following diagram commutes:*

$$\begin{tikzcd}
X \rightsquigarrow_{\perp} Y \arrow[r, "\mathsf{lift}_{\mathsf{map}}"] \arrow[d, "\eta_{\perp^*}"'] & X \underset{\mathsf{map}}{\rightsquigarrow} Y \arrow[r, "\mathsf{lift}_{\mathsf{pre}}"] \arrow[d, "\eta_{\mathsf{map}^*}"] & X \underset{\mathsf{pre}}{\rightsquigarrow} Y \arrow[d, "\eta_{\mathsf{pre}^*}"] \\
X \rightsquigarrow_{\perp^*} Y \arrow[r, "\mathsf{lift}_{\mathsf{map}^*}"'] & X \underset{\mathsf{map}^*}{\rightsquigarrow} Y \arrow[r, "\mathsf{lift}_{\mathsf{pre}^*}"'] & X \underset{\mathsf{pre}^*}{\rightsquigarrow} Y
\end{tikzcd} \tag{7.59}$$

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

As with the correctness of interpretations using the mapping and preimage arrows, the correctness of interpretations using the mapping* and preimage* arrows follows from $\mathsf{lift}_{\mathsf{map}^*}$ and $\mathsf{lift}_{\mathsf{pre}^*}$ being arrow homomorphisms, and Theorem 7.6.

**Corollary 7.42** (effectful semantic correctness). *For all expressions* $e$,

$$\begin{aligned}
[\![e]\!]_{\mathsf{pre}^*} &\equiv \mathsf{lift}_{\mathsf{pre}^*}\ (\mathsf{lift}_{\mathsf{map}^*}\ [\![e]\!]_{\perp^*}) \\
[\![e]\!]^{\Downarrow}_{\mathsf{pre}^*} &\equiv \mathsf{lift}_{\mathsf{pre}^*}\ (\mathsf{lift}_{\mathsf{map}^*}\ [\![e]\!]^{\Downarrow}_{\perp^*})
\end{aligned} \tag{7.60}$$

Unfortunately, because a statement such as "$\mathsf{k}_1 \equiv \mathsf{k}_2$" implies $\mathsf{k}_1$ terminates if and only if $\mathsf{k}_2$ terminates, we cannot use the same tactics to prove an asymmetric statement such as "$\mathsf{k}_2$ terminates with the correct answer whenever $\mathsf{k}_1$ terminates; otherwise returns $\bot$." For these kinds of termination theorems, we need to reason about the interaction of programs with their supplied branch traces.

### 7.7.6 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 7.43** (maximal domain)**.** *A computation's* ***maximal domain*** *is the largest* $\mathsf{A}^*$ *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, $\mathsf{A}^*$ 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 $\mathsf{f}$'s, and the maximal domain of $\mathsf{lift}_{\mathsf{pre}^*}\ \mathsf{g}$ is the same as $\mathsf{g}$'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. First, mapping restriction is defined by

$$\begin{aligned} &\mathsf{restrict} : (\mathsf{X} \rightharpoonup \mathsf{Y}) \Rightarrow \mathsf{Set}\ \mathsf{X} \Rightarrow (\mathsf{X} \rightharpoonup \mathsf{Y}) \\ &\mathsf{restrict}\ \mathsf{g}\ \mathsf{A}\ := \ \lambda\, \mathsf{a} \in (\mathsf{A} \cap \mathsf{domain}\ \mathsf{g}).\, \mathsf{g}\ \mathsf{a} \end{aligned} \tag{7.61}$$

**Theorem 7.44** (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 7.12) there is an $\mathsf{f} : \mathsf{X} \rightsquigarrow_{\perp} \mathsf{Y}$ such that $\mathsf{g} \equiv \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}$. Thus,

$$\begin{aligned}
\mathsf{restrict}\ (\mathsf{g}\ \mathsf{A}^{\Downarrow})\ \mathsf{A} &\equiv \mathsf{restrict}\ (\mathsf{lift}_{\mathsf{map}}\ \mathsf{f}\ \mathsf{A}^{\Downarrow})\ \mathsf{A} \qquad (7.62)\\
&\equiv \mathsf{restrict}\ (\{\langle \mathsf{a},\mathsf{b}\rangle \in \mathsf{mapping}\ \mathsf{f}\ \mathsf{A}^{\Downarrow} \mid \mathsf{b} \neq \perp\})\ \mathsf{A}\\
&\equiv \{\langle \mathsf{a},\mathsf{b}\rangle \in \mathsf{mapping}\ \mathsf{f}\ \mathsf{A} \mid \mathsf{b} \neq \perp\}\\
&\equiv \mathsf{lift}_{\mathsf{map}}\ \mathsf{f}\ \mathsf{A}\\
&\equiv \mathsf{g}\ \mathsf{A} \qquad \square
\end{aligned}$$

**Theorem 7.45** (domain closure operators)**.** *If* $\mathsf{f} : \mathsf{X} \rightsquigarrow_{\perp} \mathsf{Y}$, $\mathsf{g} : \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}$ *and* $\mathsf{h} : \mathsf{X} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Y}$*, then* $\mathsf{domain}_{\perp}\ \mathsf{f}$, $\mathsf{domain} \circ \mathsf{g}$*, 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. $\square$

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

**Theorem 7.46.** *Let* $\mathsf{f} := [\![e]\!]_{\perp *} : \mathsf{X} \rightsquigarrow_{\perp *} \mathsf{Y}$ *with maximal domain* $\mathsf{A}^*$*, and* $\mathsf{f}' := [\![e]\!]^{\Downarrow}_{\perp *}$*. For all* $\langle\langle \omega, \mathsf{t}\rangle, \mathsf{a}\rangle \in \mathsf{A}^*$*, there exists a* $\mathsf{T}' \subseteq \mathsf{T}$ *such that*

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

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

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

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

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 $\mathsf{J}$ that correspond with finite prefixes of an infinite binary tree.

**Definition 7.47** (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* $\mathsf{J}''$ *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 $\mathsf{left}$ and $\mathsf{right}$.

For a given $\mathsf{t} \in \mathsf{T}$, an index prefix $\mathsf{J}'$ serves as a convenient bounding set for the finitely many indexes $\mathsf{j}$ for which $\mathsf{t}\ \mathsf{j} \neq \bot$. Applying $\mathsf{left}$ and/or $\mathsf{right}$ 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 7.48.** *Let* $\mathsf{f} := [\![e]\!]_{\bot^*} : \mathsf{X} \rightsquigarrow_{\bot^*} \mathsf{Y}$ *with maximal domain* $\mathsf{A}^*$*, and* $\mathsf{f}' := [\![e]\!]^{\Downarrow}_{\bot^*}$*. For all* $\mathsf{a} \in ((\Omega \times \mathsf{T}) \times \mathsf{X})\backslash\mathsf{A}^*$, $\mathsf{f}'\ \mathsf{j}_0\ \mathsf{a} = \bot$*.*

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

Suppose $\mathsf{f}\ \mathsf{j}_0\ \mathsf{a}$ terminates. If an $\mathsf{if}$ subcomputation's test does not agree with $\mathsf{t}$, then $\mathsf{f}'\ \mathsf{j}_0\ \mathsf{a} = \bot$. If every $\mathsf{if}$'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 $\mathsf{j}$ for which $\mathsf{t}\ \mathsf{j} \neq \bot$ is contained within an index prefix $\mathsf{J}'$. By hypothesis, there is an $\mathsf{if}$ subcomputation at some index $\mathsf{j}'$ 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 7.49.** *For all e, the maximal domain of* $[\![e]\!]^{\Downarrow}_{\bot^*}$ *is a subset of that of* $[\![e]\!]_{\bot^*}$*.*

**Corollary 7.50.** *Let* $\mathsf{f}' := [\![e]\!]^{\Downarrow}_{\bot^*} : \mathsf{X} \rightsquigarrow_{\bot^*} \mathsf{Y}$ *with maximal domain* $\mathsf{A}^*$*, 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 7.51** (correct computation everywhere). *Let* $[\![e]\!]^{\Downarrow}_{\perp^*} : \mathsf{X} \rightsquigarrow_{\perp^*} \mathsf{Y}$ *have maximal domain* $\mathsf{A}^*$*, and* $\mathsf{X}' := (\Omega \times \mathsf{T}) \times \mathsf{X}$*. For all* $\mathsf{a} \in \mathsf{X}'$*,* $\mathsf{A} \subseteq \mathsf{X}'$ *and* $\mathsf{B} \subseteq \mathsf{Y}$*,*

$$\begin{aligned} [\![e]\!]^{\Downarrow}_{\perp^*}\ \mathsf{j}_0\ \mathsf{a} &= \mathsf{if}\ (\mathsf{a} \in \mathsf{A}^*)\ ([\![e]\!]_{\perp^*}\ \mathsf{j}_0\ \mathsf{a})\ \perp \\ [\![e]\!]^{\Downarrow}_{\mathsf{map}^*}\ \mathsf{j}_0\ \mathsf{A} &= [\![e]\!]_{\mathsf{map}^*}\ \mathsf{j}_0\ (\mathsf{A} \cap \mathsf{A}^*) \\ \mathsf{ap}_{\mathsf{pre}}\ ([\![e]\!]^{\Downarrow}_{\mathsf{pre}^*}\ \mathsf{j}_0\ \mathsf{A})\ \mathsf{B} &= \mathsf{ap}_{\mathsf{pre}}\ ([\![e]\!]_{\mathsf{pre}^*}\ \mathsf{j}_0\ (\mathsf{A} \cap \mathsf{A}^*))\ \mathsf{B} \end{aligned} \tag{7.63}$$

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.

## 7.8 Output Probabilities and Measurability

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

A mapping* computation's domain is $(\Omega \times \mathsf{T}) \times \mathsf{X}$, not $\Omega$. We assume each $\omega \in \Omega$ is randomly chosen, but not each $\mathsf{t} \in \mathsf{T}$ nor each $\mathsf{x} \in \mathsf{X}$; therefore, neither $\mathsf{T}$ nor $\mathsf{X}$ 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 $\mathsf{A} \subseteq (\Omega \times \mathsf{T}) \times \mathsf{X}$.

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

Our main results are the best we could hope for. First, the interpretations of all expressions are measurable, regardless of nontermination.

**Theorem 7.52.** *For all expressions e,* $[\![e]\!]^{\Downarrow}_{\mathsf{map}^*}$ *is measurable.*

Second, projecting a program's preimages onto $\Omega$ results in a measurable set.

**Theorem 7.53.** *If* $\mathsf{A} \subseteq (\Omega \times \mathsf{T}) \times \{\langle\rangle\}$ *is measurable, then* $\mathsf{image}\ (\mathsf{fst} \ggg \mathsf{fst})\ \mathsf{A}$ *is measurable.*

The proofs of these theorems are in Chapter 9.

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

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

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

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

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

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

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

Figure 7.10: Preimage arrow lifts needed to interpret probabilistic programs.

## 7.9 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.

### 7.9.1 Implementable Lifts

We would like to be able to compute preimages of uncountable sets, such as real intervals—but preimage (g A) B is uncomputable for most mappings g and 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 preimage, we cannot implement them.

Fortunately, we need to apply $\mathsf{arr_{pre}}$ only to certain functions. Figure 7.1 (which defines $[\![\cdot]\!]_{\mathsf{a}}$) lifts id, const b, fst and snd. Section 7.7.4, which defines the combinators used to interpret partial, probabilistic programs, lifts $\pi$ j and agrees. Measurable functions made available as language primitives, such as arithmetic, must be lifted to the preimage arrow—though to maintain generality, we put off lifting arithmetic functions until Chapter 8.

Figure 7.10 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 $\mathsf{agrees}$ 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:

$$\begin{aligned}
&\bullet\ \mathsf{A} \cap \mathsf{B},\ \varnothing,\ \{\mathsf{true}\},\ \{\mathsf{false}\} \text{ and } \{\mathsf{b}\} \text{ for every } \mathsf{const\ b} \\
&\bullet\ \mathsf{A_1} \times \mathsf{A_2},\ \mathsf{proj_1\ A} \text{ and } \mathsf{proj_2\ A} \qquad (7.64)\\
&\bullet\ \mathsf{J} \to \mathsf{X},\ \mathsf{proj\ j\ A} \text{ and } \mathsf{unproj\ j\ A\ B}
\end{aligned}$$

Before addressing representation and computability, we need to define families of sets under which these operations are closed.

**Definition 7.54** (rectangular family). $\mathsf{Rect\ X}$ *denotes the* ***rectangular family*** *of subsets of* $\mathsf{X}$. $\mathsf{Rect\ X}$ *must contain* $\varnothing$ *and* $\mathsf{X}$, *and be closed under finite intersections. Products must satisfy the following rules:*

$$\mathsf{Rect}\ \langle \mathsf{X_1}, \mathsf{X_2} \rangle \ = \ (\mathsf{Rect\ X_1}) \boxtimes (\mathsf{Rect\ X_2}) \qquad (7.65)$$

$$\mathsf{Rect}\ (\mathsf{J} \to \mathsf{X}) \ = \ (\mathsf{Rect\ X})^{\boxtimes \mathsf{J}} \qquad (7.66)$$

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

$$\mathcal{A}_1 \boxtimes \mathcal{A}_2 \ := \ \{\mathsf{A_1} \times \mathsf{A_2} \mid \mathsf{A_1} \in \mathcal{A}_1, \mathsf{A_2} \in \mathcal{A}_2\} \qquad (7.67)$$

$$\mathcal{A}^{\boxtimes \mathsf{J}} \ := \bigcup_{\mathsf{J'} \subset \mathsf{J}\ \textit{finite}} \left\{ \textstyle\prod_{\mathsf{j} \in \mathsf{J}} \mathsf{A_j} \ \middle|\ \mathsf{A_j} \in \mathcal{A}, \mathsf{j} \in \mathsf{J'} \iff \mathsf{A_j} \subset \bigcup \mathcal{A} \right\} \qquad (7.68)$$

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 $\mathsf{unproj}$.

Further, all of the operations in (7.64) 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 (7.68), sets in $\mathsf{Rect}\ (\mathsf{J} \to \mathsf{X})$ have no more than finitely many projections that are proper subsets of $\mathsf{X}$. They can be modeled by *finite* binary trees, where unrepresented projections are implicitly $\mathsf{X}$. Figure 7.11

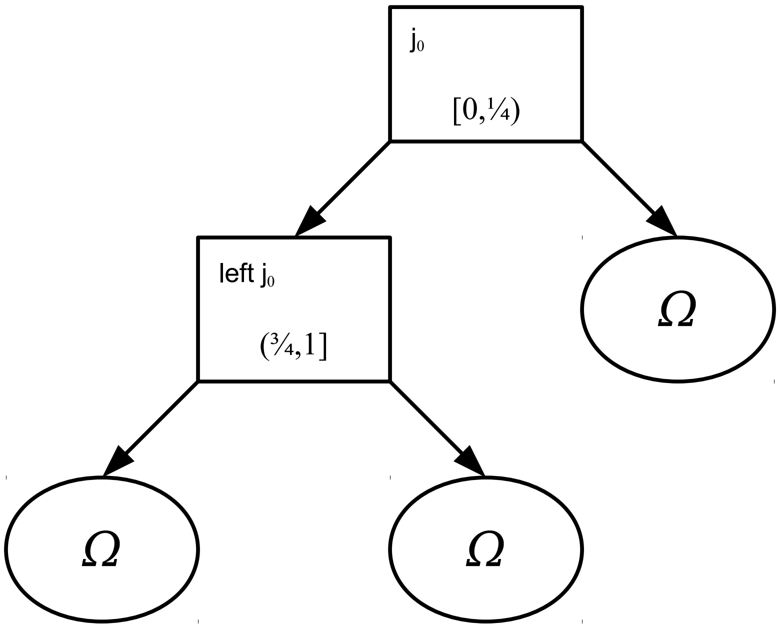


Figure 7.11: A finite binary tree model of $\mathsf{unproj}\ (\mathsf{left}\ \mathsf{j}_0)\ (\mathsf{unproj}\ \mathsf{j}_0\ \mathit{\Omega}\ [0,\frac{1}{4}))\ (\frac{3}{4},1]$. Because of $\mathit{\Omega}$'s self-similarity, and because rectangles of $\mathsf{J} \to [0,1]$ are defined so that only finitely many projections are not $[0,1]$, every rectangular subset of $\mathit{\Omega}$ has a finite binary tree model.

illustrates a model of a member of $\mathsf{Rect}\ (\mathsf{J} \to [0,1])$; i.e. a rectangular subset of $\mathit{\Omega}$.

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

**Theorem 7.55** ($\mathsf{T}$ model)**.** *If* $\mathsf{T}' \in \mathsf{Rect}\ (\mathsf{J} \to \mathsf{Bool}_\bot)$ *and* $\mathsf{j} \in \mathsf{J}$*, then* $\mathsf{proj}\ \mathsf{j}\ \mathsf{T}' = \mathsf{proj}\ \mathsf{j}\ (\mathsf{T}' \cap \mathsf{T})$*. If* $\mathsf{B} \subseteq \mathsf{Bool}_\bot$*, then* $\mathsf{unproj}\ \mathsf{j}\ (\mathsf{T}' \cap \mathsf{T})\ \mathsf{B} = \mathsf{unproj}\ \mathsf{j}\ \mathsf{T}'\ \mathsf{B} \cap \mathsf{T}$.

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

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

The statement about $\mathsf{unproj}$ is an easy corollary. □

### 7.9.2 Approximate Preimage Mapping Operations

Implementing $\mathsf{lazy}_{\mathsf{pre}}$ (defined in Figure 7.7) requires computing $\mathsf{pre}$, 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 (Figure 7.5), we see that $\mathsf{ap}_{\mathsf{pre}}$ is defined using $(\cap)$ and that $(\circ_{\mathsf{pre}})$ is defined using $\mathsf{ap}_{\mathsf{pre}}$, 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 $\mathsf{B}$ in a big union. At this point, we need to approximate.

**Theorem 7.56** (pair preimage approximation)**.** *Let* $\mathsf{g}_1 : \mathsf{X} \rightharpoonup \mathsf{Y}_1$ *and* $\mathsf{g}_2 : \mathsf{X} \rightharpoonup \mathsf{Y}_2$*. For all* $\mathsf{B} \subseteq \mathsf{Y}_1 \times \mathsf{Y}_2$*,* $\mathsf{preimage}\ \langle \mathsf{g}_1, \mathsf{g}_2 \rangle_{\mathsf{map}}\ \mathsf{B} \subseteq \mathsf{preimage}\ \mathsf{g}_1\ (\mathsf{proj}_1\ \mathsf{B}) \cap \mathsf{preimage}\ \mathsf{g}_2\ (\mathsf{proj}_2\ \mathsf{B})$.

*Proof.* By monotonicity of preimages and projections, and by Lemma 7.20. □

It is not hard to use Theorem 7.56 to show that

$$\begin{aligned}
&\langle \cdot, \cdot \rangle'_{\mathsf{pre}} : (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y}_1) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y}_2) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y}_1 \times \mathsf{Y}_2) \\
&\langle \langle \mathsf{Y}'_1, \mathsf{p}_1 \rangle, \langle \mathsf{Y}'_2, \mathsf{p}_2 \rangle \rangle'_{\mathsf{pre}} \ :=\ \langle \mathsf{Y}'_1 \times \mathsf{Y}'_2, \lambda \mathsf{B}.\, \mathsf{p}_1\ (\mathsf{proj}_1\ \mathsf{B}) \cap \mathsf{p}_2\ (\mathsf{proj}_2\ \mathsf{B}) \rangle
\end{aligned} \tag{7.69}$$

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:

$$\begin{aligned}
(\mathsf{A}_1 \times \mathsf{A}_2) \vee (\mathsf{B}_1 \times \mathsf{B}_2) &= (\mathsf{A}_1 \vee \mathsf{B}_1) \times (\mathsf{A}_2 \vee \mathsf{B}_2) \\
(\textstyle\prod_{\mathsf{j} \in \mathsf{J}} \mathsf{A}_\mathsf{j}) \vee (\textstyle\prod_{\mathsf{j} \in \mathsf{J}} \mathsf{B}_\mathsf{j}) &= \textstyle\prod_{\mathsf{j} \in \mathsf{J}} \mathsf{A}_\mathsf{j} \vee \mathsf{B}_\mathsf{j}
\end{aligned} \tag{7.70}$$

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

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

$$\begin{aligned}
&(\uplus'_{\mathsf{pre}}) : (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y}) \Rightarrow (\mathsf{X} \underset{\mathsf{pre}}{\rightharpoonup} \mathsf{Y}) \\
&\mathsf{h}_1 \uplus'_{\mathsf{pre}} \mathsf{h}_2 \;:=\; \mathsf{let}\;\; \mathsf{Y}' := \mathsf{range}_{\mathsf{pre}}\ \mathsf{h}_1 \vee \mathsf{range}_{\mathsf{pre}}\ \mathsf{h}_2 \\
&\qquad\qquad\qquad\quad\;\; \mathsf{p} := \lambda \mathsf{B}.\, \mathsf{ap}_{\mathsf{pre}}\ \mathsf{h}_1\ \mathsf{B} \vee \mathsf{ap}_{\mathsf{pre}}\ \mathsf{h}_2\ \mathsf{B} \\
&\qquad\qquad\quad\;\; \mathsf{in}\;\; \langle \mathsf{Y}', \mathsf{p} \rangle
\end{aligned} \tag{7.71}$$

To interpret programs that may not terminate, or that terminate with probability 1, we need to approximate $\mathsf{ifte}^{\Downarrow}_{\mathsf{pre}^*}$ (7.52), which is defined in terms of $\mathsf{agrees}$. Defining its approximation in terms of an approximation of $\mathsf{agrees}$ 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 $\mathsf{Bool}$ under $\mathsf{agrees}$ (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 $\mathsf{agrees}$-free equivalence:

$$\begin{aligned}
\mathsf{ifte}^{\Downarrow}_{\mathsf{pre}^*}\ \mathsf{k}_1\ \mathsf{k}_2\ \mathsf{k}_3\ \mathsf{j}\ \mathsf{A} \;\equiv\; \mathsf{let}\;\; & \langle \mathsf{C}_\mathsf{k}, \mathsf{p}_\mathsf{k} \rangle := \mathsf{k}_1\ \mathsf{j}_1\ \mathsf{A} \\
& \langle \mathsf{C}_\mathsf{b}, \mathsf{p}_\mathsf{b} \rangle := \mathsf{branch}_{\mathsf{pre}^*}\ \mathsf{j}\ \mathsf{A} \\
& \mathsf{C}_2 := \mathsf{C}_\mathsf{k} \cap \mathsf{C}_\mathsf{b} \cap \{\mathsf{true}\} \\
& \mathsf{C}_3 := \mathsf{C}_\mathsf{k} \cap \mathsf{C}_\mathsf{b} \cap \{\mathsf{false}\} \\
& \mathsf{A}_2 := \mathsf{p}_\mathsf{k}\ \mathsf{C}_2 \cap \mathsf{p}_\mathsf{b}\ \mathsf{C}_2 \\
& \mathsf{A}_3 := \mathsf{p}_\mathsf{k}\ \mathsf{C}_3 \cap \mathsf{p}_\mathsf{b}\ \mathsf{C}_3 \\
\mathsf{in}\;\; & \mathsf{k}_2\ \mathsf{j}_2\ \mathsf{A}_2 \uplus_{\mathsf{pre}} \mathsf{k}_3\ \mathsf{j}_3\ \mathsf{A}_3
\end{aligned} \tag{7.72}$$

where $\mathsf{j}_1 = \mathsf{left}\ \mathsf{j}$ and so on. Unfortunately, a straightforward approximation of this would still take unnecessary branches, when $\mathsf{A}_2$ or $\mathsf{A}_3$ overapproximates $\varnothing$.

$\mathsf{C}_\mathsf{b}$ is the branch trace projection at $\mathsf{j}$ (with $\perp$ removed). The set of indexes for which $\mathsf{C}_\mathsf{b}$ 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 $\mathsf{ifte}^{\Downarrow'}_{\mathsf{pre}^*}$ takes *no branches* when $\mathsf{C}_\mathsf{b} = \{\mathsf{true}, \mathsf{false}\}$, but approximates with a finite computation, expressions interpreted using $\mathsf{ifte}^{\Downarrow'}_{\mathsf{pre}^*}$ 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 $\mathsf{Y}$, and it returns subsets of $\mathsf{A}_2 \uplus \mathsf{A}_3$.

$$\begin{aligned}
&X \underset{\text{pre}}{\rightharpoonup}' Y ::= \langle \mathsf{Rect}\ Y, \mathsf{Rect}\ Y \Rightarrow \mathsf{Rect}\ X \rangle \\
&\varnothing'_{\text{pre}} \ := \ \langle \varnothing, \lambda B.\, \varnothing \rangle \\
&\mathsf{ap}'_{\text{pre}} : (X \underset{\text{pre}}{\rightharpoonup}' Y) \Rightarrow \mathsf{Rect}\ Y \Rightarrow \mathsf{Rect}\ X \\
&\mathsf{ap}'_{\text{pre}}\ \langle Y', p \rangle\ B := p\ (B \cap Y') \\
&(\circ'_{\text{pre}}) : (Y \underset{\text{pre}}{\rightharpoonup}' Z) \Rightarrow (X \underset{\text{pre}}{\rightharpoonup}' Y) \Rightarrow (X \underset{\text{pre}}{\rightharpoonup}' Z) \\
&\langle Z', p_2 \rangle \circ'_{\text{pre}} h_1 := \langle Z', \lambda C.\, \mathsf{ap}'_{\text{pre}}\ h_1\ (p_2\ C) \rangle
\end{aligned}$$

$$\begin{aligned}
&\langle \cdot, \cdot \rangle'_{\text{pre}} : (X \underset{\text{pre}}{\rightharpoonup}' Y_1) \Rightarrow (X \underset{\text{pre}}{\rightharpoonup}' Y_2) \Rightarrow (X \underset{\text{pre}}{\rightharpoonup}' Y_1 \times Y_2) \\
&\langle \langle Y'_1, p_1 \rangle, \langle Y'_2, p_2 \rangle \rangle'_{\text{pre}} \ := \\
&\quad \langle Y'_1 \times Y'_2, \lambda B.\, p_1\ (\mathsf{proj}_1\ B) \cap p_2\ (\mathsf{proj}_2\ B) \rangle \\
&(\uplus'_{\text{pre}}) : (X \underset{\text{pre}}{\rightharpoonup}' Y) \Rightarrow (X \underset{\text{pre}}{\rightharpoonup}' Y) \Rightarrow (X \underset{\text{pre}}{\rightharpoonup}' Y) \\
&\langle Y'_1, p_1 \rangle \uplus'_{\text{pre}} \langle Y'_2, p_2 \rangle \ := \\
&\quad \langle Y'_1 \vee Y'_2, \lambda B.\, \mathsf{ap}'_{\text{pre}}\ \langle Y'_1, p_1 \rangle\ B \vee \mathsf{ap}'_{\text{pre}}\ \langle Y'_2, p_2 \rangle\ B \rangle
\end{aligned}$$

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

$$\begin{aligned}
&X \underset{\text{pre}}{\rightsquigarrow}' Y ::= \mathsf{Rect}\ X \Rightarrow (X \underset{\text{pre}}{\rightharpoonup}' Y) \\
&(\ggg'_{\text{pre}}) : (X \underset{\text{pre}}{\rightsquigarrow}' Y) \Rightarrow (Y \underset{\text{pre}}{\rightsquigarrow}' Z) \Rightarrow (X \underset{\text{pre}}{\rightsquigarrow}' Z) \\
&(h_1 \ggg'_{\text{pre}} h_2)\ A \ := \ \mathsf{let}\ \ h'_1 := h_1\ A \\
&\qquad\qquad\qquad\qquad\quad\ \ h'_2 := h_2\ (\mathsf{range}'_{\text{pre}}\ h'_1) \\
&\qquad\qquad\qquad\qquad \mathsf{in}\ \ h'_2 \circ'_{\text{pre}} h'_1 \\
&(\&\&\&'_{\text{pre}}) : (X \underset{\text{pre}}{\rightsquigarrow}' Y_1) \Rightarrow (X \underset{\text{pre}}{\rightsquigarrow}' Y_2) \Rightarrow (X \underset{\text{pre}}{\rightsquigarrow}' \langle Y_1, Y_2 \rangle) \\
&(h_1 \ \&\&\&'_{\text{pre}}\ h_2)\ A \ := \ \langle h_1\ A, h_2\ A \rangle'_{\text{pre}}
\end{aligned}$$

$$\begin{aligned}
&\mathsf{ifte}'_{\text{pre}} : (X \underset{\text{pre}}{\rightsquigarrow}' \mathsf{Bool}) \Rightarrow (X \underset{\text{pre}}{\rightsquigarrow}' Y) \Rightarrow (X \underset{\text{pre}}{\rightsquigarrow}' Y) \Rightarrow (X \underset{\text{pre}}{\rightsquigarrow}' Y) \\
&\mathsf{ifte}'_{\text{pre}}\ h_1\ h_2\ h_3\ A \ := \ \mathsf{let}\ \ h'_1 := h_1\ A \\
&\qquad\qquad\qquad\qquad\qquad\quad\ \ h'_2 := h_2\ (\mathsf{ap}'_{\text{pre}}\ h'_1\ \{\mathsf{true}\}) \\
&\qquad\qquad\qquad\qquad\qquad\quad\ \ h'_3 := h_3\ (\mathsf{ap}'_{\text{pre}}\ h'_1\ \{\mathsf{false}\}) \\
&\qquad\qquad\qquad\qquad\qquad \mathsf{in}\ \ h'_2 \uplus'_{\text{pre}} h'_3 \\
&\mathsf{lazy}'_{\text{pre}} : (1 \Rightarrow (X \underset{\text{pre}}{\rightsquigarrow}' Y)) \Rightarrow (X \underset{\text{pre}}{\rightsquigarrow}' Y) \\
&\mathsf{lazy}'_{\text{pre}}\ h\ A \ := \ \mathsf{if}\ (A = \varnothing)\ \varnothing'_{\text{pre}}\ (h\ 0\ A)
\end{aligned}$$

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

$$\begin{aligned}
&X \underset{\text{pre*}}{\rightsquigarrow}' Y ::= \mathsf{AStore}\ (\Omega \times T)\ (X \underset{\text{pre}}{\rightsquigarrow}' Y) \\
&\mathsf{random}'_{\text{pre*}} : X \underset{\text{pre*}}{\rightsquigarrow}' [0,1] \\
&\mathsf{random}'_{\text{pre*}}\ j \ := \\
&\quad \mathsf{fst}_{\text{pre}} \ggg'_{\text{pre}} \mathsf{fst}_{\text{pre}} \ggg'_{\text{pre}} \pi_{\text{pre}}\ j \\
&\mathsf{branch}'_{\text{pre*}} : X \underset{\text{pre*}}{\rightsquigarrow}' \mathsf{Bool} \\
&\mathsf{branch}'_{\text{pre*}}\ j \ := \\
&\quad \mathsf{fst}_{\text{pre}} \ggg'_{\text{pre}} \mathsf{snd}_{\text{pre}} \ggg'_{\text{pre}} \pi_{\text{pre}}\ j \\
&\mathsf{fst}'_{\text{pre*}} := \eta'_{\text{pre*}}\ \mathsf{fst}_{\text{pre}};\ \cdots
\end{aligned}$$

$$\begin{aligned}
&\mathsf{ifte}^{\Downarrow\prime}_{\text{pre*}} : (X \underset{\text{pre*}}{\rightsquigarrow}' \mathsf{Bool}) \Rightarrow (X \underset{\text{pre*}}{\rightsquigarrow}' Y) \Rightarrow (X \underset{\text{pre*}}{\rightsquigarrow}' Y) \Rightarrow (X \underset{\text{pre*}}{\rightsquigarrow}' Y) \\
&\mathsf{ifte}^{\Downarrow\prime}_{\text{pre*}}\ k_1\ k_2\ k_3\ j \ := \\
&\quad \mathsf{let}\ \ \langle C_k, p_k \rangle := k_1\ (\mathsf{left}\ j)\ A \\
&\qquad\quad \langle C_b, p_b \rangle := \mathsf{branch}_{\text{pre*}}\ j\ A \\
&\qquad\qquad C_2 := C_k \cap C_b \cap \{\mathsf{true}\} \\
&\qquad\qquad C_3 := C_k \cap C_b \cap \{\mathsf{false}\} \\
&\qquad\qquad A_2 := p_k\ C_2 \cap p_b\ C_2 \\
&\qquad\qquad A_3 := p_k\ C_3 \cap p_b\ C_3 \\
&\quad\ \ \mathsf{in}\ \ \mathsf{case}\ \ C_b \\
&\qquad\quad \{\mathsf{true}, \mathsf{false}\} \ \longrightarrow\ \langle \top, \lambda B.\, A_2 \vee A_3 \rangle \\
&\qquad\quad \{\mathsf{true}\} \qquad\quad\ \ \longrightarrow\ k_2\ (\mathsf{left}\ (\mathsf{right}\ j))\ A_2 \\
&\qquad\quad \{\mathsf{false}\} \qquad\quad \longrightarrow\ k_3\ (\mathsf{right}\ (\mathsf{right}\ j))\ A_3
\end{aligned}$$

(c) Preimage* arrow combinators for probabilistic choice and guaranteed termination. Figure 7.8 (AStore arrow transformer) defines $\eta'_{\text{pre*}}$, $(\ggg'_{\text{pre*}})$, $(\&\&\&'_{\text{pre*}})$, $\mathsf{ifte}'_{\text{pre*}}$ and $\mathsf{lazy}'_{\text{pre*}}$.

Figure 7.12: Implementable arrows that approximate preimage arrows. Specific lifts such as $\mathsf{fst}_{\text{pre}} := \mathsf{arr}_{\text{pre}}\ \mathsf{fst}$ are computable (see Figure 7.10), but $\mathsf{arr}'_{\text{pre}}$ is not.

Therefore, $\mathsf{ifte}^{\Downarrow\prime}_{\text{pre*}}$ may return $\langle Y, \lambda B.\, A_2 \vee A_3 \rangle$ when $C_b = \{\mathsf{true}, \mathsf{false}\}$. 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 $\mathsf{Rect}\ Y$.

Figure 7.12b defines the final approximating preimage arrow. This arrow, the lifts in Figure 7.10, and the semantic function $[\![\cdot]\!]_{\mathsf{a}}$ in Figure 7.1 define an approximating semantics for partial, probabilistic programs.

### 7.9.3 Correctness

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

To use structural induction on the interpretation of $e$, we need a theorem that allows representing it as a finite expression (Definition 9.28). Because $\mathsf{ifte}^{\Downarrow'}_{\mathsf{pre}^*}$ does not branch when either branch could be taken, an equivalent finite expression exists for each rectangular domain subset $\mathsf{A}$.

**Theorem 7.57** (equivalent finite expression)**.** *For all* $\mathsf{A} \in \mathsf{Rect}\ \langle\langle \Omega, \mathsf{T}\rangle, \mathsf{X}\rangle$*, there exists a finite expression* $e'$ *for which, if* $\mathsf{h}'' := [\![e']\!]^{\Downarrow'}_{\mathsf{pre}^*}$*, then* $\mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}''\ \mathsf{j}_0\ \mathsf{A})\ \mathsf{B} = \mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}'\ \mathsf{j}_0\ \mathsf{A})\ \mathsf{B}$ *for all* $\mathsf{B} \in \mathsf{Rect}\ \mathsf{Y}$*.*

*Proof.* Let $\mathsf{T}' := \mathsf{proj}_2\ (\mathsf{proj}_1\ \mathsf{A})$, and let the index prefix $\mathsf{J}'$ contain every $\mathsf{j}'$ 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 $\mathsf{J}'$ with the equivalent expression $\mathsf{if}\ e_1\ \bot\ \bot$. Because $e$ is well-defined, recurrences must be guarded by $\mathsf{if}$, so this process terminates after finitely many applications. □

**Corollary 7.58** (terminating)**.** *For all* $\mathsf{A} \in \mathsf{Rect}\ \langle\langle \Omega, \mathsf{T}\rangle, \mathsf{X}\rangle$ *and* $\mathsf{B} \in \mathsf{Rect}\ \mathsf{Y}$*,* $\mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}'\ \mathsf{j}_0\ \mathsf{A})\ \mathsf{B}$ *terminates.*

**Theorem 7.59** (sound)**.** *For all* $\mathsf{A} \in \mathsf{Rect}\ \langle\langle \Omega, \mathsf{T}\rangle, \mathsf{X}\rangle$ *and* $\mathsf{B} \in \mathsf{Rect}\ \mathsf{Y}$*,* $\mathsf{ap}_{\mathsf{pre}}\ (\mathsf{h}\ \mathsf{j}_0\ \mathsf{A})\ \mathsf{B} \subseteq \mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}'\ \mathsf{j}_0\ \mathsf{A})\ \mathsf{B}$*.*

*Proof.* By construction and Corollary 7.58 (recall non-"$\equiv$" statements imply termination). □

**Theorem 7.60** (monotone)**.** $\mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}'\ \mathsf{j}_0\ \mathsf{A})\ \mathsf{B}$ *is monotone in both* $\mathsf{A}$ *and* $\mathsf{B}$*.*

*Proof.* Lattice operators $(\cap)$ and $(\vee)$ are monotone, as is $(\times)$. Therefore, $\mathsf{id}_{\mathsf{pre}}$ and the other lifts in Figure 7.10 are monotone, and each approximating preimage arrow combinator preserves monotonicity. Approximating preimage* arrow combinators, which are defined in terms of approximating preimage arrow combinators (Figure 7.12b) likewise preserve monotonicity, as does $\eta'_{\mathsf{pre}^*}$; 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 $\mathsf{A}$ for which $\mathsf{C_b} = \{\mathsf{true}\}$ or $\mathsf{C_b} = \{\mathsf{false}\}$, and thus preserves monotonicity in those cases. 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 $\mathsf{A}$, suppose $\mathsf{A}_1 \subseteq \mathsf{A}_2$. Apply Theorem 7.57 with $\mathsf{A}_1$ to yield $e'$; clearly, it is also an equivalent finite expression for $\mathsf{A}_2$. Monotonicity follows from structural induction on the interpretation of $e'$.

For monotonicity in $\mathsf{B}$, apply Theorem 7.57 with a fixed $\mathsf{A}$. □

**Theorem 7.61** (decreasing)**.** *If* $\mathsf{A} \in \mathsf{Rect}\ \langle\langle \Omega, \mathsf{T}\rangle, \mathsf{X}\rangle$ *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 Figure 7.10 are decreasing.

By definition and applying basic lattice properties,

$$\begin{aligned}
\mathsf{ap}'_{\mathsf{pre}}\ ((\mathsf{h}_1 \ggg'_{\mathsf{pre}} \mathsf{h}_2)\ \mathsf{A})\ \mathsf{B} &\equiv \mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}_1\ \mathsf{A})\ \mathsf{B}' \ \text{ for some } \mathsf{B}' \\
\mathsf{ap}'_{\mathsf{pre}}\ ((\mathsf{h}_1 \mathbin{\&\&\&'_{\mathsf{pre}}} \mathsf{h}_2)\ \mathsf{A})\ \mathsf{B} &\equiv \mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}_1\ \mathsf{A})\ (\mathsf{proj}_1\ \mathsf{B}) \ \cap\ \mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}_2\ \mathsf{A})\ (\mathsf{proj}_2\ \mathsf{B}) \\
\mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{ifte}'_{\mathsf{pre}}\ \mathsf{h}_1\ \mathsf{h}_2\ \mathsf{h}_3\ \mathsf{A})\ \mathsf{B} &\equiv \mathsf{let}\ \ \mathsf{A}_2 := \mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}_1\ \mathsf{A})\ \{\mathsf{true}\} \\
&\qquad\quad\ \mathsf{A}_3 := \mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}_1\ \mathsf{A})\ \{\mathsf{false}\} \\
&\qquad \mathsf{in}\ \ \mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}_2\ \mathsf{A}_2)\ \mathsf{B}\ \vee\ \mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}_3\ \mathsf{A}_3)\ \mathsf{B} \\
\mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{lazy}'_{\mathsf{pre}}\ \mathsf{h}\ \mathsf{A})\ \mathsf{B} &\equiv \mathsf{if}\ (\mathsf{A} = \varnothing)\ \varnothing\ (\mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}\ 0\ \mathsf{A})\ \mathsf{B})
\end{aligned} \tag{7.73}$$

Thus, approximating preimage arrow combinators return decreasing computations when given decreasing computations. This property transfers trivially to approximating preimage* arrow combinators. Apply Theorem 7.57 and use structural induction. □

### 7.9.4 Preimage Refinement Algorithm

Given these properties, we might try to compute exact preimages of $\mathsf{B}$ by computing preimages with respect to increasingly fine discretizations of $\mathsf{A}$.

**Definition 7.62** (preimage refinement algorithm). *Let* $\mathsf{B} \in \mathsf{Rect}\ \mathsf{Y}$ *and*

$$\begin{aligned} &\mathsf{refine} : \mathsf{Rect}\ \langle\langle \Omega, \mathsf{T}\rangle, \mathsf{X}\rangle \Rightarrow \mathsf{Rect}\ \langle\langle \Omega, \mathsf{T}\rangle, \mathsf{X}\rangle \\ &\mathsf{refine}\ \mathsf{A}\ :=\ \mathsf{ap}'_{\mathsf{pre}}\ (\mathsf{h}'\ \mathsf{j}_0\ \mathsf{A})\ \mathsf{B} \end{aligned} \tag{7.74}$$

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

$$\begin{aligned} &\mathsf{refine}^* : \mathsf{Set}\ (\mathsf{Rect}\ \langle\langle \Omega, \mathsf{T}\rangle, \mathsf{X}\rangle) \Rightarrow \mathsf{Set}\ (\mathsf{Rect}\ \langle\langle \Omega, \mathsf{T}\rangle, \mathsf{X}\rangle) \\ &\mathsf{refine}^*\ \mathcal{A}\ :=\ \mathsf{image}\ \mathsf{refine}\ \left(\textstyle\bigcup_{\mathsf{A}\in\mathcal{A}} \mathsf{split}\ \mathsf{A}\right) \end{aligned} \tag{7.75}$$

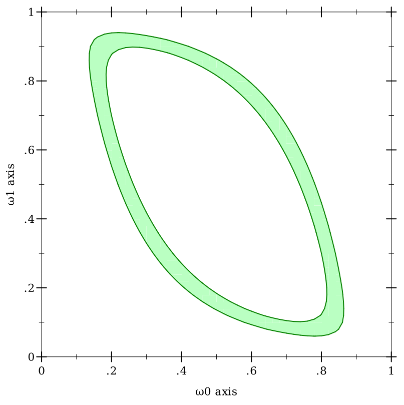
*For any positive-measure* $\mathsf{A}_0 \in \mathsf{Rect}\ \langle\langle \Omega, \mathsf{T}\rangle, \mathsf{X}\rangle$*, iterate* $\mathsf{refine}^*$ *on* $\{\mathsf{A}_0\}$.

Figure 7.13 illustrates the preimage refinement algorithm.

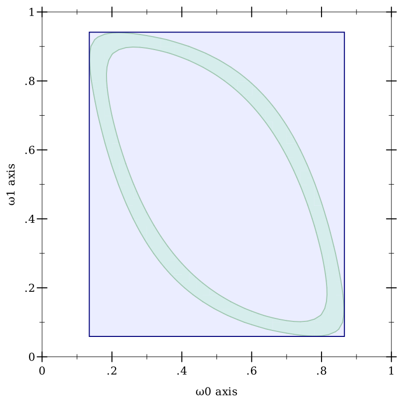
Theorem 7.61 (decreasing) guarantees $\mathsf{refine}\ \mathsf{A}$ is never larger than $\mathsf{A}$. Theorem 7.60 (monotone) guarantees refining a *partition* of $\mathsf{A}$ never does worse than refining $\mathsf{A}$ itself. Theorem 7.59 (sound) guarantees the algorithm is sound: the exact preimage of $\mathsf{B}$ is always contained in the covering partition $\mathsf{refine}^*$ returns.

We would like it to be precise in the limit, up to null sets: covering partitions' measures should converge to the true 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 $\mathsf{rational?}\ \mathsf{random}$, where $\mathsf{rational?}$ returns $\mathsf{true}$ when its argument is rational and loops otherwise. (This is definable using a $(\leq)$ primitive.) The preimage of $\{\mathsf{true}\}$ (the rational numbers) has measure 0, but its Jordan outer measure is 1.

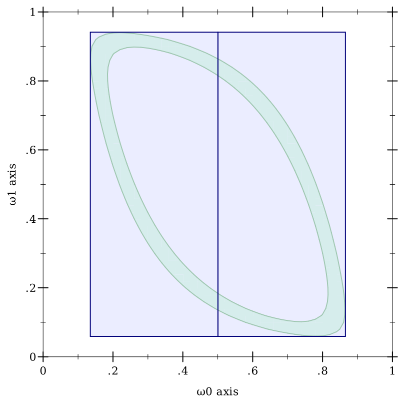
We conjecture that a minimal requirement for preimage refinement's measures to



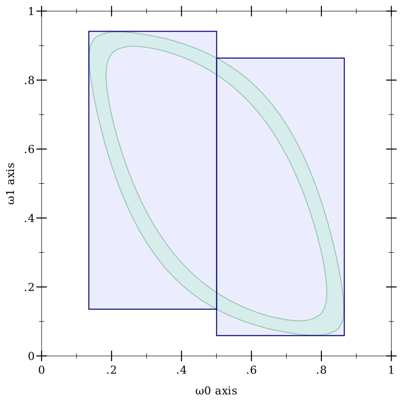
(a) Exact preimage of $\mathsf{B}$



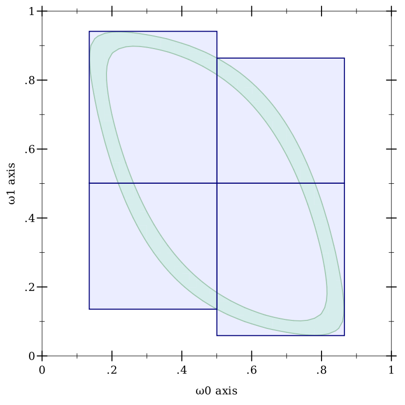
(b) Initial partition $\mathcal{A}_0 := \{\mathsf{A}_0\}$



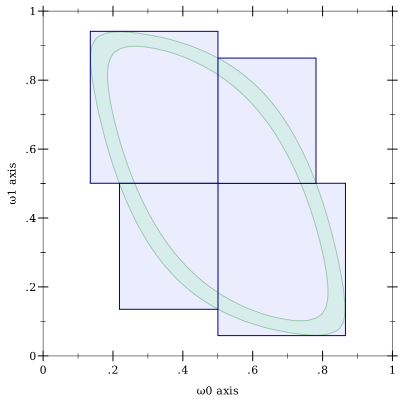
(c) $\mathcal{A}'_0 := \bigcup_{\mathsf{A}\in\mathcal{A}_0} \mathsf{split\ A}$



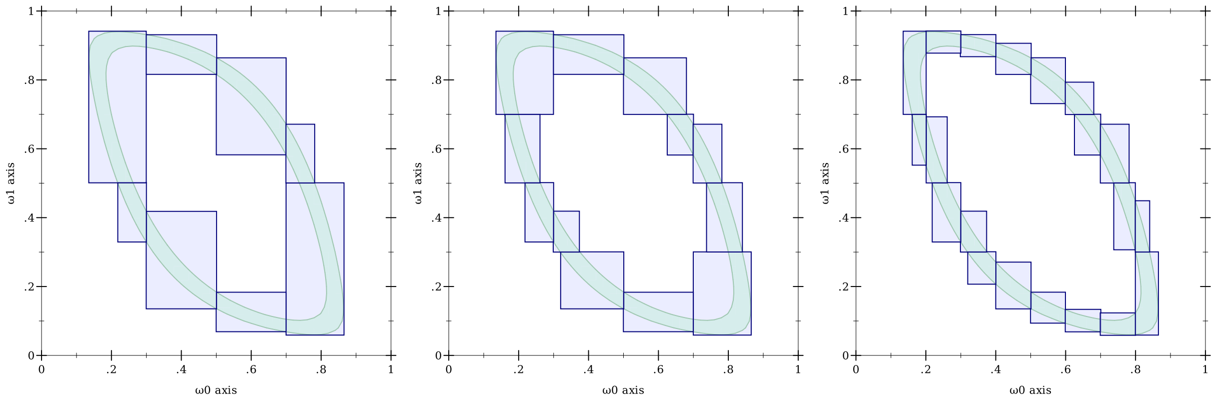
(d) $\mathcal{A}_1 :=$ image refine $\mathcal{A}'_0$



(e) $\mathcal{A}'_1 := \bigcup_{\mathsf{A}\in\mathcal{A}_1} \mathsf{split\ A}$



(f) $\mathcal{A}_2 :=$ image refine $\mathcal{A}'_1$



(g) Further preimage refinements $\mathcal{A}_3 := \mathsf{refine}^* \ \mathcal{A}_2$, $\mathcal{A}_4 := \mathsf{refine}^* \ \mathcal{A}_3$ and $\mathcal{A}_5 := \mathsf{refine}^* \ \mathcal{A}_4$

Figure 7.13: Preimage refinement algorithm on $\langle\langle\Omega, \mathsf{T}\rangle, \mathsf{X}\rangle$. Only two dimensions of $\Omega$ are shown. In this example, the covering partition appears to converge in measure to the exact. (In the worst case, 8.12a represents an open set, which in the limit, preimage refinement overapproximates only on the boundary.)

converge is that the program must terminate 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.

## 7.10 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***.

If sets are restricted to be finite, the arrows used as translation targets in the exact semantics, defined in 7.2, 7.4, 7.5, 7.7 and 7.8, can be implemented directly in any practical $\lambda$-calculus. Computing exact preimages is very inefficient, even under the interpretations of very small programs. Still, we have found our Typed Racket [47] implementation useful for finding theorem candidates and counterexamples.

Given a rectangular set library, the approximating preimage arrows defined in Figures 7.10 and 7.12b can be implemented with few changes in any practical $\lambda$-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`. [XXX: move]

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.

Chapter 8 details the implementation of Dr. Bayes.

## 7.11 Conclusions

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.

Using arrows drastically simplifies the correctness proofs. Almost every semantic correctness theorem proceeds from a proof that a lift distributes over five combinators. There are seven theorems in total corresponding to the morphisms in our roadmap (7.3), but the three center morphisms (pointing downward) are done in one proof, as are the two bottom morphisms (pointing rightward). In total, there are 20 cases, plus 11 for the original (and very simple) proof by induction that arrow homomorphisms distribute over program terms.

In contrast, the corresponding theorems with separate semantic functions would require seven proofs by structural induction over at least 11 rules (12 for programs that access the random store), for a total of at least 77 cases. This reduction in complexity by semantic abstraction would have been difficult without targeting $\lambda_{\text{ZFC}}$, which allows such arrows to carry out uncountably infinite computations.

Further, because the approximating semantics targets a computable $\lambda_{\text{ZFC}}$ sublanguage, it is directly implementable. The next chapter details creating a *practical* implementation.

# Chapter 8

# Preimage Computation Implementation

## 8.1 Introduction

To maintain generality, the preceeding chapter leaves out some details; in particular, how to

1. Represent and compute with abstract sets.
2. Compute approximate preimages under real primitives (especially $\mathbb{R} \times \mathbb{R} \to \mathbb{R}$ functions).
3. Use preimage refinement to compute conditional probabilities *efficiently*.

In this chapter, we provide these details.

## 8.2 Abstract Sets and Concrete Values

While any kind of abstract sets with finite representations and computable operations would do, we use rectangles for their efficiency and simplicity, especially emptiness checking.

In a host language such as Haskell with sufficiently advanced typeclasses or an equivalent, it is possible to use polymorphism to represent rectangles in an extensible way. For each required value type $\mathsf{X}$, we need to define, in the host language,

- A type of rectangles of $\mathsf{X}$ with an associated type of members of $\mathsf{X}$.
- Representations of the sets $\varnothing$ and $\mathsf{X}$ (i.e. $\top$ in Figure 7.12c).
- Intersection $(\cap)$ and join $(\vee)$.
- A singleton constructor and a membership test.

The membership test is used in sampling algorithms, to determine whether points sampled from the rectangular cover of a preimage set lie within the preimage set.

```
class Eq s => Set s where
  type Member s                     -- type of members of s
  empty :: s                        -- lattice bottom
  univ  :: s                        -- lattice top
  (/\) :: s -> s -> s               -- intersection
  (\/) :: s -> s -> s               -- join
  singleton :: Member s -> s        -- singleton set
  member :: Member s -> s -> Bool   -- membership test
```

Figure 8.1: A Haskell typeclass for rectangular sets.

Figure 8.1 shows the definition of a Haskell typeclass `Set` that encapsulates these types, values and operations. `Set` uses a type family `Member` to associate with each rectangle type `s` a value type `Member s`. Each required $\mathsf{Rect}\ \mathsf{X}$ is represented by a type instance of `Set`. For example, the code in Figure 8.2 represents $\mathsf{Rect}\ \langle \mathsf{X}_1, \mathsf{X}_2 \rangle$ using the data type `PairSet s1 s2`, and declares it as an instance of `Set` by defining `Member (PairSet s1 s2)` to be a 2-tuple type, and defining the empty pair set, the universal pair set, and the required operations.

In a language without typeclasses and type families, or equally expressive type-level features, the representation is best done monomorphically:[1] all required value types $\mathsf{X}_1, \mathsf{X}_2, ..., \mathsf{X}_n$ are considered as one universal type $\mathsf{X} := \bigcup_{i=1}^{n} \mathsf{X}_i$. The same types, values and operations are necessary; i.e. the type of rectangles of $\mathsf{X}$ and of values of $\mathsf{X}$, representations of $\varnothing$ and $\mathsf{X}$, intersection, join, singleton, and membership.

Of course, it is good factorization to have separate representations for each type $\mathsf{X}_i$. Figure 8.3 shows a fragment of a Typed Racket representation of rectangular sets, operations and values, with rectangles of $\mathbb{R}$ (`Real-Set`), $\mathsf{Bool}$ (`Bool-Set`), $\langle \mathsf{X}, \mathsf{X} \rangle$ (`Pair-Set`), $\{\langle\rangle\}$ (`Null-Set`), $\mathsf{J} \to [0,1]$ (`Omega-Set`) and $\mathsf{J} \to \mathsf{Bool}_\perp$ (`Trace-Set`). The type `Nonempty-Set` is the union of these types and `Univ-Set`, which represents $\mathsf{X}$. The `Set` type additionally represents $\varnothing$. The `Value` type represents members of $\mathsf{X}$ and is defined similarly, but mostly uses built-in Racket types such as `Real` and `Pair`.

The `intersect` function receives any two `Set` instances and dispatches to a more

[1]It is possible to encode typeclasses and type families into a polymorphic type system by parameterizing every function on function tables that represent typeclasses, but this encoding is difficult to work with.

```
data PairSet s1 s2 = EmptyPairSet | UnivPairSet | PairSet s1 s2
  deriving(Show, Eq)

prod :: (Set s1, Set s2) => s1 -> s2 -> PairSet s1 s2
prod a1 a2 | a1 == empty || a2 == empty  = EmptyPairSet
           | a1 == univ  && a2 == univ   = UnivPairSet
           | otherwise                   = PairSet a1 a2

instance (Set s1, Set s2) => Set (PairSet s1 s2) where
  type Member (PairSet s1 s2) = (Member s1, Member s2)

  empty = EmptyPairSet
  univ  = UnivPairSet

  EmptyPairSet /\ _ = EmptyPairSet
  _ /\ EmptyPairSet = EmptyPairSet
  UnivPairSet /\ a = a
  a /\ UnivPairSet = a
  PairSet a1 a2 /\ PairSet b1 b2 = prod (a1 /\ b1) (a2 /\ b2)

  EmptyPairSet \/ a = a
  a \/ EmptyPairSet = a
  UnivPairSet \/ _ = UnivPairSet
  _ \/ UnivPairSet = UnivPairSet
  PairSet a1 a2 \/ PairSet b1 b2 = prod (a1 \/ b1) (a2 \/ b2)

  member EmptyPairSet _ = False
  member UnivPairSet  _ = True
  member (x1,x2) (PairSet a1 a2) = member x1 a1 && member x2 a2

  singleton (x1,x2) = prod (singleton x1) (singleton x2)
```

Figure 8.2: An instance of `Set`, representing rectangular sets $\mathsf{Rect}\ \langle \mathsf{X}_1, \mathsf{X}_2 \rangle$.

specific intersection function based on their runtime data types. The intersection of two differently typed rectangles is empty because the types represent disjoint sets. Every operation on `Set` or `Value` is computed in a similar way.

Typed Racket's true union types make it easy to represent *nonempty* sets of pairs, simply by leaving `Empty-Set` out of the type in `Pair-Set`'s fields. The `pair-set-intersect` function is derived from the identity $(\mathsf{A}_1 \times \mathsf{A}_2) \cap (\mathsf{B}_1 \times \mathsf{B}_2) \ = \ (\mathsf{A}_1 \cap \mathsf{B}_1) \times (\mathsf{A}_2 \cap \mathsf{B}_2)$.

Subsets of $\mathsf{Bool}$ are easy to represent. Subsets of $\{\langle\rangle\}$ are trivial.

We need the representation of real sets to have closed intervals because $\Omega = \mathsf{J} \to [0,1]$.

```
;; Lattice bottom and top
(define-singleton-type Empty-Set empty-set)
(define-singleton-type Univ-Set univ-set)

;; Type of rectangular sets
(define-type Set (U Empty-Set Nonempty-Set))

;; Type of *nonempty* rectangular sets
(define-type Nonempty-Set
  (U Univ-Set Real-Set Bool-Set Pair-Set Null-Set Omega-Set Trace-Set))

;; Type of members of rectangular sets
(define-type Value
  (Rec Value (U Real Boolean (Pair Value Value) Null Omega-Val Trace-Val)))

(: intersect (Set Set -> Set))
;; Returns the intersection of two rectangular sets
(define (intersect A B)
  (cond [(and (real-set? A) (real-set? B))  (real-set-intersect A B)]
        [(and (bool-set? A) (bool-set? B))  (bool-set-intersect A B)]
        [(and (pair-set? A) (pair-set? B))  (pair-set-intersect A B)]
        [(and (null-set? A) (null-set? B))  null-set]
        [(and (omega-set? A) (omega-set? B))  (omega-set-intersect A B)]
        [(and (trace-set? A) (trace-set? B))  (trace-set-intersect A B)]
        [(univ-set? A)  B]
        [(univ-set? B)  A]
        [else  empty-set]))

;; Type of *nonempty* rectangular sets of pairs
(struct: Pair-Set ([fst : Nonempty-Set] [snd : Nonempty-Set])
  #:transparent)
(define pair-set? Pair-Set?)

(: prod (Set Set -> (U Empty-Set Pair-Set)))
;; Constructs pair sets from possibly empty sets
(define (prod A1 A2)
  (if (or (empty-set? A1) (empty-set? A2))
      empty-set
      (Pair-Set A1 A2)))

(: pair-set-intersect (Pair-Set Pair-Set -> (U Empty-Set Pair-Set)))
;; Intersection specialized to pair sets
(define (pair-set-intersect A B)
  (match-define (Pair-Set A1 A2) A)
  (match-define (Pair-Set B1 B2) B)
  (prod (intersect A1 B1) (intersect A2 B2)))
```

Figure 8.3: Part of a Typed Racket implementation of monomorphic, rectangular sets.

```
;; Type of rectangular real sets (intervals)
(struct: Real-Set ([mn : Flonum] [mx : Flonum] [mn? : Boolean] [mx? : Boolean])
  #:transparent)

(: interval (Flonum Flonum Boolean Boolean -> (U Empty-Set Real-Set)))

(: real-set-intersect (Real-Set Real-Set -> (U Empty-Set Real-Set)))
;; Intersection specialized to real sets
(define (real-set-intersect A B)
  (match-define (Real-Set a1 a2 a1? a2?) A)
  (match-define (Real-Set b1 b2 b1? b2?) B)
  (define-values (c1 c1?)
    (cond [(> a1 b1)  (values a1 a1?)]
          [(< a1 b1)  (values b1 b1?)]
          [else       (values a1 (and a1? b1?))]))
  (define-values (c2 c2?)
    (cond [(> a2 b2)  (values b2 b2?)]
          [(< a2 b2)  (values a2 a2?)]
          [else       (values a2 (and a2? b2?))]))
  (interval c1 c2 c1? c2?))

(: real-set-singleton (Real -> Real-Set))
;; Returns the smallest Real-Set containing the given Real
(define (real-set-singleton a)
  (define b (fl a))
  (cond [(not (rational? a))  ; No +nan.0 or infinities
         (raise-argument-error 'real-set-singleton "rational?" a)]
        [(< b a)  (Real-Set b (flnext b) #f #f)]
        [(< a b)  (Real-Set (flprev b) b #f #f)]
        [else     (Real-Set b b #t #t)]))
```

Figure 8.4: Part of a Typed Racket implementation of closed, open and half-open intervals.

Because preimage refinement splits $\Omega$ into disjoint sets, we also need half-open and open intervals. We therefore need to represent intervals with four values: two extended-real endpoints, and two booleans that determine whether each endpoint is in the interval (i.e. whether each endpoint is closed).

Figure 8.4 lists part of the code for representing closed, open and half-open intervals. For efficiency, endpoints are 64-bit floating-point numbers, but this does not threaten soundness. Because the floating-point numbers contain `-inf.0` and `+inf.0`, every real interval can be covered by at least one floating-point interval. The (unlisted) `interval` function returns a Real-Set or Empty-Set given open or closed endpoints. It ensures neither endpoint is +nan.0, returns empty-set if the endpoints are out of order or are equal but at least one is open, and forces −inf.0 and +inf.0 endpoints to be open. The real-set-intersect function intersects real sets; the unlisted real-set-join is similar, but always returns a (nonempty) Real-Set. The unlisted real-set-member? is simple enough: it returns #t when its value argument is between its set argument's endpoints, or equal to a closed endpoint.

The function real-set-singleton is also defined in Figure 8.4. Because ℝ values are represented by the type Real, which includes exact rationals such as 1/7, it cannot simply return the closed interval (Real-Set a a #t #t). Fortunately, because the floating-point numbers contain −inf.0 and +inf.0 and there are only finitely many of them, every real number that is not represented exactly by a float is between two closest floats. The fl function converts an exact rational to a floating-point number by rounding its argument to the nearest one. The logic after (define b (fl a)) determines whether b is rounded down or up, or is not rounded, and uses Racket's math/flonum library's flnext and flprev in the rounding cases to construct the smallest open floating-point interval containing a. If b is not rounded, it returns a closed interval with both endpoints b.

Testing real-set-singleton on 3/4 and 1/7, we get

```
> (real-set-singleton 3/4)
(Real-Set 0.75 0.75 #t #t)

> (real-set-singleton 1/7)
(Real-Set 0.14285714285714285 0.14285714285714288 #f #f)

> (real-set-member? 3/4 (real-set-singleton 3/4))
#t

> (real-set-member? 1/7 (real-set-singleton 1/7))
#t
```

Using the Racket's math/bigfloat library to get a 128-bit approximation of $\pi$, and using the #e number prefix to construct exact rational numbers that are smaller and larger than the smallest and largest positive floating-point numbers, we get the following intervals:

```
> (real-set-singleton (bigfloat->real pi.bf))
(Real-Set 3.141592653589793 3.1415926535897936 #f #f)

> (real-set-singleton #e1e-350)
(Real-Set 0.0 4.9406564584125e-324 #f #f)

> (real-set-singleton #e1e350)
(Real-Set 1.7976931348623157e+308 +inf.0 #f #f)
```

These are the tightest sound approximations of $\{3/4\}$, $\{1/7\}$, $\{\pi\}$, $\{10^{-350}\}$ and $\{10^{350}\}$ possible with floating-point intervals.

### 8.2.1 Infinite Binary Trees

Rectangular families of sets (Definition 7.54) are defined so that rectangles of any $\mathsf{J} \to \mathsf{A}$ have only finitely many projections that are proper subsets of $\mathsf{A}$. For example, for $\Omega := \mathsf{J} \to [0,1]$, if $\Omega' \in \mathsf{Rect}\ \Omega$, then $\mathsf{proj}\ \mathsf{j}\ \Omega' \subset [0,1]$ for only finitely many $\mathsf{j} \in \mathsf{J}$. Further, the index set $\mathsf{J}$ is part of a binary indexing scheme, so such values have a tree structure we can use to represent them. We can thus use self-similarly to represent $\Omega$ rectangles by a finite data structure: a subtree in which every projection is $[0,1]$ is represented by (the representation of) $\Omega$ itself.

We need a constructor for building binary trees recursively. The following function receives a node value $\mathsf{a}$ and two tree encodings $\mathsf{l}$ and $\mathsf{r}$, and returns a tree encoding that maps $\mathsf{j}_0$ to $\mathsf{a}$, and has $\mathsf{l}$ and $\mathsf{r}$ as the left and right subtrees.

$$\begin{aligned}
&\mathsf{tree\text{-}node} : \mathsf{A} \Rightarrow (\mathsf{J} \to \mathsf{A}) \Rightarrow (\mathsf{J} \to \mathsf{A}) \Rightarrow (\mathsf{J} \to \mathsf{A}) \\
&\mathsf{tree\text{-}node\ a\ l\ r} \ := \ \{\langle \mathsf{j}_0, \mathsf{a}\rangle\} \cup \\
&\qquad \{\langle \mathsf{left\ j}, \mathsf{a}\rangle \mid \langle \mathsf{j}, \mathsf{a}\rangle \in \mathsf{l}\} \cup \\
&\qquad \{\langle \mathsf{right\ j}, \mathsf{a}\rangle \mid \langle \mathsf{j}, \mathsf{a}\rangle \in \mathsf{r}\}
\end{aligned} \tag{8.1}$$

From $\mathsf{tree\text{-}node}$, we define a function to construct instances of $\mathsf{Rect}\ (\mathsf{J} \to \mathsf{A})$ from a projection, and left and right subtree rectangles. It is essentially a trinary cartesian product.

$$\begin{aligned}
&\mathsf{tree\text{-}prod} : \mathsf{Rect\ A} \Rightarrow \mathsf{Rect}\ (\mathsf{J} \to \mathsf{A}) \Rightarrow \mathsf{Rect}\ (\mathsf{J} \to \mathsf{A}) \Rightarrow \mathsf{Rect}\ (\mathsf{J} \to \mathsf{A}) \\
&\mathsf{tree\text{-}prod\ A\ L\ R} \ := \ \{\mathsf{tree\text{-}cons\ a\ l\ r} \mid \mathsf{a} \in \mathsf{A}, \mathsf{l} \in \mathsf{L}, \mathsf{r} \in \mathsf{R}\}
\end{aligned} \tag{8.2}$$

Any $\mathsf{Rect}\ (\mathsf{J} \to \mathsf{A})$ can be constructed from $\mathsf{J} \to \mathsf{A}$ itself, finitely many projections, and finitely many applications of tree-prod. For example,

$$\mathsf{tree\text{-}prod}\ [0, \tfrac{1}{2}]\ (\mathsf{tree\text{-}prod}\ [\tfrac{1}{2}, 1]\ \Omega\ \Omega)\ \Omega \tag{8.3}$$

constructs an instance $\Omega' \in \mathsf{Rect}\ \Omega$ for which $\mathsf{proj}\ \mathsf{j}_0\ \Omega' = [0, \frac{1}{2}]$ and $\mathsf{proj}\ (\mathsf{left}\ \mathsf{j}_0)\ \Omega' = [\frac{1}{2}, 1]$, and all other projections are $[0, 1]$.

In Figure 8.5, tree-prod is represented by a data type `Omega-Node`, and $\Omega$ is represented by the singleton value `univ-omega-set`. Representations of Rect $\Omega$ instances are constructed as in (8.3); for example

```
(define omega-rect
  (Omega-Node (Real-Set 0.0 0.5 #t #t)
              (Omega-Node (Real-Set 0.5 1.0 #t #t)
                          univ-omega-set
                          univ-omega-set)
              univ-omega-set))
```

Functions `omega-set-project` and `omega-set-unproject` respectively implement proj and unproj for $\Omega$ rectangles; for example

```
> (omega-set-project j0 omega-rect)
(Real-Set 0.0 0.5 #t #t)

> (omega-set-project (right j0) omega-rect)
(Real-Set 0.0 1.0 #t #t)

> (omega-set-unproject (left j0) omega-rect (Real-Set 0.0 0.75 #t #t))
(Omega-Node (Real-Set 0.0 0.5 #t #t)
            (Omega-Node (Real-Set 0.5 0.75 #t #t)
                        univ-omega-set
                        univ-omega-set)
            univ-omega-set)
```

Figure 8.6 lists an implementation of values in $\Omega$, which are infinite binary trees, as a lazy data structure. The `Omega-Val` data type represents the tree-node function. Instances of `(Promise A)` are lazy values: they are created using special syntax `(delay a)` where `a` is of type `A`, and are computed and cached using the function `force` [XXX: cite?]. Thus, an

```
;; Binary indexing scheme
(define-type J (Listof Boolean))
(define j0 null)

(: left (J -> J))
(define (left j) (cons #t j))

(: right (J -> J))
(define (right j) (cons #f j))


;; Type representing Omega
(define-singleton-type Univ-Omega-Set univ-omega-set)

;; Type representing a subrectangle of Omega
(struct: Omega-Node ([axis : Real-Set] [left : Omega-Set] [right : Omega-Set])
  #:transparent)

(define-type Omega-Set (U Univ-Omega-Set Omega-Node))
(define-predicate omega-set? Omega-Set)

(: omega-set-project (J Omega-Set -> Real-Set))
;; Returns Z's axis at index j
(define (omega-set-project j Z)
  (let loop ([j  (reverse j)] [Z Z])
    (match Z
      [(? univ-omega-set?)  unit-interval]
      [(Omega-Node A L R)
       (cond [(null? j)  A]
             [(first j)  (loop (rest j) L)]
             [else       (loop (rest j) R)])])))

;; Functionally equivalent to univ-omega-set, but has fields for recursion
(define univ-omega-node
  (Omega-Node unit-interval univ-omega-set univ-omega-set))

(: omega-set-unproject (J Omega-Set Real-Set -> (U Empty-Set Omega-Set)))
;; Functionally updates Z's axis at index j by intersecting it with B
(define (omega-set-unproject j Z B)
  (let loop ([j  (reverse j)] [Z Z])
    (match Z
      [(? univ-omega-set?)  (loop j univ-omega-node)]
      [(Omega-Node A L R)
       (cond [(null? j)  (omega-set-node (real-set-intersect A B) L R)]
             [(first j)  (omega-set-node A (loop (rest j) L) R)]
             [else       (omega-set-node A L (loop (rest j) R))])])))
```

Figure 8.5: Part of a Typed Racket representation of Rect $\Omega$, as *finite* binary trees.

```
(struct: Omega-Val ([value : (Promise Real)]
                    [left  : (Promise Omega-Val)]
                    [right : (Promise Omega-Val)])
  #:transparent)

(: omega-set-member? (Omega-Val Omega-Set -> Boolean))
(define (omega-set-member? z Z)
  (match* (z Z)
    [(z (? univ-omega-set?))  #t]
    [((Omega-Val a l r) (Omega-Node A L R))
     (and (real-set-member? (force a) A)
          (omega-set-member? (force l) L)
          (omega-set-member? (force r) R))]))

(: omega-set-sample (Omega-Set -> Omega-Val))
(define (omega-set-sample Z)
  (match Z
    [(? univ-omega-set?)
     (omega-set-sample univ-omega-node)]
    [(Omega-Node A L R)
     (Omega-Val (delay (real-set-sample A))
                (delay (omega-set-sample L))
                (delay (omega-set-sample R)))]))
```

Figure 8.6: A Typed Racket representation of values $\omega \in \Omega$, as lazy binary trees.

Omega-Val's infinite left and right subtrees are represented by (Promise Omega-Val), which are promises to produce subtrees.

For lazy trees, it is easy to write recursive functions that may not terminate. The two listed functions omega-set-member? and omega-set-sample always terminate, however: both recur on the structure of Omega-Set, and are thus well-founded.

Representations of branch traces and rectangles are similar to Omega-Val and Omega-Set.

### 8.2.2 Disjoint Bottom and Top Unions

The set representations up to this point are the minimum necessary for a language with real numbers and lists. Whether more complicated representations are necessary depends on the presence of certain language features and primitives.

Suppose, for example, that we extend $[\![\cdot]\!]^{\Downarrow}_{\mathsf{a^*}}$ by this rule:

$$[\![\mathsf{strict\text{-}if}\ e_1\ e_2\ e_3]\!]^{\Downarrow}_{\mathsf{a^*}} \ :=\ \mathsf{ifte}_{\mathsf{a^*}}\ [\![e_1]\!]^{\Downarrow}_{\mathsf{a^*}}\ [\![e_2]\!]^{\Downarrow}_{\mathsf{a^*}}\ [\![e_3]\!]^{\Downarrow}_{\mathsf{a^*}} \tag{8.4}$$

Recall that if is interpreted using $\mathsf{ifte}_{\mathsf{a^*}}$ and $\mathsf{lazy}_{\mathsf{a^*}}$, so that programs with if-guarded recursion have a well-defined interpretation, and preimage computation always terminates. Compare these two expressions, in which $e$ is any test expression that may evaluate to true or false:

$$\begin{gathered} \mathsf{if}\ e\ \langle\rangle\ \mathsf{random} \\ \mathsf{strict\text{-}if}\ e\ \langle\rangle\ \mathsf{random} \end{gathered} \tag{8.5}$$

The if expression is interpreted as an application of $\mathsf{ifte}^{\Downarrow}_{\mathsf{pre^*}}$, whose approximation (Figure 7.12c) takes at most one branch. The image of the program domain under the if expression is therefore $\{\langle\rangle\}$ or $[0,1]$, or is not computed at all. In contrast, strict-if is interpreted as an application of $\mathsf{ifte}_{\mathsf{pre^*}}$, whose approximation (Figure 7.12b) takes *both* branches. The image of the program domain under the strict-if expression is therefore $\{\langle\rangle\} \uplus [0,1]$.

In fact, in the absence of a form or a primitive such as strict-if, neither image nor preimage computation attempts to join sets of different types. The implementation of $(\vee)$ may return anything in these circumstances (though it is safest to raise an error).

We have found strict-if useful for a few things.

One is defining strict versions of boolean operators, which are faster than their lazy (i.e. short-cutting) counterparts:

$$\begin{aligned} \mathsf{a\ and\ b}\ &:\equiv\ \mathsf{if\ a\ b\ false} \\ \mathsf{a\ and^*\ b}\ &:\equiv\ \mathsf{strict\text{-}if\ a\ b\ false} \end{aligned} \tag{8.6}$$

Here, ":≡" denotes defining special syntax rather than defining a function. (Otherwise, both conjunctions would be strict.)

Another is to assert that prop? x for some predicate prop? and value x:

$$\mathsf{assert\ prop?\ x}\ :\equiv\ \mathsf{strict\text{-}if\ (prop?\ x)\ x\ fail} \tag{8.7}$$

Here, $\mathsf{fail}$ is interpreted as a computation that always returns $\varnothing$ for images and preimages. This expression thus restricts the program domain to the set of values for which $\mathsf{prop?}\ \mathsf{x}$ evaluates to $\mathsf{true}$ regardless of branch traces, which cannot be done using $\mathsf{if}$.

Another is pasting together piecewise monotone functions (Section 8.3.4).

With $\mathsf{strict\text{-}if}$, there must be a type to represent disjoint unions such as $\{\langle\rangle\} \uplus [0, 1]$. One that is relatively easy to use is

```
(struct: Bot-Union-Set ([hash : (HashTable Symbol Nonempty-Set)])
  #:transparent)
```

which maps symbols to instances of associated set types. This type also allows user data types to be represented easily: every structure definition is assigned a symbol, which is mapped to product sets within instances of `Bot-Union-Set`. Intersections and joins are done by looping over symbols.

Suppose we add a primitive $\mathsf{real?}$ that returns $\mathsf{true}$ when its argument is a real number and $\mathsf{false}$ otherwise. As a preimage computation, it could be defined as

$$\begin{array}{rcll}
\mathsf{real?_{pre}}\ \mathsf{A} & :\equiv & \mathsf{case}\ \ \langle \mathsf{A} \cap \mathbb{R}, \mathsf{A} \backslash \mathbb{R} \rangle & \\
& & \langle \varnothing, \varnothing \rangle \ \longrightarrow\ \langle \varnothing, \lambda\, \mathsf{B}.\, \varnothing \rangle & \\
& & \langle \mathsf{A_t}, \varnothing \rangle \ \longrightarrow\ \mathsf{const_{pre}}\ \mathsf{true}\ \mathsf{A_t} & \\
& & \langle \varnothing, \mathsf{A_f} \rangle \ \longrightarrow\ \mathsf{const_{pre}}\ \mathsf{false}\ \mathsf{A_f} & \\
& & \langle \mathsf{A_t}, \mathsf{A_f} \rangle \ \longrightarrow\ \langle \mathsf{Bool}, \lambda\, \mathsf{B}.\, (\mathsf{if}\ (\mathsf{true} \in \mathsf{B})\ \mathsf{A_t}\ \varnothing) \cup (\mathsf{if}\ (\mathsf{false} \in \mathsf{B})\ \mathsf{A_f}\ \varnothing) \rangle &
\end{array} \tag{8.8}$$

We potentially have a problem implementing this: we do not have relative complement for implementing $\mathsf{A} \backslash \mathbb{R}$. If we have a limited number of data types, however, we can do this:

$$\mathsf{A} \backslash \mathbb{R} \ = \ \mathsf{A} \cap (\mathsf{X_1} \cup \mathsf{X_2} \cup \ldots \cup \mathsf{X_n}) \tag{8.9}$$

where $\mathbb{R}$ does not appear in the union $\mathsf{X_1} \cup \mathsf{X_2} \cup \ldots \cup \mathsf{X_n}$, which can be represented by a `Bot-Union-Set`. Unfortunately, computing this in the presence of user data types can be very inefficient and requires some static analysis to determine which are used in a particular program.

Instead, we might represent $\mathsf{X_1} \cup \mathsf{X_2} \cup \ldots \cup \mathsf{X_n}$ using a **top union**:

```
(struct: Top-Union-Set ([hash : (HashTable Symbol Nonuniversal-Set)])
  #:transparent)
```

where `Nonuniversal-Set` is a new subtype of `Set` that does not include `Univ-Omega-Set`, `Univ-Trace-Set` nor other universal sets. For example, a `Top-Union-Set` that maps `'real` to `empty-set` would represent the set of all values except the reals.

With top unions, it is easy to abstract $\mathsf{real?}_{\mathsf{pre}}$ to arbitrary predicates:

$$\begin{array}{l} \mathsf{predicate}_{\mathsf{pre}}\ \mathsf{X_t}\ \mathsf{X_f}\ \mathsf{A}\ := \\ \quad \mathsf{case}\ \ \langle \mathsf{A} \cap \mathsf{X_t}, \mathsf{A} \cap \mathsf{X_f} \rangle \\ \quad\ \ \begin{array}{lcl} \langle \varnothing, \varnothing \rangle & \longrightarrow & \langle \varnothing, \lambda \mathsf{B}.\, \varnothing \rangle \\ \langle \mathsf{A_t}, \varnothing \rangle & \longrightarrow & \mathsf{const}_{\mathsf{pre}}\ \mathsf{true}\ \mathsf{A_t} \\ \langle \varnothing, \mathsf{A_f} \rangle & \longrightarrow & \mathsf{const}_{\mathsf{pre}}\ \mathsf{false}\ \mathsf{A_f} \\ \langle \mathsf{A_t}, \mathsf{A_f} \rangle & \longrightarrow & \langle \mathsf{Bool}, \lambda \mathsf{B}.\, (\mathsf{if}\ (\mathsf{true} \in \mathsf{B})\ \mathsf{A_t}\ \varnothing) \cup (\mathsf{if}\ (\mathsf{false} \in \mathsf{B})\ \mathsf{A_f}\ \varnothing) \rangle \end{array} \end{array} \tag{8.10}$$

Thus, $\mathsf{real?}_{\mathsf{pre}} \equiv \mathsf{predicate}_{\mathsf{pre}}\ \mathbb{R}\ (\top \backslash \mathbb{R})$. In the implementation, $\top \backslash \mathbb{R}$ would be represented by an instance of `Top-Union-Set`.

### 8.2.3 Testing

Dr. Bayes's rectangular sets include sets of booleans, $\{\langle\rangle\}$, pairs, real sets, tagged structures, and bottom and top disjoint unions. Real sets are represented by finite, sorted lists of nonadjacent intervals, which complicates the set library further. We plan to add set representations for other basic data types, such as symbols and strings.

Even without representing sets of symbols and strings, the set library is the largest part of Dr. Bayes's codebase: at just over 3000 lines of code, it comprises half.

Not only is the set library large and complicated, but errors in it are difficult to diagnose. By analogy, if Dr. Bayes is Java, then the bottom* and preimage* arrows are bytecode, and rectangular set operations are machine code. Blaming an error from Dr. Bayes's output on the set library is like blaming an error from Java program output on an error in the CPU's microprogram for an opcode. Worse, because Dr. Bayes outputs stochastic approximations, we are lucky if a noncatastrophic error in the set library is detectable.

Fortunately, unlike CPU microcode, rectangular set operations are correct if and only if they obey a small collection of laws.

The first part of the collection of laws regards sets not as boxes of values, but as values themselves in a bounded lattice. There are eight algebraic laws that define a bounded lattice. In terms of $(\cap)$, $(\vee)$, $\varnothing$ and $\bot$, the algebraic laws are

$$\begin{aligned}
&\varnothing \vee \mathsf{A} = \mathsf{A} && (\vee) \text{ identity} \\
&\top \cap \mathsf{A} = \mathsf{A} && (\cap) \text{ identity} \\
&\mathsf{A} \vee \mathsf{B} = \mathsf{B} \vee \mathsf{A} && (\vee) \text{ commutativity} \\
&\mathsf{A} \cap \mathsf{B} = \mathsf{B} \cap \mathsf{A} && (\cap) \text{ commutativity} \\
&(\mathsf{A} \vee \mathsf{B}) \vee \mathsf{C} = \mathsf{A} \vee (\mathsf{B} \vee \mathsf{C}) && (\vee) \text{ associativity} \\
&(\mathsf{A} \cap \mathsf{B}) \cap \mathsf{C} = \mathsf{A} \cap (\mathsf{B} \cap \mathsf{C}) && (\cap) \text{ associativity} \\
&\mathsf{A} \vee (\mathsf{A} \cap \mathsf{B}) = \mathsf{A} && (\vee)\text{-}(\cap) \text{ absorption} \\
&\mathsf{A} \cap (\mathsf{A} \vee \mathsf{B}) = \mathsf{A} && (\cap)\text{-}(\vee) \text{ absorption}
\end{aligned} \tag{8.11}$$

If these laws hold in the implementation, then at an abstract level in which we do not consider the contents of the sets, the implementation is correct.

But we must consider their contents, because we will be sampling within them, and we will be testing membership to determine whether the samples lie inside a preimage set. For our lattice, membership in its elements is characterized by these two laws:

$$\begin{aligned}
&\mathsf{x} \in \mathsf{A} \text{ or } \mathsf{x} \in \mathsf{B} \implies \mathsf{x} \in (\mathsf{A} \vee \mathsf{B}) && (\vee) \text{ membership} \\
&\mathsf{x} \in \mathsf{A} \text{ and } \mathsf{x} \in \mathsf{B} \iff \mathsf{x} \in (\mathsf{A} \cap \mathsf{B}) && (\cap) \text{ membership}
\end{aligned} \tag{8.12}$$

These are taken from the definitions of $(\cup)$ and $(\cap)$, but the first has $(\iff)$ replaced by $(\implies)$ because $(\vee)$ overapproximates (i.e. if $\mathsf{x} \in (\mathsf{A} \vee \mathsf{B})$, it may be in neither $\mathsf{A}$ nor $\mathsf{B}$).

If the preceeding 10 laws hold, the implementation is correct.

The first eight laws refer to $(=)$, which we have not discussed the implementation of yet. The set representations given in this section can easily be made canonical, so that equality can be decided structurally. By default, Racket's `equal?` primitive decides equality structurally for types with the `#:transparent` property, as Haskell's `(==)` primitive does by default for types in the `Eq` typeclass.

Dr. Bayes's set representations are currently canonical, but may not be in the future: the only equality requirement is that $\mathsf{A} = \varnothing$ be decidable. (Hopefully it is also efficient.) So to decide equality nonstructurally, we implement $(\subseteq)$ as `subseteq?` and use Lemma 4.1:

$$\mathsf{A} = \mathsf{B} \iff \mathsf{A} \subseteq \mathsf{B} \text{ and } \mathsf{B} \subseteq \mathsf{A} \qquad (=) \text{ extensionality} \tag{8.13}$$

Of course, we must now test `subseteq?` to ensure it has the properties of $(\subseteq)$. The only essential property is derived from its definition, from Axiom 1 in Chapter 3:

$$\mathsf{A} \subseteq \mathsf{B} \iff \mathsf{x} \in \mathsf{A} \implies \mathsf{x} \in \mathsf{B} \qquad (\subseteq) \text{ definition} \tag{8.14}$$

If the preceeding 12 laws hold, the implementation is correct. For canonical sets, $(=)$ extensionality is testable; otherwise we use it to define set equality (i.e. it holds by definition).

The testing regime is this: some large number of times,

1. Randomly generate $\mathsf{A}$, $\mathsf{B}$ and $\mathsf{C}$.
2. Randomly generate $\mathsf{x} \in \mathsf{A}$ and $\mathsf{y} \in \mathsf{B}$.
3. Evaluate the preceeding 12 laws.

The number of iterations for a typical testing run is 100,000, for which the current implementation takes about a minute on current hardware.

If step 2 randomly generated just $\mathsf{x} \in \top$, then $\mathsf{x} \in \mathsf{A}$ would be rare, and $\mathsf{x} \in \mathsf{A} \implies \mathsf{x} \in \mathsf{B}$ would too often be equivalent to $\mathsf{false} \implies \mathsf{x} \in \mathsf{B}$, which is always $\mathsf{true}$. Of course, we cannot always test with $\mathsf{x} \in \mathsf{A} = \mathsf{true}$, so for more complete coverage we also generate $\mathsf{y} \in \mathsf{B}$ and test $\mathsf{y} \in \mathsf{A} \implies \mathsf{y} \in \mathsf{B}$. We ensure boundary conditions, such as intersections and joins between two barely overlapping or adjacent intervals, happen often enough by choosing interval endpoints from $\{-\infty, -4, -3, -2, -1, 0, 1, 2, 3, 4, +\infty\}$. We choose members of intervals from a similar small set that includes those endpoints, except the infinities.

To be even more certain that the implementation is correct, we additionally test an alternative lattice characterization: that the elements have an associated partial order in which every two elements has a meet and a join. In this case, the partial order is $(\subseteq)$. To be a partial order, it should have these properties:

$$\begin{aligned}
&\mathsf{A} \subseteq \mathsf{A} && (\subseteq) \text{ reflexivity} \\
&\mathsf{A} \subseteq \mathsf{B} \text{ and } \mathsf{B} \subseteq \mathsf{A} \implies \mathsf{A} = \mathsf{B} && (\subseteq) \text{ antisymmetry} \\
&\mathsf{A} \subseteq \mathsf{B} \text{ and } \mathsf{B} \subseteq \mathsf{C} \implies \mathsf{A} \subseteq \mathsf{C} && (\subseteq) \text{ transitivity}
\end{aligned} \tag{8.15}$$

For canonical sets, $(\subseteq)$ antisymmetry is testable; otherwise it holds by definition.

The partial order is related to the lattice operators by the following properties, of which the first two provide alternative definitions for $(\subseteq)$ in terms of $(\vee)$ or $(\cap)$, or vice-versa:

$$\begin{aligned}
&\mathsf{B} = \mathsf{A} \vee \mathsf{B} \iff \mathsf{A} \subseteq \mathsf{B} && (\vee)\text{-}(\subseteq) \text{ definition} \\
&\mathsf{A} = \mathsf{A} \cap \mathsf{B} \iff \mathsf{A} \subseteq \mathsf{B} && (\cap)\text{-}(\subseteq) \text{ definition} \\
&\mathsf{A} \subseteq \mathsf{A} \vee \mathsf{B} && (\vee) \text{ increasing} \\
&\mathsf{A} \cap \mathsf{B} \subseteq \mathsf{A} && (\cap) \text{ decreasing} \\
&\mathsf{A}_1 \subseteq \mathsf{A}_2 \text{ and } \mathsf{B}_1 \subseteq \mathsf{B}_2 \implies \mathsf{A}_1 \vee \mathsf{B}_1 \subseteq \mathsf{A}_2 \vee \mathsf{B}_2 && (\vee) \text{ monotone} \\
&\mathsf{A}_1 \subseteq \mathsf{A}_2 \text{ and } \mathsf{B}_1 \subseteq \mathsf{B}_2 \implies \mathsf{A}_1 \cap \mathsf{B}_1 \subseteq \mathsf{A}_2 \cap \mathsf{B}_2 && (\cap) \text{ monotone}
\end{aligned} \tag{8.16}$$

For noncanonical sets, the first two properties are equivalent to the middle two.

Errors introduced by changing the set library are caught quickly, usually within a few hundred iterations. We are quite certain of the correctness of our current implementation of rectangular sets and set operations, having tested the preceeding 21 lattice and membership properties on millions of random inputs.

## 8.3 Preimages Under Real Functions

Chapter 7 leaves computing approximate preimages under arithmetic and other primitives up to implementors. In this section, we formalize a unified approach to doing so for one- and two-argument real functions, and give examples from Dr. Bayes's implementation.

The general idea is to compute preimages by computing images of inverses. While how to do so seems obvious for certain kinds of one-argument functions, for two-argument functions it is not. Generalizing the computation of preimages under two-argument functions requires a theory of per-axis function inversion, which we have not been able to find in the literature.

We start with one-argument functions for simplicity, and extend to two-argument functions by regarding a one-argument function and its inverse as a cyclic group of order 2, and generalizing to similar groups of order 3. The resulting theory should generalize naturally to functions with any number of arguments, but we leave it for future work.

Working with intervals algorithmically is easier if we have notation in which the kind of interval is not baked into the syntax.

**Definition 8.1** (interval). $[\![\mathsf{a}_1, \mathsf{a}_2, \alpha_1, \alpha_2]\!]$ *denotes an interval, where* $\mathsf{a}_1, \mathsf{a}_2 \in \overline{\mathbb{R}}$ *are extended real endpoints, and* $\alpha_1, \alpha_2 \in \mathsf{Bool}$ *determine whether* $\mathsf{a}_1$ *and* $\mathsf{a}_2$ *are contained in the interval.*

Some intervals, using $[\![\cdot, \cdot, \cdot, \cdot]\!]$ notation:

$$\begin{aligned} [\![0, 1, \mathsf{true}, \mathsf{false}]\!] &= [0, 1) \\ [\![-\infty, 0, \mathsf{false}, \mathsf{true}]\!] &= (-\infty, 0] \\ [\![-\infty, +\infty, \mathsf{false}, \mathsf{false}]\!] &= (-\infty, +\infty) = \mathbb{R} \\ [\![-\infty, +\infty, \mathsf{true}, \mathsf{true}]\!] &= [-\infty, +\infty] = \overline{\mathbb{R}} \end{aligned} \tag{8.17}$$

### 8.3.1 Invertible Primitives

We consider only total, strictly monotone functions on subsets of $\mathbb{R}$. Further on, we recover more generality by using language conditionals to implement *piecewise* monotone functions.

One reason we consider only strictly monotone functions is that they are easy to invert. Recall that a function is invertible (bijective) if and only if it is injective (one-to-one) and surjective (onto).

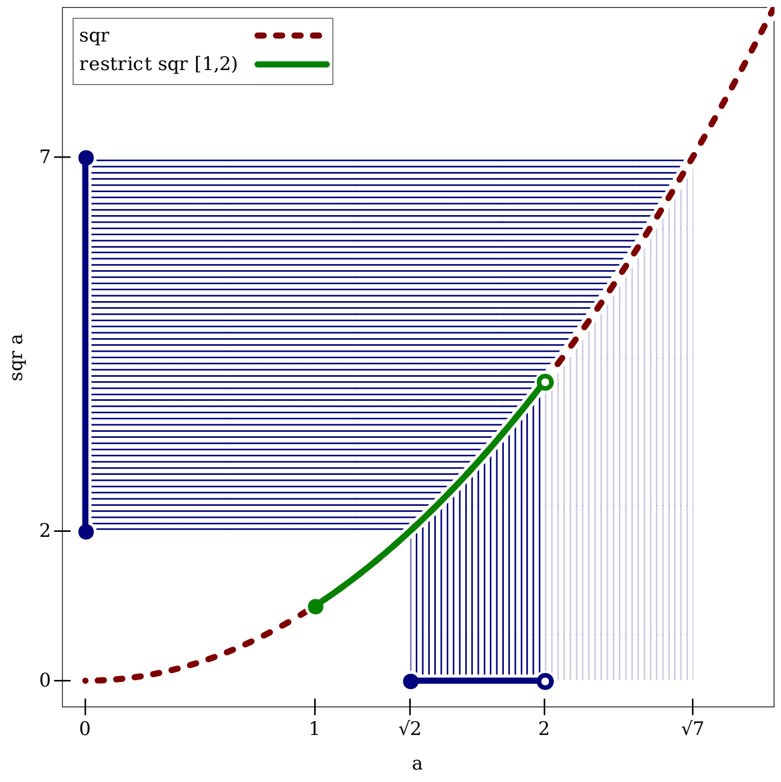


Figure 8.7: Computing the preimage of the interval $[2,7]$ under $\mathsf{sqr}$ restricted to $[1,2)$, by computing roots and intersecting with $[1,2)$.

**Lemma 8.2** (strictly monotone, surjective implies invertible, continuous)**.** *If* $\mathsf{g} : \mathsf{A} \to \mathsf{B}$ *is strictly montone,* $\mathsf{g}$ *is injective. If* $\mathsf{g}$ *is additionally surjective,* $\mathsf{g}$ *and its inverse are continuous.*

Preimages under invertible functions can be computed using their inverses. Because we are deriving preimage arrow computations, we are primarily interested in computing preimages under restricted functions.

**Lemma 8.3** (preimages from inverse images)**.** *If* $\mathsf{A}' \subseteq \mathsf{A}$*,* $\mathsf{B}' \subseteq \mathsf{B}$*, and* $\mathsf{g} : \mathsf{A} \to \mathsf{B}$ *has inverse* $\mathsf{g}^{-1} : \mathsf{B} \to \mathsf{A}$*, then* $\mathsf{preimage}\ (\mathsf{restrict}\ \mathsf{f}\ \mathsf{A}')\ \mathsf{B}' \ = \ \mathsf{A}' \cap \mathsf{image}\ \mathsf{g}^{-1}\ \mathsf{B}'$.

These facts suggest that we can compute images (or preimages) of intervals under any strictly monotone, surjective $\mathsf{g}$ by applying $\mathsf{g}$ (or its inverse) to interval endpoints to yield an interval, as in Figure 8.7. This is evident for endpoints in $\mathsf{A}$. Limit endpoints like $+\infty$ require a larger $\overline{\mathsf{g}}$ defined on a compact superset of $\mathsf{A}$.

**Theorem 8.4** (images of intervals by endpoints)**.** *Let* $\overline{\mathsf{A}}$ *and* $\overline{\mathsf{B}}$ *be compact subsets of* $\overline{\mathbb{R}}$*,* $\overline{\mathsf{g}} : \overline{\mathsf{A}} \to \overline{\mathsf{B}}$ *be strictly monotone and surjective, and* $\mathsf{g}$ *be the restriction of* $\overline{\mathsf{g}}$ *to some* $\mathsf{A} \subseteq \overline{\mathsf{A}}$*. For all nonempty* $[\![\mathsf{a}_1, \mathsf{a}_2, \alpha_1, \alpha_2]\!] \subseteq \mathsf{A}$*,*

- *If* $\overline{\mathsf{g}}$ *is increasing,* $\mathsf{image}\ \mathsf{g}\ [\![\mathsf{a}_1, \mathsf{a}_2, \alpha_1, \alpha_2]\!] = [\![\overline{\mathsf{g}}\ \mathsf{a}_1, \overline{\mathsf{g}}\ \mathsf{a}_2, \alpha_1, \alpha_2]\!]$.
- *If* $\overline{\mathsf{g}}$ *is decreasing,* $\mathsf{image}\ \mathsf{g}\ [\![\mathsf{a}_1, \mathsf{a}_2, \alpha_1, \alpha_2]\!] = [\![\overline{\mathsf{g}}\ \mathsf{a}_2, \overline{\mathsf{g}}\ \mathsf{a}_1, \alpha_2, \alpha_1]\!]$.

*Proof.* Because $\overline{\mathsf{A}}$ is compact and totally ordered, every subset of $\overline{\mathsf{A}}$ has a lower and an upper bound in $\overline{\mathsf{A}}$. Therefore, the endpoints of every interval subset of $\mathsf{A}$ are in $\overline{\mathsf{A}}$.

Let $(\mathsf{a}_1, \mathsf{a}_2] \subseteq \mathsf{A}$. Suppose $\overline{\mathsf{g}}$ is strictly increasing; thus $\mathsf{a}_1 < \mathsf{a} \leq \mathsf{a}_2$ if and only if $\overline{\mathsf{g}}\ \mathsf{a}_1 < \overline{\mathsf{g}}\ \mathsf{a} \leq \overline{\mathsf{g}}\ \mathsf{a}_2$, so $\mathsf{image}\ \mathsf{g}\ (\mathsf{a}_1, \mathsf{a}_2] = \mathsf{image}\ \overline{\mathsf{g}}\ (\mathsf{a}_1, \mathsf{a}_2] = (\overline{\mathsf{g}}\ \mathsf{a}_1, \overline{\mathsf{g}}\ \mathsf{a}_2]$. The remaining cases are similar. $\square$

To use Theorem 8.4 to compute preimages under $\mathsf{g}$ by computing images under its inverse $\mathsf{g}^{-1}$, we must know if $\mathsf{g}^{-1}$ is increasing or decreasing. The following lemma can help.

**Lemma 8.5** (inverse direction)**.** *If* $\mathsf{g} : \mathsf{A} \to \mathsf{B}$ *is strictly monotone and surjective with inverse* $\mathsf{g}^{-1} : \mathsf{B} \to \mathsf{A}$*, then* $\mathsf{g}$ *is increasing if and only if* $\mathsf{g}^{-1}$ *is increasing.*

**Example 8.6** (nonnegative square)**.** The extension of $\mathsf{sqr}^+ : [0, +\infty) \to \mathbb{R}$, where $\mathsf{sqr}^+\ \mathsf{a} := \mathsf{a} \cdot \mathsf{a}$, to the compact superdomain $[0, +\infty]$ is

$$\begin{aligned} &\overline{\mathsf{sqr}^+} : [0, +\infty] \to [0, +\infty] \\ &\overline{\mathsf{sqr}^+}\ \mathsf{a} := \lim_{\mathsf{a}' \to \mathsf{a}} \mathsf{sqr}^+\ \mathsf{a}' \;=\; \mathsf{if}\ (\mathsf{a} = +\infty)\ +\infty\ (\mathsf{sqr}^+\ \mathsf{a}) \end{aligned} \tag{8.18}$$

(With respect to $\mathbb{R}$'s standard topology, which is first-countable, $\mathsf{sqr}^+$ is continuous and thus limit-preserving.) The extension of its inverse $\mathsf{sqrt}^+$ is $\overline{\mathsf{sqrt}^+} : [0, +\infty] \to [0, +\infty]$, defined similarly, which by Lemma 8.5 is also strictly increasing. Thus,

$$\begin{aligned}
\mathsf{image}\ \mathsf{sqr}^{+}\ [5,+\infty) &= [\overline{\mathsf{sqr}^{+}}\ 5, \overline{\mathsf{sqr}^{+}}\ +\infty) \\
&= [25,+\infty) \\
\mathsf{preimage}\ (\mathsf{restrict}\ \mathsf{sqr}^{+}\ [1,2))\ [2,7] &= [1,2) \cap \mathsf{image}\ \mathsf{sqrt}^{+}\ [2,7] \\
&= [1,2) \cap [\overline{\mathsf{sqrt}^{+}}\ 2, \overline{\mathsf{sqrt}^{+}}\ 7] \\
&= [1,2) \cap [\sqrt{2}, \sqrt{7}] \\
&= [\sqrt{2}, 2)
\end{aligned} \tag{8.19}$$

by Theorem 8.4 and Lemma 8.3. ♢

### 8.3.2 Two-Argument Primitives

We do not expect to be able to compute preimages under $\mathbb{R} \times \mathbb{R} \rightharpoonup \mathbb{R}$ primitives by simply inverting them. Two-argument invertible real functions are difficult to define and are usually pathological. Instead, we compute approximate preimages only, using inverses with respect to one argument (with the other held constant).

**Definition 8.7** (axial inverse). *Let* $\mathsf{g_c} : \mathsf{A} \times \mathsf{B} \to \mathsf{C}$. *Functions* $\mathsf{g_a} : \mathsf{B} \times \mathsf{C} \to \mathsf{A}$ *and* $\mathsf{g_b} : \mathsf{C} \times \mathsf{A} \to \mathsf{B}$ *defined so that*

$$\mathsf{g_c}\ \langle \mathsf{a}, \mathsf{b} \rangle = \mathsf{c} \iff \mathsf{g_a}\ \langle \mathsf{b}, \mathsf{c} \rangle = \mathsf{a} \iff \mathsf{g_b}\ \langle \mathsf{c}, \mathsf{a} \rangle = \mathsf{b} \tag{8.20}$$

*are* ***axial inverses*** *with respect to* $\mathsf{g_c}$*'s first and second arguments.*

We call $\mathsf{g_c}$ ***axis-invertible*** or ***trijective*** when it has axial inverses $\mathsf{g_a}$ and $\mathsf{g_b}$. We call $\mathsf{g_a}$ the ***first axial inverse*** of $\mathsf{g_c}$ because it is the inverse of $\mathsf{g_c}$ along the first axis: $\mathsf{g_a}$ with only $\mathsf{c}$ varying, or $\lambda\, \mathsf{c} \in \mathsf{C}.\, \mathsf{g_a}\ \langle \mathsf{b}, \mathsf{c} \rangle$, is the inverse of $\mathsf{g_c}$ with only $\mathsf{a}$ varying, or $\lambda\, \mathsf{a} \in \mathsf{A}.\, \mathsf{g_c}\ \langle \mathsf{a}, \mathsf{b} \rangle$. Similarly, $\mathsf{g_b}$ is the ***second axial inverse***.

**Example 8.8.** Let $\mathsf{add_c} : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$, $\mathsf{add_c}\ \langle \mathsf{a}, \mathsf{b} \rangle := \mathsf{a} + \mathsf{b}$. Its axial inverses are $\mathsf{add_a}\ \langle \mathsf{b}, \mathsf{c} \rangle := \mathsf{c} - \mathsf{b}$ and $\mathsf{add_b}\ \langle \mathsf{c}, \mathsf{a} \rangle := \mathsf{c} - \mathsf{a}$. ♢

We have chosen the axial inverse function types carefully: they are the only types for which $\mathsf{g_c}$, $\mathsf{g_a}$ and $\mathsf{g_b}$ form a cyclic group.

**Lemma 8.9** (axial inverse cyclic group). *The following statements are equivalent.*

- $\mathsf{g_c}$ *has axial inverses* $\mathsf{g_a}$ *and* $\mathsf{g_b}$*.*
- $\mathsf{g_a}$ *has axial inverses* $\mathsf{g_b}$ *and* $\mathsf{g_c}$*.*
- $\mathsf{g_b}$ *has axial inverses* $\mathsf{g_c}$ *and* $\mathsf{g_a}$*.*

*Equivalently, every axis-invertible function generates a cyclic group of order 3 by inversion in the first axis.*

This fact is analogous to how mutual inverses $\mathsf{g}$ and $\mathsf{g}^{-1}$ form a cyclic group of order 2 generated by inversion. Similar to using mutual inversion to compute images and preimages under both $\mathsf{sqr}^+$ and $\mathsf{sqrt}^+$, Lemma 8.9 allows computing preimages under two-argument functions related by axial inversion.

**Example 8.10.** Define $\mathsf{sub_c} : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ by $\mathsf{sub_c}\ \langle \mathsf{a}, \mathsf{b} \rangle := \mathsf{a} - \mathsf{b}$. Because $\mathsf{sub_c} = \mathsf{add_b}$, $\mathsf{sub_a} = \mathsf{add_c}$ and $\mathsf{sub_b} = \mathsf{add_a}$. $\diamondsuit$

Unlike inverses, axial inverses do not provide a direct way to compute exact preimages. Instead, they provide a way to compute a preimage's smallest rectangular bounding set.

**Theorem 8.11** (preimage bounds from axial inverse images). *Let* $\mathsf{A}' \subseteq \mathsf{A}$, $\mathsf{B}' \subseteq \mathsf{B}$, $\mathsf{C}' \subseteq \mathsf{C}$, *and* $\mathsf{g_c} : \mathsf{A} \times \mathsf{B} \to \mathsf{C}$ *with axial inverses* $\mathsf{g_a}$ *and* $\mathsf{g_b}$*. If* $\mathsf{g'_c} := \mathsf{restrict}\ \mathsf{g_c}\ (\mathsf{A}' \times \mathsf{B}')$*, then*

$$\begin{aligned} \mathsf{preimage}\ \mathsf{g'_c}\ \mathsf{C}' \ \subseteq\ & (\mathsf{A}' \cap \mathsf{image}\ \mathsf{g_a}\ (\mathsf{B}' \times \mathsf{C}')) \times \\ & (\mathsf{B}' \cap \mathsf{image}\ \mathsf{g_b}\ (\mathsf{C}' \times \mathsf{A}')) \end{aligned} \tag{8.21}$$

*Further, the right-hand side is the smallest rectangular superset.*

*Proof.* The smallest rectangle containing $\mathsf{preimage}\ \mathsf{g'_c}\ \mathsf{C}'$ is

$$\begin{aligned} \mathsf{preimage}\ \mathsf{g'_c}\ \mathsf{C}' \ \subseteq\ & (\mathsf{image}\ \mathsf{fst}\ (\mathsf{preimage}\ \mathsf{g'_c}\ \mathsf{C}')) \times \\ & (\mathsf{image}\ \mathsf{snd}\ (\mathsf{preimage}\ \mathsf{g'_c}\ \mathsf{C}')) \end{aligned} \tag{8.22}$$

Starting with the first set in the product, expand definitions, distribute $\mathsf{fst}$, replace $\mathsf{g_c}\ \langle \mathsf{a}, \mathsf{b} \rangle = \mathsf{c}$ by $\mathsf{g_a}\ \langle \mathsf{b}, \mathsf{c} \rangle = \mathsf{a}$, and simplify:

$$\begin{aligned}
&\mathsf{image\ fst\ (preimage\ g'_c\ C')} \\
&\quad = \mathsf{image\ fst}\ \{\langle \mathsf{a}, \mathsf{b} \rangle \in \mathsf{A}' \times \mathsf{B}' \mid \mathsf{g_c}\ \langle \mathsf{a}, \mathsf{b} \rangle \in \mathsf{C}'\} \\
&\quad = \{\mathsf{a} \in \mathsf{A}' \mid \exists\, \mathsf{b} \in \mathsf{B}'.\, \mathsf{g_c}\ \langle \mathsf{a}, \mathsf{b} \rangle \in \mathsf{C}'\} \\
&\quad = \{\mathsf{a} \in \mathsf{A}' \mid \exists\, \mathsf{b} \in \mathsf{B}', \mathsf{c} \in \mathsf{C}'.\, \mathsf{g_c}\ \langle \mathsf{a}, \mathsf{b} \rangle = \mathsf{c}\} \\
&\quad = \{\mathsf{a} \in \mathsf{A}' \mid \exists\, \mathsf{b} \in \mathsf{B}', \mathsf{c} \in \mathsf{C}'.\, \mathsf{g_a}\ \langle \mathsf{b}, \mathsf{c} \rangle = \mathsf{a}\} \\
&\quad = \{\mathsf{g_a}\ \langle \mathsf{b}, \mathsf{c} \rangle \mid \mathsf{b} \in \mathsf{B}', \mathsf{c} \in \mathsf{C}', \mathsf{g_a}\ \langle \mathsf{b}, \mathsf{c} \rangle \in \mathsf{A}'\} \\
&\quad = \mathsf{A}' \cap \{\mathsf{g_a}\ \langle \mathsf{b}, \mathsf{c} \rangle \mid \mathsf{b} \in \mathsf{B}', \mathsf{c} \in \mathsf{C}'\} \\
&\quad = \mathsf{A}' \cap \mathsf{image\ g_a}\ (\mathsf{B}' \times \mathsf{C}')
\end{aligned}$$

The second set in the product is similar. □

**Example 8.12.** Let $\mathsf{add'_c} := \mathsf{restrict\ add_c}\ ([0,1] \times [0,2])$. By Theorem 8.11,

$$\begin{aligned}
\mathsf{preimage\ add'_c}\ [0, \tfrac{1}{2}] &\subseteq ([0,1] \cap \mathsf{image\ add_a}\ ([0,2] \times [0, \tfrac{1}{2}])) \times \\
&\qquad ([0,2] \cap \mathsf{image\ add_b}\ ([0, \tfrac{1}{2}] \times [0,1])) \\
&= ([0,1] \cap [-2, \tfrac{1}{2}]) \times ([0,2] \cap [-1, \tfrac{1}{2}]) \\
&= [0, \tfrac{1}{2}] \times [0, \tfrac{1}{2}]
\end{aligned}$$

is the smallest rectangular subset of $[0,1] \times [0,2]$ containing the preimage of $[0, \frac{1}{2}]$ under $\mathsf{add'_c}$. Figure 8.8 illustrates the calculation. ♢

At this point, we have an analogue of Lemma 8.3, in that we can compute (approximate) preimages by computing images under (axial) inverses. Computing images using interval endpoints requires analogues of Lemma 8.2 (strictly monotone, surjective implies invertible, continuous), Theorem 8.4 (images of intervals by endpoints), and Lemma 8.5 (inverse direction).

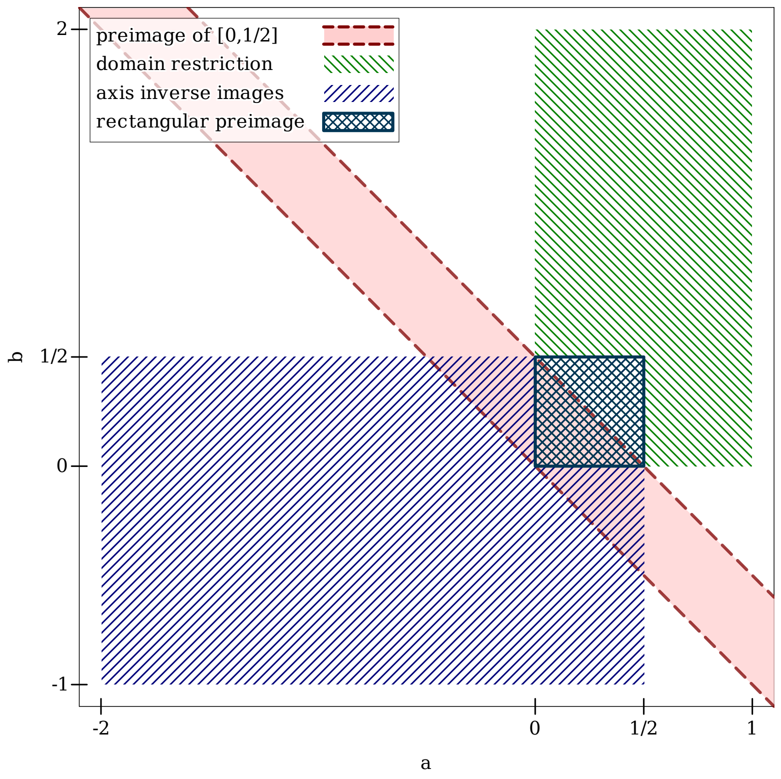


Figure 8.8: Computing an approximate preimage of $[0, \frac{1}{2}]$ under addition restricted to $[0,1] \times [0,2]$ (Example 8.12). The preimage is approximated by intersecting the domain with an overapproximation computed using axial inverses.

We first need a notion of function properties that hold for one argument for every fixed value of the other argument. We will say that $\mathsf{g_c} : \mathsf{A} \times \mathsf{B} \to \mathsf{C}$ has property $\mathsf{P}$ ***in its first axis*** when $\mathsf{P}\ (\lambda\, \mathsf{a} \in \mathsf{A}.\, \mathsf{g_c}\ \langle \mathsf{a}, \mathsf{b} \rangle)$ for all $\mathsf{b} \in \mathsf{B}$. Similarly, $\mathsf{g_c}$ has property $\mathsf{P}$ ***in its second axis*** when $\mathsf{P}\ (\lambda\, \mathsf{b} \in \mathsf{B}.\, \mathsf{g_c}\ \langle \mathsf{a}, \mathsf{b} \rangle)$ for all $\mathsf{a} \in \mathsf{A}$.

Now Lemma 8.2's analogue is an easy corollary.

**Theorem 8.13** (strictly monotone, surjective implies axis-invertible, continuous)**.** *Let* $\mathsf{g_c} : \mathsf{A} \times \mathsf{B} \to \mathsf{C}$ *for totally ordered* $\mathsf{A}$*,* $\mathsf{B}$ *and* $\mathsf{C}$*. If* $\mathsf{g_c}$ *is surjective and either strictly increasing or strictly decreasing in each axis, then* $\mathsf{g_c}$ *is axis-invertible; further, it and its axial inverses are continuous.*

*Proof.* XXX: todo □

**Example 8.14.** In each axis, $\mathsf{add_c}$ is surjective and strictly increasing. In each axis, $\mathsf{sub_c}$ is

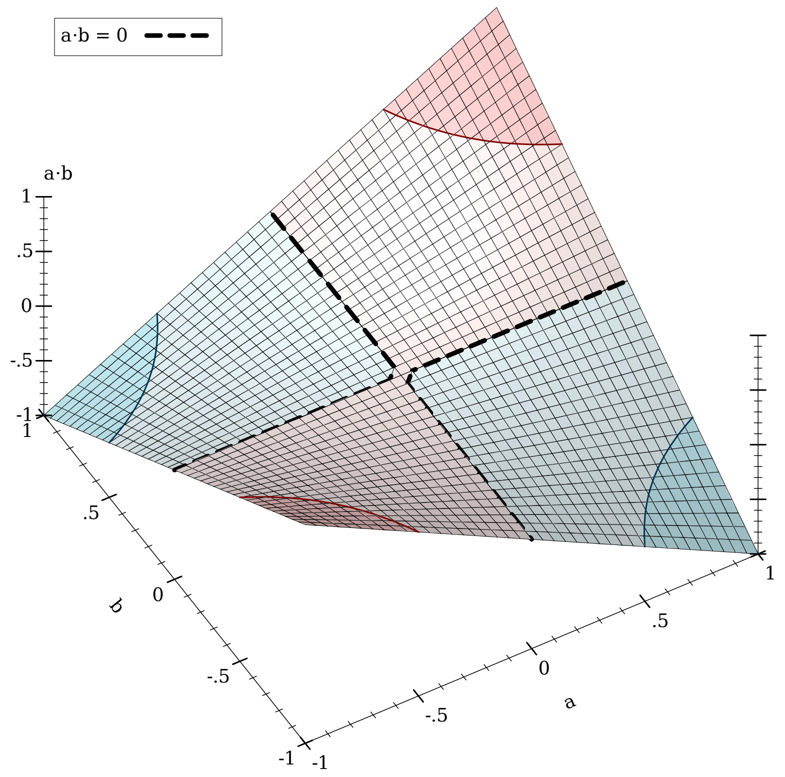


Figure 8.9: Multiplication on $\mathbb{R} \times \mathbb{R}$ is not surjective nor strictly monotone in each axis: $\mathsf{a} \cdot 0 = 0$ and $0 \cdot \mathsf{b} = 0$ for all $\mathsf{a}$ and $\mathsf{b}$ (Example 8.16). Fortunately, restricted to each open quadrant, multiplication is surjective and strictly increasing or decreasing in each axis.

surjective, and is strictly increasing/decreasing in its first/second axis. Therefore, both are axis-invertible. $\diamondsuit$

Restriction usually makes a function not surjective in each axis.

**Example 8.15.** Let $\mathsf{add}'_\mathsf{c} : [0,1] \times [0,1] \to [0,2]$, defined by restricting $\mathsf{add}_\mathsf{c}$. It is surjective, but not in each axis: the range of $\lambda\,\mathsf{b} \in \mathsf{B}.\,\mathsf{add}'_\mathsf{c}\ \langle 0, \mathsf{b}\rangle$ is $[0,1]$, not $[0,2]$. $\diamondsuit$

Fortunately, restriction sometimes does the opposite.

**Example 8.16.** Define $\mathsf{mul}_\mathsf{c} : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ by $\mathsf{mul}_\mathsf{c}\ \langle \mathsf{a}, \mathsf{b}\rangle := \mathsf{a} \cdot \mathsf{b}$. It is not surjective nor strictly monotone in each axis because $\mathsf{mul}_\mathsf{c}\ \langle 0, \mathsf{b}\rangle = 0$ for all $\mathsf{b} \in \mathsf{B}$. (See Figure 8.9.) But $\mathsf{mul}^{++}_\mathsf{c} : (0,+\infty) \times (0,+\infty) \to (0,+\infty)$, and $\mathsf{mul}_\mathsf{c}$ restricted to the other open quadrants, are surjective and strictly increasing or decreasing in each axis. $\diamondsuit$

Theorem 8.4 justifies computing images of intervals with infinite endpoints under one-argument functions by applying an extended function to the endpoints. Its two-argument analogue is more involved because extended, two-argument functions may not be defined at every point.

**Example 8.17.** $\mathsf{add_c}$ cannot be extended to $\overline{\mathsf{add_c}} : \overline{\mathbb{R}} \times \overline{\mathbb{R}} \to \overline{\mathbb{R}}$ in the same way $\mathsf{sqr}^+$ is extended to $\overline{\mathsf{sqr}^+}$ because

$$\lim_{\langle \mathsf{a}', \mathsf{b}' \rangle \to \langle \mathsf{a}, \mathsf{b} \rangle} \mathsf{add_c}\ \langle \mathsf{a}', \mathsf{b}' \rangle \tag{8.23}$$

diverges when $\langle \mathsf{a}, \mathsf{b} \rangle$ is $\langle -\infty, +\infty \rangle$ or $\langle +\infty, -\infty \rangle$. ♢

The previous example suggests that extensions of strictly increasing, two-argument functions are always well-defined except at off-diagonal corners. This is true, and similar statements hold for axes with other directions, and for more restricted domains.

**Theorem 8.18** ($\overline{\mathbb{R}} \times \overline{\mathbb{R}}$ extension)**.** *Let* $\mathsf{A}$, $\mathsf{B}$, $\mathsf{C}$ *be open subsets of* $\mathbb{R}$, *and* $\mathsf{g_c} : \mathsf{A} \times \mathsf{B} \to \mathsf{C}$ *be surjective and strictly increasing or decreasing in each axis. Let* $\overline{\mathsf{A}}$, $\overline{\mathsf{B}}$ *and* $\overline{\mathsf{C}}$ *be the closures of* $\mathsf{A}$, $\mathsf{B}$ *and* $\mathsf{C}$ *in* $\overline{\mathbb{R}}$. *The following extension is well-defined:*

$$\begin{aligned} &\overline{\mathsf{g_c}} : (\overline{\mathsf{A}} \times \overline{\mathsf{B}}) \backslash \mathsf{N} \to \overline{\mathsf{C}} \\ &\overline{\mathsf{g_c}}\ \langle \mathsf{a}, \mathsf{b} \rangle \ := \lim_{\langle \mathsf{a}', \mathsf{b}' \rangle \to \langle \mathsf{a}, \mathsf{b} \rangle} \mathsf{g_c}\ \langle \mathsf{a}', \mathsf{b}' \rangle \end{aligned} \tag{8.24}$$

*where* $\mathsf{N} := \{\langle \min \overline{\mathsf{A}}, \max \overline{\mathsf{B}} \rangle, \langle \max \overline{\mathsf{A}}, \min \overline{\mathsf{B}} \rangle\}$ *if* $\mathsf{g_c}$ *is increasing in each axis or decreasing in each axis, and* $\mathsf{N} := \{\langle \min \overline{\mathsf{A}}, \min \overline{\mathsf{B}} \rangle, \langle \max \overline{\mathsf{A}}, \max \overline{\mathsf{B}} \rangle\}$ *if* $\mathsf{g_c}$ *is increasing/decreasing or decreasing/increasing.*

*Proof.* Suppose $\mathsf{g_c}$ is increasing/increasing, and let $\mathsf{xs} : \mathbb{N} \to \mathsf{A} \times \mathsf{B}$ be a sequence of $\mathsf{g_c}$'s domain values, and $\mathsf{ys} := \mathsf{map}\ \mathsf{g_c}\ \mathsf{xs}$.

Interior case: $\mathsf{xs}$ converges to $\langle \mathsf{a}, \mathsf{b} \rangle \in \mathsf{A} \times \mathsf{B}$. The limit of $\mathsf{ys}$ is $\mathsf{g_c}\ \langle \mathsf{a}, \mathsf{b} \rangle$ because $\mathsf{g_c}$ preserves limits by its continuity (by Theorem 8.13) in the first-countable space $\mathbb{R} \times \mathbb{R}$.

Corner case: $\mathsf{xs}$ converges to $\langle \max \overline{\mathsf{A}}, \max \overline{\mathsf{B}} \rangle$. It thus has a strictly increasing subsequence. By monotonicity, $\mathsf{ys}$ has a strictly increasing subsequence. Because $\mathsf{ys}$ is bounded by

$\mathsf{max}\ \overline{\mathsf{C}}$, $\overline{\mathsf{g_c}}\ \langle \mathsf{max}\ \overline{\mathsf{A}}, \mathsf{max}\ \overline{\mathsf{B}}\rangle = \mathsf{max}\ \overline{\mathsf{C}}$. A similar argument proves $\overline{\mathsf{g_c}}\ \langle \mathsf{min}\ \overline{\mathsf{A}}, \mathsf{min}\ \overline{\mathsf{B}}\rangle = \mathsf{min}\ \overline{\mathsf{C}}$.

Border case: $\mathsf{xs}$ converges to $\langle \mathsf{max}\ \overline{\mathsf{A}}, \mathsf{b}'\rangle$ for some $\mathsf{b}' \in \mathsf{B}$. Define

$$\mathsf{xs}' \ :=\ \mathsf{map}\ (\lambda\, \langle \mathsf{a}, \mathsf{b}\rangle .\, \langle \mathsf{g_a}\ \langle \mathsf{b}', \mathsf{g_c}\ \langle \mathsf{a}, \mathsf{b}\rangle\rangle, \mathsf{b}'\rangle)\ \mathsf{xs} \tag{8.25}$$

where $\mathsf{g_a}$ is $\mathsf{g_c}$'s first axial inverse. Now $\mathsf{ys} = \mathsf{map}\ \mathsf{g_c}\ \mathsf{xs}'$. Because $\mathsf{xs}'$ has a subsequence that is strictly increasing in the first of each pair, and because the second of each pair is the constant $\mathsf{b}'$, by monotonicity, $\mathsf{ys}$ has a strictly increasing subsequence. It is bounded by $\mathsf{max}\ \overline{\mathsf{C}}$, so $\overline{\mathsf{g_c}}\ \langle \mathsf{max}\ \overline{\mathsf{A}}, \mathsf{b}'\rangle = \mathsf{max}\ \overline{\mathsf{C}}$. Similar arguments prove $\overline{\mathsf{g_c}}\ \langle \mathsf{min}\ \overline{\mathsf{A}}, \mathsf{b}'\rangle = \mathsf{min}\ \overline{\mathsf{C}}$, $\overline{\mathsf{g_c}}\ \langle \mathsf{a}', \mathsf{max}\ \overline{\mathsf{B}}\rangle = \mathsf{max}\ \overline{\mathsf{C}}$, and $\overline{\mathsf{g_c}}\ \langle \mathsf{a}', \mathsf{min}\ \overline{\mathsf{B}}\rangle = \mathsf{min}\ \overline{\mathsf{C}}$.

The cases for $\mathsf{g_c}$'s other possible directions are similar. □

Following the proof of Theorem 8.18, extensions of two-argument functions can be defined by two corner cases, four border cases, and an interior case.

**Example 8.19.** Define $\mathsf{pow_c} : (0,1) \times (0,+\infty) \to (0,1)$ by $\mathsf{pow_c}\ \langle \mathsf{a}, \mathsf{b}\rangle := \mathsf{exp}\ (\mathsf{b} \cdot \mathsf{log}\ \mathsf{a})$, which is increasing/decreasing. Its extension to a subset of $\overline{\mathbb{R}} \times \overline{\mathbb{R}}$ is

$$\begin{aligned}
&\overline{\mathsf{pow_c}} : ([0,1] \times [0,+\infty])\backslash \mathsf{N} \to [0,1] \\
&\overline{\mathsf{pow_c}}\ \langle \mathsf{a}, \mathsf{b}\rangle \ :=\ \mathsf{case}\ \ \langle \mathsf{a}, \mathsf{b}\rangle \\
&\qquad\qquad\qquad \begin{array}{lcl}
\langle 0, +\infty\rangle & \longrightarrow & 0 \\
\langle 1, 0\rangle & \longrightarrow & 1 \\
\langle 0, \mathsf{b}\rangle & \longrightarrow & 0 \\
\langle 1, \mathsf{b}\rangle & \longrightarrow & 1 \\
\langle \mathsf{a}, 0\rangle & \longrightarrow & 1 \\
\langle \mathsf{a}, +\infty\rangle & \longrightarrow & 0 \\
\mathsf{else} & \longrightarrow & \mathsf{pow_c}\ \langle \mathsf{a}, \mathsf{b}\rangle
\end{array}
\end{aligned} \tag{8.26}$$

where $\mathsf{N} := \{\langle 0, 0\rangle, \langle 1, +\infty\rangle\}$. ♢

The analogue of Theorem 8.4 (images of intervals by endpoints) is easiest to state if we have predicates that indicate a function's direction in each axis. Define $\mathsf{inc}_1 : (\mathsf{A} \times \mathsf{B} \to \mathsf{C}) \Rightarrow \mathsf{Bool}$ so that $\mathsf{inc}_1\ \mathsf{g}$ if and only if $\mathsf{g}$ is strictly increasing in its first axis, and similarly $\mathsf{inc}_2$ so that $\mathsf{inc}_2\ \mathsf{g}$ if and only if $\mathsf{g}$ is strictly increasing in its second axis.

**Theorem 8.20** (images of rectangles by interval endpoints)**.** *Let* $\mathsf{A}, \mathsf{B}, \mathsf{C}$ *be open subsets of* $\mathbb{R}$*, and* $\mathsf{g_c} : \mathsf{A} \times \mathsf{B} \to \mathsf{C}$ *be surjective and strictly increasing or decreasing in each axis, with* $\overline{\mathsf{g_c}}$ *as defined in Theorem 8.18. If* $\mathsf{A}' := [\![\mathsf{a}_1, \mathsf{a}_2, \alpha_1, \alpha_2]\!] \subseteq \mathsf{A}$ *and* $\mathsf{B}' := [\![\mathsf{b}_1, \mathsf{b}_2, \beta_1, \beta_2]\!] \subseteq \mathsf{B}$*, then*

$$\begin{aligned}
\mathsf{C}' &:= \mathsf{image}\ \mathsf{g_c}\ ([\![\mathsf{a}_1, \mathsf{a}_2, \alpha_1, \alpha_2]\!] \times [\![\mathsf{b}_1, \mathsf{b}_2, \beta_1, \beta_2]\!]) \\
&= \mathsf{let}\ \ \langle \mathsf{a}'_1, \mathsf{a}'_2, \alpha'_1, \alpha'_2 \rangle := \mathsf{cond}\ \ \begin{array}{lcl} (\mathsf{inc}_1\ \mathsf{g_c}) & \longrightarrow & \langle \mathsf{a}_1, \mathsf{a}_2, \alpha_1, \alpha_2 \rangle \\ \mathsf{else} & \longrightarrow & \langle \mathsf{a}_2, \mathsf{a}_1, \alpha_2, \alpha_1 \rangle \end{array} \\
&\qquad\qquad \langle \mathsf{b}'_1, \mathsf{b}'_2, \beta'_1, \beta'_2 \rangle := \mathsf{cond}\ \ \begin{array}{lcl} (\mathsf{inc}_2\ \mathsf{g_c}) & \longrightarrow & \langle \mathsf{b}_1, \mathsf{b}_2, \beta_1, \beta_2 \rangle \\ \mathsf{else} & \longrightarrow & \langle \mathsf{b}_2, \mathsf{b}_1, \beta_2, \beta_1 \rangle \end{array} \\
&\qquad \mathsf{in}\ \ [\![\overline{\mathsf{g_c}}\ \langle \mathsf{a}'_1, \mathsf{b}'_1 \rangle, \overline{\mathsf{g_c}}\ \langle \mathsf{a}'_2, \mathsf{b}'_2 \rangle, \alpha'_1\ \mathsf{and}\ \beta'_1, \alpha'_2\ \mathsf{and}\ \beta'_2]\!]
\end{aligned} \tag{8.27}$$

*Proof.* Because $\mathsf{g_c}$ is continuous and $\mathsf{A}' \times \mathsf{B}'$ is a connected set, $\mathsf{C}'$ is a connected set, which in $\mathbb{R}$ is an interval. Thus, we need to determine only its endpoints and whether it contains each endpoint.

Suppose $\mathsf{g_c}$ is increasing/increasing. In this case, $\mathsf{a}'_1 = \mathsf{a}_1$, $\mathsf{b}'_1 = \mathsf{b}_1$, and so on. By monotonicity, $\mathsf{C}'$ is contained in $[\overline{\mathsf{g_c}}\ \langle \mathsf{a}'_1, \mathsf{b}'_1 \rangle, \overline{\mathsf{g_c}}\ \langle \mathsf{a}'_2, \mathsf{b}'_2 \rangle]$. If $\alpha'_1$ or $\beta'_1$ is $\mathsf{false}$, $\mathsf{C}'$ cannot contain $\overline{\mathsf{g_c}}\ \langle \mathsf{a}'_1, \mathsf{b}'_1 \rangle$. If $\alpha'_2$ or $\beta'_2$ is $\mathsf{false}$, $\mathsf{C}'$ cannot contain $\overline{\mathsf{g_c}}\ \langle \mathsf{a}'_2, \mathsf{b}'_2 \rangle$. Therefore $\mathsf{C}' = [\![\overline{\mathsf{g_c}}\ \langle \mathsf{a}'_1, \mathsf{b}'_1 \rangle, \overline{\mathsf{g_c}}\ \langle \mathsf{a}'_2, \mathsf{b}'_2 \rangle, \alpha'_1\ \mathsf{and}\ \beta'_1, \alpha'_2\ \mathsf{and}\ \beta'_2]\!]$.

We still must prove $\langle \mathsf{a}'_1, \mathsf{b}'_1 \rangle$ and $\langle \mathsf{a}'_2, \mathsf{b}'_2 \rangle$ are in $\overline{\mathsf{g_c}}$'s domain. First, recall $\overline{\mathsf{g_c}} : (\overline{\mathsf{A}} \times \overline{\mathsf{B}}) \backslash \mathsf{N} \to \overline{\mathsf{C}}$, where $\overline{\mathsf{A}}$, $\overline{\mathsf{B}}$ and $\overline{\mathsf{C}}$ are the closures of $\mathsf{A}$, $\mathsf{B}$ and $\mathsf{C}$, and $\mathsf{N} = \{\langle \mathsf{min}\ \overline{\mathsf{A}}, \mathsf{max}\ \overline{\mathsf{B}} \rangle, \langle \mathsf{max}\ \overline{\mathsf{A}}, \mathsf{min}\ \overline{\mathsf{B}} \rangle\}$. Because $\mathsf{A}' \subseteq \mathsf{A}$ and $\mathsf{B}' \subseteq \mathsf{B}$, and $\mathsf{A}$ and $\mathsf{B}$ are open sets, $\mathsf{a}_1 \neq \mathsf{max}\ \overline{\mathsf{A}}$, $\mathsf{a}_2 \neq \mathsf{min}\ \overline{\mathsf{A}}$, $\mathsf{b}_1 \neq \mathsf{max}\ \overline{\mathsf{B}}$, and $\mathsf{b}_2 \neq \mathsf{min}\ \overline{\mathsf{B}}$, so for all $\mathsf{a} \in \overline{\mathsf{A}}$ and $\mathsf{b} \in \overline{\mathsf{B}}$,

$$\begin{array}{ll}
\langle \mathsf{a}_1, \mathsf{b}_1 \rangle \neq \langle \mathsf{max}\ \overline{\mathsf{A}}, \mathsf{b} \rangle & \qquad \langle \mathsf{a}_2, \mathsf{b}_2 \rangle \neq \langle \mathsf{min}\ \overline{\mathsf{A}}, \mathsf{b} \rangle \\
\langle \mathsf{a}_1, \mathsf{b}_1 \rangle \neq \langle \mathsf{a}, \mathsf{max}\ \overline{\mathsf{B}} \rangle & \qquad \langle \mathsf{a}_2, \mathsf{b}_2 \rangle \neq \langle \mathsf{a}, \mathsf{min}\ \overline{\mathsf{B}} \rangle
\end{array} \tag{8.28}$$

Therefore, $\langle \mathsf{a}_1, \mathsf{b}_1 \rangle \notin \mathsf{N}$ and $\langle \mathsf{a}_2, \mathsf{b}_2 \rangle \notin \mathsf{N}$, as desired.

The remaining cases for $\mathsf{g_c}$ are similar. □

**Example 8.21.** Because $\mathsf{inc}_1\ \mathsf{pow_c}$ and $\mathsf{not}\ (\mathsf{inc}_2\ \mathsf{pow_c})$,

$$\begin{aligned}
&\mathsf{image}\ \mathsf{pow_c}\ ((0, \tfrac{1}{2}] \times [2, +\infty)) \\
&= \mathsf{let}\ \ \langle \mathsf{a}_1, \mathsf{a}_2, \alpha_1, \alpha_2 \rangle := \langle 0, \tfrac{1}{2}, \mathsf{false}, \mathsf{true} \rangle \\
&\qquad\quad \langle \mathsf{b}_1, \mathsf{b}_2, \beta_1, \beta_2 \rangle := \langle +\infty, 2, \mathsf{false}, \mathsf{true} \rangle \\
&\qquad \mathsf{in}\ \ [\![ \overline{\mathsf{pow_c}}\ \langle \mathsf{a}_1, \mathsf{b}_1 \rangle, \overline{\mathsf{pow_c}}\ \langle \mathsf{a}_2, \mathsf{b}_2 \rangle, \alpha_1\ \mathsf{and}\ \beta_1, \alpha_2\ \mathsf{and}\ \beta_2 ]\!] \\
&= [\![ \overline{\mathsf{pow_c}}\ \langle 0, +\infty \rangle, \overline{\mathsf{pow_c}}\ \langle \tfrac{1}{2}, 2 \rangle, \mathsf{false\ and\ false}, \mathsf{true\ and\ true} ]\!] \\
&= [\![ 0, \tfrac{1}{4}, \mathsf{false}, \mathsf{true} ]\!] \\
&= (0, \tfrac{1}{4}]
\end{aligned}$$

♢

To use Theorem 8.20 to compute approximate preimages under some $\mathsf{g_c}$ by computing images under its axial inverses, we must know whether each axis of $\mathsf{g_a}$ and $\mathsf{g_b}$ is increasing or decreasing. It helps to have an analogue of Lemma 8.5 (inverse direction).

**Theorem 8.22** (axial inverse directions)**.** *Let* $\mathsf{g_c} : \mathsf{A} \times \mathsf{B} \to \mathsf{C}$ *be surjective and strictly increasing or decreasing in each axis, with axial inverses* $\mathsf{g_a}$ *and* $\mathsf{g_b}$*. Then*

1. $\mathsf{inc}_1\ \mathsf{g_a}$ *if and only if* $(\mathsf{inc}_1\ \mathsf{g_c})\ \mathsf{xor}\ (\mathsf{inc}_2\ \mathsf{g_c})$.
2. $\mathsf{inc}_2\ \mathsf{g_a}$ *if and only if* $\mathsf{inc}_1\ \mathsf{g_c}$.

*Proof.* For 1, let $\mathsf{c} \in \mathsf{C}$, $\mathsf{b}_1, \mathsf{b}_2 \in \mathsf{B}$, $\mathsf{a}_1 := \mathsf{g_a}\ \langle \mathsf{b}_1, \mathsf{c} \rangle$ and $\mathsf{a}_2 := \mathsf{g_a}\ \langle \mathsf{b}_2, \mathsf{c} \rangle$. Let $\mathsf{c}' := \mathsf{g_c}\ \langle \mathsf{a}_1, \mathsf{b}_2 \rangle$; note $\mathsf{c} = \mathsf{g_c}\ \langle \mathsf{a}_1, \mathsf{b}_1 \rangle = \mathsf{g_c}\ \langle \mathsf{a}_2, \mathsf{b}_2 \rangle$. Suppose $\mathsf{inc}_1\ \mathsf{g_c}$ and $\mathsf{inc}_2\ \mathsf{g_c}$; then $\mathsf{a}_1 > \mathsf{a}_2 \iff \mathsf{c} < \mathsf{c}'$ and $\mathsf{b}_1 < \mathsf{b}_2 \iff \mathsf{c} < \mathsf{c}'$, so $\mathsf{b}_1 < \mathsf{b}_2 \iff \mathsf{a}_1 > \mathsf{a}_2$. The remaining cases are similar.

For statement 2, fix $\mathsf{b} \in \mathsf{B}$ and apply Lemma 8.5. □

By Theorem 8.22, we can use $\mathsf{g_c}$'s axis directions to determine $\mathsf{g_a}$'s, and by Lemma 8.9 (axial inverse cyclic group), use $\mathsf{g_a}$'s to determine $\mathsf{g_b}$'s.

### 8.3.3 Primitive Implementation

Because floating-point functions are defined on subsets of $\overline{\mathbb{R}}$, it would seem we could compute preimages under strictly monotone, real functions by applying their floating-point counterparts to interval endpoints. This is mostly true, but as with `real-set-singleton`, we must take care with rounding. We must also account for floating-point negative zero.

As with all interval arithmetic, to compute sound approximations of interval images, we must round the results *outward*: round the lower endpoints down, and round the upper endpoints up. Unlike with most interval arithmetic, soundness is not just a nice theoretical guarantee. For the lowest-rejection-rate sampling algorithm presented further on, it is critical.

The sampler chooses a random value $\mathsf{a}$, restricts $\Omega$ at index $\mathsf{j}$ to $[\mathsf{a},\mathsf{a}]$ using $\Omega' :=$ $\mathsf{unproj\ j}\ \Omega\ [\mathsf{a},\mathsf{a}]$, and computes a preimage under the program's interpretion as a function, restricted to $\Omega'$. If in the forward pass, the approximation of the image of $\Omega'$ is not sound, the reverse pass will often falsely compute an empty preimage.

Here is a more concrete example. As a preimage arrow computation, square root is

$$\begin{aligned} &\mathsf{sqrt_{pre}} : [0,+\infty) \underset{\mathsf{pre}}{\rightsquigarrow} [0,+\infty) \\ &\mathsf{sqrt_{pre}\ A}\ :=\ \langle \mathsf{image\ sqrt^{+}\ A}, \mathsf{preimage\ (restrict\ sqrt^{+}\ A)} \rangle \end{aligned} \tag{8.29}$$

Suppose $\mathsf{A} = [\frac{1}{2},\frac{1}{2}]$, and that the implementation mistakenly computes $\mathsf{image\ sqrt^{+}}\ [\frac{1}{2},\frac{1}{2}]$ as $[0.7071067811865476, 0.7071067811865476]$. The number 0.7071067811865476 is the closest 64-bit floating-point number to $\sqrt{\frac{1}{2}}$; i.e. the implementation's floating-point square root is compliant with the IEEE 754 floating-point standard [XXX: cite].

Now suppose that, on the reverse phase, we compute the preimage of $\mathbb{R}$ under $\mathsf{restrict\ sqrt^{+}}\ [\frac{1}{2},\frac{1}{2}]$. By Lemma 8.3, the implementation of $\mathsf{pre}$ should compute

$$\begin{aligned} &\mathsf{preimage\ (restrict\ sqrt^{+}}\ [\tfrac{1}{2},\tfrac{1}{2}])\ (\mathbb{R} \cap [0.7071067811865476, 0.7071067811865476]) \\ &\quad=\ [\tfrac{1}{2},\tfrac{1}{2}] \cap \mathsf{image\ sqr^{+}}\ [0.7071067811865476, 0.7071067811865476] \end{aligned} \tag{8.30}$$

If it again uses compliant floating-point arithmetic but does not round outward, it computes

$$[\tfrac{1}{2},\tfrac{1}{2}] \cap [0.5000000000000001, 0.5000000000000001]\ =\ \varnothing \tag{8.31}$$

In fact, an implementation that does not round intervals outward would falsely compute $\varnothing$

for about half of the floating-point numbers between 0.0 and 1.0.

The IEEE 754 floating-point standard mandates a settable rounding mode, and that common operations must use it to determine which of the nearest floating-point numbers to round to. Unfortunately, there is no portable way to set the rounding mode. In Racket, we have a few other options.

1. Use `math/bigfloat`, which wraps the MPFR arbitrary-precision floating-point library [XXX: cite], which *does* provide a way to set the rounding mode for its operations.
2. Use the `math/flonum` library's functions for **double-doubles**, which are two nonoverlapping floating-point numbers that when added together represent a number with a 105-bit significand [XXX: cite]. Convert flonums to double-doubles, operate on them, and manually round the high-order number of the result up or down based on the sign of the low-order number.
3. Use the `math/flonum` library's `flnext` and `flprev` to bump the endpoints up or down.

We use option 2 for functions with 105-bit implementations, such as arithmetic, $\mathsf{exp}$ and $\mathsf{log}$, and otherwise use option 3.

For option 3, how far the endpoints are bumped up or down depends on the maximum error in the output of the function's floating-point implementation. For example, we use the normal distribution's inverse cumulative density function $\mathsf{F}_\mathsf{N}^{-1}$ and (its inverse $\mathsf{F}_\mathsf{N}$) to transform uniformly distributed random numbers (i.e. each $\omega\ \mathsf{j}$) into normally distributed random numbers. As a preimage arrow computation, it is

$$\begin{aligned} &\mathsf{normal_{pre}} : (0,1) \underset{\mathsf{pre}}{\rightsquigarrow} \mathbb{R} \\ &\mathsf{normal_{pre}}\ \mathsf{A}\ :=\ \langle \mathsf{image}\ \mathsf{F}_\mathsf{N}^{-1}\ \mathsf{A}, \mathsf{preimage}\ (\mathsf{restrict}\ \mathsf{F}_\mathsf{N}^{-1}\ \mathsf{A})\rangle \end{aligned} \tag{8.32}$$

Racket's `math/distributions` library implements $\mathsf{F}_\mathsf{N}^{-1}$ with `flnormal-inv-cdf` and $\mathsf{F}_\mathsf{N}$ with `flnormal-cdf`, whose outputs are always within four floating-point numbers of the exact outputs. The implementation of $\mathsf{normal_{pre}}$ therefore bumps lower endpoints down by 4 and upper endpoints up by 4.

For the code in this section, we use a `/rndu` suffix (read "with rounding up") for the names of functions that round up, and a `/rndd` for the name of functions that round down. In Racket, prefixing floating-point functions with `fl` is conventional, so the name of the floating-point addition function that rounds down is `fl+/rndd`, and the name of the floating-point square root function that rounds up is `flsqrt/rndu`.

Figure 8.10 lists code that computes sound image and preimage approximations under strictly monotone, surjective real functions. Such functions are represented by instances of `Bijection`. Each instance contains a `Boolean` indicating whether the function is increasing, its domain, range, an implementation with rounding down and up, and an inverse implementation with rounding down and up. For example,

```
(define pos-sqr-bij
  (Bijection #t nonnegative-reals nonnegative-reals
             flsqr/rndd flsqr/rndu
             flsqrt/rndd flsqrt/rndu))

(define sqrt-bij
  (bijection-inverse pos-sqr-bij))
```

The preimage arrow computation `sqrt-pre` computes `(bijection-image sqrt-bij A)` in the forward phase and `(bijection-preimage sqrt-bij A B)` in the reverse phase.

A simple example shows how floating-point's signed zeros can cause problems: the implementation of the reciprocal function. Let $\overline{\mathsf{recip}^+}$ be the extension of $\mathsf{recip}^+ : (0, +\infty) \to (0, +\infty)$ to the compact superdomain $[0, +\infty]$, defined by

$$\begin{aligned}
&\overline{\mathsf{recip}^+} : [0, +\infty] \to [0, +\infty] \\
&\overline{\mathsf{recip}^+}\ \mathsf{a} \ :=\ \lim_{\mathsf{a}' \to \mathsf{a}} \ \mathsf{recip}^+\ \mathsf{a}' \ =\ \begin{array}{l} \mathsf{case}\ \ \mathsf{a} \\ \quad \begin{array}{lll} 0 & \longrightarrow & +\infty \\ +\infty & \longrightarrow & 0 \\ \mathsf{else} & \longrightarrow & \mathsf{recip}^+\ \mathsf{a} \end{array} \end{array}
\end{aligned} \tag{8.33}$$

Suppose we implement it this way:

```
;; Represents an R -> R bijection, its direction, domain and range
(struct: Bijection
    ([inc? : Boolean] [domain : Real-Set] [range : Real-Set]
     [gb/rndd : (Flonum -> Flonum)] [gb/rndu : (Flonum -> Flonum)]
     [ga/rndd : (Flonum -> Flonum)] [ga/rndu : (Flonum -> Flonum)]))

(: bijection-inverse (Bijection -> Bijection))
;; Returns the inverse of a bijection (see Lemma 8.5)
(define (bijection-inverse g)
  (match-define (Bijection inc? X Y gb/rndd gb/rndu ga/rndd ga/rndu) g)
  (Bijection inc? Y X ga/rndd ga/rndu gb/rndd gb/rndu))

(: real-image (Boolean (Flonum -> Flonum) (Flonum -> Flonum) Real-Set
                -> Real-Set))
;; Returns a sound approximation of the image of A under g (Theorem 8.4)
(define (real-image inc? g/rndd g/rndu A)
  (match-define (Real-Set a1 a2 a1? a2?) A)
  (cond [inc?  (Real-Set (g/rndd a1) (g/rndu a2) a1? a2?)]
        [else  (Real-Set (g/rndd a2) (g/rndu a1) a2? a1?)]))

(: bijection-image (Bijection Real-Set -> (U Empty-Set Real-Set)))
;; Computes the image of A under bijection g
(define (bijection-image g A)
  (match-define (Bijection inc? X Y gb/rndd gb/rndu _ _) g)
  (let ([A  (real-set-intersect A X)])
    (if (empty-set? A)
        empty-set
        (real-set-intersect Y (real-image inc? gb/rndd gb/rndu A)))))

(: bijection-preimage (bijection Real-Set Real-Set -> (U Empty-Set Real-Set)))
;; Returns an approximate preimage of B under g restricted to A (Lemma 8.3)
(define (bijection-preimage g A B)
  (match-define (Bijection inc? X Y _ _ ga/rndd ga/rndu) g)
  (let ([A  (real-set-intersect A X)]
        [B  (real-set-intersect B Y)])
    (if (or (empty-set? A) (empty-set? B))
        empty-set
        (real-set-intersect A (real-image inc? ga/rndd ga/rndu B)))))
```

Figure 8.10: Typed Racket code for computing images and preimages under strictly monotone, surjective real functions.

```
(define pos-recip-bij
  (Bijection #f positive-reals positive-reals
             (λ (a) (fl//rndd 1.0 a)) (λ (a) (fl//rndu 1.0 a))
             (λ (a) (fl//rndd 1.0 a)) (λ (a) (fl//rndu 1.0 a))))
```

Because $\mathsf{recip}^+$ is surjective, $\mathsf{image}\ \mathsf{recip}^+\ (0, +\infty) = (0, +\infty)$. With this implementation, we get the expected result only when the left endpoint is *positive* floating-point zero, or `+0.0`:

```
> (bijection-image pos-recip-bij (Real-Set +0.0 +inf.0 #f #f))
(Real-Set 0.0 +inf.0 #f #f)

> (bijection-image pos-recip-bij (Real-Set -0.0 +inf.0 #f #f))
empty-set
```

The issue is that `(fl/ 1.0 +0.0)` returns `+inf.0`, but `(fl/ 1.0 -0.0)` returns `-inf.0`, as per the IEEE 754 floating-point standard. The implementation should compute

$$\mathsf{image}\ \mathsf{recip}^{+}\ (0, +\infty) \;=\; \left(\overline{\mathsf{recip}^{+}}\ +\infty, \overline{\mathsf{recip}^{+}}\ 0\right) \;=\; (0, +\infty) \tag{8.34}$$

but tries to return $(0, -\infty)$, which is the empty set.

In interval arithmetic, the typical solution is to allow `+0.0` only as a *lower* endpoint, and `-0.0` as only as an *upper* endpoint [XXX: cite]. We have not determined whether this solution generalizes to nonarithmetic functions, however, so we define

```
(define (pos-recip/rndd a)
  (if (fl= a 0.0) +inf.0 (fl/ 1.0 a)))
```

and similarly `pos-recip/rndu`, and define `pos-recip-bij` in terms of these functions.

Figure 8.11 lists code that computes sound image and preimage approximations under two-dimensional real functions that are surjective and strictly increasing or decreasing in each axis. Such functions are represented by instances of `Trijection`. Each instance contains two `Boolean` values indicating whether each axis is increasing, its axis domains, its range, an implementation with rounding down and up, and two axial inverse implementations with rounding down and up. For example, the implementations of addition and subtraction are

```
(define add-trij
  (Trijection #t #t reals reals reals
              fl+/rndd fl+/rndu
              flrev-/rndd flrev-/rndu
              fl-/rndd fl-/rndu))

(define sub-trij
  (trijection-second-inverse add-trij))
```

```
;; Represents an R x R -> R trijection, its directions, domain and range
(struct: Trijection
 ([inc1? : Boolean] [inc2? : Boolean]
  [domain1 : Real-Set] [domain2 : Real-Set] [range : Real-Set]
  [gc/rndd : (Flonum Flonum -> Flonum)] [gc/rndu : (Flonum Flonum -> Flonum)]
  [ga/rndd : (Flonum Flonum -> Flonum)] [ga/rndu : (Flonum Flonum -> Flonum)]
  [gb/rndd : (Flonum Flonum -> Flonum)] [gb/rndu : (Flonum Flonum -> Flonum)]))

(: real2d-image (Boolean Boolean
                 (Flonum Flonum -> Flonum)
                 (Flonum Flonum -> Flonum)
                 Real-Set Real-Set -> Real-Set))
;; Returns a sound approximation of the image of AxB under g (Theorem 8.20)
(define (real2d-image inc1? inc2? g/rndd g/rndu A B)
  (define-values (a1 a2 a1? a2?)
    (match-let ([(Real-Set a1 a2 a1? a2?)  A])
      (cond [inc1?  (values a1 a2 a1? a2?)]
            [else   (values a2 a1 a2? a1?)])))
  (define-values (b1 b2 b1? b2?)
    (match-let ([(Real-Set b1 b2 b1? b2?)  B])
      (cond [inc2?  (values b1 b2 b1? b2?)]
            [else   (values b2 b1 b2? b1?)])))
  (Real-Set (g/rndd a1 b1) (g/rndu a2 b2) (and a1? b1?) (and a2? b2?)))

(: trijection-preimage (Trijection Real-Set Real-Set Real-Set
                        -> (Values (U Empty-Set Real-Set)
                                   (U Empty-Set Real-Set))))
;; Returns an approximate preimage of C under g restricted to AxB
;; (Theorem 8.11, Theorem 8.22)
(define (trijection-preimage g A B C)
  (match-define (Trijection gc-inc1? gc-inc2? X Y Z
                            _ _ ga/rndd ga/rndu gb/rndd gb/rndu) g)
  (define ga-inc1? (xor gc-inc1? gc-inc2?))
  (define ga-inc2? gc-inc1?)
  (define gb-inc1? (xor ga-inc1? ga-inc2?))
  (define gb-inc2? ga-inc1?)
  (let ([A  (real-set-intersect A X)]
        [B  (real-set-intersect B Y)]
        [C  (real-set-intersect C Z)])
    (if (or (empty-set? A) (empty-set? B) (empty-set? C))
        (values empty-set empty-set)
        (values (real-set-intersect
                 A (real2d-image ga-inc1? ga-inc2? ga/rndd ga/rndu B C))
                (real-set-intersect
                 B (real2d-image gb-inc1? gb-inc2? gb/rndd gb/rndu C A))))))
```

Figure 8.11: Typed Racket code for computing images and preimages under two-dimensional real functions that are surjective and strictly increasing or decreasing in each axis.

where `flrev-/rndd` implements $\mathsf{add_a}\ \langle \mathsf{b}, \mathsf{c}\rangle := \mathsf{c} - \mathsf{b}$ with rounding down.

### 8.3.4 Piecewise Monotone Primitives

Using $\mathsf{ifte_{pre}}$, it is easy to provide primitives that are piecewise monotone with finitely many pieces. We first need predicates to distinguish the parts, so we define

$$\begin{aligned} &\mathsf{negative?_{pre}} : \mathbb{R} \underset{\mathsf{pre}}{\rightsquigarrow} \mathsf{Bool} \\ &\mathsf{negative?_{pre}} \ := \ \mathsf{predicate_{pre}}\ (-\infty, 0]\ (0, +\infty) \end{aligned} \tag{8.35}$$

as well as $\mathsf{positive?_{pre}}$ in the same way.

From $\mathsf{sqr^+_{pre}}$ (nonnegative square) and $\mathsf{neg_{pre}}$ (negation) primitives, we define

$$\begin{aligned} \mathsf{sqr_{pre}} \ &:= \ \mathsf{ifte_{pre}}\ \mathsf{negative?_{pre}}\ (\mathsf{neg_{pre}} \ggg_{\mathsf{pre}} \mathsf{sqr^+_{pre}})\ \mathsf{sqr^+_{pre}} \\ \mathsf{sqr_{pre^*}} \ &:= \ \eta_{\mathsf{pre^*}}\ \mathsf{sqr_{pre}} \end{aligned} \tag{8.36}$$

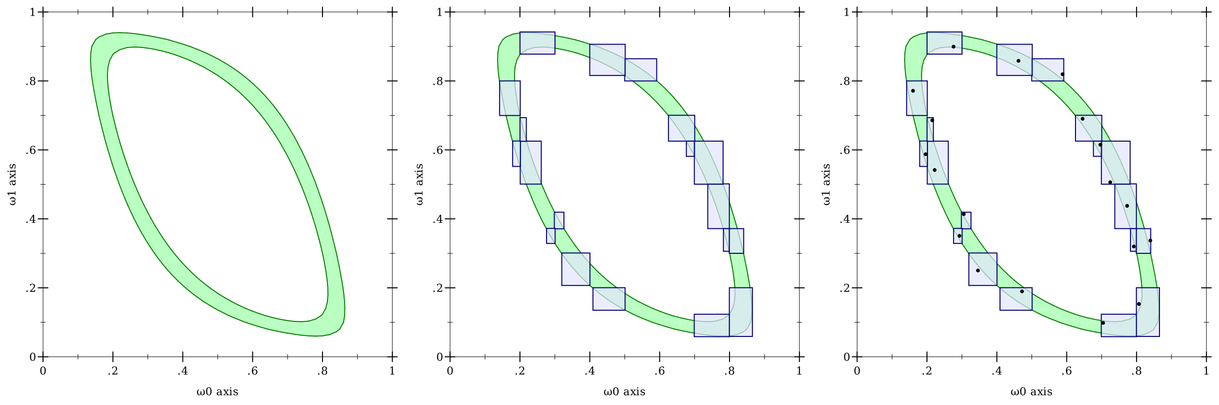
We then extend $[\![\cdot]\!]^{\Downarrow}_{\mathsf{pre^*}}$ by the rule $[\![\mathsf{sqr}\ e]\!]^{\Downarrow}_{\mathsf{pre^*}} \ := \ [\![e]\!]^{\Downarrow}_{\mathsf{pre^*}} \ggg_{\mathsf{pre^*}} \mathsf{sqr_{pre^*}}$. Equivalently, we could provide $\mathsf{sqr^+}$, $\mathsf{neg}$ and $\mathsf{negative?}$ as primitives and define

$$\mathsf{sqr\ a} \ := \ \mathsf{strict\text{-}if\ (negative?\ a)\ (sqr^+\ (neg\ a))\ (sqr^+\ a)} \tag{8.37}$$

as a standard library function for probabilistic programs.

From implementations of multiplication restricted to each open quadrant, or $\mathsf{mul^{++}_{pre}}$, $\mathsf{mul^{+-}_{pre}}$, $\mathsf{mul^{-+}_{pre}}$ and $\mathsf{mul^{--}_{pre}}$, we can define multiplication on all of $\mathbb{R} \times \mathbb{R}$ with

$$\begin{aligned} \mathsf{mul_{pre}} \ := \ &\mathsf{ifte_{pre}}\ (\mathsf{fst_{pre}} \ggg_{\mathsf{pre}} \mathsf{positive?_{pre}}) \\ &\quad (\mathsf{ifte_{pre}}\ (\mathsf{snd_{pre}} \ggg_{\mathsf{pre}} \mathsf{positive?_{pre}}) \\ &\quad\quad \mathsf{mul^{++}_{pre}} \\ &\quad\quad (\mathsf{ifte_{pre}}\ (\mathsf{snd_{pre}} \ggg_{\mathsf{pre}} \mathsf{negative?_{pre}}) \\ &\quad\quad\quad \mathsf{mul^{+-}_{pre}} \\ &\quad\quad\quad (\mathsf{const_{pre}}\ 0))) \\ &\quad \text{... [similar code using } \mathsf{mul^{-+}_{pre}} \text{ and } \mathsf{mul^{--}_{pre}}\text{] ...} \\ \mathsf{mul_{pre^*}} \ := \ &\eta_{\mathsf{pre^*}}\ \mathsf{mul_{pre}} \end{aligned} \tag{8.38}$$



(a) Exact preimage of B (b) Parts sampled from a rectangular cover instead of enumerated (c) Points sampled within the parts

Figure 8.12: A more efficient alternative to the preimage refinement algorithm. Instead of enumerating a rectangular cover, its parts are sampled, and each part is then sampled from. Points that lie outside the exact preimage (which is easy to test) are rejected.

and extend $[\![\cdot]\!]^{\Downarrow}_{\mathsf{pre}^*}$ by the rule $[\![e_1 \cdot e_2]\!]^{\Downarrow}_{\mathsf{pre}^*} \ := \ [\![\langle e_1, e_2\rangle]\!]^{\Downarrow}_{\mathsf{pre}^*} \ggg_{\mathsf{pre}^*} \mathsf{mul}_{\mathsf{pre}^*}$.

## 8.4 Sampling Methods

Chapter 7 defines the preimage refinement algorithm (Definition 7.62), which repeatedly splits a partition of the program domain, restricts the program to each part, and refines each part by computing a preimage. While it appears to converge for programs that terminate with probability 1, 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 $\Omega$. Splitting each axis into only 4 disjoint intervals yields a partition of $\Omega$ of size $4^{10} = 1,048,576$.

Fortunately, Bayesian practitioners tend to be satisified with sampling methods, which are usually more efficient than enumeration methods. To approximately answer conditional queries, it suffices to sample within the preimage of the condition set. More precisely, if $\mathsf{g} := [\![p]\!]^{\Downarrow}_{\mathsf{map}^*}$, we can answer almost any query $\mathsf{Pr}[\mathsf{B}'|\mathsf{B}]$ by sampling within $\mathsf{A} := \mathsf{preimage\ g\ B}$, if the probability of A is positive.

It is easy to sample within A by sampling within $\Omega$ and rejecting samples not in A.

To determine whether samples are in $\mathsf{A}$, we can use the interpretation of the program $p$ as a bottom* arrow computation $\mathsf{f} := [\![p]\!]^{\Downarrow}_{\perp *}$. Unfortunately, the time required to accept a fixed number of samples also tends to be exponential in the number of dimensions. To solve this problem, we sample within a rectangular cover of $\mathsf{A}$, as computed by preimage refinement, instead of within $\Omega$. But we do not need to enumerate the cover's parts, as Figure 8.16 illustrates: for each sample, we first sample a part, and then sample a value within the part.

## 8.4.1 Partitioned Sampling

More generally, without considering probabilistic programming at all, we want to sample values in a probability space $\mathsf{X}, \mathsf{P}$ by first sampling a set from a *partition* of $\mathsf{X}$ and then sampling from that set.[2]

First, to restrict probability measures to measurable, positive-probability sets and renormalize them, we define

$$\begin{aligned} &\mathsf{condition} : \mathsf{Set}\ \mathsf{X} \rightharpoonup [0,1] \Rightarrow \mathsf{Set}\ \mathsf{X} \Rightarrow \mathsf{Set}\ \mathsf{X} \rightharpoonup [0,1] \\ &\mathsf{condition}\ \mathsf{P}\ \mathsf{A}\ :=\ \lambda \mathsf{A}' \in \mathsf{domain}\ \mathsf{P}.\, \mathsf{P}\ (\mathsf{A}' \cap \mathsf{A})\ /\ \mathsf{P}\ \mathsf{A} \end{aligned} \tag{8.39}$$

**Definition 8.23** (partitioned sampling)**.** *Let* $\mathsf{X}, \mathsf{P}$ *be an arbitrary probability space,* $\mathsf{N}$ *be an at-most-countable index set, and* $\mathsf{s} : \mathsf{N} \to \mathsf{Set}\ \mathsf{X}$ *be a partition of* $\mathsf{X}$ *into* $|\mathsf{N}|$ *measurable parts. The following procedure samples from* $\mathsf{X}$*:*

1. *Choose* $\mathsf{n} \in \mathsf{N}$ *with probability* $\mathsf{P}\ (\mathsf{s}\ \mathsf{n})$.
2. *Choose* $\mathsf{a} \in \mathsf{s}\ \mathsf{n}$ *according to* $\mathsf{condition}\ \mathsf{P}\ (\mathsf{s}\ \mathsf{n})$.

It is not hard to show that partitioned sampling chooses an $\mathsf{a} \in \mathsf{X}$ according to $\mathsf{P}$.

**Example 8.24** (partitioned sampling from a standard normal)**.** Let $\mathsf{P}$ be the standard normal distribution's probability measure. To sample according to $\mathsf{P}$, let $\mathsf{N} := \{\mathsf{neg}, \mathsf{pos}\}$ and

[2]This is not *stratified* sampling, which samples a fixed number of times from each partition.

$\mathsf{s} = [\mathsf{neg} \mapsto (-\infty, 0], \mathsf{pos} \mapsto (0, +\infty)]$, and define $\mathsf{Q} : \mathsf{N} \to \mathsf{Set}\ \mathbb{R} \rightharpoonup [0,1]$ by

$$\begin{aligned} \mathsf{Q\ neg\ A} &= \mathsf{P}\ ((-\infty, 0] \cap \mathsf{A})\ /\ \tfrac{1}{2} \\ \mathsf{Q\ pos\ A} &= \mathsf{P}\ ((0, +\infty) \cap \mathsf{A})\ /\ \tfrac{1}{2} \end{aligned} \tag{8.40}$$

Then

1. Choose $\mathsf{n} = \mathsf{neg}$ or $\mathsf{n} = \mathsf{pos}$, each with probability $\frac{1}{2}$.
2. Choose $\mathsf{a} \in \mathsf{s\ n}$ according to $\mathsf{Q\ n}$. ♢

Partitioned sampling has two weaknesses. First, it requires $\mathsf{P\ (s\ n)}$ to be easy to compute for all $\mathsf{n} \in \mathsf{N}$. If this were true, we would not need to sample in the first place—i.e. it assumes a solution to the overall problem we are trying to solve. Second, it assumes sampling according to $\mathsf{condition\ P\ (s\ n)}$ is easy, which is also not reasonable, as sampling according to a conditioned distribution is a subproblem we are trying to solve.

But suppose we could easily sample a partition index according to a different distribution over $\mathsf{N}$, and according to a different distribution over part $\mathsf{s\ n}$ for each $\mathsf{n} \in \mathsf{N}$. Doing so and returning weighted samples to adjust for the differences in distribution comprises ***partitioned importance sampling***.

First, to restrict a probability measure $\mathsf{P}$ to a measurable set $\mathsf{A}$ *without* renormalizing it, we define

$$\begin{aligned} &\mathsf{subcond} : \mathsf{Set\ X} \rightharpoonup [0,1] \Rightarrow \mathsf{Set\ X} \Rightarrow \mathsf{Set\ X} \rightharpoonup [0,1] \\ &\mathsf{subcond\ P\ A}\ :=\ \lambda \mathsf{A}' \in \mathsf{domain\ P}.\, \mathsf{P}\ (\mathsf{A}' \cap \mathsf{A}) \end{aligned} \tag{8.41}$$

This returns a **subprobability measure**: a measure whose largest output is less than 1.

**Definition 8.25** (partitioned importance sampling)**.** *Suppose we have*

- *An arbitrary probability space* $\mathsf{X}, \mathsf{P}$.
- *An at-most-countable index set* $\mathsf{N}$.
- *A probability mass function* $\mathsf{p} : \mathsf{N} \to [0,1]$ *such that* $\mathsf{p\ n} > 0$ *for all* $\mathsf{n} \in \mathsf{N}$.
- *A partition* $\mathsf{s} : \mathsf{N} \to \mathsf{Set\ X}$ *of* $\mathsf{X}$ *into* $|\mathsf{N}|$ *measurable parts.*
- *Candidate probability measures* $\mathsf{Q} : \mathsf{N} \to \mathsf{Set\ X} \rightharpoonup [0,1]$, *one for each partition.*

*To sample from* $\mathsf{X}$ *according to* $\mathsf{P}$*,*

1. *Choose* $\mathsf{n} \in \mathsf{N}$ *with probability* $\mathsf{p\ n}$*.*
2. *Choose* $\mathsf{a} \in \mathsf{X}$ *according to* $\mathsf{Q\ n}$*.*
3. *Compute* $\mathsf{w} := \dfrac{1}{\mathsf{p\ n}} \cdot \mathsf{diff}^{+}\ (\mathsf{subcond\ P\ (s\ n)})\ (\mathsf{Q\ n})\ \mathsf{a}$*.*
4. *Return the weighted sample* $\langle \mathsf{a}, \mathsf{w} \rangle$*.*

The function $\mathsf{diff}^{+}\ (\mathsf{subcond\ P\ (s\ n)})\ (\mathsf{Q\ n})$, with type $\mathsf{X} \to [0, +\infty)$, is a **Radon-Nikodým**[3] **derivative**. If $\mathsf{P}$ has density $\mathsf{f}$, $\mathsf{Q\ n}$ has density $\mathsf{g}$, and $\mathsf{a} \in \mathsf{s\ n}$ implies $\mathsf{g\ a} > 0$, then[4]

$$\mathsf{diff}^{+}\ (\mathsf{subcond\ P\ (s\ n)})\ (\mathsf{Q\ n})\ \mathsf{a}\ =\ \mathsf{if}\ (\mathsf{a} \in \mathsf{s\ n})\ (\mathsf{f\ a}\ /\ \mathsf{g\ a})\ 0 \tag{8.42}$$

Chapter 10 has definitions and more details. We use $\mathsf{diff}^{+}$ in a more general sense, but in this section, it is usually fine to think of its return values as quotients of densities.

An importance sampling algorithm is correct when all expected values computed using its weighted samples are the equal to the true expected values. This is true of partitioned importance sampling under reasonable conditions, which are analogous to the support of $\mathsf{subcond\ P\ (s\ n)}$ being no larger than that of $\mathsf{Q\ n}$. The formal statement of the theorm and its proof are in Chapter 10.

Partitioned importance sampling allows quite a lot of freedom: parts can be chosen with arbitrary nonzero probability, and each part can have its own candidate distribution, which may be defined on a superset of the part.

**Example 8.26** (2D partitioned importance sampling)**.** Figure 8.13 shows the result of partitioned importance sampling in a partition of the unit square. In this instance,

- $\mathsf{X} := [0, 1] \times [0, 1]$ and $\mathsf{P}$ is the uniform measure on $\mathsf{X}$ (i.e. area).
- $\mathsf{N} := \{\mathsf{left}, \mathsf{right}\}$.
- $\mathsf{p} := [\mathsf{left} \mapsto 0.4, \mathsf{right} \mapsto 0.6]$.

[3]Pronounced "RADon neekohDIM," and named after Austrian mathematician Johann Radon and Polish mathematician Otto Nikodým.

[4]The equality in (8.42) holds $(\mathsf{Q\ n})$-almost everywhere.

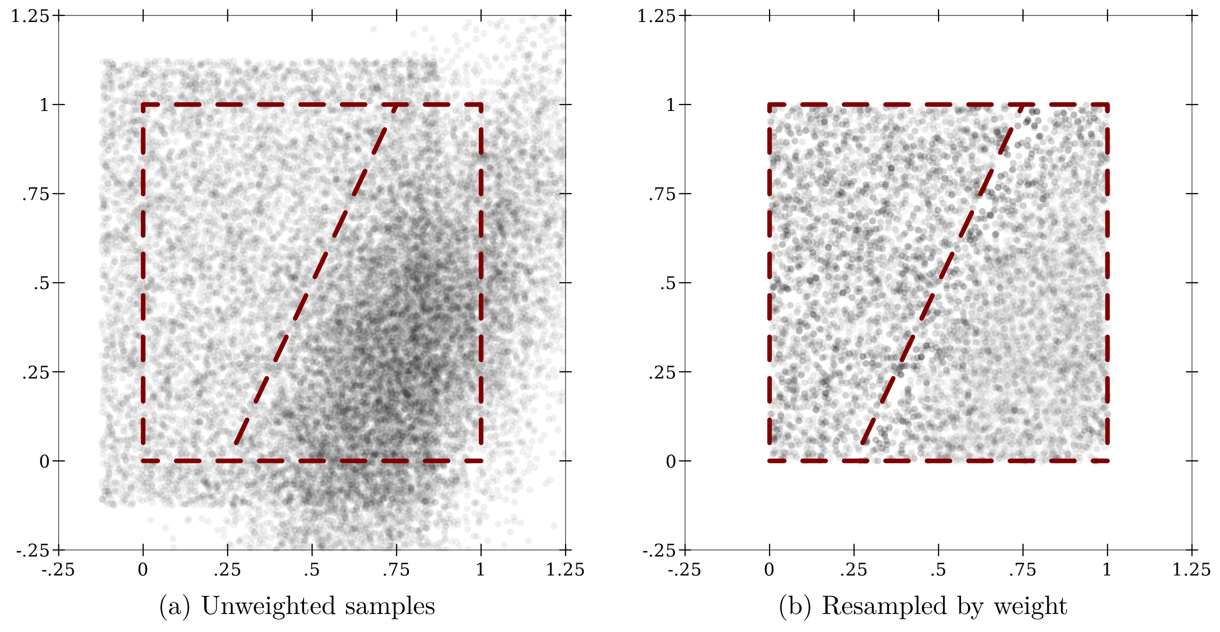


Figure 8.13: Partitioned importance sampling used to sample uniformly in a partition of the unit square, using two overlapping candidate distributions.

- $\mathsf{s\ left} = \{\langle \mathsf{x}, \mathsf{y}\rangle \in \mathsf{X} \mid \mathsf{y} > 2 \cdot \mathsf{x} - \frac{1}{2}\}$, similarly for $\mathsf{s\ right}$.
- $\mathsf{Q\ left}$ is the uniform measure on a superset of $\mathsf{s\ left}$, and $\mathsf{Q\ right}$ is a multivariate Gaussian measure centered at $\langle 0.8, 0.3\rangle$.

The implementation does not actually construct most of these objects. It constructs

- A density function $\mathsf{f} : \mathbb{R} \times \mathbb{R} \Rightarrow [0, +\infty)$ to represent $\mathsf{P}$.
- A family of predicates $\mathsf{s?} : \mathsf{N} \Rightarrow \mathsf{X} \Rightarrow \mathsf{Bool}$ to decide $\mathsf{a} \in \mathsf{s\ n}$.
- Candidate densities $\mathsf{g} : \mathsf{N} \Rightarrow \mathsf{X} \Rightarrow [0, +\infty)$ to represent $\mathsf{Q}$.

It computes weights using $\mathsf{diff}^{+}\ (\mathsf{subcond\ P\ (s\ n)})\ (\mathsf{Q\ n})\ \mathsf{a}\ =\ \mathsf{f\ a}\ /\ \mathsf{g\ n\ a}$ when $\mathsf{s?\ n\ a} = \mathsf{true}$. It directly represents only $\mathsf{N}$ and $\mathsf{p}$, but we will find even this to be infeasible shortly. $\diamondsuit$

Two properties make the preceeding example simple. First, the partition has finitely many parts. Second, the measures $\mathsf{subcond\ P\ (s\ n)}$ and $\mathsf{Q\ n}$ have densities, which ensures $\mathsf{diff}^{+}\ (\mathsf{subcond\ P\ (s\ n)})\ (\mathsf{Q\ n})$ exists and is easy to compute.

When sampling in the domain of programs, neither property holds in general.

### 8.4.2 Partitioning Probabilistic Program Domains

For the random source part $\Omega := \mathsf{J} \to [0,1]$ of probabilistic language domains, which consists of infinite binary trees of reals, it is not clear that partitioned importance sampling is applicable. The main problem is that infinite-dimensional Radon-Nikodým derivatives do not generally exist. [XXX: weaken this statement]

Fortunately, they *can* exist if the two measures differ in only finitely many axes. More precisely, let $\mathsf{P}_1 : \mathsf{Set}\ \Omega \to [0,1]$ and $\mathsf{P}_2 : \mathsf{Set}\ \Omega \to [0,1]$ be probability measures, and $\mathsf{J}' \subseteq \mathsf{J}$ be a finite set of tree indexes. Suppose $\mathsf{P}_1$ can be factored into a distribution $\mathsf{P}_1'$ over finite prefixes $\mathsf{J}' \to [0,1]$ and a distribution over suffixes $(\mathsf{J} \backslash \mathsf{J}') \to [0,1]$, and $\mathsf{P}_2'$ can be similarly factored into $\mathsf{P}_2'$ and *the same* distribution over suffixes. Then, under reasonable conditions (which are analogous to the support of $\mathsf{P}_1'$ being no larger than that of $\mathsf{P}_2'$), $\mathsf{diff}^+\ \mathsf{P}_1\ \mathsf{P}_2$ exists and can be computed using $\mathsf{diff}^+\ \mathsf{P}_1'\ \mathsf{P}_2'$. Chapter 10 contains a formal statement and proof.

To ensure $\mathsf{subcond\ P\ (s\ n)}$ and $\mathsf{Q\ n}$ differ in only finitely many axes, we partition $\Omega$ according to branch traces. Each branch trace corresponds with a program that reads any $\omega \in \Omega$ at only finitely many indexes $\mathsf{J}' \subseteq \mathsf{J}$. [XXX: connect idea better]

In the remainder of this subsection, assume a fixed program $p$. Let $\mathsf{f} := [\![p]\!]^{\Downarrow}_{\perp^*} : \langle\langle\Omega, \mathsf{T}\rangle, \langle\rangle\rangle \rightsquigarrow_{\perp^*} \mathsf{Y}$ be its interpretation as a bottom* arrow computation, with maximal domain $\mathsf{A}^*$. Define $\mathsf{T}^* := \mathsf{image\ (fst \ggg snd)\ A}^*$ as its ***maximal branch trace set*** and $\Omega^* := \mathsf{image\ (fst \ggg fst)\ A}^*$ as its ***maximal random source set***.

We need a notion of the random sources that agree with a given branch trace $\mathsf{t} \in \mathsf{T}$; i.e. those $\omega \in \Omega$ for which $\mathsf{f}\ \langle\langle\omega, \mathsf{t}\rangle, \langle\rangle\rangle \neq \perp$.

**Definition 8.27** (induced random sources)**.** *Let* $\mathsf{t} \in \mathsf{T}$ *be a branch trace. The* ***random sources induced by*** $\mathsf{t}$ *are a subset of* $\Omega$ *defined by* $\Omega' := \{\omega \in \Omega \mid \mathsf{f}\ \langle\langle\omega, \mathsf{t}\rangle, \langle\rangle\rangle \neq \perp\}$.

Equivalently, $\Omega'$ is the set of $\omega \in \Omega$ for which $\langle\langle\omega, \mathsf{t}\rangle, \langle\rangle\rangle \in \mathsf{A}^*$.

[XXX: graph of induced partition from program $\mathsf{if\ (random < random)\ true\ false}$?]

**Theorem 8.28.** *Let* $\mathsf{t} \in \mathsf{T}$ *induce* $\Omega'$. $\Omega' \neq \varnothing$ *if and only if* $\mathsf{t} \in \mathsf{T}^*$.

*Proof.* By definition of $\mathsf{T}^*$, $\Omega' \neq \varnothing$ if and only if there is an $\omega \in \Omega'$ such that $\langle\langle\omega, \mathsf{t}\rangle, \langle\rangle\rangle \in \mathsf{A}^*$. $\square$

Using $\mathsf{T}$ as the partition index set and defining the partition's parts as induced random sources *almost* works, in the sense that the required Radon-Nikodým derivatives exist. Unfortunately, we cannot use $\mathsf{T}$ or $\mathsf{T}^*$ as the partition index set because many branch traces can induce the same random sources.

**Example 8.29.** For the program

$$\mathsf{if}\ (\mathsf{random} < \mathsf{p})\ 0\ \mathsf{random} \tag{8.43}$$

there are at least two branch traces in the program's maximal domain: $\mathsf{t}_0 := [\mathsf{j}_0 \mapsto \mathsf{true}, * \mapsto \bot]$ and $\mathsf{t}_1 := [\mathsf{j}_0 \mapsto \mathsf{false}, * \mapsto \bot]$. There is also $[\mathsf{j}_0 \mapsto \mathsf{false}, \mathsf{left}\ \mathsf{j}_0 \mapsto \mathsf{true}, * \mapsto \bot]$, because it agrees with every execution that $[\mathsf{j}_0 \mapsto \mathsf{false}, * \mapsto \bot]$ agrees with. In fact, there are infinitely many branch traces in $\mathsf{T}^*$ that induce the same random sources as either $\mathsf{t}_0$ or $\mathsf{t}_1$. [XXX: tree diagrams?] $\diamondsuit$

We need to find a subset of branch traces whose induced random sources are disjoint. The main idea is to define equivalence classes of branch traces that induce the same random sources, and use the "smallest" branch trace in each class as a part index.

To identify the "smallest" trace in each class, we must define an ordering over them. One fairly natural way is to say a branch trace is smaller than another when it describes fewer branch decisions; i.e. its tree has fewer non-$\bot$ elements. Two branch traces that differ by returning respectively $\mathsf{true}$ and $\mathsf{false}$ for the same $\mathsf{j}$ may represent different execution paths, so they must be incomparable.

**Definition 8.30** (brach trace partial order)**.** $\mathsf{t}_1 \leq \mathsf{t}_2$ *when for all* $\mathsf{j} \in \mathsf{J}$*,* $\mathsf{t}_1\ \mathsf{j} = \bot$ *or* $\mathsf{t}_1\ \mathsf{j} = \mathsf{t}_2\ \mathsf{j}$*.*

To find the minimum of a set of equivalent traces, it helps to be able to compute the greatest lower bound, or infimum. We claim that this function does so:

$$\begin{aligned} &\mathsf{trace\text{-}inf} : \mathsf{Set}\ \mathsf{T} \Rightarrow \mathsf{T} \\ &\mathsf{trace\text{-}inf}\ \mathsf{T'} \ := \ \lambda \mathsf{j} \in \mathsf{J}.\,\mathsf{case}\ \ \mathsf{proj}\ \mathsf{j}\ \mathsf{T'} \\ &\qquad\qquad\qquad\qquad\qquad \{\mathsf{b}\} \ \longrightarrow \ \mathsf{b} \\ &\qquad\qquad\qquad\qquad\qquad \mathsf{else} \ \longrightarrow \ \bot \end{aligned} \tag{8.44}$$

**Theorem 8.31** (trace infimum)**.** *Let* $\mathsf{T'} \subseteq \mathsf{T}$ *and* $\mathsf{t}_* := \mathsf{trace\text{-}inf}\ \mathsf{T'}$*. Then*

- *For all* $\mathsf{t'} \in \mathsf{T'}$*,* $\mathsf{t}_* \leq \mathsf{t'}$*.*
- *For all* $\mathsf{t} \in \mathsf{T}$*, if for all* $\mathsf{t'} \in \mathsf{T'}$*,* $\mathsf{t} \leq \mathsf{t'}$*, then* $\mathsf{t} \leq \mathsf{t}_*$*.*

*Proof.* Let $\mathsf{t'} \in \mathsf{T'}$ and $\mathsf{j} \in \mathsf{J}$. If $\mathsf{proj}\ \mathsf{j}\ \mathsf{T'} = \{\mathsf{b}\}$ for $\mathsf{b} \in \mathsf{Bool}_\bot$, then $\mathsf{t}_*\ \mathsf{j} = \mathsf{t'}\ \mathsf{j} = \mathsf{b}$. Otherwise, $\mathsf{t}_*\ \mathsf{j} = \bot$. Thus $\mathsf{t}_* \leq \mathsf{t'}$.

Let $\mathsf{t} \in \mathsf{T}$ and suppose that for all $\mathsf{t'} \in \mathsf{T'}$, $\mathsf{t} \leq \mathsf{t'}$. Let $\mathsf{j} \in \mathsf{J}$. If $\mathsf{proj}\ \mathsf{j}\ \mathsf{T'} = \{\mathsf{b}\}$, then there are two cases: $\mathsf{t}\ \mathsf{j} = \bot$, or $\mathsf{t}\ \mathsf{j} = \mathsf{t}_*\ \mathsf{j} = \mathsf{b}$. Otherwise there exists a $\mathsf{t'} \in \mathsf{T'}$ such that $\mathsf{t}\ \mathsf{j} \neq \mathsf{t'}\ \mathsf{j}$, so $\mathsf{t}\ \mathsf{j} = \bot$. Thus $\mathsf{t} \leq \mathsf{t}_*$. □

Any two comparable traces in $\mathsf{T}^*$ induce the same random sources.

**Theorem 8.32** (comparable implies equivalent)**.** *Let* $\mathsf{t}_1, \mathsf{t}_2 \in \mathsf{T}^*$ *induce* $\Omega_1$*,* $\Omega_2$*. If* $\mathsf{t}_1 \leq \mathsf{t}_2$ *or* $\mathsf{t}_2 \leq \mathsf{t}_1$*, then* $\Omega_1 = \Omega_2$*.*

*Proof.* It suffices to consider $\mathsf{t}_1 \leq \mathsf{t}_2$; the $\mathsf{t}_2 \leq \mathsf{t}_1$ case follows from reflexivity of $(=)$.

Suppose $\omega \in \Omega_1$, so $\mathsf{f}\ \langle\langle\omega, \mathsf{t}_1\rangle, \langle\rangle\rangle \neq \bot$. Let $\mathsf{J'} \subseteq \mathsf{J}$ such that $\mathsf{j} \in \mathsf{J'}$ if and only if $\langle\omega, \mathsf{t}_1\rangle$ agrees with the $\mathsf{ifte}^{\Downarrow}_{\bot *}$ subcomputation at index $\mathsf{j}$. For all $\mathsf{j} \in \mathsf{J'}$, $\mathsf{t}_1\ \mathsf{j} \neq \bot$, so $\mathsf{t}_2\ \mathsf{j} = \mathsf{t}_1\ \mathsf{j}$ by definition of $(\leq)$. Therefore, $\langle\omega, \mathsf{t}_2\rangle$ also agrees with every $\mathsf{ifte}^{\Downarrow}_{\bot *}$ subcomputation at every index $\mathsf{j} \in \mathsf{J'}$, so $\mathsf{f}\ \langle\langle\omega, \mathsf{t}_2\rangle, \langle\rangle\rangle \neq \bot$. Therefore $\omega \in \Omega_2$, so $\Omega_1 \subseteq \Omega_2$.

Suppose $\omega \notin \Omega_1$. Let $\mathsf{J'} \subseteq \mathsf{J}$ such that $\mathsf{j} \in \mathsf{J'}$ if and only if $\mathsf{t}_1\ \mathsf{j} \neq \bot$. Because $\mathsf{f}\ \langle\langle\omega, \mathsf{t}_1\rangle, \langle\rangle\rangle = \bot$, there exists a $\mathsf{j} \in \mathsf{J'}$ such that the $\mathsf{ifte}^{\Downarrow}_{\bot *}$ subcomputation at index $\mathsf{j}$ disagrees with $\langle\omega, \mathsf{t}_1\rangle$. Because $\mathsf{t}_1\ \mathsf{j} = \mathsf{t}_2\ \mathsf{j}$ by definition of $(\leq)$, the $\mathsf{ifte}^{\Downarrow}_{\bot *}$ subcomputation at index $\mathsf{j}$ also disagrees with $\langle\omega, \mathsf{t}_2\rangle$, so $\mathsf{f}\ \langle\langle\omega, \mathsf{t}_2\rangle, \langle\rangle\rangle = \bot$. Therefore $\omega \notin \Omega_2$, so $\Omega_2 \subseteq \Omega_1$. □

**Corollary 8.33** (infimum in $\mathsf{T}^*$ implies equivalent)**.** *Let* $\mathsf{t}_1, \mathsf{t}_2 \in \mathsf{T}^*$ *induce* $\Omega_1$,$\Omega_2$*, and define* $\mathsf{t}_* := \mathsf{trace\text{-}inf}\ \{\mathsf{t}_1, \mathsf{t}_2\}$*. If* $\mathsf{t}_* \in \mathsf{T}^*$*, then* $\Omega_1 = \Omega_2$*.*

If $\mathsf{T}^*$ is partitioned into equivalence classes of traces that induce the same random sources, each part in the partition contains a smallest member with respect to $(\leq)$.

**Theorem 8.34.** *Let* $\mathsf{t} \in \mathsf{T}^*$ *induce* $\Omega'$*,* $\mathsf{T}'$ *be the largest subset of* $\mathsf{T}^*$ *that induces* $\Omega'$*, and* $\mathsf{t}_* := \mathsf{trace\text{-}inf}\ \mathsf{T}'$*. Then* $\mathsf{t}_* \in \mathsf{T}'$*.*

*Proof.* Let $\omega \in \Omega'$. By definition of $\mathsf{trace\text{-}inf}$, every $\mathsf{ifte}^{\Downarrow}_{\perp *}$ subcomputation agrees with $\langle \omega, \mathsf{t}_* \rangle$. Therefore $\mathsf{f}\ \langle \langle \omega, \mathsf{t}_* \rangle, \langle \rangle \rangle \neq \perp$, so $\mathsf{t}_*$ induces $\Omega'$. □

**Theorem 8.35** (infimum not in $\mathsf{T}^*$ implies disjoint)**.** *Let* $\mathsf{t}_1, \mathsf{t}_2 \in \mathsf{T}^*$ *induce* $\Omega_1$,$\Omega_2$*, and define* $\mathsf{t}_* := \mathsf{trace\text{-}inf}\ \{\mathsf{t}_1, \mathsf{t}_2\}$*. If* $\mathsf{t}_* \notin \mathsf{T}^*$*, then* $\Omega_1 \cap \Omega_2 = \varnothing$*.*

*Proof.* Let $\mathsf{T}_1$,$\mathsf{T}_2$ be the largest subsets of $\mathsf{T}^*$ that induce $\Omega_1$, $\Omega_2$. Let $\mathsf{t}_{1*} := \mathsf{trace\text{-}inf}\ \mathsf{T}_1$ and $\mathsf{t}_{2*} := \mathsf{trace\text{-}inf}\ \mathsf{T}_2$. Because $\mathsf{t}_* \notin \mathsf{T}^*$, $\mathsf{t}_{1*} \neq \mathsf{t}_{2*}$.

Let $\omega \in \Omega_1$. For every $\mathsf{j} \in \mathsf{J}$ for which $\mathsf{t}_{1*}\ \mathsf{j} \neq \perp$, there is an $\mathsf{ifte}^{\Downarrow}_{\perp *}$ subcomputation at index $\mathsf{j}$ that agrees with $\langle \omega, \mathsf{t}_{1*} \rangle$. But because $\mathsf{t}_{2*} \neq \mathsf{t}_{1*}$, there exists a $\mathsf{j} \in \mathsf{J}$ for which an $\mathsf{ifte}^{\Downarrow}_{\perp *}$ subcomputation at index $\mathsf{j}$ disagrees with $\langle \omega, \mathsf{t}_{2*} \rangle$. Therefore $\omega \notin \Omega_2$. By a symmetric argument, $\omega \in \Omega_2$ implies $\omega \notin \Omega_1$. □

We can get our sought-after index set by defining the set of smallest branch traces.

**Definition 8.36** (minimal branch traces)**.** *The set of* ***minimal branch traces*** $\mathsf{T}_*$ *is the set of minimal elements in* $\mathsf{T}^*$*, or*

$$\mathsf{T}_* \ := \ \{\mathsf{t}_1 \in \mathsf{T}^* \mid \forall \mathsf{t}_2 \in \mathsf{T}^*.\, \mathsf{t}_2 \leq \mathsf{t}_1 \implies \mathsf{t}_2 = \mathsf{t}_1\} \tag{8.45}$$

**Theorem 8.37.** $\mathsf{T}_*$ *induces* $\Omega^*$*.*

*Proof.* Let $\mathsf{t} \in \mathsf{T}^*$ induce $\Omega'$ and $\mathsf{T}'$ be the largest subset of $\mathsf{T}^*$ that induces $\Omega'$. Its minimum $\mathsf{t}_* := \mathsf{trace\text{-}inf}\ \mathsf{T}'$ is in $\mathsf{T}_*$. □

**Theorem 8.38** ($\mathsf{T}_*$ partitions)**.** *Let* $\mathsf{t}_1, \mathsf{t}_2 \in \mathsf{T}_*$ *induce respectively* $\Omega_1$ *and* $\Omega_2$*. If* $\mathsf{t}_1 \neq \mathsf{t}_2$*, then* $\Omega_1 \cap \Omega_2 = \varnothing$*.*

*Proof.* Let $\mathsf{t}_* := \mathsf{trace\text{-}inf}\ \{\mathsf{t}_1, \mathsf{t}_2\}$. Because $\mathsf{t}_1$ and $\mathsf{t}_2$ are minimal, $\mathsf{t}_* \notin \mathsf{T}^*$. By Theorem XXX, $\Omega_1 \cap \Omega_2 = \varnothing$. □

We can thus sample a partition index $\mathsf{t} \in \mathsf{T}_*$, which induces a unique part from a partition of $\Omega^*$.

A program's minimal branch trace set $\mathsf{T}_*$ contains only the actual branches taken when running the program on every $\omega \in \Omega^*$. Therefore, one way to sample from $\mathsf{T}_*$ with the correct probability—at least, for programs that halt with probability 1—would be to choose an $\omega \in \Omega$ uniformly, and run the program on $\omega$ while recording each branch decision.

But this sampling scheme has problems similar to those of partitioned sampling (Definition 8.23). First, it assumes the probabilities of branch traces, which are the probabilities of the $\Omega^*$ subsets they induce, are easy to compute. Second, we are interested in sampling from an *arbitrarily low-probability subset* of $\Omega^*$, which may be covered by the partition induced by an *arbitrarily low-probability subset* of $\mathsf{T}_*$.

It appears we have a chicken-and-egg problem, in that

1. Sampling in a small subset of $\Omega^*$ requires sampling in a small subset of $\mathsf{T}_*$.
2. Sampling in a small subset of $\mathsf{T}_*$ requires sampling in a small subset of $\Omega^*$.

Fortunately, if we allow ourselves subsets of a larger set than $\mathsf{T}_*$, and allow ourselves to sample within overapproximating *covers* of $\Omega^*$ subsets, we can use approximate preimage computation to sample from $\mathsf{T}_*$ and $\Omega^*$ subsets simultaneously.

### 8.4.3 Approximate Partitions of Probabilistic Program Domains

The idea is to define a set of branch traces $\mathsf{T}_+$ that is derived only from the *shape* of a program, not its actual executions. We ensure that $\mathsf{T}_* \subseteq \mathsf{T}_+$, and that every $\mathsf{t} \in \mathsf{T}_+ \backslash \mathsf{T}_*$ induces $\varnothing$, so that $\mathsf{T}_+$ induces the same partition as $\mathsf{T}_*$. We define an algorithm for sampling from $\mathsf{T}_+$,

$$\begin{aligned}
\mathsf{Idxs}\ \ &::=\ \langle\rangle \mid \langle \mathsf{Idxs}, \mathsf{Idxs}\rangle \\
&\ \mid\ \mathsf{if\text{-}idxs}\ \mathsf{J}\ (1 \Rightarrow \mathsf{Idxs})\ (1 \Rightarrow \mathsf{Idxs})
\end{aligned}$$

$$\begin{aligned}
&\mathsf{arr}_{\mathsf{idxs}^*} : (\mathsf{x} \Rightarrow \mathsf{y}) \Rightarrow (\mathsf{J} \Rightarrow \mathsf{Idxs}) \\
&\mathsf{arr}_{\mathsf{idxs}^*}\ \mathsf{f}\ \mathsf{j}\ :=\ \langle\rangle
\end{aligned}$$

$$\begin{aligned}
&(\ggg_{\mathsf{idxs}^*}) : (\mathsf{J} \Rightarrow \mathsf{Idxs}) \Rightarrow (\mathsf{J} \Rightarrow \mathsf{Idxs}) \Rightarrow (\mathsf{J} \Rightarrow \mathsf{Idxs}) \\
&(\mathsf{i}_1 \ggg_{\mathsf{idxs}^*} \mathsf{i}_2)\ \mathsf{j}\ :=\ \langle \mathsf{i}_1\ (\mathsf{left}\ \mathsf{j}), \mathsf{i}_2\ (\mathsf{right}\ \mathsf{j})\rangle
\end{aligned}$$

$$\begin{aligned}
&(\&\&\&_{\mathsf{idxs}^*}) : (\mathsf{J} \Rightarrow \mathsf{Idxs}) \Rightarrow (\mathsf{J} \Rightarrow \mathsf{Idxs}) \Rightarrow (\mathsf{J} \Rightarrow \mathsf{Idxs}) \\
&(\mathsf{i}_1 \ggg_{\mathsf{idxs}^*} \mathsf{i}_2)\ \mathsf{j}\ :=\ \langle \mathsf{i}_1\ (\mathsf{left}\ \mathsf{j}), \mathsf{i}_2\ (\mathsf{right}\ \mathsf{j})\rangle
\end{aligned}$$

$$\begin{aligned}
&\mathsf{ifte}_{\mathsf{idxs}^*} : (\mathsf{J} \Rightarrow \mathsf{Idxs}) \Rightarrow (\mathsf{J} \Rightarrow \mathsf{Idxs}) \Rightarrow (\mathsf{J} \Rightarrow \mathsf{Idxs}) \Rightarrow (\mathsf{J} \Rightarrow \mathsf{Idxs}) \\
&\mathsf{ifte}_{\mathsf{idxs}^*}\ \mathsf{i}_1\ \mathsf{i}_2\ \mathsf{i}_3\ \mathsf{j}\ :=\ \mathsf{let}\ \ \mathsf{idxs}_2 := \lambda 0.\, \mathsf{i}_2\ (\mathsf{left}\ (\mathsf{right}\ \mathsf{j})) \\
&\qquad\qquad\qquad\qquad\quad\ \ \mathsf{idxs}_3 := \lambda 0.\, \mathsf{i}_3\ (\mathsf{right}\ (\mathsf{right}\ \mathsf{j})) \\
&\qquad\qquad\qquad\qquad\ \ \mathsf{in}\ \ \langle \mathsf{i}_1\ \mathsf{j}, \mathsf{if\text{-}idxs}\ \mathsf{j}\ \mathsf{idxs}_2\ \mathsf{idxs}_3\rangle
\end{aligned}$$

$$\begin{aligned}
&\mathsf{lazy}_{\mathsf{idxs}^*} : (1 \Rightarrow (\mathsf{J} \Rightarrow \mathsf{Idxs})) \Rightarrow (\mathsf{J} \Rightarrow \mathsf{Idxs}) \\
&\mathsf{lazy}_{\mathsf{idxs}^*}\ \mathsf{i}\ \mathsf{j}\ :=\ \mathsf{i}\ 0\ \mathsf{j}
\end{aligned}$$

$$\begin{aligned}
&\mathsf{random}_{\mathsf{idxs}^*} : \mathsf{J} \Rightarrow \mathsf{Idxs} \\
&\mathsf{random}_{\mathsf{idxs}^*}\ \mathsf{j}\ :=\ \langle\rangle
\end{aligned}$$

(a) Branch index arrow. Computations return a lazy tree of type Idxs, of feasible branch decisions, ignoring the actual values of if conditions. The arrow is directly implementable in any $\lambda$-calculus.

$$\begin{aligned}
&\mathsf{sample\text{-}traces} : \mathsf{Idxs} \to \langle \mathbb{R}, \mathsf{Rect}\ \mathsf{T}\rangle \\
&\mathsf{sample\text{-}traces}\ \mathsf{idxs}\ :=\ \mathsf{sample\text{-}traces}^*\ \mathsf{idxs}\ \langle 1, \mathsf{T}\rangle \\[1ex]
&\mathsf{sample\text{-}traces}^* : \mathsf{Idxs} \Rightarrow \langle \mathbb{R}, \mathsf{Rect}\ \mathsf{T}\rangle \Rightarrow \langle \mathbb{R}, \mathsf{Rect}\ \mathsf{T}\rangle \\
&\begin{array}{lll}
\mathsf{sample\text{-}traces}^*\ \langle\rangle\ \mathsf{pt} & := & \mathsf{pt} \\
\mathsf{sample\text{-}traces}^*\ \langle \mathsf{idxs}_1, \mathsf{idxs}_2\rangle\ \mathsf{pt} & := & \mathsf{let}\ \ \mathsf{pt}' := \mathsf{sample\text{-}traces}^*\ \mathsf{idxs}_1\ \mathsf{pt} \\
 & & \ \ \mathsf{in}\ \ \mathsf{sample\text{-}traces}^*\ \mathsf{idxs}_2\ \mathsf{pt}' \\
\mathsf{sample\text{-}traces}^*\ (\mathsf{if\text{-}idxs}\ \mathsf{j}\ \mathsf{idxs}_2\ \mathsf{idxs}_3)\ \langle \mathsf{p_t}, \mathsf{T}'\rangle & := & \mathsf{let}\ \ \langle \mathsf{p_b}, \mathsf{b}\rangle := \mathsf{sample\text{-}branch}\ \mathsf{Bool}_\perp \\
 & & \qquad\ \mathsf{pt}' := \langle \mathsf{p_t} \cdot \mathsf{p_b}, \mathsf{unproj}\ \mathsf{j}\ \mathsf{T}'\ \{\mathsf{b}\}\rangle \\
 & & \ \ \mathsf{in}\ \ \mathsf{case}\ \ \mathsf{b} \\
 & & \qquad \begin{array}{lll}
 \mathsf{true} & \longrightarrow & \mathsf{sample\text{-}traces}^*\ (\mathsf{idxs}_2\ 0)\ \mathsf{pt}' \\
 \mathsf{false} & \longrightarrow & \mathsf{sample\text{-}traces}^*\ (\mathsf{idxs}_3\ 0)\ \mathsf{pt}' \\
 \perp & \longrightarrow & \mathsf{pt}'
 \end{array}
\end{array}
\end{aligned} \tag{8.46}$$

(b) The stochastic function sample-traces samples a $\mathsf{T}'$, and returns $\mathsf{T}'$ and its probability.

Figure 8.14: Branch index collecting semantics.

which does not require running a probabilistic program on any $\omega \in \Omega$. We then extend this algorithm to use preimage computation to sample in arbitrarily good approximations of small subsets of $\mathsf{T}_*$ and $\Omega^*$.

Defining $\mathsf{T}_+$ in terms of a program's branching shape requires an additional abstract interpretation. Figure 8.14a defines the ***indexes arrow***. Its type is $\mathsf{J} \Rightarrow \mathsf{Idxs}$, which does not refer to a domain or codomain type of program values because its computations do not receive or compute program values. Instead, they build lazy trees of possible branching decisions, ignoring the actual values of if conditions. For example, lifted, pure functions are interpreted as $\lambda \mathsf{j}.\, \langle\rangle$, which takes the function's computation index and returns no decisions.

Composition and pairing of subcomputations $\mathsf{i}_1$ and $\mathsf{i}_2$ both return $\langle \mathsf{i}_1\ (\mathsf{left\ j}), \mathsf{i}_2\ (\mathsf{right\ j})\rangle$: a node with two children that contain the feasible branch decisions in their subcomputations.

Only $\mathsf{ifte}_{\mathsf{idxs}^*}$ does more than simple structural recursion: it returns $\mathsf{if\text{-}idxs\ j\ idxs}_2\ \mathsf{idxs}_3$ to represent a decision at computation index $\mathsf{j}$. The children $\mathsf{idxs}_2$ and $\mathsf{idxs}_3$ are lazy, abstract representations of the $\mathsf{if}$'s branches. Like a concrete execution, a branch trace sampler is expected to compute and recur through only one of them.

Figure 8.14b defines sample-traces, which, to make its extension for use with preimage refinement easier, samples *rectangles* of branch traces given an $\mathsf{idxs} : \mathsf{Idxs}$. It returns a pair $\langle \mathsf{p_t}, \mathsf{T}'\rangle$, where $\mathsf{T}'$ is the sampled rectangle and $\mathsf{p_t}$ is the probability with which it was chosen. It assumes a stochastic procedure $\mathsf{sample\text{-}branch} : \mathsf{Set\ Bool}_\bot \Rightarrow \langle \mathbb{R}, \mathsf{Bool}_\bot\rangle$, where sample-branch B returns any member of B with some nonzero, constant probability. At index $\mathsf{j}$, for the branch choice $\langle \mathsf{p_b}, \mathsf{b}\rangle := \mathsf{sample\text{-}branch\ Bool}_\bot$, $\mathsf{T}'$ is restricted using $\mathsf{unproj\ j\ T'}\ \{\mathsf{b}\}$.

Although sample-traces returns rectangles, it is easy to transform one into a single trace using trace-inf; i.e. trace-inf (snd (sample-traces idxs)) samples a branch trace.

Let $\mathsf{idxs} := [\![p]\!]^{\Downarrow}_{\mathsf{idxs}^*}\ \mathsf{j}_0$.

**Definition 8.39** (feasible branch traces). *The **feasible branch traces*** $\mathsf{T}_+$ *are those* $\mathsf{t} \in \mathsf{T}$ *for which* $\Pr[\mathsf{t} = \mathsf{trace\text{-}inf\ (snd\ (sample\text{-}traces\ idxs))}] > 0$.

Because sample-traces* imposes a total order on evaluation, any terminating application of it induces a total order on the indexes $\mathsf{j}$ in applications matching $\mathsf{if\text{-}idxs\ j\ idxs}_2\ \mathsf{idxs}_3$. Let $\mathsf{j}_1, \mathsf{j}_2, ..., \mathsf{j_n}$ be those indexes, with corresponding branch choices $\mathsf{b}_1, \mathsf{b}_2, ..., \mathsf{b_n}$. Define $\mathsf{T}'_1, \mathsf{T}'_2, ..., \mathsf{T}'_\mathsf{n}$ by $\mathsf{T}'_0 := \mathsf{T}$ and $\mathsf{T}'_\mathsf{i} := \mathsf{unproj\ j_i\ T'_{i-1}}\ \{\mathsf{b_i}\}$.

**Theorem 8.40** (sample-traces soundness). $\mathsf{T}_* \subseteq \mathsf{T}_+$.

*Proof.* Let $\mathsf{t} \in \mathsf{T}_*$. It suffices to show that there exists an $\mathsf{n}$ and a sequence of branch choices $\mathsf{b}_1, \mathsf{b}_2, ..., \mathsf{b_n}$ for which $\mathsf{t} = \mathsf{trace\text{-}inf\ T'_n}$.

First, we prove by induction the seemingly weaker statement that there exist $\mathsf{n}$ and branch choices for which $\mathsf{t} \in \mathsf{T}'_\mathsf{n}$. Let $\mathsf{j}_1, \mathsf{j}_2, ..., \mathsf{j_n}$ be the in-order indexes at which $\mathsf{t\ j_i} \neq \bot$.

Clearly $\mathsf{t} \in \mathsf{T}'_0 = \mathsf{T}$. If $\mathsf{t} \in \mathsf{T}'_{\mathsf{i}-1}$, then $\mathsf{b_i} := \mathsf{t}\ \mathsf{j_i}$ implies $\mathsf{t} \in \mathsf{T}'_\mathsf{i} = \mathsf{unproj}\ \mathsf{j_i}\ \mathsf{T}'_{\mathsf{i}-1}\ \{\mathsf{b_i}\}$.

For any $\mathsf{j} \in \{\mathsf{j}_1, \mathsf{j}_2, ..., \mathsf{j_n}\}$, $\{\mathsf{t}\ \mathsf{j}\} = \mathsf{proj}\ \mathsf{j}\ \mathsf{T}'_\mathsf{n}$. For any other $\mathsf{j}$, $\mathsf{t}\ \mathsf{j} = \bot$ and $\mathsf{proj}\ \mathsf{j}\ \mathsf{T}'_\mathsf{n} = \mathsf{Bool}_\bot$. By definition of trace-inf, therefore $\mathsf{t} = \mathsf{trace\text{-}inf}\ \mathsf{T}'_\mathsf{n}$. □

For $\mathsf{T}_+$ to induce a partition, every $\mathsf{t} \in \mathsf{T}_+ \backslash \mathsf{T}_*$ must induce $\varnothing$.

**Theorem 8.41** (sample-traces non-$\varnothing$-unique)**.** *For all* $\mathsf{t} \in \mathsf{T}_+$*, if* $\mathsf{t} \notin \mathsf{T}_*$ *then* $\mathsf{t}$ *induces* $\varnothing$*.*

*Proof.* Let $\mathsf{j}_1, \mathsf{j}_2, ..., \mathsf{j_n}$ and $\mathsf{b}_1, \mathsf{b}_2, ..., \mathsf{b_n}$ for a terminating evaluation of $\mathsf{sample\text{-}traces}^*\ \mathsf{idxs}\ \langle 1, \mathsf{T}\rangle$.

Suppose $\mathsf{T}'_\mathsf{n} \cap \mathsf{T}_* = \varnothing$. Then there exists an $\mathsf{i}$ such that $\mathsf{T}'_{\mathsf{i}-1} \cap \mathsf{T}_* \neq \varnothing$ and $\mathsf{T}'_\mathsf{i} \cap \mathsf{T}_* = \varnothing$. Thus $\mathsf{b_i} \notin \mathsf{proj}\ \mathsf{j_i}\ \mathsf{T}_*$, so $\mathsf{f}$ does not agree with any $\mathsf{t} \in \mathsf{T}'_\mathsf{i}$.

Let $\mathsf{t} := \mathsf{trace\text{-}inf}\ \mathsf{T}'_\mathsf{n}$, which by definition of trace-inf and sample-traces$^*$ is in $\mathsf{T}'_\mathsf{n}$. Because $\mathsf{T}'_\mathsf{n} \subseteq \mathsf{T}'_\mathsf{i}$, $\mathsf{f}$ does not agree with $\mathsf{t}$, so $\mathsf{t}$ induces $\varnothing$. □

**Corollary 8.42** (sample-traces partitioning)**.** $\mathsf{T}_+$ *induces a partition of* $\Omega^*$*.*

To be used for partitioned importance sampling, the probability returned by sample-traces must be correct.

**Theorem 8.43** (sample-traces correctness)**.** *If* $\langle \mathsf{p}'_\mathsf{t}, \mathsf{T}'\rangle := \mathsf{sample\text{-}traces}\ \mathsf{idxs}$*, then* $\mathsf{Pr}[\mathsf{T}'] = \mathsf{p}'_\mathsf{t}$*.*

*Proof.* Let $\mathsf{p}_{\mathsf{b}_1}, \mathsf{p}_{\mathsf{b}_2}, ..., \mathsf{p}_{\mathsf{b_n}}$ be the probabilities returned from sample-branch for $\mathsf{b}_1, \mathsf{b}_2, ..., \mathsf{b_n}$. The probability of $\mathsf{b}_1, \mathsf{b}_2, ..., \mathsf{b_n}$ is thus $\mathsf{p}'_\mathsf{t} := \mathsf{p}_{\mathsf{b}_1} \cdot \mathsf{p}_{\mathsf{b}_2} \cdot ... \cdot \mathsf{p}_{\mathsf{b_n}}$. Because the transformation from $\mathsf{b}_1, \mathsf{b}_2, ..., \mathsf{b_n}$ to $\mathsf{T}'_\mathsf{n}$ is injective, $\mathsf{Pr}[\mathsf{T}'_\mathsf{n}] = \mathsf{p}'_\mathsf{t}$. □

Further, sample-traces should terminate with probability 1.

**Theorem 8.44** (sample-traces termination)**.** sample-traces idxs *terminates with probability* 1*.*

*Proof.* For each branch choice $\mathsf{b_i}$, there is a nonzero probability that $\mathsf{b_i} = \bot$, which is a recursion base case. □

We finally have a way to use partitioned importance sampling to sample within the preimage of some set $\mathsf{B}$. Define

$$\mathsf{f}\ :=\ [\![p]\!]^{\Downarrow}_{\bot^*}\ \mathsf{j}_0 \qquad \mathsf{h}'\ :=\ [\![p]\!]^{\Downarrow'}_{\mathsf{pre}^*}\ \mathsf{j}_0 \qquad \mathsf{idxs}\ :=\ [\![p]\!]^{\Downarrow}_{\mathsf{idxs}^*}\ \mathsf{j}_0 \tag{8.47}$$

```
Idxs ::= ⟨⟩ | ⟨Idxs, Idxs⟩
          | if-idxs J (1 ⇒ Idxs) (1 ⇒ Idxs)
          | random-idxs J

arr_idxs* : (x ⇒ y) ⇒ (J ⇒ Idxs)
arr_idxs* f j  :=  ⟨⟩

(⋙_idxs*) : (J ⇒ Idxs) ⇒ (J ⇒ Idxs) ⇒ (J ⇒ Idxs)
(i1 ⋙_idxs* i2) j  :=  ⟨i1 (left j), i2 (right j)⟩

(&&&_idxs*) : (J ⇒ Idxs) ⇒ (J ⇒ Idxs) ⇒ (J ⇒ Idxs)
(i1 ⋙_idxs* i2) j  :=  ⟨i1 (left j), i2 (right j)⟩

ifte_idxs* : (J ⇒ Idxs) ⇒ (J ⇒ Idxs) ⇒ (J ⇒ Idxs) ⇒ (J ⇒ Idxs)
ifte_idxs* i1 i2 i3 j  :=  let  idxs2 := λ0. i2 (left (right j))
                                idxs3 := λ0. i3 (right (right j))
                           in  ⟨i1 j, if-idxs j idxs2 idxs3⟩

lazy_idxs* : (1 ⇒ (J ⇒ Idxs)) ⇒ (J ⇒ Idxs)
lazy_idxs* i j  :=  i 0 j

random_idxs* : J ⇒ Idxs
random_idxs* j  :=  random-idxs j
```

Figure 8.15: The final indexes arrow, which collects information about feasible branches and random choices.

to interpret $p$ as a bottom arrow computation, an approximating preimage arrow computation, and a lazy tree of feasible branch decisions. Define refine $\mathsf{A} := \mathsf{ap}'_{\mathsf{pre}^*}\ (\mathsf{h}'\ \mathsf{A})\ \mathsf{B}$. Then

1. Let $\langle \mathsf{p_t}, \mathsf{T}' \rangle :=$ sample-traces idxs.
2. Let $\mathsf{t} :=$ trace-inf $\mathsf{T}'$.
3. Let $\mathsf{A}' :=$ refine $((\Omega \times \{\mathsf{t}\}) \times \{\langle\rangle\})$.
4. Let $\Omega' :=$ image (fst ⋙ fst) $\mathsf{A}'$. If $\Omega' = \varnothing$, reject.
5. Choose $\omega \in \Omega'$ according to Q t. If $\mathsf{f}\ \langle\langle \omega, \mathsf{t}\rangle, \langle\rangle\rangle \notin \mathsf{B}$, reject.
6. Compute weight $\mathsf{w} := \dfrac{1}{\mathsf{p_t}} \cdot \mathsf{diff}^+\ (\mathsf{subcond}\ \mathsf{P}\ \Omega'')\ (\mathsf{Q}\ \mathsf{t})\ \omega$, where $\Omega''$ is the set of random sources induced by t.

Computing $\mathsf{diff}^+\ (\mathsf{subcond}\ \mathsf{P}\ \Omega'')\ (\mathsf{Q}\ \mathsf{t})$ does not require $\Omega''$, as we will demonstrate shortly.

Samples are rejected for two reasons. The first is when $\Omega' = \varnothing$ because sample-trace overapproximates. The second is when $\omega \in \Omega'$ but $\omega \notin \Omega''$ because h′ overapproximates. To reduce the rejection rate, we must reduce overapproximation as much as possible. We can address both causes by partitioning $\Omega$ more finely than the partition induced by branch traces. Doing so requires an update to the indexes arrow and another sampling algorithm.

Figure 8.15 shows an updated indexes arrow. The Idxs type has one more variant, constructed by random-idxs : J ⇒ Idxs. The only difference between the remainder of the

$$\mathsf{f} \ := \ [\![p]\!]^{\Downarrow}_{\bot *} \ \mathsf{j}_0 \qquad \mathsf{h}' \ := \ [\![p]\!]^{\Downarrow'}_{\mathsf{pre}*} \ \mathsf{j}_0 \qquad \mathsf{idxs} \ := \ [\![p]\!]^{\Downarrow}_{\mathsf{idxs}*} \ \mathsf{j}_0$$

$$\text{where } \mathsf{f} : \mathsf{Rect}\ \langle\langle \Omega, \mathsf{T}\rangle, \langle\rangle\rangle \rightsquigarrow_\bot \mathsf{Y}$$

```
refine : Rect ⟨Ω, T⟩ ⇒ Rect ⟨Ω, T⟩
refine A := image fst (ap'_{pre*} (h' (A × {⟨⟩})) B)

sample-part : Idxs ⇒ ⟨ℝ, Rect ⟨Ω, T⟩⟩ ⇒ ⟨ℝ, Rect ⟨Ω, T⟩⟩
sample-part idxs ⟨p_n, ∅⟩                        := ⟨0, ∅⟩
sample-part ⟨⟩ pr                                := pr
sample-part ⟨idxs_1, idxs_2⟩ pr                  := let pr' := sample-part idxs_1 pr
                                                      in sample-part idxs_2 pr'
sample-part (random-idxs j) ⟨p_n, Ω' × T'⟩        := let ⟨p_i, B⟩ := sample-real-part (proj j Ω')
                                                      in ⟨p_n · p_i, refine (unproj j Ω' B × T')⟩
sample-part (if-idxs j idxs_2 idxs_3) ⟨p_n, Ω' × T'⟩ := let ⟨p_b, b⟩ := sample-branch (proj j T' ∪ {⊥})
                                                            pr' := ⟨p_n · p_b, refine (Ω' × unproj j T' {b})⟩
                                                      in case b
                                                           true  ⟶ sample-part (idxs_2 0) pr'
                                                           false ⟶ sample-part (idxs_3 0) pr'
                                                           ⊥     ⟶ pr'

sample-preimage Idxs ⇒ ⟨Ω_⊥, ℝ⟩
sample-preimage idxs :=
  let ⟨p_n, A⟩ := sample-part idxs ⟨1, refine (Ω × T)⟩
    in case A
         ∅       ⟶ ⟨⊥, 0⟩
         Ω' × T' ⟶ let ⟨q_ω, ω⟩ := sample-source (Ω' × T')
                            t := trace-inf T'
                            w := if (f ⟨⟨ω, t⟩, ⟨⟩⟩ ∈ B) (1/p_n · 1/q_ω) 0
                      in ⟨ω, w⟩
```

Figure 8.16: Sampling from the preimage of $\mathsf{B}$ under the program $p$ interpreted as a random variable, using preimage refinement and a uniform candidate distribution.

code and that in Figure 8.14a is $\mathsf{random}_{\mathsf{idxs}*}\ \mathsf{j} := \mathsf{random\text{-}idxs}\ \mathsf{j}$ instead of $\mathsf{random}_{\mathsf{idxs}*}\ \mathsf{j} := \langle\rangle$.

The proofs of the preceeding theorems indicate the properties the new partition sampler must have.

- It must be sound: for any $\mathsf{t} \in \mathsf{T}_*$, with positive probability, it constructs a $\mathsf{T}'$ whose infimum is $\mathsf{t}$.
- It must partition: if it constructs a $\mathsf{T}'$ whose infimum is not in $\mathsf{T}_*$, $\mathsf{T}'$ must induce $\varnothing$.
- It must terminate: branch choices must be $\bot$ with positive probability.
- All branch trace sets must have minimum traces (i.e. no set-valued branch choices).
- The combination of choices made must correspond with exactly one output.

Figure 8.16 defines the ***preimage refinement sampling algorithm***, in which sample-part is an extension of sample-traces*. The key differences are

- It samples from a rectangular cover of a partition of $\Omega \times \mathsf{T}$ instead of from a rectangular partition of $\mathsf{T}$.
- For random-idxs j, it uses sample-real-part to sample from a partition of proj j $\Omega'$.
- It uses refine to shrink the part's covering rectangle after every choice.
- It stops immediately if it receives $\varnothing$, which refine may return.
- It chooses branches from proj j $\mathsf{T}' \cup \{\bot\}$ instead of $\mathsf{Bool}_\bot$, which allows refine to rule out branch choices that disagree with $\Omega'$.

We assume that for each input, sample-real-part : Rect $\mathbb{R} \Rightarrow \langle \mathbb{R}, \mathsf{Rect}\ \mathbb{R} \rangle$ computes a deterministic partition, assigns each part a nonzero probability, and returns the correct probability for the part it chooses. If so, sample-part is sound, it partitions, and it terminates; all branch sets have minimum traces, its transformation from random choices to parts is injective, and it returns the correct probabilities.

Besides sample-part, Figure 8.16 defines sample-preimage, which returns weighted samples of points in the preimage of B under program $p$'s interpretation as a function. It does so by partitioned importance sampling. It first uses sample-part to return a rectangle covering a part in the partition and the probability with which the part was sampled. If the part is $\varnothing$, it returns $\bot$ weighted by 0. If the part is nonempty, it samples from the random sources and weights the sample. For sample $\omega$ and trace t, if f $\langle\langle \omega, \mathsf{t} \rangle, \langle\rangle\rangle \notin \mathsf{B}$ then $\langle \omega, \mathsf{t} \rangle$ is not in the preimage of B, so it weights $\omega$ by 0, which is equivalent to rejecting it.

If the sample's image is in B, sample-preimage computes the sample's weight as $1/\mathsf{p_n} \cdot 1/\mathsf{q}_\omega$. To correctly do partitioned importance sampling, the weight should be $1/\mathsf{p_n} \cdot$ $\mathsf{diff}^+$ (subcond P $\Omega''$) (Q n) $\omega$, where $\mathsf{p_n}$ is the probability of choosing the part. The leading term is thus correct, so we need to show $1/\mathsf{q}_\omega = \mathsf{diff}^+$ (subcond P $\Omega''$) (Q n) $\omega$.

To state the theorem, we need some definitions. Let $\mathsf{h} := [\![p]\!]^{\Downarrow}_{\mathsf{pre}^*}\ \mathsf{j}_0$ be the interpretation of $p$ as a preimage arrow computation, and $\Omega'' :=$ image (fst $\ggg$ fst) ($\mathsf{ap_{pre}}$ (h $(\Omega' \times \mathsf{T}') \times \{\langle\rangle\}$) B) be the exact part under its covering rectangle $\Omega'$. Let $\mathsf{J}' \subseteq \mathsf{J}$ such that $\mathsf{j} \in \mathsf{J}'$ if and only if $\mathsf{sample\text{-}part}$ is applied to $\mathsf{random\text{-}idxs}\ \mathsf{j}$. This is the set of indexes of random values in any $\omega \in \Omega''$ that are actually used while running the program, and it is finite. Let $\mathsf{n}$ be the partition index of the part covered by $\Omega' \times \mathsf{T}'$.

**Theorem 8.45.** *Let* $\mathsf{Q}\ \mathsf{n}$ *have a density* $\mathsf{q}$ *when restricted to indexes* $\mathsf{J}'$ *and be uniform on* $\mathsf{J} \backslash \mathsf{J}'$*. Suppose* $\mathsf{sample\text{-}source}\ (\Omega' \times \mathsf{T}')$ *chooses* $\omega \in \Omega'$ *according to* $\mathsf{Q}\ \mathsf{n}$*. If* $\omega \in \Omega''$*, then* $\mathsf{diff}^+\ (\mathsf{subcond}\ \mathsf{P}\ \Omega'')\ (\mathsf{Q}\ \mathsf{n})\ \omega \ = \ 1/(\mathsf{q}\ (\mathsf{restrict}\ \omega\ \mathsf{J}'))$.

*Proof.* This is a straightforward application of Theorem 10.25, so we need only meet the conditions. Because $\mathsf{J}'$ is finite,

- The subprobability measure $\mathsf{subcond}\ \mathsf{P}\ \Omega''$ can be factored into $\mathsf{P}' : \mathsf{Set}\ (\mathsf{J}' \to [0,1]) \Rightarrow [0,1]$ and a uniform probability measure on $\mathsf{J} \backslash \mathsf{J}' \to [0,1]$.
- The probability measure $\mathsf{Q}\ \mathsf{n}$ can be factored into $\mathsf{Q}' : \mathsf{Set}\ (\mathsf{J}' \to [0,1]) \Rightarrow [0,1]$ and the same uniform probability measure.

A density for $\mathsf{P}'$ that is uniform on $\Omega''$ is

$$\begin{aligned} &\mathsf{p} : (\mathsf{J}' \to [0,1]) \to [0,+\infty) \\ &\mathsf{p}\ \omega \ := \ \mathsf{if}\ (\omega \in \Omega'')\ 1\ 0 \end{aligned} \tag{8.48}$$

By assumption, the density of $\mathsf{Q}'$ is $\mathsf{q} : (\mathsf{J}' \to [0,1]) \to [0,+\infty)$. Therefore, if $\omega \in \Omega''$,

$$\mathsf{diff}^+\ (\mathsf{subcond}\ \mathsf{P}\ \Omega'')\ (\mathsf{Q}\ \mathsf{n})\ \omega \ = \ \frac{\mathsf{p}\ (\mathsf{restrict}\ \omega\ \mathsf{J}')}{\mathsf{q}\ (\mathsf{restrict}\ \omega\ \mathsf{J}')} \ = \ \frac{1}{\mathsf{q}\ (\mathsf{restrict}\ \omega\ \mathsf{J}')} \tag{8.49}$$

□

Thus, if $\mathsf{sample\text{-}source}\ (\Omega' \times \mathsf{T}') = \langle \mathsf{q}\ (\mathsf{restrict}\ \omega\ \mathsf{J}'), \omega \rangle$, then preimage refinement sampling is correct.

### 8.4.4 Random Source Sampling

An easy way to ensure preimage refinement sampling is correct is to sample uniformly, so that the density at every $\omega \in \Omega'$ is the reciprocal of the volume of $\Omega'$. Let $\mathsf{m} : \mathsf{Set}\ \mathbb{R} \rightharpoonup [0, +\infty]$ be Lebesgue measure on $\mathbb{R}$. Define

$$\begin{aligned}
&\mathsf{sample\text{-}source} : \mathsf{Rect}\ \langle \Omega, \mathsf{T} \rangle \Rightarrow \langle \mathbb{R}, \Omega \rangle \\
&\mathsf{sample\text{-}source}\ (\Omega' \times \mathsf{T}') \ :=\ \mathsf{let}\ \ \mathsf{q}_\omega := 1 \ / \ \textstyle\prod_{\mathsf{j} \in \mathsf{J}}\ \mathsf{m}\ (\mathsf{proj}\ \mathsf{j}\ \Omega') \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \omega := \lambda \mathsf{j} \in \mathsf{J}.\, \mathsf{sample\text{-}uniform}\ (\mathsf{proj}\ \mathsf{j}\ \Omega') \\
&\qquad\qquad\qquad\qquad\qquad\quad \mathsf{in}\ \ \langle \mathsf{q}_\omega, \omega \rangle
\end{aligned} \tag{8.50}$$

Because $\mathsf{Rect}\ \langle \Omega, \mathsf{T} \rangle$ is defined so that only finitely many axes of $\Omega'$ are strict subsets of $[0, 1]$, $\mathsf{q}_\omega$ is well-defined whenever the volume of $\Omega'$ is nonzero.[5] In particular, $\mathsf{m}\ (\mathsf{proj}\ \mathsf{j}\ \Omega') < 1$ if $\mathsf{j} \in \mathsf{J}'$, otherwise 1.

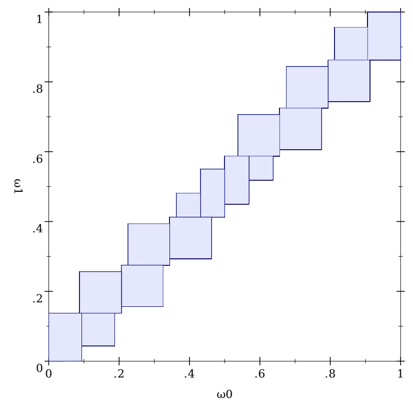
An implementation of sample-source cannot compute a product over all J, nor construct a mapping with domain J. The representations of $\mathsf{Rect}\ \Omega$ and $\omega \in \Omega$ given in Figures 8.5 and 8.6 make getting around this easy. The function `omega-set-sample` in Figure 8.6 implements $\lambda \mathsf{j} \in \mathsf{J}.\, \mathsf{sample\text{-}uniform}\ (\mathsf{proj}\ \mathsf{j}\ \Omega')$ by building a lazy tree. Further, because rectangles may have only finitely many nonfull axes, it is easy to write a total recursive function to compute $\prod_{\mathsf{j} \in \mathsf{J}}\ \mathsf{m}\ (\mathsf{proj}\ \mathsf{j}\ \Omega')$ to measure the volumes of $\Omega$ rectangles. The measure of an `Omega-Node` instance is the product of its axis's measure and the measures of its subtrees. The measure of `univ-omega-set` is `1`.

Figure 8.17 shows the result of sampling within the preimage of $[-0.05, 0.05]$ under the interpretation of this program as a random variable:

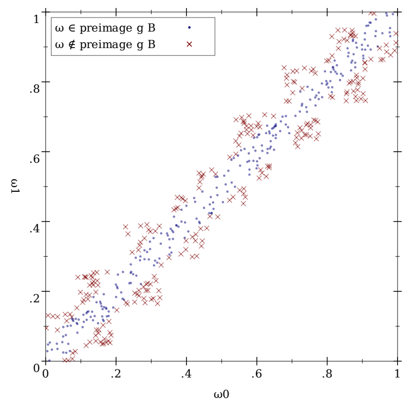
```
(define/drbayes e
  (let ([x  (random)]
        [y  (random)])
    (- y x)))
```

Figure 8.17a shows the results returned by the implementation of sample-part: sampled parts

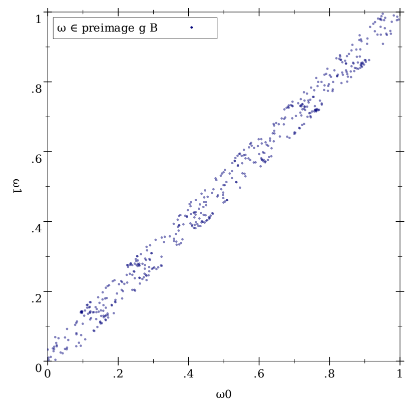
[5] In practice, we do not have to consider this case. The implementation of sample-source may return reciprocal densities, so it returns the volume of $\Omega'$, which is always well-defined.



(a) sample-part results



(b) sample-source results



(c) sample-source* results

Figure 8.17: 500 samples taken using Dr. Bayes's implementation of sample-partition. Subfigures (b) and (c) demonstrate the difference between the uniform sample-source and nonuniform sample-source*. In (b), 255 samples are accepted; in (c), all 500 samples are accepted.

from a rectangle covering the true preimage, which surrounds the line $\omega_1 = \omega_0$. (There are many duplicates.) Figure 8.17b shows the result of sampling once within each part uniformly; in this case, 255 out of 500 samples are inside the preimage set.

Suppose we had sampled within the preimage of $[-0.001, 0.001]$, or indeed $[-\epsilon, \epsilon]$ for any $\epsilon > 0$. The proportion of accepted samples drops with $\epsilon$. We can mitigate this problem using finer partitions of proj j $\Omega'$. However, there is a solution that does not require finer partitions and accepts more samples than any repartitioning.

The key insight is that the singleton interval $[\mathsf{b}, \mathsf{b}] = \{\mathsf{b}\}$ is also a rectangle. To sample within a part, for each axis $\mathsf{j} \in \mathsf{J}'$, we choose a $\mathsf{b} \in$ proj j $\Omega'$, update $\Omega'$ using unproj j Omega′ $\{\mathsf{b}\}$, and use refine to get better bounds for sampling the other axes.

The following function implements the idea.

$$
\begin{aligned}
&\mathsf{sample\text{-}source}^* : [\mathsf{J}] \Rightarrow \langle \mathbb{R}, \mathsf{Rect}\ \langle \Omega, \mathsf{T} \rangle \rangle \Rightarrow \langle \mathbb{R}, \Omega \rangle \\
&\mathsf{sample\text{-}source}^*\ \langle\rangle\ \langle \mathsf{q}_\omega, \Omega' \times \mathsf{T}' \rangle \quad := \ \mathsf{let}\ \ \omega := \lambda \mathsf{j} \in \mathsf{J}.\, \mathsf{sample\text{-}uniform}\ (\mathsf{proj}\ \mathsf{j}\ \Omega') \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{in}\ \ \langle \mathsf{q}_\omega, \omega \rangle \\
&\mathsf{sample\text{-}source}^*\ \langle \mathsf{j}, \mathsf{js} \rangle\ \langle \mathsf{q}_\omega, \Omega' \times \mathsf{T}' \rangle := \ \mathsf{let}\ \ \mathsf{B} := \mathsf{proj}\ \mathsf{j}\ \Omega' \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{b} := \mathsf{sample\text{-}uniform}\ \mathsf{B} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{A}' := \mathsf{refine}\ (\mathsf{unproj}\ \mathsf{j}\ \Omega'\ \{\mathsf{b}\} \times \mathsf{T}') \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{in}\ \ \mathsf{sample\text{-}source}^*\ \mathsf{js}\ \langle \mathsf{q}_\omega \cdot 1/(\mathsf{m}\ \mathsf{B}), \mathsf{A}' \rangle
\end{aligned}
\tag{8.51}
$$

Here, $\mathsf{[J]}$ is the type of lists of $\mathsf{J}$, or $\langle \mathsf{J}, \langle \mathsf{J}, ...\langle \mathsf{J}, \langle\rangle\rangle\rangle\rangle$. The caller is expected to linearize $\mathsf{J}'$, the indexes of random values that are actually used while running the program, as $\mathsf{js} : \mathsf{[J]}$. (Dr. Bayes's implementation of sample-part returns js in addition to the covering rectangle and its probability.) The density of the sampled $\omega$ is computed as $\prod_{\mathsf{j} \in \mathsf{J}'} 1/(\mathsf{m}\ (\mathsf{proj\ j}\ \Omega'_{\mathsf{j}}))$, where $\Omega'_{\mathsf{j}}$ are the ever-shrinking inputs to sample-source*. Roughly, it is the joint density of *dependent* uniform random variables evaluated at $\mathsf{restrict}\ \omega\ \mathsf{J}'$.

Figure 8.17c shows the result of using Dr. Bayes's implementation of sample-source* to sample within parts. No samples are rejected: in all cases, choosing an $\omega_0$ and updating $\Omega'$ with $\{\omega_0\}$ allows preimage refinement to determine the range of values for $\omega_1$ for which $\omega$ is in the preimage set.

XXX: extension: no $\perp$ branch choices when sampling in $\mathsf{T}_+$; characterize the programs for which this extension doesn't terminate; explain options to detect or fix (does this go here?)

## 8.5 Examples

normal-normal

normal-normal with circular condition

thermometer

normal-normals, apparent quadratic time vs. n*log(n) time depending on data layout

# Chapter 9

# Measurability Theorems

Proving measurability is critical in proving correctness, in that it establishes that the outputs of all programs have sensible distributions. While critical, it is somewhat distracting to the main narrative. Instead of ignoring measurability, however, as is so often done, we have moved it near the end, where readers that are somehow *still starving for even more mathematics* can devour it and—possibly—finally be satiated.

## 9.1 Basic Definitions

For readers familiar with topology, we review the necessary fundamentals by analogy to topology. However, we have tried to include enough of the fundamentals that readers not familiar with basic topology can verify the proofs. [XXX: cite book for imported lemmas]

The analogue of a topology of open sets is a $\sigma$-algebra of measurable sets.

**Definition 9.1** ($\sigma$-algebra, measurable set)**.** *A collection of sets $\mathcal{A} \subseteq \mathcal{P}\ \mathsf{X}$ is called a **$\sigma$-algebra** on $\mathsf{X}$ if it contains $\mathsf{X}$ 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 9.2** (measurable mapping)**.** *Let $\mathcal{A}$ and $\mathcal{B}$ be $\sigma$-algebras on $\mathsf{X}$ and $\mathsf{Y}$. 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 $\sigma$-algebras $\mathcal{A}$ and $\mathcal{B}$ are clear from context, we will simply say $\mathsf{g}$ is **measurable**.

Measurability is usually a weaker condition than continuity. For example, with respect to the $\sigma$-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 $\sigma$-algebras are defined analogously to product topologies.

**Definition 9.3** (finite product $\sigma$-algebra)**.** *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be $\sigma$-algebras on $\mathsf{X}_1$ and $\mathsf{X}_2$, and define $\mathsf{X} := \mathsf{X}_1 \times \mathsf{X}_2$. The* ***product $\sigma$-algebra*** *$\mathcal{A}_1 \otimes \mathcal{A}_2$ is the smallest (i.e. coarsest) $\sigma$-algebra for which* $\mathsf{mapping\ fst\ X}$ *and* $\mathsf{mapping\ snd\ X}$ *are measurable.*

**Definition 9.4** (arbitrary product $\sigma$-algebra)**.** *Let $\mathcal{A}$ be a $\sigma$-algebra on $\mathsf{X}$. The* ***product $\sigma$-algebra*** *$\mathcal{A}^{\otimes \mathsf{J}}$ is the smallest $\sigma$-algebra for which, for all $\mathsf{j} \in \mathsf{J}$,* $\mathsf{mapping}\ (\pi\ \mathsf{j})\ (\mathsf{J} \to \mathsf{X})$ *is measurable.*

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

**Theorem 9.6** (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 $\mathsf{A}^*$ is in $\mathcal{A}$.*

*Proof.* Because $\mathsf{g}\ \mathsf{A}^*$ is measurable, $\mathsf{preimage}\ (\mathsf{g}\ \mathsf{A}^*)\ \mathsf{Y} = \mathsf{A}^*$ 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 $\mathsf{A}^*$. Fortunately, that is true without any extra conditions.

**Lemma 9.7.** *Let* $\mathsf{g} : \mathsf{X} \rightharpoonup \mathsf{Y}$ *be an* $\mathcal{A}$*-*$\mathcal{B}$*-measurable mapping. For any* $\mathsf{A} \in \mathcal{A}$, $\mathsf{restrict}\ \mathsf{g}\ \mathsf{A}$ *is* $\mathcal{A}$*-*$\mathcal{B}$*-measurable.*

**Theorem 9.8.** *Let* $\mathsf{g} : \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}$ *be an* $\mathcal{A}$*-*$\mathcal{B}$*-measurable mapping arrow computation with maximal domain* $\mathsf{A}^*$*. For all* $\mathsf{A} \subseteq \mathsf{A}^*$ *with* $\mathsf{A} \in \mathcal{A}$, $\mathsf{g}\ \mathsf{A}$ *is* $\mathcal{A}$*-*$\mathcal{B}$*-measurable.*

*Proof.* By Theorem 7.44 (mapping arrow restriction) and Lemma 9.7. □

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.

### 9.2.1 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 9.9** (images of preimages)**.** *If* $\mathsf{g} : \mathsf{X} \rightharpoonup \mathsf{Y}$ *and* $\mathsf{B} \subseteq \mathsf{Y}$, $\mathsf{image}\ \mathsf{g}\ (\mathsf{preimage}\ \mathsf{g}\ \mathsf{B}) \subseteq \mathsf{B}$.

**Lemma 9.10** (expanded post-composition)**.** *Let* $\mathsf{g}_1 : \mathsf{X} \rightharpoonup \mathsf{Y}$ *and* $\mathsf{g}_2 : \mathsf{Y} \rightharpoonup \mathsf{Z}$ *such that* $\mathsf{range}\ \mathsf{g}_1 \subseteq \mathsf{domain}\ \mathsf{g}_2$*, 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 9.11** (mapping arrow monotonicity)**.** *Let* $\mathsf{g} : \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}$*. For any* $\mathsf{A}' \subseteq \mathsf{A} \subseteq \mathsf{A}^*$, $\mathsf{g}\ \mathsf{A}' \subseteq \mathsf{g}\ \mathsf{A}$.

*Proof.* By Theorem 7.44 (mapping arrow restriction). □

**Theorem 9.12** (maximal domain subsets)**.** *Let* $\mathsf{g} : \mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y}$*. For all* $\mathsf{A} \subseteq \mathsf{A}^*$, $\mathsf{domain}\ (\mathsf{g}\ \mathsf{A}) = \mathsf{A}$.

*Proof.* Follows from Theorem 7.45. $\square$

Now we can prove measurability.

**Lemma 9.13** (($\circ_{\mathsf{map}}$) 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 9.14** (($\ggg_{\mathsf{map}}$) 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 $\mathsf{g}_1$'s and $\mathsf{g}_2$'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,

$$(\mathsf{g}_1 \ggg_{\mathsf{map}} \mathsf{g}_2)\ \mathsf{A}^{**} \ = \ \begin{array}{l}\mathsf{let}\ \ \mathsf{g}_1' := \mathsf{g}_1\ \mathsf{A}^{**} \\ \qquad \mathsf{g}_2' := \mathsf{g}_2\ (\mathsf{range}\ \mathsf{g}_1') \\ \mathsf{in}\ \ \mathsf{g}_2' \circ_{\mathsf{map}} \mathsf{g}_1' \end{array} \tag{9.1}$$

By Theorem 9.8, $\mathsf{g}_1'$ is an $\mathcal{A}$-$\mathcal{B}$-measurable mapping. Unfortunately, $\mathsf{g}_2'$ 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 9.13, $\mathsf{g}_2'' \circ_{\mathsf{map}} \mathsf{g}_1'$ 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 9.10 is true if $\mathsf{range}\ \mathsf{g}_1' \subseteq \mathsf{domain}\ \mathsf{g}_2'$ and $\mathsf{g}_2' \subseteq \mathsf{g}_2''$.

By Theorem 9.12, $\mathsf{A}^{**} \subseteq \mathsf{A}^*$ implies $\mathsf{domain}\ \mathsf{g}_1' = \mathsf{A}^{**}$. By Theorem 9.11 and Lemma 9.9,

$$\begin{aligned} \mathsf{range}\ \mathsf{g}_1' \ &= \ \mathsf{image}\ (\mathsf{g}_1\ \mathsf{A}^{**})\ (\mathsf{preimage}\ (\mathsf{g}_1\ \mathsf{A}^*)\ \mathsf{B}^*) \\ &= \ \mathsf{image}\ (\mathsf{g}_1\ \mathsf{A}^{*})\ (\mathsf{preimage}\ (\mathsf{g}_1\ \mathsf{A}^*)\ \mathsf{B}^*) \\ &\subseteq \ \mathsf{B}^* \end{aligned} \tag{9.2}$$

$\mathsf{range}\ \mathsf{g}_1' \subseteq \mathsf{B}^*$ implies (by Theorem 9.12) that $\mathsf{domain}\ \mathsf{g}_2' = \mathsf{range}\ \mathsf{g}_1'$, and (by Theorem 9.11) that $\mathsf{g}_2' \subseteq \mathsf{g}_2''$. $\square$

### 9.2.2 Pairing

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

**Lemma 9.15** ($\langle\cdot,\cdot\rangle_{\mathsf{map}}$ 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 9.16** ($(\&\&\&_{\mathsf{map}})$ 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 $\mathsf{A}_1^*$ and $\mathsf{A}_2^*$ be respectively $\mathsf{g}_1$'s and $\mathsf{g}_2$'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 9.15 is $\mathcal{A}$-$(\mathcal{B}_1 \otimes \mathcal{B}_2)$-measurable. □

### 9.2.3 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 9.17** (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 9.18** ($\mathsf{ifte}_{\mathsf{map}}$ 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 $\mathsf{g}_1$'s, $\mathsf{g}_2$'s and $\mathsf{g}_3$'s maximal domains. The maximal domain of $\mathsf{ifte}_{\mathsf{map}}\ \mathsf{g}_1\ \mathsf{g}_2\ \mathsf{g}_3$ is $\mathsf{A}^{**}$, defined by

$$\begin{aligned} \mathsf{A}_2^{**} &:= \mathsf{A}_2^* \cap \mathsf{preimage}\ (\mathsf{g}_1\ \mathcal{A}_1^*)\ \{\mathsf{true}\} \\ \mathsf{A}_3^{**} &:= \mathsf{A}_3^* \cap \mathsf{preimage}\ (\mathsf{g}_1\ \mathcal{A}_1^*)\ \{\mathsf{false}\} \\ \mathsf{A}^{**} &:= \mathsf{A}_2^{**} \uplus \mathsf{A}_3^{**} \end{aligned} \tag{9.3}$$

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,

$$\begin{aligned}\mathsf{ifte}_{\mathsf{map}}\ \mathsf{g}_1\ \mathsf{g}_2\ \mathsf{g}_3\ \mathsf{A}^{**} \ = \ \mathsf{let}\ & \mathsf{g}_1' := \mathsf{g}_1\ \mathsf{A}^{**} \\ & \mathsf{g}_2' := \mathsf{g}_2\ (\mathsf{preimage}\ \mathsf{g}_1'\ \{\mathsf{true}\}) \\ & \mathsf{g}_3' := \mathsf{g}_3\ (\mathsf{preimage}\ \mathsf{g}_1'\ \{\mathsf{false}\}) \\ \mathsf{in}\ & \mathsf{g}_2' \uplus_{\mathsf{map}} \mathsf{g}_3'\end{aligned} \tag{9.4}$$

By hypothesis, $\mathsf{g}_1'$, $\mathsf{g}_2'$ and $\mathsf{g}_3'$ are measurable mappings. By Theorem 7.44 (mapping arrow restriction), $\mathsf{g}_2'$ and $\mathsf{g}_3'$ have disjoint domains. Apply Lemma 9.17. □

### 9.2.4 Laziness

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

**Theorem 9.19** (measurability of $\varnothing$). *For any $\sigma$-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 $\varnothing \in \mathcal{A}$. □

**Theorem 9.20** (measurability under $\mathsf{lazy}_{\mathsf{map}}$). *Let* $\mathsf{g} : 1 \Rightarrow (\mathsf{X} \underset{\mathsf{map}}{\rightsquigarrow} \mathsf{Y})$. *If* $\mathsf{g}\ 0$ *is $\mathcal{A}$-$\mathcal{B}$-measurable, then* $\mathsf{lazy}_{\mathsf{map}}\ \mathsf{g}$ *is $\mathcal{A}$-$\mathcal{B}$-measurable.*

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

$$\mathsf{lazy}_{\mathsf{map}}\ \mathsf{g}\ \mathsf{A}^{**} \ = \ \mathsf{if}\ (\mathsf{A}^{**} = \varnothing)\ \varnothing\ (\mathsf{g}\ 0\ \mathsf{A}^{**}) \tag{9.5}$$

If $\mathsf{A}^{**} = \varnothing$, then $\mathsf{lazy}_{\mathsf{map}}\ \mathsf{g}\ \mathsf{A}^{**} = \varnothing$; apply Theorem 9.19. If $\mathsf{A}^{**} \neq \varnothing$, then $\mathsf{lazy}_{\mathsf{map}}\ \mathsf{g} = \mathsf{g}\ 0$, which is $\mathcal{A}$-$\mathcal{B}$-measurable. □

## 9.3 Measurable Probabilistic Computations

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

**Definition 9.21** (measurable mapping* arrow computation). *Let $\mathcal{A}$ and $\mathcal{B}$ be $\sigma$-algebras on* $(\Omega \times \mathsf{T}) \times \mathsf{X}$ *and* $\mathsf{Y}$. *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 9.22.** *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 $\mathsf{J}$: if $\mathsf{g}\ \mathsf{j}$ is measurable, so are $\mathsf{g}\ (\mathsf{left}\ \mathsf{j})$ and $\mathsf{g}\ (\mathsf{right}\ \mathsf{j})$. □

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

**Definition 9.23** (standard $\sigma$-algebra). *For a set* $\mathsf{X}$ *used as a type,* $\Sigma\ \mathsf{X}$ *denotes its* ***standard*** ***$\sigma$-algebra****, which must be defined under the following constraints:*

$$\Sigma\ \langle \mathsf{X}_1, \mathsf{X}_2 \rangle = \Sigma\ \mathsf{X}_1 \otimes \Sigma\ \mathsf{X}_2 \tag{9.6}$$

$$\Sigma\ (\mathsf{J} \to \mathsf{X}) = (\Sigma\ \mathsf{X})^{\otimes \mathsf{J}} \tag{9.7}$$

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

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

$$\Sigma\ \mathsf{Bool}\ ::=\ \mathcal{P}\ \mathsf{Bool} \tag{9.8}$$

$$\Sigma\ \mathsf{T}\ ::=\ \mathcal{P}\ \mathsf{T} \tag{9.9}$$

**Lemma 9.24** (measurable mapping arrow lifts). $\mathsf{arr_{map}}\ \mathsf{id}$, $\mathsf{arr_{map}}\ \mathsf{fst}$ *and* $\mathsf{arr_{map}}\ \mathsf{snd}$ *are measurable.* $\mathsf{arr_{map}}\ (\mathsf{const}\ \mathsf{b})$ *is measurable if* $\{\mathsf{b}\}$ *is a measurable set. For all* $\mathsf{j} \in \mathsf{J}$, $\mathsf{arr_{map}}\ (\pi\ \mathsf{j})$ *is measurable.*

**Corollary 9.25** (measurable mapping* arrow lifts). $\mathsf{arr_{map^*}}\ \mathsf{id}$, $\mathsf{arr_{map^*}}\ \mathsf{fst}$ *and* $\mathsf{arr_{map^*}}\ \mathsf{snd}$ *are measurable.* $\mathsf{arr_{map^*}}\ (\mathsf{const}\ \mathsf{b})$ *is measurable if* $\{\mathsf{b}\}$ *is a measurable set.* $\mathsf{random_{map^*}}$ *and* $\mathsf{branch_{map^*}}$ *are measurable.*

**Theorem 9.26** ($\mathsf{AStore}$ combinators preserve measurability)**.** *Every* $\mathsf{AStore}$ *arrow combinator produces measurable mapping* computations from measurable mapping* computations.*

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

**Theorem 9.27** ($\mathsf{ifte}^{\Downarrow}_{\mathsf{map^*}}$ measurability)**.** $\mathsf{ifte}^{\Downarrow}_{\mathsf{map^*}}$ *is measurable.*

*Proof.* $\mathsf{branch_{map^*}}$ is measurable, and $\mathsf{arr_{map}}\ \mathsf{agrees}$ is measurable by (9.8). □

We can now prove all nonrecursive programs measurable by induction.

**Definition 9.28** (finite expression)**.** *A **finite expression** is any expression for which no subexpression is a first-order application.*

**Theorem 9.29** (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 9.30** (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* $\mathsf{A} := (\Omega \times \{\mathsf{t}\}) \times \mathsf{X}$*. 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 $\mathsf{J}'$ contain every $\mathsf{j}$ 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 $\mathsf{j}$ is not in $\mathsf{J}'$ with the equivalent expression $\bot$. Because $e$ is well-defined, recurrences must be guarded by $\mathsf{if}$, so this process terminates after finitely many first-order applications. □

**Theorem 9.31** (all probabilistic expressions are measurable)**.** *For all expressions $e$, $[\![e]\!]^{\Downarrow}_{\mathsf{map^*}}$ is measurable.*

*Proof.* Let $\mathsf{g} := [\![e]\!]^{\Downarrow}_{\mathsf{map}^*}$ and $\mathsf{g}' := \mathsf{g}\ \mathsf{j}_0\ ((\Omega \times \mathsf{T}) \times \mathsf{X})$. By Corollary 7.51 (correct computation everywhere), $\mathsf{g}' = \mathsf{g}\ \mathsf{j}_0\ \mathsf{A}^*$ where $\mathsf{A}^*$ is $\mathsf{g}$'s maximal domain; thus we need only show $\mathsf{g}'$ is a measurable mapping.

By Theorem 7.44 (mapping arrow restriction),

$$\mathsf{g}' \ = \ \bigcup_{\mathsf{t} \in \mathsf{T}} \mathsf{g}\ \mathsf{j}_0\ ((\Omega \times \{\mathsf{t}\}) \times \mathsf{X}) \tag{9.10}$$

By Theorem 9.30 (approximation with finite expressions), for every $\mathsf{t} \in \mathsf{T}$, there is a finite expression whose interpretation agrees with $\mathsf{g}$ on $(\Omega \times \{\mathsf{t}\}) \times \mathsf{X}$. Therefore, by Theorem 9.29 (all finite expressions are measurable), $\mathsf{g}\ \mathsf{j}_0\ ((\Omega \times \{\mathsf{t}\}) \times \mathsf{X})$ is a measurable mapping. By Theorem 7.44 (mapping arrow restriction), they have disjoint domains. By Lemma 9.17 (union of measurable mappings), their union is measurable. □

Theorem 9.31 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 $\sigma$-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.

## 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

$$\mathsf{P}\ (\mathsf{image}\ (\mathsf{fst} \ggg \mathsf{fst})\ (\mathsf{preimage}\ (\mathsf{g}\ \mathsf{j}_0\ \mathsf{A}^*)\ \mathsf{B})) \tag{9.11}$$

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

*Proof.* Because $\Sigma\ \mathsf{X}_2 = \mathcal{P}\ \mathsf{X}_2$, $\mathsf{A}$ 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 $\mathsf{image\ fst}$ distributes over unions, $\mathsf{image\ fst\ A}$ is a countable union of sets in $\Sigma\ \mathsf{X}_1$. □

**Theorem 9.33.** *Let* $\mathsf{g} : \mathsf{X} \underset{\mathsf{map}^*}{\rightsquigarrow} \mathsf{Y}$ *be measurable. If* $\mathsf{X}$ *is at most countable and* $\Sigma\ \mathsf{X} = \mathcal{P}\ \mathsf{X}$*, for all* $\mathsf{B} \in \Sigma\ \mathsf{Y}$*,* $\mathsf{image\ (fst \ggg fst)\ (preimage\ (g\ j_0\ A^*)\ B)} \in \Sigma\ \Omega$*.*

*Proof.* $\mathsf{T}$ is countable and $\Sigma\ \mathsf{T} = \mathcal{P}\ \mathsf{T}$ by (9.9); apply Theorem 9.32 twice. □

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

# Chapter 10

# Sampling Theorems

This chapter contains proofs of measure-theoretic theorems stated in Chapter 8.

## 10.1 Basic Definitions

While the following review is necessarily incomplete, we have tried to include enough discussion for readers unfamiliar with measure theory, and enough formalism that the proofs can be verified without consulting an outside text. For example, we do not define measure-theoretic integration, but we contrast it with integration typically learned in differential calculus, and import theorems about its properties and interactions with other operations we use.

### 10.1.1 Measures

Measure theory is named for its primary abstraction of length, area, volume and probability—and anything else for which assigning reals to sets in an additive way makes sense.

**Definition 10.1** (measure)**.** *A partial function* $\mathsf{m} : \mathsf{Set}\ \mathsf{X} \rightharpoonup [0, +\infty]$ *with domain* $\mathcal{A} :=$ $\mathsf{domain}\ \mathsf{m}$ *is a* ***measure*** *if*

- $\mathcal{A}$ *is a* $\sigma$*-algebra*
- $\mathsf{m}\ \varnothing = 0$
- *It is* $\sigma$***-additive****: for any disjoint collection* $\mathsf{A} : \mathbb{N} \Rightarrow \mathsf{Set}\ \mathsf{X}$ *of sets in* $\mathcal{A}$*,*

$$\mathsf{m}\ \left(\bigcup_{\mathsf{n} \in \mathbb{N}} \mathsf{A}\ \mathsf{n}\right) = \sum_{\mathsf{n} \in \mathbb{N}} \mathsf{m}\ (\mathsf{A}\ \mathsf{n}) \tag{10.1}$$

From here on, we rely again on the notion of a set $\mathsf{X}$'s standard $\sigma$-algebra $\mathit{\Sigma}\ \mathsf{X}$, and assume the domain of a measure $\mathsf{m} : \mathsf{Set\ X} \rightharpoonup [0,1]$ is $\mathit{\Sigma}\ \mathsf{X}$.

We will need to distinguish three kinds of measures.

**Definition 10.2** (probability, finite, and $\sigma$-finite measures)**.** *A measure* $\mathsf{m} : \mathsf{Set\ X} \rightharpoonup [0,+\infty]$ *may be*

- *A* ***probability measure*** *if* $\mathsf{m\ X} = 1$*.*
- *A* ***finite measure*** *if* $\mathsf{m\ X} < +\infty$*.*
- *A* $\sigma$***-finite measure*** *if there is a collection* $\mathsf{A} : \mathbb{N} \Rightarrow \mathit{\Sigma}\ \mathsf{X}$ *such that* $\mathsf{m\ (A\ n)} < +\infty$ *for all* $\mathsf{n} \in \mathbb{N}$*, and* $\bigcup_{\mathsf{n}\in\mathbb{N}} \mathsf{A\ n} = \mathsf{X}$*.*

*Trivially, probability measures are also finite measures, which in turn are also* $\sigma$*-finite.*

A ubiquitous example of a $\sigma$-finite measure is **Lebesgue measure**,[1] which maps sets of $\mathbb{R}^{\mathsf{n}}$ (for $\mathsf{n} \geq 1$) to their lengths, areas and volumes. Indeed, the Lebesgue measure of $\mathbb{R}$ is $+\infty$, but $\mathbb{R}$ is the union of countably many sets with finite measure; e.g. $\mathbb{R} = \bigcup_{\mathsf{n}\in\mathbb{N}}[-\mathsf{n},\mathsf{n}]$.

**Counting measure** simply returns the cardinality of a set. If $\mathsf{X}$ is countable and $\mathsf{m} : \mathsf{Set\ X} \rightharpoonup [0,+\infty]$ is counting measure, then $\mathsf{m}$ is $\sigma$-finite. If $\mathsf{X}$ is finite, $\mathsf{m}$ is finite.

Image measure defines measures over the outputs of functions in terms of measures over their inputs.

**Definition 10.3** (image measure)**.** *Let* $\mathsf{m} : \mathsf{Set\ X} \rightharpoonup [0,+\infty]$ *be a measure and* $\mathsf{g} : \mathsf{X} \rightharpoonup \mathsf{Y}$ *be measurable. Then* $\mathsf{g}$*'s* ***image measure*** *with respect to* $\mathsf{m}$ *is* $\mathsf{m}' : \mathsf{Set\ Y} \rightharpoonup [0,+\infty]$*, defined by* $\mathsf{m}'\ \mathsf{B} = \mathsf{m\ (preimage\ g\ B)}$*.*

Measures provide a way to differentiate between propositions that are always true, and propositions that may be false, but are true for certain practical purposes. To determine the latter, we need the concept of a **null set**: a set of measure zero. For example, with Lebesgue measure, $\{4\}$, or any other singleton, is a null set. In general, so is any countable union of null sets.

[1]Pronounced "lehBEG," and named after French mathematician Henri Lebesgue.

**Definition 10.4** (almost everywhere)**.** *A measurable predicate* $\mathsf{p?}$ *holds* ***almost everywhere*** *with respect to measure* $\mathsf{m} : \mathsf{Set}\ \mathsf{X} \rightharpoonup [0, +\infty]$ *when it holds on the* complement *of a null set:*

$$\begin{aligned} &\mathsf{ae?} : (\mathsf{Set}\ \mathsf{X} \rightharpoonup [0, +\infty]) \Rightarrow (\mathsf{X} \rightharpoonup \mathsf{Bool}) \Rightarrow \mathsf{Bool} \\ &\mathsf{ae?}\ \mathsf{m}\ \mathsf{p?}\ :=\ \mathsf{m}\ (\mathsf{preimage}\ \mathsf{p?}\ \{\mathsf{false}\}) = 0 \end{aligned} \tag{10.2}$$

If $\mathsf{m}$ is a probability measure, $\mathsf{ae?}\ \mathsf{m}\ \mathsf{p?}$ is equivalent to $\mathsf{m}\ (\mathsf{preimage}\ \mathsf{p?}\ \{\mathsf{true}\}) = 1$, or to $\mathsf{p?}$ holding on a set of measure 1. If $\mathsf{m}$ is a finite measure, it is equivalent to $\mathsf{p?}$ holding on a set of measure $\mathsf{m}\ \mathsf{X}$. If $\mathsf{m}$ is any other kind of measure, $\mathsf{ae?}\ \mathsf{m}\ \mathsf{p?}$ must be determined using null sets. If we were to say $\mathsf{p?}$ holds almost everywhere when $\mathsf{m}\ (\mathsf{preimage}\ \mathsf{p?}\ \{\mathsf{true}\}) = \mathsf{m}\ \mathsf{X} = +\infty$, we would have to say $\lambda \mathsf{x} \in \mathbb{R}.\, \mathsf{x} > 0$ holds almost everywhere with respect to Lebesgue measure.

In this chapter, we are most interested in almost-everywhere equality of mappings.

**Definition 10.5** (almost-everywhere equality)**.** *Two total mappings are* ***equal almost everywhere*** *with respect to a measure* $\mathsf{m} : \mathsf{Set}\ \mathsf{X} \rightharpoonup [0, +\infty]$ *when they are not equal only on a null set, or* $\mathsf{ae\text{-}equal?}\ \mathsf{m}\ \mathsf{g}_1\ \mathsf{g}_2$ *where*

$$\begin{aligned} &\mathsf{ae\text{-}equal?} : (\mathsf{Set}\ \mathsf{X} \rightharpoonup [0, +\infty]) \Rightarrow (\mathsf{X} \to \mathsf{Y}) \Rightarrow (\mathsf{X} \to \mathsf{Y}) \Rightarrow \mathsf{Bool} \\ &\mathsf{ae\text{-}equal?}\ \mathsf{m}\ \mathsf{g}_1\ \mathsf{g}_2\ :=\ \mathsf{ae?}\ \mathsf{m}\ \lambda \mathsf{a} \in \mathsf{domain}\ \mathsf{g}_1.\, \mathsf{g}_1\ \mathsf{a} = \mathsf{g}_2\ \mathsf{a} \end{aligned} \tag{10.3}$$

From here on, we use the more common "$\mathsf{g}_1 = \mathsf{g}_2$ ($\mathsf{m}$-a.e.)" instead of $\mathsf{ae\text{-}equal?}\ \mathsf{m}\ \mathsf{g}_1\ \mathsf{g}_2$.

### 10.1.2 Integration

While measure-theoretic integration—called **Lebesgue integration**—is $\lambda_{\mathrm{ZFC}}$-definable, defining it will not illuminate the proofs further on. The main things to know are:

- Lebesgue integration is done with respect to any base measure for the domain of integration. In contrast, Riemann integration[2] (as taught in differential calculus) is done only with respect to length, area or volume in $\mathbb{R}$, $\mathbb{R}^2$ and other finite products $\mathbb{R}^{\mathsf{n}}$.
- Lebesgue integration with respect to Lebesgue measure is strictly more general than

[2]Pronounced "REEmahn," and named after the German mathematician Bernhard Riemann.

Riemann integration on $\mathbb{R}^{\mathsf{n}}$, in that it can integrate more functions. Further, a Lebesgue integral is equivalent to a corresponding Riemann integral when the latter exists.

- Lebesgue integration with respect to *counting* measure is summation.

Rather than define Lebesgue integration from first principles, we only *functionalize* it: we assume it has been defined, and turn it from special notation into a lambda. Using the notation for Lebesgue integration, we define

$$\begin{aligned}&\mathsf{int} : (\mathsf{X} \to \mathbb{R}) \Rightarrow (\mathsf{Set}\ \mathsf{X} \rightharpoonup [0, +\infty]) \Rightarrow (\mathsf{Set}\ \mathsf{X} \rightharpoonup [-\infty, +\infty]) \\ &\mathsf{int}\ \mathsf{g}\ \mathsf{m}\ := \ \lambda \mathsf{A} \in \mathsf{domain}\ \mathsf{m}.\ \int_{\mathsf{A}} \mathsf{g}\ d\mathsf{m}\end{aligned} \tag{10.4}$$

Now $\mathsf{int}\ \mathsf{g}\ \mathsf{m}$ is an *indefinite* integral of $\mathsf{g}$: another partial function, defined on the domain of $\mathsf{m}$, that returns the definite integral on a given set $\mathsf{A}$. For example, if $\mathsf{g}\ \mathsf{x} := \mathsf{x}^2$ and $\mathsf{m} : \mathsf{Set}\ \mathbb{R} \rightharpoonup [0, +\infty]$ is Lebesgue measure, $\mathsf{int}\ \mathsf{g}\ \mathsf{m}$ measures areas under the curve $\mathsf{y} = \mathsf{x}^2$.

We can compute areas under the curve $\mathsf{y} = \mathsf{x}^2$ using Riemann integration:

$$\mathsf{int}\ \mathsf{g}\ \mathsf{m}\ [0, 1)\ =\ \int_{[0,1)} \mathsf{g}\ d\mathsf{m}\ =\ \int_0^1 \mathsf{x}^2\ d\mathsf{x}\ =\ \frac{1^3}{3} - \frac{0^3}{3}\ =\ \frac{1}{3} \tag{10.5}$$

Of course, $\mathsf{int}\ \mathsf{g}\ \mathsf{m}$ accepts any $\mathsf{A} \in \mathsf{domain}\ \mathsf{m}$. Because $\mathsf{domain}\ \mathsf{m}$ is a $\sigma$-algebra, this includes countable unions, countable intersections, and complements of intervals.

For real-valued functions, Lebesgue integration gives another, sometimes more convenient way to characterize almost-everywhere equality: two functions are equal almost everywhere if and only if their indefinite integrals are equal.

**Lemma 10.6** (real function a.e. equality)**.** *If* $\mathsf{m} : \mathsf{Set}\ \mathsf{X} \rightharpoonup [0, +\infty]$ *is a $\sigma$-finite measure and* $\mathsf{g}_1, \mathsf{g}_2 : \mathsf{X} \to \mathbb{R}$ *are measurable, then* $\mathsf{g}_1 = \mathsf{g}_2$ *(*$\mathsf{m}$*-a.e) if and only if* $\mathsf{int}\ \mathsf{g}_1\ \mathsf{m} = \mathsf{int}\ \mathsf{g}_2\ \mathsf{m}$.

The type of $\mathsf{int}$ might suggest its intended use; in particular, $\mathsf{Set}\ \mathsf{X} \rightharpoonup [-\infty, +\infty]$ is similar to $\mathsf{Set}\ \mathsf{X} \rightharpoonup [0, +\infty]$, which we use as the type of measures.[3] We have functionalized indefinite integration to emphasize that, in this chapter and much of measure-theoretic

[3]In fact, $\mathsf{Set}\ \mathsf{X} \rightharpoonup [-\infty, +\infty]$ is the type we would use for *signed* measures if we needed them.

practice, integration's primary purpose is not to compute concrete areas and volumes, but to *transform measures*. This is supported by the following imported theorem.

**Lemma 10.7** (indefinite integration yields measures)**.** *If* $\mathsf{g} : \mathsf{X} \to [0, +\infty)$ *is measurable and* $\mathsf{m} : \mathsf{Set}\ \mathsf{X} \rightharpoonup [0, +\infty]$ *is a measure, then* $\mathsf{int}\ \mathsf{g}\ \mathsf{m}$ *is a measure.*

For example, if $\mathsf{g} : \mathbb{R} \to [0, +\infty)$ is a probability density function and $\mathsf{m}$ is Lebesgue measure on $\mathbb{R}$, then $\mathsf{int}\ \mathsf{g}\ \mathsf{m}$ is a probability measure.

Lemma 10.7 implies there is a function

$$\mathsf{int}^+ : (\mathsf{X} \to [0, +\infty)) \Rightarrow (\mathsf{Set}\ \mathsf{X} \rightharpoonup [0, +\infty]) \Rightarrow (\mathsf{Set}\ \mathsf{X} \rightharpoonup [0, +\infty]) \tag{10.6}$$

which agrees with $\mathsf{int}$ for all arguments $\mathsf{g} : \mathsf{X} \to [0, +\infty)$. We thus begin defining an algebra of measures and operations on them with $\mathsf{int}^+$.

We should expect integration to be positive linear, and it is. In the following, assume that arithmetic is lifted to operate pointwise on mappings.

**Lemma 10.8.** *Let* $\mathsf{g}_1, \mathsf{g}_2 : \mathsf{X} \to [0, +\infty)$ *be measurable and* $\mathsf{m} : \mathsf{Set}\ \mathsf{X} \rightharpoonup [0, +\infty]$ *be a measure. Then* $\mathsf{int}\ (\mathsf{g}_1 + \mathsf{g}_2)\ \mathsf{m}\ =\ \mathsf{int}\ \mathsf{g}_1\ \mathsf{m} + \mathsf{int}\ \mathsf{g}_2\ \mathsf{m}$.

**Lemma 10.9.** *Let* $\mathsf{g} : \mathsf{X} \to [0, +\infty)$ *be measurable,* $\alpha \geq 0$ *and* $\mathsf{m} : \mathsf{Set}\ \mathsf{X} \rightharpoonup [0, +\infty]$ *be a measure. Then* $\mathsf{int}\ (\alpha \cdot \mathsf{g})\ \mathsf{m}\ =\ \alpha \cdot \mathsf{int}\ \mathsf{g}\ \mathsf{m}$.

Lastly, compositions within integrals can be moved into the base measure.

**Lemma 10.10** (image measure integration)**.** *Let* $\mathsf{m} : \mathsf{Set}\ \mathsf{X} \rightharpoonup [0, +\infty]$ *be a measure,* $\mathsf{g}_1 : \mathsf{X} \to \mathsf{Y}$ *and* $\mathsf{g}_2 : \mathsf{Y} \to \mathbb{R}$ *be measurable, and* $\mathsf{m}'$ *be* $\mathsf{g}_1$*'s image measure with respect to* $\mathsf{m}$*. For all* $\mathsf{B} \in \Sigma\ \mathsf{Y}$*,* $\mathsf{int}\ (\mathsf{g}_2 \circ_{\mathsf{map}} \mathsf{g}_1)\ \mathsf{m}\ (\mathsf{preimage}\ \mathsf{g}_1\ \mathsf{B})\ =\ \mathsf{int}\ \mathsf{g}_2\ \mathsf{m}'\ \mathsf{B}$. *[XXX: need more conditions]*

### 10.1.3 Differentiation

In differential calculus, indefinite integration has an inverse: differentiation. In measure theory, indefinite Lebesgue integration also has an inverse, which is also called differentiation.

One significant difference is that, because indefinite Lebesgue integration returns measures, differentiation operates on measures.

In differential calculus, differentiation is defined only for differentiable functions. In measure theory, the analogous property is absolute continuity.

**Definition 10.11** (absolute continuity)**.** *Given measures* $\mathsf{m}_1, \mathsf{m}_2 : \mathsf{Set\ X} \rightharpoonup [0,+\infty]$*, we say* $\mathsf{m}_1$ *is* ***absolutely continuous*** *with respect to* $\mathsf{m}_2$ *if* $\mathsf{m}_1 \ll \mathsf{m}_2$*, where*

$$\begin{aligned} (\ll) &: (\mathsf{Set\ X} \rightharpoonup [0,+\infty]) \Rightarrow (\mathsf{Set\ X} \rightharpoonup [0,+\infty]) \Rightarrow \mathsf{Bool} \\ \mathsf{m}_1 \ll \mathsf{m}_2 &:= \ \forall\, \mathsf{A} \in \mathsf{domain\ m}_2.\, \mathsf{m}_2\ \mathsf{A} = 0 \implies \mathsf{m}_1\ \mathsf{A} = 0 \end{aligned} \tag{10.7}$$

By Definition 10.11, $\mathsf{m}_1 \ll \mathsf{m}_2$ means $\mathsf{m}_1$ has at least as many measure-zero sets as $\mathsf{m}_2$, and is therefore, in a sense, smaller. If $\mathsf{P}$ and $\mathsf{Q}$ are probability measures, $\mathsf{P} \ll \mathsf{Q}$ essentially means $\mathsf{P}$'s support is no larger than $\mathsf{Q}$'s support.

As for integration, for differentiation, we functionalize special notation:

$$\begin{aligned} &\mathsf{diff}^+ : (\mathsf{Set\ X} \rightharpoonup [0,+\infty]) \Rightarrow (\mathsf{Set\ X} \rightharpoonup [0,+\infty]) \Rightarrow (\mathsf{X} \to [0,+\infty)) \\ &\mathsf{diff}^+\ \mathsf{m}_1\ \mathsf{m}_2 \ := \ \frac{d\mathsf{m}_1}{d\mathsf{m}_2} \end{aligned} \tag{10.8}$$

This returns a **Radon-Nikodým derivative**. Such derivatives are named after the following theorem, which gives circumstances under which $\mathsf{diff}^+\ \mathsf{m}_1\ \mathsf{m}_2$ exists, and states that $\mathsf{int}^+$ is the left inverse of $\mathsf{diff}^+$ (with second arguments held constant).

**Lemma 10.12** (Radon-Nikodým)**.** *If* $\mathsf{m}_1, \mathsf{m}_2 : \mathsf{Set\ X} \rightharpoonup [0,+\infty]$ *are* $\sigma$*-finite measures and* $\mathsf{m}_1 \ll \mathsf{m}_2$*, then* $\mathsf{diff}^+\ \mathsf{m}_1\ \mathsf{m}_2$ *exists and* $\mathsf{m}_1 = \mathsf{int}^+\ (\mathsf{diff}^+\ \mathsf{m}_1\ \mathsf{m}_2)\ \mathsf{m}_2$*.*

The function $\mathsf{diff}^+\ \mathsf{m}_1\ \mathsf{m}_2 : \mathsf{X} \to [0,+\infty)$ is often called the *density* of $\mathsf{m}_1$ with respect to $\mathsf{m}_2$, but we call them *derivatives*, reserving *density* for derivatives with respect to Lebesgue measure. By Lemma 10.6, any $\mathsf{g} : \mathsf{X} \rightharpoonup [0,+\infty)$ for which $\mathsf{g} = \mathsf{diff}^+\ \mathsf{m}_1\ \mathsf{m}_2$ ($\mathsf{m}_2$-a.e.) meets the Radon-Nikodým theorem's conclusion $\mathsf{m}_1 = \mathsf{int}^+\ \mathsf{g}\ \mathsf{m}_2$. We therefore say that Radon-Nikodým derivatives are unique up to equality $\mathsf{m}_2$-a.e.

By analogy to differential calculus, we should expect $\mathsf{diff}^+$ to be the left inverse of $\mathsf{int}^+$ (with second arguments held constant). It is, up to equality $\mathsf{m}_2$-a.e.

**Lemma 10.13.** *If* $\mathsf{g}_1 : \mathsf{X} \to [0, +\infty)$ *is measurable and* $\mathsf{m}_2 : \mathsf{Set\ X} \rightharpoonup [0, +\infty]$ *is a* $\sigma$*-finite measure, then* $\mathsf{int}^+\ \mathsf{g}_1\ \mathsf{m}_2 \ll \mathsf{m}_2$ *and* $\mathsf{g}_1 = \mathsf{diff}^+\ (\mathsf{int}^+\ \mathsf{g}_1\ \mathsf{m}_2)\ \mathsf{m}_2$ *(*$\mathsf{m}_2$*-a.e.).*

The preceeding two theorems are analogous to the fundamental theorem of calculus.

We should expect differentiation to be positive linear, and it is.

**Lemma 10.14.** *Let* $\mathsf{m}_1, \mathsf{m}_2, \mathsf{m} : \mathsf{Set\ X} \rightharpoonup [0, +\infty]$ *be* $\sigma$*-finite measures with* $\mathsf{m}_1 \ll \mathsf{m}$ *and* $\mathsf{m}_2 \ll \mathsf{m}$*. Then* $\mathsf{m}_1 + \mathsf{m}_2 \ll \mathsf{m}$ *and* $\mathsf{diff}^+\ (\mathsf{m}_1 + \mathsf{m}_2)\ \mathsf{m}\ =\ \mathsf{diff}^+\ \mathsf{m}_1\ \mathsf{m} + \mathsf{diff}^+\ \mathsf{m}_2\ \mathsf{m}$ *(*$\mathsf{m}$*-a.e.).*

**Lemma 10.15.** *Let* $\mathsf{m}_1, \mathsf{m}_2 : \mathsf{Set\ X} \rightharpoonup [0, +\infty]$ *be* $\sigma$*-finite measures with* $\mathsf{m}_1 \ll \mathsf{m}_2$*. For all* $\alpha \geq 0$ *and* $\beta > 0$*,* $\alpha \cdot \mathsf{m}_1 \ll \beta \cdot \mathsf{m}_2$ *and* $\mathsf{diff}^+\ (\alpha \cdot \mathsf{m}_1)\ (\beta \cdot \mathsf{m}_2)\ =\ \dfrac{\alpha}{\beta} \cdot \mathsf{diff}^+\ \mathsf{m}_1\ \mathsf{m}_2$ *(*$\mathsf{m}$*-a.e.).*

As in differential calculus, there is a chain rule.

**Lemma 10.16** (chain rule)**.** *Let* $\mathsf{m}_1, \mathsf{m}_2, \mathsf{m}_3 : \mathsf{Set\ X} \rightharpoonup [0, +\infty]$ *be* $\sigma$*-finite measures with* $\mathsf{m}_1 \ll \mathsf{m}_2$ *and* $\mathsf{m}_2 \ll \mathsf{m}_3$*. Then* $\mathsf{m}_1 \ll \mathsf{m}_3$ *and* $\mathsf{diff}^+\ \mathsf{m}_1\ \mathsf{m}_2 \cdot \mathsf{diff}^+\ \mathsf{m}_2\ \mathsf{m}_3\ =\ \mathsf{diff}^+\ \mathsf{m}_1\ \mathsf{m}_3$ *(*$\mathsf{m}_3$*-a.e.).*

We need two more differentiation rules, which have no direct analogues in differential calculus. Importing them makes our algebra of measures complete enough to prove importance sampling correct. The first is a rule for reciprocals.

**Lemma 10.17** (reciprocal rule)**.** *Let* $\mathsf{m}_1, \mathsf{m}_2 : \mathsf{Set\ X} \rightharpoonup [0, +\infty]$ *be* $\sigma$*-finite measures with* $\mathsf{m}_2 \ll \mathsf{m}_1$ *and* $\mathsf{m}_1 \ll \mathsf{m}_2$*. Then* $\mathsf{diff}^+\ \mathsf{m}_1\ \mathsf{m}_2\ =\ 1\ /\ \mathsf{diff}^+\ \mathsf{m}_2\ \mathsf{m}_1$ *(*$\mathsf{m}_1$*-a.e. and* $\mathsf{m}_2$*-a.e.).*

The second provides a way to integrate out derivatives, or to use differentiation to change the base measure in Lebesgue integration.

**Lemma 10.18** (change of measure)**.** *Let* $\mathsf{m}_1, \mathsf{m}_2 : \mathsf{Set\ X} \rightharpoonup [0, +\infty]$ *be* $\sigma$*-finite measures with* $\mathsf{m}_1 \ll \mathsf{m}_2$*, and* $\mathsf{g} : \mathsf{X} \to \mathbb{R}$ *be measurable. Then* $\mathsf{int\ g\ m}_1\ =\ \mathsf{int}\ (\mathsf{g} \cdot \mathsf{diff}^+\ \mathsf{m}_1\ \mathsf{m}_2)\ \mathsf{m}_2$*.*

Suppose we have a joint and candidate probability densities $\mathsf{p}, \mathsf{q} : \mathbb{R}^{\mathsf{n}} \rightharpoonup [0, +\infty)$, and we sample according to $\mathsf{q}$ and weight the samples by $\mathsf{p} \ / \ \mathsf{q}$. The weighted samples represent $\mathsf{p}$ if expected values estimated using them are correct; i.e. for all measurable $\mathsf{g} : \mathbb{R}^{\mathsf{n}} \to \mathbb{R}$,

$$\mathsf{int\ g\ P} \ = \ \mathsf{int\ (g \cdot p\ /\ q)\ Q} \tag{10.9}$$

where $\mathsf{P} := \mathsf{int}^+ \ \mathsf{p\ m}$ and $\mathsf{Q} := \mathsf{int}^+ \ \mathsf{q\ m}$, and $\mathsf{m}$ is Lebesgue measure on $\mathbb{R}^{\mathsf{n}}$.

The density route to a proof is simple, and requires that $\mathsf{q}$ be nonzero everywhere. By definition and Lemma 10.13 ($\mathsf{diff}^+$ is a left inverse of $\mathsf{int}^+$), $\mathsf{q} = \mathsf{diff}^+ \ \mathsf{Q\ m}$ ($\mathsf{m}$-a.e.), so by Lemma 10.18 (change of measure) with $\mathsf{m}_1 = \mathsf{Q}$,

$$\begin{aligned} \mathsf{int\ (g \cdot p\ /\ q)\ Q} \ &= \ \mathsf{int\ (g \cdot p\ /\ q \cdot q)\ m} \\ &= \ \mathsf{int\ (g \cdot p)\ m} \end{aligned} \tag{10.10}$$

which again by Lemmas 10.13 and 10.18 is $\mathsf{int\ g\ P}$.

Taking the measure route demonstrates how to prove more general importance sampling theorems. We again require $\mathsf{q}$ to be nonzero everywhere; then

$$\begin{aligned} \mathsf{p\ /\ q} \ = \ \mathsf{p} \cdot 1 \ / \ \mathsf{q} \ &= \ \mathsf{diff}^+ \ \mathsf{P\ m} \cdot 1 \ / \ \mathsf{diff}^+ \ \mathsf{Q\ m} && (\mathsf{m}\text{-a.e.}) && \text{Lemma 10.13} \\ &= \ \mathsf{diff}^+ \ \mathsf{P\ m} \cdot \mathsf{diff}^+ \ \mathsf{m\ Q} && (\mathsf{m}, \mathsf{Q}\text{-a.e.}) && \text{Lemma 10.17} \\ &= \ \mathsf{diff}^+ \ \mathsf{P\ Q} && (\mathsf{Q}\text{-a.e.}) && \text{Lemma 10.16} \end{aligned} \tag{10.11}$$

Because $\mathsf{g} \cdot \mathsf{p} \ / \ \mathsf{q} = \mathsf{g} \cdot \mathsf{diff}^+ \ \mathsf{P\ Q}$ ($\mathsf{Q}$-a.e.),

$$\begin{aligned} \mathsf{int\ (g \cdot p\ /\ q)\ Q} \ &= \ \mathsf{int\ (g \cdot diff^+\ P\ Q)\ Q} && \text{Lemma 10.6} \\ &= \ \mathsf{int\ g\ P} && \text{Lemma 10.18} \end{aligned} \tag{10.12}$$

The more general method is this: instead of densities $\mathsf{p}$ and $\mathsf{q}$, define measures $\mathsf{P}$ and $\mathsf{Q}$, derive $\mathsf{diff}^+ \ \mathsf{P\ Q}$, and apply Lemma 10.18.

The proof of correctness of *partitioned* importance sampling proceeds this way, but requires more machinery to construct a measure-theoretic model of the sampling process.

### 10.1.4 Transition Kernels

In naïve probability theory, conditional density functions model probabilistic processes that depend on the outcome of another. In measure-theoretic probability, this is accomplished using transition kernels, which are not much more than functions that return measures.

**Definition 10.19** (transition kernel)**.** *A function* $\mathsf{k} : \mathsf{X} \to \mathsf{Set}\ \mathsf{Y} \rightharpoonup [0,+\infty]$ *is a* ***transition kernel*** *when both of the following hold.*

- *For all* $\mathsf{a} \in \mathsf{X}$*,* $\mathsf{k}\ \mathsf{a}$ *is a measure.*
- *For all* $\mathsf{B} \in \Sigma\ \mathsf{Y}$*,* $\lambda \mathsf{a} \in \mathsf{X}.\, \mathsf{k}\ \mathsf{a}\ \mathsf{B}$ *is measurable.*

For any measure property, we say $\mathsf{k}$ has that property when it holds for all $\mathsf{k}\ \mathsf{a}$. Therefore, $\mathsf{k}$ is a **probability kernel**, **finite kernel**, or **$\sigma$-finite kernel** when for all $\mathsf{a} \in \mathsf{X}$, $\mathsf{k}\ \mathsf{a}$ is respectively a probability measure, finite measure, or $\sigma$-finite measure.

Product models of dependent processes can be built by starting with a probability measure and iteratively extending it using probability kernels.

**Lemma 10.20** (finite kernel products)**.** *Let* $\mathsf{m} : \mathsf{Set}\ \mathsf{X} \to [0,+\infty]$ *be a finite measure and* $\mathsf{k} : \mathsf{X} \to \mathsf{Set}\ \mathsf{Y} \rightharpoonup [0,+\infty]$ *be a finite kernel. There exists a unique $\sigma$-finite measure* $\mathsf{m} \times \mathsf{k} : \mathsf{Set}\ \langle \mathsf{X}, \mathsf{Y} \rangle \rightharpoonup [0,+\infty]$ *that is determined by its output on rectangles; i.e. defined by extending the following to a product measure: for all* $\mathsf{A} \in \Sigma\ \mathsf{X}$ *and* $\mathsf{B} \in \Sigma\ \mathsf{Y}$*,*

$$(\mathsf{m} \times \mathsf{k})\ (\mathsf{A} \times \mathsf{B}) \;=\; \mathsf{int}^{+}\ (\lambda \mathsf{a} \in \mathsf{X}.\, \mathsf{k}\ \mathsf{a}\ \mathsf{B})\ \mathsf{m}\ \mathsf{A} \tag{10.13}$$

*If* $\mathsf{m}$ *is a probability measure and* $\mathsf{k}$ *a probability kernel,* $\mathsf{m} \times \mathsf{k}$ *is a probability measure.*

For example, if $\mathsf{N} : \mathbb{R} \to \mathsf{Set}\ \mathbb{R} \rightharpoonup [0,1]$ takes a mean $\mu$ and returns a normal probability measure centered on $\mu$ with standard deviation 1, then the interpretation of

$$\begin{aligned} X &\sim \text{Normal}(0,1) \\ Y &\sim \text{Normal}(X,1) \end{aligned} \tag{10.14}$$

as a measure-theoretic joint distribution is $(\mathsf{N}\ 0) \times \mathsf{N}$.

For the proofs in the next section, we need a way to turn integrals with respect to $\mathsf{m} \times \mathsf{k}$ measures into nested integrals.

**Lemma 10.21** (Fubini's for transition kernels)**.** *Let* $\mathsf{m} : \mathsf{Set}\ \mathsf{X} \to [0,+\infty]$ *be a finite measure and* $\mathsf{k} : \mathsf{X} \to \mathsf{Set}\ \mathsf{Y} \rightharpoonup [0,+\infty]$ *be a finite kernel. If* $\mathsf{g} : \mathsf{X} \times \mathsf{Y} \to \overline{\mathbb{R}}$ *is measurable, and nonnegative or* $(\mathsf{m} \times \mathsf{k})$*-integrable, then*

$$\begin{aligned} &\mathsf{int}\ \mathsf{g}\ (\mathsf{m} \times \mathsf{k})\ (\mathsf{X} \times \mathsf{Y}) \\ &\quad = \ \mathsf{int}\ (\lambda\, \mathsf{a} \in \mathsf{X}.\, \mathsf{int}\ (\lambda\, \mathsf{b} \in \mathsf{Y}.\, \mathsf{g}\ \langle \mathsf{a}, \mathsf{b} \rangle)\ (\mathsf{k}\ \mathsf{a})\ \mathsf{Y})\ \mathsf{m}\ \mathsf{X} \end{aligned} \tag{10.15}$$

## 10.2 Sampling Proofs

Recall that $\mathsf{subcond}\ \mathsf{P}\ \mathsf{A}' \ := \ \lambda\, \mathsf{A} \in \mathsf{domain}\ \mathsf{P}.\, \mathsf{P}\ (\mathsf{A}' \cap \mathsf{A})$.

**Theorem 10.22** (partitioned importance sampling correctness)**.** *Let* $\mathsf{X}$*,* $\mathsf{P}$*,* $\mathsf{N}$*,* $\mathsf{s}$*,* $\mathsf{p}$*, and* $\mathsf{Q}$ *as in Definition 8.25 (partitioned importance sampling) such that* $\mathsf{subcond}\ \mathsf{P}\ (\mathsf{s}\ \mathsf{n}) \ll \mathsf{Q}\ \mathsf{n}$ *for all* $\mathsf{n} \in \mathsf{N}$*. Define* $\mathsf{P_N} : \mathsf{Set}\ \mathsf{N} \to [0,1]$ *by integrating* $\mathsf{p}$ *with respect to counting measure.*

*If* $\mathsf{g} : \mathsf{X} \to \mathbb{R}$ *is a* $\mathsf{P}$*-integrable mapping, and*

$$\begin{aligned} &\mathsf{g}' : \mathsf{N} \times \mathsf{X} \to \mathbb{R} \\ &\mathsf{g}'\ \langle \mathsf{n}, \mathsf{a} \rangle \ := \ \mathsf{g}\ \mathsf{a} \cdot \frac{1}{\mathsf{p}\ \mathsf{n}} \cdot \mathsf{diff}^+\ (\mathsf{subcond}\ \mathsf{P}\ (\mathsf{s}\ \mathsf{n}))\ (\mathsf{Q}\ \mathsf{n})\ \mathsf{a} \end{aligned} \tag{10.16}$$

*then* $\mathsf{int}\ \mathsf{g}'\ (\mathsf{P_N} \times \mathsf{Q})\ (\mathsf{N} \times \mathsf{X}) \ = \ \mathsf{int}\ \mathsf{g}\ \mathsf{P}\ \mathsf{X}$*.*

*Proof.* Let $\mathsf{w_1}\ \mathsf{n} := \frac{1}{\mathsf{p}\ \mathsf{n}}$ and $\mathsf{w_2}\ \mathsf{n} := \mathsf{diff}^+\ (\mathsf{subcond}\ \mathsf{P}\ (\mathsf{s}\ \mathsf{n}))\ (\mathsf{Q}\ \mathsf{n})$. Starting from the left side,

$$\begin{aligned} &\mathsf{int}\ \mathsf{g}'\ (\mathsf{P_N} \times \mathsf{Q})\ (\mathsf{N} \times \mathsf{X}) && \text{(10.17)} \\ &\quad = \ \mathsf{int}\ (\lambda\, \langle \mathsf{n}, \mathsf{a} \rangle \in \mathsf{N} \times \mathsf{X}.\, \mathsf{g}\ \mathsf{a} \cdot \mathsf{w_1}\ \mathsf{n} \cdot \mathsf{w_2}\ \mathsf{n}\ \mathsf{a})\ (\mathsf{P_N} \times \mathsf{Q})\ (\mathsf{N} \times \mathsf{X}) && \text{Def of } \mathsf{g}' \\ &\quad = \ \mathsf{int}\ (\lambda\, \mathsf{n} \in \mathsf{N}.\, \mathsf{int}\ (\lambda\, \mathsf{a} \in \mathsf{X}.\, \mathsf{g}\ \mathsf{a} \cdot \mathsf{w_1}\ \mathsf{n} \cdot \mathsf{w_2}\ \mathsf{n}\ \mathsf{a})\ (\mathsf{Q}\ \mathsf{n})\ \mathsf{X})\ \mathsf{P_N}\ \mathsf{N} && \text{Lemma 10.21} \\ &\quad = \ \mathsf{int}\ (\lambda\, \mathsf{n} \in \mathsf{N}.\, \mathsf{int}\ (\mathsf{g} \cdot \mathsf{w_1}\ \mathsf{n} \cdot \mathsf{w_2}\ \mathsf{n})\ (\mathsf{Q}\ \mathsf{n})\ \mathsf{X})\ \mathsf{P_N}\ \mathsf{N} && \text{Lift } (\cdot) \\ &\quad = \ \mathsf{int}\ (\lambda\, \mathsf{n} \in \mathsf{N}.\, \mathsf{w_1}\ \mathsf{n} \cdot \mathsf{int}\ (\mathsf{g} \cdot \mathsf{w_2}\ \mathsf{n})\ (\mathsf{Q}\ \mathsf{n})\ \mathsf{X})\ \mathsf{P_N}\ \mathsf{N} && \text{Lemma 10.9} \\ &\quad = \ \mathsf{int}\ (\lambda\, \mathsf{n} \in \mathsf{N}.\, \mathsf{w_1}\ \mathsf{n} \cdot \mathsf{int}\ \mathsf{g}\ (\mathsf{subcond}\ \mathsf{P}\ (\mathsf{s}\ \mathsf{n}))\ \mathsf{X})\ \mathsf{P_N}\ \mathsf{N} && \text{Def } \mathsf{w_2}\text{, Lemma 10.18} \end{aligned}$$

Because $\mathsf{P_N}$ is defined with respect to counting measure, turn integration into summation:

$$
\begin{aligned}
&= \sum_{\mathsf{n} \in \mathsf{N}} \mathsf{p\ n} \cdot \frac{1}{\mathsf{p\ n}} \cdot \mathsf{int\ g\ (subcond\ P\ (s\ n))\ X} && \text{Def of } \mathsf{w}_1 \\
&= \sum_{\mathsf{n} \in \mathsf{N}} \mathsf{int\ g\ (subcond\ P\ (s\ n))\ X} && \mathsf{p\ n} > 0 \\
&= \sum_{\mathsf{n} \in \mathsf{N}} \mathsf{int\ g\ P\ (s\ n)} && \text{XXX: justify} \\
&= \mathsf{int\ g\ P}\ (\textstyle\bigcup_{\mathsf{n} \in \mathsf{N}}\ \mathsf{(s\ n)}) && \sigma\text{-additivity} \\
&= \mathsf{int\ g\ P\ X} && \text{Def of } \mathsf{s} \qquad \square
\end{aligned}
$$

When $\mathsf{m}_1$ and $\mathsf{m}_2$ are measures on infinite spaces, it is not clear that $\mathsf{diff}^+\ \mathsf{m}_1\ \mathsf{m}_2$ exists or how to compute it. It seems it should exist when $\mathsf{m}_1$ and $\mathsf{m}_2$ differ only on a finite projection of their domains, and that it should be easy to compute when the distributions of those finite projections can be defined by densities.

We start with a theorem for $\mathsf{diff}^+\ (\mathsf{m}_1 \times \mathsf{k})\ (\mathsf{m}_2 \times \mathsf{k})$ that says we may ignore $\mathsf{k}$.

**Theorem 10.23.** *Let* $\mathsf{m}_1, \mathsf{m}_2 : \mathsf{Set\ X} \rightharpoonup [0, +\infty]$ *be finite measures such that* $\mathsf{m}_1 \ll \mathsf{m}_2$*, and* $\mathsf{k} : \mathsf{X} \to \mathsf{Set\ Y} \rightharpoonup [0, +\infty]$ *be a finite kernel. Then* $\mathsf{m}_1 \times \mathsf{k} \ll \mathsf{m}_2 \times \mathsf{k}$ *and* $\mathsf{diff}^+\ (\mathsf{m}_1 \times \mathsf{k})\ (\mathsf{m}_2 \times \mathsf{k}) = (\lambda\, \langle \mathsf{a}, \mathsf{b} \rangle \in \mathsf{X} \times \mathsf{Y}.\, \mathsf{diff}^+\ \mathsf{m}_1\ \mathsf{m}_2\ \mathsf{a})$ *(*$\mathsf{m}_2 \times \mathsf{k}$*-a.e.).*

*Proof.* XXX: prove absolute continuity

Let $\mathsf{A} \in \mathit{\Sigma}\ \mathsf{X}$ and $\mathsf{B} \in \mathit{\Sigma}\ \mathsf{Y}$. Integrating the left-hand side,

$$
\begin{aligned}
&\mathsf{int\ (diff^+\ (m_1 \times k)\ (m_2 \times k))\ (m_2 \times k)\ (A \times B)} && (10.18) \\
&\quad = \mathsf{(m_1 \times k)\ (A \times B)} && \text{Lemma 10.12}
\end{aligned}
$$

Integrating the right-hand side,

$$\begin{aligned}
&\mathsf{int}\ (\lambda\,\langle \mathsf{a},\mathsf{b}\rangle \in \mathsf{X}\times\mathsf{Y}.\,\mathsf{diff}^{+}\ \mathsf{m}_1\ \mathsf{m}_2\ \mathsf{a})\ (\mathsf{m}_2\times\mathsf{k})\ (\mathsf{A}\times\mathsf{B}) && (10.19)\\
&\quad=\ \mathsf{int}\ (\lambda\,\mathsf{a}\in\mathsf{X}.\,\mathsf{int}\ (\lambda\,\mathsf{b}\in\mathsf{Y}.\,\mathsf{diff}^{+}\ \mathsf{m}_1\ \mathsf{m}_2\ \mathsf{a})\ (\mathsf{k}\ \mathsf{a})\ \mathsf{B})\ \mathsf{m}_2\ \mathsf{A} && \text{Lemma 10.21}\\
&\quad=\ \mathsf{int}\ (\mathsf{diff}^{+}\ \mathsf{m}_1\ \mathsf{m}_2\cdot\lambda\,\mathsf{a}\in\mathsf{X}.\,\mathsf{int}\ (\lambda\,\mathsf{b}\in\mathsf{Y}.\,1)\ (\mathsf{k}\ \mathsf{a})\ \mathsf{B})\ \mathsf{m}_2\ \mathsf{A} && \text{Lemma 10.9, Lift }(\cdot)\\
&\quad=\ \mathsf{int}\ (\lambda\,\mathsf{a}\in\mathsf{X}.\,\mathsf{k}\ \mathsf{a}\ \mathsf{B})\ \mathsf{m}_1\ \mathsf{A} && \text{Lemma 10.18}\\
&\quad=\ (\mathsf{m}_1\times\mathsf{k})\ (\mathsf{A}\times\mathsf{B}) && \text{Lemma 10.20}
\end{aligned}$$

Therefore, because $\mathsf{m}_1\times\mathsf{k}$ is uniquely defined by its output on all such $\mathsf{A}\times\mathsf{B}$,

$$\begin{aligned}
&\mathsf{int}\ (\mathsf{diff}^{+}\ (\mathsf{m}_1\times\mathsf{k})\ (\mathsf{m}_2\times\mathsf{k}))\ (\mathsf{m}_2\times\mathsf{k})\\
&\quad=\ \mathsf{int}\ (\lambda\,\langle \mathsf{a},\mathsf{b}\rangle\in\mathsf{X}\times\mathsf{Y}.\,\mathsf{diff}^{+}\ \mathsf{m}_1\ \mathsf{m}_2\ \mathsf{a})\ (\mathsf{m}_2\times\mathsf{k})
\end{aligned}\qquad (10.20)$$

Apply Lemma 10.6 (real function a.e. equality). □

It is not hard to extend the preceeding theorem to arbitrary sublists of finite lists, or to arbitrary finite substructures of any algebraic data type, by induction. But we need a version of it for arbitrary finite substructures of infinite binary trees, which we have defined non-inductively as mappings $\Omega := \mathsf{J}\to[0,1]$ from tree indexes to reals.

One solution is to define an injective transformation $\mathsf{g}$ from any $\omega\in\Omega$ to a pair $\langle\omega_{\mathsf{fin}},\omega_{\mathsf{inf}}\rangle$, where $\omega_{\mathsf{fin}}$ is a finite substructure of $\omega$ and $\omega_{\mathsf{inf}}$ is the rest of it, and apply Theorem 10.23. The proof is easier to do first in generality, without specifying the structure of $\omega$, requiring the substructure to be finite, or requiring the pairs to contain projections.

**Theorem 10.24.** *Let* $\mu_1,\mu_2 : \mathsf{Set}\ \mathsf{Z}\rightharpoonup[0,+\infty]$ *be* $\sigma$*-finite measures. If there exist finite measures* $\mathsf{m}_1,\mathsf{m}_2 : \mathsf{Set}\ \mathsf{X}\rightharpoonup[0,+\infty]$ *such that* $\mathsf{m}_1\ll\mathsf{m}_2$*, a finite kernel* $\mathsf{k}:\mathsf{X}\to\mathsf{Set}\ \mathsf{Y}\rightharpoonup[0,+\infty]$*, and an injective, measurable function* $\mathsf{g}:\mathsf{Z}\to\mathsf{X}\times\mathsf{Y}$ *such that for all* $\mathsf{D}\in\Sigma\ \langle\mathsf{X},\mathsf{Y}\rangle$*,*

$$\begin{aligned}
(\mathsf{m}_1\times\mathsf{k})\ \mathsf{D}\ &=\ \mu_1\ (\mathsf{preimage}\ \mathsf{g}\ \mathsf{D})\\
(\mathsf{m}_2\times\mathsf{k})\ \mathsf{D}\ &=\ \mu_2\ (\mathsf{preimage}\ \mathsf{g}\ \mathsf{D})
\end{aligned}\qquad (10.21)$$

*then* $\mu_1 \ll \mu_2$ *and* $\mathsf{diff}^+\ \mu_1\ \mu_2 \ = \ \lambda \mathsf{z} \in \mathsf{Z}.\, \mathsf{diff}^+\ \mathsf{m}_1\ \mathsf{m}_2\ (\mathsf{fst}\ (\mathsf{g}\ \mathsf{z}))$ *($\mu_2$-a.e.).*

*Proof.* Suppose $\mathsf{m}_1$, $\mathsf{m}_2$, $\mathsf{k}$ and $\mathsf{g}$ as stated. XXX: prove absolute continuity

Let $\mathsf{C} \in \Sigma\ \mathsf{Z}$ and $\mathsf{D} := \mathsf{image}\ \mathsf{g}\ \mathsf{C}$; then

$$\begin{aligned}
&\mathsf{int}\ (\mathsf{diff}^+\ \mu_1\ \mu_2)\ \mu_2\ \mathsf{C} && \qquad (10.22)\\
&= \mu_1\ \mathsf{C} && \text{Lemma 10.12}\\
&= (\mathsf{m}_1 \times \mathsf{k})\ \mathsf{D} && \text{Def } (\mathsf{m}_1 \times \mathsf{k})\text{, injectivity}\\
&= \mathsf{int}\ (\mathsf{diff}^+\ (\mathsf{m}_1 \times \mathsf{k})\ (\mathsf{m}_2 \times \mathsf{k}))\ (\mathsf{m}_2 \times \mathsf{k})\ \mathsf{D} && \text{Lemma 10.12}\\
&= \mathsf{int}\ (\lambda \langle \mathsf{a}, \mathsf{b}\rangle \in \mathsf{X} \times \mathsf{Y}.\, \mathsf{diff}^+\ \mathsf{m}_1\ \mathsf{m}_2\ \mathsf{a})\ (\mathsf{m}_2 \times \mathsf{k})\ \mathsf{D} && \text{Theorem 10.23}\\
&= \mathsf{int}\ ((\lambda \langle \mathsf{a}, \mathsf{b}\rangle \in \mathsf{X} \times \mathsf{Y}.\, \mathsf{diff}^+\ \mathsf{m}_1\ \mathsf{m}_2\ \mathsf{a}) \circ_{\mathsf{map}} \mathsf{g})\ \mu_2\ \mathsf{C} && \text{Lemma 10.10}\\
&= \mathsf{int}\ (\lambda \mathsf{z} \in \mathsf{Z}.\, \mathsf{diff}^+\ \mathsf{m}_1\ \mathsf{m}_2\ (\mathsf{fst}\ (\mathsf{g}\ \mathsf{z})))\ \mu_2\ \mathsf{C} && \text{Def of } \circ_{\mathsf{map}}
\end{aligned}$$

Apply Lemma 10.6 (real function a.e. equality). □

Thus, two measures $\mu_1$ and $\mu_2$ on infinite structures that can be decomposed into products $\mathsf{m}_1 \times \mathsf{k}$ and $\mathsf{m}_2 \times \mathsf{k}$ such that $\mathsf{diff}^+\ \mathsf{m}_1\ \mathsf{m}_2$ exists—using any injective transformation—have a Radon-Nikodým derivative that can be uniquely defined in terms of $\mathsf{diff}^+\ \mathsf{m}_1\ \mathsf{m}_2$.

Application to infinite binary trees mostly requires defining the transformation.

**Theorem 10.25.** *Let* $\mathsf{J}' \subseteq \mathsf{J}$ *be finite, and define* $\mathsf{X} := \mathsf{J}' \to [0,1]$ *and* $\mathsf{Y} := (\mathsf{J}\backslash\mathsf{J}') \to [0,1]$. *Let* $\mathsf{P}', \mathsf{Q}' : \mathsf{Set}\ \mathsf{X} \rightharpoonup [0,1]$ *be finite measures such that* $\mathsf{P}' \ll \mathsf{Q}'$*, and let* $\mathsf{k} : \mathsf{X} \to \mathsf{Set}\ \mathsf{Y} \rightharpoonup [0,1]$ *be a finite kernel. Let* $\mathsf{g} : \Omega \to \mathsf{X} \times \mathsf{Y}$ *be defined by* $\mathsf{g}\ \omega \ = \ \langle \mathsf{restrict}\ \omega\ \mathsf{J}', \mathsf{restrict}\ \omega\ (\mathsf{J}\backslash\mathsf{J}')\rangle$. *If* $\mathsf{P}, \mathsf{Q} : \mathsf{Set}\ \Omega \rightharpoonup [0,1]$ *are defined so that for all* $\Omega' \in \Sigma\ \Omega$,

$$\begin{aligned}
\mathsf{P}\ \Omega' &= (\mathsf{P}' \times \mathsf{k})\ (\mathsf{image}\ \mathsf{g}\ \Omega')\\
\mathsf{Q}\ \Omega' &= (\mathsf{Q}' \times \mathsf{k})\ (\mathsf{image}\ \mathsf{g}\ \Omega')
\end{aligned} \qquad (10.23)$$

*then* $\mathsf{P} \ll \mathsf{Q}$ *and* $\mathsf{diff}^+\ \mathsf{P}\ \mathsf{Q} \ = \ \lambda \omega \in \Omega.\, \mathsf{diff}^+\ \mathsf{P}'\ \mathsf{Q}'\ (\mathsf{restrict}\ \omega\ \mathsf{J}')$ *($\mathsf{Q}$-a.e.).*

*Proof.* The inverse of $\mathsf{g}$ is $\mathsf{g}^{-1} : \mathsf{X} \times \mathsf{Y} \to \Omega$, defined by

$$\mathsf{g}^{-1}\ \langle \omega_{\mathsf{fin}}, \omega_{\mathsf{inf}} \rangle \;=\; \lambda \mathsf{j} \in \mathsf{J}.\,\mathsf{if}\ (\mathsf{j} \in \mathsf{J}')\ (\omega_{\mathsf{fin}}\ \mathsf{j})\ (\omega_{\mathsf{inf}}\ \mathsf{j}) \tag{10.24}$$

Thus $(\mathsf{P}' \times \mathsf{k})\ \mathsf{D} = \mathsf{P}\ (\mathsf{preimage\ g\ D})$ for all $\mathsf{D} \in \Sigma\ \langle \mathsf{X}, \mathsf{Y} \rangle$; similarly for $(\mathsf{Q}' \times \mathsf{k})\ \mathsf{D}$. Apply Theorem 10.24. □

In particular, if additionally $\mathsf{P}'$ and $\mathsf{Q}'$ can be defined by densities $\mathsf{p} : (\mathsf{J}' \to [0,1]) \to [0,+\infty)$ and $\mathsf{q} : (\mathsf{J}' \to [0,1]) \to [0,+\infty)$, then

$$\mathsf{diff}^{+}\ \mathsf{P}\ \mathsf{Q} \;=\; \lambda \omega \in \Omega.\,\frac{\mathsf{p}\ (\mathsf{restrict}\ \omega\ \mathsf{J}')}{\mathsf{q}\ (\mathsf{restrict}\ \omega\ \mathsf{J}')} \qquad (\mathsf{Q}\text{-a.e.}) \tag{10.25}$$

# Chapter 11

# Related Work

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

## 11.1 Implementations

Almost all of the languages defined by their implementations support conditional queries and compute converging approximations. The reports on these languages generally describe interpreters, compilers, and algorithms for sampling with probabilistic conditions. When they work correctly, they are useful.

Koller and Pfeffer [28] efficiently compute exact, discrete distributions for the outputs of programs in a Scheme-like language. BUGS [33] focuses on efficient approximate computation for probabilistic theories with a finitely many statements, and uses approximation methods that Bayesians typically use. BLOG [36] exists specifically to allow stating distributions over countably infinite vectors. BLAISE [10] allows stating both distributions and approximation methods for random variables. Church [17] is a Scheme-like probabilistic language with approximate inference, and focuses on expressiveness.

Kiselyov [26] embeds a probabilistic language in O'Caml for efficient computation. It uses continuations to enumerate or sample random variable values, and has a `fail` construct for the *complement* of conditioning. The sampler looks ahead for `fail` and can handle it efficiently. This may be justified by commutativity (Thm. 5.10), depending on interaction with other language features.

Recently, Wingate et al [56, 57] have defined the semantics of *nonstandard interpretations* that enable efficient inference, but do not define the languages. [XXX: recheck, relate to computation indexing]

## 11.2 Semantics

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 account of probability, and with it defines the probability monad, whose discrete version is a distribution-valued variation of the set or list monad.

Ramsey and Pfeffer [45] define the probability monad measure-theoretically and implement a language for finite probability. We do not build on this work because the probability monad does not build a global probability space, making it difficult to reason about conditioning.

Using inverse transform sampling, Park [41] extends a $\lambda$-calculus with probabilistic choice according to any of a general class of probability measures. This is the same technique we use in Chapter 8 to turn uniform probabilistic choice into choice according to other distributions, though preimage computation makes doing so more difficult.

Some recent work in probabilistic language semantics tackles conditioning. Pfeffer's IBAL [44] 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.

## 11.3 Techniques

The forward phase in computing preimages takes a subdomain and returns an overapproximation of the function's range for that subdomain. This clearly generalizes interval arithmetic [24] to all first-order algebraic types. We further generalize interval arithmetic by doing it backwards as well as forwards.

Our general approach—creating an exact semantics and deriving an implementable, approximating semantics—is similar to abstract interpretation [13]. Most functional programming researchers would hesitate to call Chapter 5's approximating semantics abstract, however. While the approximations it computes converge, they are not conservative.

The work in Chapter 7 is much more in line with typical abstract interpretation: $[\![\cdot]\!]^{\Downarrow}_{\text{pre}^*}$ is the concrete semantics, $[\![\cdot]\!]^{\Downarrow'}_{\text{pre}^*}$ is an abstract semantics, the concrete values are arbitrary sets, and the abstract values are rectangles. It has many properties common to abstract semantics: it is sound, it labels expressions, the abstract domain is a lattice, and the exact semantics 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 conditional branches of completely unrolled, infinite programs, and are not fixed. There is no Kleene iteration to find a fixed point. In our abstract semantics, infinite computations are done in a library of $\lambda_{\text{ZFC}}$-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.

## 11.4 Somewhat 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 forwards and backwards, 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 $y \in Y$ to $y' \in Y$ induces a corresponding change from $x \in X$ to some $x' \in X$. 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.

# Chapter 12

# Conclusions and Future Work

## 12.1 Conclusions

We started by defining $\lambda_{\mathrm{ZFC}}$, a call-by-value $\lambda$-calculus with infinite sets and set operations, so that we could interpret Bayesian notation categorically.

We then investigated a general approach to trustworthy Bayesian languages: defining an exact semantics that interprets notation as measure-theoretic models, and then deriving a directly implementable approximating semantics. We restricted our investigation to countable distributions and theories with finitely many statements, as it is the first point in the design space where approximation is necessary, and requires no deep measure theory.

In a slight change of tactics, we fixed a canonical probability space of uniformly random, infinite binary trees, and interpreted programs as measure-theoretic random variables, and then as computations that compute exact preimages. The approximating semantics interprets programs as computations that compute conservative approximations of preimages, using rectangles instead of sets. We demonstrated that the language is useful by implementing the approximating semantics and encoding typical Bayesian theories. We also encoded theories without density models, which can only be interpreted using measure theory.

In short, we developed trustworthy, useful languages for Bayesian modeling and inference by founding them solidly on functional programming theory and measure-theoretic probability.

## 12.2 Future Work

Future work falls into two categories: expressiveness and optimization.

### 12.2.1 Expressiveness

Adding a new feature and its semantics to a Turing-equivalent language makes the language more **expressive** if the only way to encode the new feature in the original language with the original semantics is by a global transformation [XXX: cite Felleisen].

An example is adding lambdas. For a first-order language like we have defined, adding lambdas requires closure conversion and defunctionalization: turning every lambda value into a data structure containing bound variable values and a function pointer, and changing every application site to apply a global dispatching function that decodes such closures [XXX: cite]. It may be simpler, more efficient, or more elegant to add lambda terms to the language itself, despite the fact that ensuring higher-order application is measurable is difficult.

Other examples of expressive new features are mutation, exceptions and parameters [XXX: cite] (or more generally continuations [XXX: cite] and continuation marks [XXX: cite]). Once lambdas are available, these can be encoded by globally transforming programs into monadic computations [XXX: cite; include Kimball]. We want to know whether such global transformations are the simplest, most efficient, or most elegant ways to extend our probabilistic language's expressiveness.

Non-examples are loops and other constructs that are merely syntactic sugar, which can be implemented using local transformations. We will be able to provide all such features by making Racket syntax transformers available to programs.

It is not clear that lambdas are the best recursive abstraction for encoding Bayesian theories. Lambdas are good for creating abstractions because their bodies and the computations they carry out are generally unobservable. However, these facts may cause them to not meet Bayesian needs very well: Bayesians define theories in order to study them, and often to discover the behavior of a process cannot be directly observed. Abstracting such processes using lambdas would require returning every intermediate value. We do not know what the right recursive abstraction is for such cases, let alone whether it can be implemented using a local transformation. This dimension of probabilistic language design clearly needs study.

In our experience, probabilistic programs are more difficult to debug than other kinds of programs. Therefore, one way to increase expressiveness while reducing errors is by adding a type system. A type system similar to Typed Racket [XXX: cite] would be a good choice. Because Typed Racket was originally meant for converting untyped programs into typed programs by adding a few annotations, it has true union types, and **occurrence typing**, which allows identifiers to have different types in each branch of a conditional based on the result of its test expression. Occurrence typing is similar to how the forward phase in preimage computation applies the interpretation of each *true* branch to the preimage of $\{\mathsf{true}\}$ under the test, and the *false* branch to the preimage of $\{\mathsf{false}\}$. Union types are similar to our representation of disjoint unions of sets of tagged data structures.

In fact, there are many similarities between preimage computation and type checking and inference, which we plan to investigate.

### 12.2.2 Optimization

A type system would not only help reduce programmer error, but would provide information about program terms that an optimizer could use. For example, because our implementation's sets are monomorphic, every set operation must dispatch to a more specific operation based on the runtime data types of its arguments. If a type system determined that a certain computation consumed and produced only pairs, such dispatch would be unnecessary.

Importance sampling is known to suffer from high variance in its estimates when used on high-dimensional models [XXX: cite]. The higher the variance, the more samples are necessary to get good estimates. A major piece of future optimization work is to use different sampling algorithms, such as Markov Chain Monte Carlo (MCMC) [XXX: cite]. We expect the split-and-refine part of sampling to be particularly amenable, because it divides the program domain into an at-most-countable partition. Markov-chain sampling of parts from this partition may easily sidestep a common problem with MCMC methods: that they often mix poorly when sampling within narrow, non-convex shapes. In any case, to use MCMC, we would need to solve this problem, because the preimages of conditions in typical Bayesian queries are narrow and non-convex.

Our semantics trades efficiency for simplicity by threading a constant, tree-shaped random source. This makes each `random` computation linear-time in the depth it appears in the completely inlined program, which can turn functions that should be linear-time into quadratic-time. Passing subtrees instead would make `random` constant-time, restoring these functions' apparent time complexity. Passing subtrees would also allow combinators to detect lack of change in their received subtrees and return cached values.

Besides rectangles, we intend to try other simple, but more expressive set representations, such as parallelotopes [4]. For representing sets of strings, regular expressions look promising [XXX: cite paper from CS 686], as does the idea of using *abstract* regular expressions to represent sets of lengths and other integral values.

Model equivalence in distribution extends readily to uncountable spaces. It defines a standard for measure-theoretic optimizations, which can only be done in the exact semantics. Examples of optimizations we could prove correct and implement using this equivalence are variable collapse, a probabilistic analogue of constant folding that can increase efficiency by an order of magnitude, and a probabilistic analogue of constraint propagation to speed up conditional queries.

More broadly, we hope to advance Bayesian practice by providing rich modeling languages with an efficient, correct implementations, which allow general recursion and every computable, probabilistic condition.

## 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 $\kappa$-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] Ryan Culpepper. *Refining Syntactic Sugar: Tools for Supporting Macro Development*. PhD thesis, Northeastern University, 2010. To Appear.

[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] Robert Mateescu and Rina Dechter. Mixed deterministic and probabilistic networks. *Annals of Mathematics and Artificial Intelligence*, 2008.

[35] Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(1), 2008.

[36] 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.

[37] James R. Munkres. *Topology.* Prentice Hall, second edition, 2000.

[38] Paul J. Nahin. *Duelling Idiots and Other Probability Puzzlers.* Princeton University Press, 2000.

[39] Russell O'Connor. Certified exact transcendental real number computation in Coq. In *TPHOLs'08*, pages 246–261, 2008.

[40] Toby Ord. The many forms of hypercomputation. *Applied Mathematics and Computation*, 178:143–153, 2006.

[41] Sungwoo Park, Frank Pfenning, and Sebastian Thrun. A probabilistic language based upon sampling functions. *Transactions on Programming Languages and Systems*, 31(1), 2008.

[42] L. C. Paulson. Set theory for verification: II. Induction and recursion. *Journal of Automated Reasoning*, 15:167–215, 1995.

[43] Lawrence C. Paulson. Set theory for verification: I. From foundations to functions. *Journal of Automated Reasoning*, 11:353–389, 1993.

[44] Avi Pfeffer. The design and implementation of IBAL: A general-purpose probabilistic language. In *Statistical Relational Learning*. MIT Press, 2007.

[45] Norman Ramsey and Avi Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *Principles of Programming Languages*, 2002.

[46] S M Samuels. The Radon-Nikodym theorem as a theorem in probability. *The American Mathematical Monthly*, 85(3):155–165, March 1978.

[47] Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of typed Scheme. In *Principles of Programming Languages*, pages 395–406, 2008.

[48] Neil Toronto and Jay McCarthy. From Bayesian notation to pure Racket, via measure-theoretic probability in $\lambda_{\mathrm{ZFC}}$. In *Impl. and Appl. of Functional Languages*, 2010.

[49] Neil Toronto and Jay McCarthy. Computing in Cantor's paradise with $\lambda_{\mathrm{ZFC}}$. In *Functional and Logic Programming Symposium*, pages 290–306, 2012.

[50] Neil Toronto, Bryan S. Morse, Kevin Seppi, and Dan Ventura. Super-resolution via recapture and Bayesian effect modeling. In *Computer Vision and Pattern Recognition*, 2009.

[51] 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.

[52] Athanassios Tzouvaras. Cardinality without enumeration. *Studia Logica: An International Journal for Symbolic Logic*, 80(1):121–141, June 2005.

[53] Gabriel Uzquiano. Models of second-order Zermelo set theory. *The Bulletin of Symbolic Logic*, 5(3):289–302, 1999.

[54] Philip Wadler. Monads for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*. 2001.

[55] Benjamin Werner. Sets in types, types in sets. In *TACS'97*, pages 530–546, 1997.

[56] 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.

[57] David Wingate, Andreas Stuhlmüller, and Noah D. Goodman. Lightweight implementations of probabilistic programming languages via transformational compilation. In *Artificial Intelligence and Statistics*, 2011.